



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA

– **TELECOM** ESCUELA  
TÉCNICA **VLC** SUPERIOR  
DE INGENIERÍA DE  
TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Telecommunications Engineering

Development of a system for monitoring and control of the sensors in the data processing module of the Hyper-Kamiokande neutrino detector.

End of Degree Project

Bachelor's Degree in Telecommunication Technologies and Services Engineering

AUTHOR: Martínez Sánchez, Borja

Tutor: Ballester Merelo, Francisco José

ACADEMIC YEAR: 2023/2024



## Resumen

Este TFG se realiza en el marco del proyecto internacional Hyper-Kamiokande, que tiene por objetivo construir el detector más avanzado y de mayor tamaño de neutrinos basado en la detección de luz de Cherenkov en agua ultrapura. Este detector será construido en Japón, en la antigua mina de Kamioka (prefectura de Gifu) y será el sucesor de Super-Kamiokande. El principal cambio de Super-Kamiokande a Hyper-Kamiokande es el uso de sensores PMT mucho más sensibles para captar el fenómeno conocido como “luz de Cherenkov”. Esto provoca que la electrónica de conversión tenga que ponerse cerca de los PMT, en las propias vasijas. Por tanto, se hace necesario que la propia electrónica sea capaz de monitorizar su estado a través de diversos sensores y reportarse a través de un enlace *Ethernet* a los servidores situados en un *data center* encima del observatorio. Este TFG trata sobre la implementación de estas funcionalidades, conocidas como *slow control* para la DPB, que es el módulo que la UPV tiene como tarea diseñar y enviar estos datos a través de un enlace de *Ethernet*. Además, estas medidas también serán muy útiles a la hora de realizar tests con los prototipos para llegar a un diseño y fabricación finales de unos módulos que serán montados en el observatorio durante el año 2026.

## Resum

Aquest TFG es realitza en el marc del projecte internacional *Hyper-Kamiokande*, que té per objectiu construir el detector més avançat i de major grandària de neutrins basat en la detecció de llum de Cherenkov en aigua ultrapura. Aquest detector serà construït al Japó, en l'antiga mina de Kamioka (prefectura de Gifu) i serà el successor de Super-Kamiokande. El principal canvi de Super-Kamiokande a Hyper-Kamiokande és l'ús de sensors PMT molt més sensibles per a captar el fenomen conegut com a “llum de Cherenkov”. Això provoca que l'electrònica de conversió haja de posar-se prop dels PMT, en els propis atuells. Per tant, es fa necessari que la pròpia electrònica siga capaç de monitorar el seu estat a través de diversos sensors i reportar-se a través d'un enllaç *Ethernet* als servidors situats en un *data center* damunt de l'observatori. Aquest TFG tracta sobre la implementació d'estes funcionalitats, conegudes com *slow control* per a la DPB, que és el mòdul que la UPV té com a tasca dissenyar i enviar aquestes dades a través d'un enllaç de *Ethernet*. A més, estes mesures també seran molt útils a l'hora de realitzar tests amb els prototips per a arribar a un disseny i fabricació finals d'uns mòduls que seran muntats en l'observatori durant l'any 2026.

## Abstract

This TFG is carried out in the framework of the international project Hyper-Kamiokande, which aims to build the most advanced and largest neutrino detector based on the detection of Cherenkov light in ultrapure water. This detector will be built in Japan, in the former Kamioka mine (Gifu prefecture) and will be the successor of Super-Kamiokande. The main change from Super-Kamiokande to Hyper-Kamiokande is the use of much more sensitive PMT sensors to capture the phenomenon known as “Cherenkov light”. This results in the conversion electronics having to be placed close to the PMTs, in the vessels themselves. Therefore, it is necessary for the electronics themselves to be able to monitor their status through various sensors and report through an *Ethernet* link to the servers located in a data center on top of the observatory. This TFG is about the implementation

---

of these functionalities, known as slow control for the DPB, which is the module that UPV has the task to design and send these data through an *Ethernet* link. In addition, these measurements will also be very useful when testing the prototypes to reach a final design and manufacture of modules that will be mounted in the observatory during the year 2026.

## RESUMEN EJECUTIVO

La memoria del TFG del GTIST debe desarrollar en el texto los siguientes conceptos, debidamente justificados y discutidos, centrados en el ámbito de la IT

| CONCEPT (ABET)   | CONCEPTO (traducción)   | ¿Cumple?<br>(S/N) | ¿Dónde?<br>(páginas)    |
|--|---|-------------------|-------------------------|
| 1. IDENTIFY:   | 1. IDENTIFICAR:   |                   |                         |
| 1.1. Problem statement and opportunity   | 1.1. Planteamiento del problema y oportunidad   | S                 | 1                       |
| 1.2. Constraints (standards, codes, needs, requirements & specifications)          | 1.2. Toma en consideración de los condicionantes (normas técnicas y regulación, necesidades, requisitos y especificaciones) | S                 | 1                       |
| 1.3. Setting of goals  | 1.3. Establecimiento de objetivos   | S                 | 2                       |
| 2. FORMULATE:  | 2. FORMULAR:  |                   |                         |
| 2.1. Creative solution generation (analysis)                                       | 2.1. Generación de soluciones creativas (análisis)  | S                 | 55, 59,70, 73,79,83, 87 |
| 2.2. Evaluation of multiple solutions and decision-making (synthesis)              | 2.2. Evaluación de múltiples soluciones y toma de decisiones (síntesis)   | S                 | 27, 59,68, 70,73,79, 83 |
| 3. SOLVE:  | 3. RESOLVER:  |                   |                         |
| 3.1. Fulfilment of goals   | 3.1. Evaluación del cumplimiento de objetivos   | S                 | 95-96                   |
| 3.2. Overall impact and significance (contributions and practical recommendations) | 3.2. Evaluación del impacto global y alcance (contribuciones y recomendaciones prácticas)                                   | S                 | 95-96                   |

I would like to take this opportunity in my TFG to thank all those who have accompanied me during these 4 years of my studies for putting up with me and making this period of my life enjoyable. I would also like to thank Francisco Ballester and the I3M team for trusting me and providing me with my first professional experience as an engineer. Specifically, I want to express my gratitude to Alejandro Gómez Gambín for his time and guidance as my mentor during my internship, and for everything I have learned while working with him- Lastly, I must thank my family and parents for their unwavering emotional and financial support, and for always believing in me.

# Contents

## I Introduction: Hyper-Kamiokande Project

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>TFG goals</b>                            | <b>1</b> |
| <b>2</b> | <b>The neutrino itself</b>                  | <b>3</b> |
| <b>3</b> | <b>HKK project</b>                          | <b>5</b> |
| 3.1      | HKK project structure . . . . .             | 5        |
| 3.2      | HKK physics basis . . . . .                 | 8        |
| 3.3      | HKK predecessor, Super-Kamiokande . . . . . | 9        |
| 3.4      | HKK objectives . . . . .                    | 10       |
| 3.5      | HKK project organization . . . . .          | 12       |

## II Leveraged technology

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>DPB SoM election</b>                                  | <b>15</b> |
| 1.1      | Understanding System-on-Module . . . . .                 | 15        |
| 1.1.1    | Advantages of SoM Technology: . . . . .                  | 15        |
| <b>2</b> | <b>DPB2 Prototype</b>                                    | <b>17</b> |
| 2.1      | Zynq UltraScale+ Architecture . . . . .                  | 17        |
| 2.1.1    | Difference between PS and PL . . . . .                   | 18        |
| 2.1.2    | JTAG interface . . . . .                                 | 19        |
| 2.1.3    | UART interface . . . . .                                 | 20        |
| 2.1.4    | Ethernet interface . . . . .                             | 21        |
| 2.1.5    | I <sup>2</sup> C interface . . . . .                     | 22        |
| 2.1.6    | GPIO interface . . . . .                                 | 25        |
| 2.1.7    | RS-485 communication protocol . . . . .                  | 26        |
| <b>3</b> | <b>PetaLinux embedded OS</b>                             | <b>27</b> |
| 3.1      | Unix/Linux environment and OS election . . . . .         | 27        |
| 3.1.1    | Used Linux libraries, drivers and applications . . . . . | 28        |
| 3.2      | Sysfs file system . . . . .                              | 29        |
| <b>4</b> | <b>Development environments used</b>                     | <b>31</b> |
| 4.1      | Vivado Design Suite . . . . .                            | 31        |
| 4.2      | Vitis IDE . . . . .                                      | 32        |
| 4.3      | Robot Framework . . . . .                                | 34        |

---

### III DPB sensors capabilities descriptions for *slow control* tasks

|  |           |
|--|-----------|
| <b>1 I<sup>2</sup>C devices</b>            | <b>37</b> |
| 1.1 INA3221 Current sensor . . . . .       | 37        |
| 1.2 MCP9844 Temperature sensor . . . . .   | 42        |
| 1.3 AFBR-5715ALZ SFP Transceiver . . . . . | 44        |
| <b>2 Xilinx AMS gathered data</b>          | <b>49</b> |

### IV Tasks Development and Results

|  |           |
|--|-----------|
| <b>1 Preparation of the environment to be used on the board</b>                  | <b>55</b> |
| 1.1 Platform setup and configuration . . . . .                                   | 55        |
| 1.2 Vitis Project Creation . . . . .   | 56        |
| <b>2 Application workflow</b>  | <b>59</b> |
| <b>3 Application initialization</b>  | <b>67</b> |
| <b>4 Monitoring thread development</b>   | <b>73</b> |
| 4.1 Sensor data readout functions . . . . .                                      | 73        |
| 4.2 Parse monitoring data into JSON string and send it to the DAQ . . . . .      | 74        |
| <b>5 Alarms threads development</b>  | <b>79</b> |
| 5.1 Configure shared memory segment and synchronization semaphores . . . . .     | 80        |
| 5.2 Detection and handling sensors alarms functions . . . . .                    | 81        |
| 5.3 Parse alarms data into JSON string and send it to the DAQ . . . . .          | 81        |
| <b>6 Command handling thread development</b>                                     | <b>83</b> |
| 6.1 Parse commands from the DAQ into JSON string for processing . . . . .        | 84        |
| 6.2 Define the command cases and develop functions to handle each case . . . . . | 85        |
| <b>7 Develop manufacturing test software</b>                                     | <b>87</b> |
| 7.1 Adapt previously developed software for manufacturing test . . . . .         | 87        |
| 7.2 Automation of the tests using Robot framework . . . . .                      | 89        |

### V Conclusions and future work

|                          |           |
|--------------------------|-----------|
| <b>1 Lessons learned</b> | <b>95</b> |
| <b>2 Future work</b>     | <b>97</b> |

|                     |           |
|---------------------|-----------|
| <b>Bibliography</b> | <b>99</b> |
|---------------------|-----------|

### VI Annexes

|                              |            |
|------------------------------|------------|
| <b>A Additional Listings</b> | <b>103</b> |
|------------------------------|------------|



# List of Figures

|     |  |    |
|-----|--|----|
| 3.1 | HKK water tank concept sketch . . . . .  | 6  |
| 3.2 | Cross-section of the Hyper-Kamiokande . . . . .  | 6  |
| 3.3 | PMT structure . . . . .  | 7  |
| 3.4 | Interior of the vessel . . . . .   | 7  |
| 3.5 | How a PMT detects <i>Cherenkov light</i> phenomenon . . . . .                            | 8  |
| 3.6 | Interior of the Super-Kamiokande, predecessor of the Hyper-Kamiokande . . . . .          | 9  |
| 3.7 | Hyper-Kamiokande neutrino oscillation investigation fields [5] . . . . .                 | 11 |
| 3.8 | Diagram of communication between the different modules of the vessel . . . . .           | 12 |
|     |  |    |
| 2.1 | Zynq UltraScale+ PS and PL blocks . . . . .  | 19 |
| 2.2 | Daisy-chained JTAG . . . . .   | 20 |
| 2.3 | UART Controller schema . . . . .   | 21 |
| 2.4 | Addressing and data frames I <sup>2</sup> C . . . . .                                    | 23 |
| 2.5 | Structure of the I <sup>2</sup> C of our DPB . . . . .                                   | 24 |
| 2.6 | SDA and SCL I <sup>2</sup> C communication signals . . . . .                             | 25 |
|     |  |    |
| 3.1 | Role of sysfs in user-hardware communication . . . . .                                   | 30 |
|     |  |    |
| 4.1 | Vivado GUI . . . . .   | 32 |
| 4.2 | Vitis Embedded Software Development Flow [18] . . . . .                                  | 33 |
| 4.3 | Vitis GUI . . . . .  | 33 |
| 4.4 | Robot framework workflow [20] . . . . .  | 34 |
| 4.5 | Example of keyword driven test . . . . .   | 34 |
|     |  |    |
| 1.1 | Operation of the alarms MCP9844 Temperature Sensor . . . . .                             | 42 |
|     |  |    |
| 1.1 | Vitis IDE Platform project wizard . . . . .  | 57 |
|     |  |    |
| 2.1 | Main application execution flow . . . . .  | 59 |
| 2.2 | Monitoring thread execution flow . . . . .   | 61 |
| 2.3 | I <sup>2</sup> C alarms execution flow . . . . .   | 62 |
| 2.4 | AMS alarms thread execution flow . . . . .   | 63 |
| 2.5 | <i>Slow Control</i> thread execution flow . . . . .                                      | 64 |
|     |  |    |
| 4.1 | Monitoring thread value . . . . .  | 74 |
| 4.2 | ZeroMQ Publisher-Subscriber simple pattern [28] . . . . .                                | 76 |
| 4.3 | PL Temperature monitored evolution . . . . .   | 77 |
|     |  |    |
| 5.1 | Difference between AMS temperature alarm with hysteresis on and hysteresis off . . . . . | 80 |
| 5.2 | JSON strings received in Python application after triggering alarms . . . . .            | 82 |

|     |   |    |
|-----|---|----|
| 6.1 | Request-Reply pattern [29]                    | 83 |
| 7.1 | Example of keyword-driven test                | 90 |
| 7.2 | Example of data-driven test                   | 90 |
| 7.3 | Basic example of successful Robot test report | 91 |

# List of Tables

|     |   |     |
|-----|---|-----|
| 3.1 | SK Phases . . . . .   | 9   |
| 2.1 | DPB GPIO pin distribution . . . . .                         | 26  |
| 1.1 | INA3221 Current Sensor Registers . . . . .                  | 41  |
| 1.3 | MCP9844 Temperature Sensor Registers . . . . .              | 43  |
| 1.4 | SFP transceiver EEPROM page 1 registers . . . . .           | 45  |
| 1.5 | SFP transceiver EEPROM page 2 registers . . . . .           | 47  |
| 1.6 | Breakdown of SFP transceiver status bits . . . . .          | 47  |
| 1.7 | Breakdown of the <i>flags</i> of SFP transceivers . . . . . | 48  |
| 2.1 | SYSMON channels and their details . . . . .                 | 50  |
| 2.2 | AMS alarms register set . . . . .                           | 51  |
| 7.1 | Basic data structures in ctypes, C y Python . . . . .       | 88  |
| A.1 | Setting DPB command list . . . . .                          | 103 |
| A.2 | Reading DPB command list . . . . .                          | 104 |

# Acronyms

**ACK** Acknowledgment.

**ADC** Analog-to-Digital Converter.

**AMD** Advanced Micro Devices.

**AMS** Analog Mixed-Signal.

**APB** Advanced Peripheral Bus.

**API** Application Programming Interface.

**APU** Accelerated Processing Unit.

**ARM** Advanced RISC Machine.

**AXI** Advanced eXtensible Interface.

**COM** Computer on Module.

**CP** Charge Conjugation and Parity.

**CPU** Central Processing Unit.

**DAQ** Data Acquisition Centre.

**DHCP** Dynamic Host Configuration Protocol.

**DMA** Direct Memory Access.

**DPB** Data Processing Board.

**DSP** Digital Signal Processor.

**DUT** Device Under Testing.

**EEPROM** Electrically Erasable Programmable Read-Only Memory.

**EMIO** Extended Multiplexed I/O.

**eMMC** embedded Multi-Media Card.

**FD** Far Detector.

**FIFO** First In First Out.

**FinFET** Fin Field-Effect Transistor.

**FPD** Full Power Domain.

**FPGA** Filed Programmable Gate Array.

**GCC** GNU Compiler Collection.

**GEM** Gigabit Ethernet Module.

**GMI** Gigabit Media Independent Interface.

**GND** Ground.

**GPIO** General Purpose I/O.

**GUI** Graphical User Interface.

**HKK** Hyper-Kamiokande.

**HV** High Voltage.

**I/O** Input/Output.

**IBM** Institute for Molecular Imaging Instrumentation.

**I<sup>2</sup>C** Inter-Integrated Circuit.

**IC** Integrated Circuit.

**ID** Inner Detector.

**IDE** Integrated Development Environment.

**IEEE** Institute of Electrical and Electronics Engineers.

**IIO** Industrial I/O.

**IOP** I/O Processor.

**IP** Internet Protocol.

**JSON** JavaScript Object Notation.

**JTAG** Joint Test Action Group.

**LPD** Low Power Domain.

**LSB** Least Significant Bit.

**LV** Low Voltage.

**MAC** Media Access Control.

**MDIO** Management Data I/O.

**MIO** Multiplexed I/O.

**MPSoC** Multi-Processing System on Chip.

**MSB** Most Significant Bit.

**NACK** Negative Acknowledgment.

**OS** Operating System.

**PCB** Printed Circuit Board.

**PHY** Physical Layer.

**PL** Programmable Logic.

**PLL** Phase-Locked Loop.

**PMT** Photo-multiplier Tube.

**POSIX** Portable Operating System Interface uniX.

**PS** Processing System.

**PS-GTR** Processing System-Gigabit Transceiver.

**RAM** Random Access Memory.

**RGMI** Reduced Gigabit Media Independent Interface.

**RISC** Reduced Instruction Set Computer.

**ROM** Read Only Memory.

**RPU** Real-time Processing Unit.

**RS** Recommended Standard.

**SCL** Clock Signal.

**SDA** Data Signal.

**SDK** Software Development Kit.

**SFP** Small Form-factor Pluggable.

**SK** Super-Kamiokande.

**SMARC** Smart Mobility ARChitecture.

**SoC** System on Chip.

**SoM** System on Module.

**SSH** Secure SHell.

**SYSMON** SYStem MONitor.

**TAP** Test Access Port.

**Tcl** Tool command language.

**TCO** Total Cost of Ownership.

**TCP** Transmission Control Protocol.

**TFG** Final Degree Project.

**TO** Time Out.

**TSMC** Taiwan Semiconductor Manufacturing Company.

**UART** Universal Asynchronous Receiver-Transmitter.

**UPV** Polytechnic University of Valencia.

**UUID** Universally Unique Identifier.

**VHV** Very High Voltage.

**VLAN** Virtual Local Area Network.

**XPPU** Extended Peripheral Protection Unit.

## **Part I**

# **Introduction: Hyper-Kamiokande Project**





# Chapter 1

## TFG goals

It is well known that telecommunications have played a pivotal role in the course of society since its emergence, being a discipline that nowadays is necessary for almost everything, whether for 2-person telephone communication or military applications.

The field of telecommunications encompasses several branches of knowledge, one of them being electronic systems, where I specialize. Therefore, this Final Degree Thesis of the Degree in Telecommunication Technologies and Services Engineering is clear evidence of the importance of electronic systems in the advancement of human beings to explore and investigate the behaviour of the universe in all of its aspects. Specifically, this thesis will deal with the design of software for an embedded system, a type of electronic system that is very recurrent during the degree in telecommunications engineering and indispensable in a vast majority of electronic components worldwide.

This TFG is part of the Hyper-Kamiokande (HKK) project, which will be described in more detail in the following chapters. It is a massive global project that started in 2018 with around 300 researchers from 22 different countries, and over time the number has grown, which means that project coordination is crucial in the development of the whole project. The responsibilities of the UPV form the nucleus of the experiment's electronics, given the module's design originating from the UPV, particularly the Institute for Molecular Imaging Technologies (I3M).

It has to be taken into account that the electronics of this project have to be inaccessible for at least 10 years as they will be sealed in a vessel, so this thesis can be considered a step forward in the development of the DPB software through the design of an application that will allow to control the state of the electronic components inside the vessel and to communicate the DPB with the DAQ. In addition, it will serve as a basis for the development of automated manufacturing tests for the larger-scale production of the final version of the DPB. All this has to be developed taking into account that most of the processing capacity and memory of the DPB is destined for the data captured by the PMTs, so I have to try to optimize the designed software. Not to mention, I must detect and correct any possible bug that could cause any memory leak or lead the application to crash since neither the vessel nor the DPB will be accessible for at least 10 years and a failure to manually manipulate the electronics in a vessel would lead to the loss of that vessel until the next emptying of the observatory and a great amount of valuable information would be lost. This software will be developed on a Linux platform already designed by project colleagues.

The objectives of this dissertation are summarized as follows:

- **Develop DPB software** in order to allow us to read the information gathered from the sensors or other modules, process commands coming from the DAQ, and transmit the gathered data as packets to the DAQ.
- **Learn how to work in embedded environments** using the AMD Vitis IDE, which allows us to develop C or C++ application code that will run on Xilinx products. Therefore, we will be able to debug and run code on our DPB SoM.
- **Program software in Linux** for our DPB as it runs on an embedded OS derived from Linux, PetaLinux. Therefore, to develop software that will run on this OS, Linux drivers will be used and modified if necessary to achieve the desired functionality. The operation and execution flow of the drivers themselves must be understood in order to be used.
- **Develop *slow control* system** with the aim of precisely monitoring and managing low-frequency signals or events from the DPB, prioritizing stability and accuracy over real-time responsiveness.
- **Create data structures** using JavaScript Object Notation (JSON) format to parse the gathered information and be able to communicate with the DAQ by following these data structures.
- **Manage alarm systems** asserted by the sensing components on the board to be able to act and report in case any of the components is operating outside manufacturer's margins and may compromise the operation or reliability of the DPB.
- **Test preparation and automation** using the Robot framework for testing the mass production of boards (about 900 will be produced for the detector). The aim is to integrate the previously designed software into the test software and to prepare and enumerate the test cases in the Robot framework in order to be able to verify all necessary test cases automatically.

