Nuprism Rev1 with Xilinx XU1 SoM User Guide

Table of Contents

1. UDP Packet Format

The format of the packet can be seen on the following table:

UDP Packet Format Table	
2 Bytes	Number of User Words
4 Bytes	Packet ID
4 Bytes	Frame ID
8 Bytes	Timestamp
4 Bytes	Trigger Count
4 Bytes	Reserved
4 Bytes	User word 0
4 Bytes	User word 1
4 Bytes	User word 2
4 Bytes	User word 3
1024 Bytes	If data packet: 128 ADC samples x 4 channels, 16 bits per sample If tail word packet: 10 Tail words, 16 bits per tail word, followed by don't cares

Where:

Number of User Words : This is the number of user words (= 4).

Packet ID: This field starts at 0 and increases by 1 for each packet sent per frame. This can be used by the receiver to reassemble the frame.

Frame ID: This is the frame ID. This field starts at 0 and increases by 1 when each frame is completed. All packets that have this frame ID belong to the same acquisition.

TIMESTAMP: This is a 64 bit timestamp which signifies the transmission time. It starts at 0 on loading of the FPGA increases by 1 every 20 ns. This can be used to establish a chronology of the packets and do QoS measurements if necessary. It also serves as a unique packet ID. It also can be used to do latency measurements along with the acquisition timestamp that is present in the tail words.

Trigger Count**:** This is a trigger count which is the same number for all ADCs and can be used to group acquisitions from the same trigger together. It starts at 0 on loading of the FPGA increases by 1 every trigger (from any source). If the trigger rate is extremely high (e.g. above 10 MHz) the trigger count may be different for different ADCs while the trigger rate is high, so this should be avoided.

User words:

User Word 0: number of samples per trigger (default 512)

User Word 1: Reserved

User word 2: Reserved

User word 3: Bits 31:24: This is a per-ADC user word that denotes the ADC from which the data in the current packet was acquired (from 0 to 4)

ADC Data: These data are composed of 16 bits ADC values, in 4 channel sequence, e.g. for ADC 0, it would be Sample 0 from ADC0 Channel 0, Sample 0 from ADC0 Channel 1, Sample 0 from ADC0 Channel 2, Sample 0 from ADC0 Channel 3, Sample 1 from ADC0 Channel 0, Sample 1 from ADC0 Channel 1, Sample 1 from ADC0 Channel 2, Sample 1 from ADC0 Channel 3, etcetera.

There are 1024 data bytes per packet, i.e. 512 ADC samples, i.e. 128 4-channel sample blocks per packet. So for example a UDP frame of 512 samples would start with 4 packets for ADC0 (packet 0 would be 128 samples of all channels of ADC0, packet 1 would be 128 samples of all channels of ADC 0, packet 2 would be 128 samples of all channels of ADC 0, packet 3 would be 128 samples of all channels of ADC 0). Then the data from ADC1 follows (user word 3 bits 31:24 denotes the ADC number), i.e. packet 4 would be 128 samples of all channels of ADC1, packet 5 would be 128 samples of all channels of ADC 1, packet 6 would be 128 samples of all channels of ADC 1, packet 7 would be 128 samples of all channels of ADC 1. Similarly for ADCs 2, 3, and 4. In total there are 20 packets of ADC data (5 ADCs x 4 Channels each). This would be followed by one packet of tail word data, so a total of 21 packets per acquisition.

Tail Word Data:

Tail_word[0] = trigger_count [15:0]

Tail_word[1] = trigger_count [31:16]

Tail_word[2] = global_timestamp[15:0]

Tail_word[3] = global_timestamp[31:16]

Tail_word[4] = global_timestamp[47:32]

Tail_word[5] = global_timestamp[63:48]

Tail_word[6] = trigger_info [15:0]

Tail_word[7] = trigger_info[31:16]

Where:

trigger_count is the number of triggers received (from any source) by the acquisition system

Note that the difference between trigger count and event counter is the number of triggers that were not acted upon (generally because either the system was in the middle of processing another trigger, or trigger processing was not enabled).

global_timestamp is a 64 bit timestamp sourced from the same timestamp as the UDP header timestamp (thus allowing for latency measurement between when the packet was acquired and when it was transmitted). The timestamp clock is 50 MHz, so every timestamp tick represents 20 nanoseconds.

trigger_info contains information regarding the trigger. This is as follows:

Bits 3:0: trigger type, where the values of this are 0:disabled, 1:emulated, 2:one_shot, 3:external, 4: per-ADC self trigger

Bits 8:4: Triggered ADCs. This will have bits turned on according to which ADC(s) caused the trigger. Bits 8 to 4 correspond to ADCs 4 to 0, respectively. Any channel of an ADC that crosses the threshold will cause that ADC to trigger.

Bits 28:9: This will have bits turned on according to which channel caused the trigger. Bits 28 to 9 correspond to channels 19 to 0, respectively.

Bits 31:29: reserved

2. General Control Methodology

Control of the system is achieved by opening a TCP socket to port 40 of the card's IP. Commands are text strings terminated by newline (symbol \n, ASCII code 0xA) or optionally by carriage return and newline (symbols \r\n ASCII codes 0xD 0xA), for example:

```
start_periodic_acquisition 192.168.0.29 1500
```

for periodic acquisition, send to destination IP 192.168.0.29, to port 1500.

The UDP packets will be sent from a different port for each ADC. ADC 0 will be sent from port 5001, ADC 1 will be sent from port 5002, ADC 2 will be sent from port 5003, ADC 3 will be sent from port 5004, ADC 4 will be sent from port 5005. Though the packets may theoretically arrive out of order, they generally will arrive in order in a per channel basis. However, since each ADC transmission is done by an independent software thread, the transmissions from the various channels will usually be intermingled in a random fashion.

For example here is a wireshark display of the packets for a transmission to 192.168.0.29, port 4915:

58065	273.094926000	Source port: 5002	Destination port: 4915
58066	273.094988000	Source port: 5001	Destination port: 4915
58067	273.094997000	Source port: 5003	Destination port: 4915
58068	273.095002000	Source port: 5001	Destination port: 4915
58069	273.095007000	Source port: 5003	Destination port: 4915
58070	273.095013000	Source port: 5002	Destination port: 4915
58071	273.095018000	Source port: 5003	Destination port: 4915
58072	273.095251000	Source port: 5002	Destination port: 4915
58073	273.095266000	Source port: 5001	Destination port: 4915
58074	273.095273000	Source port: 5004	Destination port: 4915
58075	273.095279000	Source port: 5004	Destination port: 4915
58076	273.095285000	Source port: 5004	Destination port: 4915
58077	273.095289000	Source port: 5005	Destination port: 4915
58078	273.095295000	Source port: 5005	Destination port: 4915
58079	273.095300000	Source port: 5005	Destination port: 4915

Responses are received in text form, but encoded in URL encoding. In many cases, the first character of the response is a 1 or 0, 1 denoting success and 0 denoting an error. This is followed by an optional text message, then terminated by a newline (symbol \n, ASCII code 0xA). The entire string except the newline at the end is URL-encoded to allow for non-printable characters to be sent back (though this capability is not used by the commands mentioned here). For example, the common replies are:

```
1+OK
```

and:

```
0+ERROR
```

where the "+" is actually a space (URL encoding of the space character)

Therefore, parsing the response is done as follows:

(a) Obtain a new line from the socket (terminated by a "\n"). This is the response to the command.
(b) Decode the response using URL decoding
(c) Look at the first character to determine if the command was successful
(d) Look at the rest of the string for a text message describing the return code recovered in step (c)

If only ascertainment of success/failure of the command is desired, the only thing that needs to be checked is the first character of the response, so no URL decoding needs to be done.

3. Commands available for controlling the system

The following commands are available for controlling the system:

---

cal_dac_vs_hv_curve

The command cal_dac_vs_hv_curve calibrates the HV voltage in order for the command set_hv_voltage to be able to function. The command is:

```
cal_dac_vs_hv_curve [dac_step]
```

where dac_step is the step in DAC codes (128 if omitted). Note that after execution of this command, ldo_en will be enabled and HV will be disabled, and HV DAC code will be set to produce around 36 Volts when HV is enabled. To enable HV, execute enable_hv.

---

check_cmp

This command checks and verifies that a comparator is working. For this to work, you need to supply an external known square wave signal to the mezzanine board. You enter the necessary parameters and then the command checks if the frequency of counts received from the comparators is matching the input frequency.

```
check_cmp cmp_num input_freq(kHz) noise_lvl(mV) signal_voltage(mV) noise_width(mV)
```

The parameters are:
cmp_num: number of the comparator to check. (1,2,3)
input_freq: frequency of the known test signal (in kHz)
noise_lvl: the average baseline noise for the comparator in mV. (Use get_noise_level_cmp command to get that)
signal_voltage: the amplitude of the known test signal (mv)
noise_width: the width of the noise for the comparator in mV (use get_cmp_noise_level command to get that)

The command returns a MATLAB script with the recorded data which has the counts for each comparator voltage which can be used to analyse the results in detail. There is also a TEST PASSED or FAILED check in the beginning of the result to notify if the comparator is working as expected.

