Holography Backend User Guide

Contents index

Figures index

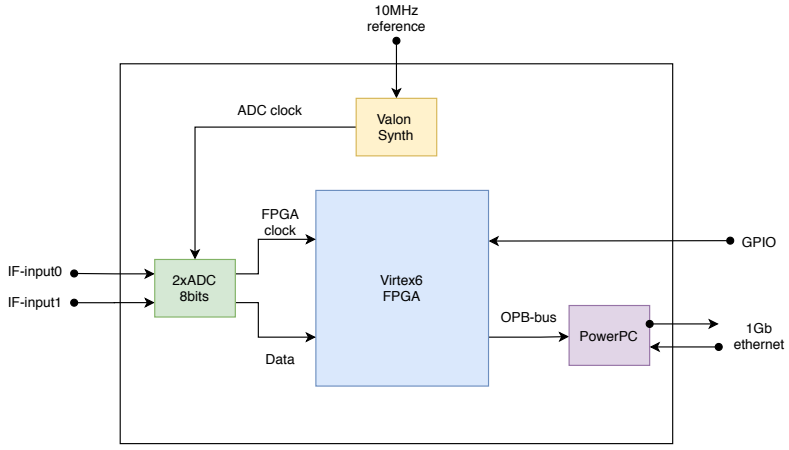


Figure 1: Basic diagram of the ROACH system

1. Introduction

The back-end for the CCAT-p holography system consist of a vector voltmeter that measures both magnitude and relative phase between two inputs, and a finite state machine that assigns a timestamp in the measurements.

This project was implemented in the platform ROACH2 from CASPER. A general diagram of the hardware is shown in figure 1, which emphasizes the 4 relevant subsystems used by the vector voltmeter.

The process can be resumed as follows:

- The Valon synthesizer generates the clock signal and for the ADCs. This clock signal can be locked to an external 10MHz reference signal.
- The ADC samples at two times the valon frequency, because of its internal structure. Thus $Sampling\,Frequency = 2 \cdot Valon\,Frequency$
- The ADC gives the clock signal to the FPGA. Therefore, the entire system is locked to the reference of the Valon.
- The samples are read by the FPGA, which processes them with standard digital techniques and stores them in internal memories. These can be accessed by the PowerPC microprocessor.
- The PowerPC runs a telnet server that can be accessed via the 1Gb Ethernet connection. This server handles requests for writing and reading the aforementioned memories.

The next subsection gives more detail of the ROACH subsystems and how to interact with them. Some codes were developed to hide the complexity of handling the binary data collected. A summary can be found at the end of this section.

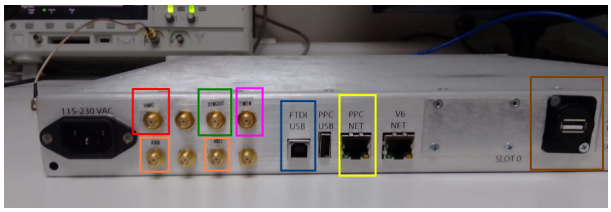


Figure 2: Roach input/output ports. Orange:IF inputs, Red:10MHz input/output (Careful setting the valon reference, if you connect something when is configured as internal its going to output a 10Mhz signal) Green:sync out pulse signal, Pink;Irig input, Blue:PowerPc serial interface to set the IP, Yellow: PowerPc ethernet phy,Brown:Valon serial interface.

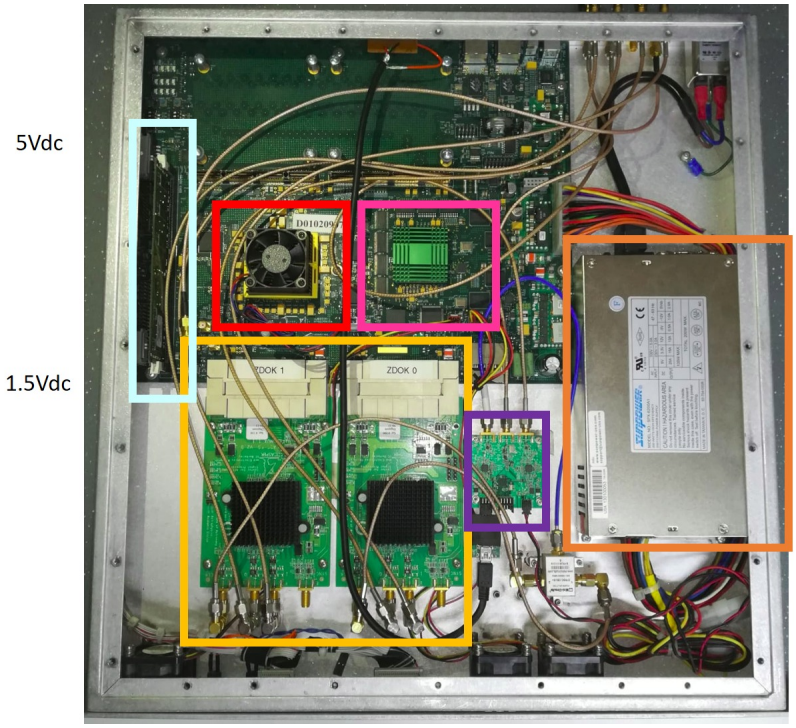


Figure 3: Example of the inside of the ROACH2 system. Inside the red rectangle is the Virtex6 FPGA, in pink rectangle is located the PowerPC, the yellow rectangle contains two ADC boards, the orange rectangle contains the power supply and in the white rectangle contains the DRAM and the GPIOs of the FPGA.

2. System description

2.1. Valon synthesizer

The Valon synthesizer has three ports, two of them are independent channels with frequencies in the range (137.5-4400)MHz and the last port is used to receive an external reference. Prior to its use, the synthesizer must be configured to use the internal or external reference. If this is not done, the

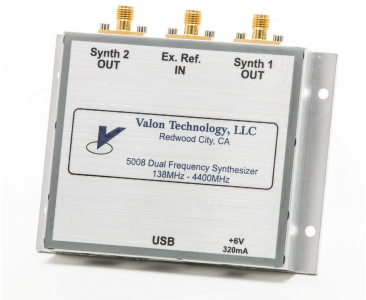


Figure 4: Valon 5008 synthesizer.

synthesizer will work with the previous configuration.