## Chapter 2

# The neutrino itself

Prior to the development of the HKK project, it is useful to know more about the main particle to be detected during this project, the neutrino. Therefore, in this chapter a brief explanation of the knowledge about the neutrino throughout history, and the configurations and phenomena used for the detection of the neutrino and the study of its nature.

Within the field of physics there are countless subfields that study different aspects of everything around us. The project on which this dissertation is based is based on the speciality of physics called *Particle physics* [1], which is also known as high-energy physics, because many of these particles can only be seen in large collisions provoked in particle accelerators. This discipline of physics is responsible for demonstrating the existence of particles classified according to certain characteristics as bosons or fermions. Nonetheless, it also encounters the difficulty of having been able to demonstrate particles that are almost non-detectable to this day.

Within these elusive particles lies the neutrino, a subatomic entity generated during a radioactive decay and scattering phenomenon. In this instance, the neutrino arises from beta decay, as proposed in Fermi's theory, wherein a sizeable neutral particle ( $n^0$ ) disintegrates into a proton ( $p^+$ ), an electron ( $e^-$ ), and a neutrino ( $\bar{\nu}_e$ ).



The first person to postulate the existence of the neutrino theoretically was Wolfgang Pauli [2] in 1930, but it remained undetected for 25 years because this hypothetically predicted particle had to be massless, chargeless and without strong interaction. Finally, in 1956, Clyde Cowan, Frederick Reines, Francis B. "Kiko" Harrison, Herald W. Kruse, and Austin D. McGuire were able to demonstrate the existence of the neutrino experimentally by using a beam of neutrons to pump a tank of pure water. By observing the subsequent emission of the protons, they were able to demonstrate the existence of the neutrino. This test was called the neutrino experiment.

Over the years, different types of radioactive decays have been discovered that can give rise to neutrinos, such as natural and artificial nuclear reactions, supernova events or the spin-down of a neutron star. Furthermore, it has been discovered that there are different leptonic flavours of neutrinos originating from the weak interactions, electron neutrino, muon neutrino and tau neutrino, each flavor is associated with the correspondingly named charged lepton and similar to some other neutral particles, neutrinos oscillate between different flavors in flight as a consequence.

*I have done a terrible thing: I have postulated a particle that cannot be detected.*

– *Wolfgang Pauli, 1930*

This quote comes from Wolfgang Pauli when he postulated the existence of the neutrino as a particle without electric charge or mass to balance the equation, and it is that the discipline that studies the phenomena caused by neutrinos from space has been encountering great difficulties in detecting it for years detecting neutrinos because they interact with almost nothing or only weakly.

A configuration for detecting a decent amount of neutrinos, based on the *Cherenkov light* phenomenon, will be explained in further detail in the following sections. This precise configuration is the physics basis of the project on which this TFG has been developed.

## Chapter 3

# HKK project

This chapter aims to present the structure and objectives of the HKK project, the evolution of its predecessor, SK, and the physics phenomena on which the project is based.

### 3.1 HKK project structure

Hyper-Kamiokande [3] is a neutrino detector project still under construction (estimated to start operation in 2027), which takes place in the Kamioka mines in Japan, surpassing the performance of its predecessor, Super-Kamiokande. Although the project is based in Japan, it involves research institutes from 22 different countries. The aim of the project is to search for anti-neutrinos coming from supernovas, proton decays and detect neutrinos from natural sources such as the Earth, the atmosphere, the Sun and the cosmos, as well as to study neutrino oscillations from the neutrino beam of the artificial accelerator.

Hyper-Kamiokande is planned to be the world's largest neutrino detector, surpassing its predecessor Super-Kamiokande, which is 71 meters high and 68 meters in diameter. The detector, filled with ultrapure water, will have about 40,000 photomultiplier tubes as detectors inside the detector and 10,000 detectors outside the detector. Although HKK is bigger than SK, by including almost 4 times the number of PMTs of its predecessor, HKK achieves a 40% photo-cathode coverage, the same as SK.

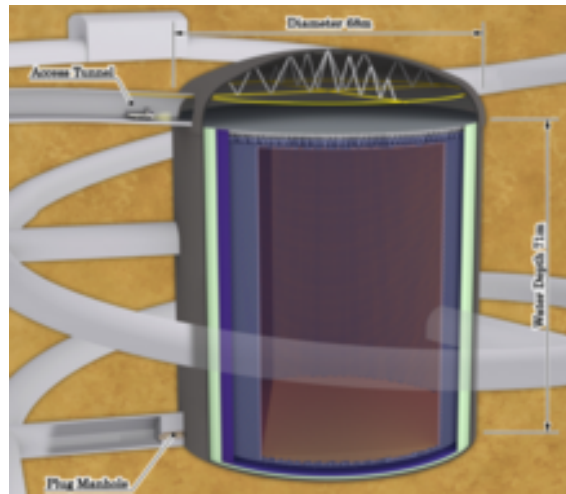


Figure 3.1: HKK water tank concept sketch

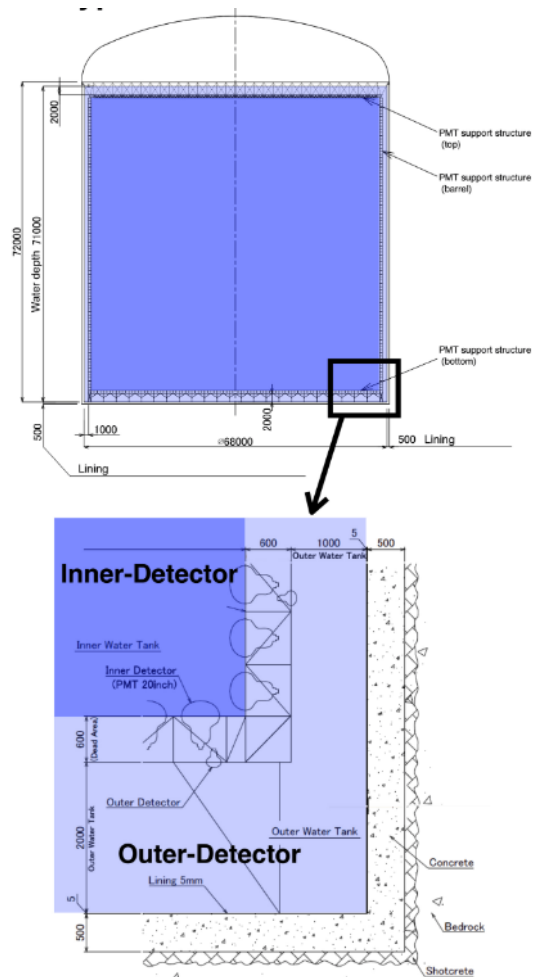
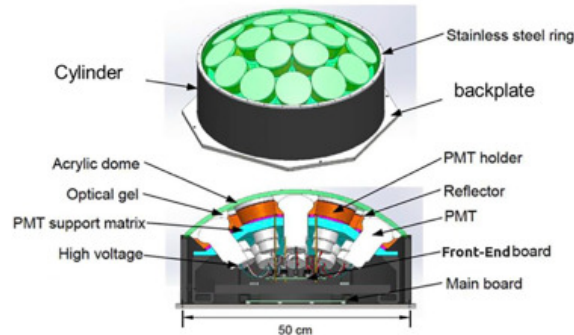


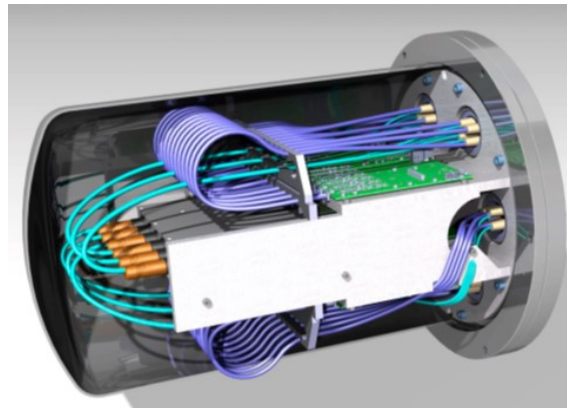
Figure 3.2: Cross-section of the Hyper-Kamiokande

The detector design comprises a cylindrical tank with outer dimensions of 71 meters in height and 68 meters in diameter. It is filled with 260,000 metric tons of ultrapure water to create a water Cherenkov detector. This tank will be surrounded by highly sensitive photodetectors, which boast a 50% higher efficiency compared to the SK ones, thus allowing for greater precision in measuring light intensity and detection time. These photomultiplier tubes (PMTs), specifically the Hamamatsu R12860 model, will enhance the detection of signatures such as those produced in neutrino interactions. Consequently, this setup will enable researchers to more accurately measure the direction and velocity of neutrinos passing through the detector.



**Figure 3.3: PMT structure**

The PMTs, along with the rest of the electronics, will be housed in hermetically sealed vessels submerged in the water inside the detector, following the same structure as the SK.



**Figure 3.4: Interior of the vessel**

As can be seen in the previous figure, the electronics are concentrated inside the vessel, where the information from the PMTs is sampled in the digitizers and then sent to the DPB. The DPB is responsible for communicating the different modules both outside and inside the vessel, it acts as a hub inside the vessel.

Since the electronics are located in a place that is difficult to access, as it would mean emptying the detector of water, high reliability is required in this project, at least 10 years. For this reason, robust systems have been chosen and the electronics used must be monitored.



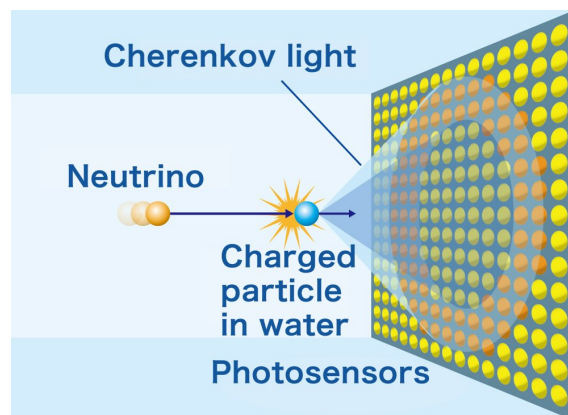
## 3.2 HKK physics basis

The physics apparatus used to study neutrinos is referred to as a neutrino detector, built to be isolated from any other influence like cosmic rays or background radiation. These neutrino detectors are huge structures that work following a neutrino detection technique of the existent ones let it be scintillators (like in the Cowan-Reines neutrino experiment), radiochemical methods, radio detectors or *Cherenkov light* detectors. The experiment that gives name to this chapter is based on the latter: the *Cherenkov light* detection [4].

These detectors are huge water-filled tanks enriched with deuterium and gadolinium. This environment is ideal for neutrino interaction as the interaction of one of these subatomic particles with the electrons or nuclei of water can produce a charged particle faster than the speed of light in water. This produces a cone of light called *Cherenkov light* and can be defined as the equivalent of light to a sonic boom in acoustic waves.

The water tank is surrounded by photosensible sensors called Phototubes, a cell filled with gas or a vacuum tube sensitive to light. The most used kind of phototube is the PMT due to its high sensitivity.

This PMT detects the *Cherenkov light* produced by the neutrino interaction. By sensing the pattern of light, a lot of information of the neutrino can be inferred, such as direction, energy and sometimes the flavor information of the incident neutrino.



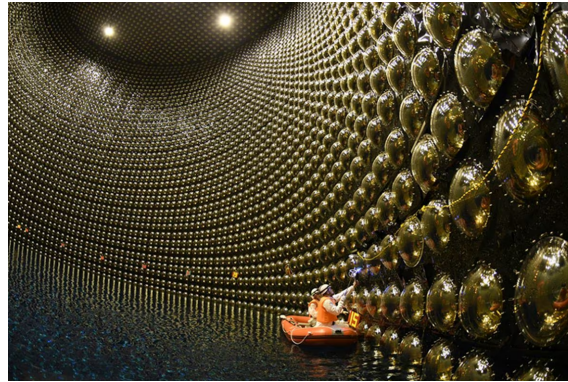
**Figure 3.5: How a PMT detects *Cherenkov light* phenomenon**

These detections are exceedingly rare due to the low probability of a neutrino interacting with matter. Therefore, the larger the water tank and the greater the number of PMTs, the more interactions can be detected within the same time-frame. Furthermore, the concept of *fiducial volume* must be taken into account, a recurring concept in particle physics experiments that involves considering only results from a specific region of the detector, as results outside of that zone may be confusing or of limited validity for several reasons. Increasing this valid detection zone is the key to increasing detection possibilities

### 3.3 HKK predecessor, Super-Kamiokande

The largest neutrino detector currently in operation is the Super-Kamiokande. "Kamiokande" is a fusion of several words: KAMIOKA Neutrino Detection Experiment. Situated beneath Mount Ikeno near the city of Hida in the Gifu Prefecture, Japan, Kamioka is the facility that oversees this detector. The SK consists of a 36.2m high and 33.3m diameter ultrapure water tank with PMT detectors for inner and outer tank detection, and the operation of this neutrino detector also relies on the *Cherenkov light* phenomenon captured by PMTs to collect data.

The main difference with its successor (HKK) lies in the size of the ultrapure water tank and the amount of PMTs. It is estimated that this increase in tank size and number of PMTs will make it possible to capture with HKK in 10 years an amount of data that would take 100 years to capture in SK.



**Figure 3.6: Interior of the Super-Kamiokande, predecessor of the Hyper-Kamiokande**

The detector has undergone up to four revisions for various reasons, such as cascade failures or the replacement of the 6000 PMTs, along with upgrades to electronics in the latest iteration, Super-Kamiokande IV. These phases have not led to an increase in the number of PMTs or their percentage of coverage, but rather to measures to protect the technology used.

| Phase                    |              | SK-I                   | SK-II                  | SK-III                 | SK-IV                  |
|--------------------------|--------------|------------------------|------------------------|------------------------|------------------------|
| Period                   | Start<br>End | 1996 Apr.<br>2001 Jul. | 2002 Oct.<br>2005 Oct. | 2006 Jul.<br>2008 Sep. | 2008 Sep.<br>2018 Jun. |
| Number of PMTs           | ID<br>OD     | 11146 (40%)            | 5182 (19%)<br>1885     | 11129 (40%)            | ID 11129 (40%)         |
| Anti-implosion container |              | No                     | Yes                    | Yes                    | Yes                    |
| OD segmentation          |              | No                     | No                     | Yes                    | Yes                    |
| Front-end electronics    |              | ATM (ID)<br>QTC (OD)   | ATM (ID)<br>QTC (OD)   | ATM (ID)<br>QTC (OD)   | QBEE                   |

**Table 3.1: SK Phases**

### 3.4 HKK objectives

The HKK experiment stands at the forefront of contemporary neutrino research, poised to unlock profound insights into the fundamental properties of these elusive particles. With its innovative design and enhanced capabilities, HKK ventures into uncharted territories of particle physics, aiming to shed light on mysteries ranging from neutrino oscillations to the enigmatic nature of dark matter.

In the realm of neutrino oscillation measurements, HKK endeavours to employ both accelerator and atmospheric neutrinos to unravel mysteries such as determining the mass hierarchy, investigating CP violation in the lepton sector, and precisely measuring oscillation parameters. Additionally, it aims to explore phenomena such as sterile neutrinos and potential violations of Lorentz invariance.

In solar neutrino measurements, HKK aims to address discrepancies observed between solar and reactor neutrino measurements, particularly focusing on the  $\theta_{12}$  sector. It intends to achieve this by studying day-night asymmetry in solar neutrino flux and exploring novel avenues such as monitoring solar fusion reactions and observing higher-energy neutrino flux.

HKK seeks to build upon the nucleon decay research legacy of Kamiokande and SK (Super-Kamiokande) by significantly enhancing limits on proton decays. It plans to utilize advanced PMT technology to improve performance, especially in detecting gamma rays from neutron capture, which is crucial for reducing neutrino backgrounds in proton decay searches.

In terms of supernova burst neutrinos, HKK aims to detect and analyze a substantial number of neutrinos from Galactic supernovae, allowing for detailed studies of these explosive events. It complements other experiments such as DUNE in its sensitivity to various types of supernova neutrinos.

For supernova relic neutrinos, HKK could contribute significantly by focusing on higher energy regions, complementing the efforts of SK. Introducing gadolinium enhances sensitivity by distinguishing neutrino interactions from background events.

In dark matter searches, HKK aims to improve upon SK's capabilities in detecting dark matter through neutrino signals, particularly from neutralino annihilation in regions of high dark matter density like the core of the Sun and the Galactic center. It also seeks to detect low-mass neutralinos, which are challenging to detect in direct-detection experiments.

In the pursuit of understanding the universe at its most fundamental level, the HKK experiment represents a beacon of scientific exploration. Through its multifaceted approach and collaborative efforts, HKK is poised to unravel some of the most profound mysteries of the cosmos, shaping our understanding of particle physics for generations to come.

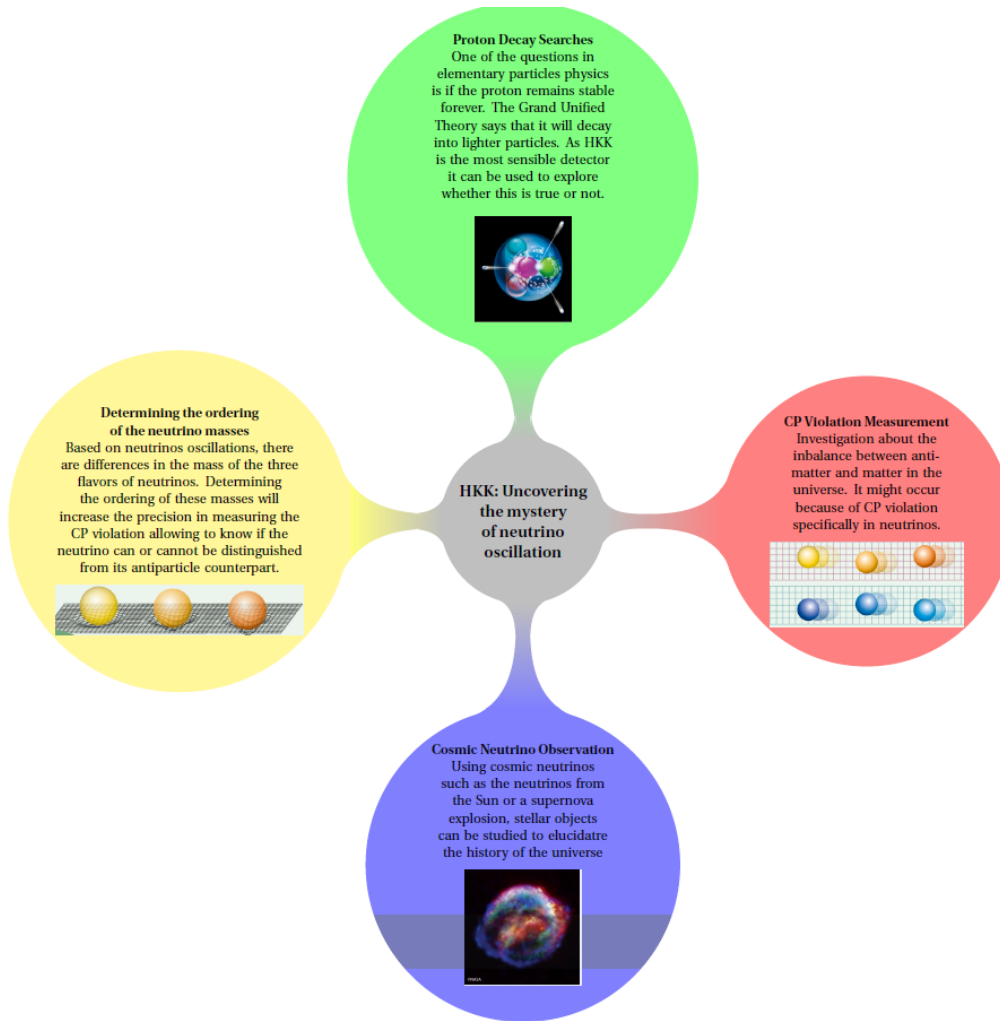
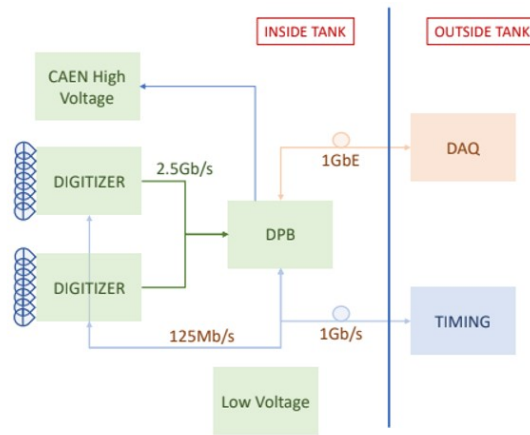


Figure 3.7: Hyper-Kamiokande neutrino oscillation investigation fields [5]

### 3.5 HKK project organization

As it has been mentioned previously, the HKK project involves research institutes from all over the world, which are in charge of different tasks within the project whether they are related to physics, electronics or any other relevant field. The project tasks have been divided in 7 different FD (Far Detector) groups [6]. Apart from the Far Detector groups, there are also the Near Detector and Beam Facility groups.