---

check_cmp_all

This command checks and verifies that all comparators are working. For this to work, you need to supply an external known square wave signal to the mezzanine board. The frequency should be 50kHz and amplitude should be 100mV.

```
check_cmp_all cmp_1_noise_lvl(mV) cmp_2_noise_lvl(mV) cmp_3_noise_lvl(mV)
```

The parameters are:
cmp_1_noise_lvl: the noise floor for comparator 1 in mV
cmp_2_noise_lvl: the noise floor for comparator 2 in mV
cmp_3_noise_lvl: the noise floor for comparator 3 in mV

The command returns a MATLAB script that on running will display the recorded data. Along with that, the output also contains the phrase “TEST PASSED” or “TEST FALED” for each comparator.

---

deser_clk_phase_shift

In order to manually phase shift the clock oscillators:

```
deser_clk_phase_shift psen_mask updown_mask num_steps
```

where psen_mask and updown_mask are 3-bit masks for psen and up/down (bit0 = clk1, bit1 = clk2; bit2 = clk3). The parameter num_steps is the number of steps. This command should only be used during development by advanced users.

---

disable_adc_test_signals

In order to enable the test signals from the ADCs, use the:

```
disable_adc_test_signals adc_num
```

where adc_num is 0 to 4.

---

disable_emulated_trigger

To disabled the emulated trigger, use the command:

```
DISABLE_EMULATED_TRIGGER
```

---

disable_fast_led

To disable the fast led, use the command:

```
disable_fast_led
```

---

disable_hv

To disable the high voltage generation circuit in the daughterboard, issue the command:

```
disable_hv
```

---

disable_ldo_en

To disable ldo_en, use:

```
disable_ldo_en
```

---

disable_mezzanine_dac

To disable the mcp4801 DAC in the LPC Mezzanine, use the command:

```
disable_mezzanine_dac
```

---

disable_vcc_lf_en

To disable vcc_lf_en, use:

```
disable_vcc_lf_en
```

This command is synonymous to disable_fast_led

---

disable_vcc_ls_en

To disable vcc_ls_en, use:

```
disable_vcc_ls_en
```

This command is synonymous to disable_mezzanine_dac

---

disable_vcc_qp_en

To disable vcc_qp_en, use:

```
disable_vcc_qp_en
```

This command is synonymous to disable_hv

---

disable_vccl_en

To disable vccl_en, use:

```
disable_vccl_en
```

---

do_not_send_fast_led_to_ext_trig_out

To stop sending the fast led control signal out via the external trigger output, use this command:

```
do_not_send_fast_led_to_ext_trig_out
```

---

do_one_shot_fast_led_pulse

To send a one-shot pulse to the fast LED when it is in one-shot mode, use the command:

```
do_one_shot_fast_led_pulse
```

---

enable_adc_test_signals

In order to enable the test signals from the ADCs, use the:

```
enable_adc_test_signals adc_num
```

where adc_num is 0 to 4.

---

enable_emulated_trigger

To enable the emulated trigger, use the command:

```
ENABLE_EMULATED_TRIGGER
```

---

enable_fast_led

To enable the fast led, use the command:

```
enable_fast_led
```

---

enable_hv

To enable the high voltage generation circuit in the daughterboard, issue the command:

```
enable_hv
```

---

enable_mezzanine_dac

To enable the mcp4801 DAC in the LPC Mezzanine, use the command:

```
ENABLE_MEZZANINE_DAC
```

---

enable_ldo_en

To enable ldo_en, use:

```
enable_ldo_en
```

---

enable_vcc_lf_en

To enable vcc_lf_en, use:

```
enable_vcc_lf_en
```

This command is synonymous to enable_fast_led

---

enable_vcc_ls_en

To enable vcc_ls_en, use:

```
enable_vcc_ls_en
```

This command is synonymous to enable_mezzanine_dac

---

enable_vcc_qp_en

To enable vcc_qp_en, use:

```
enable_vcc_qp_en
```

This command is synonymous to enable_hv

---

enable_vccl_en

To enable vccl_en, use:

```
enable_vccl_en
```

---

exec_pmt_cmd

The exec_pmt_cmd command executes a command via UART on the selected PMT. The command is:

```
exec_pmt_cmd command
```

For example:

```
OUTGOING: EXEC_PMT_CMD 01LV
INCOMING: 01LV0029280%0D
```

Note the commands are using the old UART protocol and are converted to MODBUS for actual communications with the PMT.

---

get_adc_channel_num_test_enable_mask

In order to get the mask of the ADC channels with the ADC channel test mode, the command is:

```
get_adc_channel_num_test_enable_mask
```

The results of the command is the mask in decimal.

---

get_adc_data_format

In order to get the data format of the ADC, use the following command :

```
get_adc_channel_num_test_enable adc_num
```

The results of the command is the data format in decimal, 0 for two's complement and 1 for offset binary.

---

get_adc_mask

In order to get the mask of the ADCs that are enabled for acquisition and transmission:

```
get_adc_mask
```

The results of the command is "1 OK: the_mask" where the_mask is the mask in decimal.

---

get_base_clock_source

In order to get the status of the selected clock of the clock multiplexer that feeds CLK1 to the clock cleaner, use the command:

```
gel_base_clock_source
```

This returns "1 OK: 1" if the source is the mezzanine clock, and "1 OK: 0" if it is the RJ45.

---

get_clnr_status_pins

In order to get the clock cleaner status pins, use the following commands:

```
get_clnr_status_pins
```

The results of the command is in decimal, where:

Bit 3: LOL: Loss of Lock Loss-of-Lock Status Flag for Digital PLL. Logic-high indicates digital PLL not locked.

Bit 2: LOS0: Loss-of-Signal Status Flag for Input Reference 0. Logic-high indicates input reference failure.

Bit 1: LOS1: Loss-of-Signal Status Flag for Input Reference 1. Logic-high indicates input reference failure.

Bit 0: HOLD: Holdover Status Flag for Digital PLL. Logic-high indicates digital PLL in holdover status.

---

get_data_ad7124

This comannds allow us to read one entry from the AD7124 SPI Controller’s FIFO. If there is no data in the FIFO the command return “RX Fifo Empty”

```
COMMAND:  get_data_ad7124
RESPONSE: Data+read%3A+0xffffffff%0A1+OK
```

In case there is no data in the RX FIFO

```
COMMAND:  get_data_ad7124
RESPONSE: 0+SPI+Read+Failed.+Check+Last+Error+Below%3A%0AAD7124_RX_FIFO_EMPTY
```

---

get_error_count_nonzero_channels

In order to get the status of the error counters, use:

```
get_error_count_nonzero_channels
```

This returns "1 OK: mask" where mask is a hexadecimal number that is which channels have recorded an error. This command will typically be used only if manual error measurement is being done.

---

get_emulated_trigger_speed

To get the emulated trigger speed in Hz, use the command:

```
GET_EMULATED_TRIGGER_SPEED
```

The return value will be the trigger speed in Hz.

---

get_enable_emulated_trigger_status

To get the emulated trigger status (enabled or disabled), use the command:

```
GET_ENABLE_EMULATED_TRIGGER_STATUS
```

The return value will be 1 if enabled, 0 if disabled.

---

get_enable_fast_led_status

To get the fast LED enable status (enabled or disabled), use the command:

```
get_enable_fast_led_status
```

The return value will be 1 if enabled, 0 if disabled.

---

get_fast_led_mask

In order to get the mask of the fast LEDs that are enabled for fast LED pulsing, use:

```
get_fast_led_mask
```

The results of the command is "1 OK the_mask" where the_mask is the mask in decimal.
Bit0 = Fast LED 1, Bit1 = Fast LED 2, Bit2 = Fast LED 3

---

get_fast_led_mode

To check if the fast LED mode, use the command:

```
get_fast_led_mode
```

The return value will be 0 if periodic, 1 if one-shot, 2 if synced to external trigger

---

get_fast_led_speed

To get the fast LED speed in Hz, use the command:

```
GET_FAST_LED_SPEED
```

The return value will be the fast LED speed in Hz.