Be extra careful, when the reference is set to internal the port will output a 10MHz signal, so be sure to disconnect delicate equipment.

To configure the synthesizer a mini-USB connector can be used. It runs an UART protocol and accepts commands using standard serial programs like minicom.

The codes presented here use the NRAO python package to program the valon synthesizer. This package can be found here.

Although the Valon enclosure specifies a supply voltage of 6 Vdc, the datasheet mentions that 5 Vdc supply is also allowed. The Valon is delivered already assembled inside the ROACH enclosure, uses a 5Vdc rail from the ROACH power supply, the power level for the ADC clock is set adequately, so in operation you just have to change the frequency and the reference that the Valon is using.

Some recommendation are:

1. A power level of -5 dBm must be fed to the ADC input to get the maximum dynamic range.
2. To change the sampling frequency, the valon must be programmed at half the desired value.
3. Check that the reference signal is connected.
4. If there is no response from the Valon synthesizer, check if the tty/port set in the python code is the actual used by the Valon.[1]

2.2. ADC boards

The ADC boards are based on the EV8AQ160 chip. The datasheet can be found here. The input ports of the ADCs are located at the back panel of the ROACH enclosure, labeled accordingly.

A board is composed by 4 ADCs with interleaved sampling time. This means that the clock signal is splitted and phase shifted into 4 signals. The phases are 0°, 90°, 180° and 270°.

To synchronize the internal ADCs, three calibrations must be done before a measurement takes place.

- MMCM Calibration: According 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*

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

test ramps that they use to detect glitches and adjust the MMCM appropriately to eliminate them. This calibration must be performed every time the FPGA is reprogrammed or the clock rate is changed.

- OGP (Offset Gain Phase) calibration: This procedure makes a fine calibration writing internal registers in the four ADCs. The ADC board is fed with a known sinusoidal signal and, using least squares, the offset, gain and phase corrections are calculated. 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. To use this correction the ADC boards must be fed with a known signal.

Here a note can be found, describing the characterization of the ADC board made by the SMA back-end team.

The codes to calibrate the ADC boards are based on the calibration repository of NRAO. After every calibration the obtained values are stored, which can be uploaded instead of calibrating each time.

To summarize, **when programming the ROACH with a model, MMCM must be done, the OGP and INL calibration could be loaded but usually the MMCM is enough, the recommendation of NRAO is to perform the OGP and INL approx each 6 months (or when you change the ADC clock frequency). The good part of this is that you could perform the MMCM without feed a 10MHz signal (it uses an internal calibration signal). If clock rate is changed it is recommendable do the three calibrations.** [2]

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

Finally, some words about the time delays between the ADCs samples. Because the FPGA has to route all the inputs to the logic gates to process the data, there can be a delay between the two ADCs inputs given by the route selected by the Xilinx compilation. This is translated to a constant phase shift in the input. But there is another problem, when powering the ADCs some cores start to sample previous than others, so there is a variation in the absolute phase between the ADCs, so if you take a measurement, power off the roach and take a new measure the absolute phase will change, so you only can compare phases of one power cycle.

The good news is that the delay introduced is a fixed amount of FPGA clock cycles, so it can be compensated using a programmable delay. A script was developed to calibrate this phase difference, using a 10MHz signal at both inputs and 0° splitter.

This synchronization between the ADCs is not mandatory when measuring relative phases between inputs, but it will lead to different results in the absolute phase. Also between each power cycle the phases shift will vary, so you cannot related them easily if you not calibrate them first.

2.3. PowerPC

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

[2] This is mandatory if a precise measurement is desired, the system can work without the calibration but it could not achieve it best performance.

[3] Saturation can be checked by taking a snapshot of the ADC input.

Almost the entire system is read-only with some remarkable exceptions. The most relevant is the temporary directory located in /var/tmp that has 378.2Mbytes of space available, but gets erased every time the ROACH is turned off.

Also the file */dev/roach/mem* is writable, this is where the registers of the FPGA are mapped in order to be written or read.

The standard method of communication with the PowerPC is using a telnet session. The system will ask for an username, the default username is **root** and without password.

Failing this, another method is using the type B USB connector, with a UART program like minicom. This enables to change the default IP address in the /etc/rc.local file to set a preferred interface. Also is advisable to check the 1Gbe ports ports at the back panel, because only one of them has access to the PowerPC.

At the PowerPC boot stage two telnet servers starts to run. The first one has the port 23, which is the default telnet port, gives access to the BORPH system. The second one is has the 7147 port number and runs the Karoo Array Telescope Control Protocol (katcp). This protocol is a layer between a general purpose computer and the FPGA. Therefore, every read and write operation pass through this protocol.

Here some basic commands can be found to interface the ROACH system. In order simplify the use of this protocol the python package corr can be used, which translates katcp commands to simple python statements.

As an example, a standard reading procedure can be:

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. Use the corr package to request a data read through telnet.
4. The BORPH system uses the mmap command to open the /dev/roach/mem file and reads the addresses requested by katcp.
5. The PowerPC sends the response using the katcp protocol.
6. The desktop computer receives the response, and recovers the data using the corr package.

2.4. Virtex6 FPGA

2.4.1. Vector voltmeter subsystem

The basic diagram implemented in the FPGA consists of two spectrometers, a correlator between the two inputs and a timestamp subsystem.

The clock signal for the whole FPGA system is locked to the ADC clock and equals to 1/8 of the Valon frequency (or 1/16 of the sampling frequency of the ADCs). The system was compiled using a 135MHz clock frequency, which is the maximum working frequency guaranteed by the Xilinx toolflow. Beyond this threshold there could be path delays and the system would become erratic. Using lower frequencies is not a problem.

