MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Academics : Industrial Technologie (TIN)

WATCHMAN : Data Acquisition system

Realized by

Anthony Schluchin

Professor
Marco Mazza, HEIA-FR
marco.mazza@hefr.ch

External Experts
Gary S. Varner, University of Hawai'i at Manoa
varner@phys.hawaii.edu

Kurtis Nishimura, University of Hawai'i at Manoa
kurtisn@phys.hawaii.edu

Honolulu, HES-SO//Master, 2019

Accepted by the HES-SO//Master (Switzerland, Lausanne) on the proposal of

Prof. Marco Mazza, supervisor of the Master Thesis
Gary S. Varner, main expert

Honolulu, February the 8th 2018

Prof. Marco Mazza
Supervisor and professor at the HEIA-FR

Prof. Gary S. Varner
Main expert

Acknowledgements

I would like to address my warmest thanks to the IDLAB crew for welcoming me to Hawaii and for the great moments spent at the TGI's Friday.

A special thanks to my supervisors Professor Gary Varner and Kurtis Nishimura for all the advice and time devoted to me.

To my professor Marco Mazza, a great thank for giving me the chance to do my thesis lost in the middle of the Pacific Ocean on a pleasant island.

A big thanks go to Jonathan for his permanent positive spirit and endless motivation to go surfing.

Finally, I would like to sincerely thank my family and entourage who encouraged me from the other side of the Earth.

Abbreviations

ADC	Analog to Digital Converter
API	Application Program Interface
APU	Application Processor Unit
BSP	Board Support Package
DAC	Digital to Analog Converter
DAQ	Data AcQuisition
DDR	Double Data Rate
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
GIC	Generic Interrupt Controller
IACT	Imaging Atmospheric Cherenkov Telescopes
IP	Internet Protocol
lwIP	Lightweight IP
netif	Network interface
OS	Operating System
PCB	Protocol Control Block
PMT	PhotoMultiplier Tubes
SCU	Snoop Control Unit
TARGET	The TeV Array Readout Gigasample-per-second Electronics with Trigger
TCP	Transmission Control Protocol
TTC	Triple Timer Counter
UDP	User Datagram Protocol
UH	University of Hawai'i
WATCHMAN	WATer CHerenkov Monitor for Anti-Neutrinos

Table of contents

1. Introduction

The WATCHMAN project is a collaboration of different universities all over the United State of America and the United Kingdom. The different institutions involved in the project are: Lawrence Livermore National Lab, UC Berkeley, Brookhaven National Lab, University of Hawaii, Iowa state, Penn state, University of Pennsylvania, University of Sheffield. The neutrino detector will be located at 25 kilometers away from the Hartlepool Nuclear Power Station in the Boulby Underground Laboratory in England. The main goal of this project is to develop a detector capable of monitoring nuclear reactors from tens of kilometers away, in order to be part of the Nuclear Non-Proliferation Treaties[1].

1.1. Description of the WATCHMAN neutrino detector

The fission reactions in the nuclear reactors produce *anti-matter* neutrinos, and their interaction with hydrogen atoms makes a super-fast light flash that can be detected and identified. The probability of this interaction occurring is really small, but with a large number of neutrinos (nuclear reactors – 10^{20} neutrinos per second) the probability is increased.

The design of the WATCHMAN detector is based on the different elements needed for this interaction:

- Lots of water (H_2O)
- Super-fast light flash detector (PMT)
- Capture agent to chemically dope the water (Gadolinium)

Due to the natural radioactivity coming from the sun and the galaxy at the surface of the earth, the detector must be constructed deep underground.

Figure 1-1 : concept of the WATCHMAN detector

The detector will consist in a tank of 20 to 25 meters in diameter with a height of approximately the same size, filled with pure water doped in sulfate gadolinium. Its location will be near the Boulby Laboratory, more than 1 kilometer depth underground. Finally, there will be an inner structure composed of 3500 to 4500 large PMTs. In order to filter cosmic ray particles coming from the surface and to reject the gamma rays generated by the rocks and steel components, there will be an outer veto.

[1] An international treaty with the objective to prevent the spreading of nuclear weapons

1.2. Description of the data acquisition system

The signal to digitalize coming from the PMTs is a small pulse with an amplitude of some mV and a duration of some ns, therefor the DAQ must be fast and have good resolution at the same time.

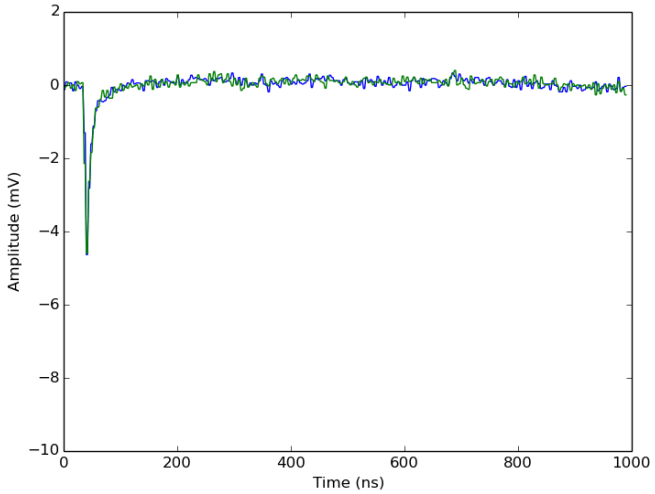


Figure 1-2 : Example of pulse coming from a PMT

The development of the data acquisition system represents an import part of the project in terms of money and time. In order to minimize the price of the detector, three groups work on the design of a system with high performances and low costs. Each group has its own design. One group is using “homemade” ADC and the others are using industrial solutions, which are slower and more expensive, but they need less time for the development.

1.2.1. CAEN system

CAEN system solution is based on industrial ADC from Renesas, the ISAL214P50 which is 14 bits resolution ADC going up to 500MSPS. To collect the data from this converter, CAEN group uses a Cyclone IV DAC from intel. They use an optical link with a bandwidth of 80MB/s to send the data to the computer for a post-processing.

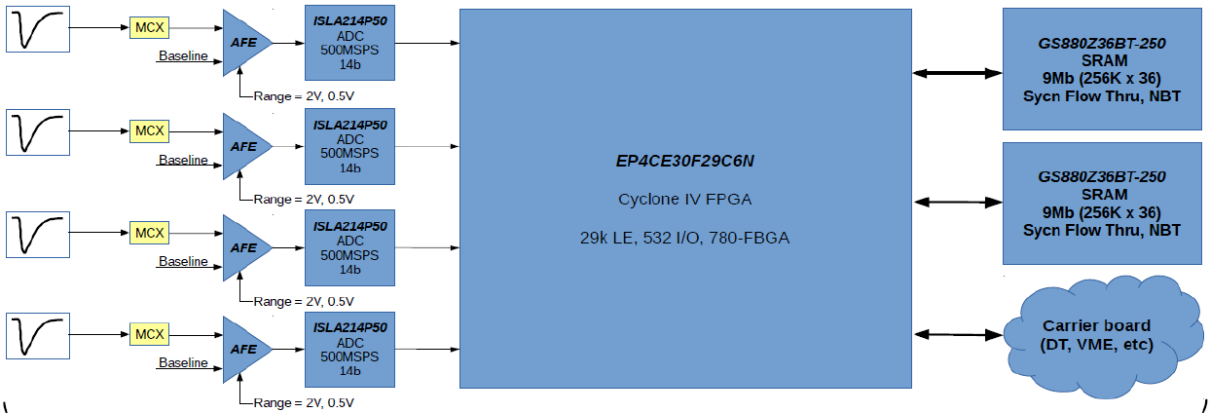


Figure 1-3 : CAEN system

This system has no trigger, which means the ADC is continuously digitizing, because of this they need a special firmware (DPP-DAW) to filter the data and find the pulse. The disadvantage of the continuous digitalization is the power consumption, around 15kW/PMT.

The cost for this system would be around 1200$/PMT, including the development and the material.

1.2.2. ANNIE

For the ANNIE system, this group is using an industrial ADC from TI, the ADS5407 which has a resolution of 12 bits and can go up to 500MHz. This converter has two channels. To recover the data, a FPGA Arria V from intel is used. For the transmission of the data to the computer, they have a second board with 18 SFP ports connected to the multigigabit transceivers of a FPGA Arria V.

Figure 1-4 : 16 channels board

Figure 1-5 : Optical fiber center

As for the CAEN system, this one does not have a trigger, and therefore the ADC is continuously sampling. The firmware used to recover, process and store the data on the computer is based on the ToolDAQ software.

The expected cost for the ANNIE system, including the development, the material and some unforeseen expenses, is around 770$/PMT.

1.2.3. University of Hawai'i – Manoa

The WATCHMAN readout module of the UH is based on an ASIC technology called the TARGETC. This ASIC has been developed by Prof. Gary Varner at the University of Hawai'i. This ASIC is a 16 channels Wilkinson ADC which can manage a sampling rate of 1 Giga-samples/second with an external trigger and an external memory manager. Compared to the industrial ADCs working in the same range, the TARGETC is quite low-power.
To manage this ASIC, a MicroZed Board is used. This board is a commercial System-On-Module with a Zynq. A Zynq is composed of a Programmable Logic side and a Processing System side. The FPGA will be in charge of generating the clocks and signals to control the TARGETC and recover the data. The PL side will also oversee the trigger system. The ARM will process the data in order to extract the important information such as the amplitude and the time of a pulse, these are what we call features extraction. The ethernet communication to send the data to the computer will also be handled by the processor

Only 4 PMTs are connected to the TARGETC, because every PMT's output is distributed to 4 gain stages (x10, x1, x0.1, x0.01), so the input dynamic range of the digitalization is increased. Every x10 gain stage is equipped with a comparator to trigger the pulse coming from the PMT.

This module has several advantages over the other two systems. It uses a power efficient ADC which has a bigger sampling rate and it is a low-cost solution, around 200$. The development time is the only disadvantage.

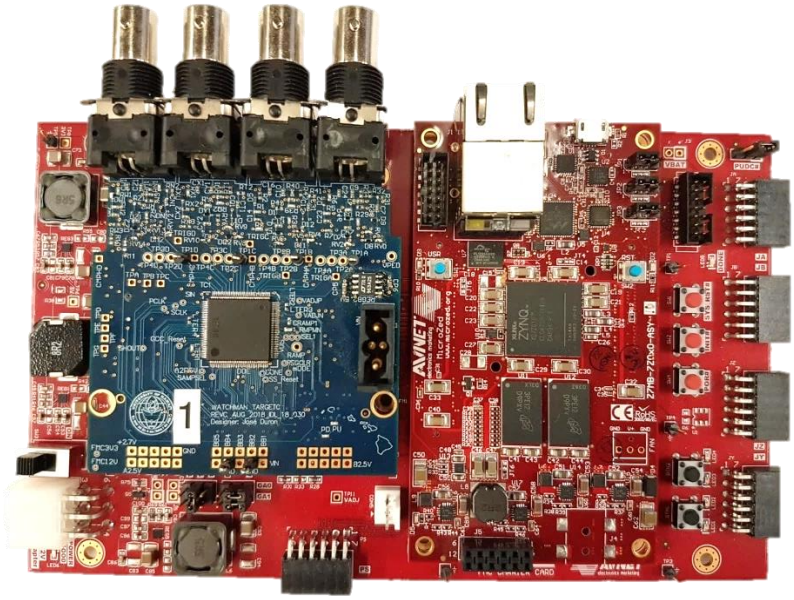


Figure 1-6: The Hawaii WATCHMAN readout system

The prototype system is composed of a FMC Carrier board on which one the Microzed and the first FMC prototype board equipped with the TARGETC are plugged.

1.3. Project description

1.3.1. Collaboration

The project WATCHMAN TARGETC is a collaboration of three students under the supervision of Dr. Kurtis Nishimura and Dr. Gary Varner. Each branch of the project is affiliated to a student:

- Hardware: Jose Duron
- Firmware: Jonathan Hendriks
- Software: Anthony Schluchin

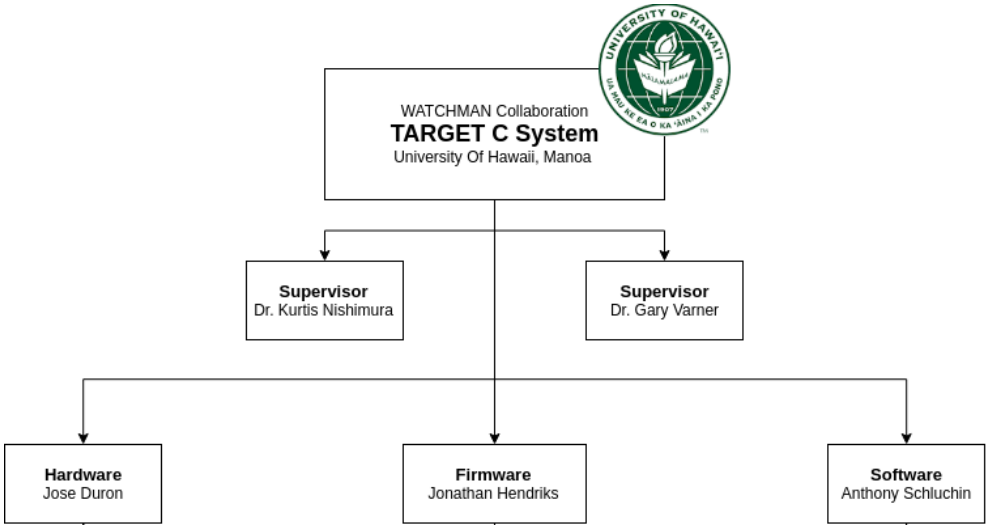


Figure 1-7: Team's organization and responsibilities

1.3.2. Objectives

The following objectives were defined before arriving to Hawaii:

“Prototype and evaluate a Data AcQuisition (DAQ) system for WATCHMAN readout. This will involve developing utilities to collect data from the distributed set of front end cards and archiving for offline analysis.

Coding to support online Data Quality monitoring and error reporting will also need to be developed and verified. Configuration scripts for initializing and managing data taking runs will be prototyped, with documentation to support changes in the future as operational needs require.

Given the remote location, auto-restart from a fault condition, such as power loss, or loss of network access, will be very important. Therefore stress testing of the start-up scripting will be an important task.”

After discussion on the first days, these objectives were too specific for the current status of the project. The system was at an early stage of development. Therefore, new objectives were defined.

- Ethernet communication
  - Evaluate the protocol UDP
  - Evaluate the lwIP stack
- General User Interface
  - Interface with the system (commands and data)
  - Data storage
  - Real time analysis
  - System configuration
- Autonomous system
  - Failure detection
  - Log file
  - Self-reboot
- Data processing
  - Pedestal subtraction
  - Transfer function correction
  - Windows stitching
  - Sample timing
  - Feature extraction
- Software documentation

2. TARGETC

The TeV Array Readout Gigasample-per-second Electronics with Trigger (TARGET) C is a new application-specific integrated circuit (ASIC) of the TARGET family. This circuit is designed by Dr. Gary Varner at the University of Manoa. This ASIC is designed for the readout of signals from photosensors in the cameras of imaging atmospheric Cherenkov telescopes (IACTs) for ground-based gamma-ray astronomy. The TARGET family is composed of:

- TARGET5
- TARGET7
- TARGETX
- TARGETC

2.1. Overview

The TARET C is based on the Wilkinson ADC (Analog to digital converter) which compares the analog input value to a controlled ramp. During all the ramp increase a counter is incremented. Once the analog input is lower that the ramp's value, the counter is stopped or its value is latched into a register. The digital value of the counter is the analog input representation in binary format.

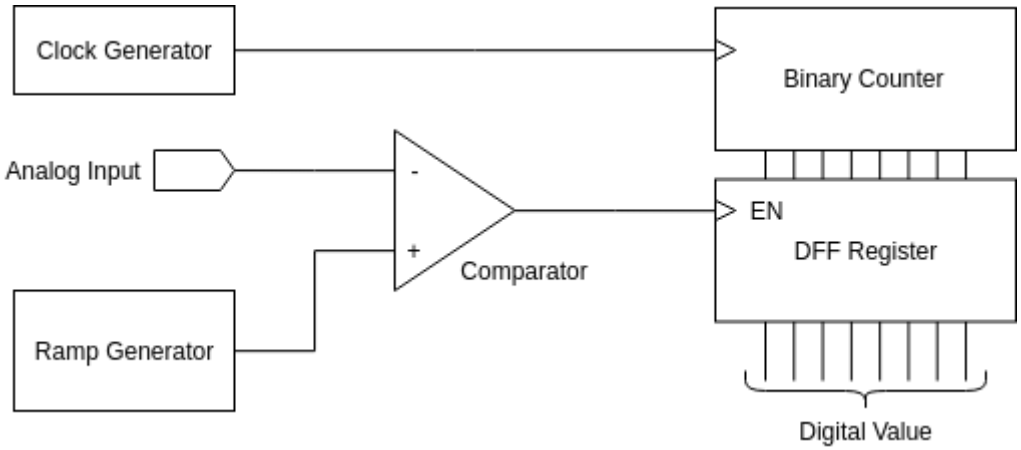


Figure 2-1: Wilkinson ADC principle

The Wilkinson ADC is only one the digitization part the ASIC beforehand there is the sampling stage composed of 64 capacitors. The principle built in the TARGET is multiple delay lines, each of them is connected to a different capacitor in this sampling array. The delay between lines are adjusted to respect a 1ns delay. As a consequence, each capacitor will be enable at 1 ns of interval and therefore the sampling rate is 1 Giga-Samples per second.

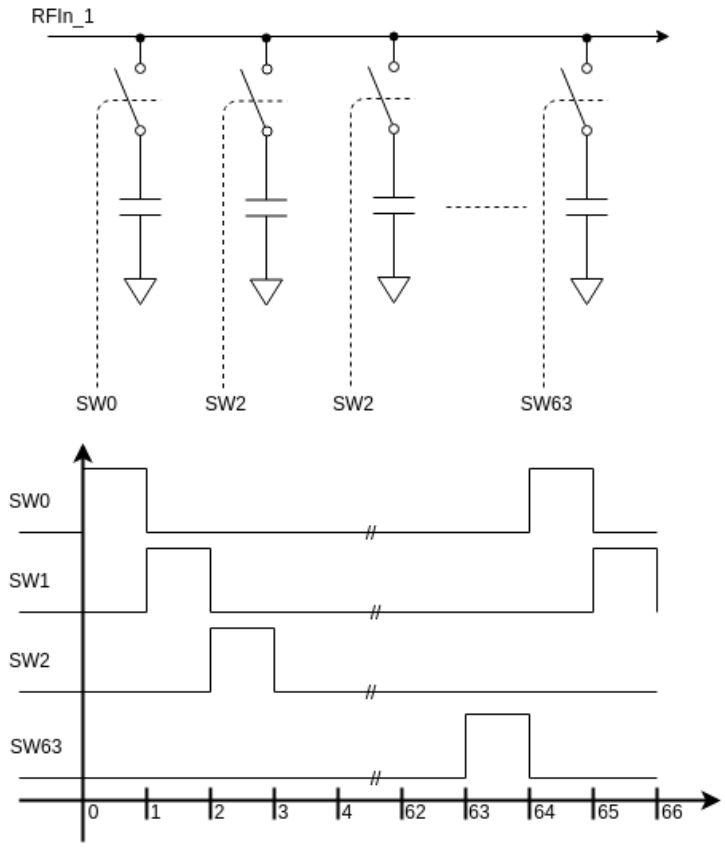


Figure 2-2: TARGETC capacitor sampling array

To store the samples into the analog memory, some internal signals are asserted. The sampling continues after a full cycle, like seen in the *Figure 2-2*. To not overwrite the same storage cell, the storage address is updated after each period of 64 ns. The storage address is an 8 bits wide data line, so a total of 2^8= 256 storage locations and a total analog memories storage of 256 * 64 * 16 = 16K * 16 = 256K.

The data is stored in memory by window of 32 samples, therefore the EVEN (Sample 0 to 31) and the ODD parts (Sample 32 to 63) are distinguished. Internally to the ASIC, this distinction is done by an extra bit added to the storage address.

In case of a trigger event, a selected window is read. The read address is this time a 9 bits data line which includes the extra bit. Once the address is sent to the ASIC, the Wilkinson digitization begins. Each 32 cells of the channel for this particular window is being compared to the increasing voltage on the Wilkinson capacitor. The maximum conversion time is defined by the width of the result (11 bits) and frequency of the counter.

$$Time = \frac{1}{f_{wilkinson}} * 2^{width}$$

Once the digitization process is finished, the samples from each channel are shifted out on a serial interface.

2.2. New features

Previous design worked with an internal trigger system indicating that an event was recorded and that it should be processed. On the TARGETC, the mechanism has been removed, it is up to the designer to integrate a trigger detection system.

In consequence, the WATCHMAN TARGETC prototype board uses four operational amplifiers mounted as comparators. Four independent and adjustable threshold voltages are compared to the highest gain lines (x10), which are Channel inputs 3, 7, 11 and 15.

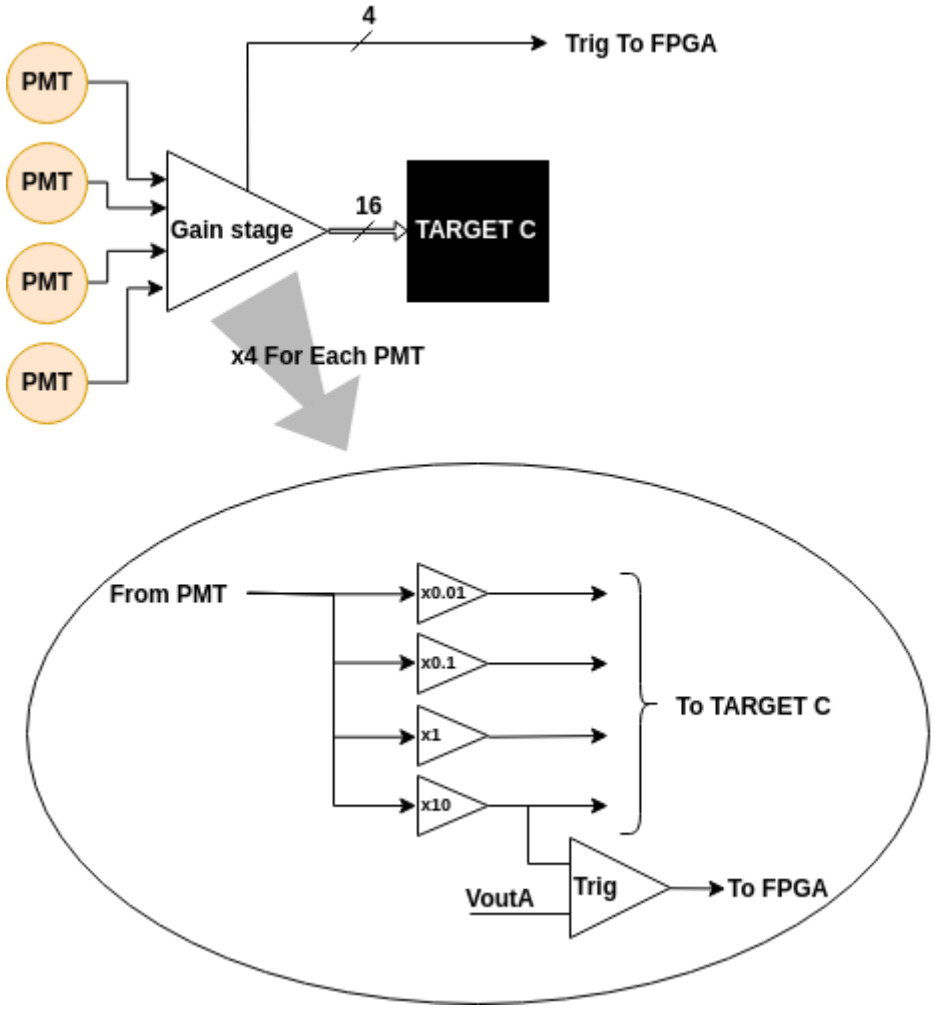


Figure 2-3: Gain stages and trigger system of the WATCHMAN prototype board

The other difference with earlier designs is the storage management unit. TARGETX for instance has an internal flag Write Enable, which is tested at the moment of storing the data in memory the flag is tested. If the storage address is unavailable the system will not overwrite the location, but this also means that the data inside the sampling cells will be lost, so this slice of time is lost. For the storage system, it is up to the user to keep track of which storage address is available or not.

3. Zynq: PS side

For this project a Zynq was the solution, because it has a PL side which is able to manage the homemade protocol of communication of the TARGETC, and a PS side which can make the system autonomous, because of its remote location, and to process the data and send them to a computer. This chapter covers the tasks done on the PS side.

