casperfpga Documentation

Release 0.1

Collaboration for Astronomy Signal Processing and Electronics R

Jun 24, 2022

casperfpga Documentation

Welcome to the documentation for `casperfpga`, the Python-based communications package for CASPER Hardware!

CHAPTER 1

New Users

`casperfpga` is a Python library used to interact and interface with CASPER Hardware. Functionality includes being able to reconfigure firmware, as well as read and write registers across the various communication interfaces. The info here describes the following:

1. *Installing casperfpga*
2. *Migrating from corr*
3. *casperfpga Sourcecode*

Should you be an existing *corr* user, wondering where some of your functionality has gone when interfacing to your ROACH/2, 'Migtrating from corr' above offers a detailed explanation on how to migrate to `casperfpga`.

Should you want to dive straight into its usage with the toolflow proper, please see here.

CHAPTER 2

Existing Users

From commit a5e7dcc and earlier the method of instantiating e.g. a SKARAB object was as follows:

```
In [1]: import casperfpga
In [2]: skarab = casperfpga.SkarabFpga('skarab010103')
In [3]: roach = casperfpga.katcp_fpga.KatcpFpga('roach020203')
```

As of commit 4adffc0 the method of instantiating a ROACH or SKARAB was altered to be done intelligently. `casperfpga` automatically works out whether the hostname given in its instantiation is a ROACH, SKARAB or SNAP board.

```
In [1]: import casperfpga
In [2]: skarab = casperfpga.CasperFpga('skarab010103')
DEBUG:root:skarab010103 seems to be a SKARAB
INFO:casperfpga.transport_skarab:skarab010103: port(30584) created & connected.
DEBUG:root:casperfpga.casperfpga:skarab010103: now a CasperFpga
In [3]: roach = casperfpga.CasperFpga('roach020203')
DEBUG:root:roach020203 seems to be a ROACH
INFO:casperfpga.transport_katcp:roach020203: port(7147) created and connected.
DEBUG:root:casperfpga.casperfpga:roach020203: now a CasperFpga
```

CHAPTER 3

Contributing

If you would like to contribute towards this library, fork the casperfpga repo, add your changes to the fork and issue a pull request to the parent repo.

3.1 How to install casperfpga

This section explains how to install `casperfpga`, a python library used to interact with CASPER hardware.

Once you have cloned the casperfpga repository, ensure that you are on the correct branch (usually **master** unless you are a contributor) and always pull regularly to make sure you have the latest version of casperfpga. almon are considered "anadromous" which means they live in both fresh and salt water. They are born in freshwater where they spend a few months to a few years (depending on the species) before moving out to the ocean

3.1.1 Installing casperfpga

`casperfpga` is now available on the Python Package Index (PyPI) and can be installed via `pip`. However, should you need to interface with a SNAP board, your installation workflow involves the extra step of installing against `casperfpga's requirements.txt`.

```
$ git clone https://github.com/casper-astro/casperfpga
$ cd casperfpga/
$ git checkout master
$ sudo apt-get install python-pip
$ sudo pip install -r requirements.txt
$ sudo pip install casperfpga
```

The distribution on the Python Package Index is, of course, a built-distribution; this contains an already-compiled version of the SKARAB programming utility `progska`, written in `C`. Operating Systems tested using `pip install casperfpga` include:

1. Ubuntu 14.04 LTS
2. Ubuntu 16.04 LTS

3. Ubuntu 18.04 LTS
4. Debian 8.x

Unfortunately the success of your installation using `pip` depends on the host OS of the installation, and you might need to rebuild the utility using the C-compiler native to your OS. In short follow the more traditional method of installing custom Python packages.

```
# remove current casperfpga install files
$ cd /usr/local/lib/python2.7/dist-packages
$ sudo rm -rf casper*

# clone the repository to your working directory
$ cd /path/to/working/directory
$ git clone https://github.com/casper-astro/casperfpga.git
$ cd casperfpga
$ git checkout master
$ sudo pip install -r requirements.txt
$ sudo python setup.py install
```

Testing that the installation worked

To check that casperfpga has been installed correctly open an ipython session and import casperfpga. To avoid errors, move out of your cloned casperfpga repository directory before doing this test. `casperfpga.__version__` will output the build and githash version of your casperfpga library.

```
$ cd ..
$ ipython
```

```
In [1]: import casperfpga
In [2]: casperfpga.__version__
```

If you receive any errors during this step please feel free to contact anyone on the CASPER Mailing List, or check the Mailing List Archive to see if your issue has been resolved already.

3.1.2 Using casperfpga

The introductory tutorials for current CASPER hardware serve as a guide to the entire process of:

- Creating an FPGA design in Simulink using the CASPER and Xilinx Blocksets
- Building the design using the toolflow, and lastly
- Reconfiguring your CASPER Hardware with the generated .fpg file using `casperfpga`

`casperfpga` is written in python and mainly used to communicate with CASPER Hardware and reconfigure it's firmware. Hence the medium of communication is usually done through an ipython session, as shown below:

```
import casperfpga
fpga = casperfpga.CasperFpga('skarab_host or roach_name')
fpga.upload_to_ram_and_program('your_file.fpg')
```

Once again, please feel free to contact the CASPER Mailing List should you encounter any problems or have any questions.

3.2 Migrating from corr

So, you've been using `corr` to communicate and reconfigure your ROACH/2 with .bof files, when suddenly `fpga.progdev('file.bof')` doesn't work anymore. Well, this page aims to help migrate you and your existing system to using casperfpga and use .fpg files to reconfigure your CASPER Hardware.

Let's first start with some common commands used when communicating with your ROACH/2 in an ipython session using `corr`.

3.2.1 1. Instantiating your ROACH object

The old way

```
In [1]: import corr
In [2]: fpga = corr.katcp_wrapper.FpgaClient('hostname')
```

The new way

```
In [1]: import casperfpga
In [2]: fpga = casperfpga.CasperFpga('hostname')
```

3.2.2 2. Estimating the board's clock speed in MHz

The old way

```
In [3]: fpga.est_brd_clk()
```

The new way

```
In [3]: fpga.estimate_board_clock()
```

3.2.3 3. Reconfiguring your board's firmware

The old way

```
In [4]: fpga.progdev('design_file.bof')
```

The new way

`casperfpga` has moved towards utilising the fpg files that are generated by the toolflow as its header contains much more information about the design. This information is then parsed and is used to create instances of objects such as the communication interface (e.g. 10GbE, 40GbE) and snapshot blocks - both of which are discussed later. This information is taken from, but not limited to, the following sources:

1. core_info.tab

2. design_info.tab
3. git_info.tab

```
In [4]: fpga.upload_to_ram_and_program('design_file.fpg')
```

It is also worth noting at this moment that existing `corr` users will most likely be interfacing with ROACH/2 boards, where the usual .bof files are stored on the ROACH's Network File System (NFS). This however is **not** the case with SKARAB, where an fpg file is stored on and programmed/uploaded from the **client-side**.

This in turn means that if you connect to a board and want to find out what registers are available to you, you would need the fpg file that was written to it in order to parse the information again (and access via fpga.listdev()). This is because the Register-Memory Mapping is not stored on the SKARAB, and is kept local to the ipython session in which it was last programmed/reconfigured.

This is not to say `progdev` has been deprecated... but it has.

3.2.4 4. Accessing 10GbE core(s) and snapshot blocks

The old way

`corr` users will be familiar with

1. Being able to manually configure a 10GbE Core with a name, MAC, IP, port, etc
2. `tap_start` to program a 10GbE device and start the TAP driver
3. `snapshot_get`

The new way

`casperfpga` has moved towards creating separate classes for objects such as GbE interfaces, snapshots, among others. As a result, casperfpga creates instances of these objects using the information on your design provided as part of the header in the fpg file. i.e. There is now a **snap** *class* and **tengbe** and **fortygbe** *classes*.

1. The **snap** class still has methods such as *print_snap()*, which prints out a spreadsheet-style list of all values of the snapshot block to the screen
2. The **tengbe** class is not as intuitive with manually configuring 10GbE cores, however you are still able to:
    1. *get_10gbe_core_details()*
    2. *print_10gbe_core_details()*
    3. *get_arp_details()*
    4. *print_arp_details()*

3.3 casperfpga Sourcecode

3.3.1 CasperLogHandlers

class CasperLogHandlers.**CasperConsoleHandler**(*name*, **args*, ***kwargs*)

Stream Log Handler for casperfpga records

- Trying a custom logger before incorporating into corr2
- This inherits from the logging.Handler - Stream or File

__init__(*name*, **args*, ***kwargs*)
New method added to test logger functionality across transport layers

- Need to add handlers for both Stream AND File

Parameters

- **name** (*str*) – Name of the StreamHandler
- **max_len** – How many log records to store in the FIFO

clear_log()
Clear the list of stored log messages

emit(*record*)
Handle a log record

Parameters **record** – Log record as a string

Returns True/False = Success/Fail

format(*record*)

Parameters **record** – Log record as a string, of type logging.LogRecord

Returns Formatted record

get_log_strings(*num_to_print=1*)
Get all log messages in FIFO associated with a logger entity :param num_to_print:

print_messages(*num_to_print=1*)
Print log messages stored in FIFO :param num_to_print:

set_max_len(*max_len*)

Parameters **max_len** –

CasperLogHandlers.**configure_console_logging**(*logger_entity*, *console_handler_name=None*)
Method to configure logging to console using the casperfpga logging entity

(A similar method exists in casperfpga)

Parameters

- **logger_entity** – Logging entity to create and add the ConsoleHandler to
- **console_handler_name** (*Optional*) – will use logger_entity.name by default

CasperLogHandlers.**configure_file_logging**(*logging_entity*, *filename=None*, *file_dir=None*)
Method to configure logging to file using the casperfpga logging entity

Parameters

- **logging_entity** – Logging entity to create and add the FileHandler to
- **filename** (*Optional*) –
  - must be in the format of filename.log
  - Will default to casperfpga_{hostname}.log
- **file_dir** (*Optional*) – must be a valid path

CasperLogHandlers.**getLogger**(**args*, ***kwargs*)
Custom method allowing us to add default handlers to a logger

Parameters

- **logger_name** – Mandatory, logger needs to have a name!
- **log_level** – All Instrument-level entities log at logging.DEBUG - All Board-level entities log at logging.ERROR

Returns

Tuple

- Boolean Success/Fail, True/False
- Logger entity with ConsoleHandler added as default

CasperLogHandlers.**getNewLogger**(**args*, ***kwargs*)

Custom method allowing us to add default handlers to a logger

Returns

Tuple

- Boolean Success/Fail, True/False
- Logger entity with FileHandler added as default

3.3.2 adc

class adc.**HMCAD1511**(*interface*, *controller_name*, *cs=255*)

```
A_WB_W_3WIRE = 0
CGAIN_DICT_0 = {0:  0, 1:  1, 2:  2, 3:  3, 4:  4, 5:  5, 6:  6, 7:  7, 8:  0, 9:  1,
CGAIN_DICT_1 = {1:  0, 1.25:  1, 2:  2, 2.5:  3, 4:  4, 5:  5, 8:  6, 10:  7, 12.5:  8
CGAIN_ORDER = {1:  [[0, 1, 2, 3, 4, 5, 6, 7]], 2:  [[0, 1, 2, 3], [4, 5, 6, 7]], 4:  [
DICT = [{'rst':  1}, None, None, None, None, None, None, None, None, None, None, None,
FGAIN = (0.00390625, 0.001953125, 0.0009765625, 0.00048828125, 0.000244140625, 0.00012
FGAIN_ORDER = {1:  [0, 2, 5, 7, 3, 1, 6, 4], 2:  [0, 2, 1, 3, 4, 6, 5, 7], 4:  [0, 1,
M_ADC0_CS1 = 1
M_ADC0_CS2 = 2
M_ADC0_CS3 = 4
M_ADC0_CS4 = 8
M_ADC1_CS1 = 16
M_ADC1_CS2 = 32
M_ADC1_CS3 = 64
M_ADC1_CS4 = 128
M_ADC_SCL = 512
M_ADC_SDA = 256
STATE_3WIRE_START = 1
STATE_3WIRE_STOP = 3
STATE_3WIRE_TRANS = 2
```

__init__(*interface*, *controller_name*, *cs=255*)
HMCAD1511 High Speed Multi-Mode 8-Bit 1 GSPS A/D Converter

interface: an instance of casperfpga.CasperFpga controller_name: the name of the adc16_interface cs: Set cs to 0xff if you want all ADC chips share the same configuartion. Or if you want to config them separately (e.g. calibrating interleaving adc gain error for each ADC chip), set cs to 0b1, 0b10, 0b100, 0b1000... for different HMCAD1511 python objects

Here is an example of configuring a register. E.g.