---

get_fpga_temp

To get the FPGA junction temperature from the system monitor IP in the FPGA, issue the command:

```
get_fpga_temp
```

The return value will be "1 OK: TEMP" where TEMP is the temperature in celsius.

get_hdc1080_temp

In order to get a temperature reading from the hdc1080 humidity sensor chip use the following command:

```
get_hdc1080_temp
```

The result is the temperature in celsius.

get_humidity

In order to get a humidity reading from the hdc1080 humidity sensor chip use the following command:

```
get_humidity
```

The result is the relative humidity in percent.

get_hv_limit

In order to get the value of the hi voltage limit in the daughterboard, use the following command:

```
get_hv_limit
```

The result is the voltage limit in volts.

get_hv_voltage

In order to get the value of the hi voltage in the daughterboard, use the following command:

```
get_hv_voltage
```

The result is the voltage in volts.

get_hw_version

```
get_hw_version
```

Returns the hardware version (Vivado project timestamp). Timestamp is HHDDMMYY.

get_lpc_cmp_freq

```
get_lpc_cmp_freq clknum
```

Parameter: clk_num, the number of the clock that is to be measured. For comparators they are 4,5,6 for comparators A,B and C respectively.
Returns the frequency(Hz) of the signal at the output of comparator for the corresponding clock.

get_noise_level_cmp

Get the noise level for the comparators. Usage is as follows:

```
get_noise_level_cmp cmp_num
```

The parameter cmp_num is optional. If not specified, then the command will output the noise level statistics for all comparators. Comparator numbers is in range 1, 2 and 3. The command returns a MATLAB script that on running dislays the mean, max, min and noise width in mV.

get_ntc_temp

This command returns the temeprature reading for each NTC element. The output is self explanatory.

```
COMMAND: get_ntc_temp
RESPONSE:NTC_1_20.3525degC%0ANTC_2_21.174degC%0ANTC_3_21.9401degC%0ANTC_4_20.2929d
         egC%0A1+OK
```

From the example above, we can read the temperatures. NTC_1 is at 20.3525 degC, NTC_2 is at 21.174 degC, NTC_3 is at 21.9401 degC and NTC_4 is at 20.2929 degC.

---

get_num_samples_per_packet

```
get_num_samples_per_packet
```

This command returns the number of samples per packet. The reply is of the format:

```
1 OK: num_of_samples
```

where num_of_samples is the number of samples in decimal.

---

get_PMT_FW

Checks firmware version running on selected PMT.

```
Get_PMT_FW val
```

Val is 0-19.

Returns either UART FW present, Modbus FW present, or neither (in which case either PMT is not programmed or disconnected).

---

get_pre_trigger_delay

To get the pre-trigger delay in samples, use the command:

```
GET_PRE_TRIGGER_DELAY 0
```

The return value will be the number of samples the input is delayed, between 0 and 511, (though actual delays are slightly different due to an additional fixed latency between the ADC sampling strobe and the data appearing at the input of the digital acquisition circuit)

---

get_pressure

In order to get a pressure reading from the pressure sensor, use the following command:

```
get_pressure
```

The result is the pressure in mbar.

---

get_pressure_sensor_temp

In order to get a temperature reading from the pressure sensor, use the following command:

```
get_pressure_sensor_temp
```

The result is the temperature in celsius.

---

get_selected_pmt

The get_selected_pmt command returns to which PMT the UART is currently connected. The command is:

```
get_selected_pmt
```

The returned value will be a number from 0 to 19.

---

get_self_trigger_thresholds

To get the self trigger threshold, use the following command:

```
get_self_trigger_threshold
```

The value returned will be a 16-bit signed number, in decimal.

---

get_self_trigger_on_falling_edge

To see whether the falling edge (i.e. going below threshold) of self trigger should is used, use the following command:

```
get_self_trigger_on_falling_edge
```

The return value is 0 or 1, 0 being rising edge is used (going above threshold) and 1 being falling edge is used (going below threshold)

---

get_sw_version

```
get_sw_version
```

Returns the software compilation (Vitis project) timestamp date and time string.

---

get_temp

In order to get a temperature reading from the temperature monitors, use the following command:

```
get_temp monitor_index
```

where monitor_index is 1 to 3. The result is the temperature in celsius.

---

get_trigger_freq

To get the emulated trigger speed in Hz, use the command:

```
get_trigger_freq
```

The return value will "1 OK: freq_in_HZ".

---

get_vcc_lf_ok

To get the value of vcc_lf_ok:

```
get_vcc_lf_ok
```

The return value will "1 OK: val", where "val" is either 0 or 1.

---

get_vcc_ls_ok

To get the value of vcc_ls_ok:

```
get_vcc_ls_ok
```

The return value will "1 OK: val", where "val" is either 0 or 1.

---

get_vcc_qp_ok

To get the value of vcc_qp_ok:

```
get_vcc_qp_ok
```

The return value will "1 OK: val", where "val" is either 0 or 1.

---

ldo_get_power

```
ldo_get_power ldo_num
```

This command returns the power in watts from the LDO denoted in ldo_num, where ldo_num is between 1 and 8.

---

ldo_get_voltage

```
ldo_get_voltage ldo_num
```

This command returns the voltage in volts from the LDO denoted in ldo_num, where ldo_num is between 1 and 8.

---

ldo_get_shunt_voltage

```
ldo_get_shunt_voltage ldo_num
```

This command returns the shunt voltage in volts from the LDO denoted in ldo_num, where ldo_num is between 1 and 8.

---

mmeter_read_mag_field

Returns magnetic field in x, y, and z axes in Gauss.

```
mmeter_read_mag_field
```

---

mmeter_calibrate

Performs calibration of MMC5983 magnetic sensor. Takes ~ a minute and requires chip to be moved and rotated for full surface calibration.

```
mmeter_calibrate
```

---

mmeter_get_new_offset

Calculates offset induced by thermal variation. Run this command before running mmeter_read_mag_field for the first time.

```
mmeter_get_new_offset
```

---

mmeter_initiate_continuous_measurements

Initiates continuous measurements mode.

```
mmeter_initiate_continuous_measurements MODR MBW MSET
```

Where MODR is the continuous measurement rate – how often a new measurement is taken.

0 – One Shot

1 – 1 Hz

2 – 10 Hz

3 – 20 Hz

4 – 50 Hz

5 – 100 Hz

6 – 200 Hz

7 – 1000 Hz

MBW is the bandwidth – how long each measurement takes.

0 – 100 Hz, 8ms/measurement

1 – 200 Hz, 4ms/measurement

2 – 400 Hz, 2ms/measurement

3 – 800 Hz, 0.5ms/measurement

MSET is the periodic set rate – SET (re-aligning magnetic domains) occurs per number of measurements.

0 – 1 measurement

1 – 25 measurements

2 – 75 measurements

3 – 100 measurements

4 – 250 measurements

5 – 500 measurements

6 – 1000 measurements

7 – 2000 measurements

Note bandwidth and continuous measurement frequency must be set together for correct operation:

Continuous Measurement Frequency	**Bandwidth**
One Shot	100 Hz
1 Hz	100 Hz
10 Hz	100 Hz
20 Hz	100 Hz
50 Hz	100 Hz
100 Hz	100 Hz
200 Hz	200 Hz
1000 Hz	800 Hz

---

optimize_clock_delays

```
optimize_clock_delays
```

This command optimizes the clock delays for all ADCs by measuring the eye openings of each channel and adjusting the phases of the ADC clocks to achieve optimal sampling. This command is run automatically at boot and there should not normally be a need to run it after that.

---

pmt_get_frequent_regs

```
pmt_get_frequent_regs pmt_selected
```

This command reads the frequently polled registers for of the selected PMT, which are HVCurVal, HVVolVal, HVVolNom, STATUS1, and MCUTemp. The field "pmt_selected" can be between 0 and 19 (either in decimal or hex (if prefixed by "0x")). The response, if successful, is “1 OK” followed by the values of the registers, in decimal, separated by spaces. The order of the registers read in the response is: HVVolNom HVCurVal HVVolVal STATUS1 MCUTemp.

pmt_read_reg

```
pmt_read_reg pmt_selected reg
```

This command reads the register of the selected PMT. The field "pmt_selected" can be between 0 and 19 (either in decimal or hex (if prefixed by "0x")). The field "reg" can be either specified as a number (either in decimal or hex (if prefixed by "0x)) or one of "modBusAddr" "STATUS1" "MCUTemp" "TripTime" "RampUpSpd" "RampDwnSpd" "HVCurrMax" "HVVolNom" "HVVolMarg" "HVCurVal" "HVVolVal" "HVVolRef" (without the quotes).

---

pmt_read_all_regs

```
pmt_read_all_regs pmt_selected
```

This command reads all the registers from a specified pmt. The command output 13 different register values. The ouptut is in the following order: UNIQUE_ID, MAGIC_KEY, STATUS, MCUTemp, TRIP_TIME, RAMP_UP_SPEED, RAMP_DOWN_SPEED, MAX_HV_CURRENT, NOM_HV_VOLTAGE, MARG_HV_VOLTAGE, HV_CURRENT, HV_VOLTAGE, REF_HV_VOLTAGE

```
Command:  pmt_read_all_regs 9
Response: 1+OK+46139+0+0+24+65535+65535+65535+65535+65535+65535+80+31+31687
```

---

pmt_read_n_regs

```
pmt_read_n_regs pmt_selected reg num_regs_to_read
```

This command reads the register of the selected PMT. The field "pmt_selected" can be between 0 and 19 (either in decimal or hex (if prefixed by "0x")). The field "reg" can be either specified as a number (either in decimal or hex (if prefixed by "0x)) or one of "modBusAddr" "STATUS1" "MCUTemp" "TripTime" "RampUpSpd" "RampDwnSpd" "HVCurrMax" "HVVolNom" "HVVolMarg" "HVCurVal" "HVVolVal" "HVVolRef" (without the quotes). The parameter “num_regs_to_read” is the number of registers to read. The response, if successful, is “1 OK” followed by the values of the registers, in decimal, separated by spaces, followed by the modbus response for debugging purposes. For example:

```
Command:  pmt_read_n_regs 1 HVVolNom 4
Response: 1 OK 0 0 14336 4096 modbus response : 02030800000000380010009a33
```

---

pmt_toggle_hv

```
pmt_write_reg pmt_selected val
```

This command toggles the high voltage of the selected PMT. The field "pmt_selected" can be between 0 and 19 (either in decimal or hex (if prefixed by "0x")). The field "val" can be 0 or 1, which turns off or on the HV, respectively.