The maximum frequency that you could set in the Valon is 1080MHz, with the aforementioned clock frequency.

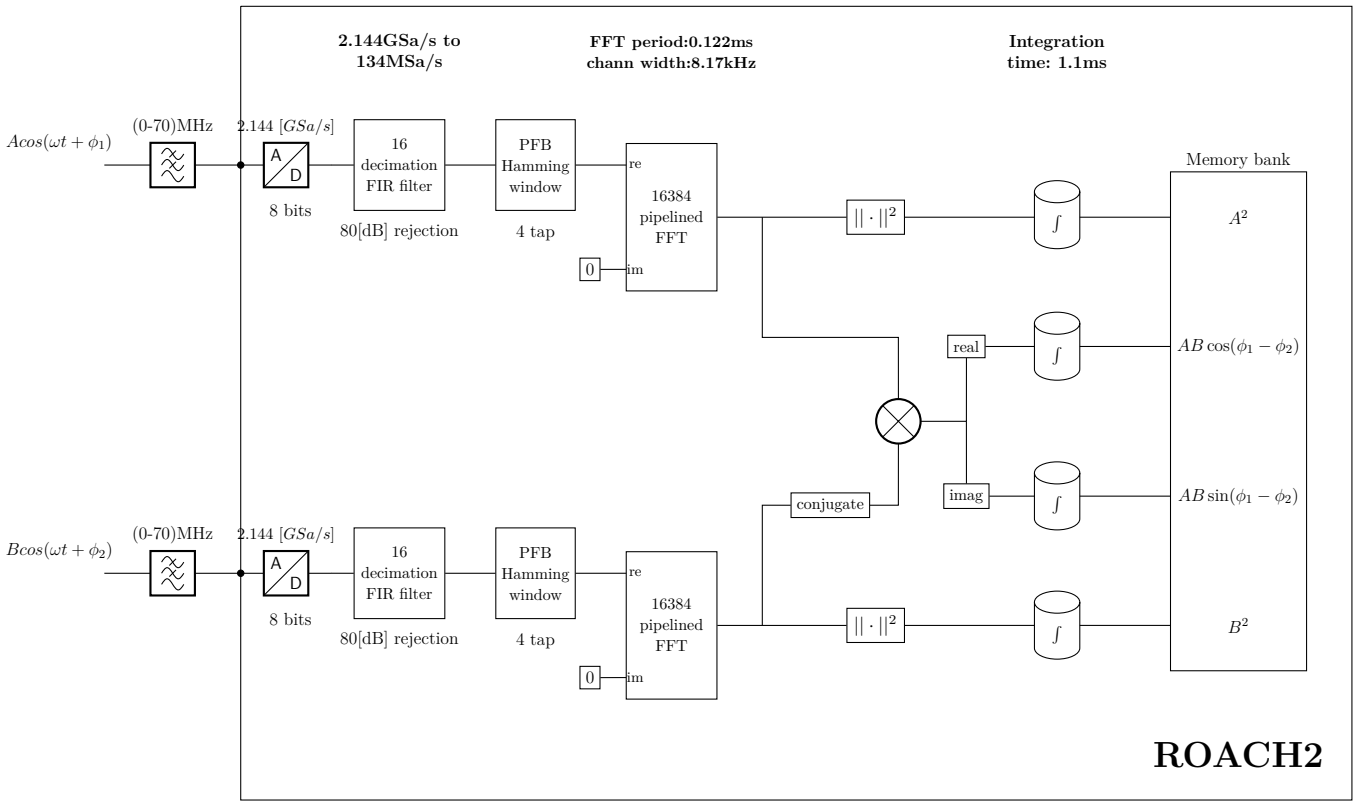


Figure 5: Vector voltmeter block diagram.

Each sample has 8 bits and, because the FPGA clock is 1/16th of the ADC clock, in each clock cycle the system has 16 samples available.

Considering that the theoretical SNR of a 8bit ADC is 49.9dB, which is less that specifications, the oversampling technique was implemented It consist in sampling over the Nyquist sampling frequency and then decimate using for a FIR filter. A decimation factor of 16 gives an improvement of 12.04dB in the SNR.

After the decimation stage the data is windowed to decrease the known leakage problem of DFT implementations. The window is implemented using a polyphase architecture based on the Hamming window. This polyphase architecture reduces significantly the side lobes, while maintaining the mainlobe width of the window, improving the spectral localization of the signal. More information about the PFB can be found here.

Then a pipelined Fast Fourier Transform (FFT) is computed over the windowed data. A pipelined FFT delivers one output channel per clock cycle. The FFT size is 16384, of which only 8192 channels contains useful information. Thus, a system running at 135MHz will have 67.5MHz of total bandwidth and 8.24kHz per channel.

The coherent sum implied by the FFT translates to a processing gain of the signal, which enhances the SNR by 39.13 dB in this implementation.

The power of the signal is obtained by calculating the square magnitude of the complex output of each FFT channel. As shown in Figure 5 it is not needed to compute the square root of the magnitude, because the final values are expressed in decibels.

Similarly the cross correlation is calculated using the complex conjugate of one FFT output multiplied by the second FFT output. This complex multiplication is synchronized to be performed in the same channel on both FFT outputs.

After the calculation of the self and cross correlation, there is an accumulation stage, equivalent to the integration in a telescope. The number of accumulations performed is programmed by setting one register in the FPGA.

Finally, there are several dual port Block RAMs (BRAM), where one port is written by the FPGA and the other port is read by the PowerPC. The BRAMs were divided in the following groups.