Make an instance of adc adc = HMCAD1511(interface,'adc16_interface')

Select the 2nd and 3rd ADCs, but unselect the 1st ADC. # cs stands for chip select. The last bit of cs is for the 1st ADC adc.cs = 0b110

Target fields you want to configure. They belong to one register # Please refer to HMCAD1511 datasheet for more details # en_lvds_term LVDS buffers # term_lclk<2:0> LCLKN and LCLKP buffers # term_frame<2:0> FCLKN and FCLKP buffers # term_dat<2:0> output data buffers

Get the register address and masks rid, mask = adc._getMask('en_lvds_term') val = adc._set(0x0, 0b1, mask) rid, mask = adc._getMask('term_lclk') val = adc._set(val, 0b011, mask) # 0b11 corresponds to 94ohm rid, mask = adc._getMask('term_frame') val = adc._set(val, 0b011, mask) rid, mask = adc._getMask('term_dat') val = adc._set(val, 0b011, mask)

write value into the register adc.write(val, rid)

Please find more examples of of usage in adc.HMCAD1511.init() or snapadc.py

cGain(*gains*, *cgain_cfg=False*, *fgain_cfg=False*)
Set the coarse gain of the ADC channels

Coarse gain control (parameters in dB). Input gain must be a list of integers. Coarse gain range for HMCAD1511: 0dB ~ 12dB E.g.

cGain([1,5,9,12]) # Quad channel mode in dB step cGain([32,50],cgain_cfg=True) # Dual channel mode in x step cGain([10],fgain_cfg=True) # Single channel mode in dB

step, with fine gain enabled

Coarse gain options when by default cgain_cfg=False: 0 dB, 1 dB, 2 dB, 3 dB, 4 dB, 5 dB, 6 dB, 7 dB, 8 dB, 9 dB, 10 dB, 11 dB and 12 dB

Coarse gain options when cgain_cfg=True: 1x, 1.25x, 2x, 2.5x, 4x, 5x, 8x, 10x, 12.5x, 16x, 20x, 25x, 32x, 50x

fGain(*gains*, *numChannel=1*)
Set the fine gain of the 8 ADC cores

Fine gain control (parameters in dB), input gain rounded towards 0 dB Fine gain range for HMCAD1511: -0.0670dB ~ 0.0665dB E.g.

fGain([-0.06, -0.04, -0.02, 0, 0, 0.02, 0.04, 0.06])

init(*numChannel=4*, *clkDivide=1*, *lowClkFreq=False*)
Reset and initialize ADCs

Please see adc.HMCAD1511.setOperatingMode() for more explanations of the parameters

interleave(*data*, *numChannel*)
Reshape and return ADC data

powerDown()

powerUp()

reset()

selectInput(*inputs*)
Input select

E.g. selectInput([1,2,3,4]) (in four channel mode) selectInput([1,1,1,1]) (in one channel mode)

setOperatingMode(*numChannel*, *clkDivide=1*, *lowClkFreq=False*)
Set interleaving mode and clock divide factor

Available Interleaving mode numChannel=1 – 8 ADC cores per channel numChannel=2 – 4 ADC cores per channel numChannel=4 – 2 ADC cores per channel

Activate lowClkFreq when Single channel Fs < 240 MHz Dual channel Fs < 120 MHz Quad channel Fs < 60 MHz

Availale clock divide factors: 1, 2, 4, and 8

E.g. setOperatingMode(1) setOperatingMode(4, 4)

test(*mode='off'*, *_bits_custom1=None*, *_bits_custom2=None*)
Test ADC LVDS

Set LVDS test patterns

E.g. test('off') test('en_ramp') Ramp pattern 0-255 test('dual_custom_pat', 0xabcd, 0xdcba) Alternate between two custom patterns test('single_custom_pat', 0xaaaa) Repeat a custom pattern test('pat_deskew') Deskew pattern (10101010) test('pat_sync') Sync pattern (11110000)

write(*data*, *addr*)

class adc.**HMCAD1520**(*interface*, *controller_name*, *cs=255*)
HMCAD1520 High Speed Multi-Mode 8/12/14-Bit 1000/640/105 MSPS A/D Converter

Please see docstring of HMCAD1511 for brief description

__init__(*interface*, *controller_name*, *cs=255*)
HMCAD1511 High Speed Multi-Mode 8-Bit 1 GSPS A/D Converter

interface: an instance of casperfpga.CasperFpga controller_name: the name of the adc16_interface cs: Set cs to 0xff if you want all ADC chips share the same configuartion. Or if you want to config them separately (e.g. calibrating interleaving adc gain error for each ADC chip), set cs to 0b1, 0b10, 0b100, 0b1000... for different HMCAD1511 python objects

Here is an example of configuring a register. E.g.

Make an instance of adc adc = HMCAD1511(interface,'adc16_interface')

Select the 2nd and 3rd ADCs, but unselect the 1st ADC. # cs stands for chip select. The last bit of cs is for the 1st ADC adc.cs = 0b110

Target fields you want to configure. They belong to one register # Please refer to HMCAD1511 datasheet for more details # en_lvds_term LVDS buffers # term_lclk<2:0> LCLKN and LCLKP buffers # term_frame<2:0> FCLKN and FCLKP buffers # term_dat<2:0> output data buffers

Get the register address and masks rid, mask = adc._getMask('en_lvds_term') val = adc._set(0x0, 0b1, mask) rid, mask = adc._getMask('term_lclk') val = adc._set(val, 0b011, mask) # 0b11 corresponds to 94ohm rid, mask = adc._getMask('term_frame') val = adc._set(val, 0b011, mask) rid, mask = adc._getMask('term_dat') val = adc._set(val, 0b011, mask)

write value into the register adc.write(val, rid)

Please find more examples of of usage in adc.HMCAD1511.init() or snapadc.py

init(*numChannel=4*, *clkDivide=1*, *lowClkFreq=False*, *resolution=12*)
Reset and initialize ADCs

setOperatingMode(*numChannel*, *clkDivide=1*, *lowClkFreq=False*, *resolution=12*)
Set operating mode and clock divide factor

Available operating mode numChannel=1 Single channel 12-bit numChannel=2 Dual channel 12-bit numChannel=4 Quad channel 12-bit numChannel=0 Quad channel 14-bit (not supported yet)

Available resolutions: resolution=8 resolution=12 resolution=14 (not supported yet)

Availale clock divide factors: 1, 2, 4, and 8

Activate lowClkFreq when High speed, single channel Fs < 240 MHz High speed, dual channel Fs < 120 MHz High speed, quad channel Fs < 60 MHz Precision mode Fs < 30 MHz

E.g. setOperatingMode(1, 4, False, 8) # 1 channel, 8-bit resolution, 8-bit width setOperatingMode(1, 4, False, 12) # 1 channel, 12-bit resolution, 12-bit width setOperatingMode(4, 1, False, 14) # 4 channels, 14-bit resolution, 16-bit width. (Currently not supported)

3.3.3 attribute_container

class attribute_container.**AttributeContainer**
An iterable class to make registers, snapshots, etc more accessible.

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

clear()

keys()

names()

remove_attribute(*attribute*)
Remove an attribute from this container by name.

Parameters **attribute** – the name of the attribute to remove

3.3.4 bitfield

class bitfield.**Bitfield**(*name*, *width_bits*, *fields=None*)
Describes a chunk of memory that consists of a number of Fields.

__init__(*name*, *width_bits*, *fields=None*)

Parameters

- **name** (*str*) – name of the device
- **width_bits** (*int*) – Bit-width of the Bitfield
- **fields** (*int*) – number of fields - default to None

field_add(*newfield*, *auto_offset=False*)
Add a Field to this bitfield.

field_get_by_name(*field_name*)
Get a field from this bitfield by its name.

Parameters **field_name** (*str*) – name of field to search for

field_names()

fields_add(*fields*)
Add a dictionary of Fields to this bitfield.

fields_clear()
Reset the fields in this bitstruct.

fields_string_get()
Get a string of all the field names.

class bitfield.**Field**(*name*, *numtype*, *width_bits*, *binary_pt*, *lsb_offset*)
A Field object is a number of bits somewhere in a Bitfield object.

__init__(*name*, *numtype*, *width_bits*, *binary_pt*, *lsb_offset*)
Initialise a Field object.

Parameters

- **name** – The name of the field
- **numtype** – A numerical description of the type:
  - 0 is unsigned
  - 1 is signed 2's comp
  - 2 is boolean
- **width_bits** – The width of the field, in bits
- **binary_pt** – The binary point position, in bits
- **lsb_offset** – The offset in the memory field, in bits:
  - 1 means it hasn't been set yet.

bitfield.**clean_fields**(*parent_name*, *parent_type*, *field_str*)
Take the Simulink string for a field and return a list

Parameters

- **parent_name** – the BitField that will run this
- **parent_type** – register, snapshot, etc
- **field_str** – the string to be parsed

3.3.5 casperfpga

3.3.6 clockswitch

class clockswitch.**HMC922**(*interface*, *controller_name*)
HMC922 DIFFERENTIAL SPDT SWITCH

__init__(*interface*, *controller_name*)
x.__init__(...) initializes x; see help(type(x)) for signature

getSwitch()
getSwitch returns the current signal path being selected

setSwitch(*clk*)
setSwitch('a') or setSwitch('b')

3.3.7 fortygbe

class fortygbe.**FortyGbe**(*parent*, *name*, *address*, *length_bytes*, *device_info=None*, *position=None*)

__init__(*parent*, *name*, *address*, *length_bytes*, *device_info=None*, *position=None*)
Implements the Gbe class. This is normally initialised from_device_info.

Parameters

- **parent** – The parent object, normally a CasperFpga instance
- **name** – The name of the device
- **position** – Optional - defaulted to None
- **address** – Integer
- **length_bytes** – Integer
- **device_info** – Information about the device

static convert_128_to_64(*w128*)

fabric_disable()
Disables 40G core fabric interface. :return:

fabric_enable()
Enables 40G core fabric interface. :return:

classmethod from_device_info(*parent*, *device_name*, *device_info*, *memorymap_dict*, ***kwargs*)
Process device info and the memory map to get all necessary info and return a TenGbe instance.

Parameters

- **parent** – the parent device, normally an FPGA instance
- **device_name** – the unique device name
- **device_info** – information about this device
- **memorymap_dict** – a dictionary containing the device memory map

Returns a TenGbe object

get_arp_details(*port_dump=None*)
Get ARP details from this interface.

Parameters **port_dump** (*list*) – A list of raw bytes from interface memory; if not supplied, fetch from hardware.

get_gbe_core_details(*read_arp=False*, *read_cpu=False*)
Get the details of the ethernet core from the device memory map. Updates local variables as well.

get_hw_gbe_stats(*rst_counters=False*)
Get the traffic statistics of the ethernet core from the device memory map. ::param:: rst_counters: reset the counters after reading them. :return:

get_ip()
Retrieve core's IP address from HW.

Returns IpAddress object

get_mac()
Retrieve core's configured MAC address from HW.

Returns Mac object

get_port()
Retrieve core's port from HW.

Returns int

get_stats()
Retrieves some statistics for this core. Needs to have the debug registers compiled-in to the core at 32b.

ip_address

mac

multicast_receive(*ip_str*, *group_size*, *port=7148*)
Send a request to KATCP to have this tap instance send a multicast group join request.

Parameters

- **ip_str** – A dotted decimal string representation of the base mcast IP address.
- **group_size** – An integer for how many additional mcast addresses (from base) to subscribe to. Must be (2^N-1), ie 0, 1, 3, 7, 15 etc.
- **port** – The UDP port on which you want to receive. Note that only one port is possible per interface (ie it's global and will override any other port you may have configured).

port

post_create_update(*raw_device_info*)
Update the device with information not available at creation.

Parameters **raw_device_info** – info about this block that may be useful

print_arp_details(*refresh=False*, *only_hits=False*)
Print nicely formatted ARP info. :param refresh: :param only_hits:

print_gbe_core_details(*arp=False*, *cpu=False*, *refresh=True*)
Prints 40GbE core details.

static process_snap_data(*d*)

read_rxsnap()
Read the RX snapshot embedded in this GbE yellow block

read_txsnap()
Read the TX snapshot embedded in this GbE yellow block

set_port(*port*)

Parameters **port** –

3.3.8 gbe

class gbe.**Gbe**(*parent*, *name*, *address*, *length_bytes*, *device_info=None*)
A (multi)gigabit network interface on a device.

__init__(*parent*, *name*, *address*, *length_bytes*, *device_info=None*)
Most of initialised from_device_info in child classes

Parameters

- **parent** –
- **name** –

- **device_info** –

fabric_disable()
Enable the core fabric

fabric_enable()
Enable the core fabric

classmethod from_device_info(*parent*, *device_name*, *device_info*, *memorymap_dict*, ***kwargs*)
Process device info and the memory map to get all necessary info and return a Gbe instance.

Parameters

- **parent** – the parent device, normally an FPGA instance
- **device_name** – the unique device name
- **device_info** – information about this device
- **memorymap_dict** – a dictionary containing the device memory map

Returns a Gbe object

get_arp_details(*port_dump=None*)
Get ARP details from this interface.

Parameters **port_dump** (*list*) – A list of raw bytes from interface memory.

get_cpu_details(*port_dump=None*)
Read details of the CPU buffers.

Parameters **port_dump** –

get_gbe_core_details(*read_arp=False*, *read_cpu=False*)

Parameters

- **read_arp** –
- **read_cpu** –

ip_address

mac

multicast_receive(*ip_str*, *group_size*)
Send a multicast group join request.

Parameters

- **ip_str** – A dotted decimal string representation of the base mcast IP address.
- **group_size** – An integer for how many mcast addresses from base to respond to.

multicast_remove(*ip_str*)
Send a request to be removed from a multicast group.

Parameters **ip_str** – A dotted decimal string representation of the base mcast IP address.

port

post_create_update(*raw_device_info*)
Update the device with information not available at creation.

Parameters **raw_device_info** – info about this block that may be useful

print_arp_details(*refresh=False*, *only_hits=False*)
Print nicely formatted ARP info.

Parameters

- **refresh** –
- **only_hits** –

print_cpu_details(*refresh=False*)
Print nicely formatted CPU details info.

Parameters **refresh** –

print_gbe_core_details(*arp=False*, *cpu=False*, *refresh=True*)
Prints 10GbE core details.

Parameters

- **arp** (*boolean*) – include the ARP table
- **cpu** (*boolean*) – include the CPU packet buffers
- **refresh** – read the 10gbe details first

process_device_info(*device_info*)
Process device info to setup GbE object

Parameters **device_info** – Dictionary including:

- IP Address
- Mac Address
- Port number

read_counters()
Read all the counters embedded in this TenGBE yellow block

read_rx_counters()
Read all RX counters embedded in this TenGBE yellow block

read_rxsnap()
Read the RX snapshot embedded in this GbE yellow block

read_tx_counters()
Read all TX counters embedded in this TenGBE yellow block

read_txsnap()
Read the TX snapshot embedded in this GbE yellow block

rx_okay(*wait_time=0.2*, *checks=10*)

Is this gbe core receiving okay? i.e. _rxctr incrementing and _rxerrctr not incrementing

Parameters

- **wait_time** – seconds to wait between checks
- **checks** – times to run check

Returns True/False

setup(*mac*, *ipaddress*, *port*, *gateway=None*, *subnet_mask=None*)
Set up the MAC, IP and port for this interface

Parameters

- **mac** – String or Integer input, MAC address (e.g. '02:00:00:00:00:01')

- **ipaddress** – String or Integer input, IP address (eg '10.0.0.1')
- **port** – String or Integer input

tx_okay(*wait_time=0.2*, *checks=10*)

Is this gbe core transmitting okay? i.e. _txctr incrementing and _txerrctr not incrementing

Parameters

- **wait_time** – seconds to wait between checks
- **checks** – times to run check

Returns True/False

3.3.9 i2c

class i2c.**I2C**(*fpga*, *controller_name*, ***kwargs*)

__init__(*fpga*, *controller_name*, ***kwargs*)
I2C module for I2C yellow block

fpga: casperfpga.CasperFpga instance controller_name: The name of the I2C yellow block retry_wait: Time interval between pulling status of I2C module,

Default value is 0.02. Typical range between [0.1, 0.001].

disable_core()
Disable the wb-i2c core.

- Set the I2C enable bit to 0,
- Set the interrupt bit to 0 (disabled).

enable_core()
Enable the wb-i2c core.

