Holography Backend User Guide

Contents index

Figures index

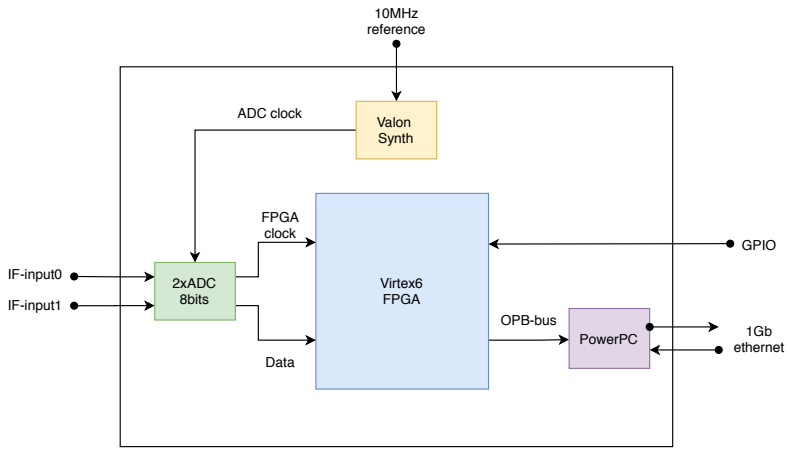


Figure 1: Basic diagram of the ROACH system

1. Introduction

The CCAT-p holography back-end consist in a vector voltmeter that measure magnitude and phase between two inputs and put a timestamp in each measure.

The development of this project was made in the platform ROACH2 from CASPER. The most general diagram of the hardware is shown in the figure 1, where we emphasize the 4 important subsystem used by the vector voltmeter.

The basic operation could be reduce in the following steps:

- The Valon synthesizer generate the clock signal and feeds the ADCs. This clock signal could be reference to an external 10MHz reference signal.
- The ADC has an internal structure that enables to sample at two times the valon frequency. So $Sampling\,Frequency = 2 \cdot Valon\,Frequency$
- The ADC gives the clock signal to the FPGA which is used to clock the entire board. So if you lock the Valon the whole system is locked to the 10MHz external reference.
- The samples taken by the ADCs gets into the Virtex6 and are processed with standard digital signal techniques and wrote values in registers which are mapped to the PowerPC microprocessor.
- The PowerPC runs a telnet server which could be access by 1Gb Ethernet connection. This server allows to request writes and reads to the mapped registers in order to interface with the FPGA.

The next subsection gives a description of the ROACH subsystems and the ways to interact with them. We developed some codes to hide the complexity of the write and read procedure and to reinterpret the binary data collected. If you don't want to get into details jump to the summary at the end of this section.

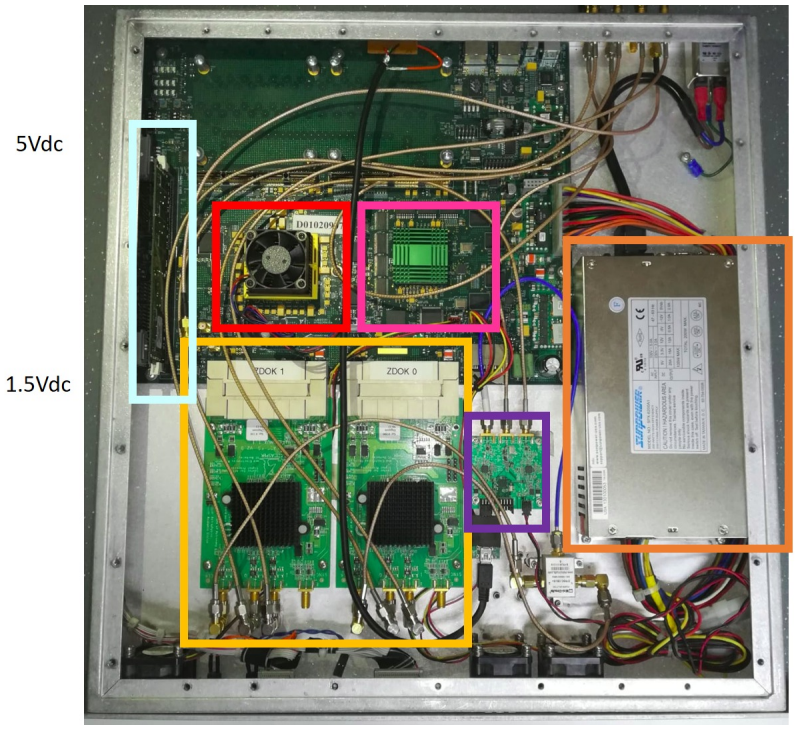


Figure 2: Example of the inside of the ROACH2 system. Inside the red rectangle is the Virtex6 FPGA, in pink rectangle is located the PowerPC, the yellow rectangle contains the two ADCs, the orange rectangle contains the power supply and in the white rectangle contains the DRAM and the GPIOs of the FPGA.

1.1. Valon synthesizer

The Valon synthesizer has three ports, two acts as independent channels with frequencies in the range (137.5-4400)MHz and one port is used to receive an external reference. Keep present that the synthesizer needs to be told if its going to work with an internal or external reference. If this is not done, the synthesizer will work with the previous configuration.

To program the synthesizer it has a mini-USB connection running an UART protocol that receive commands using standard serial programs like minicom.

In our codes we use the nrao python package to program the valon synthesizer. You could find the python package here.

Although the case says that you should supply the Valon with 6Vdc, reading the datasheet you could found that it is also allowed to be supplied by 5Vdc which is a more common voltage. We supply the valon with 5Vdc with the power supply inside the ROACH.

Some recommendation are:

1. You must ensure that 0dBm are given at the ADC input, so keep that in mind if you want to change the power level.

2. When changing the system operation frequency remember that the sampling frequency at the ADC is twice the valon frequency.

3. If you are using external reference check that the reference signal is connected.

Figure 3: Valon 5008 synthesizer.

4. If you are not getting any response of the valon, check which tty/port is used by the Valon and which one is used by the python code.[1]

1.2. ADC's

The ADCs are based in the EV8AQ160 ADCs. The datasheet could be found here. The inputs of the ADCs are in the back of the ROACH system and are labeled.

The ADCs is composed by 4 ADCs with interleaved sampling time, so given a certain clock signal the first of those 4 ADC receive the clock without any change, the second one receive the clock with a 90° phase, the third with a 180° and the fourth with 270°.

In order to ensure that the synchronization between the internal ADCs you must perform three calibrations before a measurement takes place.