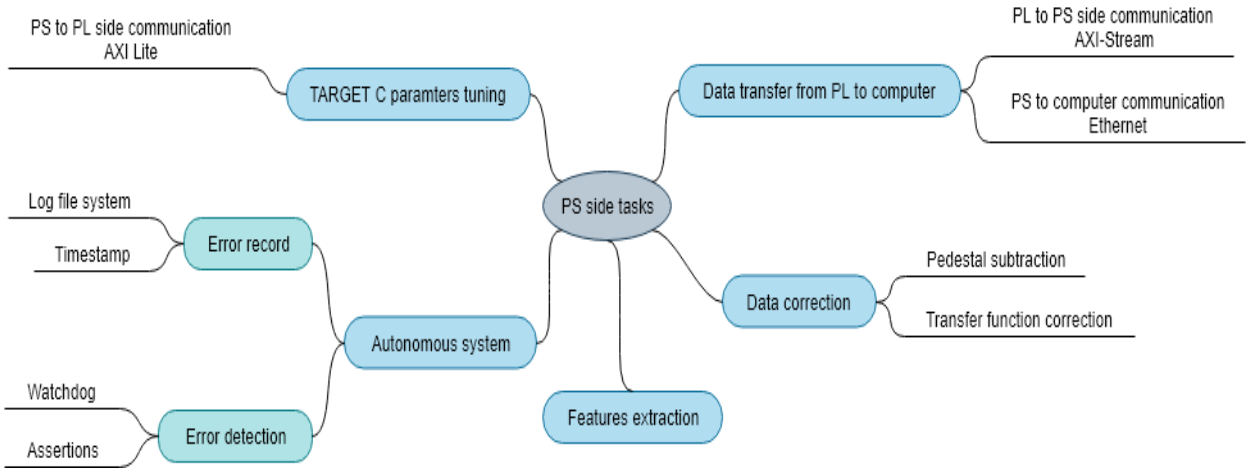


Figure 3-1: Mind map of the Zynq PS side tasks

3.1. Data transmission

The TARGETC has 16 input channels. After digitalization, it transfers the data measured on all the channels simultaneously through 16 serial lines. These data are then recovered by the PL side of the Zynq and stored into fifo until the PS is ready. Finally, it is the PS side that is in charge of getting them to the computer.

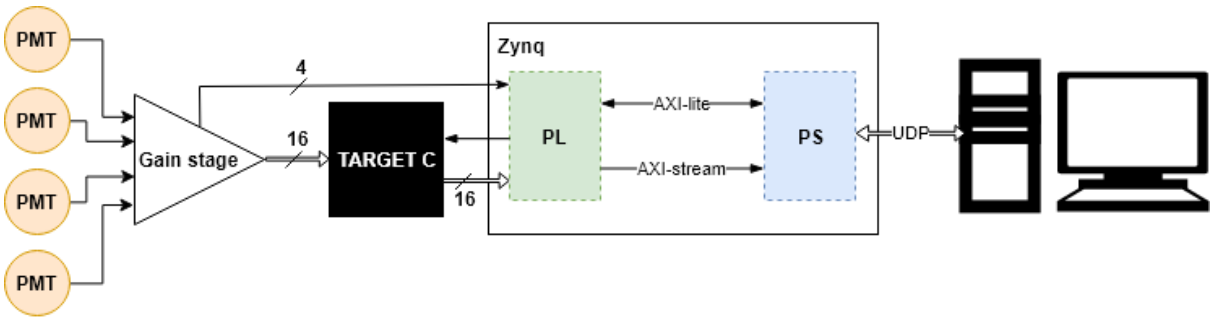


Figure 3-2: Complete data flow

3.1.1. PL to PS transmission

To communicate between the PL and the PS side of the Zynq, two protocols are used, the AXI-lite and the AXI-stream. The AXI-lite is easier and faster to set up, but its bandwidth is limited. For this reason, the AXI-stream is necessary for this task.
As shown in the *Figure 3-3*, the AXI-stream is not directly connected to the ARM. First, it goes through the AXI-DMA IP core which provides high-bandwidth memory access. The signals coming from this IP are connected to a High Performance port of the PS. This port is the link with the AXI4 bus which can access the DDR through the memory interface.

Once the data are in the DDR, the ARM can read them using the same bus AXI4 and memory interface component. The location in the DDR where the DMA has to store the data is given at the beginning by ARM. It uses the AXI-lite to communicate this address.

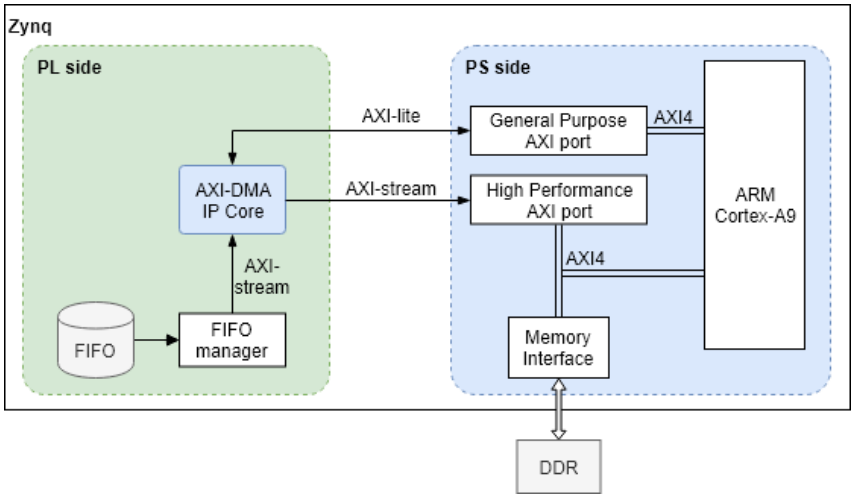


Figure 3-3: PL to PS transmission overview

3.1.2. Ethernet communication

In order to communicate between the computer and the Zynq with the protocol UDP, the use of ethernet was deemed the best choice. In this project, the communication is managed with lwIP which is a light-weight implementation of the TCP/IP protocol. Its goal is to reduce the resources used while keeping the full-scale TCP stack and it is perfect for embedded system with limited RAM.

To make it easier, and because there is no need for it, the DHCP module is not used, the IP addresses are static. The Zynq's address is fixed to 192.168.1.10 and the computer's one to 192.168.1.11. There are two types of frame in this communication, on two different ports. The first ones are the commands, send by the computer on the port 7, and echoed back by the Zynq. The other ones which contained the data recovered from the TARGETC are transmitted on the port 8.

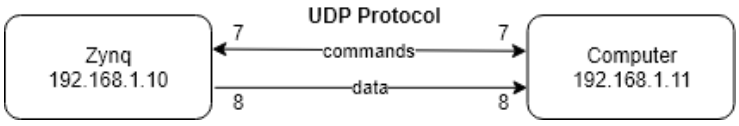


Figure 3-4: Ethernet network

The UDP protocol has been chosen because it is faster than TCP which is slowed down by all its error-checking processes, and also, it does not need to establish a connection to send data. To remedy this problem of losing packet with UDP, every command frame has a random number in it. This number can be controlled when the frame is echoed back by the Zynq, to be sure that the frame received is not a response to a frame sent previously, containing the same command.

3.2. TARGETC parameters

In order to work correctly and to be compatible with the system in which it will later be used, the TARGETC has many parameters that the user can tune. All these parameters are represented by 128 registers which are accessed by a serial communication managed by a HDL component in the PL side.

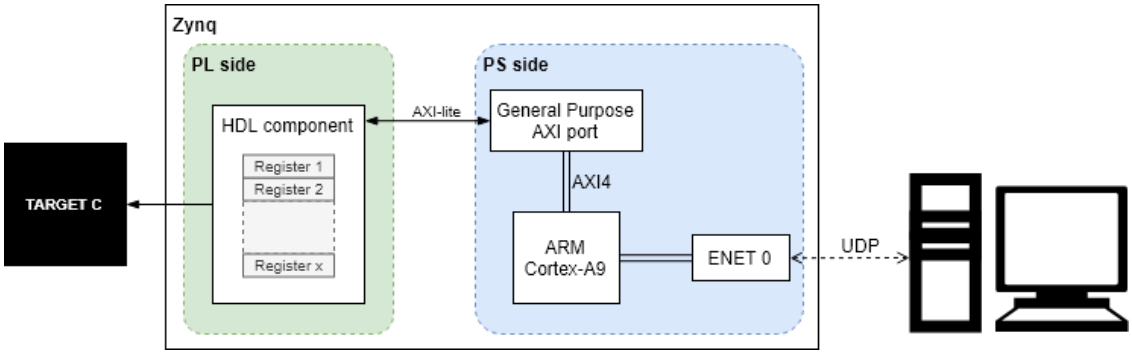


Figure 3-5: TARGETC parameters

This component receives the values of these registers from the PS side by AXI-lite. From the ARM's point of view, this AXI-lite communication appears as a simple array whose size is equal to the number of registers declared in the HDL component.

3.3. Autonomous system

The detector will be located 1km underground, so the system must be able to face and deal with some eventual problems, such as loss of power. If a problem occurs, the error needs to be detected and recorded. Then the system should either solve the issue by itself or reboot.

3.3.1. Error detection

The error detection, which is in an independent system, is very important. If an irregular situation arises and the application does not see it, it will end up by freezing, which will require a manual reboot. To prevent this scenario, an error detection system must be implemented in the software and it should withstand three kinds of error listed below.

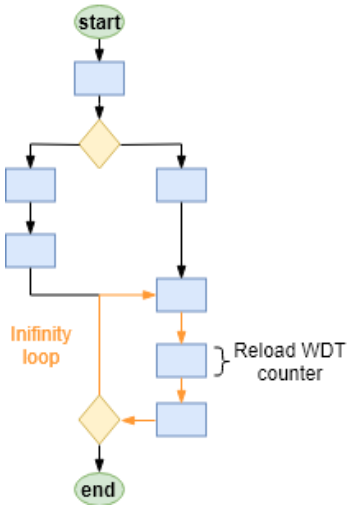


Figure 3-6: Application froze

The first error is a frozen application. It could be caused by different scenarios. For example, a bit flip occurred in the memory on the address of the PC counter. Then it is lost in the execution of the software. The only way to get out of this situation is by rebooting the system. Therefore, a watchdog must be implemented.

The watchdog works almost as a timer. Its counter is loaded with a certain value which will decrease at a certain frequency. Once there is an overflow, it means the counter is equal to 0, an interruption is raised. However, with a watchdog, there is no callback function, the system will simply reboot. To avoid the overflow of the counter, it must be reloaded periodically (see *Figure 3-6*). So, if the execution of the application is stopped, it would not manage to reload the counter and the processor will automatically reboot.

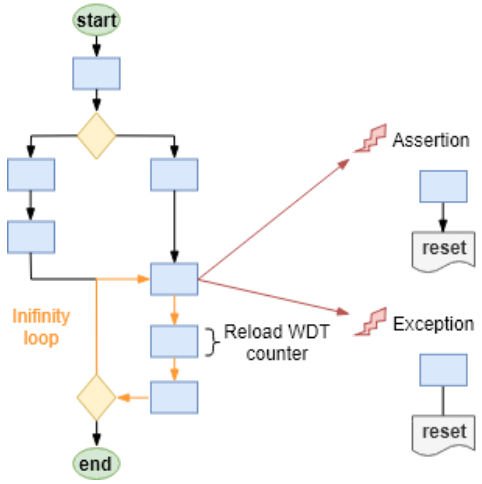


Figure 3-7: Assertions / Exceptions

The second error is a bug in the program. Most of them are discovered and resolved during the development, but some can appear only under certain conditions. A solution to detect them are the exception and the assertions which are implemented in some of the libraries included in this application, like the one for lwIP.

An assertion is a Boolean test at a specific point in the source code. It gives a true response, unless there is a bug in the program. If the response is false, the program is interrupted, and the function related to the assertion is called (see *Figure 3-7*). In this callback, the information related to the bug can be logged to solve the problem later and then the execution of a software reset is done to reboot the system.

An exception is an event occurring during the execution of a program which disrupts the normal execution of the program. Same as for the assertion, information can be logged during its callback which also ends with a software reset.

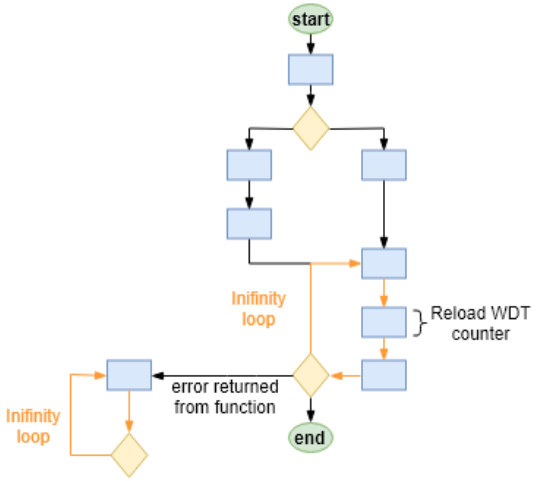


Figure 3-8: Error from a function

The last error is a function which has not worked correctly. As for the assertions, these errors will appear only under certain circumstances.

When a function does not work well, in most cases, it will return a value indicating which kind of error. By logging this return value, it will be possible to prevent this error from happening again. When this situation occurs, the program enters an infinite loop waiting on the watchdog to arise and reboot the system (see *Figure 3-8*).

3.3.2. Error record

Detecting errors in a program is important, but it is useless if they are not registered somewhere in order to resolve them. The best way is to use a log file. This file must be saved in a static memory, so that it is not lost in the event of a power failure or after a system reboot. In our system, it is stored in the SD card, which makes things easier for the user to recover it. The data stored in this file must contain sufficient information on the errors that have occurred, in order to correct the software accordingly. The three main questions that need to be answered are where, why and when did the error occur.

- **Location**: filename, function name and line number
- **Kind of error**: watchdog, assertion, function's bug
- **Timestamp**: hour and date

Because this program is a bare metal application[1] and the system is not connected to Internet, a homemade timestamp was set up. Just after booting the system, the user can send the timestamp from the computer and then the application will use it with the global timer counter, to determine the timestamp at any given moment. The time is stored in a file on the SD card and updated every second, so that in case of reboot the application can recover it without the user having to send it back.

3.4. Data correction

The data received by the TARGETC are considered as raw data, therefore they have to be corrected before sending them to the computer. Every analog memory location has its own capacitor, and they are all a little bit different because of their construction and the silicium. When the values stored in these capacitors are digitized, even if they all measured the same voltage, their responses are different because they do not have the same voltage.

The first correction to apply is the pedestal subtraction which is like an offset. The same voltage (V_{ped}) is fixed on every channel and it is measured with all the windows. This action is repeated several times, and at the end the average for every cell is calculated and stored in an array. This stored value is the pedestal value for this particular memory cell, hence every memory location (512*16*32) has its own corresponding pedestal value. Then, to apply the pedestal correction, the equation is:

$$V_{dig} = V_{measured} + V_{ped} - pedestal$$

Once the pedestal subtraction is applied, the second correction can be characterized. To digitize the analog value, the TARGETC compares the voltage of the cell's capacitor with the voltage on an external capacitor. Beside every memory capacitor is a comparator which also depends on the construction and the silicium. Consequently, they do not behave the same way, and so their transfer function is different. The second correction is therefore the linearization of the transfer function.

3.5. Features extraction

They are two kind of data frame that are sent to the computer. The first one is the complete waveform received from the PL side. The full waveform is sent only when the pulse duration was too long, which represent less than 1% of the measurements. The FPGA measures the duration of the trigger, evaluates the pulse width (normal or long) and then communicates it to the ARM with the appropriate data packet.

For the normal pulses, two features are extracted from the waveform and sent to the computer. The first feature is the "maximal" amplitude, which is in fact the minimum voltage because the pulse is negative, see *Figure 3-9*. The second one is the moment when the pulse has reached 20% of its maximal amplitude. The $V_{20\%}$ represents 20% of the difference of V_{ped}.

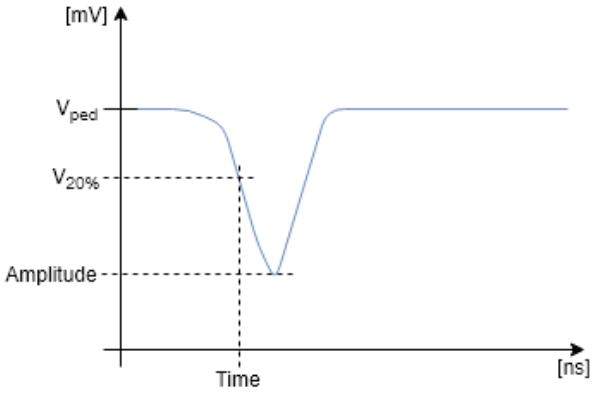


Figure 3-9: Features extracted from the pulse

[1] A bare metal application is an application without an operating system

4. Development

All the tasks described in *chapter 3: Zynq: PS side* were implemented in the ARM's software. This chapter contains the development's explanation of these tasks in the programming language C.

4.1. Ethernet communication

4.1.1. Previous work

The application started from an example called "lwIP Echo Server" provided by Xilinx. This application's example is a simple demonstration of how to use the lwIP stack. It is a server listening on port 7 and echoing back every data received with the protocol TCP. The first thing to do was to modify this example from a server TCP to a server UDP. The functions provided by the library "tcp.h" almost all exist in the library "udp.h", but with a difference in the name. For example, the function "tcp_new_ip_type" becomes "udp_new_ip_type". Then the big difference between these two protocols is that TCP establishes a connection before sending data, and the UDP does not.

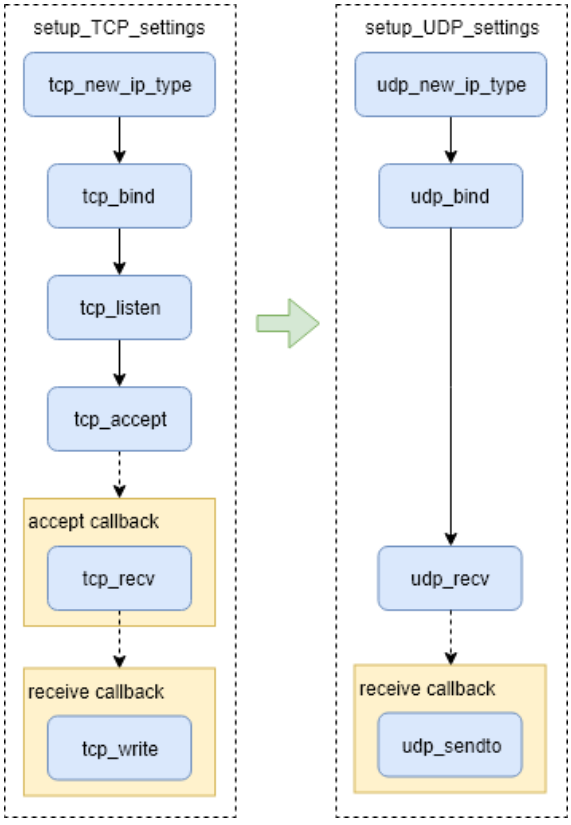


Figure 4-1: TCP vs UDP

The function "x_new_ip_type" allocates and returns the Protocol Control Block (PCB) variable for the communication.

Then, this PCB structure is linked to a port of the Zynq with the function "x_bind".

As previously explained, the connection establishment is only present for TCP. The function "tcp_listen" sets the connection's state in listening mode, and the function "tcp_accept" specifies a callback function for an accepted connection.

Finally, the function "x_recv" is used to attach a callback function when data are received. This callback has for argument the emitter's information and the data received. So, in the receive callback, the sending function is called by passing the same arguments to it.

To verify if the UDP echo server is working, a simple program in Python has been developed to send data through UDP and print the outgoing and incoming data. The result is that the data received are the same as the data sent, and that the message has been echoed on the port 7.

```
C:\Users\antho\OneDrive\Documents\WATCHMAN\Python\Python_3.5>python lwip_udp_echo_test.py
data sent: bytearray(b'\x00\x01\x02\x03\x04')
data received: b'\x00\x01\x02\x03\x04'
('192.168.1.10', 7)
```

Figure 4-2: UDP echo server test

Then the next step was to study the implementation of the lwIP in order to understand the purpose of every function and how to use them correctly.

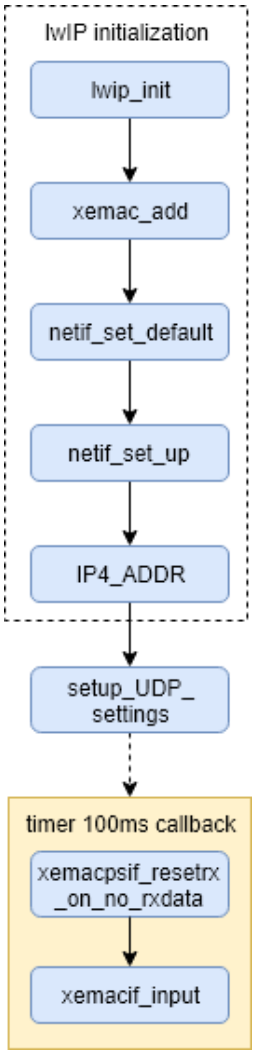


Figure 4-3: lwIP implementation

The first function, as its name suggests, initialize the lwIP stack and all its modules.

The “xemac_add” function is used to add a network interface (netif), like Eth0, to the network interface list of lwIP. It also initializes the IP address, the gateway address, the MAC address and the base address of this netif.

The function “netif_set_default”, like its name suggests it, sets the netif passed as the argument as the default network interface for lwIP.

The function “netif_set_up” sets the given netif available for processing.

When the function “xemac_add” is called, all the addresses (IP, gw, netmask) are equal to 0. To set up the right address, the function “IP4_ADDR” is used.

Finally, when lwIP is initialized, the ethernet connection can be set up.

In parallel two functions must be called every 100ms.

Under heavy Rx traffic, the Rx path can become unresponsive. If it is the case, the function “xemacpsif_resetrx_on_no_rxdata” guarantees a software reset, so the bug does not persist more than a 100ms.

The packet received on the ethernet peripheral are stored into a queue. The “xemacif_input” takes the packet from this queue and gives them to the lwIP. This event raises an interrupt and the receive callback function is called.

4.1.2. Final version

Now that the implementation of the lwIP stack and the setup of a UDP connection is understood, the complete network can be developed. It includes a bidirectional connection for the command on port 7 and a unidirectional connection that goes from the Zynq to the computer for the data on port 8, as shown in *Figure 3-4*. For the final version, the way to set up a UDP connection has been enhanced. Thanks to the function “udp_connect”, the PCB structure is no longer only bound to the port of the Zynq, but also linked to the port and IP address of the computer. After this modification, the function “udp_sendto” has been replaced by the function “udp_send” which only needs the data and the PCB that now contains the receiver’s information. Before, the buffer used to echo back the data was the same as the one containing the received data. Now this send buffer is created with the function “pbuf_alloc” which has three arguments: layer, length and type. The argument “layer” defines the header size of the frame. In this case the option selected considers the header of all the upper layers. The argument “length” defines the size of the buffer. The argument “type” defines how and where the buffer should be allocated, for this application the buffer is stored in RAM including the layers header.

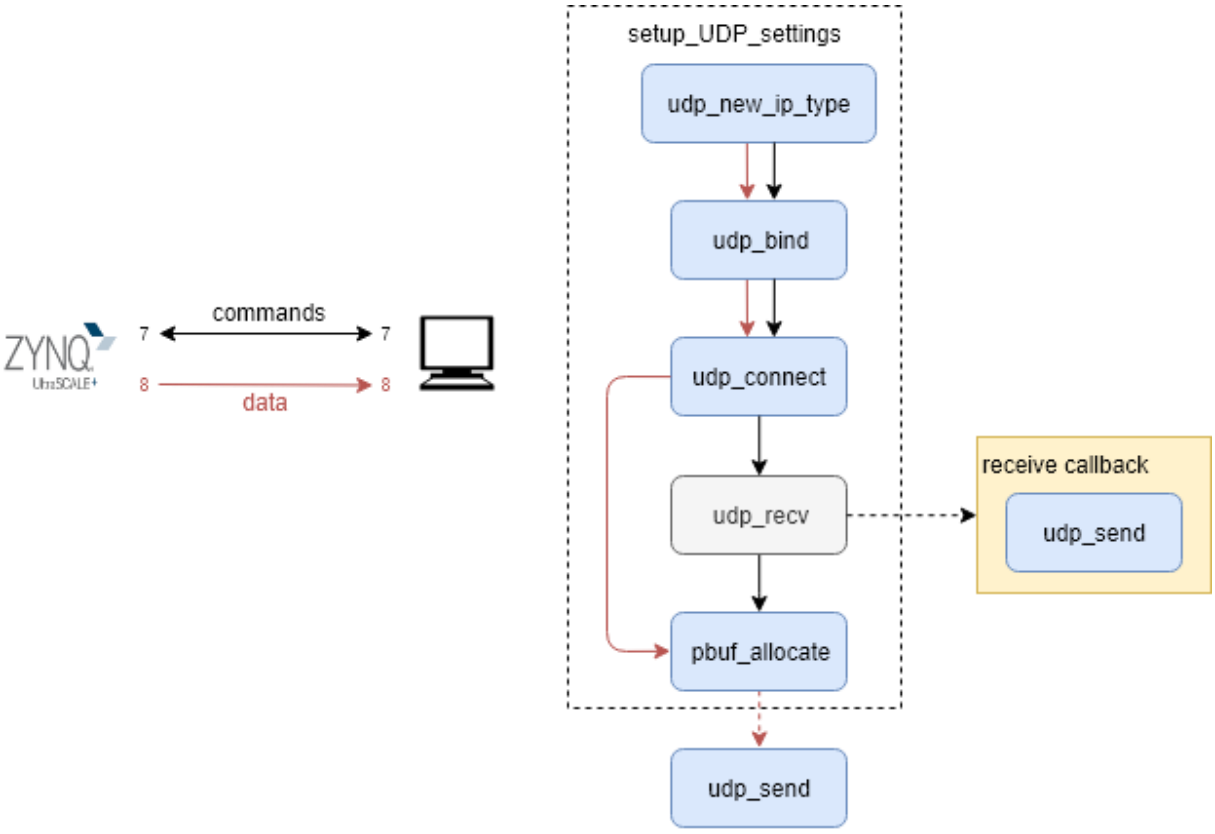


Figure 4-4: UDP setup

As shown in *Figure 4-4*, the two connections are set up the same way, except that the data connection does not receive any packets from the computer, so it is useless to implement a callback for this one.