- Set the I2C enable bit to 1,
- Set the interrupt bit to 0 (disabled).

getClock(*reference=None*)
Get I2C clock speed

If the reference clock speed is not provided, this method returns the preScale, which equals to:

preScale = int((reference*1e3/(5*target))-1)

where target is the desired I2C clock speed in kHz. Reference clock speed is in MHz.

```
getClock()          # Returns the value of the divider
getClock(100)       # Returns the I2C clock speed given a reference
                    # clock speed at 100 MHz
```

getStatus()
Get current status of the I2C module

The status is kept in a dict structure, items of which include:

- ACK Acknowledge from Slave
- BUSY Busy i2c bus

- ARB Lost Arbitration
- TIP Transfer in Progress
- INT Interrupt Pending

probe()

read(*addr*, *cmd=None*, *length=1*)
I2C read

Read arbitary number of bytes from an internal address of a slave device. Some I2C datasheets refer to internal address as command (cmd) as well.

Parameters

- **addr** – 7-bit integer, address of the slave device
- **cmd** – a byte of a list of bytes, the internal address of the slave device
- **length** – non-negative integer, the number of bytes to read

The return is a byte when length==1, or a list of bytes otherwise.

```
read(0x40)  # Read a byte from the slave device at 0x40, without
            # specifying an internal address
read(0x40,0xe3) # Read a byte from the internal address 0xe3 of the slave
                # at 0x40
read(0x40,length=3) # Read 3 bytes from the slave at 0x40 without specifying
                    # an internal address
read(0x40,[0xfa,0x0f],4)    # Read 4 bytes from the internal address [0xfa,
↪0x0f]
                            # of the slave at 0x40
```

setClock(*target*, *reference=100*)
Set I2C bus clock

The I2C module uses a divider to generate its clock from a reference clock, e.g. a system clock at 100 MHz. The acutally generated I2C clock speed might be slightly different from the target clock speed specified. Reference clock speed in MHz and target clock speed in kHz

```
setClock(10,100)    # Set I2C bus clock speed to 10 kHz, given a system clock␣
↪of 100 MHz
```

write(*addr*, *cmd=None*, *data=None*)
I2C write

Write arbitary number of bytes to an internal address of a slave device. Some I2C datasheets refer to internal address as command (cmd) as well.

Parameters

- **addr** – 7-bit integer, address of the slave device
- **cmd** – a byte of a list of bytes, the internal address of the slave device
- **data** – a byte of a list of bytes to write

```
write(0x40,0x1)     # Write 0x1 to slave device at 0x40
write(0x40,0x1,0x2) # Write 0x2 to the internal address 0x1 of the
                    # slave device at address 0x40
write(0x40,data=0x2)    # Write 0x2 to the slave at 0x40, without specifying
                        # an internal address
```

(continues on next page)

(continued from previous page)

```
write(0x40,[0x1,0x2],[0x3,0x4]) # Write [0x3,0x4] to the internal address␣
↪[0x1,0x2]
                                # of the slave at 0x40
```

class i2c.**I2C_DEVICE**(*itf*, *addr*)

I2C device base class

DICT = {}

__init__(*itf*, *addr*)

x.__init__(...) initializes x; see help(type(x)) for signature

getRegister(*rid=None*)

getWord(*name*)

read(*reg=None*, *length=1*)

setWord(*name*, *value*)

write(*reg=None*, *data=None*)

class i2c.**I2C_PIGPIO**(*sda*, *scl*, *baud*)

__init__(*sda*, *scl*, *baud*)

PIGPIO based I2C

I2C module powered by PIGPIO library.

Parameters

- **sda** – The gpio number of sda pin.
- **scl** – The gpio number of scl pin
- **baud** – The baud rate of the I2C bus

Be noticed that gpio number is different from pin number!

read(*addr*, *cmd=None*, *length=1*)

I2C read

Read arbitary number of bytes from an internal address of a slave device. Some I2C datasheets refer to internal address as command (cmd) as well.

Parameters

- **addr** – 7-bit integer, address of the slave device
- **cmd** – a byte of a list of bytes, the internal address of the slave device
- **length** – non-negative integer, the number of bytes to read

The return is a byte when length==1, or a list of bytes otherwise.

```
read(0x40)  # Read a byte from the slave device at 0x40, without
            # specifying an internal address
read(0x40,0xe3) # Read a byte from the internal address 0xe3 of the slave
                # at 0x40
read(0x40,length=3) # Read 3 bytes from the slave at 0x40 without specifying
                    # an internal address
```

(continues on next page)

(continued from previous page)

```
read(0x40,[0xfa,0x0f],4)     # Read 4 bytes from the internal address [0xfa,
↪0x0f]
                             # of the slave at 0x40
```

write(*addr*, *cmd=None*, *data=None*)
I2C write

Write arbitary number of bytes to an internal address of a slave device. Some I2C datasheets refer to internal address as command (cmd) as well.

Parameters

- **addr** – 7-bit integer, address of the slave device
- **cmd** – a byte of a list of bytes, the internal address of the slave device
- **data** – a byte of a list of bytes to write

```
write(0x40,0x1)      # Write 0x1 to slave device at 0x40
write(0x40,0x1,0x2)  # Write 0x2 to the internal address 0x1 of the
                     # slave device at address 0x40
write(0x40,data=0x2)     # Write 0x2 to the slave at 0x40, without specifying
                         # an internal address
write(0x40,[0x1,0x2],[0x3,0x4]) # Write [0x3,0x4] to the internal address␣
↪[0x1,0x2]
                                # of the slave at 0x40
```

class i2c.**I2C_SMBUS**(*devid*)

__init__(*devid*)

read(*addr*, *cmd=None*, *length=1*)

write(*addr*, *cmd=None*, *data=[]*)

3.3.10 i2c_bar

class i2c_bar.**MS5611_01B**(*itf*, *addr=119*)

OSR_PRESS = {256: 64, 512: 66, 1024: 68, 2048: 70, 4096: 72}

OSR_TEMP = {256: 80, 512: 82, 1024: 84, 2048: 86, 4096: 88}

__init__(*itf*, *addr=119*)

crc4(*data*)
input: 16bit * 8 data list

init()

read(*reg=None*, *length=1*)

readCalibration()

readPress(*rawtemp=None*, *dt=None*, *osr=4096*)
Return air pressure

```
readPress()      # return raw data with osr=4096
readPress(2007,2366)     # return return compensated data with osr=4096
```

To get rawtemp and dt:

```
rawtemp, dt = readTemp(raw=True)
```

readTemp(*raw=False*, *osr=4096*)
Return the temperature

```
readTemp()      # return the temperature
readTemp(raw=True)  # return the raw temperature data together with dt
```

reset()

toAltitude(*bar*, *temp*, *P0=1013.25*)
Convert air pressure and temperature to altitude

- bar is air pressure, temp is temperature
- P0 is air presure at sea level

```
toAltitude(1000.09,20.07)
```

To get raw temp:

```
rawtemp, dt = readTemp(raw=True)
```

To get bar:

```
bar = readPress(rawtemp, dt)
```

write(*reg=None*, *data=None*)

3.3.11 i2c_eeprom

class i2c_eeprom.**EEP24XX64**(*itf*, *addr=81*)
64 Kbit Electrically Erasable PROM

__init__(*itf*, *addr=81*)

read(*reg*, *length=1*)
Read byte(s) out of ROM

```
read(0)     # read a byte out from address 0 of the ROM
read(0x20,16)   # read 200 bytes from address 0x20 to 0x2f
```

readString()
Read a string out of the ROM

Read byte(s) and intepret as ASCII character(s). Expect a '' at the end of the string

size = 8192

write(*reg*, *data*)
Write byte(s) into ROM

```
write(0,0xff)       # write 0xff to address 0x00
write(0x10,range(8192)) # write range(8192) to address from 0x10 to 0x1fff
```

writeString(*chars*)
Write a string into the ROM

Write the input string into the ROM. Only ASCII characters are allowed A character of '' will be appended to the string to indicate the end of the string.

```
writeString('Haha')
```

3.3.12 i2c_gpio

class i2c_gpio.**PCF8574**(*itf*, *addr=32*)

Operate a PCF8574 chip. PCF8574 is a Remote 8-Bit I/O Expander for I2C BUS

__init__(*itf*, *addr=32*)

read()

write(*data*)

3.3.13 i2c_motion

class i2c_motion.**AK8963**(*itf*, *addr=12*)

DICT = {0: {'wia': 255}, 1: {'info': 255}, 2: {'DOR': 2, 'DRDY': 1, 'st1': 255},

WHOAMI = 72

__init__(*itf*, *addr=12*)

AK8963 magnetometer

If integrated with mpu9250, make an instance of mpu9250 and enable aux i2c before using AK8963

adj = array([0, 0, 0])

getAdjustment()

getWord(*name*)

init()

Initialise AKB8963

mag

read(*reg*, *length=1*)

reset()

selftest()

setWord(*name*, *value*)

whoami()

write(*reg*, *data*)

class i2c_motion.**IMUSimple**(*bus, mpuaddr=105, akaddr=None, orient=[[1, 0, 0], [0, 1, 0], [0, 0, 1]]*)

__init__(*bus, mpuaddr=105, akaddr=None, orient=[[1, 0, 0], [0, 1, 0], [0, 0, 1]]*)

IMUSimple IMU usage demo

```
bus = i2c.I2C_IPGPIO(2,3,15000)
imu = i2c_motion.IMUSimple(bus,0x69,0x0c)
imu.init()
print(imu.pose)
```

Default orientation:

- x+ east
- y+ north
- z+ upward
- IMU chip pin 1 in 4th quadrant
- IMU chip top side upwards

provide other orientations in the following format:

```
imu = IMUSimple(bus,0x69, orient =
            [[1,0,0],     # new x+ in old coordinate system
            [0,1,0],     # new y+ in old coordinate system
            [0,0,1]])    # new z+ in old coordinate system
```

accel

calcRotationMatrix(*dst*)

gyro

init()

mag

pose

class i2c_motion.**MPU9250**(*itf*, *addr=104*)

9-axis MotionTracking device that combines a 3-axis gyroscope, 3-axis accelerometer, 3-axis magnetometer and a Digital Motion Processor (DMP) all in a small 3x3x1mm package available as a pin-compatible upgrade from the MPU-6515

DICT = {0: {'XG_ST_DATA': 255}, 1: {'YG_ST_DATA': 255}, 2: {'ZG_ST_DATA': 255}, 13:

WHOAMI = 113

__init__(*itf*, *addr=104*)

accel

accel_offs

accel_scale

calibrate(*WRITE_OFFSET_REG=True*)

Not working

getRegister(*rid=None*)

getWord(*name*)

gyro

gyro_offs

gyro_scale

init(*gyro=True*, *accel=True*, *lowpower=False*)
Initialise MPU9250

interpretAccel(*data*, *scale=None*)

interpretGyro(*data*, *scale=None*)

powerOff()

read(*reg*, *length=1*)

readFIFO(*types*, *length=None*, *filename=None*, *raw=False*, *wait=0.001*)
read FIFO and sort data into categories according to given types

```
readFIFO({'accel':True,'gyro':True})
readFIFO({'accel':True,'gyro':True},length=1200)
readFIFO({'accel':True,'slv0':8},filename='/tmp/data.txt')
```

readFIFOCount()
Read the number of data available in the FIFO

reset()

setAccelSamplingRate(*conf*)
Accelerometer sampling rate

parameter conf is in range(9)

num	bandwidth (Hz)	delay (ms)	fs (Hz)
0	1130	0.75	4000
1	460	1.94	1000
2	184	5.8	1000
3	92	7.8	1000
4	41	11.8	1000
5	20	19.8	1000
6	10	35.7	1000
7	5	66.96	1000
8	460	1.94	1000

setFIFO(*accel=False*, *temp=False*, *gyro=False*, *slv0=False*, *slv1=False*, *slv2=False*, *slv3=False*)

setGyroSamplingRate(*conf*)
Gyroscope sampling rate

parameter conf is in range(10)

num	bandwidth (Hz)	delay (ms)	fs (Hz)
0	8800	0.064	32000
1	3600	0.11	32000
2	250	0.97	8000
3	184	2.9	1000
4	92	3.9	1000
5	41	5.9	1000
6	20	9.9	1000
7	10	17.85	1000
8	5	33.48	1000
9	3600	0.17	8000

setWord(*name*, *value*)

sortFIFOData(*types*, *data=None*)
Sort a serial of data into categories according to given types

```
sortFIFOData({'accel':True,'gyro':True},[0,0,16383,0,0,0])
```

whoami()

write(*reg*, *data*)

i2c_motion.**signed**(*data*, *bitwidth*)
Convert an unsigned data into a 32-bit signed data

```
signed(0xfff,12) # it's -1
signed(0xff,12) # it's 255
```

3.3.14 i2c_temp

class i2c_temp.**Si7051**(*itf*, *addr=64*, *resolution=14*)
Si7051 I2C Temperature Sensors

__init__(*itf*, *addr=64*, *resolution=14*)

cmdFirmRev = [132, 184]
Read Firmware Revision

cmdMeasure = 227
Measure temperature

cmdMeasureN = 243

cmdSNA = [250, 15]
Read Serial Number

cmdSNB = [252, 201]

cmdUserRegR = 231

cmdUserRegW = 230
Write and read user register for resolution config and VDD status checking

crc8(*data*, *poly*, *initVal=0*, *bigendian=True*)

crcInitVal = 0

crcPoly = 305
CRC generator polynomial and initial value

read(*reg=None*, *length=1*)

readTemp()

resBase = 11
Resolution related numbers

resD0Mask = 1

resD1Mask = 128

resList = [0, 1, 128, 129]
11 bit, 12 bit, 13 bit, 14 bit

resTop = 14

sn()
64-bit big-endian serial number with CRC

strFirmRev = {32: 'Firmware version 2.0', 255: 'Firmware version 1.0'}

vddOK = 0

vddStatusMask = 64

write(*reg=None*, *data=None*)

3.3.15 i2c_volt

class i2c_volt.**INA219**(*itf*, *addr=69*)
INA219 Zero-Drift, Bidirectional Current/Power Monitor With I2C Interface

ADC = {1: 8, 2: 9, 4: 10, 8: 11, 16: 12, 32: 13, 64: 14, 128: 15, '10b': 1, '