- MMCM Calibration: Acording to NRAO *The MMCM calibration is used for correcting the phase of the capture clock between the four cores. Depending on where the rising edge of the capture clock lies, glitches in the ADC samples can occur. The ADC has the ability to provide test ramps that they use to detect glitches and adjust the MMCM appropriately to eliminate them.* You must perform this calibration every time the FPGA is reprogrammed or when a change in the clock rate takes place.

- OGP (Offset Gain Phase) calibration: This calibration makes a fine calibration writing internal registers in the four ADCs. To do it, the ADC is feed with a known sinusoidal signal and using least squares the offset, gain and phase corrections are calculated and using an iterative method the calibration takes place. . The values of the OGP calibration only depend on the board clock rate, so it must be perform when a clock change takes place. Here you could find an application note about the OGP.

- INL (Integral Non Linearity) Calibration: The ADC has internal registers to correct the non linearity of the cores. In order to do it you must feed the ADCs with a known signal. This values need to be charged in every power cycle.

Here you could find a note of the characterization of this ADC made by the SMA back-end team.

[1] The python code uses the /dev/ttyUSB0

Our codes to calibrate the ADCs are based on the calibration repository of nrao. After every calibration the obtained values are saved to have the possibility to upload the corrections instead to actually make the calibration each time.

To summarize, **when you upload a model into the ROACH you must make MMCM and upload (or make) OGP and INL. If you change the clock it is recommendable make all the calibrations.** [2]

Other important thing to remember is that the maximum power level at the input of the ADCs should be less than -5dBm. Above this value the ADC starts to saturate.[3]

There is one more little tweak to mention in the ADCs utilization. Because the platform that we are dealing is a FPGA it has to route all the inputs to the logic that performs some operation into the data, there exists a delay between the two ADCs inputs given by the route that the Xilinx compilation selects, this is translated to a constant phase difference in the input. Like the delay is introduced in terms of FPGA clock cycles could be fixed using a programmable delay, so we create a script to calibrate the ADCs phase difference. In this script you must provided a 10MHz signal at both inputs using an 0° splitter.

This synchronization between the ADCs is not mandatory because in our application we are only interested in the relative phase difference between the inputs, but could lend to different results in the absolute phase.

1.3. PowerPc

The PowerPC runs a minimalist version of Linux called BORPH and the majority of the ash commands are given by the busybox software.

Almost all the system is write protected with certain remarkable exceptions, the most important one is the temporary directory located in /var/tmp that has 378.2Mbytes of space available, but gets erased every time the ROACH is turn off.

Another important writable file is */dev/roach/mem* where the registers of the FPGA gets mapped in order to be written or read.

The standard method to communicate to the PowerPC it is using a telnet session. The system will ask for an username, the default username it is **root** and it doesn't ask any password.

If you have problems to connect via telnet you could also get inside the PowerPC using a type B USB cable an using a UART program like minicom. Then you could modify the default IP address modifying the /etc/rc.local file to set your preferred interface. Another sanity check if you are unable to connect, it is to see which port are you connected to, because in the back panel of the ROACH there are two 1Gbe ports and only one gives access to the PowerPC.

In the PowerPC boot stage two telnet servers starts to run. The first one at the port 23, which is the default telnet port, lets you get inside the BORPH system. The second telnet connection available is located at the 7147 port and runs the Karoo Array Telescope Control Protocol (katcp). This protocol is used as a link between the orders send by a desktop computer and it execution inside the FPGA. So every read and write operation pass through this protocol.

Here are the basic commands to control to interface with the ROACH system. In order to make more easy the use of this protocol there exists a python package corr which translate katcp commands to simple python statements.

[2] By the way this is mandatory when you want to make a precise measurement, the system will work without the calibration but it doesn't achieve it best performance.

[3] You could look at the snapshot of the ADC input to see if there is saturation.

As an example, the standard reading procedure consists in:

1. The ADC capture some samples.
2. The data is processed inside the FPGA and gets saved into internal registers which are mapped to the PowerPC /dev/roach/mem file.
3. Using the corr package we send a request to read the data through telnet.
4. The system use the mmap command to open the /dev/roach/mem file and reads the correspondent addresses requested by katcp.
5. The PowerPC sends the response using the katcp protocol.
6. The desktop computer receives the response using the corr package.

1.4. Virtex6 FPGA

Vector voltmeter subsystem

The basic diagram implemented in the FPGA consist in two spectrometers, one correlator between the two input and a separate subsystem who delivers the timestamp.

The clock signal for the whole FPGA system is locked to the ADC clock and corresponds to 1/8 of the Valon frequency (or 1/16 of the sampling frequency of the ADCs). It is important to mention that the system was compiled using a 135MHz clock frequency, so the Xilinx toolflow ensure that this is the maximum working frequency, beyond this threshold could be path delays and the system wouldn't behaves as it should. Using lower frequencies is not a problem.

To be in the right range the maximum frequency that you could set in the Valon is 1080MHz.

The ADCs gives 8 bits for each sample and like we fix the clock at 1/16 of the sampling frequency in each clock cycle the system has 16 samples available. The theoretical SNR of a 8bit ADC is 49.9dB.

To increase the dynamic range we use a technique named oversampling, it consist in sampling over the Nyquist sampling frequency and then decimate using for that propose a FIR filter. We decimate by a factor of 16 which gives an improvement of 12.04dB in the SNR.

After the decimation stage we put a window in our data to trait the leakage problem in the DFT implementations. The window is implemented using a polyphase architecture based in the hamming window. This polyphase architecture enables to reduce significantly the side lobes and maintaining the mainlobe width of the window, this is translated into a better spectral localization of the signal. You could find more information about the PFB here.

Then we calculate a pipelined Fast Fourier Transform (FFT) over the windowed data. That the FFT is pipelined means that after the initialization the FFT delivers one output channel per clock cycle. The FFT size is 16384, consider that the input are real values, only 8192 channels contains useful information. If we run the system at 135MHz this give us 67.5MHz of bandwidth with 8.24kHz between two adjacent channels.