Two schemes of communication are possible, a command between the PC and the system and a data transfer from the Zynq to the PC. In the first scheme, the ARM processor receives a packet and parses it entirely looking for the code 0x55AA which indicates the beginning of a command packet. The search continues until the end code 0x33CC is reached. These two positions delimit the command packet. The packet decoding begins with the byte after the start code. It contains the command ID, which indicates what kind of operation the processor has to execute. The following byte is a unique random number given to the packet as a frame ID. The length of the payload depends on the command.

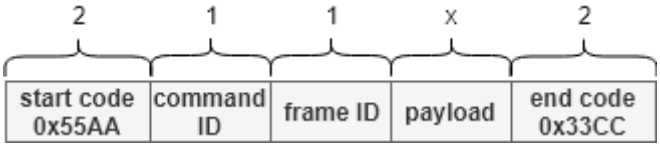


Figure 4-5: Frame command

All the commands received are echoed back by the system on the unidirectional line with the same frame content, except for the payload.

ID	Command	Description	Rx payload	Tx payload
0	Write all registers	Send the register values for the TARGETC.	256	0
1	Read all registers	Ask for a copy of the TARGETC registers values.	0	256
2	Ping	Check the connection.	0	0
3	Start / stop stream	Start / stop	0	0
4	Stop uC	Stop the system	0	0
5	Set time	Update the time	6	0
6	Get 20 windows	Ask for 20 consecutive windows	0	0
7	Get transfer function	Ask a specific set of data, to perform an offline analysis	0	0

Table 4-1: List of commands

In addition to echoing back the command 6 and 7, the system sends the requested data through the line on the port 8 (see figure below).

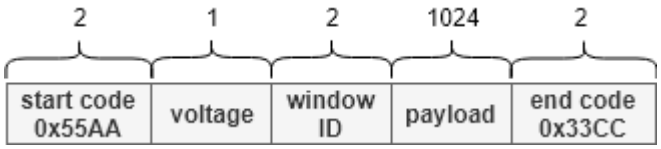


Figure 4-6: Frame data "get transfer function"

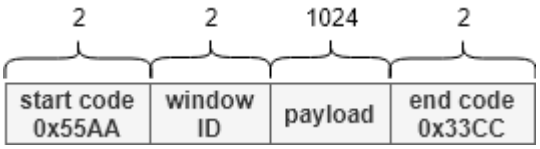


Figure 4-7: Frame data "get 20 windows"

During a normal operation, the system will have to distinguish two types of PMT pulses (as explained later in the *chapter 4.6.3 Data formatting*). The common parameters to these two types are the following:

- Length: the size of the frame (features extraction → 21 / full waveform → 2*32*nbr of window)
- Type of data: the frame ID (features extraction → 0 / full waveform → 1)
- Nbr of window: number of windows in which the pulse was measured (1 to 4)
- Channel: the channel with the best resolution
- Time window: when the first window was sampled (4ns / unit)

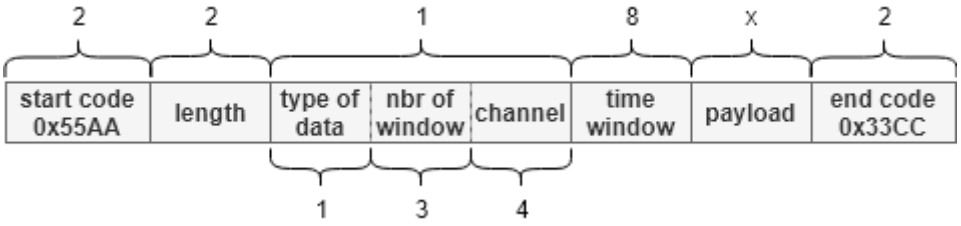


Figure 4-8: Frame data full waveform

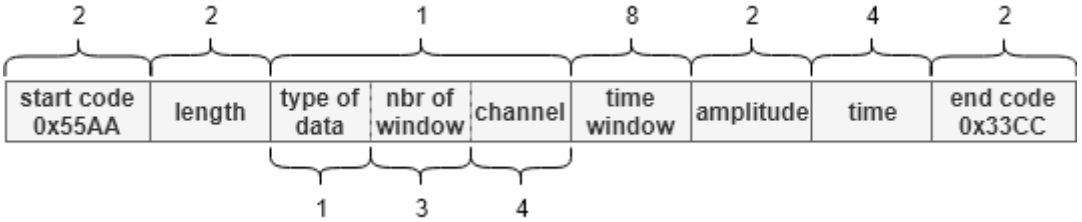


Figure 4-9: Frame data features extraction

For all the frames, if a data is bigger than a byte, the Less Significant Byte (LSB) is always sent first.

4.1.3. Remarks

After having implemented the ethernet communication, some problems with the malloc function started to occur. This function always returned 0, which means that the ARM could not allocate the memory space asked. These issues appeared seemingly randomly. For example, the function malloc could work in main.c, but not in the interrupt.c. After a few tests, it came out that the problem occurred only after the Zynq had sent its first UDP packet. The program has been run in debug mode while controlling the behavior of the memory where the buffer used to send the data was located, because if a malloc does not work, it is often due to a problem with the memory's allocations. What was seen, it is that the memory slots above the buffer were modified after a transfer. By analyzing the data stored there, it turned out that they were overwritten by the header of the upper layers, as shown in *Figure 4-10*.

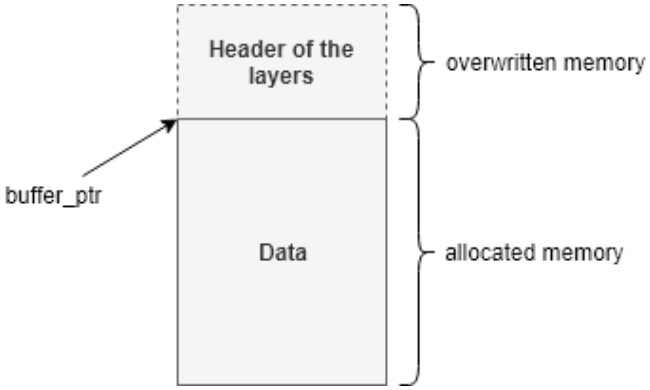


Figure 4-10: Memory's state after a UDP transfer

The buffer for the data and for the commands are initialized in the same way, using the function "pbuf_alloc" with the option PBUF_TRANSPORT for the layer argument, the maximum size of the frame for the length and the option PBUF_RAM for the type argument, as shown below.

```
buf_data = pbuf_alloc(PBUF_TRANSPORT,MAX_DATA_SIZE,PBUF_RAM);
buf_cmd = pbuf_alloc(PBUF_TRANSPORT,MAX_CMD_SIZE,PBUF_RAM);
```

Here is the explanation of the first argument.

```
typedef enum {
  /** Includes spare room for transport layer header, e.g. UDP header.
   * Use this if you intend to pass the pbuf to functions like udp_send().
   */
  PBUF_TRANSPORT,
  /** Includes spare room for IP header.
   * Use this if you intend to pass the pbuf to functions like raw_send().
   */
  PBUF_IP,
  /** Includes spare room for link layer header (ethernet header).
   * Use this if you intend to pass the pbuf to functions like ethernet_output().
   * @see PBUF_LINK_HLEN
   */
  PBUF_LINK,
  /** Includes spare room for additional encapsulation header before ethernet
   * headers (e.g. 802.11).
   * Use this if you intend to pass the pbuf to functions like netif->linkoutput().
   * @see PBUF_LINK_ENCAPSULATION_HLEN
   */
  PBUF_RAW_TX,
  /** Use this for input packets in a netif driver when calling netif->input()
   * in the most common case - ethernet-layer netif driver. */
  PBUF_RAW
} pbuf_layer;
```

And here follows the description of the third argument.

```
/**
 * @ingroup pbuf
 * Allocates a pbuf of the given type (possibly a chain for PBUF_POOL type).
 *
 * The actual memory allocated for the pbuf is determined by the
 * layer at which the pbuf is allocated and the requested size
 * (from the size parameter).
 *
 * @param layer flag to define header size
 * @param length size of the pbuf's payload
 * @param type this parameter decides how and where the pbuf
 * should be allocated as follows:
 *
 * - PBUF_RAM: buffer memory for pbuf is allocated as one large
 *             chunk. This includes protocol headers as well.
 * - PBUF_ROM: no buffer memory is allocated for the pbuf, even for
 *             protocol headers. Additional headers must be prepended
 *             by allocating another pbuf and chain in to the front of
 *             the ROM pbuf. It is assumed that the memory used is really
 *             similar to ROM in that it is immutable and will not be
 *             changed. Memory which is dynamic should generally not
 *             be attached to PBUF_ROM pbufs. Use PBUF_REF instead.
 * - PBUF_REF: no buffer memory is allocated for the pbuf, even for
 *             protocol headers. It is assumed that the pbuf is only
 *             being used in a single thread. If the pbuf gets queued,
 *             then pbuf_take should be called to copy the buffer.
 * - PBUF_POOL: the pbuf is allocated as a pbuf chain, with pbufs from
 *              the pbuf pool that is allocated during pbuf_init().
 *
 * @return the allocated pbuf. If multiple pbufs where allocated, this
 * is the first pbuf of a pbuf chain.
 */
struct pbuf * pbuf_alloc(pbuf_layer layer, u16_t length, pbuf_type type)
```

According to the explanation of the PBUF_TRANSPORT and PBUF_RAM option just above, some spare room for the transport layer header should be included. However, it was not the case, so a deeper investigation had to be done.

The structure “struct pbuf” represents a buffer of a linked list. For this application, the list has only one element, so from the structure only three parameters are useful.

- **payload**: pointer to the actual data in the buffer
- **tot_len**: total length of this buffer and all next buffers in the list belonging to the same frame
- **len**: length of this buffer

For this application, as previously mentioned, the list has only one element. So, the parameters “len” and “tot_len” have the same value and they are equal to the number of bytes to be sent. The frame’s data are not copied into the array pointed by the payload parameter, but their pointer was passed to it, which means that the payload parameter was pointing on the array containing the data.

That was the problem, because the function “pbuf_alloc” returns a pointer on a pbuf structure with its payload parameter pointing on memory location with some allocated memory above it for the layer’s header, as shown in the *Figure 4-11*. So, when the function to send the data is called, it uses the slots of memory above the location pointed, even if they are not allocated.

The right way to pass the data would be to copy them into the array pointed by the payload parameter, which is unusual and not explained in the library’s documentation.

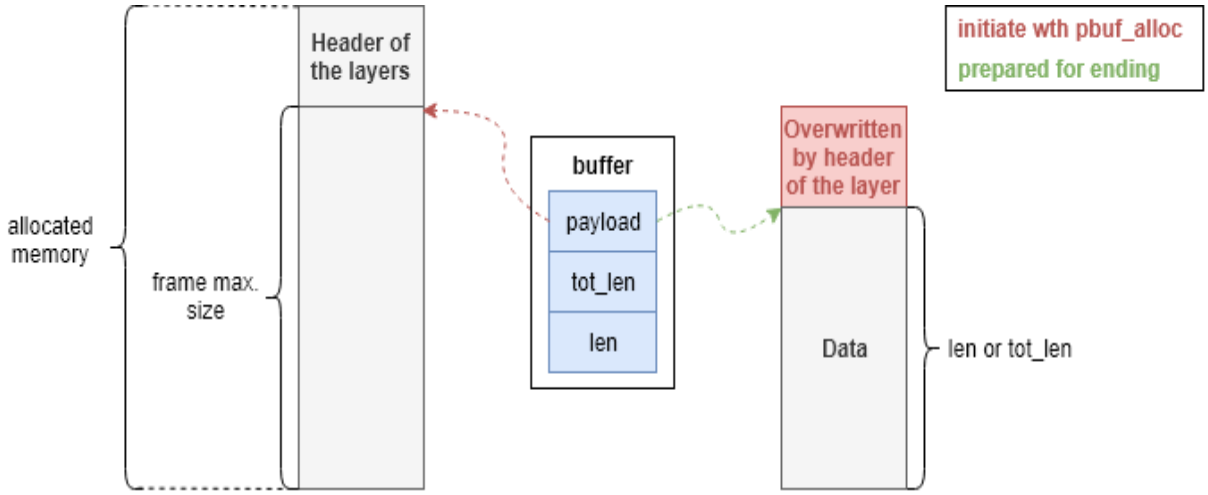


Figure 4-11: Problem with the send buffer

Another solution, to avoid the fact of copying the data from one location to another, which takes time, would be to allocate some space for the headers just above the data and make the payload parameter point on the first data.

Nevertheless, this solution is not perfect, because in the send function there is a test to see if there is some allocated space for the headers, see code and its explanation below.

Explanation of the test in the send function

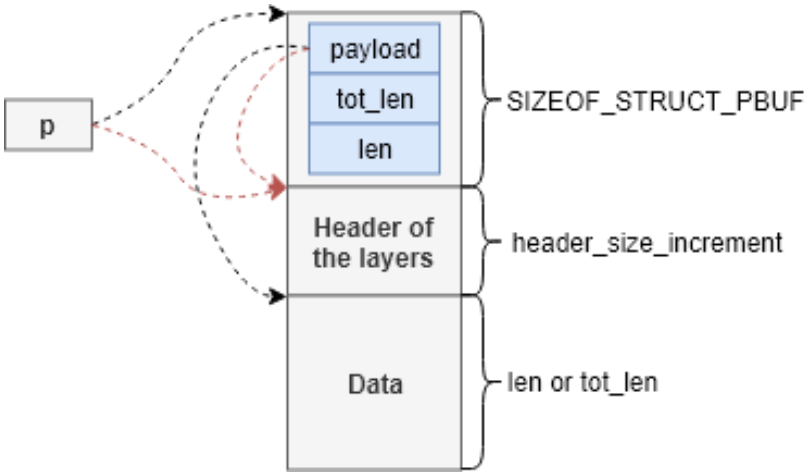


Figure 4-12: Test in send function with the right way for the header

The variable “p” is the pointer on the pbuf structure. The pointer payload is decremented by the size of the headers, and the pointer “p” is incremented by the size of the pbuf structure. After which they are compared. If the payload pointer is smaller than the pointer “p”, this means that there is no memory allocated for the headers between the pbuf structure and the data. So, if the right way to pass the data to the pbuf structure is used, this test works.

By using the second solution, the pointer payload is pointing somewhere else in the memory. Then, the fact that the pointer payload decremented is smaller than the pointer “p” incremented, depends just on its location. So if the data is declared earlier in the memory than the pbuf structure, then the test will never pass, even if there is allocated memory for the headers.

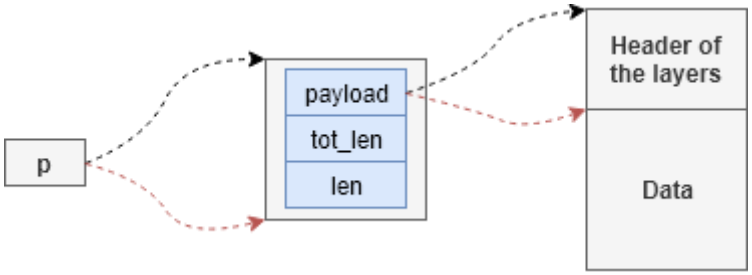


Figure 4-13: Test in send function with the other solution

```
/* pbuf types containing payloads? */
  if (type == PBUF_RAM || type == PBUF_POOL) {
    /* set new payload pointer */
    p->payload = (u8_t *)p->payload - header_size_increment;
    /* boundary check fails? */
    if ((u8_t *)p->payload < (u8_t *)p + SIZEOF_STRUCT_PBUF) {
      LWIP_DEBUGF( PBUF_DEBUG | LWIP_DBG_TRACE,
        ("pbuf_header: failed as %p < %p (not enough space for new header size)\n",
        (void *)p->payload, (void *)((u8_t *)p + SIZEOF_STRUCT_PBUF)));
      /* restore old payload pointer */
      p->payload = payload;
      /* bail out unsuccessfully */
      return 1;
    }
```

In conclusion, the final solution is to declare the array containing the data and allocating space for the layer headers above it, before calling the function “pbuf_alloc”. By doing so, the address of the data’s array is smaller than the one contained by the pointer “p” and therefore the test will pass. This solution is valid only if they are declared once at the beginning of the program to guarantee their order in the memory (array before pbuf structure).

4.2. AXI

All the preliminary work on the AXI bus, such as AXI-lite and AXI-stream has been made by Jonathan Hendriks. Once they were tested and approved, they have been implemented in the application.

This subchapter covers only the AXI-lite, because the AXI-stream is only present in the PL side of the Zynq.

4.2.1. Overview

The AXI-lite bus is used to communicate with three different components of the PL side, as shown in *Figure 4-14*.

- **TARGETC_IP_Prototype_0**: this IP is used to set the registers of the TARGETC
- **axi_iic_0**: this IP is the I²C used to communicate with the DAC
- **axi_dma_0**: this IP is the DMA used by the AXI-stream communication

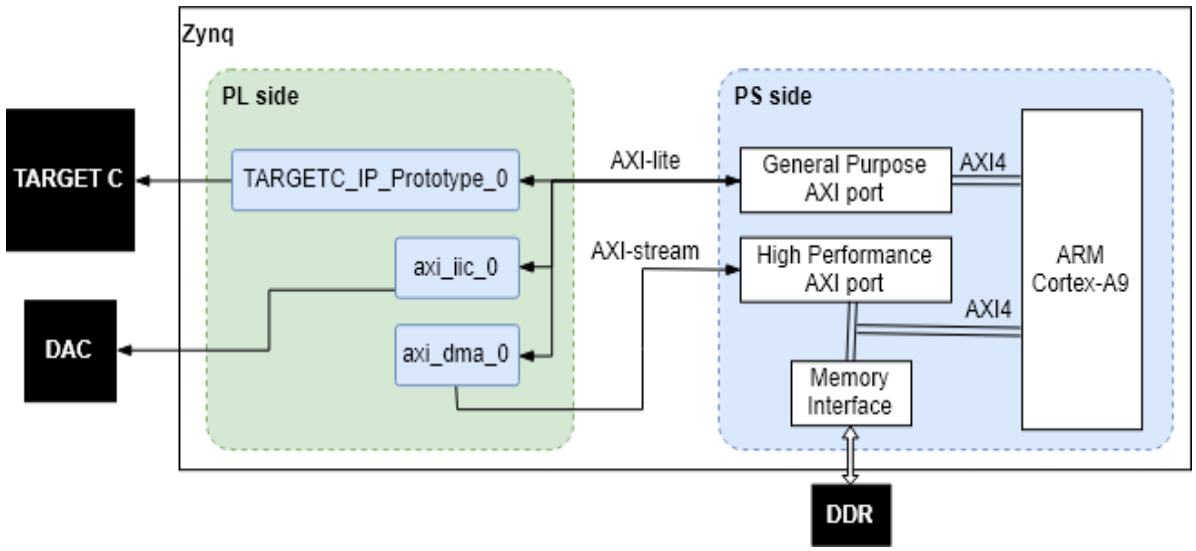


Figure 4-14: AXI overview

From the processor side, all the AXI-lite peripherals are seen as memory location, to access one of them is like accessing an array. Their address is in the file "xparameters.h", which is automatically generated in the board support package after a complete bitstream is generated in Vivado.

4.2.2. TARGETC registers

As specified just above, the TARGETC IP is used to set the registers of the TARGETC, but it has more registers which are used to interact with the PL side (see *Figure 4-15*). To change one of the registers of the ASIC, the function "WriteRegister[1]" is called by passing the register's address and the new value. To write or read to/from other registers, it is as simple as accessing an array.

All the registers are initialized to 0 at the beginning of the application. Then the ASIC's registers are set to their initial value with the function "SetTargetCRegisters[1]". Their value can only be modified when the user sends the command to do so using the UPD communication.

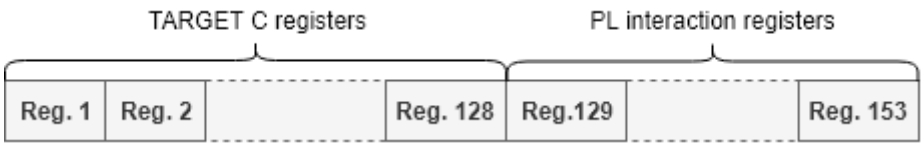


Figure 4-15: TARGETC registers

[1] Developed and explained in the Master thesis of Jonathan Hendriks

4.2.3. DAC

The prototype board (see *Figure 1-6*) is equipped with an 8 channels DAC model LTC2657 with an output voltage going up to 2.5V. This DAC is driven with the I²C protocol. The processor accesses the I²C IP using the AXI-lite communication. Xilinx provides the library "xiic.h" to easily interact with the IP.

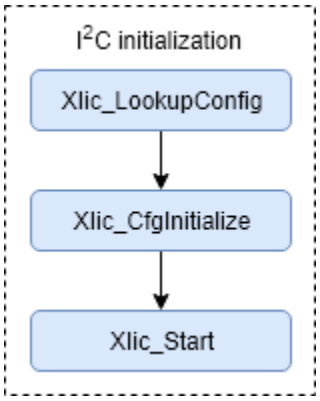


Figure 4-16: I²C initialization

The first step is to recover the I²C IP configuration in the file generated by Vivado.

The function "Xlic_CfgInitialize" creates the I²C instance and initializes it with the configuration retrieved just before.

Then the driver and its interrupt are started, but for this application there is no interrupt like there are no data received.

The I²C address of the DAC depends on how the CA0, CA1 and CA2 pins are connected. At the page 4 of the prototype's schematic (1), these lines are connected to the ground. According to the table 2 at the page 19 of the DAC's datasheet (2), the corresponding address is 0x10. Changing the voltage of one of the DAC's channel requires different steps.

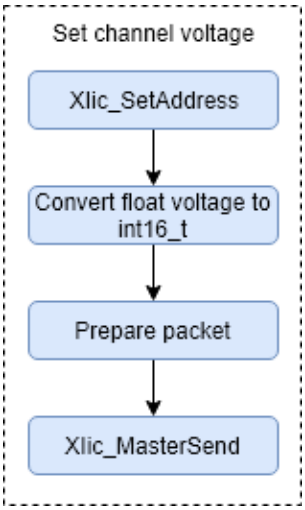


Figure 4-17: Set voltage DAC's channel

First of all, the I²C bus needs to be set up with the function "Xlic_SetAddress" which specifies the address of the device and the direction of the bus (Master to Slave or Slave to Master).

The second step is to convert the voltage into a 16 bits integer, because the corresponding argument of the function "DAC_LTC2657_SetChannelVoltage" is a float value.

Then the frame to be sent is prepared as shown in the *Figure 4-18*.

Finally, the function "Xlic_MasterSend" is called to send the data, but the length of the data given as an argument must be equal to the size of the packet plus one. This last byte is for the slave's address.

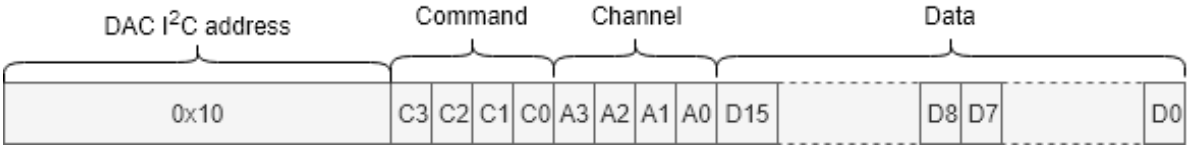


Figure 4-18: I²C packet

According to the table 1 of the DAC's datasheet (2), the command to write to a register and update the output channel of the DAC is 0x3.

As mentioned previously, the DAC has 8 channels, but not all of them are used. The channels A to D are the threshold for the external trigger system and the channel H is the voltage V_{ped}.

4.2.4. DMA

As shown in the *Figure 4-14*, a DMA stores the data from the PL side into the DDR, the protocol at hand is AXI-stream. This DMA is an IP core developed by Xilinx. The processor uses the AXI-lite to manage this unit. As for the I²C, Xilinx provides a library ("xaxidma.h") that facilitate its implementation

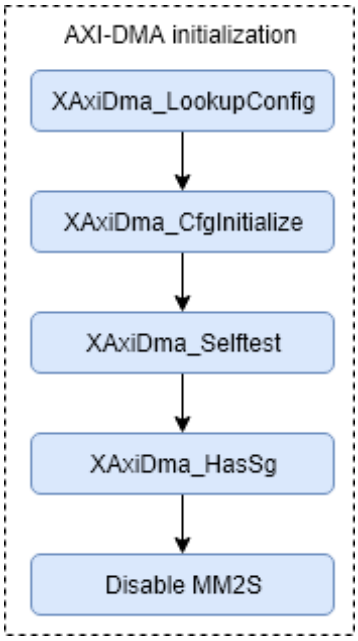


Figure 4-19: AXI-DMA initialization

The two first steps are the same as for the I²C driver. First the configuration of the DMA IP is recovered in the files generated by Vivado.

Then, with this configuration, the DMA instance is created and initialized.

Once the setup is complete, the device tests itself. This test resets the driver and checks if it is coming out of reset or not.