This TFG is developed inside of the FD4 group as our group in the I3M at the UPV belongs to this FD group. FD4 group tasks focus on developing electronics front-end inside the vessel for the inner sensors of the neutrino detector so as to be able to gather information from the sensors, transform it in order to allow the processing unit to process the data properly for sampling and sending the data from the inner sensors of the detector to a data-center.



**Figure 3.8: Diagram of communication between the different modules of the vessel**

As it can be seen in the previous figure, the electronics inside the vessel consists of high and low voltage modules used as power supplies for the rest of the components, digitizers and the DPB, and communicate outside the vessel with the DAQ. In the case of the power boards, the HV supplies the PMTs while the LV supplies the rest of the electronics inside the vessel.

The UPV is in charge of the development of the DPB of the ID focusing on collecting and transmitting information to the DAQ and on employing redundancy to maximise reliability. This is because the DPB, being the hub of the front-end electronics inside the vessel, is responsible for communicating all the modules and as the vessel is submerged in water, the electronics must be reliable enough to last for more than 10 years without needing to be repaired or replaced.

To achieve these goals, our group has to take care of designing the different parts of the DPB up to the final version, develop a software platform where we can develop the necessary software to communicate the DPB with the relevant modules and carry out the slow control tasks with reliability as a top priority in this project.

## **Part II**

# **Leveraged technology**



# Chapter 1

## DPB SoM election

In the realm of embedded systems development, System-on-Module (SoM) form factor has emerged as a transformative solution, particularly for academic institutions such as universities engaged in research and development projects. SoM refers to a compact, integrated circuit board that encapsulates essential components such as processors, memory, and I/O interfaces within a single package.

This chapter will therefore explain the definition of SoM and the advantages of choosing a SoM for the design of the DPB.

### 1.1 Understanding System-on-Module

System-on-Module (SoM) is a comprehensive computing platform condensed into a small, modular package. These modules typically include a microprocessor or System-on-Chip (SoC), memory components (both RAM and ROM), storage options, power management circuitry, and various peripheral interfaces. SoM modules are standardized in form factors such as COM Express, SMARC, and Qseven, facilitating easy integration into diverse hardware configurations.

#### 1.1.1 Advantages of SoM Technology:

##### 1. Cost Efficiency:

- **Reduced Development Costs:** One of the primary advantages of SoM technology for universities lies in its ability to lower development costs. Instead of investing resources in designing custom PCBs and integrating individual components, universities can procure pre-built SoM modules. While the upfront cost of SoM modules may seem higher compared to standalone chips, the overall development expenditure, including labour and prototyping, is significantly reduced.
- **Lower Total Cost of Ownership (TCO):** Despite initial investment differences, SoM technology often leads to a lower Total Cost of Ownership (TCO) over the project lifecycle. This is attributed to reduced development time, minimized risk of errors during hardware integration, and streamlined maintenance processes.



## 2. Time Efficiency:

- **Accelerated Development Cycles:** SoM modules expedite the development process by eliminating the need for designing intricate hardware configurations from scratch. This acceleration is particularly beneficial for universities engaged in time-sensitive research projects or academic initiatives with strict deadlines.
- **Rapid Prototyping:** SoM technology facilitates rapid prototyping, allowing researchers and students to quickly iterate through design concepts and experiment with various configurations. This agility fosters innovation and enables timely validation of hypotheses.

## 3. Risk Mitigation:

- **Enhanced Reliability:** SoM modules undergo rigorous testing and validation procedures during manufacturing, ensuring high levels of reliability and performance. By leveraging pre-tested and validated modules, universities mitigate the risk of hardware failures and associated costs, safeguarding project budgets and timelines.
- **Quality Assurance:** SoM vendors adhere to industry standards and quality control measures, providing universities with assurance regarding the integrity and functionality of the modules. This reliability is crucial for academic endeavours where consistency and reproducibility are paramount.

## 4. Resource Optimization:

- **Focused Resource Allocation:** By adopting SoM technology, universities can re-allocate resources previously dedicated to hardware design towards other aspects of research and development, such as software development, data analysis, and experimentation. This focused resource allocation enhances overall project efficiency and productivity.
- **Skills Utilization:** SoM technology reduces the dependency on specialized hardware design expertise within university research teams. Instead, academic resources can be channelled towards leveraging domain-specific knowledge and interdisciplinary collaboration, fostering a conducive environment for innovation and knowledge exchange [7], [8].

## Chapter 2

# DPB2 Prototype

The DPB2 we are currently working with is only a prototype of what will become the final DPB. It is a combination between a SoM and a carrier board. The choice of the SoM is motivated by the reasons mentioned in the previous chapter, since for us as a university institution it means a saving in engineering costs to be able to insert the SoM in a carrier board and not having to design the entire PCB from scratch and solder the processing unit to the board.

Regarding the carrier board, it was designed by Enclustra with the assistance and supervision of our group to obtain a design according to our needs and Enclustra itself was responsible for manufacturing the board. This is because Enclustra is a company dedicated to the manufacture of these plates. Therefore, delegating the design and manufacturing process of the board to Enclustra represented engineering costs savings and provided us with a guarantee against possible manufacturing defects. Among the components of the board we can highlight current and temperature sensors, 6 SFP ports, UART or JTAG ports among others.

Zynq UltraScale+ Architecture has been chosen as the SoM architecture due to its processing capacity, efficiency and the integration of PetaLinux, a reduced version of Linux for embedded systems which offers us a very complete embedded software development platform.

This chapter will discuss this selected SoM architecture and review the features it offers and their usefulness for the HKK project.

### 2.1 Zynq UltraScale+ Architecture

The Zynq UltraScale+ MPSoC platform offers designers the first truly all-programmable, heterogeneous, multiprocessing system-on-chip (SoC) device. Smart systems are increasing in complexity with applications in the automotive industry, large database deployments, and even space exploration, pushing the requirements of each new generation of SoC to its limits. Requirements for increased power control, real-time applications, intensive graphical capabilities, and processing power demand a platform with maximum flexibility. The Zynq UltraScale+ MPSoC platform provides leading-edge features that modern systems designers demand. [9]

Built on the next-generation 16 nm FinFET process node from Taiwan Semiconductor Manufacturing Company (TSMC), the Zynq UltraScale+ MPSoC contains a scalable 32 or 64-bit multi-processor CPU, dedicated hardened engines for real-time graphics and video processing, advanced

high-speed peripherals, and programmable logic. The platform delivers maximum scalability through either dual or quad-core APU devices, offloading of critical applications like graphics and video pipelining to dedicated processing blocks, and the ability to turn blocks on and off through efficient power domains and gated power islands. With a wide range of interconnect options, digital signal processing (DSP) blocks, and programmable logic choices, the Zynq UltraScale+ MPSoC has the flexibility to fit a diverse set of user application requirements.

To effectively harness the power of the Zynq UltraScale+ MPSoC, AMD has the SoC-strength tools with Vivado Design Suite, and PetaLinux, and can further accelerate development using the Vitis unified software platform for design abstraction. [10]

#### **Architecture's Essential Elements:**

- 64-bit Quadcore ARM Cortex-A53 Processors
- Dualcore ARM Cortex-R5 Real-Time Processors
- ARM Mali™-400MP Graphics Processor
- H.265/264 Video Codec Unit
- Advanced Dynamic Power Management Unit
- Configuration Security Unit
- DDR4/LPDDR4 Memory Interface Support
- 16FinFET+ Performance/Watt
- Vitis unified software platform for Design Abstraction
- Next-Generation AXI Interconnect
- Compatible with the Zynq 7000 SoCs, Software, and Ecosystem

### **2.1.1 Difference between PS and PL**

When we refer to a SoM, we must be clear about the two main blocks that conform it, PS and PL. The PS encompasses pre-designed and non-programmable components such as CPUs, memory controllers, and peripheral interfaces. Its main purpose is to handle computing tasks and manage the interface with the external world, facilitating communication with peripherals, memory access, and system-level operations.

Meanwhile, the PL offers a flexible and customizable hardware fabric that enables users to implement custom logic and tailor the functionality of the module to specific application requirements. Contrary to the PS, the PL consists of programmable hardware, typically in the form of FPGA or similar devices along with embedded transceivers and memory blocks, among others. This adaptable hardware fabric enables developers to create and implement bespoke hardware accelerators, interfaces, or custom processing pipelines, thus elevating the flexibility and performance capabilities of the SoM.

The PL functions in tandem with the PS, enabling hardware acceleration, real-time processing, and seamless integration of tailored peripherals to meet the distinct requirements of the specific application at hand.

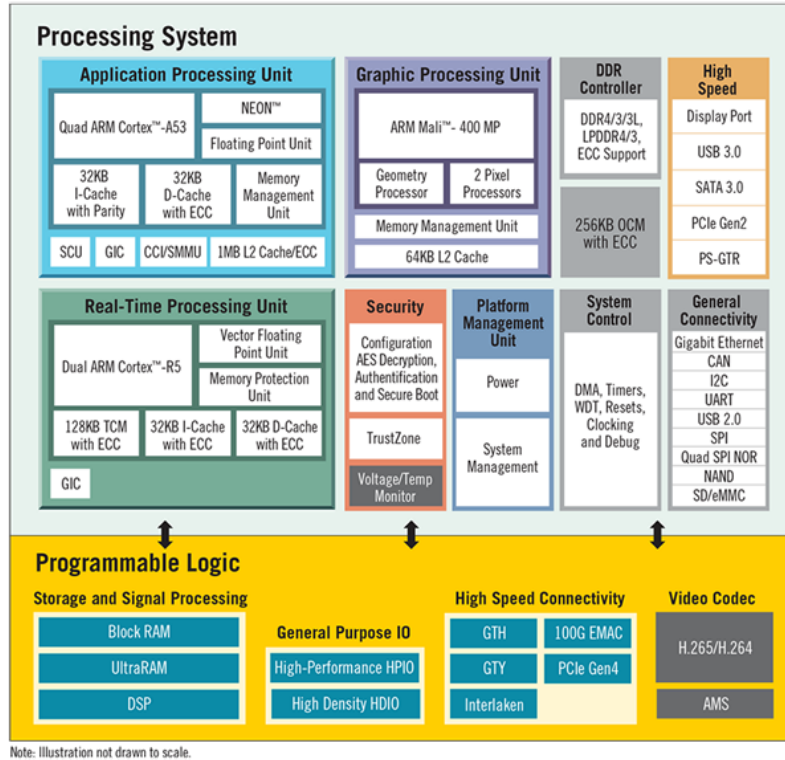


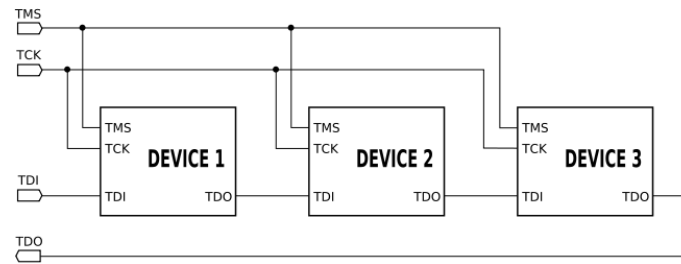
Figure 2.1: Zynq UltraScale+ PS and PL blocks

### 2.1.2 JTAG interface

JTAG is an industry standard used for verifying designs, testing PCB after being manufactured and programming FPGAs or similar devices by using boundary-scan technology. Signals are scanned into and out of the I/O cells of a device serially to control its inputs and test the outputs under various conditions.

The JTAG port consists of the following pins:

- **Test Data In (TDI)**
- **Test Data Out (TDO)**
- **Test Clock (TCK)**
- **Test Mode Select (TMS)**
- **Test Reset (TRST)** (optional)



**Figure 2.2: Daisy-chained JTAG**

The data transmission is serial since there is only one wire of transmission in each direction. Therefore, one bit of data is transmitted every rising clock edge and the direction depends on the mode select pin.

In the case of the Zynq UltraScale+ architecture, it enables a JTAG interface to the user that allows debugging features for software and PL configuration since it features PS and PL TAP and ARM debugging of the RPU and APU.

In particular, we have used the JTAG interface to load the PetaLinux image and the necessary boot files onto the eMMC using the Vivado development environment.

### 2.1.3 UART interface

The Zynq UltraScale+ architecture also count with an UART controller that functions as a full-duplex asynchronous receiver and transmitter, supporting a broad range of programmable baud rates and I/O signal formats. It offers capabilities for automatic parity generation and multi-master detection mode. The configuration and mode registers control UART operations, while the status, interrupt status, and modem status registers are used to monitor FIFO states, modem signals, and other controller functions.

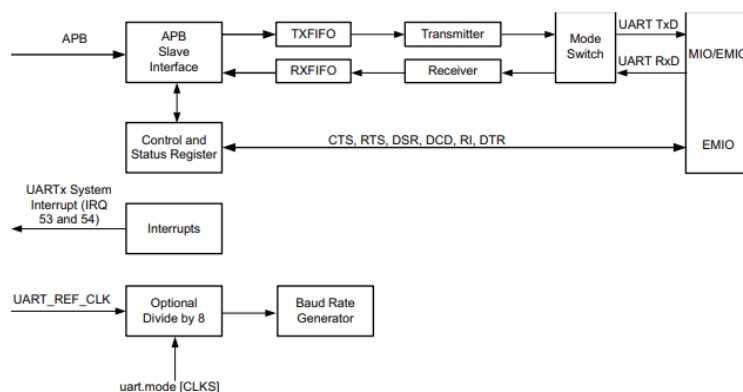
The controller comprises separate RX and TX data paths, each featuring a 64-byte FIFO. It handles data serialization and deserialization within the TX and RX FIFOs, and includes a mode switch to support various loop-back configurations for Rx/D and Tx/D signals. FIFO interrupt status bits allow for either polling or interrupt-driven handling. Data bytes are read and written using RX and TX data port registers.

In modem-like applications, the modem control module manages modem handshake signals and controls receiver and transmitter paths according to the handshaking protocol.

Key features of the UART controller include:

- Programmable baud rate generator
- Configurable receive and transmit FIFOs, with byte, two-byte, or four-byte APB access mechanisms
- Options for 6, 7, or 8 data bits
- Support for 1, 1.5, or 2 stop bits
- Parity options including odd, even, space, mark, or no parity

- Detection of parity, framing, and overflow errors
- Line break generation and detection
- Automatic echo, local loop-back, and remote loop-back channel modes
- Interrupt generation
- Modem control signals
- Dual clocks: advanced peripheral bus (APB) clocks up to 100 MHz and `uart_ref_clock` ranging from 1 MHz to 100 MHz



**Figure 2.3: UART Controller schema**

In our case, the UART interface has been used to communicate with the DPB and access its file system. Although the main communication with the DPB has been established via the Ethernet interface, the UART interface has been used mainly for debugging drivers and OS elements, since we can see kernel messages via UART, but not via Ethernet.

### 2.1.4 Ethernet interface

Zynq UltraScale+ counts with the gigabit Ethernet controller (GEM), which implements a 10/100/1000 Mb/s Ethernet MAC that is compatible with the IEEE Standard for Ethernet (IEEE Std 802.3-2008) and capable of operating in either half or full-duplex mode in 10/100 mode and full-duplex in 1000 mode. The processing system (PS) is equipped with four gigabit Ethernet controllers. Each controller can be configured independently. Each controller uses a reduced gigabit media independent interface (RGMII) v2.0.

Access to the programmable logic (PL) is through the EMIO which provides the gigabit media independent interface (GMII). Other Ethernet communications interfaces can be created in the PL using the GMII available on the EMIO interface. GEM supports the serial gigabit media-independent interface at 1000 Mb/s using the PS-GTR interface.

Registers are used to configure the features of the MAC, select different modes of operation, and enable and monitor network management statistics. The DMA controller connects to memory through the advanced eXtensible interface (AXI). It is attached to the controller's FIFO interface of

the MAC to provide a scatter-gather capability for packet data storage in an embedded processing system.

Each GEM controller provides management data input/output (MDIO) interfaces for PHY management.

Each gigabit Ethernet MAC controller has the following features:

- Compatibility with IEEE Standard 802.3-2008, supporting various transfer rates.
- Flexibility in operation modes: full/half duplex.
- Multiple I/O options for connectivity.
- MDIO interface for managing external PHY.
- Powerful DMA capabilities with scatter-gather support.
- APB slave interface for control register access.
- Comprehensive interrupt system for event notification.
- Automatic frame integrity checks and error handling.
- Configurable inter-packet gap and flow control.
- Advanced address checking and VLAN tagging capabilities.
- Support for loopback mode and checksum offloading.
- Recognition of IEEE Precision Time Protocol frames.
- Statistics counters for monitoring network performance.
- Jumbo frame support for efficient data transfer.
- Priority support for enhanced traffic management.

We have used the Ethernet interfaces to establish a connection to the DPB that is faster and more practical than JTAG and also allows us to implement interface redundancy techniques to maximise the reliability of the link.

### **2.1.5 I<sup>2</sup>C interface**

Furthermore, Zynq UltraScale+ provide us with an I<sup>2</sup>C controller which allows us to communicate with the sensors and SFPs installed in the DPB via 2 different I<sup>2</sup>C buses. The I<sup>2</sup>C controllers are versatile, capable of operating as either a master or a slave within a multi-master setup, with a clock frequency range of up to 400 kb/s. They support multi-master mode for both 7-bit and extended 10-bit addressing formats.

In master mode, transfers are initiated solely by the processor writing the slave address into the I<sup>2</sup>C address register. The processor is then alerted to any incoming data via either a data interrupt or a transfer complete interrupt. If the hold bit is activated, the I<sup>2</sup>C interface keeps the clock signal

(SCL) Low after transmitting data, facilitating smooth operation for slower processors. Master configuration allows for the use of both standard and extended addressing modes, with the latter exclusive to master mode [11].

In slave monitor mode, the I<sup>2</sup>C interface acts as a master, persistently attempting a transfer to a designated slave until either an acknowledgement (ACK) is received or a timeout occurs.

The controller also supports repeated start functionality, wherein the master can generate a subsequent start condition following the initial one, typically followed by the slave's I<sup>2</sup>C address.

A shared feature between master and slave modes is the timeout mechanism, indicated by the TO interrupt flag. If, at any stage, the SCL clock signal remains Low for a duration exceeding that specified in the timeout register, a TO interrupt is triggered to prevent operational stalls.

There are two I<sup>2</sup>C controllers located in the LPD IOP section of the PS. They adhere to I<sup>2</sup>C bus specification version 2 and feature a 16-byte FIFO buffer. Key features include programmable normal and fast bus data rates, support for multi-master configurations, and versatile operation modes.

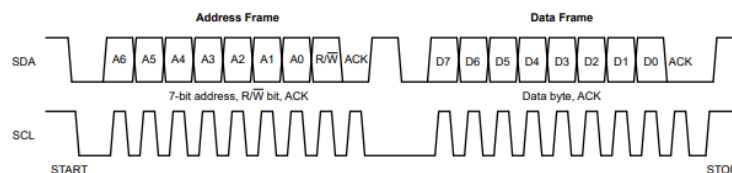
In master mode, the controllers facilitate read and write transfers, with support for both seven and 10-bit addressing formats. They incorporate clock stretching functionality to accommodate slow processor operation, preventing stalls with a TO interrupt bit. Additionally, they offer repeated start capability and slave monitor mode.

When operating in slave mode, the controllers can transmit and receive data and feature fully programmable slave response addresses. They include a HOLD bit to mitigate overflow conditions and utilize clock stretching to manage communication delays when data isn't readily available.

Furthermore, the controllers can be polled for status by software or function as interrupt-driven devices, with programmable interrupt generation capabilities.

To achieve communication between the different components on the board and the terminal, the I<sup>2</sup>C protocol is used, a communication protocol based on a Master-Slave system where the communication bus is divided into 2 lines, SCL for the clock and SDA for the data, which are connected to a pull-up resistor each, so the default level is high level.

The operation of this protocol consists of the start of the transmission by the Master which jointly indicates the address of the slave to which it is directed with an address of 7 bits, even though we have sensors that have an address of 6 bits plus a reserved bit, which can be configured to differentiate each slave in a physical way, in addition, it is indicated with a bit if the operation to be carried out is reading or writing. The data transmission is guided by the clock line and the data is transmitted in byte size, transmitting from MSB to LSB.



**Figure 2.4: Addressing and data frames I<sup>2</sup>C**

For the write operation on the slave, once communication has been established, the register to be



written to and the data to be written must be indicated. The master is responsible for receiving the corresponding ACK and NACK during communication and the end of communication sequence.

The read operation follows a similar process to the write operation, indicating the register to be read and the master is in charge of sending the corresponding ACKs and NACKs during the communication and at the end of communication sequence.

In our case the communication process will be based on the functions provided by the Linux libraries that allow us to open/close the communication and read/write registers simply by calling defined functions and indicating the necessary arguments. In addition, these functions allow us to operate with vectors in order to read or write consecutive data with a single function.