BRNG = {16: 0, 32: 1}

**DICT = {0: {'BADC': 1920, 'BRNG': 8192, 'MODE': 7, 'PG': 6144, 'RST': 32768, 'SADC': **

PG = {40: 0, 80: 1, 160: 2, 320: 3}

__init__(*itf*, *addr=69*)

getRegister(*rid=None*)

getStatus(*name='CNVR'*)

getWord(*name*)

init(*brng=16*, *pg=320*, *badc=128*, *sadc=128*, *mode=3*)
Initialise INA219

Mode, available options: MODE3 MODE2 MODE1 MODE 0 0 0 Power-down 0 0 1 Shunt voltage, triggered 0 1 0 Bus voltage, triggered 0 1 1 Shunt and bus, triggered 1 0 0 ADC off (disabled) 1 0 1 Shunt voltage, continuous 1 1 0 Bus voltage, continuous 1 1 1 Shunt and bus, continuous

BRNG, Bus voltage range, available options: 16,32

PG, for choosing the full scale range. Available options: 40, 80, 160, 320,

ADC, for bus voltage as well as shunt voltage measurement. Available options: Do one sample of the following resolution '9b', '10b', '11b', '12b', or do 12bit resolution and average over the following number of samples: 1, 2, 4, 8, 16, 32, 64, 128,

read(*reg=None*, *length=2*)

readVolt(*name*)
Read Voltage

Please switch to corresponding modes using init() before measuring voltage. Possible options are:

'shunt' 'bus'

E.g. readVolt('shunt')

setWord(*name*, *value*)

write(*reg=None*, *data=None*)

class i2c_volt.**LTC2990**(*itf*, *addr=79*)
Quad I2C Voltage, Current and Temperature Monitor

CDIFFERENTIAL = 1.942e-05

CSINGLEENDED = 0.00030518

DICT = {0: {'BUSY': 1, 'TINTREADY': 2, 'V1READY': 4, 'V2READY': 8, 'V3READY': 16, 'V4

MODE0 = {0: ['v1', 'v2', 'tr2', 'tr2'], 1: ['v1-v2', 'v1-v2', 'tr2', 'tr2'], 2: ['v

MSB = {'TEMP': {'DATA': 63, 'DV': 128, 'SO': 32, 'SS': 64}, 'VOLT': {'DATA': 127, 'DV'

TEMPFACTOR = 16.0

VCCBIAS = 2.5

__init__(*itf*, *addr=79*)

fmt = 0

getRegister(*rid=None*)

getStatus(*name=None*)

getWord(*name*)

init(*fmt='celsius'*, *repeat=False*, *mode1=3*, *mode0=7*)
Initialise LTC2990

mode0	Description
0	V1, V2, TR2(Default)
1	V1-V2, TR2
2	V1-V2, V3, V4
3	TR1, V3, V4
4	TR1, V3-V4
5	TR1, TR2
6	V1-V2, V3-V4
7	V1, V2, V3, V4

mode1	Description
0	Internal Temperature Only (Default)
1	TR1, V1 or V1-V2 Only per Mode[2:0]
2	TR2, V3 or V3-V4 Only per Mode[2:0]
3	All Measurements per Mode[2:0]

mode0 = 0

mode1 = 3

read(*reg=None*, *length=1*)

readTemp()

readVolt(*name*)
Read Voltage

Please switch to corresponding modes using init() before measuring voltage.

Possible options are:

- vcc
- v1
- v2
- v3

- v4
- v1-v2
- v3-v4

```
readVolt('v1-v2')
```

repeat = 1

setWord(*name*, *value*)

write(*reg=None*, *data=None*)

class i2c_volt.**MAX11644**(*itf*, *addr=54*)

LSB = 0.001

__init__(*itf*, *addr=54*)
x.__init__(...) initializes x; see help(type(x)) for signature

init(***kwargs*)

readVolt(*name=None*)
Read voltage Possible options are

AIN0 AIN1

reset()

i2c_volt.**str2int**(*s*)

3.3.16 katadc

Created on Feb 28, 2013

@author: paulp

class katadc.**KatAdc**(*parent*, *name*, *address*, *length*, *device_info*)
Information above KatAdc yellow blocks. Seems to be called most often via from_device_info.

__init__(*parent*, *name*, *address*, *length*, *device_info*)
Initialise a KatAdc object with the following parameters.

Parameters

- **parent** – The owner of this block.
- **name** – The name of this block.
- **address** –
- **length** –
- **device_info** –

classmethod from_device_info(*parent*, *device_name*, *device_info*, *memorymap_dict*, ***kwargs*)
Process device info and the memory map to get all necessary info and return a KatAdc instance.

Parameters

- **device_name** – the unique device name
- **device_info** – information about this device

- **memorymap_dict** – a dictionary containing the device memory map

Returns a KatAdc object

3.3.17 memory

The base class for all things memory. More or less everything on the FPGA is accessed by reading and writing memory addresses on the EPB/OPB busses. Normally via KATCP.

class memory.**Memory**(*name*, *width_bits*, *address*, *length_bytes*)
Memory on an FPGA.

__init__(*name*, *width_bits*, *address*, *length_bytes*)
A chunk of memory on a device.

Parameters

- **name** – a name for this memory
- **width_bits** – the width, in BITS, PER WORD
- **address** – the start address in device memory
- **length_bytes** – length, in BYTES

e.g. a Register has width_bits=32, length_bytes=4

e.g.2. a Snapblock could have width_bits=128, length_bytes=32768

length_in_words()

Returns the memory block's length, in Words

read(***kwargs*)
Read raw binary data and convert it using the bitfield description for this memory.

Returns (data dictionary, read time)

read_raw(***kwargs*)
Placeholder for child classes.

Returns (rawdata, timestamp)

write(***kwargs*)

write_raw(*uintvalue*)

memory.**bin2fp**(*raw_word*, *bitwidth*, *bin_pt*, *signed*)
Convert a raw number based on supplied characteristics.

Parameters

- **raw_word** – the number to convert
- **bitwidth** – its width in bits
- **bin_pt** – the location of the binary point
- **signed** – whether it is signed or not

Returns the formatted number, long, float or int

memory.**cast_fixed**(*fpnum*, *bitwidth*, *bin_pt*)
Represent a fixed point number as an unsigned number, like the Xilinx reinterpret block.

Parameters

- **fpnum** –
- **bitwidth** –
- **bin_pt** –

memory.**fp2fixed**(*num*, *bitwidth*, *bin_pt*, *signed*)
Convert a floating point number to its fixed point equivalent.

Parameters

- **num** –
- **bitwidth** –
- **bin_pt** –
- **signed** –

memory.**fp2fixed_int**(*num*, *bitwidth*, *bin_pt*, *signed*)
Compatability function, rather use the other functions explicitly.

3.3.18 network

class network.**IpAddress**(*ip*)
An IP address.

__init__(*ip*)
x.__init__(...) initializes x; see help(type(x)) for signature

static ip2str(*ip_addr*)
Convert an IP in integer form to a human-readable string.

is_multicast()
Does the data source's IP address begin with 239?

packed()

static str2ip(*ip_str*)
Convert a human-readable IP string to an integer.

class network.**Mac**(*mac*)
A MAC address. Can either be initialised with a IP-string or 32-bit integer.

__init__(*mac*)
x.__init__(...) initializes x; see help(type(x)) for signature

classmethod from_hostname(*hostname*, *port_num*)

classmethod from_roach_hostname(*hostname*, *port_num*)
Make a MAC address object from a ROACH hostname

static mac2str(*mac*)
Convert a MAC in integer form to a human-readable string.

packed()

static str2mac(*mac_str*)
Convert a human-readable MAC string to an integer.

3.3.19 qdr

Created on Fri Mar 7 07:15:45 2014

@author: paulp

class qdr.**Qdr**(*parent*, *name*, *address*, *length_bytes*, *device_info*, *ctrlreg_address*)
Qdr memory on an FPGA.

__init__(*parent*, *name*, *address*, *length_bytes*, *device_info*, *ctrlreg_address*)
Make the QDR instance, given a parent, name and info from Simulink.

- Most often called from_device_info

Parameters

- **parent** – Parent device who owns this Qdr
- **name** – A unique device name
- **address** – Address of the Qdr in memory
- **length_bytes** – Length of the Qdr in memory
- **device_info** – Information about this Qdr device
- **ctrlreg_address** –

classmethod from_device_info(*parent*, *device_name*, *device_info*, *memorymap_dict*, ***kwargs*)
Process device info and the memory map to get all necessary info and return a Qdr instance.

Parameters

- **parent** –
- **device_name** – the unique device name
- **device_info** – information about this device
- **memorymap_dict** – a dictionary containing the device memory map

Returns a Qdr object

qdr_cal(*fail_hard=True*)

qdr_cal_check(*step=-1*, *quickcheck=True*)
Checks calibration on a qdr. Raises an exception if it failed.

Parameters

- **step** – the current step
- **quickcheck** – if True, return after the first error, else process all test vectors before returning

qdr_check_cal_any_good(*current_step*, *checkoffset=4194304*)
Checks calibration on a qdr.

Parameters

- **current_step** – what is the current output delay step
- **checkoffset** – where to write the test data

Returns True if *any* of the bits were good

qdr_reset()
Resets the QDR and the IO delays (sets all taps=0).

reset()
Reset the QDR controller by toggling the lsb of the control register. Sets all taps to zero (all IO delays reset).

qdr.**find_cal_area**(*a*)
Given a vector of pass (1) and fail (-1), find contiguous chunks of 'pass'.

Parameters **a** – Vector input (list?)

Returns Tuple - (max_so_far, begin_index, end_index)

qdr.**log10**(*msg*)

qdr.**log11**(*msg*)

qdr.**log12**(*msg*)

qdr.**log13**(*msg*)

3.3.20 register

class register.**Register**(*parent*, *name*, *address*, *device_info=None*, *auto_update=False*)
A CASPER register on an FPGA.

__init__(*parent*, *name*, *address*, *device_info=None*, *auto_update=False*)

Parameters

- **parent** –
- **name** –
- **address** –
- **device_info** –
- **auto_update** –

blindwrite(***kwargs*)
As write, but without checking the result

classmethod from_device_info(*parent*, *device_name*, *device_info*, *memorymap_dict*, ***kwargs*)
Process device info and the memory map to get all necessary info and return a Register instance.

Parameters

- **parent** – the parent device
- **device_name** – the unique device name
- **device_info** – information about this device
- **memorymap_dict** – a dictionary containing the device memory map

Returns a Register object

info()
Return a string with information about this Register instance.

process_info(*info*)
Set this Register's extra information.

read(***kwargs*)
Memory.read returns a list for all bitfields, so just put those values into single values.

read_raw(***kwargs*)
Read a raw 4-byte value from the host device. Size is 4-bytes.

Parameters **kwargs** –

read_uint(***kwargs*)

write(***kwargs*)
Write fields in a register, using keyword arguments for fields

Parameters **kwargs** –

write_int(*uintvalue*, *blindwrite=False*, *word_offset=0*)
Write an unsigned integer to this device using the fpga client.

write_raw(*data*, *blindwrite=False*)
Use the katcp_client_fpga write integer function.

write_single(*value*)
Write single value.

Parameters **value** –

3.3.21 sbram

class sbram.**Sbram**(*parent*, *name*, *address*, *length_bytes*, *width_bits*, *device_info=None*)
General SBRAM memory on the FPGA.

__init__(*parent*, *name*, *address*, *length_bytes*, *width_bits*, *device_info=None*)
A chunk of memory on a device.

Parameters

- **name** – a name for this memory
- **width_bits** – the width, in BITS, PER WORD
- **address** – the start address in device memory
- **length_bytes** – length, in BYTES

e.g. a Register has width_bits=32, length_bytes=4

e.g.2. a Snapblock could have width_bits=128, length_bytes=32768

classmethod from_device_info(*parent*, *device_name*, *device_info*, *memorymap_dict*, ***kwargs*)
Process device info and the memory map to get all necessary info and return a Sbram instance.

Parameters

- **parent** – the CasperFpga that hosts this SBRAM
- **device_name** – the unique device name
- **device_info** – information about this device
- **memorymap_dict** – a dictionary containing the device memory map

Returns a Sbram object

read_raw(***kwargs*)
Read raw data from memory.

3.3.22 scroll

Playing with ncurses in Python to scroll up and down, left and right, through a list of data that is periodically refreshed.

Revs: 2010-12-11: JRM Added concat for status line to prevent bailing on small terminals. Code cleanup to prevent modification of external variables. Added left, right page controls

class scroll.**Screenline**(*data*, *xpos*, *ypos*, *cr=True*, *fixed=False*, *attributes=0*)

__init__(*data*, *xpos*, *ypos*, *cr=True*, *fixed=False*, *attributes=0*)

Parameters

- **data** – A String to be placed on the screen
- **xpos** – x position, -1 means at current xpos
- **ypos** – y position, -1 means at current ypos
- **ypos** – if True, start a new line after this one
- **fixed** – if True, always at this pos from top left, scrolling makes no difference
- **attributes** – Curses string attributes

class scroll.**Scroll**(*debug=False*)

Scrollable ncurses screen.

__init__(*debug=False*)

x.__init__(...) initializes x; see help(type(x)) for signature

add_line(*new_line*, *attributes=0*)

Add a text line to the screen buffer.

Parameters

- **new_line** –
- **attributes** –

add_string(*new_str*, *xpos=-1*, *ypos=-1*, *cr=False*, *fixed=False*, *attributes=0*)

Add a string to a position on the screen.

Parameters

- **new_str** –
- **xpos** –
- **ypos** –
- **cr** –
- **fixed** –
- **attributes** –

clear_buffer()

clear_screen()

Clear the ncurses screen.

cr()

Carriage return, go to the next line

draw_screen(*data=None*)

Draw the screen using the provided data TODO: ylimits, xlimits, proper line counts in the status

get_current_line()
Return the current y position of the internal screen buffer.

get_instruction_string()

on_keypress()
Handle key presses.

screen_setup()
Set up a curses screen object and associated options

set_current_line(*linenum*)
Set the current y position of the internal screen buffer.

Parameters **linenum** –

set_instruction_string(*new_string*)

set_xlimits(*xmin=-1*, *xmax=-1*)

set_ylimits(*ymin=-1*, *ymax=-1*)

scroll.**screen_teardown**()
Restore sensible options to the terminal upon exit

3.3.23 skarab_definitions

Description: Definitions for the Skarab Motherboard.