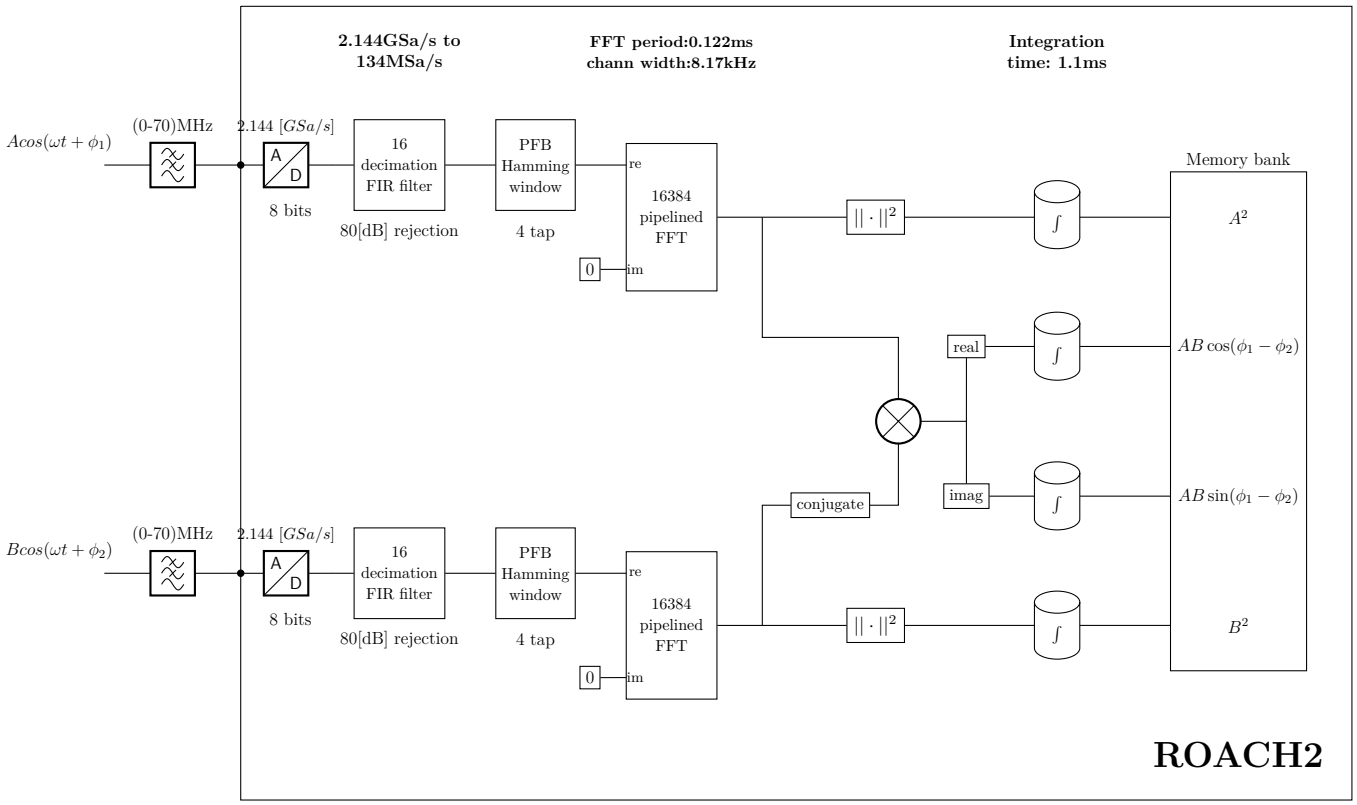


Figure 4: Vector voltmeter block diagram.

The calculation of the FFT involves a processing gain that appears thanks to the coherent sum that is used to perform the FFT, in our case this processing gain is translated in an enhancement of 39.13 dB in the SNR value.

With the spectral data we obtain the power of the signal calculating the square magnitude of the complex output of each FFT channel. Note in the diagram in 4 that we don't calculate the square root, because when we calculate the values in dB the square root doesnt affect the ratio between the inputs.

In the same way, we obtain the cross correlation calculating the complex conjugate of one FFT output and then multiplying the other FFT output. This complex multiplication is synchronized to be performed in the same channel on both FFT outputs.

After the calculation of the auto and cross correlation there is an accumulation stage which is equivalent to the integration in a telescope. We add successive outputs in order to have an "average value" of the power and the correlation. The number of accumulations performed could be programmable by setting one register in the FPGA.

Finally there are a bunch of dual port Block rams where one port is wrote by the FPGA and the other port goes to the PowerPC to be read. We have devided the Brams in the following options.

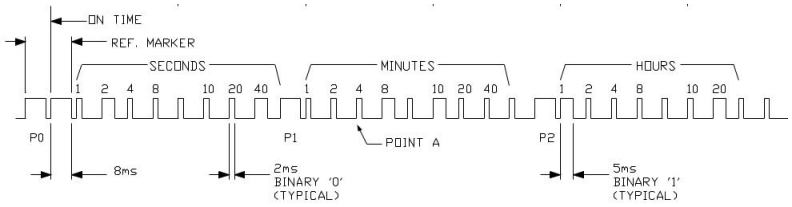


Figure 5: Example of a portion of IRIG packet.

- Spectrum Brams: Here we storage the accumulated power for each ADC channel. This Brams is continuously updated, so its function is to look at the real time spectrum in an animation. .
- Correlation Bram: Here we store the real and imaginary part of the correlation. Its continuously updated, so its function is to look at the real time correlation.
- One channel Brams: The system enables to save one FFT channel power and correlation inside those brams. Setting the right registers you could see the time evolution of this channel. This memories are intended to see the evolution in a remote desktop, because they don't ensure that no data is lost in the middle of the operation.
- PowerPC brams: Those brams saves the magnitude, correlation and time stamp of one single channel. It consists in two sub-banks that gets saved in the PowerPC temporary directory. When one sub-bank is full a flag is raised and the data is wrote in the other sub-bank. In the meanwhile, the first sub bank is read by the PowerPC and rise down the flag, then the systems wait until the second sub-bank is full and then switch again to the first one and repeat the operation for a given time.
- ADC snapshot: Memory located right after the ADC, it could be used to look at the time domain behaviour of the input signal. Is good to take a look if the ADC is saturated by the input signal.

Of those brams the most important group is the PowerPC brams, which are the ones that will save the final measurements in a holography campaign.

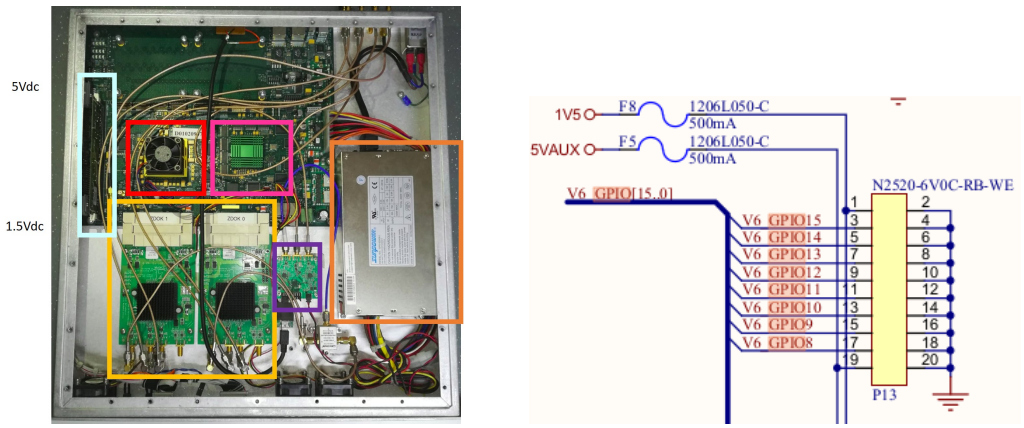
Again, those memories could be accessed and interpreted using our python codes.