- Spectrum BRAMs: Stores the accumulated power for each ADC channel. These BRAMs are continuously updated, enabling real time monitoring of the spectra in an animation.
- Correlation BRAMs: Stores the real and imaginary part of the correlation. It is continuously updated, enabling real time monitoring of the correlation.
- One channel BRAMs: Stores the time evolution of one FFT channel, both power and correlation. Setting the right registers enables to plot the time evolution of this channel. These memories are intended for monitoring using a remote computer, because data can be lost in the middle during operation.
- PowerPC BRAMs: stores the magnitude, correlation and time stamp of a single channel. It consists of two sub-banks whose data will be transferred to the PowerPC temporary directory. When one sub-bank is full, a flag is raised and the data is written in the remaining sub-bank. Meanwhile, the first sub bank is read by the PowerPC and clears the aforementioned flag, thus the system will fill the second sub-bank and repeat the process for this sub-bank.
- ADC snapshot: Stores the output of the ADC, it can be used to monitor the behaviour of the input signal. It is useful for checking if the ADC is saturated by the input signal.

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 can be accessed and interpreted using the python package presented in this report.

2.4.2. Timestamp subsystem

The timestamp subsystem is based on the IRIG-B000 protocol. This protocol consists of 100 pulses delivered in 1 second. The information is encoded by the pulse width and the location of the pulse in the packet. For example, a pulse with a duration 2ms represents a logic 0 and a 5ms duration a logic 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). Because it is not a natural way to count seconds we translate this value to Time Of the Year (TOY) format.

As input port 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.** Inside the roach we place a voltage translator from 5 to 1.5V. The GPIOs are located in the bank 13. To access this location the DRAM card must be removed (white rectangle in 7) and the GPIO bank can be seen on one side. In case of any modification, attention to the distribution is adviced. The 1.5V rail is facing to the front and the 5V rail of the bank is facing back of the ROACH.

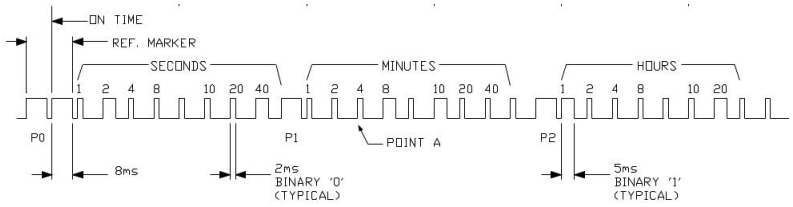
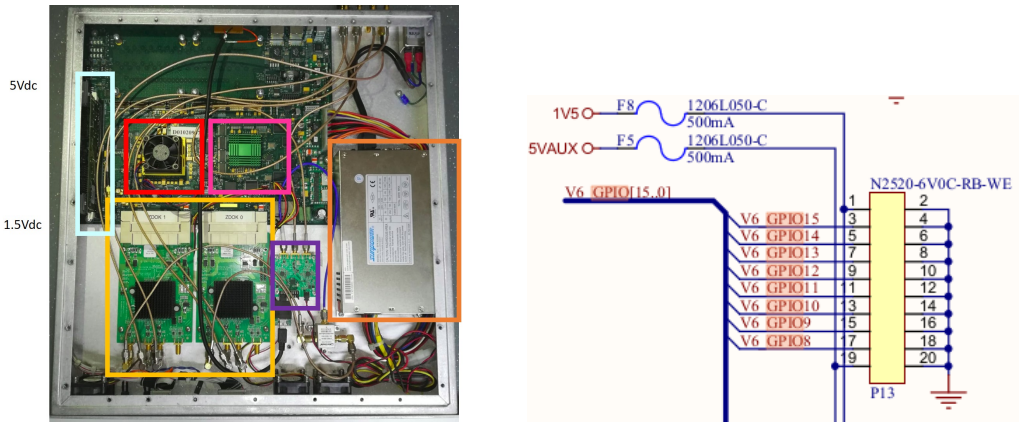


Figure 6: Example of a portion of IRIG packet.



(a) ROACH2 board image.

(b) GPIO schematics.

Figure 7: a) The GPIO is located in the bank 13 which its on the side of the memory card (white rectangle in a). b) Schematic of the 13 bank.

To obtain the time from the master clock an algorithm was implemented, which starts with the detection of the packet header, then counts the width of the pulses in FPGA's clock cycles and extracts the binary information. After decoding the BCD values, they are translated to TOY protocol. The time is decoded only using the first packet then, using the fact that the IRIG-B000 packet last 1 second and has 100 pulses, the TOY value will be increased by one second every 100 FPGA's pulses.

The sub-second time is accounted by a free running counter, which is cleared when a new second start, therefore sub-seconds figures are measured in FPGA's clock periods. This scheme is also useful to check any drift in time, because at the end of each second a comparison between the value of the free running counter and the ideal value can be done and determine if the system is locked.

The complete diagram of the algorithm is shown in the figure 8.

Also to synchronize the experiment we set an output for a pulse and save the time when it was triggered. The pulse has a variable width, but by default is set to 100ms. This pulse is approx 6.7V when the signal is not terminated, if the termination is of 50ohm the signal should go up to 3V.

2.5. Pulse output

To synchronize the measurement of the ROACH with other instruments it could generate a pulse of a configurable width. The pulse uses a GPIO of the ROACH and has a 5V height when its terminated in 50 ohm, otherwise it has a height of $\sim 6V$. At the rising edge of the signal the system saves the timestamp values.