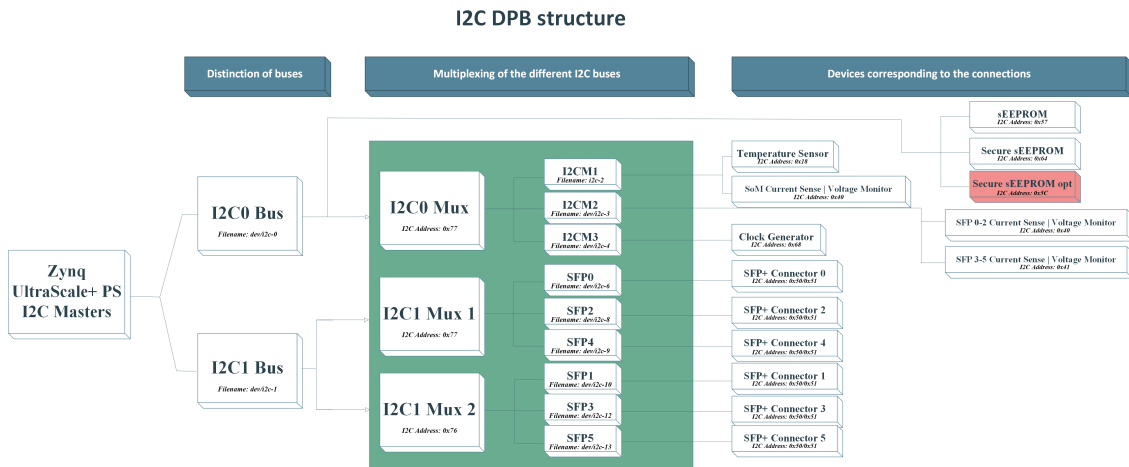
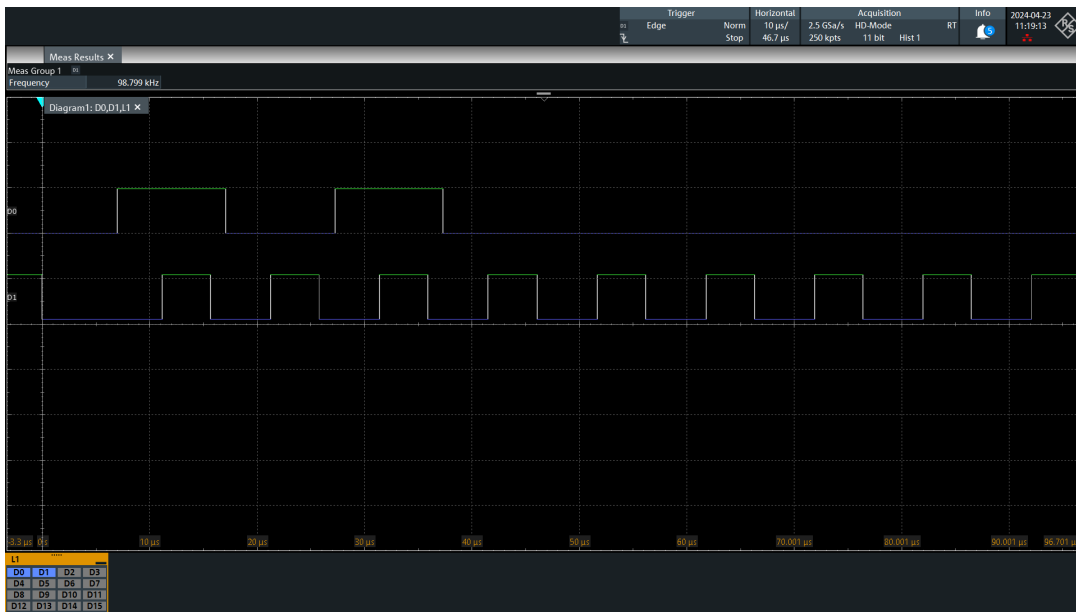


Figure 2.5: Structure of the I<sup>2</sup>C of our DPB

In the previous block diagram you can see how the I<sup>2</sup>C buses of our DPB are structured, the corresponding filename of each of the I<sup>2</sup>C bus outputs designated by the multiplexers and the slave addresses of each module with which we intend to communicate.

As can be seen, the current sensors, the SFP connectors and the temperature sensor that we intend to use all use the I<sup>2</sup>C protocol to communicate. However, the temperature sensor and the current sensors use 16-bit registers, while the SFPs use 1-byte sized registers. The I<sup>2</sup>C protocol carries byte-sized frames, so in the case of 16-bit registers it involves performing 2 consecutive operations (either read or write) on the same register address, whereas for 8-bit registers it will involve a single operation per register address.

Moreover, as it can be seen in our I<sup>2</sup>C devices datasheets, reading and writing operations are performed differently since for the read operation you have to write the address of the register you want to read in the register pointer and then read the contents in separate operations, whereas for the write operation you have to indicate the address of the register you want to write to and the data you want to write consecutively. In our case, the Linux I<sup>2</sup>C driver will make it much easier to perform any I<sup>2</sup>C operations.



**Figure 2.6: SDA and SCL I<sup>2</sup>C communication signals**

As can be seen in the figure 2.6, SDA and SCL signals generated by the Linux driver to achieve I<sup>2</sup>C communication are as expected and the clock signal works in the standard mode of 100kHz and it has been successfully measured using an oscilloscope that the I<sup>2</sup>C transmission speed we are working with is 100kHz.

### 2.1.6 GPIO interface

The Zynq UltraScale+ architecture incorporates the General Purpose Input/Output (GPIO) controller, which is a collection of input/output signals available for software applications. The GPIO comprises the MIO with 78 pins and the Extended Multiplexed Input/Output Interface (EMIO) with 288 signals, divided into 96 inputs from the Programmable Logic (PL) and 192 outputs to the PL. The GPIO is organized into six banks of registers that group related interface signals.

Each GPIO channel is independently and dynamically programmed as input, output, or interrupt sensing. Software applications can read all GPIO values within a bank using a single load instruction or write data to one or more GPIOs using a single store instruction. The GPIO control and status registers for the Zynq UltraScale+ architecture are memory-mapped beginning at base address 0xFF0A\_0000 and are protected by the XPPU [12].

Key features of the GPIO peripheral are summarized as follows:

- 78 GPIO interfaces to the device pins, routed through the MIO multiplexer, with programmable I/O drive strength, slew rate, and 3-state control.
- 96 GPIO interfaces to the PL (four allocated by software to reset PL logic), routed through the EMIO interface, providing data inputs, data outputs, and output enables.
- I/O interface organized into six banks (3 MIO and 3 EMIO).

- Interface control registers grouped by bank {0:5}.
- Input values read using the six DATA\_RO\_x registers.
- Two types of data ports for writing: full bank write using the DATA\_x registers, and split bank maskable write using the MASK\_DATA\_x\_LWS, MWS register pairs.
- The function of each GPIO can be dynamically programmed on an individual or group basis.
- Enable, bit or bank data write, output enable, and direction controls.
- Programmable interrupts on an individual GPIO basis, with status read of raw and masked interrupt, and selectable sensitivity (Level-sensitive: High or Low, or edge-sensitive: positive, negative, or both).

In our case, the GPIO pins have been used for various functions, such as enabling Aurora links with the digitizers or SFPs transmission, or even controlling SFP status. In table 2.1, we can see our DPB GPIO pin distribution.

| DIR    | PIN#    | PIN NAME            | PIN NAME            | PIN NAME            | PIN NAME            |
|--------|---------|---------------------|---------------------|---------------------|---------------------|
| OUTPUT | (65:64) |                     |                     | DMA TIMEOUT ENABLE  | DMA FORCE TLAST     |
| OUTPUT | (63:60) | AURORA_RST_DIG1_SEC | AURORA_RST_DIG1_PRI | AURORA_RST_DIG0_SEC | AURORA_RST_DIG0_PRI |
| OUTPUT | (59:56) | DMA_PAUSE           | DMA_BUF_SIZE        | DMA_ENABLE          | EN_HVLV_DRV         |
| OUTPUT | (55:52) | EN_CPU_HV_1         | EN_CPU_HV_0         | EN_CPU_LV_1         | EN_CPU_LV_0         |
| OUTPUT | (51:48) | TIMING_RST          | XVC_DISABLE         | DMA_SOURCE          | GLOBAL_AURORA_RST   |
| INPUT  | (47:44) | AURORA AFIFO FULL   | DIG_STREAM_ERR      | PLL_LOL_n           | PLL_INTR_n          |
| INPUT  | (43:40) | LINK_UP(DIG1,SEC)   | LINK_UP(DIG1,PRI)   | LINK_UP(DIG0,SEC)   | LINK_UP(DIG0,PRI)   |
| INPUT  | (39:36) | DIG1_PWR_GOOD       | DIG0_PWR_GOOD       | DIG1_FPGA_DONE      | DIG0_FPGA_DONE      |
| INPUT  | (35:32) | SFP_PWR_GOOD(5)     | SFP_RX_LOS(5)       | SFP_MOD_ABS(5)      | SFP_TX_FAULT(5)     |
| INPUT  | (31:28) | SFP_PWR_GOOD(4)     | SFP_RX_LOS(4)       | SFP_MOD_ABS(4)      | SFP_TX_FAULT(4)     |
| INPUT  | (27:24) | SFP_PWR_GOOD(3)     | SFP_RX_LOS(3)       | SFP_MOD_ABS(3)      | SFP_TX_FAULT(3)     |
| INPUT  | (23:20) | SFP_PWR_GOOD(2)     | SFP_RX_LOS(2)       | SFP_MOD_ABS(2)      | SFP_TX_FAULT(2)     |
| INPUT  | (19:16) | SFP_PWR_GOOD(1)     | SFP_RX_LOS(1)       | SFP_MOD_ABS(1)      | SFP_TX_FAULT(1)     |
| INPUT  | (15:12) | SFP_PWR_GOOD(0)     | SFP_RX_LOS(0)       | SFP_MOD_ABS(0)      | SFP_TX_FAULT(0)     |
| OUTPUT | (11: 8) | TX_DIS(5)           | TX_DIS(4)           | TX_DIS(3)           | TX_DIS(2)           |
| OUTPUT | ( 7: 4) | TX_DIS(1)           | TX_DIS(0)           | PWR_EN(5)           | PWR_EN(4)           |
| OUTPUT | ( 3: 0) | PWR_EN(3)           | PWR_EN(2)           | PWR_EN(1)           | PWR_EN(0)           |

**Table 2.1: DPB GPIO pin distribution**

### 2.1.7 RS-485 communication protocol

RS-485 is an established standard initially introduced in 1983. It outlines the electrical characteristics of drivers and receivers for use in serial communication systems. The signaling is balanced, supporting multi-point systems. The standard is jointly managed by the Telecommunications Industry Association and the Electronic Industries Alliance. RS-485 facilitates effective digital communication networks over extended distances and in electrically noisy environments. It enables the connection of multiple receivers to a linear, multi-drop bus, making it valuable in industrial control systems and similar applications.

RS-485 facilitates cost-effective local networks and multi-drop communication links, utilizing the same differential signaling over twisted pair as RS-422.

We use two RS-485 drivers which are connected to the UARTLite cores of the HV and LV boards instantiated in the PL to be able to communicate with these boards.

## Chapter 3

# PetaLinux embedded OS

In this chapter, we intend to give an introduction to the OS chosen, PetaLinux, as the platform on which the DPB works, its characteristics, and the advantages of its use.

### 3.1 Unix/Linux environment and OS election

It is well known that nowadays there are plenty of OS options available. Nevertheless, Linux has established itself as the ideal choice for a large part of the worldwide server market and is also extending its reach to personal computers and embedded systems, the latter including our DPB.

Its success is mostly due to its open-source nature, which allows software developers to fully customize and optimize it for specific applications. At a time when it is essential to reduce resource consumption as much as possible, being able to configure the technology you have available to extract its maximum performance for your particular application is ideal [13].

In addition, thanks to being an open source OS, this has led to the creation of a huge community that is dedicated to developing and supporting software such as drivers and libraries to provide a higher level of abstraction over the control of peripherals or any other component achieved by the Linux kernel in the form of functions [14]. This can be an advantage when applied to generic applications since the driver or library may not be suitable for a specific application and may require custom adaptation.

However, Linux offers a huge variety of different distributions and not all are suitable for embedded systems. Therefore, Xilinx offers two options for running Linux on its Zynq UltraScale+ boards: PetaLinux and Ubuntu Desktop. For this project, PetaLinux has been chosen for the following reasons:

- **Customizability:** PetaLinux provides a comprehensive SDK that allows for tailoring the operating system to specific hardware and software requirements. This is crucial in the DPB as both hardware and software must serve a very specific purpose, namely reliability. With PetaLinux, it's possible to customize the operating system to meet the specific demands of the Zynq MPSoC, optimizing performance and minimizing resource usage.
- **Size:** PetaLinux enables the creation of a highly optimized, minimalist system image, ideal

for embedded systems. This is significant in the DPB, where one strategy to enhance reliability is to have redundant booting images. The smaller the image, the more copies can be placed in memory. In contrast, Ubuntu Desktop is designed for desktop computers and typically includes a plethora of applications and features unnecessary for an embedded system, resulting in a larger image size.

- **Flexibility:** PetaLinux offers a high level of flexibility in terms of package selection, configuration, and even customization of the kernel's source code, allowing for on-the-fly patches before compiling the boot image. This makes it well-suited for use in a system where hardware and software are designed together, ensuring that any device-specific bugs, such as those related to SFP modules or sensors, can be patched without much difficulty. Ubuntu Desktop, on the other hand, is a more general-purpose operating system that lacks the same level of flexibility and customization, as the image provided by Xilinx is already pre-compiled.

### 3.1.1 Used Linux libraries, drivers and applications

As mentioned in the previous section, Linux libraries and drivers are outstanding tools that can considerably ease the task of software development on a Linux-based OS.

During the course of the tasks of this TFG, several drivers and libraries have been used, in addition to those specific to the C programming language, and the following is intended to name and explain the most significant drivers and libraries used in order to understand their usefulness in the developed software.

- **Pthreads** (pthread(7)): It allows us to segment the execution of the code into different threads that run in parallel and thus be able to divide and perform different tasks independently. These threads are known as POSIX threads, an API designed by the IEEE standards for Unix systems. The main difference between these threads is with those in Windows, which we find in the API designed by Microsoft and in their behaviour, as they differ in how they handle signals or synchronize at the kernel level.  
By including a lightweight application called a timer to define the period of the thread and send it to sleep during this period, we achieve the periodical execution threads that have been used to perform the constant tasks of slow control, monitoring and communication of the DPB.
- **I<sup>2</sup>C driver** (I<sup>2</sup>C application source): The I<sup>2</sup>C application abstracts and uses the Linux I<sup>2</sup>C driver itself, reducing it to different functions so that allow us to perform the necessary operations using the I<sup>2</sup>C interface, either initiating communication with the device, reading from a register or writing to it. This is a great help when dealing with the I<sup>2</sup>C devices available in the DPB since we only have to apply the functions accurately, indicating the necessary parameters and following the instructions in the datasheet of each component, but we do not have to worry about the communication process internally since this will be taken care of by the driver.
- **Xilinx AMS** (Xilinx AMS driver): The Xilinx AMS driver has been included in the Linux kernel so as to be able to access all the relevant data gathered by the AMS SYSMON. The driver provides us access to this information through the sysfs file system, the information

is encoded in ADC code and its conversion to the corresponding magnitude is done by following the guidelines of the Xilinx documentation.

In addition, the driver provides us an alarm system depending on value thresholds. Since the driver treats alarms as IIO device events, we can use the generic Linux kernel tool IIO Event Monitor, which will be explained in more detail in the next section, as a background process that will capture alarms and report them to the main application.

- **IIO Event Monitor**(IIO Event Monitor application source): As mentioned previously, this tool is a generic application for detecting IIO device events and reporting them. Consequently, it will run in the background to detect Xilinx AMS alarms and report them to our main application so it can process them.
- **Shared Memory** (`sys_shm.h(0p)`): In order to communicate our main application with the IIO Event Monitor which runs, it was necessary to use a library that would allow us to reserve and manage shared memory segments to be able to move data from one application to another.
- **Semaphores** (`semaphore.h(0p)`): When working with different threads and shared resources, synchronization is crucial to avoid race conditions and, thus, a malfunction of the application. Wherefore, the semaphores function is to synchronize the application execution to ensure its correct operation.
- **JSON-C**(`json-c_json.h(0p)`): It has been decided that the format in which the communication will be made will be JSON, so this release will allow the information collected to be encapsulated in a JSON string and also to extract the messages received from the DAQ in the same format to process them.
- **ZeroMQ**(`zmq(7)`): The communication between the DPB and the DAQ will be through different types of sockets which each will transmit or receive a specific type of information through different transport layers such as TCP or multicast. ZeroMQ library will allow us to create the necessary sockets and manage them to send or receive the desired data.
- **Regular expressions**(`regex(7)`): Since it has been necessary to work with paths and commands defined in string, and in the C programming language it is not trivial to handle strings as it can be in other languages, the use of regular expressions to be able to find specific files or expressions has been very useful in the development of the application.

Apart from these driver libraries, basic C libraries have been used to handle strings, errors, directories, and mathematical operations. In addition, most of the global constants and variables have been defined in a separate header file, which has then been included in the application to visualize the code in a cleaner way

## 3.2 Sysfs file system

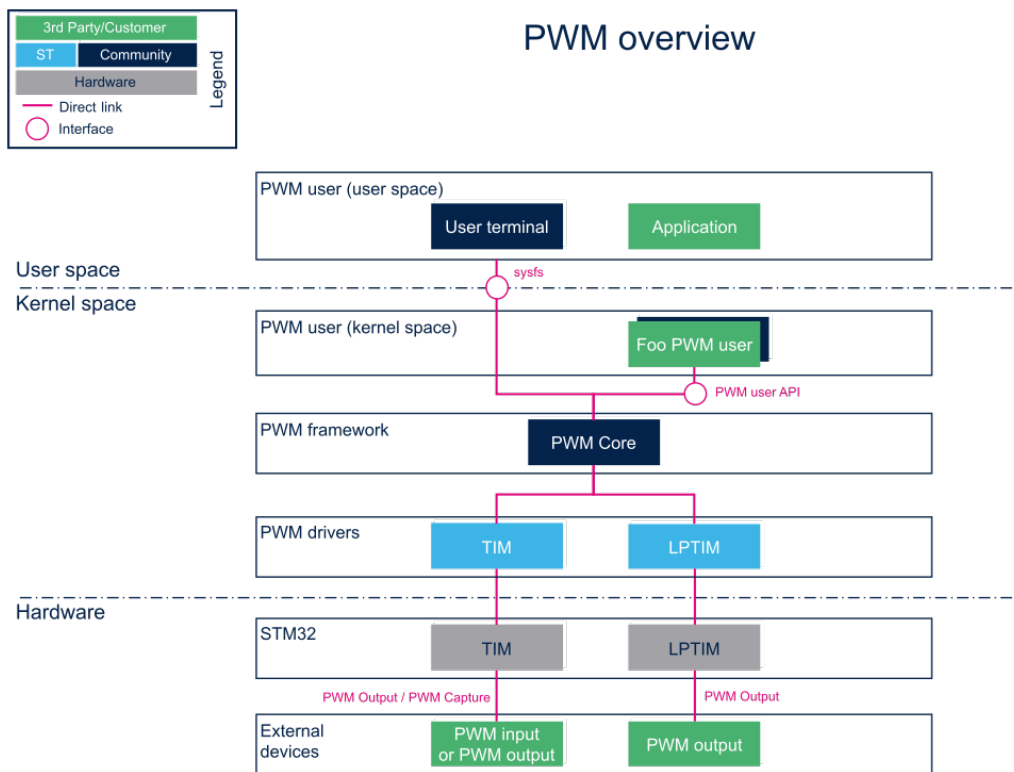
Sysfs is a mechanism within the kernel that serves as a representation of kernel objects, their attributes, and how they relate to each other. It offers both a kernel programming interface for exposing these elements via sysfs and a user interface to visualize and manipulate them, aligning

with the kernel objects they represent. In this system, kernel objects are represented as directories, their attributes as regular files, and their relationships as symbolic links [15].

As a fundamental part of kernel infrastructure, sysfs provides a relatively straightforward interface for basic tasks. While its code tends to be straightforward and descriptions clear, it can become abstract and challenging to navigate due to its core nature. To address this, this paper takes a gradual approach to explaining sysfs, beginning with a brief history, followed by essential information on mounting and accessing sysfs. It then delves into the directory structure and layout of subsystems within sysfs, providing users with an understanding of the organization and content exported through sysfs.

Sysfs serves as a conduit of information between the kernel and user space, providing ample opportunities for user space applications to utilize this data. Some existing applications include managing I/O Scheduler parameters and utilizing the udev program.

Therefore, the sysfs file system provides us with the information collected by the Xilinx AMS driver in the form of directories and files.



**Figure 3.1: Role of sysfs in user-hardware communication**

As can be seen in the figure 3.1 in this example of PWM, sysfs serves as a connection between user space and kernel space. In our case, communication between kernel space and hardware is abstracted by the employed driver.

## Chapter 4

# Development environments used

In this chapter, the used software and hardware development environments will be presented and their use and importance in the development of the tasks of this TFG will be explained.

### 4.1 Vivado Design Suite

The AMD Vivado Design Suite is aimed at enhancing productivity in designing, integrating, and implementing systems using various AMD devices such as UltraScale+, 7 series, Versal, Zynq UltraScale+ MPSoCs, and Zynq 7000 SoCs. These devices feature advanced technologies like stacked silicon interconnect, high-speed I/O interfaces up to 28 gigabytes, hardened microprocessors, analog mixed signal, and more.

The suite replaces the ISE Design Suite and integrates all its point tools, offering analytical optimization for multiple design metrics including timing, congestion, wire length, utilization, and power. Design analysis capabilities are available at each stage, enabling modifications early in the design process to reduce iterations and accelerate productivity.

All Vivado Design Suite tools feature a native Tcl interface, accessible through the GUI-based commands in the Vivado IDE or via Tcl commands entered in the Tcl Console. This allows for comprehensive design synthesis and implementation flows, including reporting and configuration.