Timestamp subsystem

The timestamp subsystem its based in the IRIG-B000 protocol. This protocol consists in 100 pulses delivered in 1 second. This pulses encoded the information in the pulse duration and the location of the pulse in the packet. So for example a pulse with a duration 2ms means a 0 and a 5ms duration means 1. The location encodes the magnitude of the value, so for example the first 4 pulses represents the units of seconds of the packet, the next 3 pulses represents tens units of seconds, etc. This type of representation is called Binary Coded Decimal (BCD). As its not a natural way to count seconds we translate this value to Time Of the Year (TOY) format.

As input for the IRIG code we use the GPIOs of the FPGA. **The GPIOs interface is LVCMOS15 which means that the input voltage should be 1.5V.** It means if you have an input

of 5V you should have a previous stage of voltage translation. The GPIOs are located in the bank 13, to access to this location first you need to take out the DRAM (white rectangle in 6 and the GPIO bank is in one side. If you modify it take a look in which direction is located the bank, the 1.5V is facing the front and the 5V of the bank is facing the back of the ROACH.



(a) ROACH2 board image. (b) GPIO schematics.

Figure 6: The GPIO location is in the bank 13 which its on the side of the memory card (white rectangle in a). The schematic for the 13 bank is in b)

The main algorithm to obtain the time from the master clock basically consists in look for the head of the package, then counts the pulses in clock cycles and extract the binary information. With the BCD values we translate it into TOY protocol. The time is decoded only in the first packet, then we use the fact that the IRIG-B000 packet last 1 second and has 100 pulses, so in order to increase in one second the TOY value we should only count 100 pulses.

The sub-second time is accounted by a free running counter that is reset when a new second start, so the sub seconds are measured in the FPGA clock periods. This also enable us to look at the time drift, because at the end of each second we could compare the value of the free running counter with the ideal value and determine if the system is locked.

The complete diagram of the algorithm is in the figure 7.

1.5. Note about the data format

The power values are unsigned long long, the imaginary and real part of the correlation are signed long long, the seconds and subseconds are unsigned int.

So for example if we read the magnitude of the spectrum we should read $8192 * 8$ bytes of the bram and then translate the binary to 8192 values. We made this by using the packet struct.

The real and imaginary part are saved interleaved in the memory, this means that to read the whole spectrum correlation we must read $8192 * 8 * 2$ bytes, then we should translate this to $8192 * 2$ signed long long values and separate odd and even index values, where the odd values are the real part and the even are the imaginary part.

Something similar happens for the single channel data that is saved in the PowerPC. The package of the PowerPC has the following format.

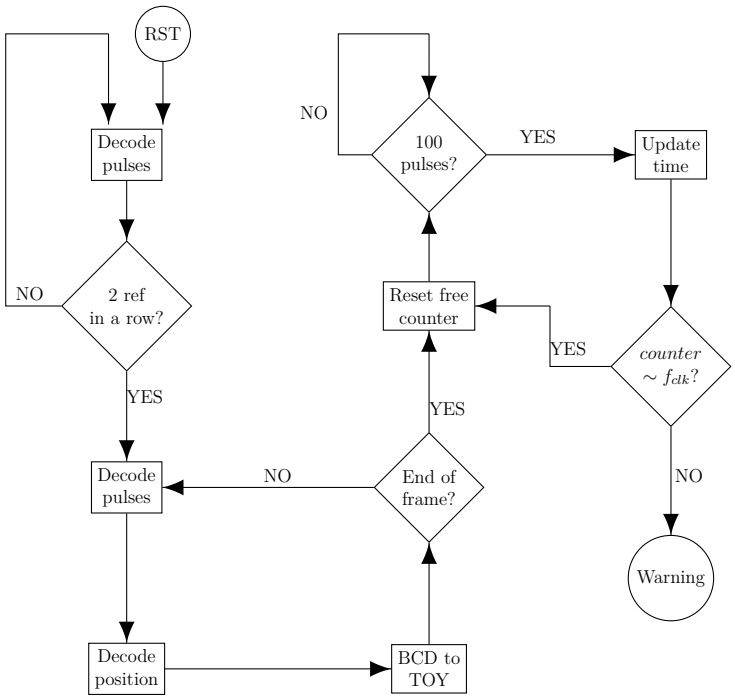


Figure 7: Flux diagram of Timestamp subsystem.

1. $8192 * 8$ bytes correspondent to *>8192Q* values of the ADC0.
2. $8192 * 8$ bytes correspondent to *>8192Q* values of the ADC1.
3. $8192 * 8 * 2$ bytes correspondent to *>16384q* values of the real and imaginary part of the correlation. The data is interleaved and the odd values correspond to the real part and the even to the imaginary.
4. $8192 * 4 * 2$ bytes that correspond to *>16384I* values, where the data is the timestamp. The odd values are the fraction of seconds and the the even values are the date time in TOY format.

1.6. Summary

1. You could set the Valon synthesizer frequency using a mini usb port with UART commands and we use this codes to program it. You could also set if the reference is internal or external.
2. The sampling rate of the ADCs is the double than the frequency set at the synthesizer. The clock at the FPGA is derived by the ADC, so the whole system is locked to the same reference.
3. You must perform 3 calibrations at the ADCs to achieve it best performance.
4. The maximum power at the input of the ADCs are -5dBm.
5. You could connect to the PowerPC using a telnet connection. The FPGA registers are mapped into /dev/roach/mem. The directory /var/tmp is writable and allow us to store the vector voltmeter data.
6. You could also connect to the port 7147 where you could send commands to read and write register of the FPGA. The package corr gives a nice python interface to the protocol that is running in this port.
7. The vector voltmeter at the FPGA is made by a correlator of two signals using for that propose a 16384 FFT, having only 8192 channels with information.
8. You could control the number of accumulation (which is related to the integration time) by setting one FPGA register.
9. You have the following interfaceable memories within the FPGA:
   - Full spectrum of the two ADC inputs.
   - Correlation of the whole bandwidth.
   - Look the magnitude and correlation for one channel.
   - Fully save one channel data inside the PowerPC temporary directory.
   - Look at the input of both ADCs in the time domain.
10. For the timestamp we use the IRIG-B000 protocol. As gateway we use the GPIO of the FPGA. This subsystem is only based in the FPGA clock cycles, so you should give it to the system as input variable.
11. You could set a threshold to determine if the system is locked. There also exists register to obtain the current time.