- Includes
- OPCCODES
- PORTS
- Register Masks
- Data structures

class skarab_definitions.**BigReadWishboneReq**(*start_address_high*, *start_address_low*, *number_of_reads*)

__init__(*start_address_high*, *start_address_low*, *number_of_reads*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**BigReadWishboneResp**(*command_id*, *seq_num*, *start_address_high*, *start_address_low*, *number_of_reads*, *read_data*)

__init__(*command_id*, *seq_num*, *start_address_high*, *start_address_low*, *number_of_reads*, *read_data*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**BigWriteWishboneReq**(*start_address_high*, *start_address_low*, *write_data*, *number_of_writes*)

__init__(*start_address_high*, *start_address_low*, *write_data*, *number_of_writes*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**BigWriteWishboneResp**(*command_id*, *seq_num*, *start_address_high*, *start_address_low*, *number_of_writes_done*, *error_status*, *padding*)

__init__(*command_id*, *seq_num*, *start_address_high*, *start_address_low*, *number_of_writes_done*, *error_status*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ClearFanControllerLogsReq**

__init__()
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ClearFanControllerLogsResp**(*command_id*, *seq_num*, *status*, *padding*)

__init__(*command_id*, *seq_num*, *status*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**Command**(*command_id*, *seq_num=None*)
The Command Packet structure for SKARAB communications

__init__(*command_id*, *seq_num=None*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

create_payload(*seq_num*)
Create payload for sending via UDP Packet to SKARAB

Returns string representation of data

static pack_two_bytes(*data*)

static unpack_two_bytes(*data*)

class skarab_definitions.**ConfigureMulticastReq**(*interface_id*, *fabric_multicast_ip_address_high*, *fabric_multicast_ip_address_low*, *fabric_multicast_ip_address_mask_high*, *fabric_multicast_ip_address_mask_low*)

__init__(*interface_id*, *fabric_multicast_ip_address_high*, *fabric_multicast_ip_address_low*, *fabric_multicast_ip_address_mask_high*, *fabric_multicast_ip_address_mask_low*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ConfigureMulticastResp**(*command_id*, *seq_num*, *interface_id*, *fabric_multicast_ip_address_high*, *fabric_multicast_ip_address_low*, *fabric_multicast_ip_address_mask_high*, *fabric_multicast_ip_address_mask_low*, *status*, *padding*)

__init__(*command_id*, *seq_num*, *interface_id*, *fabric_multicast_ip_address_high*, *fabric_multicast_ip_address_low*, *fabric_multicast_ip_address_mask_high*, *fabric_multicast_ip_address_mask_low*, *status*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**Current**

P12V2Current = 0

P12VCurrent = 1

P1V0Current = 7

P1V0MGTAVCCCurrent = 9

P1V2Current = 6

P1V2MGTAVTTCurrent = 10

P1V8Current = 5

P1V8MGTVCCAUXCurrent = 8

P2V5Current = 4

P3V3ConfigCurrent = 11

P3V3Current = 3

P5VCurrent = 2

class skarab_definitions.**DebugAddARPCacheEntryReq**(*interface_id*, *ip_address_lower_8_bits*, *mac_high*, *mac_mid*, *mac_low*)

__init__(*interface_id*, *ip_address_lower_8_bits*, *mac_high*, *mac_mid*, *mac_low*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**DebugAddARPCacheEntryResp**(*command_id*, *seq_num*, *interface_id*, *ip_address_lower_8_bits*, *mac_high*, *mac_mid*, *mac_low*, *padding*)

__init__(*command_id*, *seq_num*, *interface_id*, *ip_address_lower_8_bits*, *mac_high*, *mac_mid*, *mac_low*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**DebugConfigureEthernetReq**(*interface_id*, *fabric_mac_high*, *fabric_mac_mid*, *fabric_mac_low*, *fabric_port_address*, *gateway_arp_cache_address*, *fabric_ip_address_high*, *fabric_ip_address_low*, *fabric_multicast_ip_address_high*, *fabric_multicast_ip_address_low*, *fabric_multicast_ip_address_mask_high*, *fabric_multicast_ip_address_mask_low*, *enable_fabric_interface*)

__init__(*interface_id*, *fabric_mac_high*, *fabric_mac_mid*, *fabric_mac_low*, *fabric_port_address*, *gateway_arp_cache_address*, *fabric_ip_address_high*, *fabric_ip_address_low*, *fabric_multicast_ip_address_high*, *fabric_multicast_ip_address_low*, *fabric_multicast_ip_address_mask_high*, *fabric_multicast_ip_address_mask_low*, *enable_fabric_interface*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**DebugConfigureEthernetResp**(*command_id*, *seq_num*, *interface_id*, *fabric_mac_high*, *fabric_mac_mid*, *fabric_mac_low*, *fabric_port_address*, *gateway_arp_cache_address*, *fabric_ip_address_high*, *fabric_ip_address_low*, *fabric_multicast_ip_address_high*, *fabric_multicast_ip_address_low*, *fabric_multicast_ip_address_mask_high*, *fabric_multicast_ip_address_mask_low*, *enable_fabric_interface*, *padding*)

__init__(*command_id*, *seq_num*, *interface_id*, *fabric_mac_high*, *fabric_mac_mid*, *fabric_mac_low*, *fabric_port_address*, *gateway_arp_cache_address*, *fabric_ip_address_high*, *fabric_ip_address_low*, *fabric_multicast_ip_address_high*, *fabric_multicast_ip_address_low*, *fabric_multicast_ip_address_mask_high*, *fabric_multicast_ip_address_mask_low*, *enable_fabric_interface*, *padding*)

A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**DebugLoopbackTestReq**(*interface_id*, *test_data*)

__init__(*interface_id*, *test_data*)

A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**DebugLoopbackTestResp**(*command_id*, *seq_num*, *interface_id*, *test_data*, *valid*, *padding*)

__init__(*command_id*, *seq_num*, *interface_id*, *test_data*, *valid*, *padding*)

A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**EraseFlashBlockReq**(*block_address_high*, *block_address_low*)

__init__(*block_address_high*, *block_address_low*)

A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value

- **seq_num** – Integer value

class skarab_definitions.**EraseFlashBlockResp**(*command_id*, *seq_num*, *block_address_high*, *block_address_low*, *erase_success*, *padding*)

__init__(*command_id*, *seq_num*, *block_address_high*, *block_address_low*, *erase_success*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**EraseSpiSectorReq**(*sector_address_high*, *sector_address_low*)

__init__(*sector_address_high*, *sector_address_low*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**EraseSpiSectorResp**(*command_id*, *seq_num*, *sector_address_high*, *sector_address_low*, *erase_success*, *padding*)

__init__(*command_id*, *seq_num*, *sector_address_high*, *sector_address_low*, *erase_success*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**Fan**

FPGAFan = 4

LeftBackFan = 2

LeftFrontFan = 0

LeftMiddleFan = 1

RightBackFan = 3

class skarab_definitions.**GetCurrentLogsReq**

__init__()
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**GetCurrentLogsResp**(*command_id*, *seq_num*, *current_mon_logs*, *status*)

__init__(*command_id*, *seq_num*, *current_mon_logs*, *status*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**GetDHCPMonitorTimeoutReq**

__init__()
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**GetDHCPMonitorTimeoutResp**(*command_id*, *seq_num*, *dhcp_monitor_timeout*, *padding*)

__init__(*command_id*, *seq_num*, *dhcp_monitor_timeout*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**GetEmbeddedSoftwareVersionReq**

__init__()
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**GetEmbeddedSoftwareVersionResp**(*command_id*, *seq_num*, *version_major*, *version_minor*, *version_patch*, *qsfp_bootloader_version_major*, *qsfp_bootloader_version_minor*, *padding*)

__init__(*command_id*, *seq_num*, *version_major*, *version_minor*, *version_patch*, *qsfp_bootloader_version_major*, *qsfp_bootloader_version_minor*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**GetFanControllerLogsReq**

__init__()
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**GetFanControllerLogsResp**(*command_id*, *seq_num*, *fan_cont_mon_logs*, *status*, *padding*)

__init__(*command_id*, *seq_num*, *fan_cont_mon_logs*, *status*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**GetSensorDataReq**

__init__()
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**GetSensorDataResp**(*command_id*, *seq_num*, *sensor_data*, *status*)

__init__(*command_id*, *seq_num*, *sensor_data*, *status*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**GetVoltageLogsReq**

__init__()
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**GetVoltageLogsResp**(*command_id*, *seq_num*, *voltage_mon_logs*, *status*)

__init__(*command_id*, *seq_num*, *voltage_mon_logs*, *status*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**Mezzanine**

Mezzanine0 = 0

Mezzanine1 = 1

Mezzanine2 = 2

Mezzanine3 = 3

class skarab_definitions.**MulticastLeaveGroupReq**(*link_id*)

__init__(*link_id*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**MulticastLeaveGroupResp**(*command_id*, *seq_num*, *link_id*, *success*, *padding*)

__init__(*command_id*, *seq_num*, *link_id*, *success*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**OneWireDS2433ReadMemReq**(*device_rom*, *skip_rom_address*, *num_bytes*, *target_address_1*, *target_address_2*, *one_wire_port*)

__init__(*device_rom*, *skip_rom_address*, *num_bytes*, *target_address_1*, *target_address_2*, *one_wire_port*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**OneWireDS2433ReadMemResp**(*command_id*, *seq_num*, *device_rom*, *skip_rom_address*, *read_bytes*, *num_bytes*, *target_address_1*, *target_address_2*, *one_wire_port*, *read_success*, *padding*)

__init__(*command_id*, *seq_num*, *device_rom*, *skip_rom_address*, *read_bytes*, *num_bytes*, *target_address_1*, *target_address_2*, *one_wire_port*, *read_success*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**OneWireDS2433WriteMemReq**(*device_rom*, *skip_rom_address*, *write_bytes*, *num_bytes*, *target_address_1*, *target_address_2*, *one_wire_port*)

__init__(*device_rom*, *skip_rom_address*, *write_bytes*, *num_bytes*, *target_address_1*, *target_address_2*, *one_wire_port*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**OneWireDS2433WriteMemResp**(*command_id*, *seq_num*, *device_rom*, *skip_rom_address*, *write_bytes*, *num_bytes*, *target_address_1*, *target_address_2*, *one_wire_port*, *write_success*, *padding*)

__init__(*command_id*, *seq_num*, *device_rom*, *skip_rom_address*, *write_bytes*, *num_bytes*, *target_address_1*, *target_address_2*, *one_wire_port*, *write_success*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**OneWireReadROMReq**(*one_wire_port*)

__init__(*one_wire_port*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**OneWireReadROMResp**(*command_id*, *seq_num*, *one_wire_port*, *rom*, *read_success*, *padding*)

__init__(*command_id*, *seq_num*, *one_wire_port*, *rom*, *read_success*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**PMBusReadI2CBytesReq**(*i2c_interface_id*, *slave_address*, *command_code*, *read_bytes*, *num_bytes*)

__init__(*i2c_interface_id*, *slave_address*, *command_code*, *read_bytes*, *num_bytes*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**PMBusReadI2CBytesResp**(*command_id*, *seq_num*, *i2c_interface_id*, *slave_address*, *command_code*, *read_bytes*, *num_bytes*, *read_success*)

__init__(*command_id*, *seq_num*, *i2c_interface_id*, *slave_address*, *command_code*, *read_bytes*, *num_bytes*, *read_success*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**ProgramFlashWordsReq**(*address_high*, *address_low*, *total_num_words*, *packet_num_words*, *do_buffered_programming*, *start_program*, *finish_program*, *write_words*)

__init__(*address_high*, *address_low*, *total_num_words*, *packet_num_words*, *do_buffered_programming*, *start_program*, *finish_program*, *write_words*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ProgramFlashWordsResp**(*command_id*, *seq_num*, *address_high*, *address_low*, *total_num_words*, *packet_num_words*, *do_buffered_programming*, *start_program*, *finish_program*, *program_success*, *padding*)

__init__(*command_id*, *seq_num*, *address_high*, *address_low*, *total_num_words*, *packet_num_words*, *do_buffered_programming*, *start_program*, *finish_program*, *program_success*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value

- **seq_num** – Integer value

class skarab_definitions.**ProgramSpiPageReq**(*address_high*, *address_low*, *num_bytes*, *write_bytes*)

__init__(*address_high*, *address_low*, *num_bytes*, *write_bytes*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ProgramSpiPageResp**(*command_id*, *seq_num*, *address_high*, *address_low*, *num_bytes*, *verify_bytes*, *program_spi_page_success*, *padding*)

__init__(*command_id*, *seq_num*, *address_high*, *address_low*, *num_bytes*, *verify_bytes*, *program_spi_page_success*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**QSFPResetAndProgramReq**(*reset*, *program*)

__init__(*reset*, *program*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**QSFPResetAndProgramResp**(*command_id*, *seq_num*, *reset*, *program*, *padding*)

__init__(*command_id*, *seq_num*, *reset*, *program*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ReadFlashWordsReq**(*address_high*, *address_low*, *num_words*)

__init__(*address_high*, *address_low*, *num_words*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ReadFlashWordsResp**(*command_id*, *seq_num*, *address_high*, *address_low*, *num_words*, *read_words*, *padding*)

__init__(*command_id*, *seq_num*, *address_high*, *address_low*, *num_words*, *read_words*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**ReadHMCI2CReq**(*interface_id*, *slave_address*, *read_address*)

__init__(*interface_id*, *slave_address*, *read_address*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ReadHMCI2CResp**(*command_id*, *seq_num*, *interface_id*, *slave_address*, *read_address*, *read_bytes*, *read_success*, *padding*)

__init__(*command_id*, *seq_num*, *interface_id*, *slave_address*, *read_address*, *read_bytes*, *read_success*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**ReadI2CReq**(*i2c_interface_id*, *slave_address*, *num_bytes*)

__init__(*i2c_interface_id*, *slave_address*, *num_bytes*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ReadI2CResp**(*command_id*, *seq_num*, *i2c_interface_id*, *slave_address*, *num_bytes*, *read_bytes*, *read_success*, *padding*)

__init__(*command_id*, *seq_num*, *i2c_interface_id*, *slave_address*, *num_bytes*, *read_bytes*, *read_success*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value

- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**ReadRegReq**(*board_reg*, *reg_addr*)

__init__(*board_reg*, *reg_addr*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ReadRegResp**(*command_id*, *seq_num*, *board_reg*, *reg_addr*, *reg_data_high*, *reg_data_low*, *padding*)

__init__(*command_id*, *seq_num*, *board_reg*, *reg_addr*, *reg_data_high*, *reg_data_low*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ReadSpiPageReq**(*address_high*, *address_low*, *num_bytes*)

__init__(*address_high*, *address_low*, *num_bytes*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ReadSpiPageResp**(*command_id*, *seq_num*, *address_high*, *address_low*, *num_bytes*, *read_bytes*, *read_spi_page_success*, *padding*)

__init__(*command_id*, *seq_num*, *address_high*, *address_low*, *num_bytes*, *read_bytes*, *read_spi_page_success*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**ReadWishboneReq**(*address_high*, *address_low*)

__init__(*address_high*, *address_low*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ReadWishboneResp**(*command_id*, *seq_num*, *address_high*, *address_low*, *read_data_high*, *read_data_low*, *error_status*, *padding*)