The Vivado IDE provides both new users and advanced users with an intuitive interface and powerful tools, allowing analysis and constraint assignment throughout the design process. Designs can be opened in memory at various stages, facilitating visualization, interaction, and modification of constraints, logic, and device configurations. Design checkpoints capture snapshots of the design at different stages for analysis and comparison [16].

The tools provided by Vivado have allowed us to initially flash the PetaLinux image via JTAG, with the necessary files.



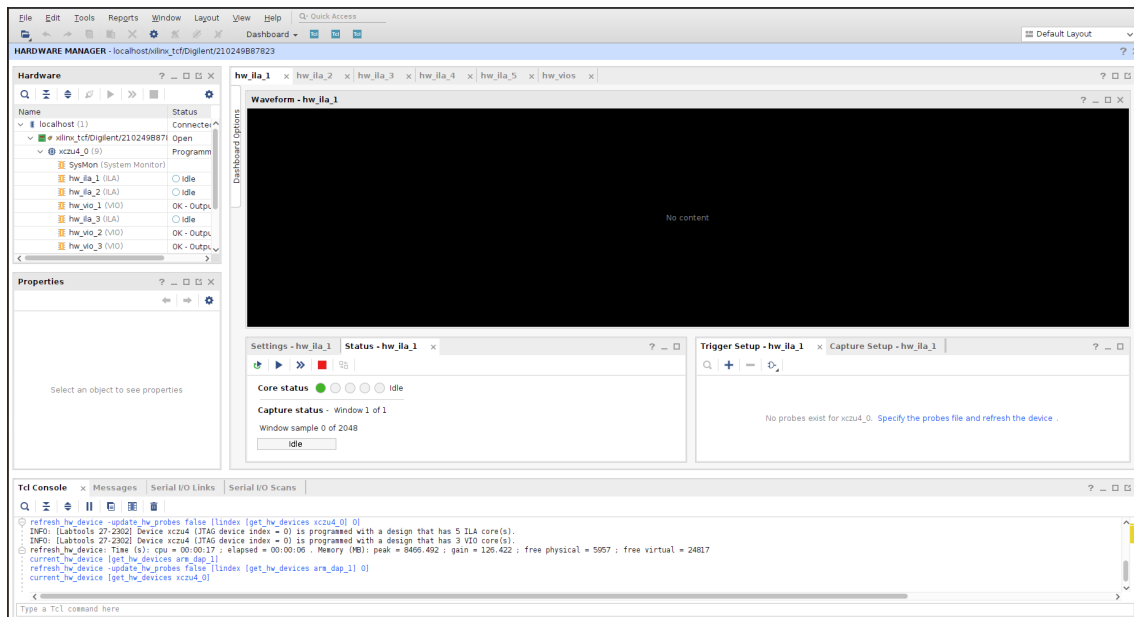


Figure 4.1: Vivado GUI

## 4.2 Vitis IDE

The Vitis unified software platform is a new tool that brings together all aspects of Xilinx software development into a unified environment. The Vitis software platform supports both the Vitis embedded software development flow, for Xilinx Software Development Kit (SDK) users looking to transition to the next generation technology, and the Vitis application acceleration development flow, for software developers looking to utilise the latest in Xilinx FPGA-based software acceleration. This document discusses the embedded software development flow and the use of the Vitis core development kit [17].

The Vitis integrated design environment (IDE) is part of the Vitis unified software platform. The Vitis IDE is designed to be used for the development of embedded software applications targeted towards Xilinx embedded processors. The Vitis IDE works with hardware designs created with Vivado Design Suite. The Vitis IDE is based on the Eclipse open-source standard. The features for software developers include:

- Feature-rich C/C++ code editor and compilation environment
- Project management
- Application build configuration and automatic Makefile generation
- Error navigation
- Integrated environment for seamless debugging and profiling of embedded targets
- Source code version control
- System-level performance analysis

- Focused special tools to configure FPGA
- Bootable image creation
- Flash programming
- Script-based command-line tool

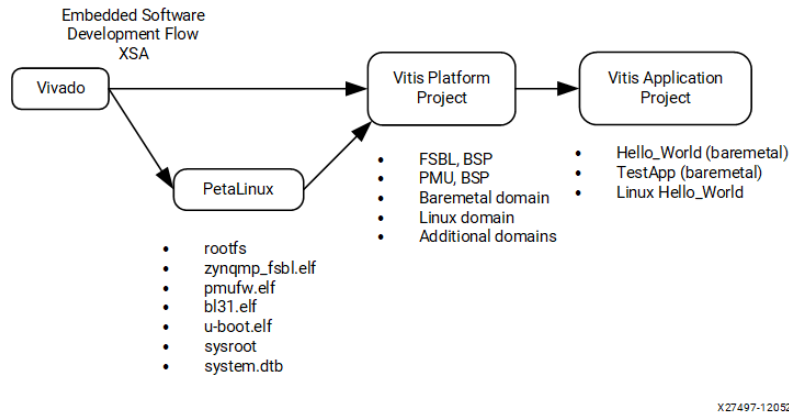


Figure 4.2: Vitis Embedded Software Development Flow [18]

As it can be seen in the previous figure 4.2, after loading the necessary files into the DPB, Vitis IDE is used in order to create the platform project on which our software will be developed and then the application project is created where the desired functions will be programmed in the programming language C.

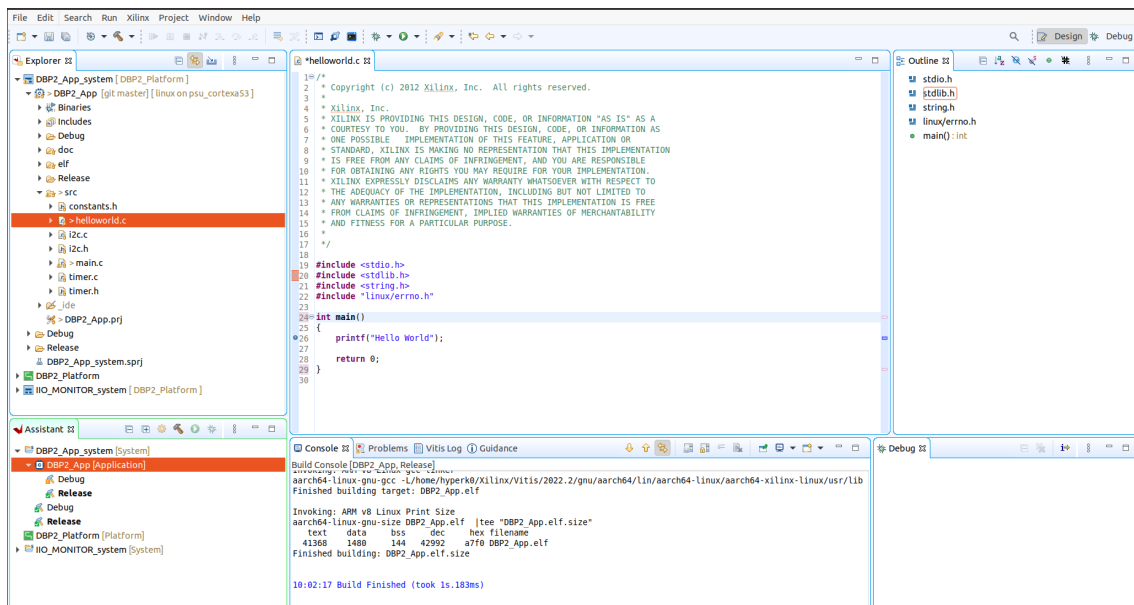


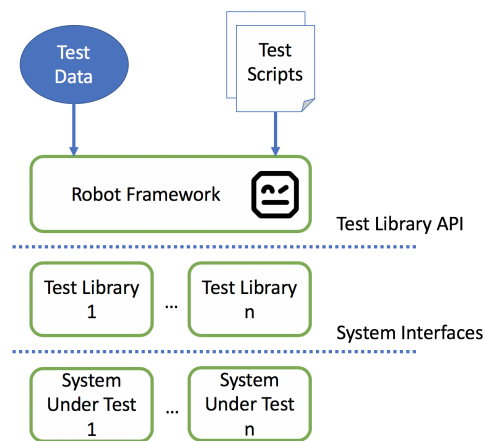
Figure 4.3: Vitis GUI

### 4.3 Robot Framework

Robot Framework is an open-source automation framework based on Python which can be used for test automation and robotic process automation. It is a very powerful tool since it can be integrated virtually with almost any other tool to develop flexible automation solutions.

Furthermore, the environment proposed by the robot framework is very intuitive and easy to work with once the test libraries have been defined since it is based on a human language rather than a programming language and has an infinite number of libraries to extend its functionalities if necessary [19].

As a Python-based development environment, you can work with the Robot Framework in any text or code editor such as Visual Studio Code. However, Visual Studio Code offers an extension that seamlessly integrates Robot Framework and even allows running tests from the editor itself.



**Figure 4.4: Robot framework workflow [20]**

As can be seen in the previous figure, Robot framework employs test library APIs to access the defined test libraries, which comprise keyword-driven and data-driven cases to facilitate case readability and implementation. Then this library has to communicate with the DUT to implement the cases, and after running the corresponding tests the Robot framework provides in an HTML file the report of the tests performed.

```

Check Initialization
  Check ZMQ Initialization
  Check IIO Monitor
  Check GPIO Base Address
  Check I2C Devices Initialization
  Check SFP Presence

```

**Figure 4.5: Example of keyword driven test**

## **Part III**

# **DPB sensors capabilities descriptions for *slow control* tasks**



# Chapter 1

## I<sup>2</sup>C devices

Regarding the sensor units available in our DPB we find, as previously mentioned in the I<sup>2</sup>C section, a temperature sensor (MCP9844), several current and voltage sensors (INA3221) for the SFP transceivers and the SoM and the SFP transceivers themselves, which provide us with very relevant information about their operating status and that we should keep track of.

### 1.1 INA3221 Current sensor

The current sensors installed in the DPB provide us with the possibility of monitoring up to 3 different channels from the same sensor. In addition, it allows us to measure the bus voltage with respect to GND (*Bus Voltage*) or the voltage difference between the IN+ and IN- terminals of each channel (*Shunt Voltage*). In our case, a resistive element with a value of 0.05  $\Omega$  is placed between IN+ and IN-, which is useful for obtaining both the current and the power consumed in each channel [21].

It should be noted that this sensor allows us to configure alerts and warnings for voltage values obtained in *Shunt Voltage* measurement mode to detect if the voltage difference between terminals of our resistor exceeds or does not reach certain values and to be able to act accordingly. We also have an alert if in *Bus Voltage* measurement mode, which informs us if all the channels being measured have a voltage higher than that marked by the limits or if any of the channels has a voltage lower than the lower limit. We are also provided with the option to obtain the sum of the *Shunt Voltage* of all channels and set a limit to configure an alert. All named alerts and warnings are collected in the *Mask/Enable* register where the sum of the *Shunt Voltage*, warnings and alarms can also be enabled or disabled.

In the following table you can see the most influential registers for our application, a short description of these registers, their default value and the type of register it is, whether it is read-only or read-write.

| <b>POINTER ADDRESS (Hex)</b> | <b>REGISTER NAME</b>     | <b>DETAILS</b>   | <b>BINARY (Power-On Reset)</b> | <b>HEX (Power-On Reset)</b> | <b>TYPE</b> |
|------------------------------|--------------------------|--|--------------------------------|-----------------------------|-------------|
| 0                            | Configura-<br>tion       | All-register reset, shunt and bus voltage ADC conversion times and operating mode.                               | 01110001<br>00100111           | 7127                        | R/W         |
| 1                            | Channel-1 Shunt Voltage  | Averaged shunt voltage value.  | 00000000<br>00000000           | 0000                        | R           |
| 2                            | Channel-1 Bus Voltage    | Averaged bus voltage value.  | 00000000<br>00000000           | 0000                        | R           |
| 3                            | Channel-2 Shunt Voltage  | Averaged shunt voltage value.  | 00000000<br>00000000           | 0000                        | R           |
| 4                            | Channel-2 Bus Voltage    | Averaged bus voltage value.  | 00000000<br>00000000           | 0000                        | R           |
| 5                            | Channel-3 Shunt Voltage  | Averaged shunt voltage value.  | 00000000<br>00000000           | 0000                        | R           |
| 6                            | Channel-3 Bus Voltage    | Averaged bus voltage value.  | 00000000<br>00000000           | 0000                        | R           |
| 7                            | Channel-1 Critical Alert | Contains limit value to compare each conversion value to determine if the corresponding limit has been exceeded. | 01111111<br>11111000           | 7FF8                        | R/W         |

Continued on next page

**Table 1.1: INA3221 Current Sensor Registers (Continued)**

| <b>POINTER ADDRESS (Hex)</b> | <b>REGISTER NAME</b>     | <b>DETAILS</b>   | <b>BINARY (Power-On Reset)</b> | <b>HEX (Power-On Reset)</b> | <b>TYPE</b> |
|------------------------------|--------------------------|--|--------------------------------|-----------------------------|-------------|
| 8                            | Channel-1 Warning Alert  | Contains limit value to compare to averaged measurement to determine if the corresponding limit has been exceeded. | 01111111<br>11111000           | 7FF8                        | R/W         |
| 9                            | Channel-2 Critical Alert | Contains limit value to compare each conversion value to determine if the corresponding limit has been exceeded.   | 01111111<br>11111000           | 7FF8                        | R/W         |
| A                            | Channel-2 Warning Alert  | Contains limit value to compare to averaged measurement to determine if the corresponding limit has been exceeded. | 01111111<br>11111000           | 7FF8                        | R/W         |

Continued on next page



**Table 1.1: INA3221 Current Sensor Registers (Continued)**

| <b>POINTER ADDRESS (Hex)</b> | <b>REGISTER NAME</b>     | <b>DETAILS</b>   | <b>BINARY (Power-On Reset)</b> | <b>HEX (Power-On Reset)</b> | <b>TYPE</b> |
|------------------------------|--------------------------|--|--------------------------------|-----------------------------|-------------|
| B                            | Channel-3 Critical Alert | Contains limit value to compare each conversion value to determine if the corresponding limit has been exceeded.             | 01111111<br>11111000           | 7FF8                        | R/W         |
| C                            | Channel-3 Warning Alert  | Contains limit value to compare to averaged measurement to determine if the corresponding limit has been exceeded.           | 01111111<br>11111000           | 7FF8                        | R/W         |
| D                            | Shunt-Voltage Sum        | Contains the summed value of the each of the selected shunt voltage conversions.   | 00000000<br>00000000           | 0000                        | R           |
| E                            | Shunt-Voltage Sum Limit  | Contains limit value to compare to the Shunt Voltage Sum register to determine if the corresponding limit has been exceeded. | 01111111<br>11111110           | 7FFE                        | R/W         |

Continued on next page

**Table 1.1: INA3221 Current Sensor Registers (Continued)**

| POINTER ADDRESS (Hex) | REGISTER NAME           | DETAILS   | BINARY (Power-On Reset) | HEX (Power-On Reset) | TYPE |
|-----------------------|-------------------------|---|-------------------------|----------------------|------|
| F                     | Mask/Enable             | Alert configuration, alert status indication, summation control and status.   | 00000000<br>00000010    | 0002                 | R/W  |
| 10                    | Power-Valid Upper Limit | Contains limit value to compare all bus voltage conversions to determine if the Power Valid level has been reached.                       | 00100111<br>00010000    | 2710                 | R/W  |
| 11                    | Power-Valid Lower Limit | Contains limit value to compare all bus voltage conversions to determine if the any voltage rail has dropped below the Power Valid range. | 00100011<br>00101000    | 2328                 | R/W  |
| FE                    | Manufacturer ID         | Contains unique manufacturer identification number.   | 01010100<br>01001001    | 5449                 | R    |
| FF                    | Die ID                  | Contains unique die identification number.  | 00110010<br>00100000    | 3220                 | R    |

**Table 1.1: INA3221 Current Sensor Registers**

It is worth mentioning that all voltage data is given in 2's complement and uses 13 bits, bit 15 of the register (MSB) determines the sign and bit 14-3 the voltage data. For *Shunt Voltage* the full

scale range is 163.8 mV and the LSB is 40 μV, in the case of *Bus Voltage* the LSB is 8 mV and although the full scale range of the ADC is 32.76 V, the full scale range in the case of *Bus Voltage* is 26 V since it is not recommended to apply more voltage.

## 1.2 MCP9844 Temperature sensor

The temperature sensor MCP9844 is a great IC to monitor the temperature of the environment where our DPB works, an essential magnitude to ensure operating conditions within the working range of our electronics [22].

This temperature sensor provides us with the events tool that facilitates the monitoring of the ambient temperature. The MCP9844 allows us to set temperature limits, only modifiable if enabled in the configuration register, both upper and lower and even critical temperature (only higher than the upper limit). Once the limits have been established, from the configuration register you can enable or disable the events and you can configure the event as an interruption or as a comparison, decide whether the event is active at high or low level and decide whether only the critical temperature limit is taken into account or all the limits are taken into account.

In addition, the sensor has several functionalities such as the option to include a certain hysteresis value to the temperature limits (only applicable in case of temperature drop), the possibility to modify the measurement resolution (lower resolution value will imply a longer conversion time) or the possibility to switch off the sensor if desired.

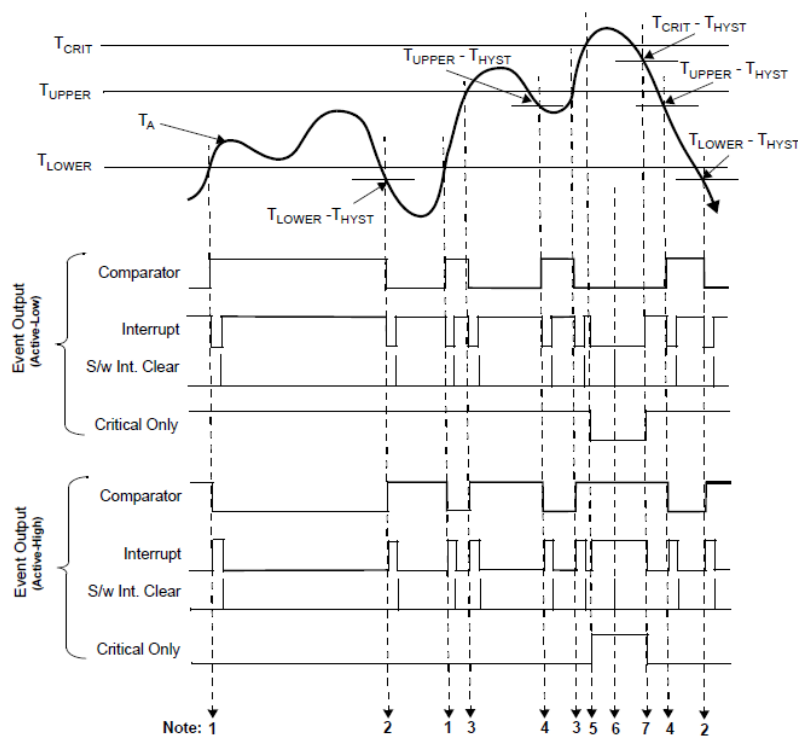


Figure 1.1: Operation of the alarms MCP9844 Temperature Sensor

Below is a table of the registers presented by this temperature sensor and their default value.

| Register Address (Hexadecimal) | Register Name                        | Default Register Data (Hexadecimal) | Power-Up Default Register Description  |
|--------------------------------|--------------------------------------|-------------------------------------|--|
| 0x00                           | Capability                           | 0x00EF                              | Event output de-asserts in shutdown I <sup>2</sup> C time out 25 ms to 35 ms. Accepts VHV at A0 Pin 0.25°C Resolution. Measures temperature below 0°C ±1°C accuracy over active range Temperature event output |
| 0x01                           | CONFIG                               | 0x0000                              | Comparator mode Active-Low output Event and critical output Output disabled Event not asserted Interrupt cleared Event limits unlocked Critical limit unlocked Continuous conversion 0°C Hysteresis            |
| 0x02                           | T <sub>UPPER</sub>                   | 0x0000                              | 0°C  |
| 0x03                           | T <sub>LOWER</sub>                   | 0x0000                              | 0°C  |
| 0x04                           | T <sub>CRIT</sub>                    | 0x0000                              | 0°C  |
| 0x05                           | T <sub>A</sub>                       | 0x0000                              | 0°C  |
| 0x06                           | Manufacturer ID                      | 0x0054                              | —  |
| 0x07                           | Microchip Device ID/ Device Revision | 0x0601                              | —  |
| 0x09                           | Resolution                           | 0x8001                              | Most Significant bit is set by default 0.25°C Measurement Resolution   |

**Table 1.3: MCP9844 Temperature Sensor Registers**

In this case, the temperature data is encoded in 2's complement and is presented as a 13-bit data, with 1 bit determining the sign and 12 bits determining the temperature data. The manufacturer provides the following equations to obtain the data in degrees Celsius.

If Temperature  $\geq 0^{\circ}\text{C}$  :

$$T_A(^{\circ}\text{C}) = (\text{UpperByte} \times 2^4 + \text{LowerByte} \times 2^{-4}) \quad (1.1)$$

If Temperature  $< 0^{\circ}\text{C}$  :

$$T_A(^{\circ}\text{C}) = (\text{UpperByte} \times 2^4 + \text{LowerByte} \times 2^{-4}) - 256 \quad (1.2)$$

Where *UpperByte* are bits 15-8 of the  $T_A$  register and *LowerByte* are bits 7-0 of the same register.

Regarding the temperature limits, these are defined by 11 bits, with 1 bit determining the sign and 10 bits to encode the absolute temperature data.