The DMA can be configured in two operation modes, the Simple and the Scatter Gather mode. For this project, the DMA must store the data to a specified address in the memory and for its simplicity the DMA is configured in simple mode. The function "XAxiDma_HasSg" return false if it is the case.

Finally, the DMA is used only to send data from the PL to the memory, this is called S2MM (stream to memory map). The other possible data transfer (MM2S) is therefore unnecessary.

At the end of a complete data transfer, the DMA raises an interrupt. The setup of this interrupt is described in the *chapter 4.4 Interrupts* and the callback function is described later in this chapter.

To initiate a DMA transfer, the processor enables the DMA and gives the device a storage address and the number of bytes that it expects to receive. However, the cache must first be flushed, so the data at the given memory location are up to date. All these steps must be done in the right order as shown in the *Figure 4-20*.

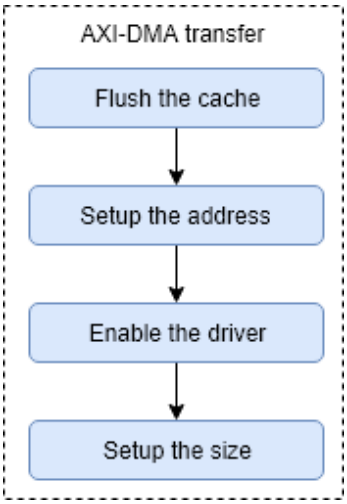


Figure 4-20: AXI-DMA transfer initiation

The processor can select two different modes of operation in the PL side, with one of the interaction registers of the *Figure 4-15*, to receive data through the DMA.

- **Mode trigger**: When a pulse is triggered by the PL side, its waveform is sent to the processor. This mode is switched on when the user sends the command to start the streaming. This mode will be the normal operation mode on the WATCHMAN detector.
- **Mode OnDemand**: The processor can ask several windows starting from a certain window. This mode is used at the beginning of the application to recover the data for generating the pedestals and to calculate the transfer function. This mode is also enabled when the user sends the command to get 20 consecutive windows or the command to recover the data to calculate the transfer function on the computer.

The processor uses a structure to manage the data transferred from the DMA, regardless the mode. This structure is implemented to create a linked list in which every element of this list represents a packet loaded into the DDR by the FPGA through the DMA. In mode OnDemand, the list has only one element and the data are read at every transfer. The structure is composed of three parameters, two pointers which make the link to the previous and next element of the list and one union. The union parameter unites the structure "struct data_axi" and an array of the same size, both located at the same memory address. The advantage of using a union is the capability to access a memory location as a simple array or as an organized structure for processing. The address of this union is the one given to the DMA.

```
typedef struct data_axi_st{
    uint64_t wdo_time;     // Timestamp of the window
    uint64_t PL_spare;     // Spare bits for the development
    uint32_t info;         // Information about the window
    uint32_t wdo_id;       // ID of the window (0 to 511)
    uint32_t data[16][32];// Payload
}data_axi;

typedef union data_axi_union{
    struct data_axi_st data_struct;           // Structure used by the uC
    uint32_t data_array[SIZE_DATA_ARRAY];     // Pointer to pass to the DMA
}data_axi_un;

typedef struct data_list_st data_list;
struct data_list_st{
    data_axi_un data;      // Current element
    data_list* previous;   // Pointer on the previous element
    data_list* next;       // Pointer on the next element
};
```

The info parameter of the data_axi structure is very important and represents the trigger information about the window to which it is link. The processor creates the linked list from the packets transferred by the DMA and uses these bits to manage the list.

- **TRIG bits (bits 0 to 3):** indicate which PMT has triggered an event
- **LAST bits (bits 4 to 7):** indicate if it is the last window of this event's pulse
- **TOO LONG bits (bits 8 to 11):** indicate if this pulse was too long, which means the full waveform must be sent to the computer.

For example, a window has the info parameter equal to 0b0000'0100'0110, this means that this window has data for the PMT 1 and 2 and that the data for the PMT 2 are the last one, so the complete waveform has been received and can be processed. As the "too long" bit corresponding to the PMT 2 is equal to 0, the processor will send only the feature extracted from this waveform.

As mentioned previously, the DMA works with an interruption. When a transfer is finished, a callback function is called. First of all, the status of the interruption is read and the interruption is re-armed by acknowledging it. If the IRQ status indicates an error, the error is reported and the callback ends. If the status is normal, the second step is to set the BUSY bit in one of the interaction registers, so the DMA waits on the processor to engage a new transfer. The next steps depend on the mode in which the application is running.

In mode OnDemand, a flag is set to indicate that a transfer has occurred and the function ends. Then, in the function which has initiated the DMA transfer, the cache is invalidated, in order to load the new value from the memory, after which the data processed and stored in an array. Finally, the DMA is released by resetting the BUSY bit and, if it is needed, another transfer is started.

In mode trigger, the cache is also invalidated. Then, the info parameter of the window received is examined. If none of the last bits are high, the processor jumps directly to allocating and linking a new element to the list as last element. In the opposite case if one of the last bits are high, the received packet indicates that a complete event's waveform was stored in the memory using the linked list. In such case the processor raises a flag corresponding to the PMT to indicate the possible processing of the data. Finally, the BUSY bit is cleared, and the callback function ends.

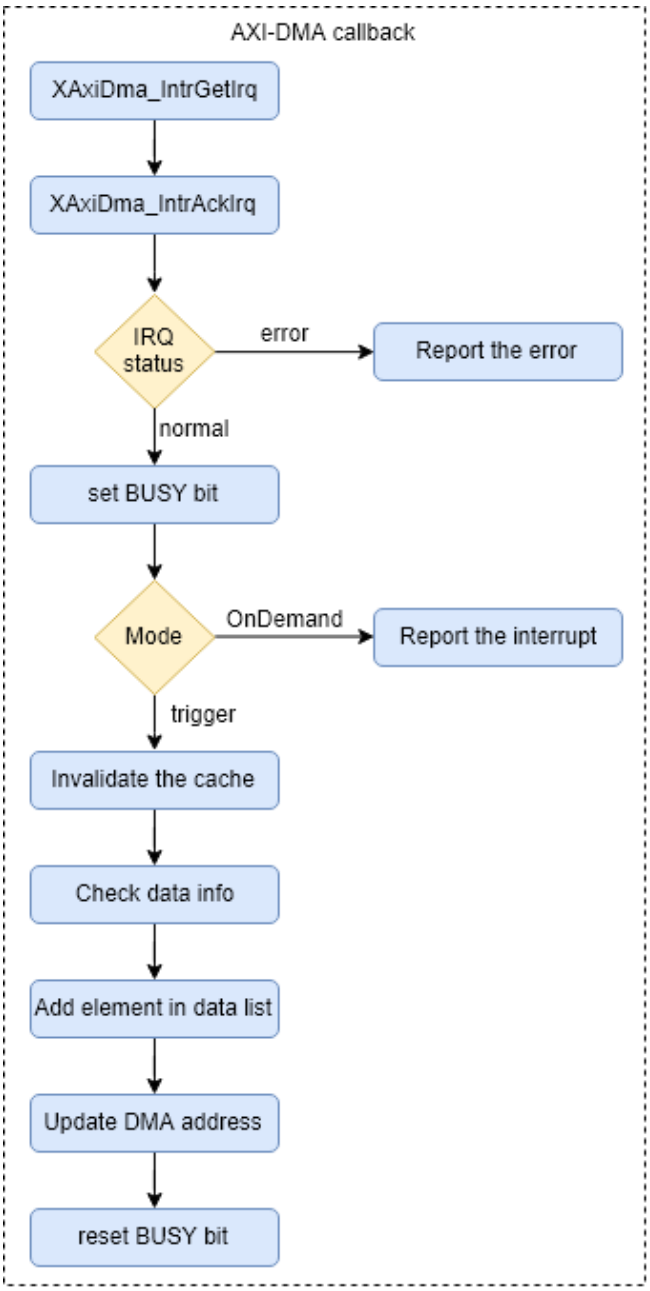


Figure 4-21: AXI-DMA callback

As earlier discussed, the processor manages the packet received by the DMA with a linked list, but only in mode trigger. To refer to this list there are two pointers, one pointing the first element (First) and one pointing the last element (Last). When no packets have been sent or when they have all been processed, the pointers “First” and “Last” point to the same memory location, which is an empty element with its “previous” and “next” parameters set to null. Once the DMA has loaded data in the first element, a new element is created and placed at the end of the list. Its parameter “previous” is pointing to the previous element and its parameter “next” is set to null, because it is the last element in the linked list. The parameter “next” of the first element and the “Last” pointer are now pointing to the new element. This new element has no LAST bit set, which means that the waveform has not been completely transferred. This sequence repeats itself until one of the elements has one of the LAST bit set. When a data is received with one of the LAST bits of the parameter “info” set high, a complete waveform can be processed. In the following example, the third element has its info parameter with the LAST bit corresponding to the PMT 0 set high.

After processing the complete waveform, the list is checked to see if some elements can be removed, but some of them can have multiple TRIG bits set, which means they contain data for several PMTs. Only the elements with no information can be removed. In this example, only the first element has data for one PMT and can be deleted. The parameter “info” of the second and the third element are updated. The TOO LONG, LAST and TRIG bits corresponding to the PMT 0 are released. After which, the second element becomes the first element of the list, its parameter “previous” is set to null, and the pointer “First” is now pointing on it. The list is now composed with three elements, two containing data for the PMT 1, and one empty, waiting on the DMA to load data in it.

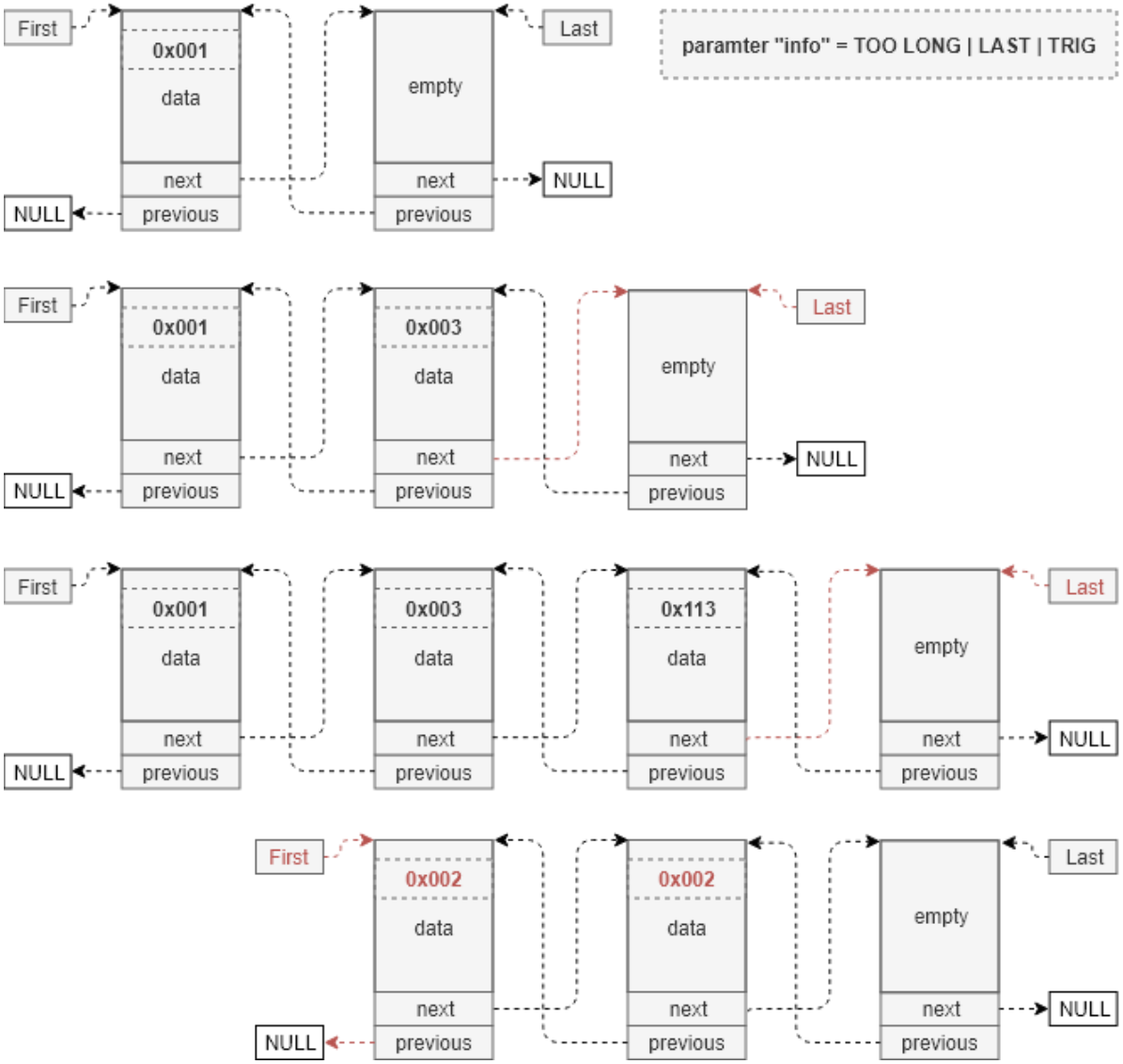


Figure 4-22: AXI-DMA list parsing sequence

4.3. Timers

The Zynq SoC's PS incorporates several timers and watchdogs which can also be used as timers. These on-chip peripherals are either private (internal) to an ARM or a shared resource (external) for both ARMs.

- System Watchdog Timer (SWDT), which can be clocked from the CPU clock or an external source.
- CPU 32-bit watch dog (SCUWDT) clocked at half the CPU frequency
- CPU 32-bit timer (SCUTIMER) clocked at half the CPU frequency
- Two Triple Timer Counters (TTCs). Each TTC contains three independent timers that can be clocked by the CPU clock or by an external source from the Zynq SoC's MIO or EMIO (in the Zynq SoC's PL).
- Shared 64-bit Global Timer (GT) clocked at half the CPU frequency. Each CPU has its own 64-bit comparator, which drives a private interrupt for each CPU

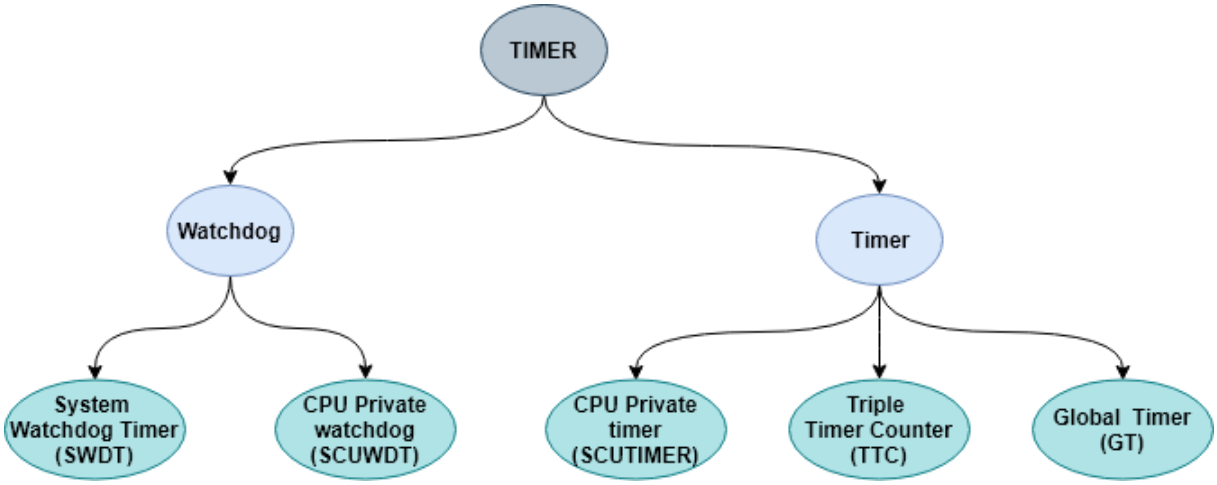


Figure 4-23: Different timers

The private timer does not require any changes in Vivado because it is internal to the CPU, which it is not the case with the Triple Timer Counters which are part of the Application Processor Unit (APU). However, the TTCs provide more flexibility on timing resources. In this project, one of each is used. The Global Timer is used for the log file system (see *chapter 4.5: Log file system*).

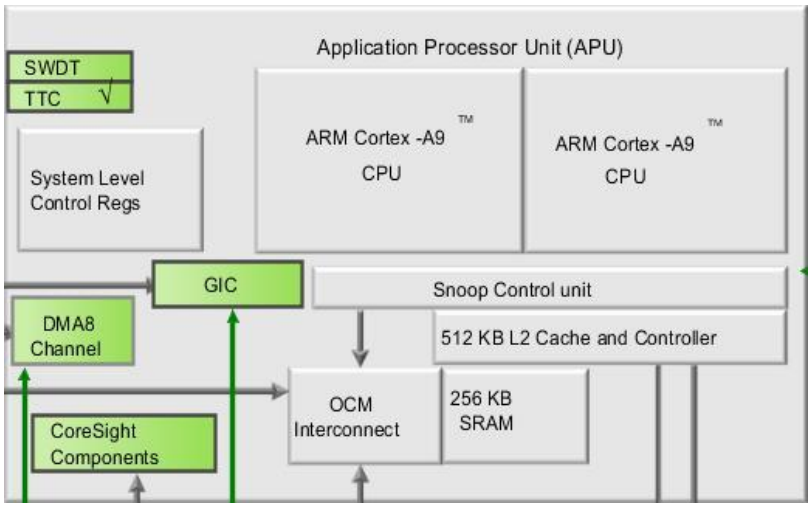


Figure 4-24: APU of the Zynq Soc Block Diagram

4.3.1. Private timer

In this application, three tasks must be repeated every 100ms. Two of these tasks are part of the lwIP implementation. The first one does a software reset on the Rx path, to avoid a long-term bug in case of heavy Rx traffic. The second one moves the Rx packet from the ethernet queue to the lwIP(see *Figure 4-3*). The last task is related to the watchdog which has its counter that must be reloaded to avoid a reboot of the system. The private or SCU timer is implemented with a period of 100ms. As for other devices described earlier in this document, Xilinx provides a library "xcsutimer.h" to implement this timer.

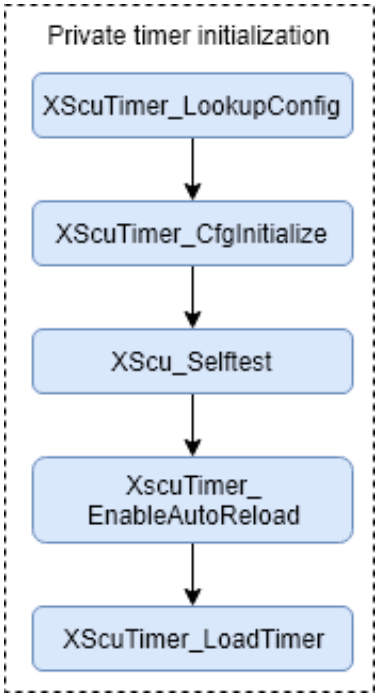


Figure 4-25: XScuTimer initialization

The three first steps of a device's initialization with a Xilinx library are almost always the same:

1. Recover the configuration in the files generated by Vivado.
2. Create and initialize the device's instance with this configuration
3. The device tests itself

The self-test is different from one device to another. For this timer, it loads its counter and then reads it back, and tests if the two values match.

The "autoreload" parameter is a very useful tool in this case. After every interruption, if the flag is reset, the counter will reload itself with the previous value.

The last step is to load the counter with its first value. A special formula is used to calculate the value corresponding to a certain period, because it depends on the processor frequency:

$$value = \frac{ARM_{CLK_{FREQ}}}{2} * t$$ with the time in seconds

When the counter of the timer overflows, the timer raises an interrupt. As for the others devices, the setup of this interrupt is described in the *chapter 4.4 Interrupts*.

The "xemacpsif_resetrx_on_no_rxdata" and "xemacif_input" functions for the UDP communication are directly executed in the timer's callback function. This differs for the function responsible to restart the watchdog. During the initialization of the system, before the "while / infinity" loop, this function is executed in the timer's callback. However, this means that if an assertion or another problem occurs the watchdog would not work. To avoid such a situation some flags are implemented. After the initialization, the callback raises a flag and the watchdog counter is reloaded from the "main".

4.3.2. Triple timer counters

The Triple Timer Counters (TTC) have multiples options (external clock, generate a wave on an output pin, …) and they can raise different interrupts:

- **Interval mode**: when the counter has counted the value contained within the interval register, counting in both ways (up or down), it generates the interval interrupt.
- **Overflow mode**: like a standard timer, the counter counts from 0 to full scale, and generates an overflow interrupts when it finishes.
- **Match mode**: in interval and overflow modes, the timer will generate the match interrupt when the counter equals the value in the match register (if this mode is enabled).

For this application, the TTC 0 is activated in the Zynq Block Design. Its task is to update the timestamp contained in the "time.txt" file every second. Again, this device has a corresponding library "xttcps.h" provided by Xilinx.

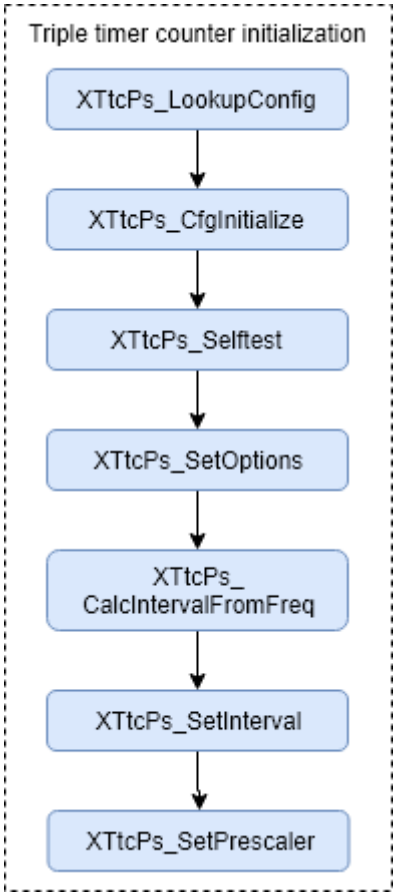


Figure 4-26: TTC initialization

This device's initialization also starts with the standard steps:

1. Recover the configuration in the files generated by Vivado.
2. Create and initialize the device's instance with this configuration
3. The device tests itself

The self-test of this timer works only right after the initialization. It reads the value of the counter control register and tests if the returned value matches the "Reset" value.

The next step is to setup the parameters, the output waveform is disabled and the interval mode is enabled. In this application, the timer must have a periodic interrupt, so the other modes would have worked too.

The interval and prescaler parameters are obtained using a function that automatically calculates and returns their values in function of the desired frequency.

Their registers are then updated with the returned values.

When the counter of the TTC matches the interval value, an interrupt is raised. As for the others devices, the setup of this interrupt is described in the *chapter 4.4 Interrupts*.

During the initialization of the system, the function to update the time file is executed in the timer's callback. However during the "while / infinity" loop, the callback raises a flag and the function is called from the "main".

4.3.3. Watchdogs

For this project, the Private watchdog is used (SCUWDT). The watchdog can work as a standard timer, meaning that when the counter is finished a callback function is called instead of a system reboot. This standard timer mode is very useful during the development, indeed the application can be tested without triggering the system reset if the counter is not reloaded fast enough.

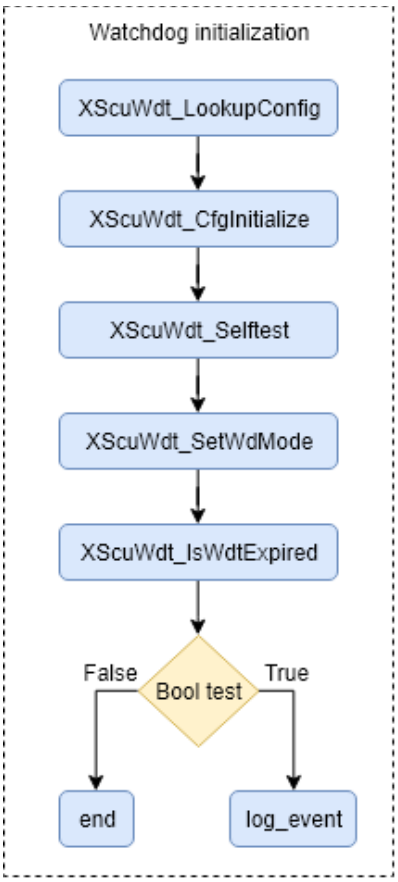


Figure 4-27: Watchdog initialization

This device's initialization also starts with the standard steps:

1. Recover the configuration in the files generated by Vivado.
2. Create and initialize the device's instance with this configuration
3. The device tests itself

The self-test for the watchdog loads a value in its counter, starts the watchdog and verifies if the counter decreases.

As mentioned just above, this device can work in two different modes (timer or watchdog). The function "XScuWdt_SetWdMode" sets it in watchdog mode.

The next function is used to know if the system is booting from a watchdog reset or from a power on.

If it is from a watchdog reset, the event must be written into the log file in order to keep a trace that something in the system went wrong.

If it is booting from a power on, no special steps are needed.

Unlike other errors, when the counter of the watchdog overflows, an interrupt is raised, but there is no callback function which would in normal cases enable the application to log the error into a file. The system directly reboots at the interrupt, immediately after which the watchdog event must be logged. The setup of its interrupt is also described in the *chapter 4.4 Interrupts*.