2. Python package

The package we created is intended to interface with the ROACH in a easy way, internally uses primarily the corr package which allows the reading and writing of every register in the FPGA only specifying the name given to the register, the value of address you want to read/write and the value you want to write into if corresponds.

In the following lines we present a standard corr code to get connected to the ROACH, upload a model into the FPGA, write a register to reset the state of the system and finally read the binary data from a memory and translate it to actual values.

```
import corr
import struct, time

roach_ip = '192.168.0.40'                    #roach IP
bofname = 'vv_casper.bof'                     #FPGA model to be upload
fpga = corr.katcp_wrapper.FpgaClient(roach_ip) #generate the connection with the ROACH

fpga.upload_program_bof(bofname, 3000) #upload the bitstream file and program the fpga
time.sleep(1)
fpga.write_int('rst',1); fpga.write_int('rst',0) #write the register 'rst' to 1 and then to 0, usefull
 ↪ to take the fpga to an inital state
raw_data = fpga.read('powA', 8192*8,0) #read 8192*8 bytes of the memory named 'powA'
data = struct.unpack('>8192Q', data) # translate data to 8192 unsigned long long data.
```

This is the basis of the what we hide in our codes. In the following subsection we present some of the main functions in the python package vv_calan.

If you don't care about the details you could jump into the subsection 2.5 where a short description of all the functions is given and then a sample code is given.

2.1. Initialization

The following codes generate the roach object which have all the methods to interface with the ROACH system. The init function needs the ROACH IP address the bitstream file path and the frequency that is given by the valon synthesizer.

```
from vv_calan import vv_calan
roach = vv_calan(roach_IP, bofname, valon_freq)
```

If you don't know the actual value of the valon frequency you could connect to the valon miniUSB port and run:

```
roach.get_valon_status()
```

It should print out the actual frequency, and then you could re-run the code that generates the roach object with the correct frequency value. It is important to have the right value of the valon frequency, because it is used in the timestamp subsystem, the integration time, the channel numbering and in the time spending saving samples in the PowerPC. So be special careful with this variable (Its recommendable running the whole script if there is a change in the clock frequency).

If you want to change the valon frequency you could run:

```
new_freq = 1080     #MHz
roach.set_valon_freq(new_freq)
```

It should print you a message with the new valon configuration.

To change the reference you could run:

```
roach.set_valon_ref(ref='i')    #set internal reference
roach.set_valon_ref(ref='e')    #set external reference
```

The external reference of the Valon is connected to one of the SMA ports in the back of the ROACH board, so if you are using this configuration be sure that the port is connected to a proper

reference.

If you want to check which are the derived values of the actual clock configuration you could see it in the internal variables of the class:

```
roach.get_sampling_freq()      #return sampling frequency at the ADCs
roach.get_fpga_clk()          #fpga clock derived from user inputs
```

With a correct clock configuration we could upload the fpga model and initialize it.

```
roach.upload_bof()
integration_time = 1*10**-3       #integration time in seconds
roach.init_vv(integration_time) #intialize the system and sets the integration time.

roach.get_aprox_clk()         #fpga clock measured
roach.get_fpga_clk()          #fpga clock derived from user inputs
```

In this portion of code upload and program the FPGA inside the ROACH board. We set the integration time, if you dont want to set integration time you could set it as zero. The default value for the integration time is 1.2136 ms.

The last two lines are a sanity check of the clock behaviour, the method roach.get_aprox_clock returns an estimation of the FPGA clock measured inside the FPGA. The estimation is not really accurate but should be near the actual value. So the two last lines should give similar results.

Now you have the vector voltmeter subsystem running. You could poke around using some commands like the following who deploys animation of ADC snapshots and the spectrums.

```
roach.adc_snapshot()       #plot snapshot of the inputs
roach.create_plot()       #create the plot object
roach.generate_plot()      #generate plots, the default option is the spectrums of both ADCs.
```

To initialize the timestamp you must ensure that the IRIG-B000 master is connected to the GPIO of the ROACH. Check if the master clock is connected to the SMA at the back panel of the ROACH.

Then you have to run the following command:

```
threshold = 5.*10**-5      #in seconds
roach.init_timestamp(unlock_error = threshold)
```

With this command you write the internal registers of the timestamp subsystem, note that this registers depends completely on the FPGA clock. You also set the threshold value which raise a warning flag if its overtaken, the default value is 10**-4.

This command tries to obtain the clock time of the master, if there is no sign of an IRIG packet prints a message asking to the user if it should try it again.

If everything goes well you could check the time with the command:

```
roach.get_hour()    #print the current time in day/hour/minutes/seconds format
roach.get_unlock()  #check if the systems is locked. 1=unlocked; 0=locked
```

Summarizing an initialization code could be:

```
from vv_calan import vv_calan
roach_ip = '192.168.0.40'                    #roach IP
bofname = 'vv_casper.bof'
valon_freq = 1080
roach = vv_calan(roach_IP, bofname, valon_freq)
roach.upload_bof()

##init vector voltmeter subsystem
integration_time = 1*10**-3        #integration time in seconds
roach.init_vv(integration_time)     #intialize the system and sets the integration time.

#check the fpga clock
roach.get_aprox_clk()          #fpga clock measured
roach.get_fpga_clk()           #fpga clock derived from user inputs

#initialize the timestamp
threshold = 5.*10**-5       #in seconds
roach.init_timestamp(unlock_error = threshold)

roach.get_hour()
```

2.2. Calibrations

In order to use the ADCs in their best performance you should make some calibrations previous the measurements.

To make the calibrations you need to feed the ADCs with a 10MHz signal in both ADCs, like its shown in figure 8.

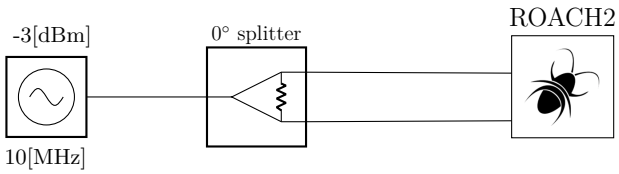


Figure 8: Diagram to calibrate the ADCs.