__init__(*command_id*, *seq_num*, *address_high*, *address_low*, *read_data_high*, *read_data_low*, *error_status*, *padding*)

A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ResetDHCPStateMachineReq**(*link_id*)

__init__(*link_id*)

A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**ResetDHCPStateMachineResp**(*command_id*, *seq_num*, *link_id*, *reset_error*, *padding*)

__init__(*command_id*, *seq_num*, *link_id*, *reset_error*, *padding*)

A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**Response**(*command_id*, *seq_num=None*)

classmethod from_raw_data(*rawdata*, *number_of_words*, *pad_words*)

Unpack the rawdata and return a Response object :param rawdata: :param number_of_words: :param pad_words: :return:

static unpack_preprocess(*rawdata*, *number_of_words*, *pad_words*)

static unpack_process(*unpacked_data*)

class skarab_definitions.**SdramProgramReq**(*first_packet*, *last_packet*, *write_words*)

__init__(*first_packet*, *last_packet*, *write_words*)

A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**SdramProgramWishboneReq**(*chunk_id*, *num_total_chunks*, *image_data*)

__init__(*chunk_id*, *num_total_chunks*, *image_data*)

A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**SdramProgramWishboneResp**(*command_id*, *seq_num*, *chunk_id*, *ack*, *padding*)

__init__(*command_id*, *seq_num*, *chunk_id*, *ack*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**SdramReconfigureReq**(*output_mode*, *clear_sdram*, *finished_writing*, *about_to_boot*, *do_reboot*, *reset_sdram_read_address*, *clear_ethernet_stats*, *enable_debug_sdram_read_mode*, *do_sdram_async_read*, *do_continuity_test*, *continuity_test_output_low*, *continuity_test_output_high*)

__init__(*output_mode*, *clear_sdram*, *finished_writing*, *about_to_boot*, *do_reboot*, *reset_sdram_read_address*, *clear_ethernet_stats*, *enable_debug_sdram_read_mode*, *do_sdram_async_read*, *do_continuity_test*, *continuity_test_output_low*, *continuity_test_output_high*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**SdramReconfigureResp**(*command_id*, *seq_num*, *output_mode*, *clear_sdram*, *finished_writing*, *about_to_boot*, *do_reboot*, *reset_sdram_read_address*, *clear_ethernet_stats*, *enable_debug_sdram_read_mode*, *do_sdram_async_read*, *num_ethernet_frames*, *num_ethernet_bad_frames*, *num_ethernet_overload_frames*, *sdram_async_read_data_high*, *sdram_async_read_data_low*, *do_continuity_test*, *continuity_test_output_low*, *continuity_test_output_high*)

__init__(*command_id*, *seq_num*, *output_mode*, *clear_sdram*, *finished_writing*, *about_to_boot*, *do_reboot*, *reset_sdram_read_address*, *clear_ethernet_stats*, *enable_debug_sdram_read_mode*, *do_sdram_async_read*, *num_ethernet_frames*, *num_ethernet_bad_frames*, *num_ethernet_overload_frames*, *sdram_async_read_data_high*, *sdram_async_read_data_low*, *do_continuity_test*, *continuity_test_output_low*, *continuity_test_output_high*)

A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**SetFanSpeedReq**(*fan_page*, *pwm_setting*)

__init__(*fan_page*, *pwm_setting*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**SetFanSpeedResp**(*command_id*, *seq_num*, *fan_speed_pwm*, *fan_speed_rpm*, *padding*)

__init__(*command_id*, *seq_num*, *fan_speed_pwm*, *fan_speed_rpm*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

exception skarab_definitions.**SkarabInvalidBitstream**

exception skarab_definitions.**SkarabProgrammingError**

class skarab_definitions.**Tempsensor**

FPGATemp = 2

FanContTemp = 7

InletTemp = 0

Mezzanine0Temp = 3

Mezzanine1Temp = 4

Mezzanine2Temp = 5

Mezzanine3Temp = 6

OutletTemp = 1

class skarab_definitions.**Voltage**

P12V2Voltage = 0

P12VVoltage = 1

P1V0MGTAVCCVoltage = 9

P1V0Voltage = 7

P1V2MGTAVTTVoltage = 10

P1V2Voltage = 6

P1V8MGTVCCAUXVoltage = 8

P1V8Voltage = 5

P2V5Voltage = 4

P3V3ConfigVoltage = 11

P3V3Voltage = 3

P5VVoltage = 2

class skarab_definitions.**WriteHMCI2CReq**(*interface_id*, *slave_address*, *write_address*, *write_data*)

__init__(*interface_id*, *slave_address*, *write_address*, *write_data*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**WriteHMCI2CResp**(*command_id*, *seq_num*, *interface_id*, *slave_address*, *write_address*, *write_data*, *write_success*, *padding*)

__init__(*command_id*, *seq_num*, *interface_id*, *slave_address*, *write_address*, *write_data*, *write_success*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**WriteI2CReq**(*i2c_interface_id*, *slave_address*, *num_bytes*, *write_bytes*)

__init__(*i2c_interface_id*, *slave_address*, *num_bytes*, *write_bytes*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**WriteI2CResp**(*command_id*, *seq_num*, *i2c_interface_id*, *slave_address*, *num_bytes*, *write_bytes*, *write_success*)

__init__(*command_id*, *seq_num*, *i2c_interface_id*, *slave_address*, *num_bytes*, *write_bytes*, *write_success*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

static unpack_process(*unpacked_data*)

class skarab_definitions.**WriteRegReq**(*board_reg*, *reg_addr*, *reg_data_high*, *reg_data_low*)

__init__(*board_reg*, *reg_addr*, *reg_data_high*, *reg_data_low*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**WriteRegResp**(*command_id*, *seq_num*, *board_reg*, *reg_addr*, *reg_data_high*, *reg_data_low*, *padding*)

__init__(*command_id*, *seq_num*, *board_reg*, *reg_addr*, *reg_data_high*, *reg_data_low*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**WriteWishboneReq**(*address_high*, *address_low*, *write_data_high*, *write_data_low*)

__init__(*address_high*, *address_low*, *write_data_high*, *write_data_low*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

class skarab_definitions.**WriteWishboneResp**(*command_id*, *seq_num*, *address_high*, *address_low*, *write_data_high*, *write_data_low*, *error_status*, *padding*)

__init__(*command_id*, *seq_num*, *address_high*, *address_low*, *write_data_high*, *write_data_low*, *error_status*, *padding*)
A command will always have the following parameters/properties

Parameters

- **command_id** – Integer value
- **seq_num** – Integer value

3.3.24 skarab_fileops

3.3.25 snap

class snap.**Snap**(*parent*, *name*, *width_bits*, *address*, *length_bytes*, *device_info=None*)
Snap blocks are triggered/controlled blocks of RAM on FPGAs.

__init__(*parent*, *name*, *width_bits*, *address*, *length_bytes*, *device_info=None*)
A chunk of memory on a device.

Parameters

- **name** – a name for this memory

- **width_bits** – the width, in BITS, PER WORD
- **address** – the start address in device memory
- **length_bytes** – length, in BYTES

e.g. a Register has width_bits=32, length_bytes=4

e.g.2. a Snapblock could have width_bits=128, length_bytes=32768

arm(*man_trig=False*, *man_valid=False*, *offset=-1*, *circular_capture=False*)
Arm the snapshot block.

Parameters

- **man_trig** (*bool*) –
- **man_valid** (*bool*) –
- **offset** (*int*) –
- **circular_capture** (*bool*) –

classmethod from_device_info(*parent*, *device_name*, *device_info*, *memorymap_dict*, ***kwargs*)
Process device info and the memory map to get all necessary info and return a Snap instance.

Parameters

- **parent** – the Casperfpga on which this snap is found
- **device_name** – the unique device name
- **device_info** – information about this device
- **memorymap_dict** – a dictionary containing the device memory map

Returns a Snap object

static packetise_snapdata(*data*, *eof_key='eof'*, *packet_length=-1*, *dv_key=None*)
Use the given EOF key to packetise a dictionary of snap data

Parameters

- **data** – a dictionary containing snap block data
- **eof_key** – the key used to identify the packet boundaries - the eof comes on the LAST VALID word in a packet
- **packet_length** – check the length of the packets against this as they are created (in 64-bit words)
- **dv_key** – the key used to identify which data samples are valid

Returns a list of packets

post_create_update(*raw_system_info*)
Update the device with information not available at creation.

Parameters **raw_system_info** – dictionary of device information

print_snap(*limit_lines=-1*, ***kwargs*)
Read and print(a snap block.)

Parameters

- **limit_lines** – limit the number of lines to print
- **offset** – trigger offset

- **man_valid** – force valid to be true
- **man_trig** – force a trigger now
- **circular_capture** – enable circular capture
- **timeout** – time out after this many seconds
- **read_nowait** – do not wait for the snap to finish reading

read(***kwargs*)
Override Memory.read to handle the extra value register.

Parameters

- **offset** – trigger offset
- **man_valid** – force valid to be true
- **man_trig** – force a trigger now
- **circular_capture** – enable circular capture
- **timeout** – time out after this many seconds
- **read_nowait** – do not wait for the snap to finish reading

read_raw(***kwargs*)
Read snap data from the memory device.

update_from_bitsnap(*info*)
Update this device with information from a bitsnap container.

Parameters **info** – device information dictionary containing Simulink block information

3.3.26 snapadc

class snapadc.**SnapAdc**(*parent*, *device_name*, *device_info*, *initialise=False*)

A_WB_W_3WIRE = 0

A_WB_W_CTRL = 1

A_WB_W_DELAY_STROBE_H = 3

A_WB_W_DELAY_STROBE_L = 2

ERROR_FRAME = 4

ERROR_LINE = 3

ERROR_LMX = 1

ERROR_MMCM = 2

ERROR_RAMP = 5

M_WB_W_DELAY_TAP = 31

M_WB_W_DEMUX_MODE = 50331648

M_WB_W_DEMUX_WRITE = 67108864

M_WB_W_ISERDES_BITSLIP_CHIP_SEL = 65280

M_WB_W_ISERDES_BITSLIP_LANE_SEL = 224

M_WB_W_RESET = 1048576

M_WB_W_SNAP_REQ = 65536

SUCCESS = 0

WB_DICT = [{'ADC16_LOCKED': 50331648, 'ADC16_ADC3WIRE_REG': 65535, 'G_ROACH2_REV': 196

__init__(*parent*, *device_name*, *device_info*, *initialise=False*)

Initialise SnapAdc Object :param parent: Parent object creating the SnapAdc Object :type parent: casperfpga.CasperFpga

Parameters

- **device_name** (*str*) – Name of SnapAdc Object
- **device_info** (*dict*) –
- **intialise** – Trigger ADC SerDes calibration.

Returns None

example device_info = {'adc_resolution': '8', 'sample_rate': '200', 'snap_inputs': '12', 'tag': 'xps:snap_adc'}

align_frame_clock()

Align the frame clock with data frame

align_line_clock(*mode='dual_pat'*)

Align the rising edge of line clock with data eye

And return the tap settings being using

bitslip(*chipSel=None*, *laneSel=None*)

Reorder the parallelize data for word-alignment purpose

Reorder the parallelized data by asserting a itslip command to the bitslip submodule of a ISERDES primitive. Each bitslip command left shift the parallelized data by one bit.

HMCAD1511/HMCAD1520 lane correspondence lane number lane name in ADC datasheet 0 1a 1 1b 2 2a 3 2b 4 3a 5 3b 6 4a 7 4b

E.g. .. code-block:: python

bitslip() # left shift all lanes of all ADCs bitslip(0) # shift all lanes of the 1st ADC bitslip(0,3) # shift the 4th lane of the 1st ADC bitslip([0,1],[3,4]) # shift the 4th and 5th lanes of the 1st

and the 2nd ADCs

calibrate_adc_offset()

calibration_adc_gain()

clksw = None

controller = None

curDelay = None

decide_delay(*data*)

Decide and return proper setting for delay tap

Find the tap setting that has the largest margin of error, i.e. the biggest distance to borders (tap=0 and tap=31) and rows with non-zero deviations/mismatches. The parameter data is a 32 by n numpy array, in which the 1st dimension index indicates the delay tap setting

delay(*tap*, *chipSel=None*, *laneSel=None*)
Delay the serial data from ADC LVDS links

Delay the serial data by Xilinx IDELAY primitives E.g.

delay(0) # Set all delay tap of IDELAY to 0 delay(4, 1, 7) # set delay on the 8th lane of the 2nd ADC to 4 delay(31, [0,1,2], [0,1,2,3,4,5,6,7])

Set all delay taps (in SNAP 1 case) to 31

classmethod from_device_info(*parent*, *device_name*, *device_info*, *initialise=False*, ***kwargs*)
Process device info and the memory map to get all the necessary info and return a SNAP ADC instance. :param parent: The parent device, normally a casperfpga instance :param device_name: :param device_info: :param initialise: :param kwargs: :return:

get_reg_id(*name*)

get_register(*rid=None*)

get_word(*name*)

init(*sample_rate=None*, *num_channel=None*)
Get SNAP ADCs into working condition

Supported frequency range: 60MHz ~ 1000MHz. Set resolution to None to let init() automatically decide the best resolution.