---

pmt_write_reg

```
pmt_write_reg pmt_selected reg data
```

This command writes the register of the selected PMT. The field "pmt_selected" can be between 0 and 19 (either in decimal or hex (if prefixed by "0x")). The field "reg" can be either specified as a number (either in decimal or hex (if prefixed by "0x)) or one of "modBusAddr" "STATUS1" "MCUTemp" "TripTime" "RampUpSpd" "RampDwnSpd" "HVCurrMax" "HVVolNom" "HVVolMarg" "HVCurVal" "HVVolVal" "HVVolRef" (without the quotes).

---

read_ad7124_reg

This command reads a specific register from the AD7124 ADC. The list of available registers can be found at Page 79 of the AD7124 datasheet from here. The register address needs to be specified in DECIMAL format.

```
COMMAND:  read_ad7124_reg 2
RESPONSE: Ad7124+Register+Read+Successful%0AData+Read%3A+0x7473be%0A1+OK
```

In the example above, we tried reading the DATA register from the ADC. And we received 0x7473BE. Note the data received is in HEX format.

---

read_all_ad7124

This command reads all the entries in the Ad7124 SPI Controller's RX FIFO. If there is no data then the command retrun RX FIFO Empty

```
COMMAND:  read_all_ad7124
RESPONSE: DataRead%5B6%5D%3A+0x0%2C0x0%2C0x0%2C0x0%2C0x0%2C0x0%0A1+OK
```

In the example above, the data received was 0 for 6 times.

---

read_hv_adc

To read the mcp3421 ADC in the LPC mezzanine use the following command:

```
read_hv_adc resolution gain
```

where resolution is 0=12 bit, 1 = 14 bit, 2 = 16 bit, 3 = 18 bit, and gain is 0 = x1, 1 = x2, 2 = x4, 3 = x8.

---

read_mezzanine_ad5685

To read the ad5685 DAC in the LPC mezzanine use the following command:

```
read_mezzanine_ad5685 dac_mask
```

where dac_mask is a 4-bit mask that defines which DAC to read from, where bit 0 is DAC A, bit 1 is DAC B, bit 2 is DAC C and bit 3 is DAC D. Only one bit should be enabled at a time. A value of 0 in the mask reads from DAC A.

---

read_mmeter_temp

To read temperature from the magnetometer, use the command:

```
read_mmeter_temp
```

The result is "1 OK: temp" where temp is the temperature in celsius.

---

record_ntc_temp

This command takes measurement for the 4 NTC temeprature senors and prints out a MATLAB script that allows the user to plot and view the recorded data.

```
read_ntc_temp num_seconds_to_measure intveral_between_each_measurement(sec)

read_ntc_temp 86400 2 <- measure for 24Hrs and 2 sec between each measurment
```

It is to be noted that the minmum interval between each measurement needs to be greater than or equal to 2 seconds. This is the minimum time required to acquire the data from the NTC Temeprature sensors.

---

reinitialize_pmts

To perform the PMT initialization sequence again, use the command:

```
reinitialize_pmt
```

The result is "1 OK ".

---

report_phase_scan

```
report_phase_scan lower_limit upper_limit
```

This command returns a Matlab script that allows for graphing of the phase margins of each ADC channel. The parameters lower_limit and upper_limit are the limits in steps for the phase scans, where each step is 15.873 ps. Typical limits are -100 and 100. The error rate for the determination of good/no good phase is 1e-6.

---

reset_all_PMT_addresses

```
reset_all_PMT_addresses
```

This command resets the PMT addresses, i.e. assigns each PMT an address between 1 and 20 that corresponds to PMT 0 to 19, respectively.

---

reset_error_counters

```
reset_error_counters
```

This command resets the error counters. This would be done if manual error measurements are desired. Prior to this command, the set_all_channels_to_ramp command should be issued.

---

reset_total_and_processed_trigger_count

In order to reset the event and trigger count (as reported in the tail words) use the following command:

```
reset_total_and_processed_trigger_count
```

It is recommended to issue this command after system startup after enabling DSP processing, since there may be an initial mismatch between those counts that may be erroneously attributed to missed triggers.

---

scan_lpc_cmp_mv_threshold

This command sweeps a DAC within a specified voltage range. Then returns the number of counts measured at the comparator’s output for each voltage.

```
scan_lpc_cmp_mv_threshold clock_num dac_index min_mv max_mv step_mv step_delay_us
```

Parameters:
clock_num: the clock number to read the counts from. For comparators A,B,C it is 4,5,6.

dac_index: the index of the dac to sweep. Available options are 0,1,2 for Comparators A,B,C
min_mv: the starting voltage level of the sweep in mV.
max_mv: the ending voltage level of the sweep in mV.
step_mv: the step between every two DAC output in mV.
step_delay_us: delay time in microseconds between each succsessive DAC output.

The command returns a MATLAB script that contains the counts form each DAC output.

---

sel_embedded_external_trigger_source

In order to select the external trigger source, use:

```
sel_embedded_external_trigger_source val
```

where val = 0 signifies the mezzanine trigger, val = 1 signifies the trigger that comes over the POE Ethernet.

---

sel_mezz_as_base_clock_source

In order to set the mezzanine as the selected clock of the clock multiplexer that feeds CLK1 to the clock cleaner, use the command:

```
sel_mezz_as_base_clock_source
```

---

sel_rj45_as_base_clock_source

In order to set the RJ45 as the selected clock of the clock multiplexer that feeds CLK1 to the clock cleaner, use the command:

```
sel_rj45_as_base_clock_source
```

---

select_sync_to_external_fast_led

```
select_sync_to_external_fast_led
```

This command sets the fast led mode to sync to trigger. Note that this means that the fast LED is synced to any source of trigger, including emulated trigger

---

select_one_shot_fast_led

```
select_one_shot_fast_led
```

This command sets the fast led mode to one shot.

---

select_periodic_fast_led

```
select_periodic_fast_led
```

This command sets the fast led mode to periodic mode.

---

select_pmt

The select_pmt commands selects which PMT communicated with, for usage with the exec_pmt_cmd command

```
select_pmt val
```

val is 0 to 19.

For example:

```
select_pmt 3
```

---

send_data_ad7124

To send generic data on the ad7124 SPI bus, use the following command. The paramter data is the data sent. Must be in Hex format without “0x”. The data sent throught the command has to be less than or equal to 32 bits. That is the transaction limit of the SPI Contorller.

```
COMMAND:  send_data_ad7124 5C
RESPONSE: Transaction+Successful%0A1+OK
```

---

send_fast_led_to_ext_trig_out

To send the fast led control signal out via the external trigger output, use this command:

```
send_fast_led_to_ext_trig_out
```

---

send_one_shot_pulse

To send a single trigger, when in one shot mode as set up via set_up_one_shot_acquisition, use the following command:

```
send_one_shot_pulse
```

---

set_adc_mask

In order to change the ADCs that are acquired and sent over UDP use the command:

```
set_adc_mask mask
```

e.g.:

```
set_adc_mask 0xC
```

The bits that are enabled in the mask correspond to the ADCs that will be acquired and transmitted via UDP, in ascending order (i.e. bit 0 corresponds to adc 0, bit 1 corresponds to adc 1, etc.).

---

set_adc_channel_num_test_enable_mask

It is possible to set the 5 lower bits of the 16-bit ADC output to ADC[2:0],Channel[1:0]. These bits will replace the lower 4 bits of the 16 bit ADC data (which are otherwise 0) and the LSB of the 12-bit ADC data (which is the 5th LSB in the 16-bit data). The command is:

```
set_adc_channel_num_test_enable_mask mask
```

The bits that are enabled in the mask correspond to the ADCs for which this test mode will be enabled. Note that this can be used in conjunction with the normal ADC test modes. In particular, setting an ADC to test mode 0 will mean that the data from the ADC will effectively be the overall channel number (from 0 to 19).

---

set_adc_custom_pattern

In order to set the custom pattern for the ADCs, use the following:

```
set_adc_custom_pattern adc_num custom_pattern
```

where adc_num is 0 to 4 and custom_pattern is a 12-bit pattern (e.g. 0xABC).

---

set_adc_data_format

In order to switch an ADC between two's complement and offset binary data:

```
set_adc_data_format adc_num val
```

where "val" is 0 for two's complement and 1 for offset binary.