### 1.3 AFBR-5715ALZ SFP Transceiver

Opto-electronic transceivers, called SFP due to its form, have the primary function of being the communication ports on the board. These transceivers have an EEPROM memory that is divided into two pages, which correspond to the slave addresses I<sup>2</sup>C 0x50 and 0x51 in our case [23].

The SFPs collect information on highly relevant real-time quantities and are located on the second page of the EEPROM (0x51), such as the temperature within the module, the supply voltage supplied to them, the laser bias current and both the transmitted and received optical power.

On the same second page of the SFP EEPROM is the possibility to use alerts and warnings based on a range already determined by the manufacturer to monitor the status of the SFP transceivers.

Although the first page of the EEPROM is mainly based on transceiver identification characters such as part number and revision or vendor name, we can also find relevant information about the status and operation of the transceiver as we can find in this memory space the wavelength of the laser to know in which window it is working and the register that tells us if the status signals TX\_DISABLE, TX\_FAULT and RX\_LOS have been configured by hardware.

In both pages of the EEPROM we find one or more registers dedicated to a Checksum that will allow us to check the data integrity of the EEPROM itself.

Below are several tables representing the EEPROM registers of the SFP transceivers.

| Byte Decimal | Data Notes                             |
|--------------|--|
| 0            | SFP physical device                    |
| 1            | SFP function defined by serial ID only |
| 2            | LC optical connector                   |
| 6            | 1000BaseSX                             |
| 11           | Compatible with 8B/10B encoded data    |
| 12           | 1200Mbps nominal bit rate (1.25Gbps)   |
| 16           | 550m of 50/125mm fiber @ 1.25Gbps      |
| 17           | 275m of 62.5/125mm fiber @ 1.25Gbps    |
| 20-35        | 'AVAGO' - Vendor Name ASCII character  |

**Table 1.4 (continued): SFP transceiver EEPROM page 1 registers**

| Byte Decimal | Data Notes   |
|--------------|--|
| 37           | Vendor OUI   |
| 38           | Vendor OUI   |
| 39           | Vendor OUI   |
| 40-55        | 'AFBR-5715ALZ' - Vendor Part Number ASCII characters |
| 56-59        | Vendor Revision Number ASCII character               |
| 60           | Hex Byte of Laser Wavelength                         |
| 61           | Hex Byte of Laser Wavelength                         |
| 63           | Checksum for bytes 0-62                              |
| 65           | Hardware SFP TX_DISABLE, TX_FAULT, & RX_LOS          |
| 68-83        | Vendor Serial Number, ASCII                          |
| 84-91        | Vendor Date Code, ASCII                              |
| 95           | Checksum for bytes 64-94                             |

**Table 1.4: SFP transceiver EEPROM page 1 registers**

| Byte Decimal | Notes              | Byte Decimal | Notes                | Byte Decimal | Notes                            |
|--------------|--------------------|--------------|----------------------|--------------|----------------------------------|
| 0            | Temp H Alarm MSB   | 26           | Tx Pwr L Alarm MSB   | 104          | Real Time Rx P <sub>AV</sub> MSB |
| 1            | Temp H Alarm LSB   | 27           | Tx Pwr L Alarm LSB   | 105          | Real Time Rx P <sub>AV</sub> LSB |
| 2            | Temp L Alarm MSB   | 28           | Tx Pwr H Warning MSB | 106          |                                  |
| 3            | Temp L Alarm LSB   | 29           | Tx Pwr H Warning LSB | 107          |                                  |
| 4            | Temp H Warning MSB | 30           | Tx Pwr L Warning MSB | 108          |                                  |
| 5            | Temp H Warning LSB | 31           | Tx Pwr L Warning LSB | 109          |                                  |
| 6            | Temp L Warning MSB | 32           | Rx Pwr H Alarm MSB   | 110          | Status/ Control                  |
| 7            | Temp L Warning LSB | 33           | Rx Pwr H Alarm LSB   | 111          |                                  |

**Table 1.5 (continued): SFP transceiver EEPROM page 2 registers**

| Byte Decimal | Notes                       | Byte Decimal | Notes                             | Byte Decimal | Notes     |
|--------------|-----------------------------|--------------|-----------------------------------|--------------|-----------|
| 8            | VCC H<br>Alarm MSB          | 34           | Rx Pwr L<br>Alarm MSB             | 112          | Flag Bits |
| 9            | VCC H<br>Alarm LSB          | 35           | Rx Pwr L<br>Alarm LSB             | 113          | Flag Bit  |
| 10           | VCC L<br>Alarm MSB          | 36           | Rx Pwr H<br>Warning<br>MSB        | 114          |           |
| 11           | VCC L<br>Alarm LSB          | 37           | Rx Pwr H<br>Warning<br>LSB        | 115          |           |
| 12           | VCC H<br>Warning<br>MSB     | 38           | Rx Pwr L<br>Warning<br>MSB        | 116          | Flag Bits |
| 13           | VCC H<br>Warning<br>LSB     | 39           | Rx Pwr L<br>Warning<br>LSB        | 117          | Flag Bits |
| 16           | Tx Bias H<br>Alarm MSB      | 95           | Checksum<br>for Bytes<br>0-94     | 120          |           |
| 17           | Tx Bias H<br>Alarm LSB      | 96           | Real Time<br>Tempera-<br>ture MSB | 121          |           |
| 18           | Tx Bias L<br>Alarm MSB      | 97           | Real Time<br>Tempera-<br>ture LSB | 122          |           |
| 19           | Tx Bias L<br>Alarm LSB      | 98           | Real Time<br>Vcc MSB              | 123          |           |
| 20           | Tx Bias H<br>Warning<br>MSB | 99           | Real Time<br>Vcc LSB              | 124          |           |
| 21           | Tx Bias H<br>Warning<br>LSB | 100          | Real Time<br>Tx Bias<br>MSB       | 125          |           |
| 22           | Tx Bias L<br>Warning<br>MSB | 101          | Real Time<br>Tx Bias<br>LSB       | 126          |           |
| 23           | Tx Bias L<br>Warning<br>LSB | 102          | Real Time<br>Tx Power<br>MSB      | 127          |           |
| 24           | Tx Pwr H<br>Alarm MSB       | 103          | Real Time<br>Tx Power<br>LSB      | 128          |           |

**Table 1.5 (continued): SFP transceiver EEPROM page 2 registers**

| Byte Decimal | Notes                 | Byte Decimal | Notes | Byte Decimal | Notes |
|--------------|-----------------------|--------------|-------|--------------|-------|
| 25           | Tx Pwr H<br>Alarm LSB |              |       |              |       |

**Table 1.5: SFP transceiver EEPROM page 2 registers**

- **Temperature (Temp):** Temperature values are encoded as 16-bit integers in two's complement, which allows both positive and negative values to be represented. Each unit in this representation is equivalent to  $\frac{1}{256}$  of a degree Celsius ( $^{\circ}\text{C}$ ).
- **Power Supply Voltage (VCC):** This parameter is represented as a 16-bit unsigned integer, which means that it can only have positive values. Each increment in this value corresponds to 100 microvolts ( $\mu\text{V}$ ).
- **Laser Bias Current (Tx Bias):** The laser bias current is decoded as a 16-bit unsigned integer, which means that it can only be positive. Each increment in this value represents 2 microamperes ( $\mu\text{A}$ ).
- **Average Transmitted Optical Power (Tx Pwr):** This parameter is represented as a 16-bit unsigned integer, where each increment corresponds to 0.1 microwatt ( $\mu\text{W}$ ) of transmitted optical power.
- **Average Optical Power Received (Rx Pwr):** Similar to the previous parameter, the average optical power received is encoded as a 16-bit unsigned integer. Each unit of this value represents 0.1 microwatt ( $\mu\text{W}$ ) of received optical power.

As can be seen in the register table on the second page of the EEPROM, there is a status register and this describes the following cases.

| Bit # | Status/Control Name  | Description  |
|-------|----------------------|--|
| 7     | Tx Disable State     | Digital state of SFP Tx Disable Input Pin (1 = Tx_Disable asserted)            |
| 6     | Soft Tx Disable      | Read/write bit for changing digital state of SFP Tx_Disable function           |
| 4     | Rx Rate Select State | Digital state of SFP Rate Select Input Pin (1 = full bandwidth of 155 Mbit)    |
| 2     | Tx Fault State       | Digital state of the SFP Tx Fault Output Pin (1 = Tx Fault asserted)           |
| 1     | Rx LOS State         | Digital state of the SFP LOS Output Pin (1 = LOS asserted)                     |
| 0     | Data Ready (Bar)     | Indicates transceiver is powered and real-time sense data is ready (0 = Ready) |

**Table 1.6: Breakdown of SFP transceiver status bits**

As for the registers dedicated to the flags, these contain the indicator bits of the previously mentioned alerts and warnings. The following table shows their distribution in the relevant registers.



| Byte | Bit # | Flag Bit Name         | Description  |
|------|-------|-----------------------|--|
| 112  | 7     | Temp High Alarm       | Set when transceiver internal temperature exceeds high alarm threshold.      |
|      | 6     | Temp Low Alarm        | Set when transceiver internal temperature exceeds low alarm threshold.       |
|      | 5     | VCC High Alarm        | Set when transceiver internal supply voltage exceeds high alarm threshold.   |
|      | 4     | VCC Low Alarm         | Set when transceiver internal supply voltage exceeds low alarm threshold.    |
|      | 3     | Tx Bias High Alarm    | Set when transceiver laser bias current exceeds high alarm threshold.        |
|      | 2     | Tx Bias Low Alarm     | Set when transceiver laser bias current exceeds low alarm threshold.         |
|      | 1     | Tx Power High Alarm   | Set when transmitted average optical power exceeds high alarm threshold.     |
|      | 0     | Tx Power Low Alarm    | Set when transmitted average optical power exceeds low alarm threshold.      |
| 113  | 7     | Rx Power High Alarm   | Set when received P_Avg optical power exceeds high alarm threshold.          |
|      | 6     | Rx Power Low Alarm    | Set when received P_Avg optical power exceeds low alarm threshold.           |
| 116  | 7     | Temp High Warning     | Set when transceiver internal temperature exceeds high warning threshold.    |
|      | 6     | Temp Low Warning      | Set when transceiver internal temperature exceeds low warning threshold.     |
|      | 5     | VCC High Warning      | Set when transceiver internal supply voltage exceeds high warning threshold. |
|      | 4     | VCC Low Warning       | Set when transceiver internal supply voltage exceeds low warning threshold.  |
|      | 3     | Tx Bias High Warning  | Set when transceiver laser bias current exceeds high warning threshold.      |
|      | 2     | Tx Bias Low Warning   | Set when transceiver laser bias current exceeds low warning threshold.       |
|      | 1     | Tx Power High Warning | Set when transmitted optical power exceeds high warning threshold.           |
|      | 0     | Tx Power Low Warning  | Set when transmitted optical power exceeds low warning threshold.            |
| 117  | 7     | Rx Power High Warning | Set when received P_Avg optical power exceeds high warning threshold.        |
|      | 6     | Rx Power Low Warning  | Set when received P_Avg optical power exceeds low warning threshold.         |

**Table 1.7: Breakdown of the *flags* of SFP transceivers**

## Chapter 2

### Xilinx AMS gathered data

Due to the sensors together with ADC converters with which Xilinx has equipped our module and its system monitoring hardware block (SYSMON), we can access a large amount of real-time information from the PS and the PL via the Linux driver “xilinx-ams” [24]. This information collected from the PS and PL is differentiated into different channels which are explained in the following table:

| <b>SYSMON Block</b> | <b>Channel</b> | <b>Details</b>                                | <b>File Descriptor</b>   |
|---------------------|----------------|---|--|
| PS Sysmon           | 7              | LPD temperature measurement.                  | <i>in_temp7_raw,</i><br><i>in_temp7_scale,</i><br><i>in_temp7_offset</i> |
|                     | 8              | FPD temperature measurement (REMOTE).         | <i>in_temp8_raw,</i><br><i>in_temp8_scale,</i><br><i>in_temp8_offset</i> |
|                     | 9              | VCC PS LPD voltage measurement (supply1).     | <i>in_voltage9_raw,</i> <i>in_voltage9_scale</i>                         |
|                     | 10             | VCC PS FPD voltage measurement (supply2).     | <i>in_voltage10_raw,</i> <i>in_voltage10_scale</i>                       |
|                     | 11             | PS Aux voltage reference (supply3).           | <i>in_voltage11_raw,</i> <i>in_voltage11_scale</i>                       |
|                     | 12             | DDR I/O VCC voltage measurement.              | <i>in_voltage12_raw,</i> <i>in_voltage12_scale</i>                       |
|                     | 13             | PS IO Bank 503 voltage measurement (supply5). | <i>in_voltage13_raw,</i> <i>in_voltage13_scale</i>                       |
|                     | 14             | PS IO Bank 500 voltage measurement (supply6). | <i>in_voltage14_raw,</i> <i>in_voltage14_scale</i>                       |
|                     | 15             | VCCO_PSIO1 voltage measurement.               | <i>in_voltage15_raw,</i> <i>in_voltage15_scale</i>                       |
|                     | 16             | VCCO_PSIO2 voltage measurement.               | <i>in_voltage16_raw,</i> <i>in_voltage16_scale</i>                       |

Continued on next page

**Table 2.1 – continued from previous page**

| <b>SYSMON Block</b> | <b>Channel</b> | <b>Details</b>   | <b>File Descriptor</b>  |
|---------------------|----------------|--|---|
|                     | 17             | VCC_PS_GTR voltage measurement (VPS_MG-TRAVCC).          | <i>in_voltage17_raw</i> , <i>in_voltage17_scale</i>                           |
|                     | 18             | VTT_PS_GTR voltage measurement (VPS_MG-TRAVTT).          | <i>in_voltage18_raw</i> , <i>in_voltage18_scale</i>                           |
|                     | 19             | VCC_PSADC voltage measurement.                           | <i>in_voltage19_raw</i> , <i>in_voltage19_scale</i>                           |
| PL SYSMON           | 20             | PL temperature measurement.                              | <i>in_temp20_raw</i> ,<br><i>in_temp20_scale</i> ,<br><i>in_temp20_offset</i> |
|                     | 21             | PL Internal voltage measurement, VCCINT.                 | <i>in_voltage21_raw</i> , <i>in_voltage21_scale</i>                           |
|                     | 22             | PL Auxiliary voltage measurement, VCCAUX.                | <i>in_voltage22_raw</i> , <i>in_voltage22_scale</i>                           |
|                     | 23             | ADC Reference P+ voltage measurement.                    | <i>in_voltage23_raw</i> , <i>in_voltage23_scale</i>                           |
|                     | 24             | ADC Reference N- voltage measurement.                    | <i>in_voltage24_raw</i> , <i>in_voltage24_scale</i>                           |
|                     | 25             | PL Block RAM voltage measurement, VCCBRAM.               | <i>in_voltage25_raw</i> , <i>in_voltage25_scale</i>                           |
|                     | 26             | LPD Internal voltage measurement, VCC_PSINTLP (supply4). | <i>in_voltage26_raw</i> , <i>in_voltage26_scale</i>                           |
|                     | 27             | FPD Internal voltage measurement, VCC_PSINTFP (supply5). | <i>in_voltage27_raw</i> , <i>in_voltage27_scale</i>                           |
|                     | 28             | PS Auxiliary voltage measurement (supply6).              | <i>in_voltage28_raw</i> , <i>in_voltage28_scale</i>                           |
|                     | 29             | PL VCCADC voltage measurement (vccams).                  | <i>in_voltage29_raw</i> , <i>in_voltage29_scale</i>                           |

**Table 2.1: SYSMON channels and their details**

The chart starts from channel 7 as the previous channels are from the AMS Control SYSMON block and display repeated information from the PL which is not used by the AMS driver.

The information obtained is displayed in ADC code in the *\_raw* file and has to be scaled with the value obtained in the *\_scale* file. In the case of temperature, an offset from the *\_offset* file must also be applied. Every file is a virtual file generated by the *sysfs* file system. The expressions used to convert the values read to the corresponding magnitude are shown below:

$$V_{XX}(V) = (in\_voltageXX\_raw \times in\_voltageXX\_scale) \times \frac{1}{2^{n\_bits}} \quad (2.1)$$

$$T_{XX}(C) = (in\_tempXX\_raw + in\_tempXX\_offset) \times \frac{1}{2^{n\_bits}} \quad (2.2)$$

Where XX defines the selected channel number in voltage or temperature and “n\_bits” defines the number of bits of the ADC used, in our case 10 bits. The offset in the case of temperature is added since a negative number is returned.

Xilinx also offers alarms applied to the voltages and temperatures measured on the previously mentioned channels and the Linux driver allows us to configure and read these alarms also using the *IIO\_EVENT\_MONITOR* tool of Linux itself. In order to detect events or enable event detection, the *IIO\_EVENT\_MONITOR* application works along with the *sysfs* file system.

*IIO\_EVENT\_MONITOR* uses the IIO framework in Linux, which facilitates the acquisition and control of industrial devices such as sensors and actuators. It consists of device drivers integrated into the kernel, a user API for interacting with the devices, and user-friendly interfaces such as the mentioned previously, *sysfs*. Its modular and flexible design makes it adaptable to a wide range of industrial devices and applications [25].

In the case of temperature, there are only alarms that are activated if a certain temperature is exceeded, while in the case of voltage, there are alarms for both overvoltage and undervoltage, but without specifying whether the limit exceeded is the lower or upper limit as the alarm is a single bit, so it does not discriminate between falling or rising event (shown as *either*) [26].

| Field Name | Bits | Type                       | Reset Value | Description   |
|------------|------|----------------------------|-------------|---|
| pl_alm_15  | 31   | Readable, write 1 to clear | 0x0         | PL Sensor Alarms – OR of bits [29:16].  |
| pl_alm_14  | 30   | Readable, write 1 to clear | 0x0         | reserved  |
| pl_alm_13  | 29   | Readable, write 1 to clear | 0x0         | reserved  |
| pl_alm_12  | 28   | Readable, write 1 to clear | 0x0         | PL ADC voltage, VCCADC.   |
| pl_alm_11  | 27   | Readable, write 1 to clear | 0x0         | PL VUser3.  |
| pl_alm_10  | 26   | Readable, write 1 to clear | 0x0         | PL VUser2.  |
| pl_alm_9   | 25   | Readable, write 1 to clear | 0x0         | PL VUser1.  |
| pl_alm_8   | 24   | Readable, write 1 to clear | 0x0         | PL VUser0.  |
| pl_alm_7   | 23   | Readable, write 1 to clear | 0x0         | PL Sensor Alarms – OR of bits [22:16].  |
| pl_alm_6   | 22   | Readable, write 1 to clear | 0x0         | VCC_PSAUX   |
| pl_alm_3   | 19   | Readable, write 1 to clear | 0x0         | VCCBRAM.  |
| pl_alm_2   | 18   | Readable, write 1 to clear | 0x0         | PL_VCCAUX   |
| pl_alm_1   | 17   | Readable, write 1 to clear | 0x0         | PL_VCCINT   |
| pl_alm_0   | 16   | Readable, write 1 to clear | 0x0         | PL Temperature  |
| ps_alm_15  | 15   | Readable, write 1 to clear | 0x0         | PS Sensor Alarms – OR of bits [13:0].   |
| ps_alm_14  | 14   | Readable, write 1 to clear | 0x0         | reserved  |
| ps_alm_13  | 13   | Readable, write 1 to clear | 0x0         | FPD Temperature.  |
| ps_alm_12  | 12   | Readable, write 1 to clear | 0x0         | VCC_PSADC voltage.  |
| ps_alm_11  | 11   | Readable, write 1 to clear | 0x0         | PS_MGTRAVTT voltage (supply10).   |
| ps_alm_10  | 10   | Readable, write 1 to clear | 0x0         | PS_MGTRAVCC voltage (supply9).  |
| ps_alm_9   | 9    | Readable, write 1 to clear | 0x0         | VCCO_PSIO2 I/O bank 502, MIO[52:77].  |
| ps_alm_8   | 8    | Readable, write 1 to clear | 0x0         | VCCO_PSIO1 I/O bank 501, MIO[26:51].  |
| ps_alm_7   | 7    | Readable, write 1 to clear | 0x0         | PS Sensor Alarms – OR of bits [6:0].  |
| ps_alm_6   | 6    | Readable, write 1 to clear | 0x0         | VCCO_PSIO0 I/O bank 500, MIO[0:25].   |
| ps_alm_5   | 5    | Readable, write 1 to clear | 0x0         | VCCO_PSIO3 I/O bank 503, boot mode, serial config, JTAG, error output, error status, SRST, POR. |
| ps_alm_4   | 4    | Readable, write 1 to clear | 0x0         | VCCO_PSDDR, bank 504, DDR I/O.  |
| ps_alm_3   | 3    | Readable, write 1 to clear | 0x0         | VCCO_PSAUX auxiliary power supply for BPU, eFuse, GPIOB logic.                                  |
| ps_alm_2   | 2    | Readable, write 1 to clear | 0x0         | FPD internal voltage, VCC_PSINTFP.  |
| ps_alm_1   | 1    | Readable, write 1 to clear | 0x0         | LPD internal voltage, VCC_PSINTLP.  |
| ps_alm_0   | 0    | Readable, write 1 to clear | 0x0         | LPD Temperature.  |

**Table 2.2: AMS alarms register set**



## **Part IV**

# **Tasks Development and Results**



## Chapter 1

# Preparation of the environment to be used on the board

This chapter aims to detail the process of adaptation and acclimatisation to the work environment in order to be able to carry out the corresponding tasks.

### 1.1 Platform setup and configuration

Starting with the environment to work on the DPB, we will use PetaLinux, a Xilinx software development tool based on a light version of Linux.