A run of init() takes approximatly 20 seconds, involving the following actions:

1. configuring frequency synthesizer LMX2581
2. configuring clock source switch HMC922
3. configuring ADCs HMCAD1511 (support HMCAD1520 in future)
4. configuring IDELAYE2 and ISERDESE2 inside of FPGA
5. Testing under dual pattern and ramp mode

E.g.

init(1000,1) 1 channel mode, 1Gsps, 8bit, since 1Gsps is only available in 8bit mode

init(320,2) 2 channel mode, 320Msps, 8bit for HMCAD1511, or 12bit for HMCAD1520

init(160,4,8) 4 channel mode, 160Msps, 8bit resolution

interleave(*data*, *mode*)
Reorder the data according to the interleaving mode

E.g. .. code-block:: python

data = numpy.arange(1024).reshape(-1,8) interleave(data, 1) # return a one-column numpy array interleave(data, 2) # return a two-column numpy array interleave(data, 4) # return a four-column numpy array

ram = None

read_ram(*ram=None*, *signed=True*)
Read RAM(s) and return the 1024-sample data

E.g. readRAM() # read all RAMs, return a list of arrays readRAM(1) # read the 2nd RAMs, return a 128X8 array readRAM([0,1]) # read 2 RAMs, return two arrays readRAM(signed=False) # return a list of arrays in unsiged format

reset()
Reset all adc16_interface logics inside FPGA

resolution = 8

select_adc(*chipSel=None*)
Select one or multiple ADCs

Select the ADC(s) to be configured. ADCs are numbered by 0, 1, 2... E.g.

selectADC(0) # select the 1st ADC selectADC([0,1]) # select two ADCs selectADC() # select all ADCs

set_demux(*numChannel=1*)

when mode==0: numChannel=4 data = data[:,[0,4,1,5,2,6,3,7]]

when mode==1: numChannel=2 data = data[:,[0,1,4,5,2,3,6,7]]

when mode==2: numChannel=1 data = data[:,[0,1,2,3,4,5,6,7]]

set_gain(*gains*, *use_linear_step=False*, *fine_gains=None*, *fgain_cfg=False*)
Set the coarse gain of the ADC channels

Args: gains (list): List of gains, e.g. [1, 2, 3, 4] use_linear_step (bool): Defaults to use dB steps for values. fine_gains (list): Fine gain values to set fgain_cfg (bool): If fine gains are to be used, set this to True

Notes: Coarse gain control (parameters in dB). Input gain must be a list of integers. Coarse gain range for HMCAD1511: 0dB ~ 12dB

E.g. cGain([1,5,9,12]) # Quad channel mode in dB step cGain([32,50],use_linear_step=True) # Dual channel mode in x step cGain([10], fgain_cfg=True) # Single channel mode in dB

step, with fine gain enabled

Coarse gain options when by default use_linear_step=False: 0 dB, 1 dB, 2 dB, 3 dB, 4 dB, 5 dB, 6 dB, 7 dB, 8 dB, 9 dB, 10 dB, 11 dB and 12 dB

Coarse gain options when use_linear_step=True: 1x, 1.25x, 2x, 2.5x, 4x, 5x, 8x, 10x, 12.5x, 16x, 20x, 25x, 32x, 50x

TODO: Test + improve support for fine gain control

snapshot()
Save 1024 consecutive samples of each ADC into its corresponding bram

synth = None

test_patterns(*chipSel=None*, *taps=None*, *mode='std'*, *pattern1=None*, *pattern2=None*)
Return a list of std/err for a given tap or a list of taps

Return the lane-wise standard deviation/error of the data under a given tap setting or a list of tap settings. By default, mode='std', taps=range(32). 'err' mode with single test pattern check data against the given pattern, while 'err' mode with dual test patterns guess the counts of the mismatches. 'guess' because both patterns could come up at first. This method always returns the smaller counts. 'ramp' mode guess the total number of incorrect data. This is implemented based on the assumption that in most cases, $cur = $pre + 1. When using 'ramp' mode, taps=None

E.g. .. code-block:: python

testPatterns(taps=True) # Return lane-wise std of all ADCs, taps=range(32) testPatterns(0,taps=range(32))

Return lane-wise std of the 1st ADC

testPatterns([0,1],2) # Return lane-wise std of the first two ADCs # with tap = 2

testPatterns(1, taps=[0,2,3], mode='std') # Return lane-wise stds of the 2nd ADC with # three different tap settings

testPatterns(2, mode='err', pattern1=0b10101010) # Check the actual data against the given test # pattern without changing current delay tap # setting and return lane-wise error counts of # the 3rd ADC,

testPatterns(2, mode='err', pattern1=0b10101010, pattern2=0b01010101) # Check the actual data against the given alternate # test pattern without changing current delay tap # setting and return lane-wise error counts of # the 3rd ADC,

testPatterns(mode='ramp') # Check all ADCs under ramp mode

3.3.27 spead

SPEAD operations - unpack and use spead data, usually from Snap blocks.

class spead.**SpeadPacket**(*headers=None*, *data=None*)

A Spead packet. Headers and data.

exception SpeadPacketError

__init__(*headers=None*, *data=None*)

Create a new SpeadPacket object

static decode_headers(*data*, *expected_version=None*, *expected_flavour=None*, *expected_hdrs=None*)

Decode the SPEAD headers given some packet data.

Parameters

- **data** – a list of packet data
- **expected_version** – an explicit version, if required
- **expected_flavour** – an explicit flavour, if required
- **expected_hdrs** – explicit number of hdrs, if required

static decode_item_pointer(*header64*, *id_bits*, *address_bits*)

Decode a 64-bit header word in the id and data/pointer portions

Parameters

- **header64** – the 64-bit word
- **id_bits** – how many bits are used for the ID
- **address_bits** – how many bits are used for the data/pointer

Returns a tuple of the ID and data/pointer

static decode_spead_magic_word(*word64*, *required_version=None*, *required_flavour=None*, *required_numheaders=None*)

Decode a 64-bit word as a SPEAD header.

Parameters

- **word64** – A 64-bit word
- **required_version** – the specific SPEAD version required, an integer
- **required_flavour** – the specific SPEAD flavour required as a string, e.g. '64,48'

- **required_numheaders** – the number of headers (NOT incl. the magic number) expected, an integer

static find_spead_header(*data64*, *expected_version=4*, *expected_flavour='64'*, *48'*)
Find a SPEAD header in the given list of 64-bit data

Parameters

- **data64** – a list of data
- **expected_version** – the version wanted
- **expected_flavour** – the flavour wanted

Returns None if no header is found, else the index and the contents of the header as a tuple

classmethod from_data(*data*, *expected_version=None*, *expected_flavour=None*, *expected_hdrs=None*, *expected_length=None*)
Create a SpeadPacket from a list of 64-bit data words Assumes the list of data starts at the SPEAD magic word.

get_strings(*headers_only=False*, *hex_nums=False*)
Get a list of the string representation of this packet.

print_packet(*headers_only=False*, *hex_nums=False*)
Print a representation of the packet.

class spead.**SpeadProcessor**(*version=4*, *flavour='64*, *48'*, *packet_length=None*, *num_headers=None*)
Set up a SPEAD processor with version, flavour, etc. Then call methods to process data.

__init__(*version=4*, *flavour='64*, *48'*, *packet_length=None*, *num_headers=None*)
Create a SpeadProcessor

:param version :param flavour :param packet_length :param num_headers

process_data(*data_packets*)
Create SpeadPacket objects from a list of data packets.

3.3.28 synth

class synth.**LMX2581**(*interface*, *controller_name*, *fosc=10*)
LMX2581 Frequency Synthesizer

CMD08 = 545119224

CMD09 = 63422521

CMD10 = 553668810

DICTS = [{'PLL_NUM_L': 65520, 'FRAC_DITHER': 1610612736, 'ID': 2147483648, 'PLL_N': 26

MASK_DIAG = {'BUFEN_PINSTATE': 2048, 'CAL_RUNNING': 2097152, 'CE_PINSTATE': 4096, 'DLD

__init__(*interface*, *controller_name*, *fosc=10*)
x.__init__(...) initializes x; see help(type(x)) for signature

getDiagnoses(*name=None*)

getRegId(*name*)

getRegister(*rid=None*)

getWord(*name*)

get_osc_values(*synth_mhz*, *ref_signal*)
This function gets oscillator values

init()

loadCfgFromFile(*filename*)

outputPower(*p=15*)

powerOff()

powerOn()

read(*addr*)

reset()

setFreq(*synth_mhz*)

setWord(*value*, *name*)

write(*data*, *addr=None*, *mask=None*)

3.3.29 tengbe

class tengbe.**TenGbe**(*parent*, *name*, *address*, *length_bytes*, *device_info=None*)
To do with the CASPER ten GBE yellow block implemented on FPGAs, and interfaced-to via KATCP memory reads/writes.

__init__(*parent*, *name*, *address*, *length_bytes*, *device_info=None*)

Parameters

- **parent** – Parent object who owns this TenGbe instance
- **name** – Unique name of the instance
- **address** –
- **length_bytes** –
- **device_info** – Information about this device

configure_core()
Setup the interface by writing to the fabric directly, bypassing tap. :param self: :return:

dhcp_start()
Configure this interface, then start a DHCP client on ALL interfaces.

fabric_disable()
Enable the core fabric

fabric_enable()
Enable the core fabric

fabric_soft_reset_toggle()
Toggle the fabric soft reset

get_arp_details(*N=256*)
Get ARP details from this interface.

get_cpu_details(*port_dump=None*)
Read details of the CPU buffers.

Parameters **port_dump** –

get_gbe_core_details(*read_arp=False*, *read_cpu=False*, *read_multicast=False*)
Get 10GbE core details.

Parameters

- **(bool)** (*read_multicast*) – Get ARP table details (default False)
- **(bool)** – Get CPU details (default False)
- **(bool)** – Get multicast address table (default False)

Returns dictionary of core details (IP address, subnet mask, MAC address, port, etc).

ip_address

mac

multicast_receive(*ip_str*, *group_size*)
Send a request to KATCP to have this tap instance send a multicast group join request.

Parameters

- **ip_str** – A dotted decimal string representation of the base mcast IP address.
- **group_size** – An integer for how many mcast addresses from base to respond to.

multicast_remove(*ip_str*)
Send a request to be removed from a multicast group.

Parameters **ip_str** – A dotted decimal string representation of the base mcast IP address.

port

post_create_update(*raw_device_info*)
Update the device with information not available at creation.

Parameters **raw_device_info** – info about this block that may be useful

read_rxsnap()
Read the RX snapshot embedded in this TenGBE yellow block

read_txsnap()
Read the TX snapshot embedded in this TenGBE yellow block

set_arp_table(*macs*)
Set the ARP table with a list of MAC addresses. The list, *macs*, is passed such that the zeroth element is the MAC address of the device with IP XXX.XXX.XXX.0, and element N is the MAC address of the device with IP XXX.XXX.XXX.N

tap_arp_reload()
Have the tap driver reload its ARP table right now.

tap_info()
Get info on the tap instance running on this interface.

tap_running()
Determine if an instance if tap is already running on for this ten GBE interface.

tap_start(*restart=False*)
Program a 10GbE device and start the TAP driver.

Parameters **restart** – stop before starting

tap_stop()
Stop a TAP driver.

tengbe.**read_memory_map_definition**(*filename*)
Read memory map definition from text file.

Returns a python dictionary:

{REGISTER_NAME1: {'offset': offset, 'size': size, 'rwflag': rwflag}, REGISTER_NAME2: {'offset': offset, 'size': size, 'rwflag': rwflag} ... }

Notes: Used by TenGbe.configure_core() to write to mmap.

3.3.30 termcolors

termcolors.py – Taken from corr

termcolors.**colorize**(*text=''*, *opts=()*, ***kwargs*)
Returns your text, enclosed in ANSI graphics codes.

Depends on the keyword arguments 'fg' and 'bg', and the contents of the opts tuple/list.

Returns the RESET code if no parameters are given.

Valid colors: 'black', 'red', 'green', 'yellow', 'blue', 'magenta', 'cyan', 'white'

Valid options: 'bold' 'underscore' 'blink' 'reverse' 'conceal' 'noreset' - string will not be auto-terminated with the RESET code

Examples:

```
colorize('hello', fg='red', bg='blue', opts=('blink',))
colorize()
colorize('goodbye', opts=('underscore',))
print colorize('first line', fg='red', opts=('noreset',))
print 'this should be red too'
print colorize('and so should this')
print 'this should not be red'
```

3.3.31 transport

class transport.**Transport**(***kwargs*)
The actual network transport of data for a CasperFpga object.

__init__(***kwargs*)
Initialise the CasperFpga object

Parameters **host** –

blindwrite(*device_name*, *data*, *offset=0*)
Write binary data to *device_name*, starting at *offset* bytes from *device_name*'s base address..

Parameters

- **device_name** (*String*) – Name of device to be read
- **data** (*Big-endian binary string*) – Data to write
- **offset** (*Integer*) – Offset from which to begin write, in bytes

Returns None

connect(*timeout=None*)

Parameters **timeout** –

deprogram()
Deprogram the FPGA connected by this transport

disconnect()

get_system_information_from_transport()

is_connected()
Is the transport layer connected to the platform?

Returns True or False

is_running()
Is the FPGA programmed and running?

Returns True or False

listdev()
Get a list of the memory bus items in this design.

Returns a list of memory devices

ping()
Use the 'watchdog' request to ping the FPGA host.

Returns True or False

post_get_system_information()
Cleanup run after get_system_information

read(*device_name*, *size*, *offset=0*)
Read *size* bytes from register *device_name*. Start reading from *offset* bytes from *device_name*'s base address. Return the read data as a big-endian binary string.

Parameters

- **device_name** (*String*) – Name of device to be read
- **size** (*Integer*) – Number of bytes to read
- **offset** (*Integer*) – Offset from which to begin read, in bytes

Returns Big-endian binary string

set_igmp_version(*version*)

Parameters **version** –

test_connection()
Write to and read from the scratchpad to test the connection to the FPGA - i.e. Is the casper FPGA connected?

Returns Boolean - True/False - Success/Fail

upload_to_flash(*binary_file*, *port=-1*, *force_upload=False*, *timeout=30*, *wait_complete=True*)
Upload the provided binary file to the flash filesystem.

Parameters

- **binary_file** – filename of the binary file to upload
- **port** – host-side port, -1 means a random port will be used
- **force_upload** – upload the binary even if it already exists on the host
- **timeout** – upload timeout, in seconds
- **wait_complete** – wait for the upload to complete, or just kick it off

upload_to_ram_and_program(*filename*, *port=-1*, *timeout=10*, *wait_complete=True*, *skip_verification=False*)
Upload an FPG file to RAM and then program the FPGA.

- Implemented in the child

Parameters

- **filename** – the file to upload
- **port** – the port to use on the rx end, -1 means a random port
- **timeout** – how long to wait, seconds
- **wait_complete** – wait for the transaction to complete, return after upload if False
- **skip_verification** – don't verify the image after uploading it

3.3.32 transport_dummy

class transport_dummy.**DummyTransport**(***kwargs*)
A dummy transport for testing

__init__(***kwargs*)
Make a Dummy Transport

Parameters **host** – IP Address should be 127.0.0.1 for a Dummy

blindwrite(*device_name*, *data*, *offset=0*)

Parameters

- **device_name** –
- **data** –
- **offset** –

connect(*timeout=None*)

Parameters **timeout** –

deprogram()
Deprogram the FPGA connected by this transport

disconnect()

Returns

get_system_information_from_transport()

is_connected()

is_running()
Is the FPGA programmed and running?

Returns True or False

listdev()
Get a list of the memory bus items in this design.

Returns a list of memory devices

static multicast_receive(*gbename*, *ip*, *mask*)

Parameters

- **gbename** –
- **ip** –
- **mask** –

ping()
Use the 'watchdog' request to ping the FPGA host.