---

set_adc_test_signal_type

In order to select the test signal types from the ADCs, use the following:

```
set_adc_test_signal_type adc_num channel_num test_signal_type
```

where adc_num is 0 to 4, channel_num is 0 to 3, and test_signal_type is as follows:

0 : Normal Operation
1: All 0s
2: All 1s
3: Toggle pattern (data alternate between 101010101010
and 010101010101)
4: Digital ramp: Data will increment by 4 every clock cycles, and wrap around
5: Custom pattern
6: Deskew pattern: 0xAAA
8: PRBS pattern

9: 8 point sine wave, data is repeating sequence: 0, 15648, 22144, 15648, 0, -15664, -22160, -15664

set_all_channels_to_normal

In order to set all ADC channels for all ADCs to normal operation (i.e. disable test mode), use:

```
set_all_channels_to_normal
```

---

set_all_channels_to_ramp

In order to set all ADC channels for all ADCs to ramp operation, use:

```
set_all_channels_to_ramp
```

---

set_all_channels_to_sine

In order to set all ADC channels for all ADCs to sine wave operation, use:

```
set_all_channels_to_ramp
```

---

set_emulated_trigger_speed

To set the emulated trigger speed, use the following command:

```
SET_EMULATED_TRIGGER_SPEED freq_in_hz
```

where freq_in_hz is the desired frequency in Hz, for example:

```
SET_EMULATED_TRIGGER_SPEED 1000
```

The actual frequency obtained is returned as the returned value, and may differ from the requested value (depends on whether the frequency divider ratio for the requested frequency is an integer)

---

set_fast_led_mask

In order to set the mask of the fast LEDs that are enabled for fast LED pulsing, use:

```
set_fast_led_mask mask
```

For the mask, Bit0 = Fast LED 1, Bit1 = Fast LED 2, Bit2 = Fast LED 3.

---

set_fast_led_dac

To write to the mcp4802 DAC in the LPC mezzanine that corresponds to the fast LED amplitude, use the following command:

```
set_fast_led_dac val
```

where val is an 8-bit value (in decimal or in hex (if prefixed by 0x)). This is equivalent to the command:

```
write_mezzanine_dac 1 1 val 1
```

---

set_fast_led_speed

To set the fast led frequency in periodic mode, use the following command:

```
SET_FAST_LED_SPEED freq_in_hz
```

where freq_in_hz is the desired frequency in Hz, for example:

```
SET_FAST_LED_SPEED 1000
```

The actual frequency obtained is returned as the returned value, and may differ from the requested value (depends on whether the frequency divider ratio for the requested frequency is an integer)

---

set_hv_limit

In order to set the value of the hi voltage limit in the daughterboard, use the following command:

```
set_hv_limit val
```

Where val is the value in volts.

---

set_hv_voltage

In order to set the value of the hi voltage in the daughterboard, use the following command:

```
set_hv_voltage val [method]
```

Where val is the value in volts. Method is the method used to determine the DAC code from the calibration data. Method = 0 for closest neighbor, method = 1 for linear interpolation. If method is omitted, then method = 1 (linear interpolation) is used. The operation will only succeed if 0 <= val <= hv_limit where hv_limit is set by the set_hv_limit command

---

set_pre_trigger_delay

To set the pre-trigger delay, use the following command:

```
SET_PRE_TRIGGER_DELAY pre_trigger_delay
```

where pre_trigger_delay is the desired delay in samples, from 0 to 511.

```
SET_PRE_TRIGGER_DELAY 35
```

will set the pre-trigger delay to 35.

---

set_pmt_switch

To set a PMT switch to a specific value, use:

```
set_pmt_switch switch_num val
```

where switch_num is 0 to 19, and val is an 8-bit unsigned number.

---

set_num_samples_per_packet

```
set_num_samples_per_packet numsamples
```

This command determines how many samples are acquired by each acquisition and sent via UDP, theoretically any number from 512 to 65536 can be chosen for numsamples, but that number must be divisible by 64. In reality FIFO length is 4096 so anything above that risks not being acquired properly.

---

set_self_trigger_thresholds

To set the self trigger threshold, use the following command:

```
set_self_trigger_threshold threshold
```

where threshold is a 16-bit signed number, for example:

```
set_self_trigger_threshold 100
```

---

set_self_trigger_on_falling_edge

To set whether the falling edge (i.e. going below threshold) of self trigger should be used, use the following command:

```
set_self_trigger_on_falling_edge val
```

where val is 0 or 1, 0 being use rising edge (going above threshold) and 1 being use falling edge (going below threshold)

---

set_slow_led_dac

To write to the mcp4802 DAC in the LPC mezzanine that corresponds to the slow LED amplitude, use the following command:

```
set_slow_led_dac val
```

where val is an 8-bit value (in decimal or in hex (if prefixed by 0x)). This is equivalent to the command:

```
write_mezzanine_dac 1 1 val 0
```

---

set_trigger_freq

To set the emulated trigger speed, use the following command:

```
set_trigger_freq freq_in_hz
```

where freq_in_hz is the desired frequency in Hz, for example:

```
set_trigger_freq 1000
```

---

set_up_one_shot_acquisition (obsolete)

The set_up_one_shot_acquisition command has the following syntax:

```
set_up_one_shot_acquisition ip port ethernet_interface
```

where “ethernet_interface” is 0 for eth0 (1Gbit/sec link) and 1 for eth1 (100Mbit/sec, POE).

For example:

```
set_up_one_shot_acquisition 192.168.0.29 1500 0
```

This sets up the card for one-shot acquisition, with the destination being the IP address (192.168.0.29) and port (1500) specified, via Ethernet 0 (1Gbit/sec link). This trigger includes emulated trigger with the emulated trigger commands.

---

start_acquisition

The start_acquisition command has the following syntax:

```
start_acquisition ip port ethernet_interface
```

where “ethernet_interface” is 0 for eth0 (1Gbit/sec link) and 1 for eth1 (100Mbit/sec, POE).

For example:

```
start_acquisition 192.168.0.29 1500 0
```

This starts UDP transmission, with the destination being the IP address (192.168.0.29) and port (1500) specified, via Ethernet 0 (1Gbit/sec link). This trigger includes emulated trigger with the emulated trigger commands.

---

start_periodic_acquisition_self_trigger

The start_periodic_acquisition_self_trigger command has the following syntax:

```
start_periodic_acquisition_self_trigger ip port ethernet_interface
```

where “ethernet_interface” is 0 for eth0 (1Gbit/sec link) and 1 for eth1 (100Mbit/sec, POE).

For example:

```
start_periodic_acquisition_self_trigger 192.168.0.29 1500 0
```

This starts UDP transmission, with the destination being the IP address (192.168.0.29) and port (1500) specified, via Ethernet 0 (1Gbit/sec link). This trigger includes emulated trigger with the emulated trigger commands.

---

start_periodic_acquisition (obsolete)

The start_periodic_acquisition command has the following syntax:

```
start_periodic_acquisition ip port ethernet_interface
```

where “ethernet_interface” is 0 for eth0 (1Gbit/sec link) and 1 for eth1 (100Mbit/sec, POE).

For example:

```
start_periodic_acquisition 192.168.0.29 1500 0
```

This starts UDP transmission, with the destination being the IP address (192.168.0.29) and port (1500) specified, via Ethernet 0 (1Gbit/sec link). This trigger includes emulated trigger with the emulated trigger commands.

---

start_periodic_acquisition_ext_trigger (obsolete)

For example:

```
start_periodic_acquisition_ext_trigger 192.168.0.29 1500 0
```

This command is identical to the start_acquisition command above.

---

stop_acquisition

```
stop_acquisition
```

This stops the UDP stream and triggering of data acquisitions.

---

turn_slow_leds_off

To turn off the slow LEDs on in the LPC mezzanine, use the following command:

```
turn_slow_leds_off
```

---

test_hv_cmp

This command allows user to test the HV system for the Mezzanine board.

```
test_hv_cmp
```

This command will report a matlab script that on running will display the PE counts for 55V HV output. The data from all thre comaprators will be recorded and retunred. The user can use this to determine if the system is working properly.
NOTE: This command will set the HV limit to 60V and will recalibrate the DAC vs HV codes. If the user shall require to use a HV higher than 60V, then they have to call apropriate commands to reset the HV limit and recalibrate the DAC vs HC codes.

---

test_mezz_leds

This command allows user to test the LEDsi on the Mezzanine board.

```
test_mezz_leds
```

This command is fully automated and required that the user viaully instpects the board to verify that the LEDs are working in order. The sequnces of the test is as folows:
1. Slow leds are turned on with their intensity going from 0 to 100% in 9 steps.
2. slow leds turn off and there is a delay.
3. Fast leds then ramp up form 0 to 100% two times, firslt by controlling the frequency and secondly by chaging the input voltage.

---

test_pmt_switches

To test the pmt_switches, use:

```
test_pmt_switches num_iterations
```

```
test_pmt_switches 100
```

This will send random data to all of the switches, and record any errors. The total sum of switches that were detected bad will be returned to the user via the response:

```
1+OK+num_errors
```

where num_errors is the number of switches whose setting was detected as being erroneous.