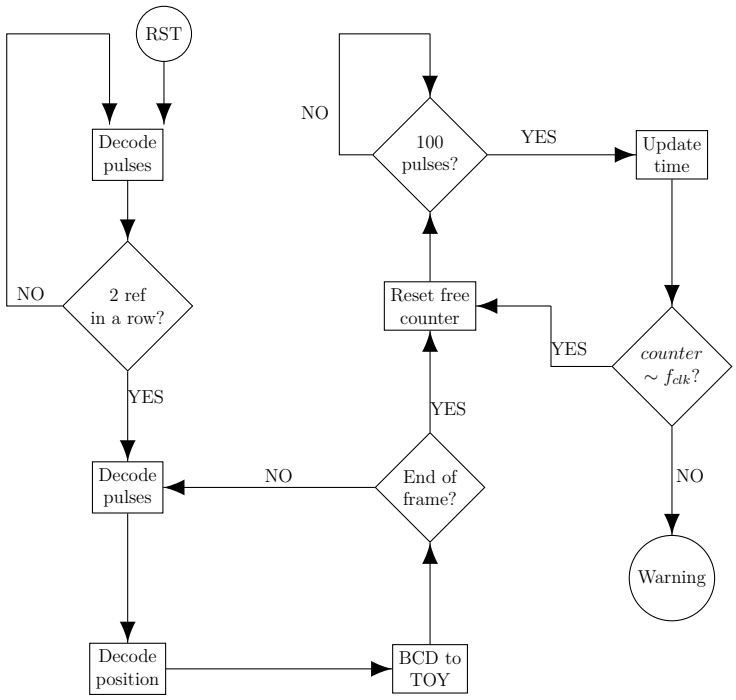


Figure 8: Flux diagram of Timestamp subsystem.

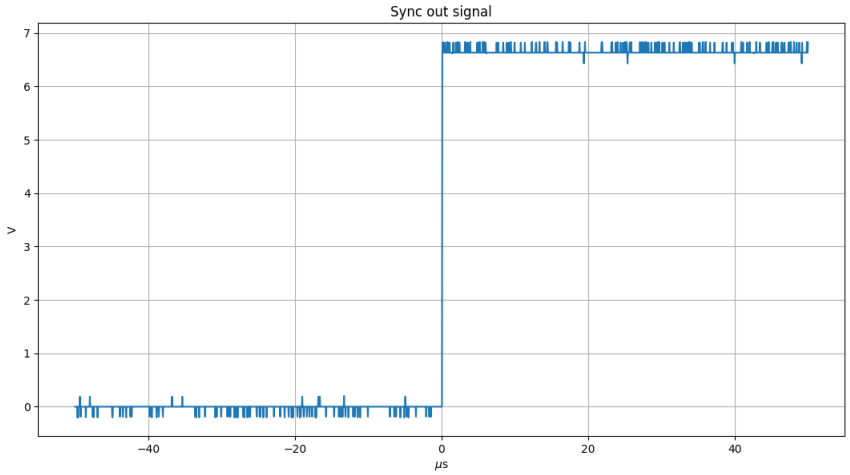


Figure 10: Non terminated sync out signal

2.6. 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.

For example, to read the magnitude of the spectrum $8192 * 8$ bytes of the bram must be read and, then, translate the binary to 8192 values. This is done using the package struct.

The real and imaginary part are saved interleaved in memory thus, to read the entire correlation,

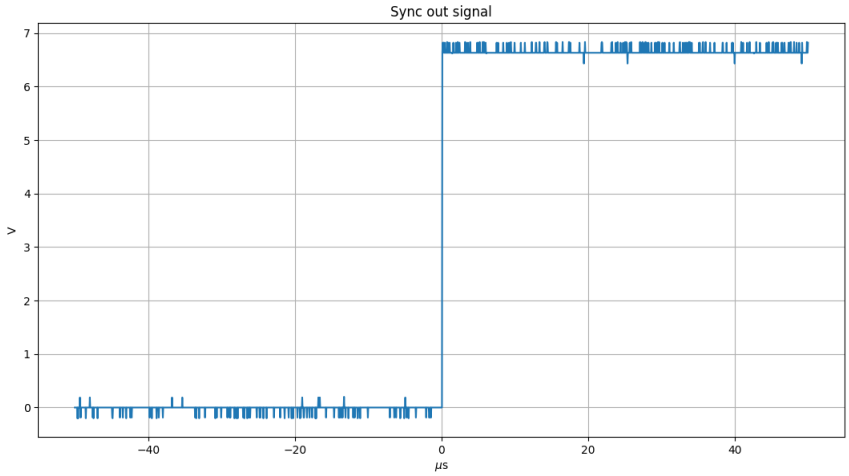


Figure 9: Rising edge of the sync out pulse

$8192 * 8 * 2$ bytes must be read, then translated to $8192 * 2$ signed long long values and separate odd and even indexes, where the odd indexes 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.

1. $8192 * 8$ bytes corresponding to *>8192Q* values of the ADC0.
2. $8192 * 8$ bytes corresponding to *>8192Q* values of the ADC1.
3. $8192 * 8 * 2$ bytes corresponding 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 corresponding 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.

2.7. Summary

1. The frequency of the Valon synthesizer can be set using a mini usb port with UART commands, using this scripts to program it. Also the reference can be set to internal or external.
2. The sampling rate of the ADCs is the double of the frequency set at the synthesizer. The clock at the FPGA is derived from the ADC clock, thus the whole system is locked to the same reference.
3. Perform the 3 ADC calibrations to achieve it best performance.
4. The maximum power at the input of the ADCs are -5dBm.
5. Communication with the PowerPC can be done using a telnet connection, using the port 23. The FPGA registers are mapped into /dev/roach/mem. The directory /var/tmp/ is writable and allows to store the vector voltmeter data.
6. Another option to communicate with the PowerPC is to connect to the port 7147 via telnet, sending read/write commands for specific registers 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 comprises a two signal correlator, implemented by a 16384 FFT, having only 8192 channels with useful information.
8. The number of accumulation (which is related to the integration time) can be changed by setting one FPGA register.
9. The following memories, within the FPGA, can be accessed:
   - Full spectrum of the two ADC inputs.
   - Correlation of the whole bandwidth.
   - Magnitude and correlation for one channel.
   - Time-stamped full data of one channel data, inside the PowerPC temporary directory.
   - The input of both ADCs in the time domain.
10. The timestamp subsystem uses the IRIG-B000 protocol. As gateway the GPIO port of the FPGA is used. This subsystem is only based in the FPGA clock cycles, it should be given to the system as input variable.
11. A threshold can be set to determine if the system is locked. Also, there are registers to obtain the current time.