As explained previously in the *chapter 4.3.1 Private timer*, the watchdog's counter must be reloaded before it overflows to avoid the system reset. In this application, the value loaded in the counter's register corresponds to 2 seconds, so it must be reloaded within this period.

The boot image of the bare metal application must be loaded into the SD card or in the flash. When the watchdog resets the system, the bootloader will search for this file "boot.bin" in order to reprogram the FPGA and the ARM, if the file is inexistent the system will simply not reboot. (To create the boot image file please refer to Digilent's website (3))

4.4. Interrupts

Once all the devices (timers, watchdog, axidma) are initialized, their interrupts can be set up. Only the interrupts of the UDP communication, the assertion and the exceptions are not initialized at the same place. To set up the interrupts, the device which manages them must be set up and then the interrupts of the devices can be linked to it. To do so, Xilinx provides a library “xscugic.h”

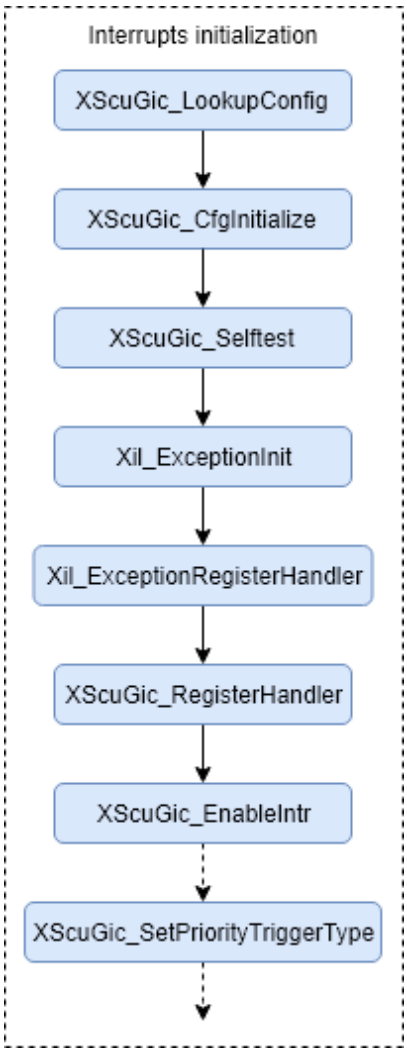


Figure 4-28: Interrupts initialization

The first step is to recover the configuration of the Generic Interrupt Controller (GIC) of the ARM from the files generated by Vivado.

As for the other devices, its instance is created and initialized with its configuration.

Then the device tests itself. To do so, it reads the ID registers of every peripheral and compares them with a “define” value to see if they all match.

The next function is just a common Application Program Interface (API) used to initialize the exceptions across all the supported ARM processors. So, it is present only to take care of backward compatibility issues.

Then the handler of the interrupt controller is connected to the interrupt handling logic of the hardware in the processor.

The next two functions are called for every device.

The first function connects the device driver handler to the interrupt controller. So, when an interrupt occurs, the callback function linked to its ID is called.

The second function is used to enable the device’s interrupt in the GIC.

The last function is also called for every device. It is used to set the priority of the different interrupts. In this case, the highest priority starts with the watchdog followed by the axidma, the timer that updates the timestamp and the other timer.

Once the interrupts are initialized and enabled, they can be activated. After the activation of the interrupts, the two timers must still be started by calling a function. As for the watchdog it must be started and then the counter must be loaded.

Assertions

When there is a bug with one of the devices, an assertion is set if the library “assert.h” is included and implemented. So, when the interrupts are enabled, the assertions are set up using the function “Xil_AssertSetCallback” to attach a callback for this assertion. The callback has two arguments, the name of the file and the line number where the problem occurred. In this system, when the function is called, these two arguments with the timestamp are saved into the log file, and then the program does a software reset which reboots the system.

Exceptions

Not all the software bugs can be detected by the assertions, so the exceptions must be implemented to complete the error detection. The exceptions are set up just after the assertions, using the function “Xil_ExceptionRegisterHandler” which links a callback to a chosen type of exception ID which are listed and explained the table below.

ID	Name	Description
0	Reset	Raised when the processor reset pin is asserted, but can also detect a software reset
1	Undefined instruction	Raised when the processor does not recognize the current executed instruction
2	Software Interrupt	Raised when a user-defined interrupt instruction occurs
3	Prefetch Abort	Raised when the program tries to fetch an instruction from an illegal address. However, the current instruction continues until the invalid one is reached and the exception is generated
4	Data Abort	Raised when an instruction attempts to write or read a data at an illegal address
5	IRQ	Raised when the interrupt request pin of the processor is asserted
6	FIQ	Raised when the fast interrupt request pin of the processor is asserted

Table 4-2: Summary of exception’s ID

For this application, the exceptions are used to detect errors. So, only the ones corresponding to a problem are implemented.

- Undefined instruction
- Prefetch abort
- Data abort

They are all linked to the same callback. The way to identify which one has triggered an exception is by reading the argument containing the exception ID. Once in the callback, the program saves this value in the log file and reboots the system using a software reset.

4.5. Log file system

Given the remote location of the detector, the system must be autonomous. However, a system may encounter new bugs, even if it was fully tested. It is not possible to test every scenario. In order to be able to correct these bugs, so they would not appear again, they must be registered with as much information as possible. The more information there is, the more likely it is possible to understand what caused them.

The best way to register a bug's event, it is to log it into a text file in the SD card. The fact that the file is saved in the SD card has two advantages.

- When there is a power loss, the data stored in this memory is not lost.
- The developer can easily access them (during the development period).

To access the SD card from the processor, its corresponding peripheral must be activated in Vivado. On the MicroZed board, the SD card uses the peripheral SD0 with its signals on MIO 40 to 45, its option CD on MIO 46 and its option WP on MIO 50, and with a bank voltage at 1.8V. Then the SD card can be mounted from the processor. To do so, the Generic Fat File System Library is used and must be added to the project's Board Support Package (BSP), and after the library "ff.h" can be included in the software. The function "f_mount" is used to mount the external memory and has three arguments.

- **fs:** the instance representing the memory
- **path:** path to the memory
- **opt:** option of initialization

For this system, the files are stored in the root directory of the drive 0, so the path is "0:/', and the volume is mounted directly, so the initialization is 1. (see *website* (4) of the bibliography).

An important information that needs to be stored in case of a crash, is the time at which this event happened. As there is no Operating System (OS) running in the ARM and no connection to Internet, the system has no timestamp. So, a solution using the Global Timer Counter has been implemented.

When the system boots from a power on, the timestamp is the 1st of January 2000 at 00h00 0s 0ms which is the default value. A command containing the right timestamp can be send by UDP.

Upon reception of this UDP command, the sent timestamp is saved into the "offset_time" using a time structure (see code below) and the Global Timer Counter is saved as the "offset_counter".

```
// Variable which contain the offset which can be changed with settime_hm
time_cplt offset_time = {
    .year = 2000,
    .month = 1,
    .day = 1,
    .hour = 0,
    .minute = 0,
    .second = 0,
    .milisecond = 0
    };
```

The timestamp can now be asked at any moment. It is equal to the addition of the "offset_time" with the difference of the actual value of the Global Timer Counter and the value "offset_counter".

$$timestamp = offset_{time} + (GlobalTimerCounter - offset_{counter})$$

The timestamp is saved every second in a text file in the SD card. This action allows to retrieve the set up time after system reboot. Then the timestamp just needs a small correction by adding a delay corresponding to the time used to reboot.

So, the log file system is composed of two files, the “time.xt” file which contains the timestamp and is updated every second, and the “log.txt” file which contains all the logged errors and watchdog events.

They are created in the same way, with the function “f_open” using the flags FA_CREATE_ALWAYS, FA_WRITE and FA_READ for the “mode” argument.

- **FA_CREATE_ALWAYS:** Creates a new file, in the case of an existing file, it will overwrite it.
- **FA_WRITE:** Data can be written into the file
- **FA_READ:** File’s data can be read

The log and time files are created only if they are not already saved on the SD card. Like that the events logged previously are not deleted, and the system can also recover the timestamp set up prior to the reset. So, if it is a watchdog reset, the last value in the file corresponds to the time of its occurance.

In case of an existing file, the timestamp of the system is updated with its last value and adjusted by a small delay. Finally, the time is overwritten with the new value.

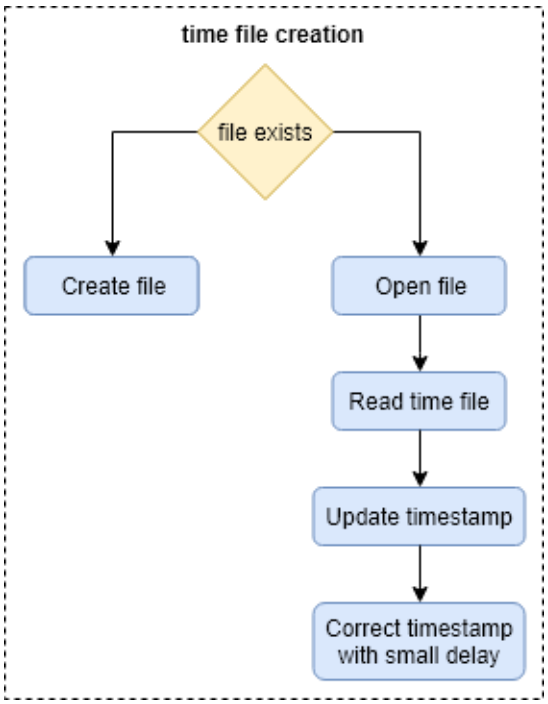


Figure 4-29: creation of time file

As also explained in some of the previous chapter, the log contains more than just the watchdog event. The assertion events are logged passing the function’s name and the line number of where the assertion was called. The exceptions are logged with their ID. Every function called from the main has a return value to indicate the success or failure of the function. If a function returns to main indicating a failure, its name and the returned value are logged. At the exception of the functions which are called before the files creation, like the one to mount the SD card.

Remarks

If the application stops without closing a file, it will then be corrupted and unusable. That is why the files are opened and closed at every writing or reading events.

4.6. Data processing

When the data are received in the processor from the FPGA, they represent the original values read from the TARGETC. However this DAC is not ideal, so they need to be corrected. The first correction to apply is the pedestal subtraction, which compensate the offset issue. The next correction concerns the transfer function, which compensate the comparators issue. Finally, some features are extracted from the waveform, because in 99% of the case the complete waveform is useless.

4.6.1. Pedestals

The TARGETC measures by window of 32 samples on its 16 channels in parallel and it has an analog memory with a capacity of 512 windows. So, this means that the memory in composed of 512*16*32 cells. A cell is in fact a capacitor. Due to the silicium and the construction tolerances, every capacitor is a bit different. The consequence of this difference is that even if they have all been used to measure the same constant DC value, their voltage will be different. A pedestal correction must be applied on the received data, to correct this offset issue. So, the pedestal is equal to the average of a reference voltage measured multiple times. This voltage is named V_{ped}. As all the cells have a different capacitor, they all have their own pedestal value.

The channels 3, 7, 11 and 15 are the ones with a gain stage of 10 and they amplify the noise, that is why they diverge from the others. Also, the channel 3 of the prototype board used for this measure is not working. By arbitrary choice, the window 0 is the one used for the plots.

Before applying the pedestal correction, every sample (red dots) has a negative offset compare to the ideal value (green lines) they should have, as shown on *Figure 4-30*. For this test, V_{ped} is a 1.25V, so the equivalent ADC count value is 1024.

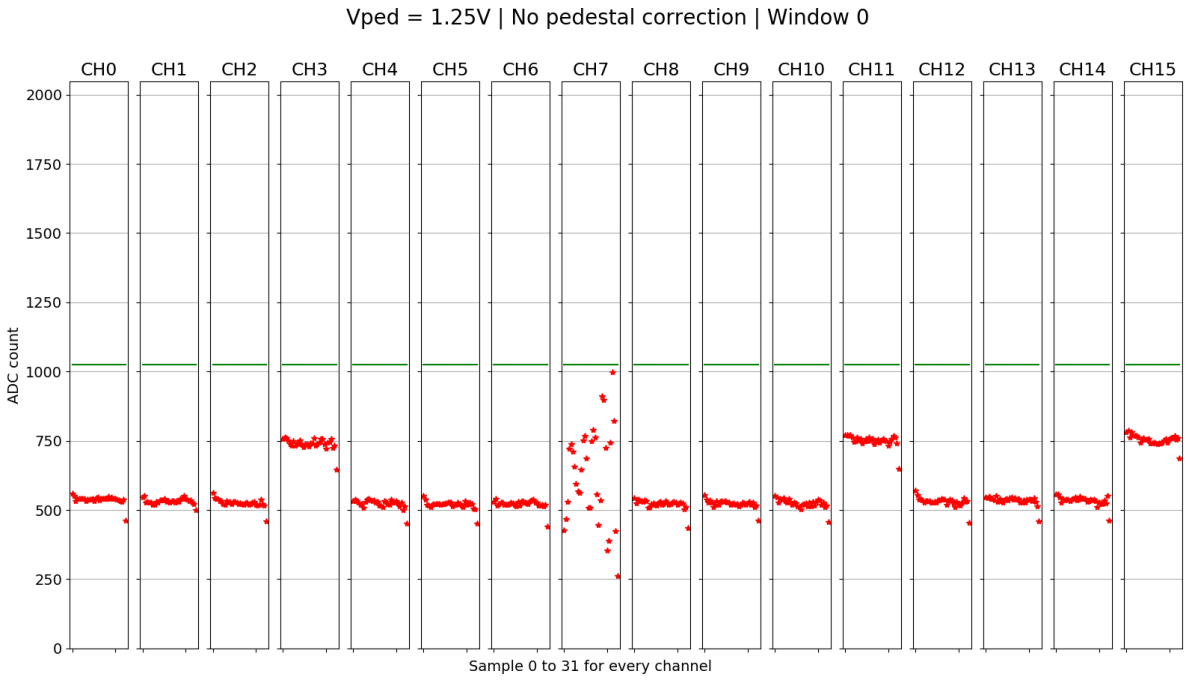


Figure 4-30: Data – V_{ped} = 1.25V | No pedestal correction | Window 0

To generate the pedestals, V_{ped} is applied on all the channels. Then the same window is measured 10 times, while adding the value of the samples at every measure. Once this step is done, the average of every sample is calculated by dividing the result by 10 and stored as the pedestal value. The windows are measured by pair, this means that the read address always changes, to avoid a problem with the capacitor of the ASIC which does not work correctly when a same window is read several times in a row. This is repeated for the 512 windows or 256 pairs of windows.

As shown in the *Figure 4-31*, once the pedestal subtraction is applied, they all have the right value, except the channels with the big gain stage, because of the noise mentioned earlier.

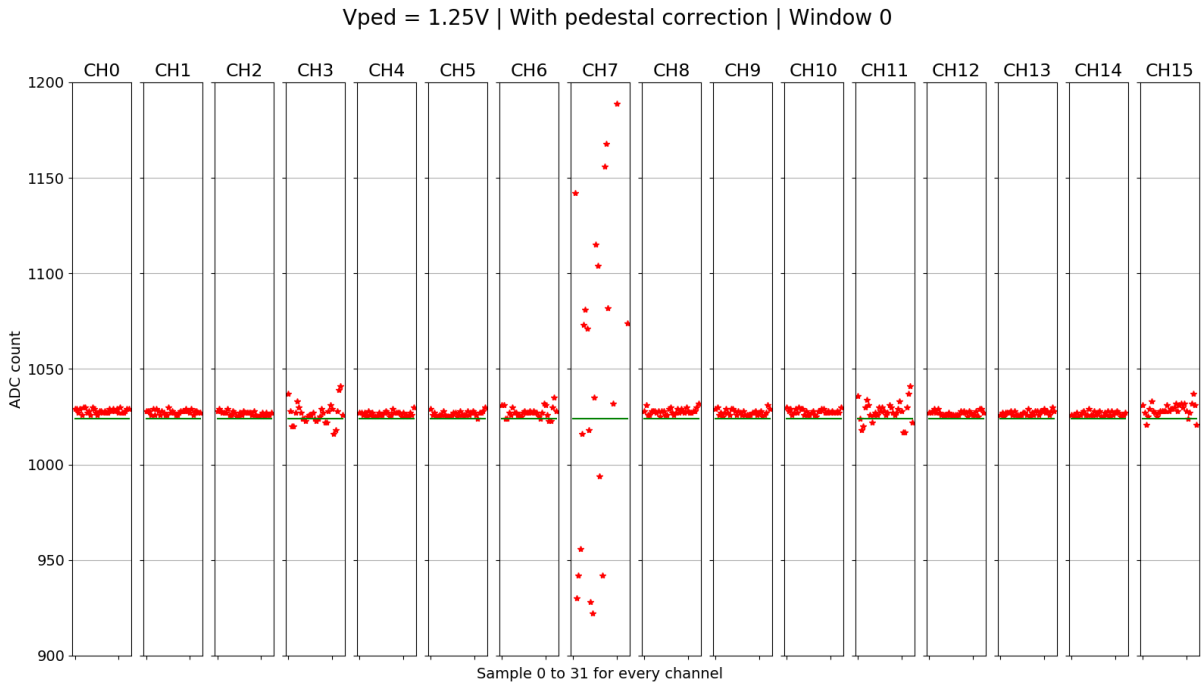


Figure 4-31: Data – V_{ped} = 1.25V | With pedestal correction | Window 0

The *Figure 4-32* represents the behavior of the pedestal subtraction over the range of the input voltage of the TARGETC, which is 0V to 2.5V. The V_{ped} voltage can be set to different values to calculate the pedestals. In this figure, they have been generated with the voltage at 1.25V and at 2V. The graph shows that modifying V_{ped} changes where the voltage measured is the most precise way, compared to the ideal value.

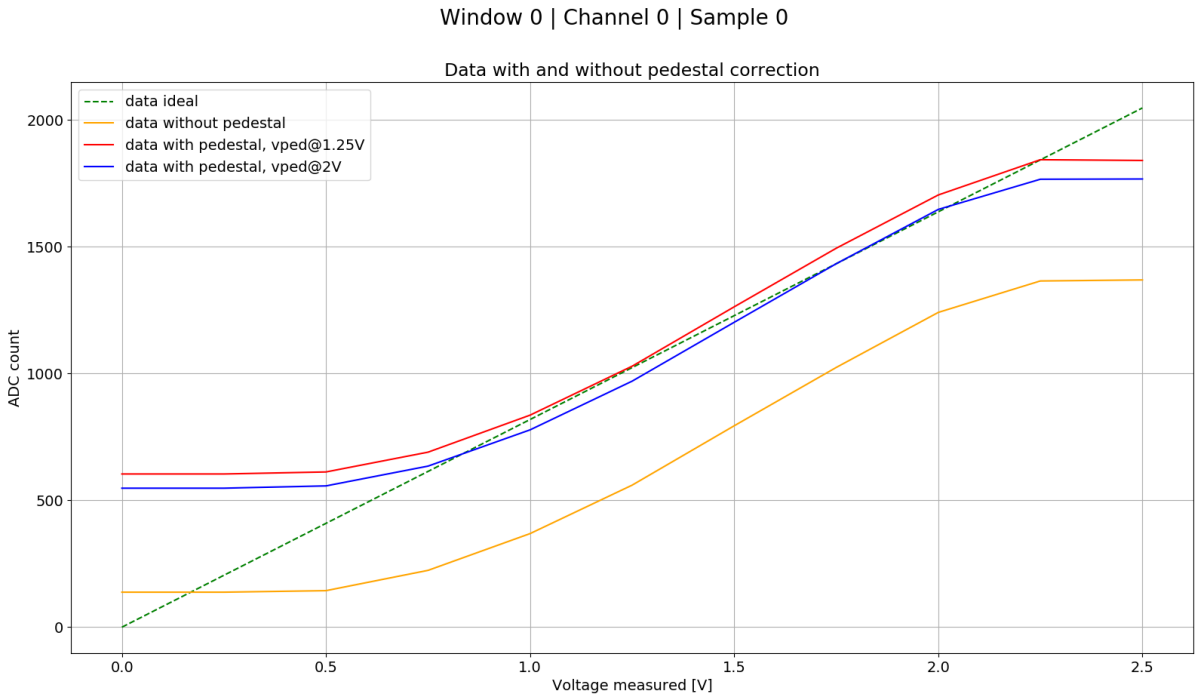


Figure 4-32: Data with and without pedestal

4.6.2. Transfer function

The TARGETC is a Wilkinson ADC. To digitize the voltage contained in the cell capacitor, it compares this voltage with the one produced by charging another capacitor. The voltage on the second capacitor is linearly increased while a counter is also linearly incremented. Once the voltage of these two capacitors are the same, the comparator's output latches the value contained into the counter in a register, which represents the digital response for the voltage stored in the cell. The comparator used is also not ideal and the conversion is not completely linear. This effect can be noticed in the *Figure 4-32*. As it is almost the same result for V_{ped} at 1.25V or 2V, the next tests are done only with one voltage (1.25V). The behavior of the transfer function is almost the same for every cell, so the next tests are done only for the sample 0 of the channel 0 of the window 0. The C_{meas} values used for the plot are generated by measuring 11 input voltages between 0V and 2.5V.

Once the pedestal subtraction is complet, the transfer function correction must be applied to counter react the non-linearity problem of the comparators. There are two approaches to correct the transfer function.

The first one is by using the difference between the ideal values and the measured values. Then a polynomial fitting of 3rd order is done on the difference curve, in order to have a polynomial expression of it. Finally, the corrected values can be found by subtracting the "polynomial" values, generated with the polynomial expression, from the measured values.

$$C_{diff}(V_{in}) = C_{meas}(V_{in}) - C_{ideal}(V_{in})$$
$$f_{poly}(V_{in}) = polyfit\big(C_{diff}(V_{in})\big)$$
$$C_{cor}(V_{in}) = C_{meas}(V_{in}) - f_{poly}(V_{in})$$

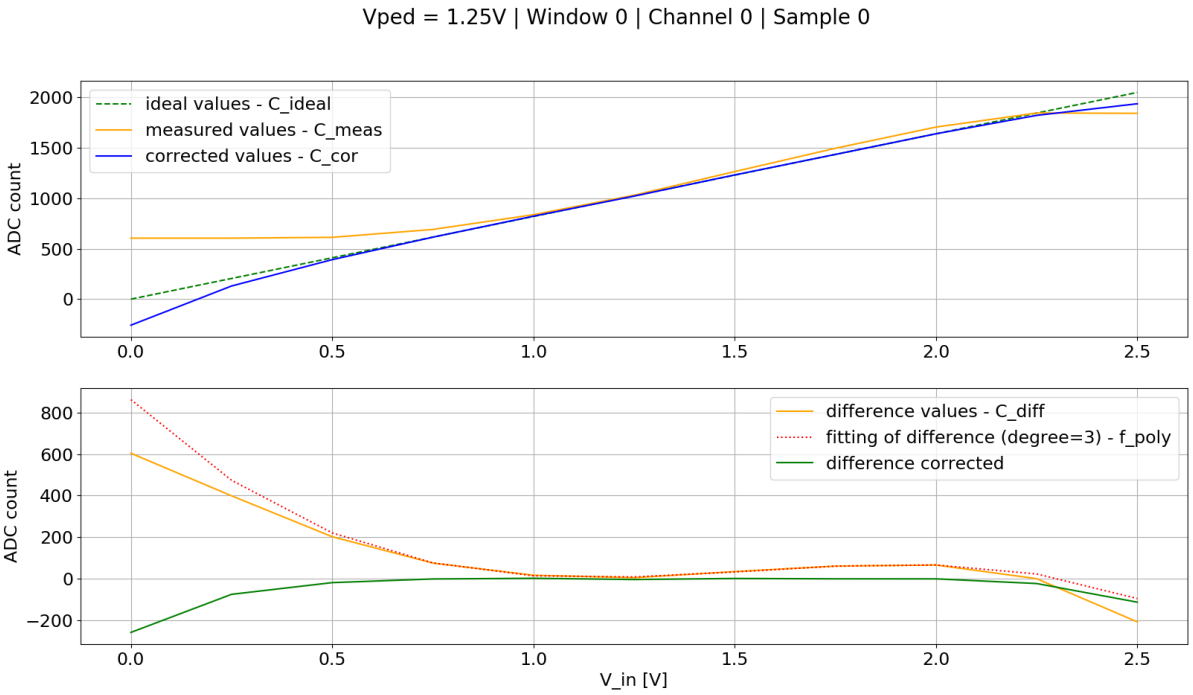


Figure 4-33: Correct transfer function (first approach)

The second approach is by doing a polynomial fitting of 3rd order directly on the measured values. The expression describing the ideal values is a simple linear function (y = a*x + b), in which case the offset coefficient (b) is 0 and the slope coefficient is equal to the maximum ADC count divided by the maximum input voltage V_{in}. The independent variable of this linear function can be replaced by the inverse function of the polynomial expression. The result of this operation is the expression to find the corrected values.

$$f_{poly}(V_{in}) = polyfit(C_{meas}(V_{in})) \rightarrow C_{poly} = f_{poly}(V_{in}) \rightarrow V_{poly} = {f_{poly}}^{-1}(C_{meas})$$

$$C_{ideal}(V_{in}) = \frac{2047}{2.5} * V_{in}$$

$$C_{cor}(V_{poly}) = \frac{2047}{2.5} * V_{poly} \rightarrow C_{cor}(C_{meas}) = \frac{2047}{2.5} * {f_{poly}}^{-1}(C_{meas})$$

The problem of the second approach is finding the inverse function of the polynomial expression of 3rd order. One solution is to inverse the independent and dependent variables. So, now the input voltage V_{in} depends on the ADC count value.

$$f_{poly}(C_{meas}) = polyfit(V_{in}(C_{meas})) \rightarrow V_{poly} = f_{poly}(C_{meas})$$

$$V_{in}(C_{ideal}) = \frac{2.5}{2047} * C_{ideal} \rightarrow C_{ideal}(V_{in}) = \frac{2047}{2.5} * V_{in}$$

$$C_{cor}(V_{poly}) = \frac{2047}{2.5} * V_{poly} \rightarrow C_{cor}(C_{meas}) = \frac{2047}{2.5} * f_{poly}(C_{meas})$$

With this solution, the final expression is implementable because the polynomial fitting can be done in C.