---

turn_slow_leds_on

To turn on the slow LEDs on in the LPC mezzanine, use the following command:

```
turn_slow_leds_on
```

---

update_ad7124

This command read the channles from the AD7124 ADC and processes the voltage and temperature for each channel. The ouput is displayed on the Serial Console.

```
COMMAND:  update_ad7124
RESPONSE: Ad7124+Update+Successful%2C+check+console+output%0A1+OK

SERIAL CONSOLE OUTPUT:
Resistance Ch[1] = 12164.6
Resistance Ch[2] = inf
Resistance Ch[3] = inf
Resistance Ch[4] = 12238.1
Ad7124 Update Successful
ChannelData[0] = 16777215
ChannelData[1] = 7569368
ChannelData[2] = 0
ChannelData[3] = 0
ChannelData[4] = 7544360
ChannelVoltage[0] = 2.5
ChannelVoltage[1] = 1.12792
ChannelVoltage[2] = 0
ChannelVoltage[3] = 0
ChannelVoltage[4] = 1.1242
ChannelTemp[0] = -1999
ChannelTemp[1] = 20.5182
ChannelTemp[2] = -142.13
ChannelTemp[3] = -142.13
ChannelTemp[4] = 20.3846
```

In the above example, the processed data is shown. There weren't any NTC element connected to channel 2 and Channel 3. Channel 0 is the ADC Ref_Out voltage and so its temperature can be ignored. The temperatue is degC. Resistance in Ohms and voltage is in Volts. The ChannelData is the raw data received from ADC.

---

write_ad7124_reg

This command writes to a specific register on the AD7124 ADC. The list of available registers can be found at Page 79 of the AD7124 datasheet from here. The register address needs to be specified in DECIMAL format. The data needed to be written to the ADC also needs to be specified in DECIMAL format.

Keep in mind that the commands only supports writing to 24 bits at max. if you enter a value what is 25 or more bits wide, then the bits after 24 will be discarded.

```
COMMAND:  write_ad7124_reg 2 5265658
RESPONSE: Ad7124+Register+Write+Successful%0A1+OK
```

The example above is used to write a value of 0x5058FA to register 2. Keep in mind that the commands does not check wheter the register that the data is written to is a write-able register. It the responsibltiy of the command issuer to make sure that the data and the address are valid and make sense.

---

write_mezzanine_ad5685

To write to the ad5685 DAC in the LPC mezzanine use the following command:

```
write_mezzanine_ad5685 dac_mask data
```

where dac_mask is a 4-bit mask that defines which DAC(s) to write to, where bit 0 is DAC A, bit 1 is DAC B, bit 2 is DAC C and bit 3 is DAC D. Data is an unsigned value that should be fit 14 bits.   The parameters can be in decimal or in hex (if prefixed by 0x).

---

write_mezzanine_dac

To write to the mcp4802 DAC in the LPC mezzanine use the following command:

```
write_mezzanine_dac gain_n shutdown_n val [dac]
```

wheren gain_n and shutdown_n are 1 or 0 and correspond to the bits thusly named in the mcp4802 datasheet, and val is an 8-bit value (in decimal or in hex (if prefixed by 0x)). The parameter DAC is the DAC that will be updated. By default this value is 0 and this means the DAC that corresponds to the slow LED amplitude. If the value of DAC is set to 1, this means the DAC that corresponds to the fast LED amplitude.

---

4. Example of using external trigger with emulated periodic transmission

The following is initiates periodic acquisition and UDP transmission of all 5 ADCs, 1024 samples per acquisition, 1000 triggers per second. The UDP packets are sent to IP of 192.168.0.29 port 1500, via ethernet interface 0 (“eth0”):

```
Command  :set_adc_mask 0x1F
Response :1 OK

Command  :get_adc_mask
Response :1 OK: 31

Command  :set_num_samples_per_packet 1024
Response :1 OK

Command  :get_num_samples_per_packet
Response :1 OK: 1024

Command  :start_acquisition 192.168.0.29 1500 0
Response :1 OK

Command  :set_emulated_trigger_speed 1000
Response :1 OK

Command  :enable_emulated_trigger
Response :1+OK

Command  :stop_acquisition
Response :1 OK
```

5. Example of using self trigger

The following is initiates self-trigger acquisition and UDP transmission of all 5 ADCs, 1024 samples per acquisition. The UDP packets are sent to IP of 192.168.0.29 port 1500, via ethernet interface 0 (“eth0”). Any channel on an ADC which crosses the trigger threshold will cause that ADC’s data to be transmitted via UDP. If channels on multiple ADCs are self triggered at the same time, then those ADCs will be sent via UDP.

```
Command  :set_adc_mask 0x1F
```

```
Response :1 OK

Command  :set_num_samples_per_packet 1024
Response :1 OK

Command  : set_self_trigger_threshold 100
Response :1 OK

Command  : set_self_trigger_on_falling_edge 0
Response :1+OK

Command  : start_periodic_acquisition_self_trigger 192.168.0.29 1500 0
Response :1 OK

Command  :stop_acquisition
Response :1 OK
```

6. Example of initiation of Capstone mode emulated periodic transmission (obsolete - use external trigger with emulated periodic transmission instead)

There is a mode of the capstone project that allows for periodic generation of triggers using the capstone logic. This is a working mode but not recommended since it does not allow for external trigger (a better approach is to use the external trigger option with emulated periodic transmission). The following is initiates periodic acquisition and UDP transmission of all 5 ADCs, 1024 samples per acquisition, 1000 triggers per second. The UDP packets are sent to IP of 192.168.0.29 port 1500, via ethernet interface 0 (“eth0”):

```
Command  :set_adc_mask 0x1F
Response :1 OK

Command  :get_adc_mask
Response :1 OK: 31

Command  :set_num_samples_per_packet 1024
Response :1 OK

Command  :get_num_samples_per_packet
Response :1 OK: 1024

Command  :set_trigger_freq 1000
Response :1 OK

Command  :get_trigger_freq
Response :1 OK: 1000

Command  :start_periodic_acquisition 192.168.0.29 1500 0
Response :1 OK

Command  :stop_acquisition
Response :1 OK
```

7. Example of Initiation of one-shot transmission (obsolete)

The following is initiates one-shot acquisition and UDP transmission of all 5 ADCs, 1024 samples per acquisition. The UDP packets are sent to IP of 192.168.0.29 port 4915, via ethernet interface 0 (“eth0”):

```
Command  :set_num_samples_per_packet 1024
Response :1 OK

Command  :set_adc_mask 0x1F
Response :1 OK

Command  :get_adc_mask
Response :1 OK: 31

Command  :set_up_one_shot_acquisition 192.168.0.29 4915 0
Response :1 OK

Command  :send_one_shot_pulse
Response :1 OK

Command  :send_one_shot_pulse
Response :1 OK

Command  :stop_acquisition
Response :1 OK
```

8. Debug and interactive command control of DSP processing

To open Jalisco, from the tsb/ip/scripts subdirectory run the command “source do_run_jalisco.cmd”

In Jalisco, press the button "Open Telnet Monitor", and in the window that opens press the "Connect" button in order to connect to port 12 of the card. This will show all TCP transactions (including commands and responses). This is a read-only port so commands on this port will have no effect.

In order to interactively control the card via TCP, press "Open Telnet to Card" in Jalisco. The right IP should be filled in, and change the port to 40 and press Enter. Then, press the "Connect" button. This will provide an interactive control terminal where commands can be input and responses observed.

9. Fast and Slow LED Operation

To control the LEDs, firstly we need to enable power to their controllers. So we start by sending these commands:

enable_ldo_en and ***enable_vccl_en***

After this, we can send commands specific to Fast and Slow Leds respectively.

Slow Led Commands

enable_mezzanine_dac

This command enables DAC we use to control the brightness of slow leds.

turn_slow_leds_on

This will turn on the logic that enables the slow leds.

set_slow_led_dac dac_val

This commands lets the user to control the brightness of the slow leds. The parameter dac_val can be any integer from 0 to 1023. With 0 being most bright and 1023 being no brightness.

turn_slow_leds_off

This commands turns off the logic that enables the slow leds

disable_mezzanine_dac

This commands will disable the dac that is used to control the slow leds brightness.

Fast Led Commands

enable_fast_led

This commands turns on the logic that controls the fast leds

Now the user can select the fast led mode, the info about that is given below:

The fast LED operation is as follows. The fast LED pulse is always 10ns wide. Fast LED modes are:

0: Periodic (unsynchronized to trigger)

1: One-Shot (unsynchronized to trigger)

2: Synchronized to trigger

In periodic mode, the fast LED pulses are generated with a user-controlled period. In one-shot mode, the fast LED is generated with as user-controlled one-shot pulses. When synchronized to trigger, the fast LED is generated shortly after the trigger (exact delay TBD via measurement but expected to be no more than +-50 nS, with jitter of 10ns).