When you have the setup in 8 ready you could run:

```
IP_generator = '192.168.0.33'   #IP of the source
roach.calibration(load=0, man_gen=1, ip_gen=IP_generator,
                  filename='cal')
roach.synchronization()     #this is not actually necessary
```

roach.calibration runs 3 calibrations MMCM, OGP and INL which are explained in the subsection 1.2. The variable **load** enables you to upload OGP and INL if you previously had made them. If load=0 the calibration files are saved with the name given in the variable **filename**. If load=1 the OGP and INL are load from the filename location. The recommendation is to make the 3 calibrations in each power cycle of the ROACH and each time the Valon frequency is changed.

If you generator supports visa commands you could set the variable **man_gen=0** and give the

IP address of the generator. With this configuration the script will take control of the equipment to perform the calibration. You could also control manually the generator setting **man_gen=1**.

When the calibration is finished it will generate plots similar to the figures 9 and 10.

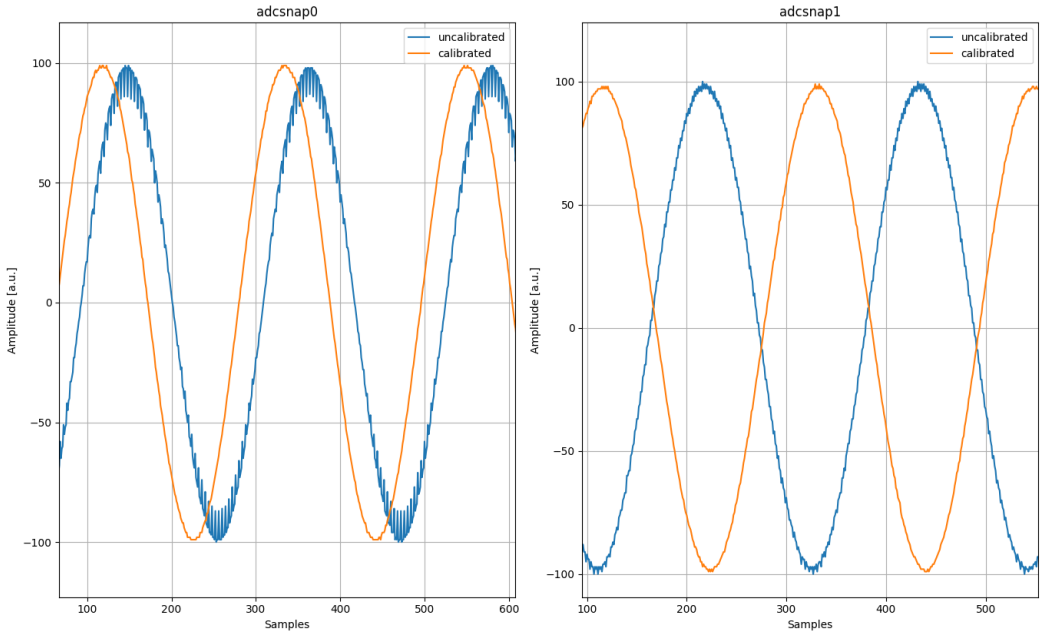


Figure 9: Effect of the calibration in time domain

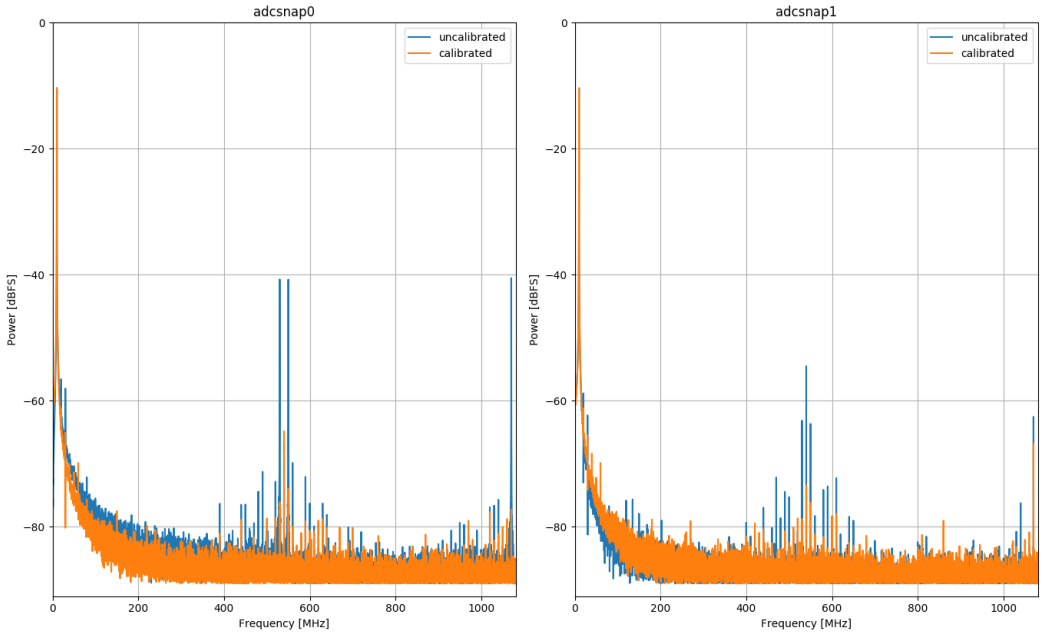


Figure 10: Effect of the calibration in frequency domain

The synchronization commands implements an iterative method with the final goal to found the

time delay value to have both ADC with the same phase. If you only want relative phase between the inputs this is not necessary.

2.3. Plots and visual items

You could access to the real time animated plots of the spectrum, correlations etc. For that propose the python class have the following methods:

```
roach.adc_snapshot()    #plots a snapshot of the ADCs input
roach.create_plot()     #create the plotting object
roach.generate_plot(plots=['spect0', 'spect1'], chann=6068, freq=[0, 67.5], manual_bw=0)
 ↪  #deploy the plots of adc0 and adc1 spectrums
```

As its name suggest roach.adc_snapshot enable to look at the ADCs values at the entrance of the FPGA system in the time domain.

roach.generate_plot() creates the plotter object. After creating this object you could generate a bunch of plots using roach.generate_plots(). This method allows to make several subplots entering key-words to the list **plots**. The key values for plots are:

- spect0: Plots the ADC0 spectrum.
- spect1: Plots the ADC1 spectrum.
- correlation: plots the real an imaginary part of the correlation.
- phase: Plots the relative phase between ADC0 and ADC1.
- chann_values: Plots the magnitude and phase for a given FFT channel.

You could join several subplots in the list **plots**. For example in the following code we plot the full spectrum and relative phase value for all FFT channels.

```
roach.generate_plot(plots=['spect0','spect1', 'phase'])
```

Usually we are interested in some portion of the spectrum, we could focus filling the variable **freq**, for example freq=[46,46.2] deploys the plot zoomed in the frequency range (46, 46.2)MHz.