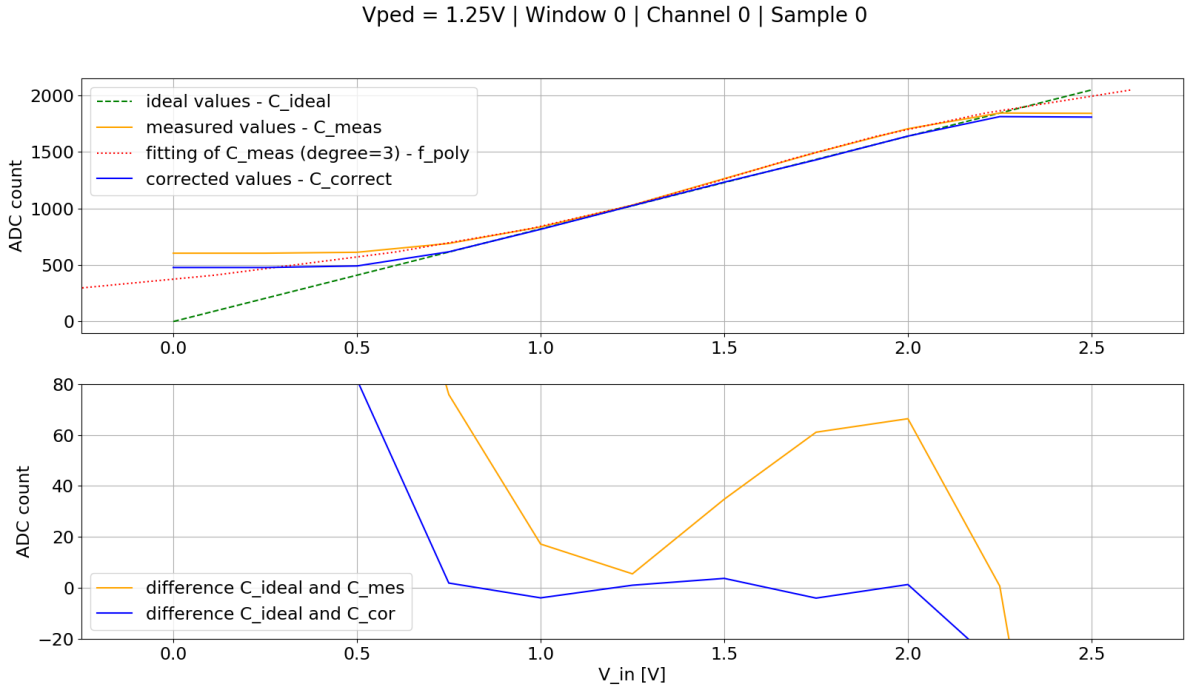


Figure 4-34: Correct transfer function (second approach)

The top graphic is plotted with the dependent variable on the horizontal axis, and the independent variable on the vertical axis, to simplify the understanding.

To improve the correction, the polynomial fitting is done only on the values included in the range 0.75V to 2V, where the transfer function is the most linear.

The *Figure 4-34* shows that the second approach is correct and working. The difference for the corrected values is much smaller than the difference for the measured values. Even if the difference is not good for a voltage smaller than 0.75V or bigger than 2V, it is not a problem, because the ADC is not going to be used in its saturation zones.

As a reminder, the ASIC contains 512 windows which have 32 storage cells for each of the 16 channels. Every cell has its own comparator which corresponds to a total of 512 * 32 * 16 = 262'144 comparators. A polynomial expression of the 3rd order is defined by 4 coefficients, and each of them are saved in memory as a double (8 bytes). The primary idea was to compute the polynomial function for every comparator. In consequence, the storage amount would have been enormous (8MB), and the processing time too long. As a result, the behavior of each comparator has to be studied and compared between each other in order to see if there is any convergence.

An overall average on the transfer function serves to compare them by plotting the histogram of the difference for each voltage value.

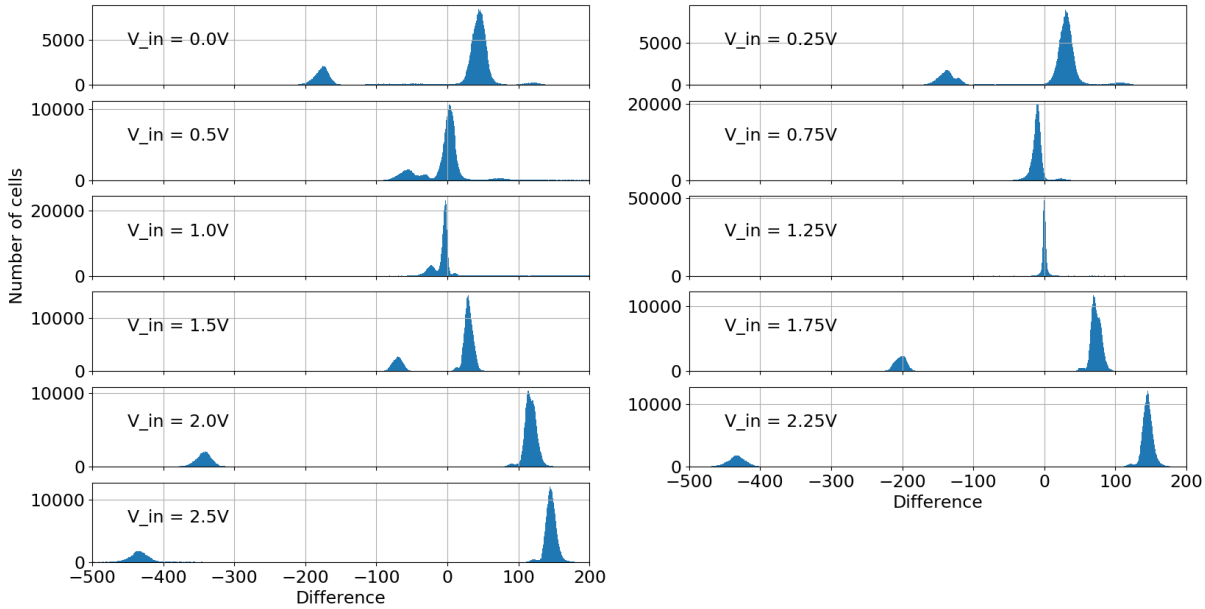


Figure 4-35: Histogram of the difference between average and every cell

As it can be noticed in the *Figure 4-35*, the repartition is not centralized on 0. Most of the samples have a positive difference and a small amount of them have a negative difference. In order to understand which memory location influences the negative differences, a further analysis of its composition is done.

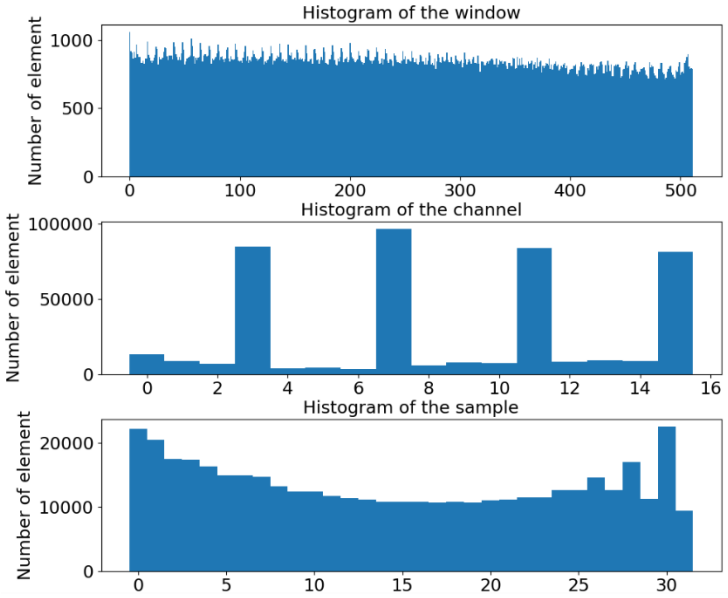


Figure 4-36: Histogram of small amount with a negative difference

As shown in the *Figure 4-36*, the channels that have a transfer function response smaller than the average are the ones with a gain of 10, there are the same ones that caused problems for the pedestal (see *Figure 4-30*). The average is recalculated ignoring those channels, and the histogram of the differences is replotted.

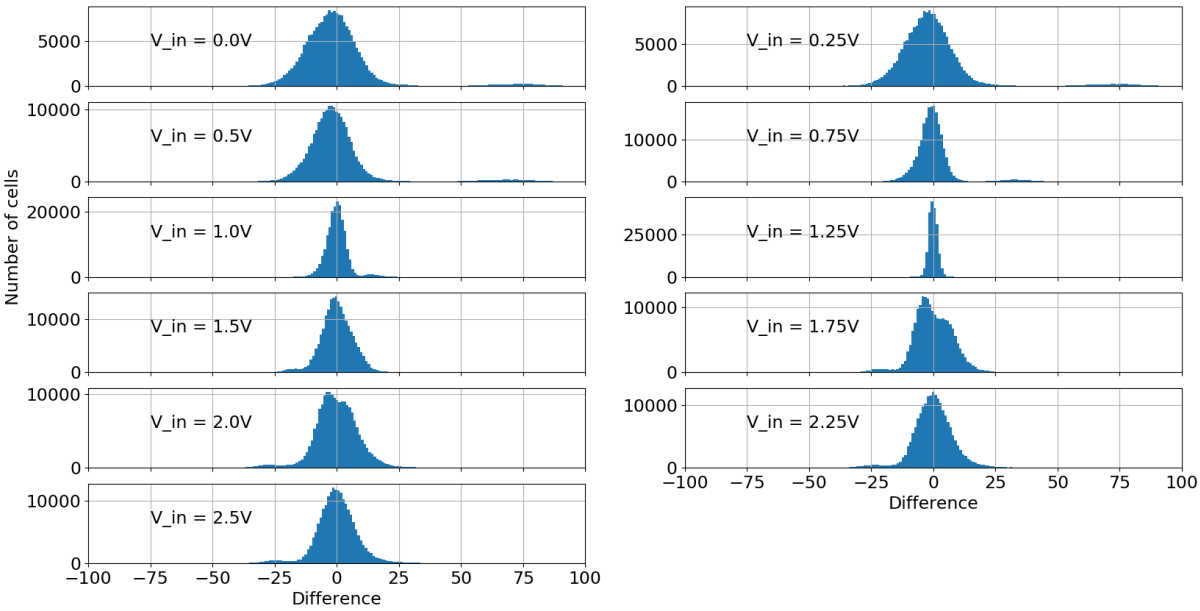


Figure 4-37: Histogram of the difference between average and every cell, except channel 3, 7, 11 and 15

The histograms of the *Figure 4-37* confirms that the behavior of each comparators is nearly the same. Consequently, the polynomial fitting done on the average can be applied to every memory location. It is not a perfect solution, but for the current status of the project it is more than acceptable.

The transfer function correction cannot take too long, because it has to be applied on every sample, in addition to the pedestal correction, before sending it to the computer. Therefore the solution is to generate the corrected value for all the possible ADC count values from 0 to 2047 and to store them in a lookup table.

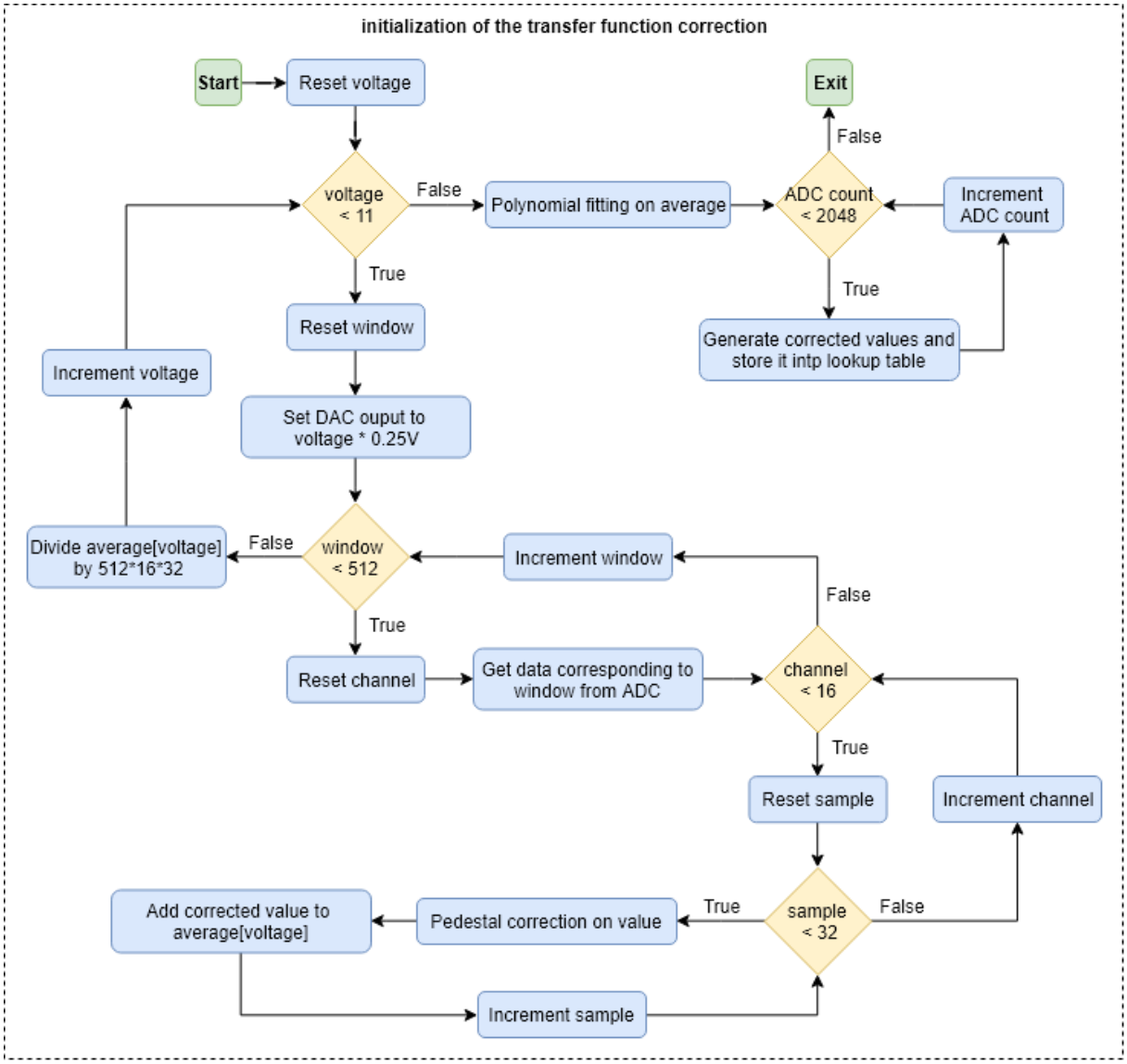


Figure 4-38: Initialization of the transfer function correction

Once the lookup table containing the generated values for the transfer function correction is initialized, it can be used in the following way. After a sample has been pedestal subtracted, the result is used as an index in the lookup table. The value at this address is the corrected value to send to the computer.

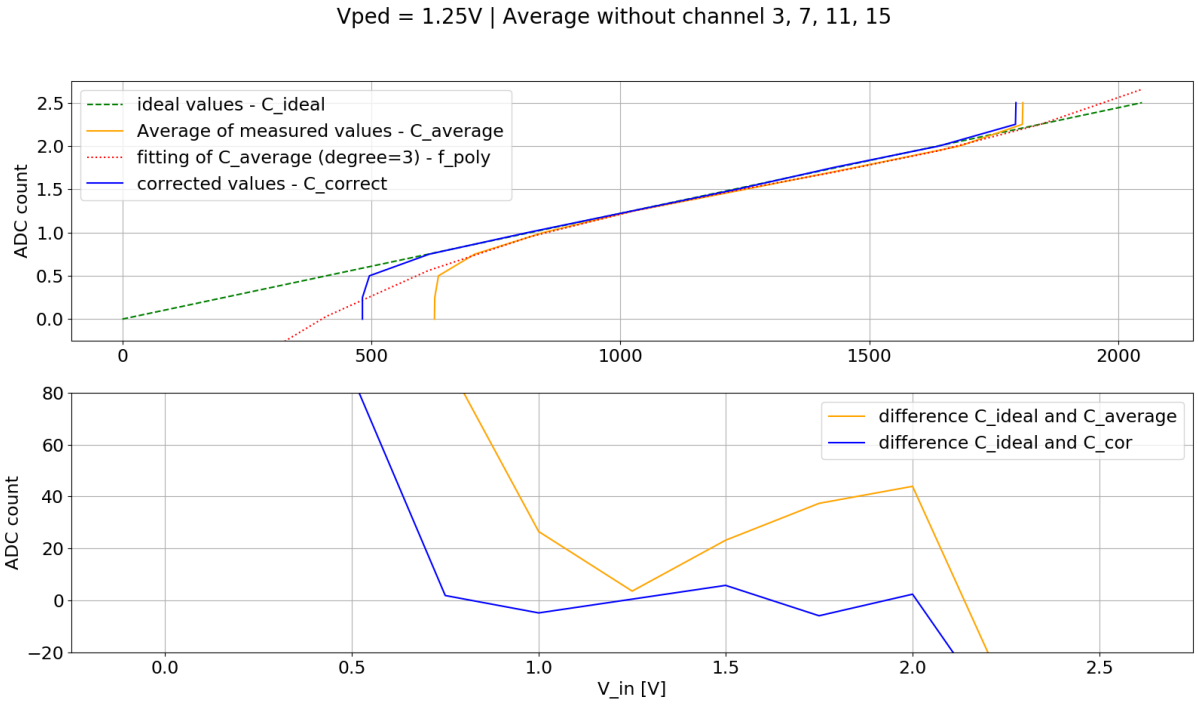


Figure 4-39: Corrected data from Zynq

The graphic of the *Figure 4-39* shows data coming from the Zynq. They are fully corrected using the pedestal subtraction and the lookup table correcting the transfer function error. Their average values plotted in this graphic and the ones used to generate the polynomial coefficients do not take into account the channels 3, 7, 11 and 15, because they are not working correctly and they alter the result.

4.6.3. Data formatting

When the Zynq has received a complete raw waveform through the DMA, it knows only which PMT has measured something. The first step is to choose the most appropriate gain stage. To do so, the processor starts with the highest gain channel (3, 7, 11, 15) and corrects this channel's raw data by applying the pedestal subtraction and the transfer function correction. After which each sample of the corrected data is compared to the predefined threshold to see if the gain is too high. If this is not the case, the processor keeps these data and goes to the next step. On the other hand, the next channel is selected and the same operations are performed. If none of the channels are working, the triggered pulse is marked as "too long" and the program continues with the data of the last one.

After channel selection and data correction, the processor builds the UDP frame based on whether the pulse is normal or too long. For a "too long" pulse, which represents less than 1% of the measures, the Zynq sends the complete waveform to the computer. For a normal pulse, some features extraction is performed on the waveform. The first information to extract is the maximum amplitude of the pulse which is in fact the smallest value of the waveform, because the DC offset is set to V_{ped} and the pulses are negative.

To find the maximum amplitude, the program parses all the data searching for the maximum, it then saves the position and the amplitude of the sample. The second information is the time where the signal reaches 20% of this amplitude, which means 20% of difference between V_{ped} and the maximum. Most of the time, this value is between two samples. The processor searches for the two samples surrounding this point, starting with the maximum's sample and going backwards. Once these two points are found, the time is extracted from their interpolation. The two information are then transmitted to the computer.

4.7. Application overview

At the beginning of the application, the initialization function of every module is called in the following order:

1. Global variables
2. SD card mounting
3. File creation
4. Devices such as the timers, the AXI-DMA and the watchdog
5. Interrupts are setup
6. lwIP stack and its modules
7. Interrupts are enabled
8. DAC with the corresponding value on each channel
9. UDP communication with the two lines
10. Registers of the TARGETC with the default values
11. Test of the TARGETC with its test pattern generator
12. Pedestal subtraction values
13. Lookup table of the transfer function correction

Once the system initialization is done, the program enters a “while / infinity” loop and then everything works on interrupt. None of the tasks are executed in the callback, but their functions are called in the main using a flag system (see *Figure 4-40*). This means that when an interrupt occurs, a specific flag is set and the callback resumes. The corresponding function is called from the main, when its flag is triggered. The outcome of using this method, is to avoid all the problems linked from staying too long in an interrupt’s callback. All the above is true, except the UDP commands “write all the registers”, “read all the registers” and “set time” which are treated directly in the callback, because the execution time of these commands is low.

The flag system mentioned earlier is composed of:

Run: set when the program enters the “while / infinity” loop, reset when the command “stop uC” is received and used as the condition for the while loop in the main

TTC: set when TTC interrupt occurs, which means every 100ms, and reset after reloading the watchdog’s counter

SCU timer: set when the SCU timer interrupt occurs, which means every second, and reset after updating the time file

Stream: set when the command “start streaming” is received, reset when the command “stop streaming” is received and used to control the data streaming, which means setting the DMA in mode trigger

Get transfer fct: set when the command “get data from transfer function” is received and reset after sending the data from the transfer function

Get 20 windows: set when the command “get 20 windows” is received and reset after sending 20 windows

Rx axidma: It is an array of 4 flags, one per PMT. They are set when an AXI-DMA transfer is finished in mode “trigger”, but only if the element received has the “LAST” bit corresponding to the same PMT at ‘1’. They are reset after processing the data of this PMT.

Axidma rx done: set when an AXI-DMA transfer is finished in mode “OnDemand” and reset when the received packet has been processed

Axidma error: set when an AXI-DMA transfer encountered an error and reset when the error is handled

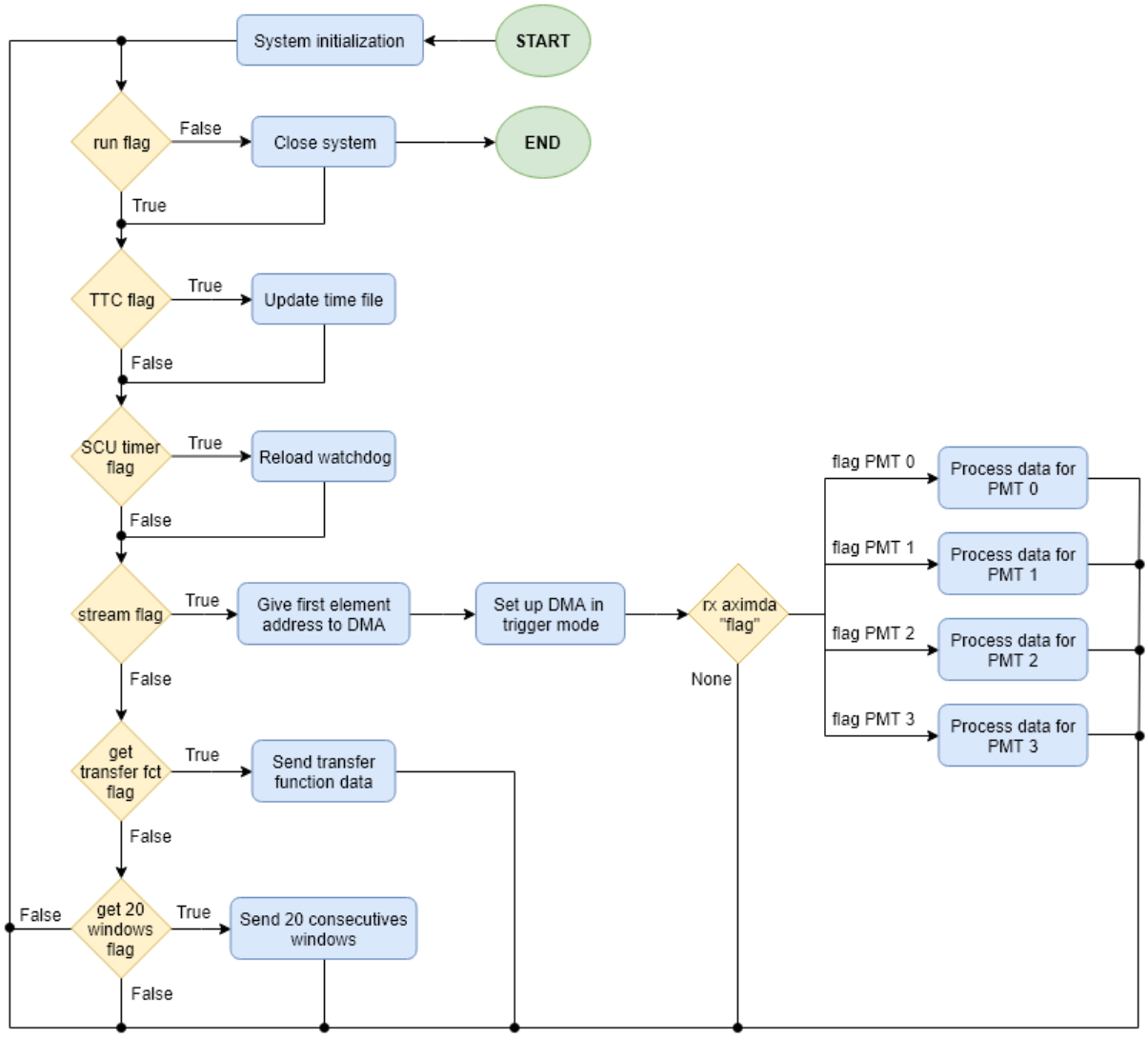


Figure 4-40: Application overview

5. Python GUI

The python General User Interface (GUI) is used to interface with the Zynq. It is always more pleasant, easier and faster to have a graphical interface than to have to run different scripts by writing commands in a terminal. The application is composed of two classes which correspond to two windows, the main one opened at the beginning and the graphic one is used when the system is streaming.