When in synchronized to trigger mode, any type of trigger will cause the fast LED to fire. This can be an external trigger or an emulated trigger.

Once that command is sent you can proceed with these commands:

set_fast_led_dac dac_val dac_gain

This command lets the user to control the brightness of the fast leds. The parameter dac_val can be any integer in range 0-1023. The value 0 corrsponds to full brightness and value 1023 corresponds to no brightness. The dac_gain can be set to 1

To control which fast led lights up, try using the command ***set_fast_led_mask mask***

Here the parameter mask is the decimal equivalent of the the three leds ON position in binary state. The mask can take in values between 0-7. Where 0-> 0 0 0 and 7-> 1 1 1 so all three leds light up in mask 7.

Finally, to turn off the LED system, we use the folowing commands:

This commmand disables the logic that controls the fast led

disable_fast_led and ***disable_ldo_en***

This command disables the power to the ICs that control the leds.

Disabling the LDO will also cut off power to comparators and other DACs, so use it wisely.

Below are some more examples of sample commands that can be sent to control the leds.

Examples of manipulating the fast LED are shown here:

```
INCOMING: enable_mezzanine_dac
INCOMING: 1+OK
OUTGOING: write_mezzanine_dac 0 1 0
INCOMING: 1+OK
OUTGOING: enable_fast_led
INCOMING: 1+OK
OUTGOING: select_periodic_fast_led
INCOMING: 1+OK
OUTGOING: set_fast_led_speed 1000
INCOMING: 1000
OUTGOING: set_fast_led_speed 10000
INCOMING: 10000
```

```
OUTGOING: set_fast_led_speed 10000
INCOMING: 10000
OUTGOING: set_fast_led_speed 100000
INCOMING: 100000
OUTGOING: get_fast_led_mode
INCOMING: 0
OUTGOING: SELECT_SYNC_TO_EXTERNAL_FAST_LED
INCOMING: 1+OK
OUTGOING: get_fast_led_mode
INCOMING: 2
OUTGOING: SET_EMULATED_TRIGGER_SPEED 1000
INCOMING: 1000
OUTGOING: enable_emulated_trigger
INCOMING: 1+OK
OUTGOING: SET_EMULATED_TRIGGER_SPEED 10000
INCOMING: 10000
OUTGOING: select_one_shot_fast_led
INCOMING: 1+OK
OUTGOING: get_fast_led_mode
INCOMING: 1
OUTGOING: do_one_shot_fast_led_pulse
INCOMING: 1+OK
OUTGOING: do_one_shot_fast_led_pulse
INCOMING: 1+OK
OUTGOING: do_one_shot_fast_led_pulse
INCOMING: 1+OK
```

10. UART Register File Raw API

Usage of the UART register file raw API is not recommended, for various reasons, which include obscure syntax, lack of abstraction, and the fact that the UART register file addresses may change from one compilaiton to another if UART register files are added or removed in the HDL. Nonetheless, if usage of the raw API is desired, this is documentation.

The UART register files each have unique addresses, which are composed of two numbers: the UART number and the Secondary Address. Both are non-negative numbers. It is necessary to reference both the correct UART Number and Secondary Address for each UART register file in order to correctly access that register file via the raw API. By convention, UARTs are referred to in Jalisco via the addressing scheme x_y, where x is the UART number and Y is the secondary address, i.e. X_Y.

The following commands are available for controlling the UART Register File Raw API:

---

get_uart_regfile_display_map

The command get_uart_regfile_display_map will return a map of the UART names to the UART number and Secondary Addresses, in the following way:

```
COMMAND: get_uart_regfile_display_map
RESPONSE:
adc3424_0+61+0+adc3424_1+62+0+adc3424_2+63+0+adc3424_3+64+0+adc3424_4+65+0+adc3424
_dma_0+56+0+adc3424_dma_1+57+0+adc3424_dma_2+58+0+adc3424_dma_3+59+0+adc3424_dma_4
+60+0+ads4249_spidiag+7+2+beam_ext_niosdac+7+1+beam_tailw+7+4+beam_udpsplt+7+3+bea
m_udptop+7+0+gptoplevel+0+0+hdc1080_1+84+0+idt_8t49n241+70+0+idt_8t49n241_i2c+69+0
+ldo1+76+0+ldo2+77+0+ldo3+78+0+ldo4+79+0+ldo5+80+0+ldo6+81+0+ldo7+82+0+ldo8+83+0+l
ps25hb+72+0+lps25hb_i2c+71+0+mac_eeprom+68+0+mac_eeprom_i2c+67+0+packet_diag_to_ts
e_mac+86+0+packet_diag_udp_2_split+85+0+packet_diag_wave0+87+0+sfp_tse+88+0+tmp1+7
3+0+tmp2+74+0+tmp3+75+0+tse+32+0+udp_inserter_0+34+0+udp_inserter_1+35+0+udp_inser
ter_2+36+0+udp_inserter_3+37+0
```

Each UART register file name is followed by its UART number and secondary address, both in decimal. While UART numbers and secondary addresses may change from compilation to compilation if register files are added or removed from the design, the register file names should stay the same. Therefore, the get_uart_regfile_display_map command should be used in order to map the register file name to the UART number and secondary address, in order for any subsequent code that uses the raw API to be independent from such changes.

---

uart_read_all_ctrl

The command uart_read_all_ctrl reads all the control registers of a register file. The syntax is:

```
uart_read_all_ctrl x y
```

where x, y are the uart number and secondary address respectively, in decimal. For example:

```
COMMAND: uart_read_all_ctrl 7 3
RESPONSE:
0+0+1+1+2+131072+3+4+4+138800+5+2+6+512+7+1024+8+0+9+0+10+0+11+0+12+0+13+0+14+0+15
+0+16+0+17+0+18+0+19+0+20+0+21+0+22+0+23+0+24+0+25+0+26+0+27+0+28+0+29+0+30+0+31+0
+32+0+33+0+34+0
```

The response is a series of register value pairs, separated by a space (after query syntax decoding, or a + before such decoding), both in decimal.

---

uart_read_all_ctrl_desc

The command uart_read_all_ctrl_desc returns the control register descriptions. The command syntax is:

uart_read_all_ctrl_desc x y

where x, y are the uart number and secondary address respectively, in decimal. For example:

```
COMMAND : uart_read_all_ctrl_desc 7 3
RESPONSE:
0+%22StreamerRst%22+1+%22StreamerEna%22+2+%22ketLengthInWords%22+3+%22packet_type%
22+4+%22PackWodBfrNew%22+5+%22test_packt_ctrl%22+6+%22img_width%22+7+%22img_height
%22+8+%22clog2_pkts2width%22+9+%22sel_int_udp_wrds%22+10+%22intudpwrd0000%22+11+%2
2intudpwrd0001%22+12+%22intudpwrd0002%22+13+%22intudpwrd0003%22+14+%22threshold00%
22+15+%22intudpwrd0100%22+16+%22intudpwrd0101%22+17+%22intudpwrd0102%22+18+%22intu
dpwrd0103%22+19+%22threshold01%22+20+%22intudpwrd0200%22+21+%22intudpwrd0201%22+22
+%22intudpwrd0202%22+23+%22intudpwrd0203%22+24+%22threshold02%22+25+%22intudpwrd03
00%22+26+%22intudpwrd0301%22+27+%22intudpwrd0302%22+28+%22intudpwrd0303%22+29+%22t
hreshold03%22+30+%22intudpwrd0400%22+31+%22intudpwrd0401%22+32+%22intudpwrd0402%22
+33+%22intudpwrd0403%22+34+%22threshold04%22
```

After URL decoding, the response is:

```
0 "StreamerRst" 1 "StreamerEna" 2 "ketLengthInWords" 3 "packet_type" 4
"PackWodBfrNew" 5 "test_packt_ctrl" 6 "img_width" 7 "img_height" 8
"clog2_pkts2width" 9 "sel_int_udp_wrds" 10 "intudpwrd0000" 11 "intudpwrd0001" 12
"intudpwrd0002" 13 "intudpwrd0003" 14 "threshold00" 15 "intudpwrd0100" 16
"intudpwrd0101" 17 "intudpwrd0102" 18 "intudpwrd0103" 19 "threshold01" 20
"intudpwrd0200" 21 "intudpwrd0201" 22 "intudpwrd0202" 23 "intudpwrd0203" 24
"threshold02" 25 "intudpwrd0300" 26 "intudpwrd0301" 27 "intudpwrd0302" 28
"intudpwrd0303" 29 "threshold03" 30 "intudpwrd0400" 31 "intudpwrd0401" 32
"intudpwrd0402" 33 "intudpwrd0403" 34 "threshold04"
```

which as can be seen is a sequence of register number (in decimal) and decscription (in quotes).