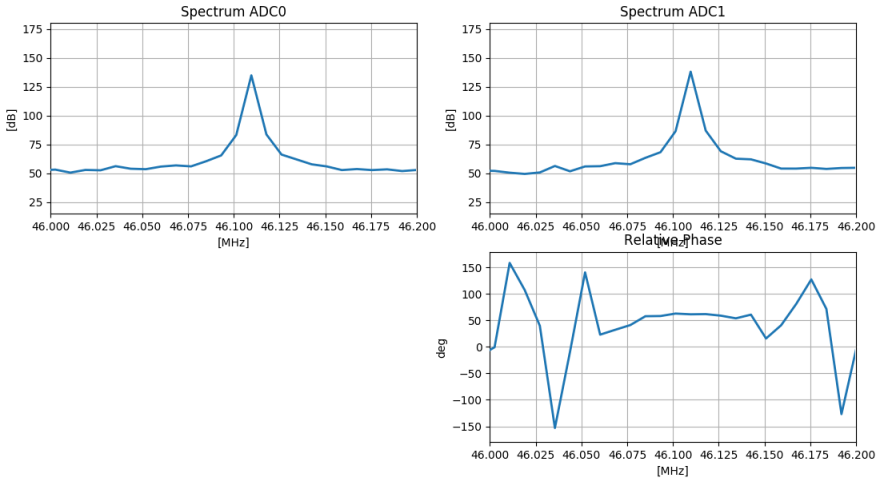


Figure 11: Plot example, with variables **plots=['spect0', 'spect1', 'phase']** and **freq=[46,46.2]**

You could also select to put by hand the bandwidth of the FFT (useful if you are working in a different Nyquist zone). To select this option set **manual_bw=1** and fill the variable **bw=[start, end]** with the proper values.

So for example we use the following codes to plot the evolution of the 48MHz channel:

```
freq = 48
chann = roach.get_index(freq)
roach.generate_plot(plots=['chann_values'], chann=chann)
```

To follow the time evolution of one FFT channel you could include the key-word **chann_values** and set the variable **chann** to the proper value.

The plot generated contains the power level of each ADC input and the relative phase of the channel. To make more clear the implementation behind the scenes, to store the power and phase of one channel we use a 8192 address bram and for each valid value the data is stored in one address. When the counter address reach the maximum it starts again in the first address, that's why you could see that values are being update in a cyclical way.

To obtain the channel number for a given frequency you could use the auxiliary method **roach.get_index()** which receive as input a frequency in MHz and returns the nearest channel.

2.4. PowerPC measuring codes

The previous subsection presented the methods to look at the results of the computations made inside the FPGA, but those measurements are taken at most at the speed of the network so it doesn't ensure that we are not losing information in the data transfer time.

In order to fully save the data, we avoid using the network and instead we save the data inside the PowerPC. Like we mention in the subsection 1.3 the PowerPC runs a minimalist version of Linux with a memory mapped file in /dev/roach/mem to have write/read access to certain brams in the FPGA. So, the main idea is to cross-compile a script for the PowerPC architecture, upload it and

run it inside the microprocessor. The upload, download and the measurement orders are given using the telnet connection in the PowerPC.

Our codes takes care of those details, if you want to see the fully set of commands that you have to issue to the telnet server take a look to the subsection 2.7.

Here is a simple example how to upload the measurement code to the PowerPC and starts a measurement in the channel correspondent to 50MHz

```
roach.ppc_upload_code()
chann = roach.get_index(50)    #obtain the correspondent FFT channel
duration = 20                  #in minutes
roach.ppc_meas(chann=chann, duration=duration)
```

With those commands the system is saving the data in the /var/tmp PowerPC folder. Is important to note that the duration of the measurements is translate into a number of readings of the brams, that depends of the clock frequency and the accumulation number.

You could check if the process is running with **roach.ppc_check_status()**. You could kill the measuring process using **roach.ppc_finish_meas**, the code running in the PowerPC handles the kill signal and gives a clean exit.

To transfer the collected data you have to issue the following command where my_IP is your local computer IP and the file would appear in the current directory with the name **raw_data**:

```
my_IP = 192.168.0.29
roach.ppc_download_data(my_IP)
```

With the data in your local computer you could decode it and store in HDF5 format using the following command:

```
raw_name = 'raw_data'      #binary file name
parse_raw_data(filename=raw_name, n_reading=None)
```

If you had killed the process a warning will be raised because the total size that the method expect is higher than the actual file size. Either way the method will try to translate the data until the end of file. Checking the files sizes would be a good practice.

2.5. Summary of functions

To initialize the system you have the following methods:

- **vv_calan(roachIP, bof_file, valon_freq)**: Initialize the vector voltmeter object.
- **upload_bof()**: Upload and program the FPGA with the **bof_file** model.
- **init_vv(integ_time)**: Initialize the vector voltmeter and sets the integration time.
- **init_timestamp**: Initialize the timestamp subsystem with the time code given by the Master clock.

To interface with the Valon synthesizer and to calibrate the ADC you could use the following commands:

- **calibration(load, man_gen,ip_gen,filename)**: Perform the MMCM, OGP and INL calibrations.
- **synchronization**: Synchronize both ADCs using for that propose a programmable delay.
- **set_valon_freq(new_freq)**: Change the valon frequency.
- **set_valon_ref(ref='i')**: Sets the Valon reference, the input could be **i** or **e**.
- **get_valon_status**: Prints the configuration of the Valon.

To deploy real time animations you could use the next commands:

- **adc_snapshot**: Returns an animation of the ADCs data.
- **create_plot**: Creates the plotter object.
- **generate_plot(plots, chann, freq, manual_bw, bw)**: Filling the variable plots with different key-words enables to look at different plots.

PowerPC measuring codes:

- **ppc_upload_code**: Uploads the executable measuring code to the PowerPC. Remeber that we only could use the tmpfs directory and get erased with every power cycle.
- **ppc_meas(chann, duration)**: Store the magnitude, correlation and timestamp for the channel **chann** for the amount of time given in **duration**.
- **ppc_finish_meas**: Kills the measuring process in the PowerPC. The executable file that we upload handles the kill signal to make a clean exit.
- **ppc_check_status**: Check if the measuring process is still running.
- **ppc_download_data(pc IP)**: Transfer the measured binary data stored in the PowerPC to the local computer.
- **parse_raw_data**: Translate the binary file to actual numbers and save it as HDF5 file. This file has the following fields:
  - PowA: store the magnitude of ADC0.[4]
  - PowB: store the magnitude of ADC1.
  - ABre: store the real part of the correlation.