3. Python package

A package was programmed as a simple interface with the ROACH, mainly based on the corr package. The latter implements reading and writing functions for every register in the FPGA, just specifying the name given to the register, the address to read/write and the values to write into, if applicable.

In this section several examples will be shown to illustrate the usage of the corr package. The tasks covered are: connecting to the ROACH, uploading a model into the FPGA, writing a register to reset the state of the system and, finally, reading binary data from a memory and translating it to python data types.

```
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 a basic task that the package developed for this project, vv_calan, simplifies. In the following subsection are presented some of the main functions of this package .

A short description of all the functions can be found in subsection 3.5, with sample codes for each function.

3.1. Initialization

The following lines of code generate the roach object, which has all the methods to interface with the ROACH system. The constructor (init function) needs the ROACH IP address, the bitstream file path and the frequency set in the Valon synthesizer.

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

To get the actual value of the Valon frequency, connect to the Valon mini-USB port and run:

```
roach.get_valon_status()
```

It should print out the actual frequency, then the constructor can be run to generate the roach object with the correct frequency value. It is important to have the right value of the Valon frequency, because it is used by the timestamp subsystem, the integration time, the channel numbering and in the time spending saving samples in the PowerPC. Special care is advised with this variable (It is recommendable running the whole script if there is a change in the clock frequency).

To change the valon frequency run:

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

It should print 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 panel of the ROACH enclosure. A proper reference signal must be fed into this port, when setting a external reference in the Valon.

To check the derived clock values for the actual Valon configuration run:

```
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, the next tasks are to upload the 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
```

The initialization also comprises setting the integration time. To disable the integration set the value to zero. The default value for the integration time is 1.2136 ms.

The last two lines provide a simple check of the clock behaviour, the method roach.get_aprox_clock returns an estimation of the FPGA clock measured in the chip. The estimation is not really accurate but should be near the actual value.

The vector voltmeter subsystem should be running. To show the data taken by the ROACH, the following commands display an animation of ADC snapshots and the spectra.

```
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 check first that the IRIG-B000 master is connected to the GPIO of the ROACH. A simple inspection of thee SMA port at the back panel of the ROACH is enough.

Then run:

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

This command writes the internal registers of the timestamp subsystem, which depends completely on the FPGA clock. The threshold value is also set, which can raise a warning flag if its overtaken, the default value is 10**-4. The method attempts to obtain the time from the master clock and, if there is not a detection of any IRIG packet, it reports to the user to try it again.

If everything goes well the time can be checked with the following commands:

```
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()
```

3.2. Calibrations

In order to use the ADCs at their best performance the following calibrations are advised, before any measurement.

To make the calibrations the both ADCs must be fed with a 10MHz sine wave signal, as shown in figure 11.

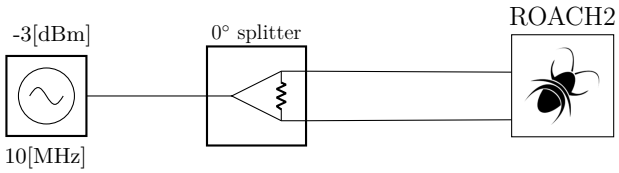


Figure 11: Diagram to calibrate the ADCs.

With the aforementioned setup, the following code needs to be 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 2.2. The variable **load** enables to upload OGP and INL data if you previously done. If load=0 the calibration files are saved with the name given in the variable **filename**. If load=1 the OGP and INL are loaded from the filename location. The recommendation is to make all 3 calibrations after every power cycle of the ROACH and every time the Valon frequency is changed.

If the signal generator supports SCPI commands, 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. The generator can be controlled manually by setting **man_gen=1**.

When the calibration is finished the method will generate plots similar to the figures 12 and 13.