---

uart_read_all_status

The command uart_read_all_status reads all the status registers of a register file. The syntax is:

```
uart_read_all_status x y
```

where x, y are the uart number and secondary address respectively, in decimal. For example:

```
COMMAND: uart_read_all_status 10 3
RESPONSE:
0+1048577+1+0+2+0+3+0+4+1+5+262411+6+267+7+1+8+0+9+1+10+0+11+1+12+0+13+0+14+0+15+0
+16+1+17+1541
```

The response is a series of register value pairs, separated by a space (after query syntax decoding, or a + before such decoding), both in decimal.

uart_read_all_status_desc

The command uart_read_all_status_desc returns the status register descriptions. The command syntax is:

```
uart_read_all_status_desc x y
```

where x, y are the uart number and secondary address respectively, in decimal. For example:

```
COMMAND : uart_read_all_status_desc 7 3
RESPONSE:
0+%22splitter_state%22+1+%22packet_count%22+2+%22packet_wrd_count%22+3+%22total_wr
d_count%22+4+%22avst2udp_ctrl%22+5+%22avst_to_udp_data%22+6+%22calc_pkt_length%22+
7+%22avst_input_out%22+8+%22avst_input_data%22+9+%22avst2splt_ctrl%22+10+%22avst2s
plt_data%22+11+%22testpkt_ctrl%22+12+%22testpkt_data%22+13+%22x1_y1%22+14+%22data_
word_cnt%22+15+%22frameID%22+16+%22internal_udp_set%22+17+%22inst_params%22
```

After URL decoding, the response is:

```
0 "splitter_state" 1 "packet_count" 2 "packet_wrd_count" 3 "total_wrd_count" 4
"avst2udp_ctrl" 5 "avst_to_udp_data" 6 "calc_pkt_length" 7 "avst_input_out" 8
"avst_input_data" 9 "avst2splt_ctrl" 10 "avst2splt_data" 11 "testpkt_ctrl" 12
"testpkt_data" 13 "x1_y1" 14 "data_word_cnt" 15 "frameID" 16 "internal_udp_set" 17
"inst_params"
```

which as can be seen is a sequence of register number (in decimal) and decscription (in quotes).

uart_regfile_ctrl_read

The command uart_regfile_ctrl_read reads a single control register. The syntax of the command is:

uart_regfile_ctrl_write uart_number register_number secondary_address

where uart_number and secondary address are decimal, and register_number is hexadecimal. For example the command:

```
uart_regfile_ctrl_read 7 7c 5
```

will read control register 0x7C from UART 7_5.

The result of this command is a decimal number containing the result of the read.

---

uart_regfile_ctrl_write

The command uart_regfile_ctrl_write writes a single control register. The syntax of the command is:

uart_regfile_ctrl_write uart_number register_number value secondary_address

where uart_number and secondary address are decimal, and register_number and value are hexadecimal. For example the command:

uart_regfile_ctrl_write 10 1f abcdef 0

Will write 0xabcdef to register 0x1f in UART register file 10_0.

This command returns a newline ('\n') as a response.

---

uart_regfile_status_read

The command uart_regfile_status_read reads a single status register. The syntax of the command is:

uart_regfile_status_read uart_number register_number secondary_address

where uart_number and secondary address are decimal, and register_number is hexadecimal. For example the command:

```
uart_regfile_status_read 7 1d 6
```

will read status register 0x1d from UART 7_6.

The result of this command is a decimal number containing the result of the read.

---

uart_write_multiple_ctrl

The command uart_write_multiple_ctrl writes multiple control registers. The command syntax is:

uart_write_multiple_ctrl x_y register_0,value_1[,register_1,value_1,....]

where x, y are the uart number and secondary address respectively, and register_n,value_n and register number and value, respectively. The UART number, secondary address, and register numbers are decimal while the values are in hexadecimal. For example the command:

```
uart_write_multiple_ctrl 7_3
0,0,1,1,2,20000,3,4,4,21e30,5,2,6,200,7,400,8,0,9,0,10,0,11,0,12,0,13,0,14,0,15,0,
16,0,17,0,18,0,19,0,20,0,21,0,22,0,23,0,24,0,25,0,26,0,27,0,28,0,29,0,30,0,31,0,32
,0,33,0,34,0
```

Will write the appropriate register/value pairs.

This command returns a newline ('\n') as a response.

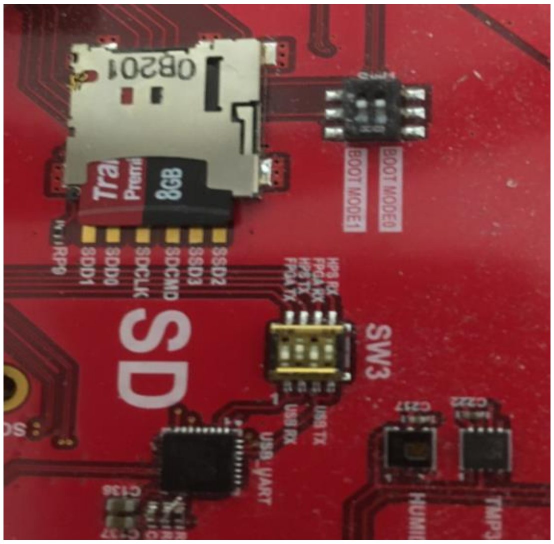
---

11. Booting from the SD Card

The following explains how to use the boot from SD

11.1. DIP Switch Settings

Set up the both the BOOT MODE and SW3 DIP switches as shown in the following image:



11.2. Initial Programming of the SD card

1. Turn off the board
2. Take out the SD card
3. The SD card needs to be formatted as FAT32
4. Copy the files in the deploy_sd subdirectory to the root directory of the SD card:
5. Connect an Ethernet cable to the one of the Ethernet connectors
6. Insert SD card to board and turn it on. To monitor the boot process, connect via UART to card (via USB connector on the card). It is recommended to use TeraTerm and the settings contained in the file xu1.INI, or if that is not possible, connect with settings: 115200 baud, 8 bit, no parity, 1 stop bit.
7. Once the card boots, the command server will run automatically as a background process.

11.3. Updating Both the Software and Firmware Images on the SD Card

1. Open an SFTP session to the board, with user name root and password root (using command line SFTP or WINSCP or similar client). Note the RSA key of the board changes after each reboot so accept the new RSA key if needed when connecting via SFTP.
2. Navigate to /mnt/sd-mmcblk1p1
3. Transfer the files in the deploy_sd subdirectory to /mnt/sd-mmcblk1p1 (overwriting when needed)
4. Reboot the card by doing one of the following:
    a. In the command prompt in the UART terminal, type the command "reboot"
    b. Open an SSH console to the card, user root and password root, and type the "reboot" command

---

12. Booting from the eMMC Card

The following explains how to use the boot from eMMC

12.1. Preparation of eMMC

1. In the petalinux "config" file, located in tsb/ip/petalinux/project-spec/configs/config, make sure the following are set:

    CONFIG_e2fsprogs=y

    CONFIG_e2fsprogs-mke2fs=y

    CONFIG_e2fsprogs-e2fsck=y

    CONFIG_tar=y

    CONFIG_dosfstools=y

2. Regenerate Petalinux

3. Boot up the petalinux distribution on the SDcard, once logged in type the following

fdisk /dev/mmcblk0

4. At the prompts, hit the following to create a new primary partition and make it 2048MB in size:

n p 1 <default> +2048M

5. Change the type of the partition to 'c' (win95 LBA)

t c

6. Create the second primary partition and accept the size defaults to make it take the rest of the eMMC

n p 2 <default> <default>

7. Change the type of the partition to '83' (linux)

t 2 83

8. Write the partition table to disk

w

9. Fdisk should write the partition table and close, if it has not, hit 'q'

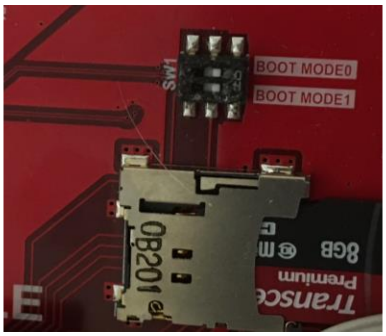
10. Reboot via the "reboot" command

11. Format the two new partitions

mkfs.vfat /dev/mmcblk0p1

mkfs.ext4 /dev/mmcblk0p2

12. Reboot.
13. Petalinux will see and auto-mount the new partitions under /media and /mnt
14. Copy the files in the deploy_emmc subdirectory to the root directory to the /mnt/sd-mmcblk0p1 directory
15. Set BOOT MODE DIP Switch Settings for eMMC, which are shown here:



16. Reboot using the "reboot" command
17. The board should now boot from emmc

12.2. Updating Both the Software and Firmware Images on eMMC

1. Open an SFTP session to the board, with user name root and password root (using command line SFTP or WINSCP or similar client). Note the RSA key of the board changes after each reboot so accept the new RSA key if needed when connecting via SFTP.
2. Navigate to /mnt/sd-mmcblk0p1
3. Transfer the files in the deploy_emmc subdirectory (overwriting when needed)
4. Reboot the card by doing one of the following:
   a. In the command prompt in the UART terminal, type the command "reboot"
   b. Open an SSH console to the card, user root and password root, and type the "reboot" command