Returns True or False

post_get_system_information()
Cleanup run after get_system_information

read(*device_name*, *size*, *offset=0*)

Parameters

- **device_name** –
- **size** –
- **offset** –

read_wishbone(*wb_address*)
Used to perform low level wishbone read from a Wishbone slave.

Parameters **wb_address** – address of the wishbone slave to read from

Returns Read Data or None

set_igmp_version(*version*)

Parameters **version** –

test_connection()
Write to and read from the scratchpad to test the connection to the FPGA

upload_to_flash(*binary_file*, *port=-1*, *force_upload=False*, *timeout=30*, *wait_complete=True*)
Upload the provided binary file to the flash filesystem.

Parameters

- **binary_file** – filename of the binary file to upload
- **port** – host-side port, -1 means a random port will be used
- **force_upload** – upload the binary even if it already exists on the host
- **timeout** – upload timeout, in seconds
- **wait_complete** – wait for the upload to complete, or just kick it off

upload_to_ram_and_program(*filename*, *port=-1*, *timeout=10*, *wait_complete=True*, *skip_verification=False*)
Upload an FPG file to RAM and then program the FPGA.

Parameters

- **filename** – the file to upload
- **port** – the port to use on the rx end, -1 means a random port
- **timeout** – how long to wait, seconds
- **wait_complete** – wait for the transaction to complete, return after upload if False
- **skip_verification** – don't verify the image after uploading it

write_wishbone(*wb_address*, *data*)
Used to perform low level wishbone write to a wishbone slave. Gives low level direct access to wishbone bus.

Parameters

- **wb_address** – address of the wishbone slave to write to
- **data** – data to write

Returns response object

class transport_dummy.**NamedFifo**(*maxlen=None*)

__init__(*maxlen=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

pop(*name=None*)

push(*name*, *value*)

3.3.33 transport_katcp

exception transport_katcp.**KatcpConnectionError**

exception transport_katcp.**KatcpRequestError**
An error occurred processing a KATCP request.

exception transport_katcp.**KatcpRequestFail**
A valid KATCP request failed.

exception transport_katcp.**KatcpRequestInvalid**
An invalid KATCP request was made.

class transport_katcp.**KatcpTransport**(***kwargs*)
A katcp transport client for a casperfpga object

__init__(***kwargs*)

Parameters

- **host** –
- **port** –
- **timeout** –
- **connect** –

blindwrite(*device_name*, *data*, *offset=0*)
Unchecked data write. :param device_name: the memory device to which to write :param data: the byte string to write :param offset: the offset, in bytes, at which to write :return: <nothing>

bulkread(*device_name*, *size*, *offset=0*)
Read size-bytes of binary data with carriage-return escape-sequenced. Uses much faster bulkread katcp command which returns data in pages using informs rather than one read reply, which has significant buffering overhead on the ROACH.

Parameters

- **device_name** – name of the memory device from which to read
- **size** – how many bytes to read

- **offset** – the offset at which to read

Returns binary data string

check_phy_counter()

connect(*timeout=None*)
Establish a connection to the KATCP server on the device.

Parameters **timeout** – How many seconds should we wait? Use instance default if None.

deprogram()
Deprogram the FPGA.

Unsubscribes all active tap devices from any multicast groups to avoid confusing switch IGMP snoop tables.

disconnect()
Disconnect from the device server. :return:

get_system_information_from_transport()

is_connected()
Is the transport layer connected to the platform?

Returns True or False

is_running()
Is the FPGA programmed and running?

Returns True or False

katcprequest(*name*, *request_timeout=-1.0*, *require_ok=True*, *request_args=()*, *max_retries=3*, *retry_delay_s=0.5*)
Make a blocking request to the KATCP server and check the result. Raise an error if the reply indicates a request failure.

Parameters

- **name** – request message to send.
- **request_timeout** – number of seconds after which the request must time out
- **require_ok** – will we raise an exception on a response != ok
- **request_args** – request arguments.
- **max_retries** – maximum number of retries in the case of errors (3).
- **retry_delay_s** – the delay between retries in the case of errors (0.5 seconds).

Returns tuple of reply and informs

listbof()

Returns a list of binary files stored on the host device.

listdev(*getsize=False*, *getaddress=False*)
Get a list of the memory bus items in this design.

Returns a list of memory devices

ping()
Use the 'watchdog' request to ping the FPGA host.

Returns True or False

program (*filename=None*)
Program the FPGA with the specified binary file.

Parameters **filename** – name of file to program, can vary depending on the formats supported by the device. e.g. fpg, bof, bin

read (*device_name*, *size*, *offset=0*)
Read size-bytes of binary data with carriage-return escape-sequenced.

Parameters

- **device_name** – name of memory device from which to read
- **size** – how many bytes to read
- **offset** – start at this offset

Returns binary data string

sendfile (*filename*, *targethost*, *port*, *result_queue*, *timeout=2*)
Send a file to a host using sockets. Place the result of the action in a Queue.Queue

Parameters

- **filename** – the file to send
- **targethost** – the host to which it must be sent
- **port** – the port the host should open
- **result_queue** – the result of the upload, nothing '' indicates success
- **timeout** –

set_igmp_version (*version*)
Sets version of IGMP multicast protocol to use

Parameters **version** – IGMP protocol version, 0 for kernel default, 1, 2 or 3

Note: won't work if config keep file is present since the needed ?igmp-version request won't exist on the KATCP interface.

status ()

Returns FPGA status

tap_arp_reload ()
Have the tap driver reload its ARP table right now.

test_connection ()
Write to and read from the scratchpad to test the connection to the FPGA

static test_host_type (*host_ip*, *timeout=5*)
Is this host_ip assigned to a Katcp board?

Parameters

- **host_ip** – as a String
- **timeout** – as an Integer

unhandled_inform (*msg*)
Overloaded from CallbackClient

What do we do with unhandled KATCP inform messages that this device receives?

Pass it onto the registered function, if it's not None

upload_to_flash(*binary_file*, *port=-1*, *force_upload=False*, *timeout=30*, *wait_complete=True*)
Upload the provided binary file to the flash filesystem.

Parameters

- **binary_file** – filename of the binary file to upload
- **port** – host-side port, -1 means a random port will be used
- **force_upload** – upload the binary even if it already exists on the host
- **timeout** – upload timeout, in seconds
- **wait_complete** – wait for the upload to complete, or just kick it off

upload_to_ram_and_program(*filename*, *port=-1*, *timeout=10*, *wait_complete=True*, *skip_verification=False*, ***kwargs*)
Upload an FPG file to RAM and then program the FPGA.

Parameters

- **filename** – the file to upload
- **port** – the port to use on the rx end, -1 means a random port
- **timeout** – how long to wait, seconds
- **wait_complete** – wait for the transaction to complete, return after upload if False
- **skip_verification** – do not verify the uploaded file before reboot

wordread(*device_name*, *size=1*, *word_offset=0*, *bit_offset=0*)

Parameters

- **device_name** – name of memory device from which to read
- **word_count** – how many words to read
- **word_offset** – start at this word offset
- **bit_offset** – start at this bit offset

Returns value in hexadecimal

3.3.34 transport_skarab

3.3.35 transport_tapcp

class transport_tapcp.**TapcpTransport**(***kwargs*)
The network transport for a tapcp-type interface.

__init__(***kwargs*)
Initialized Tapcp FPGA object

Parameters **host** – IP Address of the targeted Board

blindwrite(*device_name*, *data*, *offset=0*, *use_bulk=True*)
Unchecked data write.

Parameters

- **device_name** – the memory device to which to write
- **data** – the byte string to write

- **offset** – the offset, in bytes, at which to write
- **use_bulk** – Does nothing. Kept for API compatibility

deprogram()
Deprogram the FPGA. This actually reboots & boots from the Golden Image

get_firmware_version()
Read the version of the firmware

Returns golden_image, multiboot, firmware_major_version, firmware_minor_version

get_metadata()
Read meta data from user_flash_loc on the fpga flash drive

get_soc_version()
Read the version of the soc

Returns golden_image, multiboot, soc_major_version, soc_minor_version

get_temp()

is_connected()
Is the transport layer connected to the platform?

Returns True or False

is_running()
This is currently an alias for 'is_connected'

listdev()
Get a list of the memory bus items in this design.

Returns a list of memory devices

listdev_pl()

prog_user_image()
(Re)Program the FPGA with the file already on flash

progdev(*addr=0*)

read(*device_name*, *size*, *offset=0*, *use_bulk=True*)
Return size_bytes of binary data with carriage-return escape-sequenced.

Parameters

- **device_name** – name of memory device from which to read
- **size** – how many bytes to read
- **offset** – start at this offset, offset in bytes
- **use_bulk** – Does nothing. Kept for API compatibility

Returns binary data string

read_wishbone(*wb_address*)
Used to perform low level wishbone read from a Wishbone slave.

Parameters **wb_address** – address of the wishbone slave to read from

Returns Read Data or None

static test_host_type(*host_ip*)
Is this host_ip assigned to a Tapcp board?

Parameters **host_ip** –

upload_to_ram_and_program(*filename*, *port=None*, *timeout=None*, *wait_complete=True*, ***kwargs*)

Upload an FPG file to RAM and then program the FPGA.

- Implemented in the child

Parameters

- **filename** – the file to upload
- **port** – the port to use on the rx end, -1 means a random port
- **timeout** – how long to wait, seconds
- **wait_complete** – wait for the transaction to complete, return after upload if False
- **skip_verification** – don't verify the image after uploading it

write_wishbone(*wb_address*, *data*)

Used to perform low level wishbone write to a wishbone slave. Gives low level direct access to wishbone bus.

Parameters

- **wb_address** – address of the wishbone slave to write to
- **data** – data to write

Returns response object

transport_tapcp.**decode_csl**(*csl*)

transport_tapcp.**decode_csl_pl**(*csl*)

transport_tapcp.**get_core_info_payload**(*payload_str*)

transport_tapcp.**get_log_level**()

transport_tapcp.**set_log_level**(*level*)

3.3.36 utils

class utils.**CheckCounter**(*name*, *must_change=True*, *required=False*)

__init__(*name*, *must_change=True*, *required=False*)

Parameters

- **name** – the name of the counter
- **must_change** – this counter should be changing
- **required** – is this counter required?

utils.**check_changing_status**(*counters*, *data_function*, *wait_time*, *num_checks*)

Check a changing set of status fields.

Parameters

- **counters** – a list of CheckCounters
- **data_function** – a function that will return a single value for the fields from field_dict
- **wait_time** – seconds to wait between calls to data_function

- **num_checks** – times to run data_function

utils.**create_meta_dictionary**(*metalist*)
Build a meta information dictionary from a provided raw meta info list.

Parameters **metalist** – a list of all meta information about the system

Returns a dictionary of device info, keyed by unique device name

utils.**deprogram_hosts**(*host_list*)

Parameters **host_list** –

utils.**get_git_info_from_fpg**(*fpg_file*)
Method to get git info from an fpg-file's header :param fpg_file: filename as string :return: Dictionary of git_info

- key = git-repo
- value = git-version

utils.**get_hostname**(***kwargs*)

Parameters **kwargs** –

utils.**get_kwarg**(*field*, *kwargs*, *default=None*)

utils.**hosts_from_dhcp_leases**(*host_pref=None*, *leases_file='/var/lib/misc/dnsmasq.leases'*)
Get a list of hosts from a leases file.

Parameters

- **host_pref** – the prefix of the hosts in which we're interested
- **leases_file** – the file to read

utils.**parse_fpg**(*filename*)
Read the meta information from the FPG file.

Parameters **filename** – the name of the fpg file to parse

Returns device info dictionary, memory map info (coreinfo.tab) dictionary

utils.**program_fpgas**(*fpga_list*, *progfile*, *timeout=10*)
Program more than one FPGA at the same time.

Parameters

- **fpga_list** – a list of objects for the FPGAs to be programmed
- **progfile** – string, the file used to program the FPGAs
- **timeout** – how long to wait for a response, in seconds

utils.**pull_info_from_fpg**(*fpg_file*, *parameter*)
Pull available parameters about x-engine or f-engine from .fpg file. Available options for x-engine: 'x_fpga_clock', 'xeng_outbits', 'xeng_accumulation_len' Available options for f-engine: 'n_chans', 'quant_format', 'spead_flavour'

Parameters

- **fpg_file** – bit file path
- **parameter** – parameter string

Returns pattern value (string)

utils.**threaded_create_fpgas_from_hosts**(*host_list*, *fpga_class=None*, *port=7147*, *timeout=10*, *best_effort=False*, ***kwargs*)
Create KatcpClientFpga objects in many threads, Moar FASTAAA!

Parameters

- **fpga_class** – the class to insantiate, usually CasperFpga
- **host_list** – a comma-seperated list of hosts
- **port** – the port on which to do network comms
- **timeout** – how long to wait, in seconds
- **best_effort** – return as many hosts as it was possible to make

utils.**threaded_fpga_function**(*fpga_list*, *timeout*, *target_function*)
Thread the running of any CasperFpga function on a list of CasperFpga objects. Much faster.

Parameters

- **fpga_list** – list of KatcpClientFpga objects
- **timeout** – how long to wait before timing out
- **target_function** – a tuple with three parts:
  1. string, the KatcpClientFpga function to run e.g. 'disconnect' for fpgaobj.disconnect()
  2. tuple, the arguments to the function
  3. dict, the keyword arguments to the function e.g. (func_name, (1,2,), {'another_arg': 3})

Returns a dictionary of the results, keyed on hostname

utils.**threaded_fpga_operation**(*fpga_list*, *timeout*, *target_function*)
Thread any operation against many FPGA objects

Parameters

- **fpga_list** – list of KatcpClientFpga objects
- **timeout** – how long to wait before timing out
- **target_function** – a tuple with three parts:
  1. reference, the function object that must be run - MUST take FPGA object as first argument
  2. tuple, the arguments to the function
  3. dict, the keyword arguments to the function e.g. (func_name, (1,2,), {'another_arg': 3})

Returns a dictionary of the results, keyed on hostname

utils.**threaded_non_blocking_request**(*fpga_list*, *timeout*, *request*, *request_args*)
Make a non-blocking KatCP request to a list of KatcpClientFpgas, using the Asynchronous client.

Parameters

- **fpga_list** – list of KatcpClientFpga objects
- **timeout** – the request timeout
- **request** – the request string
- **request_args** – the arguments to the request, as a list

Returns a dictionary, keyed by hostname, of result dictionaries containing reply and informs

3.3.37 wishbonedevice

class wishbonedevice.**WishBoneDevice**(*interface*, *controller_name*)

__init__(*interface*, *controller_name*)
x.__init__(...) initializes x; see help(type(x)) for signature

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W