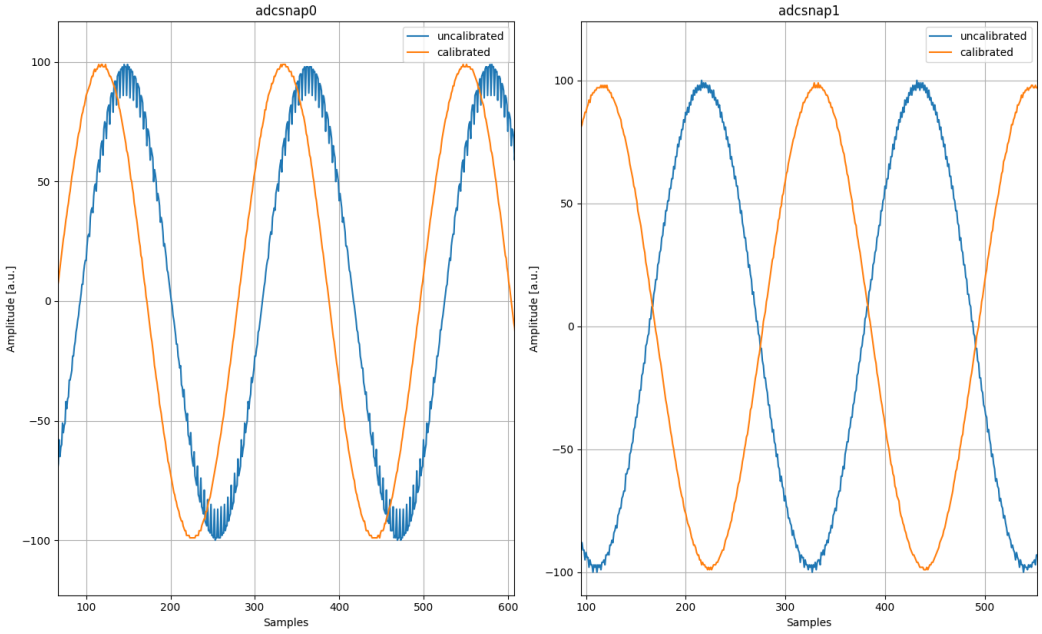


Figure 12: Effect of the calibration in time domain

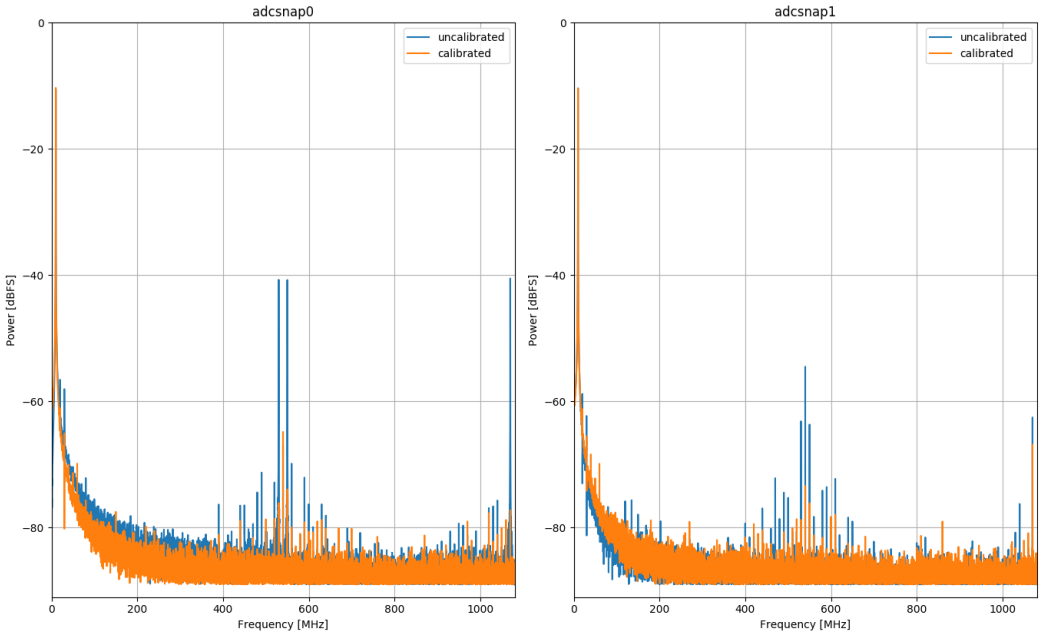


Figure 13: Effect of the calibration in frequency domain

The synchronization commands implements an iterative method to find the time delay value between both ADCs, to correct them and have with the same phase. If relative phase measurement between the inputs is desired, this calibration is not necessary.