The universal availability of the Linux source code and the infinite number of drivers available in Linux gives us greater flexibility and ease of working at the application level on the DPB. To implement this OS on the DPB we have used the Xilinx software, Vivado, and through the JTAG port we have loaded on a 16 GigaBytes eMMC as non-volatile memory, both the relevant boot files and the custom image of the PetaLinux project, *image.ub*. As boot files we find *BOOT.BIN* which is the First Stage Boot Loader, in addition to other essential files such as the device tree, and *boot.scr*, which is a script that defines the boot up process to the board. Then I have selected the eMMC as the main boot option by means of switches from the board itself. In the boot process, the OS is loaded onto the RAM and the RAM is worked on.

Once the OS has been installed, the connection with the DPB has to be configured. Despite the possibility of maintaining the connection via JTAG, the main source of communication of the DPB is going to be via Ethernet, through SSH protocol, so one of the SFP ports of the DPB has been used to make an Ethernet connection with the equipment by means of an SFP transceiver. For this purpose, the configuration of a 125 MHz PLL for the corresponding Ethernet clock signal was included in the customization of the PetaLinux version.

It should be noted that the main communication with the DPB will be via Ethernet, so the JTAG port after the initial loading of the boot files will only be considered for debugging actions.

Once the connection has been configured, a local DHCP server has been set up to assign an IP address to the DPB and facilitate the connection via SSH to the board. For this purpose, the subnet has been declared with a very basic configuration on the server:



**Listing 1.1: Subnet Configuration**

```
1 subnet 20.0.0.0 netmask 255.255.255.0 {  
2   range 20.0.0.2 20.0.0.30;  
3   option routers 20.0.0.1;  
4 }
```

The network interface of the PC connected to the DPB in question has been assigned the address 20.0.0.1 and the subnet has been declared with a small arbitrary range, and the DPB has been assigned the fixed IP address 20.0.0.33, an address outside the range, since otherwise, the server would return an error. It should be noted that the SFP ports of the DPB are designed to use optic fiber ports, so sometimes the equipment is not able to detect the connection on the Ethernet port using Ethernet cable with RJ-45, so the interface has to be deactivated and then re-activated and assigned the address 20.0.0.1 and the problem is solved, in the case of using a optic fiber port, this problem does not arise.

With the fixed IP address already assigned, it is now possible to access the board via SSH and communicate with it using the following command:

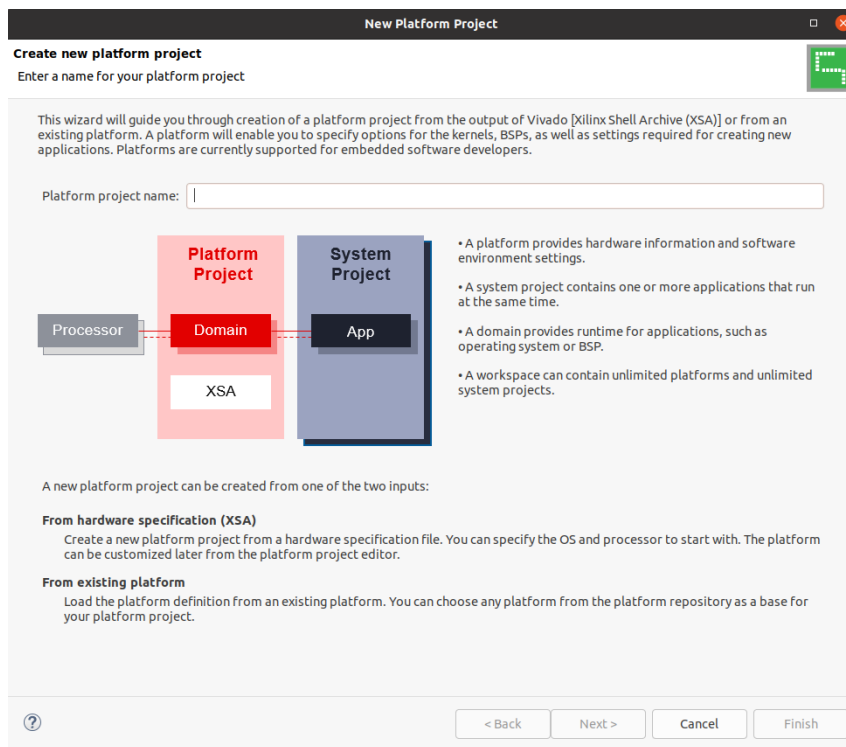
**Listing 1.2: Command to establish SSH connection with the DPB**

```
1 ssh root@20.0.0.33  
2 #Here we would enter the relevant password
```

## 1.2 Vitis Project Creation

To finish with the establishment of the working environment, we only have to create the application project that is going to be developed on a customized platform of our project, in the Vitis IDE software of Xilinx. The application project has been named DBP2\_App.

For this purpose, I first have to create the platform on which the application I want to develop will run. In order to achieve this, I have used the Platform project wizard from Vitis IDE to import the custom platform provided by my project colleague in a .xsa file.



**Figure 1.1: Vitis IDE Platform project wizard**

After creating the Platform project from the mentioned .xsa file, the information on the custom configuration of the hardware and software environment is now available so the application project that will run on this customized environment can now be created.

The application project has also been created using the available Vitis IDE Application project wizard, and with the project created, the development of the application can be started.



## Chapter 2

# Application workflow

When starting to develop the application, the flow of execution of the application must be clarified prior to the start of development, which is why this chapter will explain, with the help of several flow charts, the flow of execution initially proposed for the application and its sub-processes.

### Main execution flow diagram

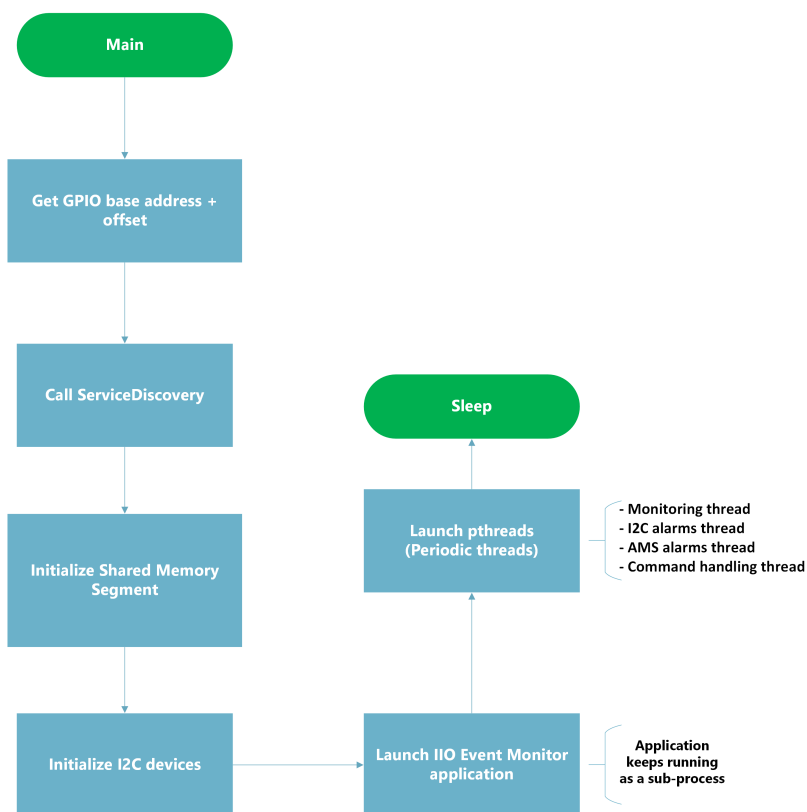


Figure 2.1: Main application execution flow

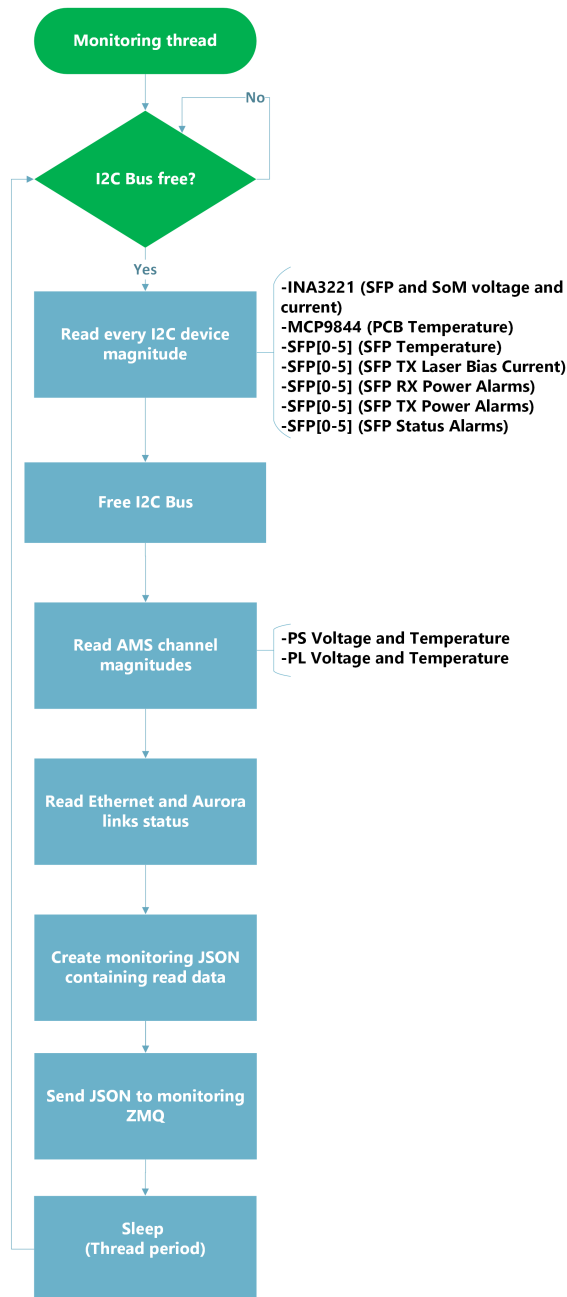
As it has been mentioned in previous sections, it has been decided to use different periodic threads rather than a single infinite loop to optimize resource usage and to be able to prioritize executions in a simple way. The periodic threads run in the background and each performs different *slow control* tasks, be it monitoring, checking alarm status and handling them, or receiving and processing DAQ commands. Wherefore, as it can be seen in the diagram 2.1, the main application is only responsible for initializing the shared memory segment, the relevant ZeroMQ sockets and I<sup>2</sup>C devices, the threads and sub-processes. It also obtains the address needed to operate the GPIO pins.

During the initialization process, the necessary semaphores are also set to avoid race conditions both during the initialization and during the execution of the threads, releasing shared resources between threads to allow the use of them in another thread. Moreover, the I<sup>2</sup>C device initialization function checks if there is a problem on the I<sup>2</sup>C bus by reporting it as an alarm and determines which SFPs are connected to take it into account when reading magnitudes or alarms from them.

Finally, the main thread remains dormant to reduce the consumption of CPU resources while waiting for a termination or interruption signal to release the devices and resources used prior to the end of the application. This sequential execution flow has been reached in the main thread and periodically in the child threads created, taking into account the premises instructed during the degree on these types of execution flow. In addition, optimization of this code should be sought, avoiding redundancy and heavy functions that could increase the board's resource consumption and energy usage. I have used the memory profiling tool Valgrind to locate possible memory leaks and thus eliminate them. It has also been very helpful in debugging segmentation faults caused by memory loss or overflow at some point in the code

---

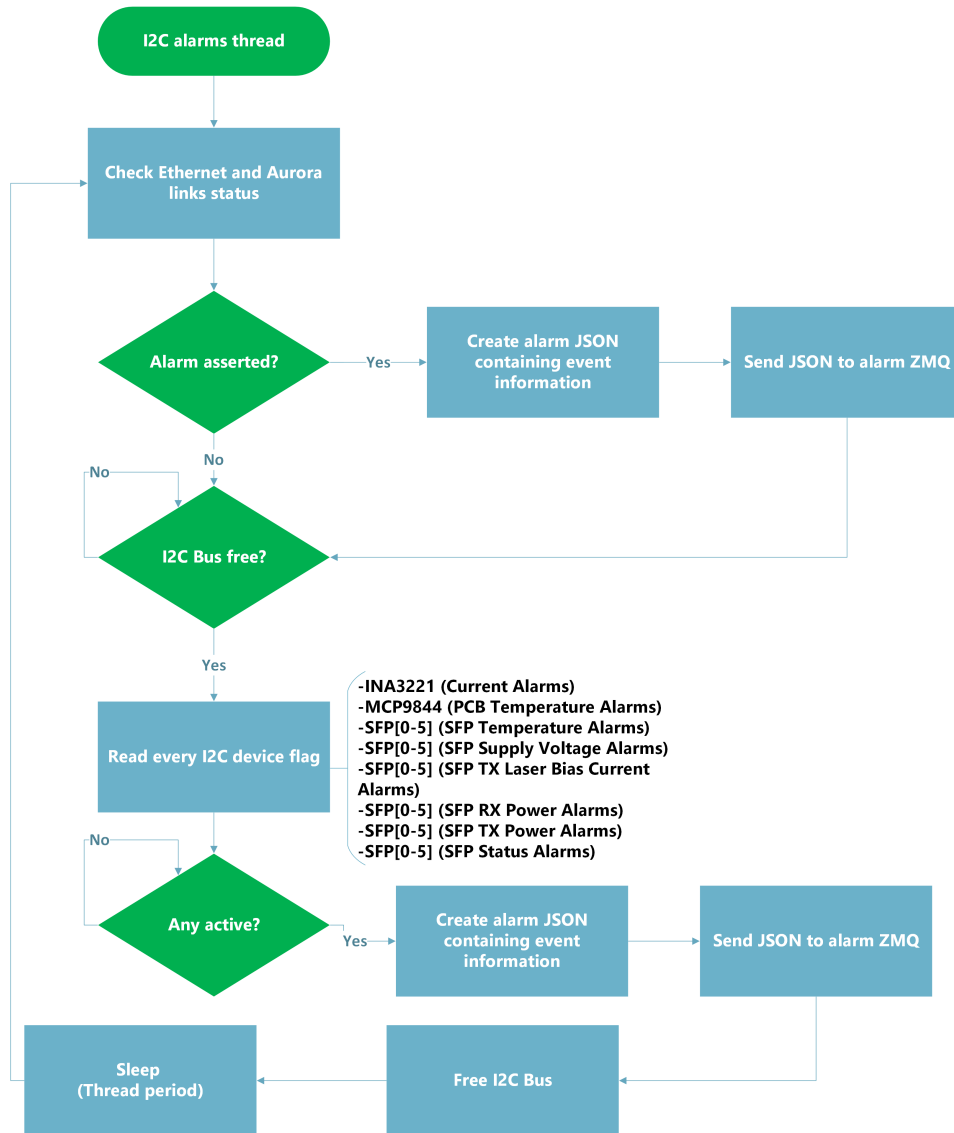
## Monitoring thread execution flow



**Figure 2.2: Monitoring thread execution flow**

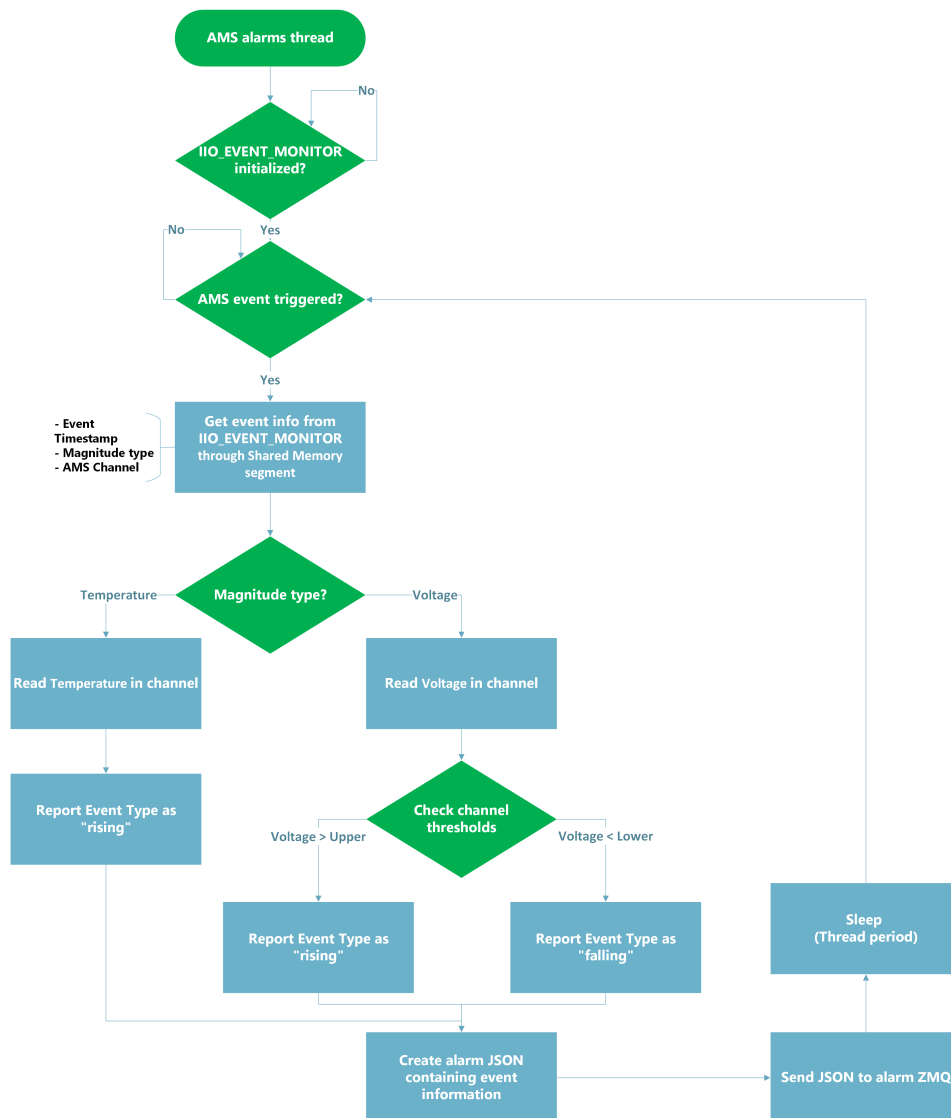
Regarding the monitoring thread, it can be seen in the diagram 2.2 that it has been proposed a linear execution flow: reading the magnitudes and states of devices and links available periodically while respecting the use of the I<sup>2</sup>C bus and finally the data collected is sent in JSON format to the DAQ.

## I<sup>2</sup>C alarms thread execution flow



**Figure 2.3: I<sup>2</sup>C alarms execution flow**

## AMS alarms thread execution flow

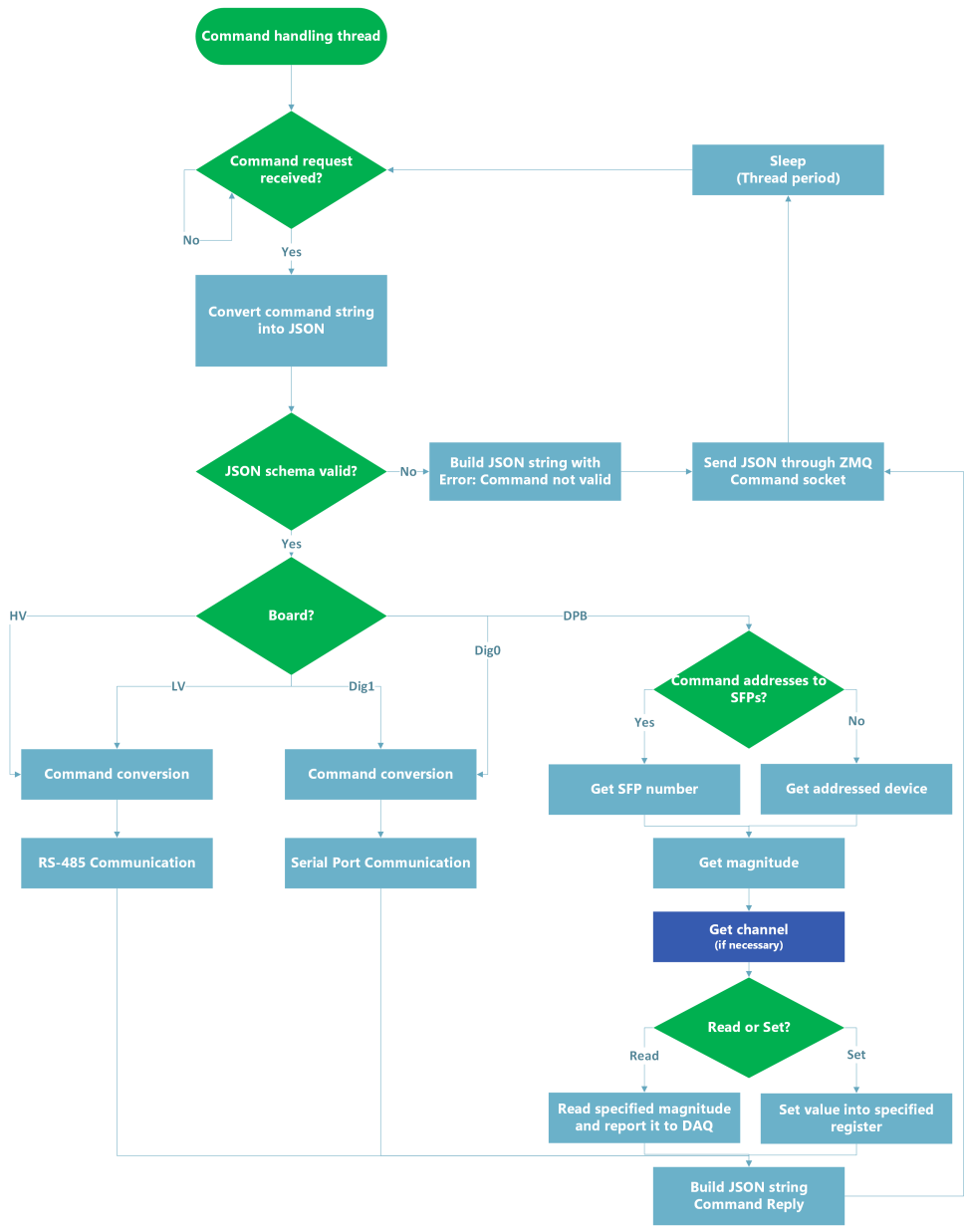


**Figure 2.4: AMS alarms thread execution flow**

As can be seen in both diagrams 2.4 and 2.3 the operation of the AMS alarm threads and the I<sup>2</sup>C devices is very similar, they report the same JSON message format but the AMS alarm thread captures the events through the IIO Event Monitor application and transmits the information to the alarm thread through the shared memory. Whereas, the I<sup>2</sup>C alarm thread requires the I<sup>2</sup>C bus to read the necessary flags to catch the alarm events, it also includes the use of GPIO and terminal commands to also take into account the status of the Ethernet and Aurora links.