5.1. Class Watchman_main_window

The first window allows the user to interact with the Zynq by sending different commands. All the commands are represented by a button. When a frame “command” is sent or received, a feedback is displayed in the listbox with the direction of transmission, the command ID and the frame ID. On the left of the buttons are the entries containing the registers values read or to be sent. If the text is red, this means that this is an invalid value. There is also a menu to save or load the registers config into a text file. The button at the bottom of the controls open the graphic window used to view the streamed data.

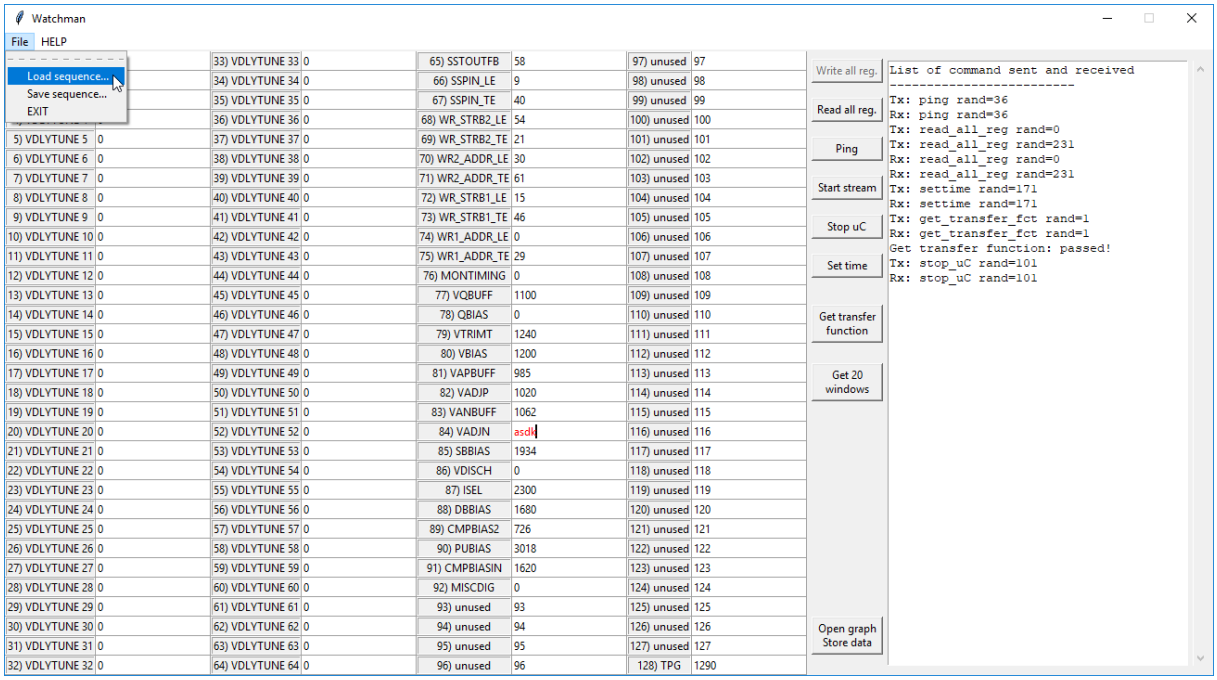


Figure 5-1: GUI – Main window

5.1.1. Methods

The argument “self” present in all methods refers to the object of the class itself (Watchman_main_window).

__init__(self, master)

This is the constructor called by Python when a new instance of this class is created. The argument “master” is an object window created before the object “Watchman_main_window”. It begins by initializing all global variables and the window itself. Then the UDP communication for the commands is also initiated. Finally, a thread looking for the command packet is created and started.

init_window(self)

This method shapes the window. It creates and places the graphical objects, known as Widgets, from the library Tkinter such as the menu, the labels, the entries, the buttons, the listbox, and the scrollbar. It also links these objects to their callback, if they have one.

savefile(self)

When the user wants to save the registers configuration, he clicks on the menu’s command “Save sequence…”. This method is the callback of this event. It opens a filedialog window which is also a Widget from the library Tkinter, so that the user can choose the name and the location of the text file. Once these information are entered, the method continues by creating the text file and saving the registers values.

loadfile(self)

This method is the callback of the menu’s command “Load sequence..”. The same operations executed by the “savefile” are called, except this time the file is opened and the registers configuration are loaded.

exit_prog(self)

Before ending, the application must first perform the following tasks:

- If the graphic window is open, order it to end and wait until it finishes
- If the system is streaming, send the command “stop streaming” and wait on the confirmation
- Wait on all the potential threads to finish

Once that the above tasks are finished, this method closes the UDP communications and destroys the window object.

is_number(self, s)

This method used in the entry’s callback analyses the string argument “s” to test if it represents a number and return the Boolean result.

entry_callback(self)

For every character written in the entries object, this method is called. It tests if the entry’s content is a valid value for a register, this means a positive number which can be represented on 12 bits. The text color is black if the value is valid, otherwise it is red. The button “write all registers” is available when all text entries are valid and not left blank.

write_txt(self, t)

The content of an object listbox can be modified by the application and the user when it is activated. In this case, the listbox is used as visual terminal, the user should not be able to modify it. Therefore when a new message needs to be written, the listbox is enabled only for the period of time to write the text contained in the argument “t”.

init_UDP_cmd(self)

Using the library socket, this method initializes the UDP communication for command line on port 7.

init_UDP_data(self)

Same as for the previously explained method, except this time the initialization is for the data line on port 8. Another difference is that the size of the reception buffer for this communication is expanded. So, the application does not miss any data frame which are considerably larger and more numerous than the command frame.

close_UDP_data(self)

The UDP communication on port 8 is also used in the class "Watchman_graphic_window". Before opening the graphic window, the data line must be closed, which is the purpose of this method.

send_command(self, cmd)

Any interaction with one of the command buttons will result in calling this method. Each button has its own value for the argument "cmd". The frame to be sent is built as shown by the *Figure 4-5*. The payload for the command "set time" is filled with the information returned by the function "localtime" part of the library time. As mentioned in its name it considers the time zone in which the GUI is launched. When this method is called for the commands "get transfer function" or "get 20 windows", it can be called to perform these additional tasks.

- If the second window is open, the command is not sent. Because the system could already be using the data line to send pulses information.
- If the previous test is negative, then the following tasks can be executed
    - The buttons to open the graphic window, to get 20 windows, to get the transfer function and to start the streaming are disabled
    - Initialize the UDP communication
    - Create and start a timer and the thread which will manage the data received

Finally, the packet is sent to the system and notified in the listbox.

open_graph(self)

If it is not already the case, this method creates the second window.

close_graph(self)

By closing the main window, this method is called to close the graph window if it was previously opened.

thread_data_int(self)

When sending the command "get transfer function" or "get 20 windows", a thread is created which executes this method. Upon arrival of a packet on port 8, it controls the data's integrity and stores them in the appropriate file according to the command ID. The thread returns after receiving the correct amount of data. In case of data loss, the thread is stuck in a loop until the corresponding timer raises a flag to release it. The disabled buttons are reactivated.

thread_timer_int(self)

This method is called by the timer to set the flag mentioned above.

thread_cmd_int(self)

At the end of the initialization, a thread runs this method to continuously overwatch the command line on port 7. The same as for the other thread it must check the integrity of the packet. Once the data is validated, it displays the command ID in the listbox. If this ID corresponds to "read all register", it will fill the entries with the received values. Finally, it closes the UDP communication for the data.

5.1.2. Running overview

In a brief description, when the application is launched, the window and the ethernet communication are initialized. Following the start up only one thread is running. It continuously watches the command line for any kind of packet. The state of the program remains like this until the user interacts with any of the graphical objects in the window. In which case, the event will be treated, and the appropriate tasks are executed.

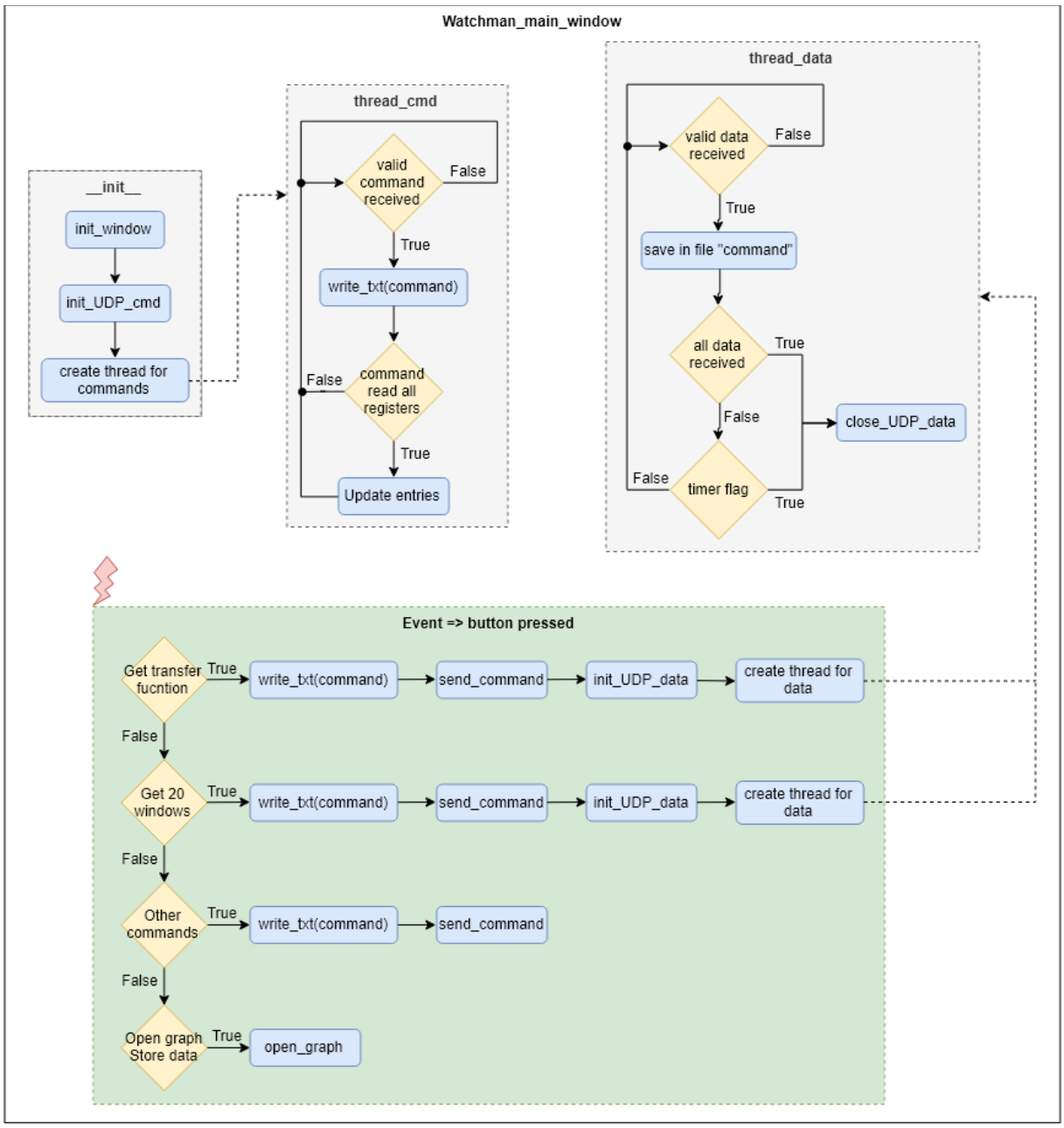


Figure 5-2: Overview of main window

5.2. Class Watchman_graphic_window

The second window is useful only when the Zynq is streaming data. These data are saved into a binary file. They are also represented by three different histograms. The “hitmap” displays the number of hits per channel. The two others on the left of the window are specific to a channel which can be selected with the scrollbar. The one at the top concerns the amplitude of the pulses. The column with the range corresponding to this amplitude is incremented. The last one is about the time of the pulses. If the packet contains a full waveform, it only increments the “too long” column. In all other cases, only the first four columns matter, and the one with the corresponding range is incremented. Finally, the binary file is closed when the user exits the window.

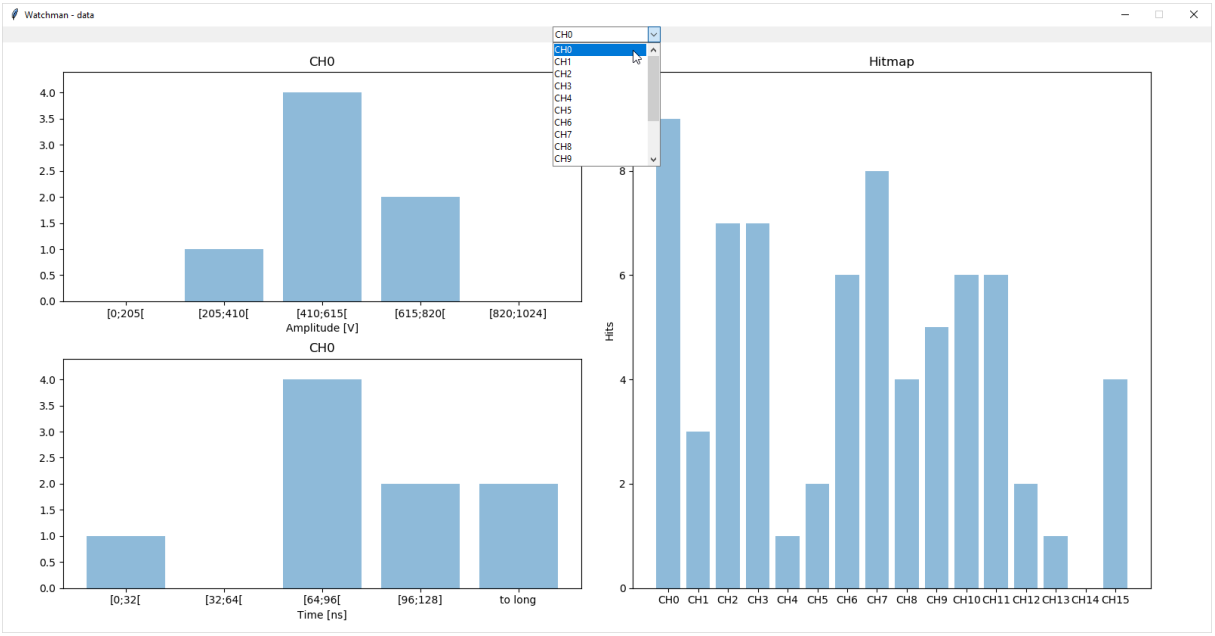


Figure 5-3: GUI – Graphic window

5.2.1. Methods

The argument “self” present in all methods refers to the object of the class itself (Watchman_graphic_window).

__init__(self, master)

This is the constructor of the class described in this chapter. The argument “master” is the window created before using the object of this class. This method creates and initialize the global variables, initialize the UDP communication and initiates the thread looking for the data packet.

init_window(self)

This method builds the graphic part of the window by creating the three histograms. A “combobox” object from the library Tkinter is also created. It is used to select the channel’s data displayed in the two left plots.

init_UDP_connection(self)

Like for the class “Watchman_main_window” this method initializes the UDP communication using the library socket, but on port 8. The socket’s buffer size is doubled, in order to catch all the packets when the Rx traffic is overwhelming.

recv_data(self)

The thread created and started at the end of the constructor executes this method. It continuously listens to the data line, looking for the data frames (see *Figure 4-8* and *Figure 4-9*). Unlike the threads of the main window, this one does not process the data. Upon reception of a packet, it stores it in a list and increases a counter. When the counter reaches 100, it creates another thread and gives it the list of packets. All created threads are added to the running threads list. However if the packet counter is not null and that no new frames were received for a period of 0.2 seconds, a timeout is reached and a new data processing thread is created.

data_processing_int(self, *args)

The data processing threads call this method. They take the first packet in the list and check its integrity. Then they modify the variables plotted in the histograms, according to the packet's content. They repeat this operation for each element of the list. When they access the plotted variables, they first acquire the lock to avoid any conflicts with any other processes accessing them and release it when they have finish with the element.

plot_int(self)

This method is called every second. It updates the histograms with the new values contained in the plotted variables. Like for the data processing thread, it must acquire the lock before using these variables. It also parses the running threads list to remove any finished one.

combo_callback(self, event)

When the user changes the channel selected in the "combobox" graphical object, this method is executed and it indicates to the "plot_int" method which channel is now selected.

exit_pogr(self)

This method is called when the user closes the graphical or the main window. Before finishing, this method waits until the last packets are processed and for all threads to finish. Then, it closes the UDP communication and the binary file. Finally, it destroys the window.

5.2.2. Running overview

When the user presses the button “Open graph / Save data” of the main window, the application creates an object of this class. The constructor initializes the window and the UDP ethernet connection, and the reception thread is initiated. It waits for incoming frame data. If sufficient packets are received or the timeout has occurred, processing threads are created and started. They will run until they have treated all their data. Beside these threads, a periodical process refreshes the plots.

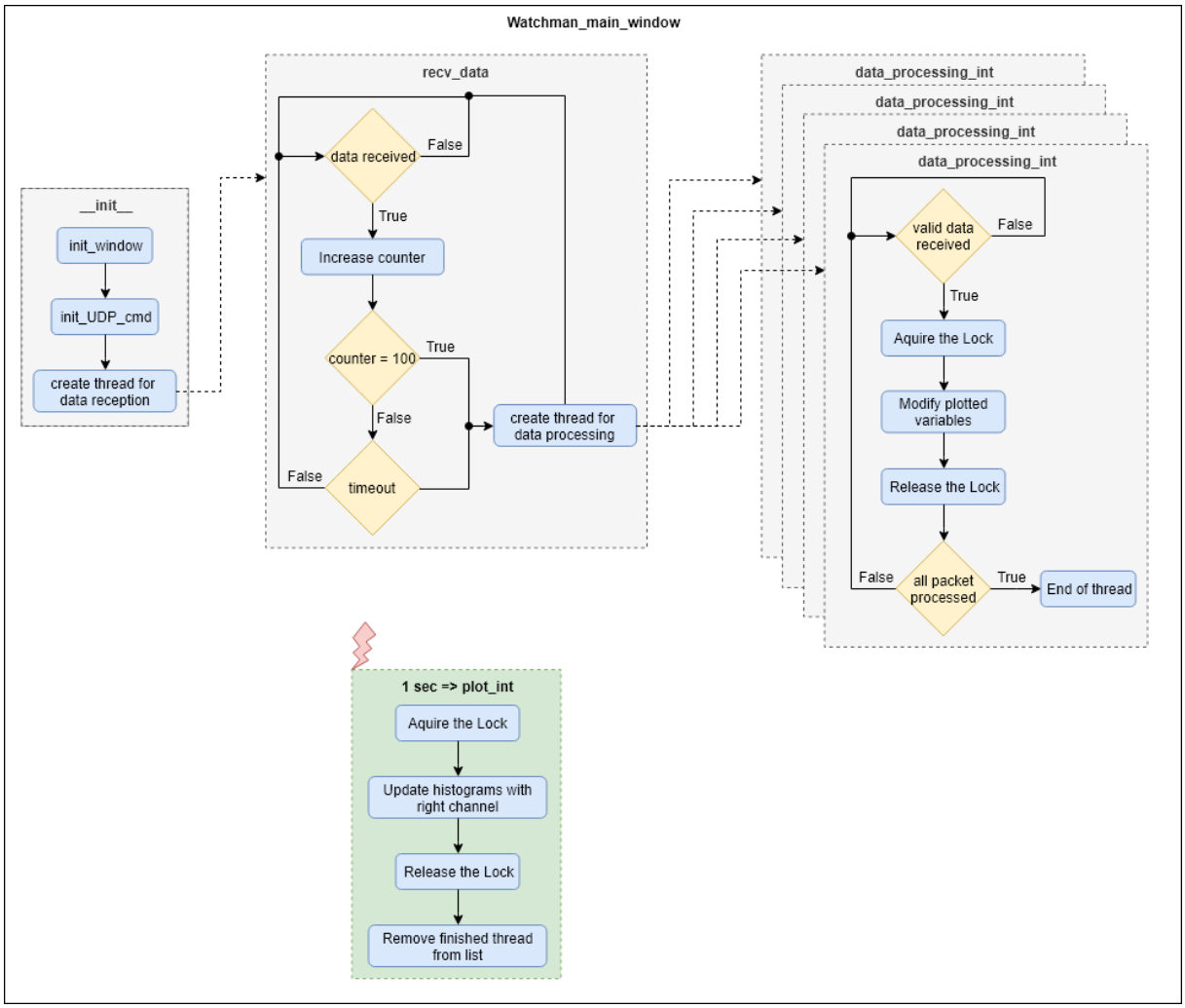


Figure 5-4: Overview of graphic window

6. Results and discussions

All the steps in the development chapter have one goal, it is to have a system capable of managing errors while reading out, processing and sending data. Therefore, two final tests have been done on those two main tasks.

6.1. Error management

This test has for goal to see how the system reacts and manages eventual errors, and then also how it logs them. To do so, four new buttons corresponding to four new commands have been implemented in the GUI. They all send a command leading to a known error which will triggered one of the monitoring processes.

Watchdog

If the program freezes, gets lost, or enters an infinity loop, the watchdog counter is no longer reloaded. In consequence, the watchdog resets the system, and it reboots. After restarting, a message is written in the log file with the timestamp of the event.

This error simulation is initiated when the Zynq received the command “err_watchdog”. It will enter an infinity loop, in which case the watchdog is triggered. Prior to the error command, the time is set by the user, so the stored message will have the right timestamp.

```
Start while loop
Command settime received
Command err_watchdog received


------START------
Global variables initialization pass!
Mounting SD card pass!
Open existing file
Log file creation pass!
Open existing file and updated the time
Time file creation pass!
setup_scu_wdt_int: WATCHDOG has expired
07.02.2019 @ 20:29:02 : WATCHDOG timer occurs!
```

Figure 6-1: Watchdog reboot (terminal view)

As shown in the *Figure 6-1*, the program has rebooted after receiving the command to simulate an infinity loop. In the booting messages, it is written that the watchdog has occurred (see last line of terminal).

Error from a function

Most of the functions implemented in the software return a status after being called. If everything went well, the program continues its execution. In the other case, the processor calls an “end” function which logs the message given in the argument and applies a software reset. The message written in the terminal before the “END” mark, is also saved into the log file by the “end” function.

```
Start while loop
Command err_function_prob received
07.02.2019 @ 20:29:34 : In main: Error function problem ask from user

-------END-------


------START------
Global variables initialization pass!
Mounting SD card pass!
Open existing file
Log file creation pass!
Open existing file and updated the time
Time file creation pass!
WATCHDOG has not expired
```

Figure 6-2: Reboot after a function returned an error (terminal view)

Exception

The exception raised here is an example of data abort (see *Table 4-2*) meaning an attempt to write or read at an illegal address, which in this test is the Protocol Control Block of the command line.

```
Start while loop
Command err_exception received
07.02.2019 @ 20:30:08 : Exception ID = 4



------START------
Global variables initialization pass!
Mounting SD card pass!
Open existing file
Log file creation pass!
Open existing file and updated the time
Time file creation pass!
WATCHDOG has not expired
```

Figure 6-3: Reboot after exception (terminal view)

Assertion

Some libraries implement “assert.h”, such as “lwip.h”. When an assertion is raised, the defined callback is summoned. In order to test if this callback function used for the application is triggered correctly, the function “Xil_Assert” simulates an assertion.

The message displayed below verifies that the defined callback function has been called.

```
Start while loop
Command err_assertion received
07.02.2019 @ 20:30:46 : Assert in file ../src/main.c @ line 285


------START------
Global variables initialization pass!
Mounting SD card pass!
Open existing file
Log file creation pass!
Open existing file and updated the time
Time file creation pass!
WATCHDOG has not expired
```

Figure 6-4: Reboot after assertion (terminal view)

Log file

All the tests listed above have been performed sequentially. Therefore they have been logged one after the other in the file. It seems that the time continues to run, because it was set before the first test and all reboot sequence will try to read the last value in the time file if existing.

```
07.02.2019 @ 20:29:02 : WATCHDOG timer occurs!
07.02.2019 @ 20:29:34 : In main: Error function problem ask from user
07.02.2019 @ 20:30:08 : Exception ID = 4
07.02.2019 @ 20:30:46 : Assert in file ../src/main.c @ line 285
```

Figure 6-5: Log file messages

6.2. Data processing

When the DMA has sent a window, a new element is added to the list as explain in the *chapter 4.2.4 DMA*. Once the processor has received a full waveform, it can process the data and send them to the computer. The trigger system in the PL side has a problem, in consequence the windows received in the PS side cannot be properly used. Therefore, a waveform simulating a pulse on a channel has been created for this test. In order to verify the correct behavior of the functions used to parse and process the list, the program has been run in "Debug" mode while looking at the evolution of the memory.