3.3. Plots and visual items

To access to the real time animated plots of the spectra, correlations, etc. the following methods can be run:

```
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 suggests roach.adc_snapshot plots the ADCs values at input of the FPGA system, in the time domain.

roach.generate_plot() creates the plotter object. After creating this object several plots can be generated using roach.generate_plots(). This method allows to make several subplots entering keywords 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.

Several subplots can be joined in the list **plots**. For example, in the following code the full spectrum and relative phase value are plotted for all FFT channels.

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

To plot a portion of the spectrum, the variable **freq** can be used, for example freq=[46,46.2] displays the spectrum zoomed in the frequency range (46, 46.2)MHz.

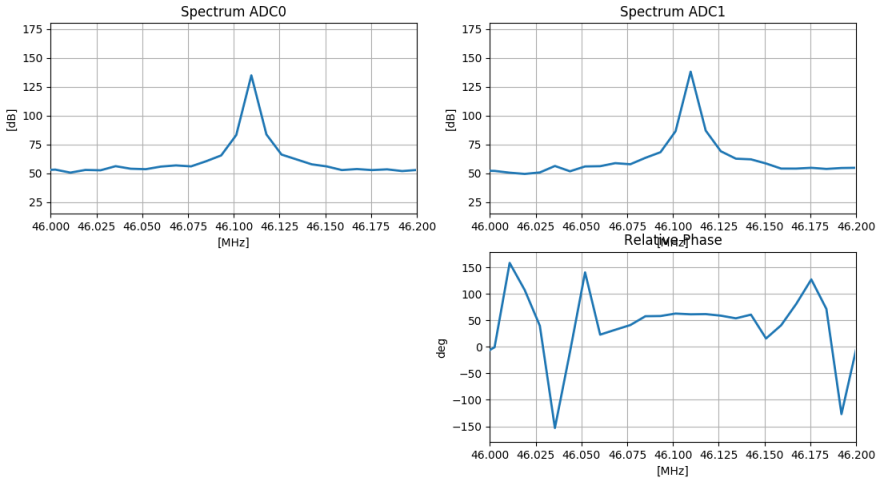


Figure 14: Plot example, with variables **plots=['spect0', 'spect1', 'phase']** and **freq=[46,46.2]**

Also, the bandwidth of the FFT can be set manually (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.

For example, the following codes are used 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.

The auxiliary method **roach.get_index()** can be used to obtain the channel number for a given frequency. It receives as input a frequency in MHz and returns the nearest channel.

3.4. PowerPC measuring codes

The previous subsection presented the methods to monitor the results of the computations made inside the FPGA. But those measurements are taken, at most, at the speed of the network, which

doesn't ensure that there is not information lost during data transfer time.

In order to fully save the data, it is stored inside the PowerPC. As mentioned in the subsection 2.3, the PowerPC runs a minimalist version of Linux with the BRAM mapped to a file in /dev/roach/mem, enabling write/read access in the FPGA. 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 measurement commands are given using the telnet connection in the PowerPC.

The **vv_calan** package handles those tasks, to read the fully set of commands available for the telnet server go to the subsection 3.7.

A simple example, shown below, of how to upload the measurement code to the PowerPC and start a measurement in the channel corresponding 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. It is important to note that the duration of the measurements translates to a number of readings of the BRAMs, a number that depends on the clock frequency and the accumulation number.

To check if the process is running use **roach.ppc_check_status()**. To kill the measurement process use **roach.ppc_finish_meas**, the code running in the PowerPC handles the kill signal and gives a clean exit.

To transfer the collected data use **roach.ppc_download_data(my_IP)**, where my_IP is your local computer IP and the data will appear in the current directory in a file with name **raw_data**:

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

To decode the file and store the data in a HDF5 formatted file, use the following command:

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

If the process was killed a warning will be raised, because the total size that the method expects 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 is a good practice.

3.5. Summary of functions of vv_calan package

System initialization methods:

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

To interface with the Valon synthesizer and to calibrate the ADC, use the following methods:

- **calibration(load, man_gen,ip_gen,filename)**: Performs the MMCM, OGP and INL calibrations.
- **synchronization**: Synchronizes both ADCs using for that propose a programmable delay.
- **set_valon_freq(new_freq)**: Changes 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 use the next methods:

- **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. Remember that the tmpfs directory is used and, therefore, is erased with every power cycle.
- **ppc_meas(chann, duration)**: Stores 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**: Checks if the measuring process is still running.
- **ppc_download_data(pc_IP)**: Transfers the measured binary data stored in the PowerPC to the local computer.
- **parse_raw_data**: Translates the binary file to actual numbers and save it as HDF5 file. This file has the following fields:
  - PowA: stores the magnitude of ADC0.[4]
  - PowB: stores the magnitude of ADC1.
  - ABre: stores the real part of the correlation.
  - ABim: stores the imaginary part of the correlation.

[4] Remeber that it is not actually the power, therefore, when passing to dB use 10*log10(PowA)(?)

- seconds: store sthe seconds of the timestamp.
- frac_sec: stores the fraction of seconds of the timestamp.

There are several auxiliary methods inside the **vv_calan** class. The following list is an excerpt, please check the docstring for more details:

- **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)**: Configures the acquisition of one FFT channel to be read by the local computer.
- **get_chann_data**: Returns the data of one channel. **init_chann_aqc** must be run previously to this.
- **get_approx_clk**: Returns an estimated clock frequency measured inside the FPGA, it is not very 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, use **init_vv** preferably.

3.6. 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()
```

```
15
16 print("Derived clk: %.4f \t Estimated clk: %.4f " %(deriv_clk, estim_clk))
17
18 #initialize the vector voltmeter and timestamp subsytems
19 integ_time = 1.*10**-3
20 roach.init_vv(integ_time)
21 thresh = 10**-4
22 roach.init_time(thresh)
23
24 #Time stamp sanity check
25 print("Timestamp hour:")
26 print(roach.get_hour())
27 print("Unlock ? "+str(roach.get_unlock()))
28
29 #vector voltmeter sanity check plots
30 roach.adc_snapshot()
31 roach.create_plot()
32 roach.generate_plot(plots=['phase', 'spect0', 'spect1'], freq=[45, 55])
33
34
35 #stops the code to prepare the calibration setup
36 while(1):
37    tmp = raw_input('Are you ready to perform the ADCs calibration (y/n)?')
38    if(tmp=='y'):
39       break
40
41 ip_gen = '192,168.0.33'
42 roach.calibration(load=0, man_gen=0, ip_gen=gen_ip)
43
44 #stop the code to disconnect the calibration setup
45 while(1):
46    tmp = raw_input('Continue (y/n)?')
47    if(tmp=='y'):
48       break
49
50
51
52 #Explore the spectrum to see where is the signal... here I suppose that is in 50MHz.
53
54 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..
55 ADC1_peak = np.argmax(roach.get_adc1_spect())
56 chann = roach.get_index(50)     #chann where I think the signal should be
57
58 print("ADC0 peak index: %i \t ADC1 peak index: %i" %(ADC0_peak, ADC1_peak))
59 print("My channel index: %i" %(chann))
60
61 #suppose that everything seems good
62
63 roach.generate_plot(plots=['chann_values'], chann=chann)
64 #look how the channel evolves in time
```

```
65
66 #start the PowerPC storage..
67 roach.ppc_upload_code()
68 duration = 20          #we store 20 minutes
69 roach.ppc_meas(chann=chann, duration=duration) #starts the measurement
70
71 roach.ppc_check_status()
72
73 #if everything is working we look at the real time evolution again
74 roach.generate_plot(plots=['chann_values'], chann=chann)
75
76 ###
77 ##suppose we finish the measurement, the script doesnt follow until we
78 ## close the animation window
79 ##
80 roach.ppc_finish_meas()    #kill the measurement process in the PowerPC
81
82
83 my_ip = '192.168.0.29'
84 roach.ppc_download_data(my_ip)
85 print(os.system('ls'))    #we should see the raw data in our directory
86
87 roach.parse_raw_data()
88 print(os.system('ls'))    #now we should have a hdf5 file.
```

3.7. Note about PowerPC codes

This subsection shows how to manually set the measurements in the PowerPC. It is not necessary to read it, but because the code was not tested on every OS distribution, it could be a useful reference.

Because the data is stored in a temporary directory, the script must be uploaded every time prior measurement. This can be done manually by running the following commands. As example the ROACH IP will be 192.168.0.40 and the IP of the local machine will be 192.168.0.29

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

```
1       telnet 192.168.0.40
2          Trying 192.168.0.40...
3          Connected to 192.168.0.40.
4          Escape character is '^]'.
5
6       roach021417 login: root
7
8       Consult /root/README for a basic guide
9       ~ # cd /tmp/
10
```

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

```
1       ~# nc -l -p 1234 > ppc_save
2
```

3. Open a second terminal in a 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. Check that the executable file is now in the ROACH tmp file, using this code. To run it first change the permissions and 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 times that this script should read the two BRAMs. To calculate the total time needed for 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 use following code, where N is the value you calculate previously.

```
busybox nohup ./ppc_save N

```

This will print several messages in the terminal, reporting that a read was performed and in which of the two memories the system is reading.

7. To kill the process with the kill command, there is a catching for this signal. When the process is finished the data file can be transferred by running in a local computer:

```
nc -l -p 1234 > raw_data

```

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

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

```

Then a copy of the binary file should be located in your local computer.