[4] Remeber that it is not actually the power, we dont calculate the square root, so when passing to dB you should use 10*log10(PowA)

- ABim: store the imaginary part of the correlation.
- seconds: store the seconds of the timestamp.
- frac_sec: store the fraction of seconds of the timestamp.

There are a bunch of other auxiliary methods inside the **vv_calan** class.
We give the following list as example, you could always look at the docstring of each function:

- **get_adc0_spect**: Returns the FFT spectrum of ADC0 in a numpy array.
- **get_adc1_spect**: Returns the FFT spectrum of ADC1 in a numpy array.
- **get_rel_phase**: Returns the phase for the whole bandwidth.
- **init_chann_aqc(chann, n_samp, continuous)**: Configure the acquisition of one FFT channel to be read by the local computer.
- **get_chann_data**: Returns the data of one channel. You should have run the **init_chann_aqc** before running this one.
- **get_approx_clk**: Returns an estimated clock frequency measured inside the FPGA, is not really precise.
- **get_fpga_clock**: Returns the derived FPGA clock using the input data given by the user.
- **get_index(freq)**: Given a frequency in MHz returns the FFT channel number.
- **get_sampling_freq**: Returns the derived sampling frequency using the user input.
- **get_hour**: Returns the hour in the timestamp subsystem.
- **get_unlock**:Returns 1 if the timestamp is unlocked and 0 if the system is locked.
- **set_integ_time**:Sets the integration time, you should prefer **init_vv**.

2.6. Complete sample code

```
from vv_calan import vv_calan
import time, os
import numpy as np

roach_ip = '192.168.0.40'
bofname = 'vv_casper.bof'
valon_freq = 1080       #check the valon is working at the rigth freq

roach = vv_calan(roach_ip, bofname, valon_freq)
roach.upload_bof()

#sanity check
deriv_clk = roach.fpga_clk()
```

```
estim_clk = roach.get_aprox_clk()

print("Derived clk: %.4f \t Estimated clk: %.4f " %(deriv_clk, estim_clk))

#initialize the vector voltmeter and timestamp subsytems
integ_time = 1.*10**-3
roach.init_vv(integ_time)
thresh = 10**-4
roach.init_time(thresh)

#Time stamp sanity check
print("Timestamp hour:")
print(roach.get_hour())
print("Unlock ? "+str(roach.get_unlock()))

#vector voltmeter sanity check plots
roach.adc_snapshot()
roach.create_plot()
roach.generate_plot(plots=['phase', 'spect0', 'spect1'], freq=[45, 55])


#stops the code to prepare the calibration setup
while(1):
    tmp = raw_input('Are you ready to perform the ADCs calibration (y/n)?')
    if(tmp=='y'):
        break

ip_gen = '192,168.0.33'
roach.calibration(load=0, man_gen=0, ip_gen=gen_ip)

#stop the code to disconnect the calibration setup
while(1):
    tmp = raw_input('Continue (y/n)?')
    if(tmp=='y'):
        break



#Explore the spectrum to see where is the signal... here I suppose that is in 50MHz.

ADC0_peak = np.argmax(roach.get_adc0_spect())  #Where is the peak in the spectrum..This isnt
    ↪ the best way, the DC should have a high value too..You could make a plot too..
ADC1_peak = np.argmax(roach.get_adc1_spect())
chann = roach.get_index(50)     #chann where I think the signal should be

print("ADC0 peak index: %i \t ADC1 peak index: %i" %(ADC0_peak, ADC1_peak))
print("My channel index: %i" %(chann))

#suppose that everything seems good

roach.generate_plot(plots=['chann_values'], chann=chann)
```

```
#look how the channel evolves in time

#start the PowerPC storage..
roach.ppc_upload_code()
duration = 20           #we store 20 minutes
roach.ppc_meas(chann=chann, duration=duration) #starts the measurement

roach.ppc_check_status()

#if everything is working we look at the real time evolution again
roach.generate_plot(plots=['chann_values'], chann=chann)

###
##suppose we finish the measurement, the script doesnt follow until we
## close the animation window
##
roach.ppc_finish_meas()     #kill the measurement process in the PowerPC


my_ip = '192.168.0.29'
roach.ppc_download_data(my_ip)
print(os.system('ls'))     #we should see the raw data in our directory

roach.parse_raw_data()
print(os.system('ls'))     #now we should have a hdf5 file.
```

2.7. Note about PowerPC codes

This subsection is just to show how to manually set the measurements in the PowerPC. It is not necessary to read it, but as we didn't test the codes in every os distribution could be useful to keep it here.

Like we saved the data in a temporary directory we must upload the script when we want to make a measure. This could be do it by hand running the following commands. As example the ROACH IP will be 192.168.0.40 and the IP of our local machine will be 192.168.0.29

1. Log in the default telnet session of the ROACH and go to the tmp directory:

```
telnet 192.168.0.40
    Trying 192.168.0.40...
    Connected to 192.168.0.40.
    Escape character is '^]'.

roach021417 login: root

Consult /root/README for a basic guide
~ # cd /tmp/

```

2. Execute the code following code in the ROACH terminal:

```
~# nc -l -p 1234 > ppc_save
```

3. Open a second terminal in your local machine, go to the directory where is located the *ppc_save* which comes with the package. Then run:

```
nc -w 3 192.168.0.40 1234 < ppc_save
```

4. With this codes you should see that the executable file is now in the ROACH tmp file. To run it you should first change the permissions and we create the storage file:

```
chmod +x ppc_save
touch save
```

5. The script ask for a number N as argument. N is the number of iterations that this script should read the two memories. To calculate the total time that it spend reading use the following formula:

$$time = \frac{2 \cdot 8192 \cdot acc\,number \cdot 16384 \cdot N}{FPGA\ clk} \tag{1}$$

Also remember that the total memory available is 378.2MB. At the end of the measure you are going to have:

$$Total\ Memory\ usage = N * 8192 * 8 * 5 * 2/2^{20}\ MB \tag{2}$$

6. To run the code manually you have to run the following code where N is the value you calculate previously.

```
busybox nohup ./ppc_save N
```

This will send you a lot of messages in the display that informs that a read was perform and in which of the two memories the system is reading.

7. You could kill the process with the kill command, there is a catching for this signal. When the process is finished you could transfer the data file running in your local computer:

```
nc -l -p 1234 > raw_data
```

Then you go to a second terminal inside the ROACH tmp/ directory and run:

```
busybox nohup nc -w 3 192.168.0.29 1234 < save
```

Then you should have the binary file in your local computer.