6.2.1. Calculation and expected values

The simulation parameters are the following:

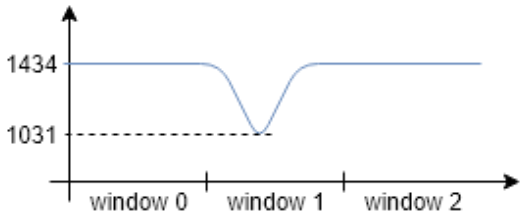


Figure 6-6: Representation of the simulated pulse

The signal is composed of three windows on channel 11. The first and last windows are set to a constant value 1434 equal to V_{ped}. The second window simulates a PMT pulse with a "maximum" amplitude on its sample 13, which is the sample 45 of the signal.

From the simulated values, the expected outcome should be as follow:

- The threshold value for the gain stage selection is set to 800. Therefore the channel selected should be the one planned (11).
- As mentioned previously, the number of windows selected should be 3.
- The returned amplitude should be 1031
- The returned time should be:

$V_{max} = 1031 \rightarrow V_{20\%} = 1434 - 0.2 * (1434 - 1031) = 1353.4$

By looking into the simulation data, the amplitude of 1353.4 is located between 34 (1372) and sample 35 (1341). The time between these two points is then interpolated:

$y = a * x + b$

$\mathrm{p}_1(34\,;\,1372)$

$\mathrm{p}_2(35\,;\,1341)$

$$a = \frac{1341 - 1372}{35 - 34} = -31 \rightarrow b = y - a * x \rightarrow b = 1372 - (-31) * 34 = 2426$$

$$x = \frac{y - b}{a} \rightarrow time = \frac{1353.4 - 2426}{-31} = 34.6$$

So, the GUI should receive frame "features extraction" (ID = 0), containing a amplitude of 1031 and a time of 34.6.

6.2.2. Simulation

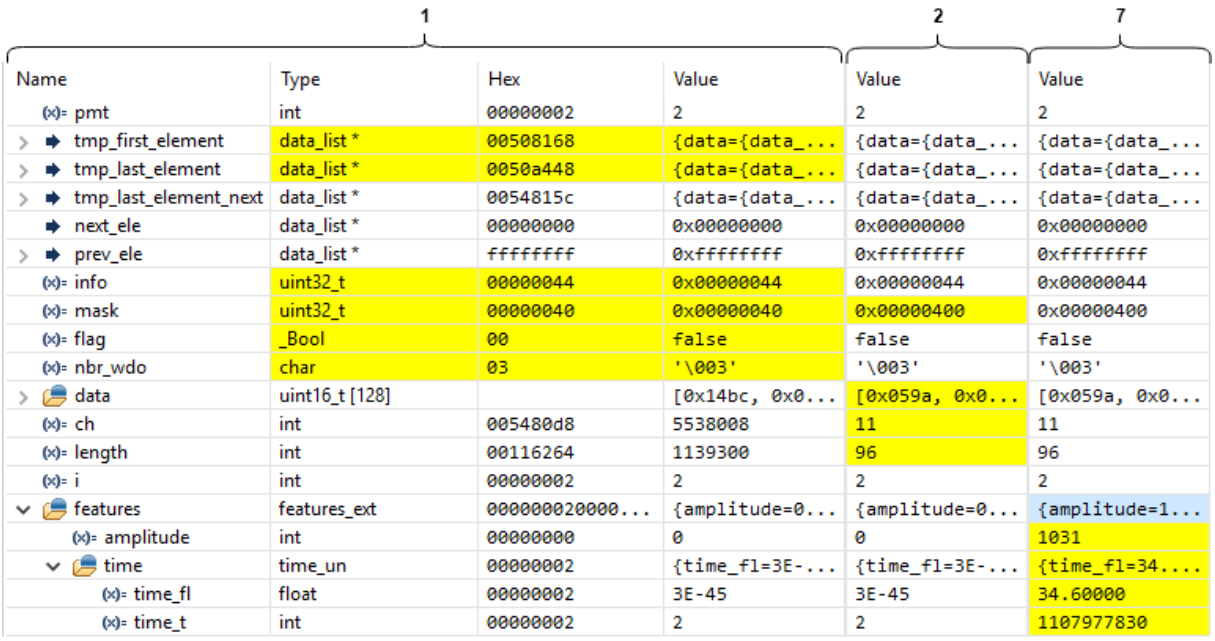
The *Figure 6-7* shows the initial state of the list, with the memory address of each elements. It has a fourth element which would be the one that the DMA will use for the next transfer.

Name	Type	Hex
first_element	data_list *	00508168
data	data_axi_un	
previous	data_list *	00000000
next	data_list *	00509c20
data	data_axi_un	
previous	data_list *	00508168
next	data_list *	0050a448
data	data_axi_un	
previous	data_list *	00509c20
next	data_list *	0050ac70
data	data_axi_un	
previous	data_list *	0050a448
next	data_list *	00000000
last_element	data_list *	0050ac70
data	data_axi_un	
previous	data_list *	0050a448
next	data_list *	00000000
&(first_element->data)	data_axi_un *	00508168
&(first_element->next->data)	data_axi_un *	00509c20
&(first_element->next->next->data)	data_axi_un *	0050a448
&(first_element->next->next->next->data)	data_axi_un *	0050ac70
flag_axidma_rx	int [4]	00000000000...

Figure 6-7: List of element before the function "dma_received_data"

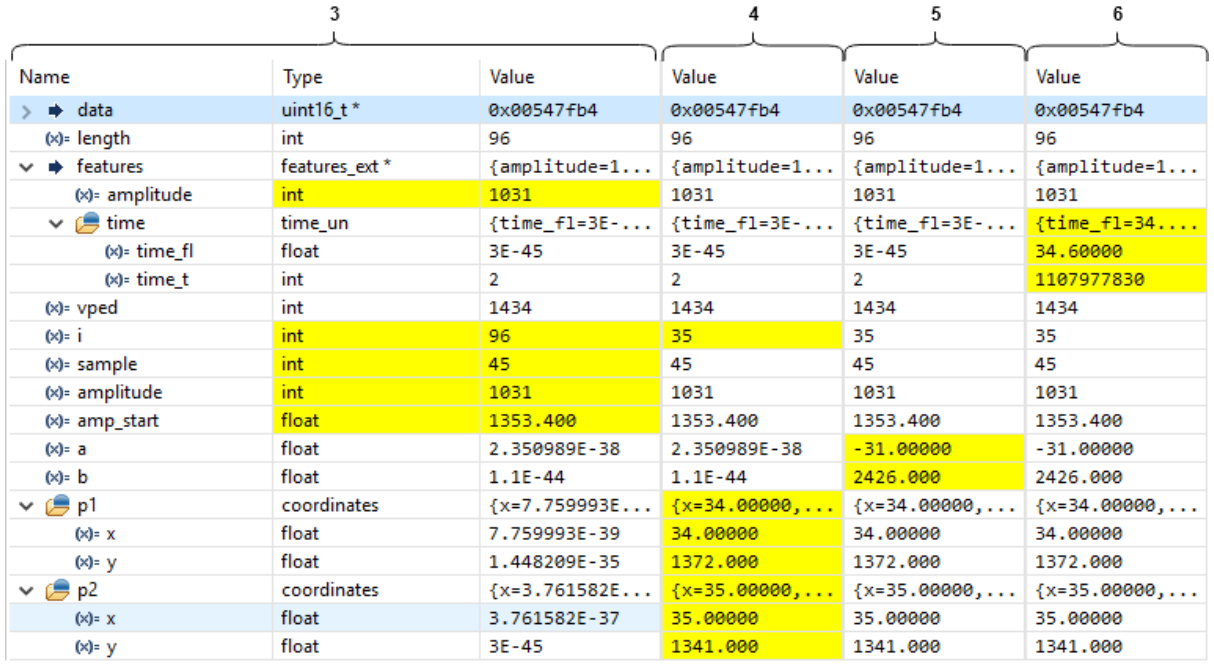
The *Figure 6-8* et *Figure 6-9* show several memory locations at certain steps in the execution of the function "dma_received_data" which is called from the main when a complete waveform has been received.

1) At this step, the program has found the first (0x508168) and last element (0x50A448) which are part of the waveform for the PMT 2. It has also counted the number of windows representing the signal (3).
2) The function "correct_data" which returns an array containing the data corrected with the pedestal subtraction and the transfer function correction. It also returns the channel selected (11) and the length of the data array (96).
3) The program has entered the function "extract features". It has found the "maximum" amplitude of the signal (45 ; 1031) and calculated the value representing 20% of its difference with V_{ped} (1353.4)
4) With $V_{20\%}$ found, the processor has parsed back the waveform data and found the two points surrounding this value. (p1 = (34 ; 1372) & p2 = (35 ; 1341))
5) The coefficients "a" and "b" from the linear function "y = a * x + b" have been found. (a = -31 & b = 2426)
6) At the end of the function, the time has been calculated. (34.6)
7) The program has returned from the function "extract_features" and can now initiate a UDP transfer to the computer



	1			2	7
Name	Type	Hex	Value	Value	Value
pmt	int	00000002	2	2	2
tmp_first_element	data_list *	00508168	{data={data_...	{data={data_...	{data={data_...
tmp_last_element	data_list *	0050a448	{data={data_...	{data={data_...	{data={data_...
tmp_last_element_next	data_list *	0054815c	{data={data_...	{data={data_...	{data={data_...
next_ele	data_list *	00000000	0x00000000	0x00000000	0x00000000
prev_ele	data_list *	ffffffff	0xffffffff	0xffffffff	0xffffffff
info	uint32_t	00000044	0x00000044	0x00000044	0x00000044
mask	uint32_t	00000040	0x00000040	0x00000400	0x00000400
flag	_Bool	00	false	false	false
nbr_wdo	char	03	'\003'	'\003'	'\003'
data	uint16_t [128]		[0x14bc, 0x0...	[0x059a, 0x0...	[0x059a, 0x0...
ch	int	005480d8	5538008	11	11
length	int	00116264	1139300	96	96
i	int	00000002	2	2	2
features	features_ext	000000020000...	{amplitude=0...	{amplitude=0...	{amplitude=1...
amplitude	int	00000000	0	0	1031
time	time_un	00000002	{time_fl=3E-...	{time_fl=3E-...	{time_fl=34....
time_fl	float	00000002	3E-45	3E-45	34.60000
time_t	int	00000002	2	2	1107977830

Figure 6-8: Memory representation for the variables used in function “dma_received_data”



	3		4	5	6
Name	Type	Value	Value	Value	Value
data	uint16_t *	0x00547fb4	0x00547fb4	0x00547fb4	0x00547fb4
length	int	96	96	96	96
features	features_ext *	{amplitude=1...	{amplitude=1...	{amplitude=1...	{amplitude=1...
amplitude	int	1031	1031	1031	1031
time	time_un	{time_fl=3E-...	{time_fl=3E-...	{time_fl=3E-...	{time_fl=34....
time_fl	float	3E-45	3E-45	3E-45	34.60000
time_t	int	2	2	2	1107977830
vped	int	1434	1434	1434	1434
i	int	96	35	35	35
sample	int	45	45	45	45
amplitude	int	1031	1031	1031	1031
amp_start	float	1353.400	1353.400	1353.400	1353.400
a	float	2.350989E-38	2.350989E-38	-31.00000	-31.00000
b	float	1.1E-44	1.1E-44	2426.000	2426.000
p1	coordinates	{x=7.759993E...	{x=34.00000,...	{x=34.00000,...	{x=34.00000,...
x	float	7.759993E-39	34.00000	34.00000	34.00000
y	float	1.448209E-35	1372.000	1372.000	1372.000
p2	coordinates	{x=3.761582E...	{x=35.00000,...	{x=35.00000,...	{x=35.00000,...
x	float	3.761582E-37	35.00000	35.00000	35.00000
y	float	3E-45	1341.000	1341.000	1341.000

Figure 6-9: Memory representation for the variables used in function “extract_features”

The last step is to remove all unnecessary elements, this is to say with no other trigger information. In this simulation, the first three element of the list were part of the waveform and were successfully removed (see Figure 6-10). The list is now empty, as “first_element” and “last_element” are pointing to the same address.

Name	Type	Hex
first_element	data_list *	0050ac70
data	data_axi_un	
previous	data_list *	00000000
next	data_list *	00000000
last_element	data_list *	0050ac70
data	data_axi_un	
previous	data_list *	00000000
next	data_list *	00000000

Figure 6-10: List of element after the function “dma_received_data”

As shown in the two figures below, the Python application has received the UDP packet. The receive values match the expected values, and the GUI has plotted them as expected. Therefore, the test is a success and verifies the good behavior of the full chain from DDR to computer.

```
length = 21
channel = 11
nbr_wdo = 3
frame_id = 0
wdo_time = 0x8389c
wdo_time = 134695.0
amp = 1031
time = 34.599998474121094
```

Figure 6-11 Python terminal view

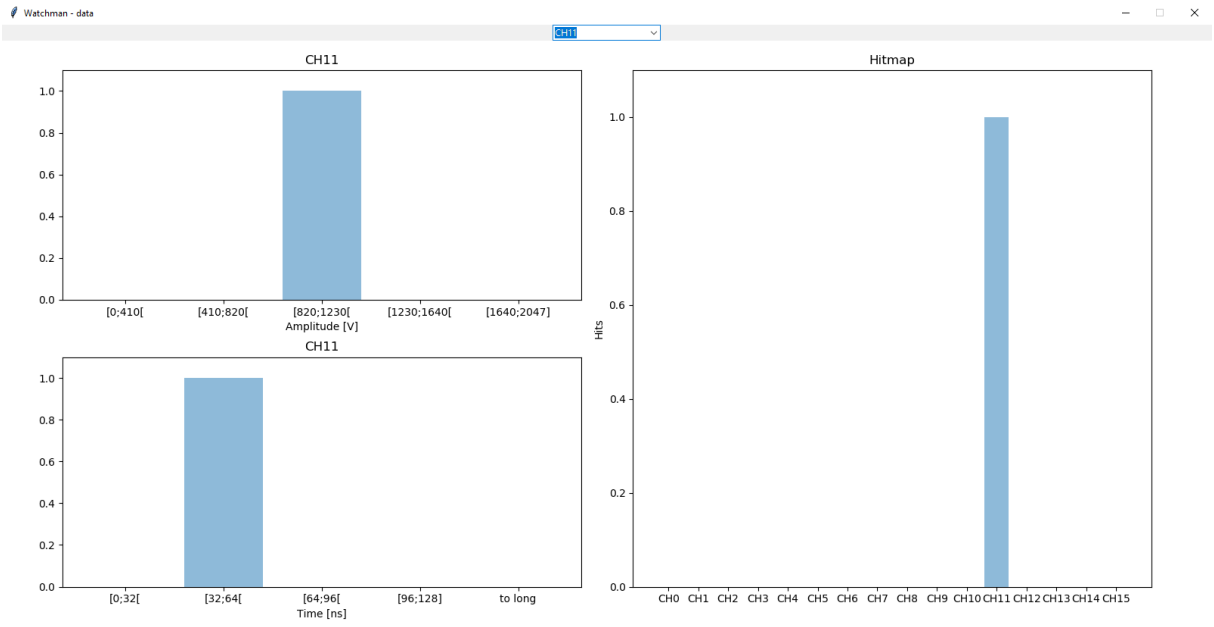


Figure 6-12: Graphical window at the end of the simulation

7. Documentation

One of the objectives was the complete documentation of all the software, Zynq application and GUI application. Indeed, the project WATCHMAN has just started and will continue for at least 2 years. So, the software will definitely be modified and improved after this thesis. It is always complicated to understand someone's work when it is poorly commented. Therefore, the program in C language and in Python have been carefully annotated in such a way that Doxygen could interpret them and generate the on-line documentation browser (HTML).

C Language

The following examples show how to write the comment in C language, so Doxygen interprets them correctly.

- Global variables and defines

```
/** @brief Pointer on the network interface */
extern struct netif *echo_netif;
```

- Enumerations, structures and unions

```
/**
 * @brief This is the enumeration of the process to stop when exiting the program
 */
typedef enum clean_state_enum {
    GLOBAL_VAR=0x1, /**< Free the global variable malloc in fct init_global_var */
    INTERRUPT=0x2,  /**< Stop the interrupt */
    UDP=0x4,        /**< Close both of the UDP communications */
} clean_state_en;
```

- Functions declaration

```
/**
* @brief    Extract the minimum amplitude of the pulse, and the time when
*           it was 20% of its value
*
* @param    data: pointer on the pulse's data
* @param    length: size of data
* @param    features: pointer on structure to return the amplitude and the time

* @return   None
*
* @note     -
*
*******************************************************************************/
void extract_features(uint16_t* data, int length, features_ext* features){
```

Python

The "Doxygen" way to write comments are different among languages.

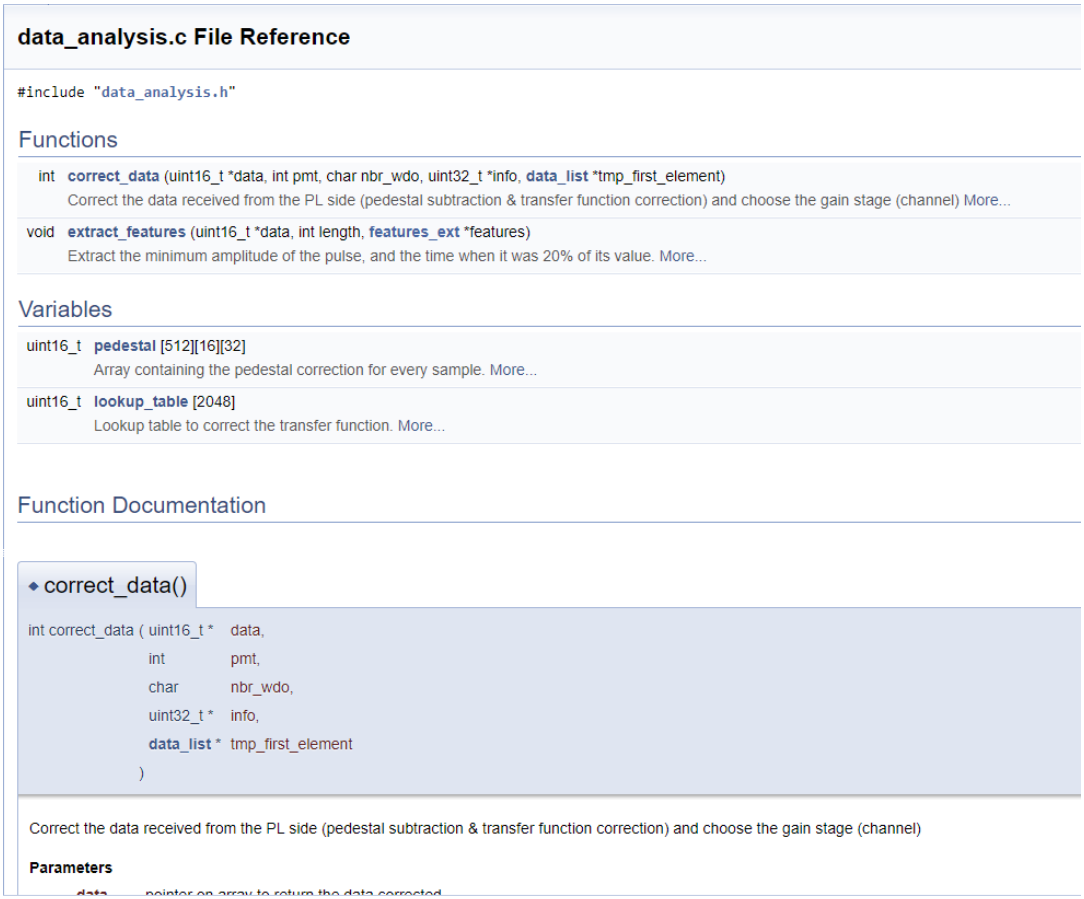
- Public Attributes (to prevent certain public attributes from being considered, like graphical object, the name must start with two underscores (example "__button")

```
## Flag which indicates if the graphical window is open or not
        self.toplevel_flag = False
```

- Methods

```
## Method callback called when a caracter is written is one of the register entry
    # @param self : The object pointer
    # @param count : Register's number
    # @param var : Variable contained by the entry
    # @param *args : Unused
    def entry_callback(self, count, var, *args):
```

Once the HTML file are generated, they can be open with any web browsers.



data_analysis.c File Reference

`#include "data_analysis.h"`

Functions

int **correct_data** (uint16_t *data, int pmt, char nbr_wdo, uint32_t *info, **data_list** *tmp_first_element)
Correct the data received from the PL side (pedestal subtraction & transfer function correction) and choose the gain stage (channel) More...

void **extract_features** (uint16_t *data, int length, **features_ext** *features)
Extract the minimum amplitude of the pulse, and the time when it was 20% of its value. More...

Variables

uint16_t **pedestal** [512][16][32]
Array containing the pedestal correction for every sample. More...

uint16_t **lookup_table** [2048]
Lookup table to correct the transfer function. More...

Function Documentation

◆ correct_data()

```
int correct_data ( uint16_t *   data,
                   int          pmt,
                   char         nbr_wdo,
                   uint32_t *   info,
                   data_list *  tmp_first_element
                 )
```

Correct the data received from the PL side (pedestal subtraction & transfer function correction) and choose the gain stage (channel)

Parameters

Figure 7-1: Example of Doxygen documentation generated from language C file

8. Conclusions

After 6 months, the young prototype board had grown into a reliable and autonomous system. The initial objectives set prior the departure to Hawaii would now be feasible.

8.1. Objectives achievement

Looking at the objectives, most of them have clearly been achieved. The Ethernet communication is running smoothly and can handle larger sets of data transfer than expected. The parse in search of the many problems encountered by using the library lwIP are resolved, even though the depth of this library has still many mysteries. The Python GUI offers first hand preview on the trigger events and proposes a large set of commands to interact with the system. The application proposes buttons to initiate some self-tests which are incorporated into the system to verify its functionality. Overall, the application was developed with the possibility to easily incorporate new features.

The system can recover from a faulty or frozen state, indeed the system has a built-in fault detection and error management system, as well as a watchdog timer. All these errors are reported and logged into a SD card. The logs have time reference from the system which helps in identifying the origin of the problem, which could be an external interference or a bug in the program. In which case the processor saves its last steps before resetting the entire system.

Regarding the data processing, the pedestal subtraction and transfer function corrections have been tested and verified. Unfortunately the latest events show that the TARGETC has a strange behavior on window read back. The data seems to be flipped, inverted or even over-written. At this point it is impossible to assess the origin of this erroneous behavior. As a consequence the timing correction on the data could not be verified, nonetheless the simulation for the feature extraction looks promising. The data is processed correctly and the features are sent to the GUI which can validate them knowing the simulated parameters.

8.2. Perspective

In the current state of the project, the GUI application in Python is more than sufficient to interact with the system. However in a near future, a new board will be mounted with four TARGETC and two MicroZed boards. This upgrade will probably need a more efficient Data Acquisition tool, such as ToolDAQ.

So far the UDP protocol is running fine and fast, but the TCP protocol might be able to handle the data rate with the advantage of guaranteeing the transfer of each packet, thus an upgrade from UDP to TCP protocol at some point should be considered.

In order to have a system completely autonomous, it would be interesting to be able to test the most important elements of the board, starting with the TARGETC and ADC return of voltages.

8.3. Personal conclusion

I always wanted to work on Zynq which offers the possibility to have a close collaboration between firmware and software, while having two very distinct projects, because of its two different parts, the FPGA and the ARM Core. I really appreciated to have the opportunity to develop an application in Python using new methods, like classes, multithreading and multi-windows. A personal objective was to improve my English level which is a success, I can now speak fluently.

Honolulu, the 7th of February 2019

Anthony Schluchin

9. Bibliography

[1] 1. **Duron, Jose.** WATCHMAN - TC prototype revC. *ID Lab Task and Scheduling.* [Online] https://www.phys.hawaii.edu/~idlab/taskAndSchedule/WATCHMAN/WATCHMAN_TC_prototype_revC.pdf.

[2] 2. **Technology, Linear.** Analog Device. *LTC2657.* [Online] https://www.analog.com/media/en/technical-documentation/data-sheets/2657f.pdf.

[3] 3. **Zeboard.** Digilent. *Zedboard Programming Guide.* [Online] https://reference.digilentinc.com/learn/programmable-logic/tutorials/zedboard-programming-guide/start.

[4] 4. **ChaN.** FatFs - Generic FAT Filesystem Module. *Electronic Lives Manufacturing.* [Online] http://elm-chan.org/fsw/ff/doc/mount.html.

[5] 5. **Taylor, Adam.** MicroZed Chronicles. *Zedboard.* [Online] http://zedboard.org/content/microzed-chronicles.

[6] 6. **Xilinx.** documentation - user guides. *xilinx.* [Online] https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

10. List of figures

11. Appendices

All the source code for Python application, as well as the C code and their Doxygen documentation, can be found on WATCHMAN GitHub repository.

https://github.com/WMidlab/WATCHMAN