### Command handling thread execution flow



**Figure 2.5: Slow Control thread execution flow**

Finally, the slow control thread is in charge of receiving the commands sent by the DAQ, checking if the command is correct and processing it, executing the action indicated by the command and sending the pertinent response to the DAQ in a JSON formatted message.

It could be seen in the diagrams that all the threads are periodically running so that they continue to run indefinitely until the application is finished. For the period of each thread an approximate time has been assigned that has been considered correct for each thread while waiting to assign values for each period with other members of the project.

---

For the monitoring thread, a period of 5 seconds has been established since it is not a priority, while for both alarm threads, a period of 100 ms has been set considering the sensor conversion times and giving maximum possible priority. Finally, the *Slow Control* thread has been assigned a period of 50 ms for maximum priority.



## Chapter 3

# Application initialization

As it has been mentioned in the previous chapter, the first step of the developed application is to initialize every device and process that will be used. Consequently, in this chapter, the application initialization process will be explained, and the most relevant code fragments in this process will be presented, even though all developed functions are explained in more detail in the documentation generated as an HTML file with the Doxygen tool, which is planned to be submitted as an annex to this thesis, being “index.html” the main HTML file of the documentation.

To begin with the initialization process, firstly, every I<sup>2</sup>C device has been gathered and globally defined in a struct. Then, the following functions have been developed to initialize each of the I<sup>2</sup>C devices:

**Listing 3.1: I<sup>2</sup>C devices struct**

```
1 struct DPB_I2cSensors{
2
3     struct I2cDevice dev_pcb_temp;
4     struct I2cDevice dev_sfp0_2_volt;
5     struct I2cDevice dev_sfp3_5_volt;
6     struct I2cDevice dev_som_volt;
7     struct I2cDevice dev_sfp0_A0;
8     struct I2cDevice dev_sfp1_A0;
9     struct I2cDevice dev_sfp2_A0;
10    struct I2cDevice dev_sfp3_A0;
11    struct I2cDevice dev_sfp4_A0;
12    struct I2cDevice dev_sfp5_A0;
13    struct I2cDevice dev_sfp0_A2;
14    struct I2cDevice dev_sfp1_A2;
15    struct I2cDevice dev_sfp2_A2;
16    struct I2cDevice dev_sfp3_A2;
17    struct I2cDevice dev_sfp4_A2;
18    struct I2cDevice dev_sfp5_A2;
19 };
```

**Listing 3.2: I<sup>2</sup>C devices struct**

```
1 int init_tempSensor (struct I2cDevice *dev) {
2     int rc = 0;
3     uint8_t manID_buf[2] = {0,0};
```

```

4     uint8_t manID_reg = MCP9844_MANUF_ID_REG;
5     uint8_t devID_buf[2] = {0,0};
6     uint8_t devID_reg = MCP9844_DEVICE_ID_REG;
7     :

```

As seen in the function fragment, only the I<sup>2</sup>C device itself is required as an input parameter for its initialization, and within the function, register addresses are determined from which to read to verify that the device has started correctly. The rest of the I<sup>2</sup>C devices follow a similar structure to be started, only the verification methods differ. Firstly, the device is started using the function *i2c\_start* from the I<sup>2</sup>C Linux driver, then we check if we are initializing the correct device by checking specific registers and comparing them to their expected value and if any step fails, the function returns a negative integer which indicates the error. For SFPs, the initialization process is a bit different since each of the two pages of their EEPROM is initialized as an independent devices. Furthermore, I have decided to use the checksum contained in the memory pages to verify the proper data integrity of the memory pages.

The following function has been developed to verify the checksum value is correct:

**Listing 3.3: Checksum validator function**

```

1 int checksum_check(struct I2cDevice *dev, uint8_t ini_reg, int size){
2     int rc = 0;
3     int sum = 0;
4     uint8_t byte_buf[size+1] ;
5
6     rc = i2c_readn_reg(dev, ini_reg, byte_buf, 1); //Read every register
           from ini_reg to ini_reg+size-1
7         if(rc < 0)
8             return rc;
9     for(int n=1;n<(size+1);n++){
10        ini_reg ++;
11        rc = i2c_readn_reg(dev, ini_reg, &byte_buf[n], 1);
12            if(rc < 0)
13                return rc;
14    }
15
16    for(int i=0;i<size;i++){
17        sum += byte_buf[i]; //Sum every register read in order to
           obtain the checksum
18    }
19    uint8_t calc_checksum = (sum & 0xFF); //Only taking the 8 LSB of
           the checksum as the checksum register is only 8 bits
20    uint8_t checksum_val = byte_buf[size];
21    if (checksum_val != calc_checksum){ //Check the obtained checksum
           equals the device checksum register
22        printf("Checksum value does not match the expected value
           \r\n");
23        return -EHWPOISON;
24    }
25    return 0;
26 }

```

---

In order to verify the checksum from any SFP, a function was needed that calculates the current checksum value and compares it to the expected value. So as to calculate the checksum, the function needs to be given the I<sup>2</sup>C device, the address of the first register to be counted in the checksum calculation, the expected checksum value register address, and the number of registers to be counted in the checksum calculation.

The function sums every register in the given range, and only takes the 8 LSB as the SFP registers size is 1 byte.

#### Listing 3.4: I<sup>2</sup>C device definition and initialization function

```
1 int init_I2cSensors(struct DPB_I2cSensors *data){
2
3     data->dev_pcb_temp.filename = "/dev/i2c-2";
4     data->dev_pcb_temp.addr = 0x18;
5     :
```

Finally, in this function, we define every filename and slave address for every I<sup>2</sup>C device and we call every initialization function mentioned previously and in case any initialization is missed, it is reported which device has failed. In order to report any device initialization errors, the sockets necessary to establish communication with the DAQ are first started. It has been established that all the sockets used work over TCP to ensure the sending and receiving of messages using ACKs, and the output buffer size for sockets and the message retention time have been reduced to avoid resource usage when no receiver is connected to the socket.

#### Listing 3.5: I<sup>2</sup>C device stopping function

```
1 int stop_I2cSensors(struct DPB_I2cSensors *data){
2
3     i2c_stop(&data->dev_pcb_temp);
4     :
```

It has also been developed the *stop\_I2cSensors* function to terminate the I<sup>2</sup>C devices by using I<sup>2</sup>C application should be permanently active.

Regarding the *IIO Event Monitor* initialization, it has been executed as a sub-process that will detect AMS alarms as events and it is executed by the following function:

#### Listing 3.6: IIO Event Monitor executing function

```
1 int iio_event_monitor_up() {
2     pid_t pid = fork(); // Create a child process
3
4     if (pid == 0) {
5         // Child process
6         // str: Path of the .elf file and arguments
7         char *args[] = {str, "-a", "/dev/iio:device0", NULL};
8
9         // Execute the .elf file
10        if (execvp(args[0], args) == -1) {
11            perror("Error executing the .elf file");
12            return -1;
13        }
14    } else if (pid > 0) {
```

```
15     // Parent process
16     // You can perform other tasks here while the child process
17     // executes the .elf file
18 } else {
19     // Error creating the child process
20     perror("Error creating the child process");
21     return -1;
22 }
23 return 0;
}
```

This function executes the *IIO Event Monitor* application through its binary file, built with Release optimizations parameters passed to the GCC compiler. It should be emphasized that this *IIO Event Monitor* is slightly customized by us so as to include shared memory configuration to communicate the main application with it. At first, it was recommended to use the function `system()` to execute the process as it was a bash command. Nevertheless, it did not result as expected so it was decided to use the function `execvp()`, which also provides us with a more visual and convenient way of passing the necessary arguments to the function.

Furthermore, due to the multi-threaded execution of our application, 5 semaphores have been enabled prior to starting the threads to prevent potential race conditions in different situations:

- **sem\_t i2c\_sync** : Determines the I<sup>2</sup>C bus usage shift.
- **sem\_t thread\_sync** : Avoids stalling when starting threads due to ZeroMQ functions are not thread-safe.
- **sem\_t file\_sync** : Avoids trying to read non-existing GPIO sysfs files, overwrite existing GPIO sysfs files or any possible race condition when dealing with GPIO or Ethernet status files.
- **sem\_t alarm\_sync** : Avoids stalling when sending multiple alarms from different threads due to ZeroMQ functions are not thread-safe.
- **sem\_t sem\_valid** : Avoids race condition when validating a JSON schema.

A semaphore is also included in the shared memory to force the *IIO Event Monitor* to boot before the AMS alarm thread.

Finally, all threads are created in the determined order and start running independently.

---

### Listing 3.7: Thread creation

```
1 pthread_create(&t_1, NULL, ams_alarms_thread, NULL); //Create thread 1
   - reads AMS alarms
2 sem_wait(&thread_sync);
3 pthread_create(&t_2, NULL, i2c_alarms_thread, (void *)&data); //Create
   thread 2 - reads I2C alarms every x milliseconds
4 sem_wait(&thread_sync);
5 pthread_create(&t_3, NULL, monitoring_thread, (void *)&data); //Create
   thread 3 - monitors magnitudes every x seconds
6 sem_wait(&thread_sync); //Avoids race conditions
7 pthread_create(&t_4, NULL, command_thread, (void *)&data); //Create
   thread 4 - waits and attends commands
```





## Chapter 4

# Monitoring thread development

Once all the necessary elements have been set up, the development of the threads has begun with the monitoring thread. It has been decided to start with this thread since, as previously presented, it exhibits the most linear and straightforward behaviour of all threads. This will serve as an introduction to multi-threaded programming and a solid foundation for the development of subsequent threads.

### 4.1 Sensor data readout functions

In order to be able to read information from the I<sup>2</sup>C sensors by using the I<sup>2</sup>C Linux driver, I have used the functions *i2c\_write()* to write the address of the register I want to read in the register pointer, so that by using the function *i2c\_read()* I can read the desired register byte to byte as all the I<sup>2</sup>C devices that have been used 2 bytes registers and allow continuous reading apart from the SFPs.

In order to read from the SFPs, I used the function *i2c\_readn\_reg()* which is an implicit combination of *i2c\_write()* to write in the register pointer and *i2c\_read()*. As the SFPs do not allow continuous reading and their register size is 1 byte, *i2c\_readn\_reg()* has been used two times to read MSB and LSB of the desired data and it has been specified the appropriate register address for each operation. In addition to using the functions provided by the Linux driver, as can be seen, I have also needed both the datasheets for each I<sup>2</sup>C device and the knowledge of serial communication acquired during the degree to truly understand how to use the tools provided by the driver for my purposes.

Regarding the data provided by the AMS, we obtain them in ADC code by calling a function that will access the sysfs files generated by the “xilinx-ams” driver, and the final magnitude is obtained by applying the conversion explained in prior theory chapters depending on whether we are dealing with voltage or temperature.

```
root@STIME-XU8-4CG-1E-D11E:/run/media/mmcblk0p1# ./DBP2_App.elf
Monitoring thread period: 10s
Found IIO device with name /dev/iio:device0 with device number 0
Temperatura ambiente: 35.500000 °C

Temperatura SFP: 37.128906 °C
Tensión SFP: 3.271400 V
Corriente polarización del láser SFP: 0.004092 A
Potencia óptica transmitida SFP: 0.000270 W
Potencia óptica recibida SFP: 0.000000 W

Temperatura AMS - Canal 7: 74.315109 °C - Iteración: 0
Temperatura AMS - Canal 8: 72.857948 °C - Iteración: 0
Temperatura AMS - Canal 20: 70.687393 °C - Iteración: 0

Tensión AMS - Canal 9: 0.819460 V - Iteración: 0
Tensión AMS - Canal 10: 0.819325 V - Iteración: 0
Tensión AMS - Canal 11: 1.754612 V - Iteración: 0
Tensión AMS - Canal 12: 1.166761 V - Iteración: 0
Tensión AMS - Canal 13: 3.152490 V - Iteración: 0
Tensión AMS - Canal 14: 1.742899 V - Iteración: 0
Tensión AMS - Canal 15: 3.154904 V - Iteración: 0
Tensión AMS - Canal 16: 1.744419 V - Iteración: 0
Tensión AMS - Canal 17: 0.835687 V - Iteración: 0
Tensión AMS - Canal 18: 1.755953 V - Iteración: 0
Tensión AMS - Canal 19: 1.753226 V - Iteración: 0
Tensión AMS - Canal 21: 0.837743 V - Iteración: 0
Tensión AMS - Canal 22: 1.756400 V - Iteración: 0
Tensión AMS - Canal 23: 1.220360 V - Iteración: 0
Tensión AMS - Canal 24: 0.000000 V - Iteración: 0
Tensión AMS - Canal 25: 0.823840 V - Iteración: 0
Tensión AMS - Canal 26: 0.821471 V - Iteración: 0
Tensión AMS - Canal 27: 0.818923 V - Iteración: 0
Tensión AMS - Canal 28: 1.757026 V - Iteración: 0
Tensión AMS - Canal 29: 1.753137 V - Iteración: 0
```

**Figure 4.1: Monitoring thread value**

In the figure 4.1, a first approximation to the desired operation of the monitoring thread can be observed, printing the read values to the terminal. This has successfully verified that the readings of the values are correct and that, when reading, only the properly initialized SFPs are taken into account, in this case only the SFP0.

## 4.2 Parse monitoring data into JSON string and send it to the DAQ

Once the correct functioning of the functions designed to read the desired magnitudes and states has been confirmed, you can proceed to use the json-c library to encapsulate the collected information into a JSON string that will be sent to the DAQ.

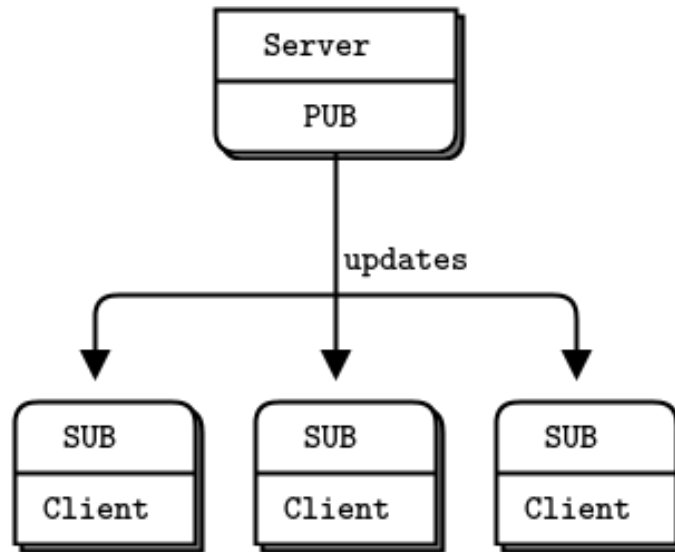
**Listing 4.1: Monitoring JSON format example**

```
1 {
2   "time": 0,
3   "device": "ID DPB",
4   "data": {
5     "HV": [],
6     "LV": [],
7     "Dig1": [],
8     "Dig0": [],
9     "DPB": [
10      {
11        "magnitudename": "PL Temperature",
12        "value": 75.256
13      }
14    ]
15  }
16 }
```

The monitoring JSON must follow the structure which can be seen in the prior listing. As a way to make sure the JSON string that is going to be sent is valid, my project colleague, whose tasks include supporting the DPB and developing and implementing the customization of the OS that runs on the DPB, has been in charge of preloading an application called *json-schema-validate* in the PetaLinux image. The purpose of this application is to use a schema, which have also been developed and loaded into the PetaLinux image by my colleague, as a template that has to comply with the JSON created to send to the DAQ. So as to validate the JSON from our application, the function *json\_schema\_validate()* stores the created JSON string in a temporary file, executes the *json-schema-validate* with a terminal command and checks if the response is positive or negative [27].

Since the alarm and command threads also use JSON strings, this schema validation application will be used in the other threads as well with a custom schema for each message type, either to validate alarm messages that are sent to the DAQ or to validate a properly formatted command message.

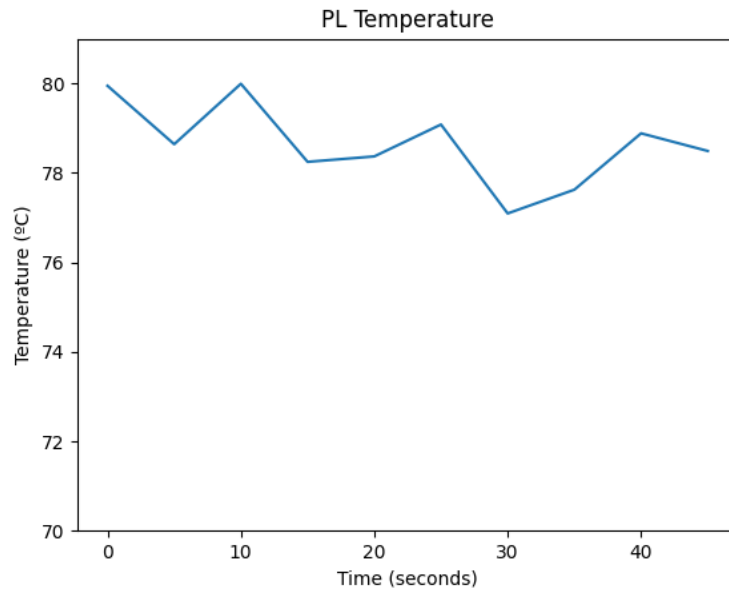
After validating the JSON string, it is sent through the corresponding ZeroMQ socket to the DAQ. Since the library to perform communication by the standard agreed by Hyper-Kamiokande collaboration is not available yet, I have designed a simple Python application to send or receive JSON strings. With this, I manage to establish and verify communication of the DPB through ZeroMQ so that the subsequent implementation of the mentioned library is much more straightforward.



**Figure 4.2: ZeroMQ Publisher-Subscriber simple pattern [28]**

To establish communication between the monitoring thread and the DAQ servers or the used Python application, the socket type for both the monitoring thread and the message-receiving application must be determined.

The communication of the monitoring thread is intended to be unidirectional, whereby the thread transmits the information gathered periodically and the clients who are monitoring the thread receive the information. That is why I have decided to opt for a simple ZeroMQ Publisher-Subscriber communication configuration as shown in figure 4.2 among other possible configurations. In this configuration, the monitoring thread is established as the Publisher, the one in charge of sending information through a specific socket and port, and the Subscribers connect to the corresponding address and port and wait on the other side of the communication to receive messages from the Publisher.



**Figure 4.3: PL Temperature monitored evolution**

In figure 4.3 can be seen an example of a magnitude arbitrarily selected from all the magnitudes sent periodically through the monitoring socket to a Python-developed application and the application plotted the evolution over time of the data of this particular magnitude sent in the JSON string, verifying the monitoring thread communication through ZeroMQ socket is working correctly as well as the magnitude value is within its typical working range since it is about the temperature of silicon while using a small passive heat-sink.



## Chapter 5

# Alarms threads development

After developing the monitoring periodic thread, the same periodic structure has been followed for the development of the alarm threads. However, the period will be much shorter than in the monitoring thread as detecting any alarm is much more critical.

It has been decided to divide the alarm thread into two different threads, one for the I<sup>2</sup>C devices and the other for the AMS alarms. This decision has been made since the I<sup>2</sup>C devices detect alarm information by reading from a register while the AMS detects it through the *IIO Event Monitor* sub-process. Furthermore, the conversion time of the I<sup>2</sup>C devices restricted the alarm triggering of the AMS too much. Therefore, linking alarm readings from I<sup>2</sup>C or GPIO devices that depend on registers or files with AMS alarms that depend on another process would unnecessarily complicate the synchronisation of alarm feedback with other shared resources such as the I<sup>2</sup>C bus and would delay the reaction time to any alarms detected.

Regarding the I<sup>2</sup>C devices alarms thread, it has been necessary to use a POSIX semaphore to synchronize the I<sup>2</sup>C bus usage and avoid race conditions between any other thread. In regards to the operation of the thread, it calls functions that read the flag registers of the I<sup>2</sup>C devices and checks if there is an active flag, cleans it if necessary, and depending on which flag is activated, the event is communicated to the DAQ.

In order to develop the AMS alarms thread, we first need the Linux kernel tool *IIO Event Monitor*, to work along the “xilinx-ams” driver so it allows us to catch AMS alarms and the communication between the driver and *IIO Event Monitor* occurs through the previously mentioned Linux IIO framework. For this purpose, the *IIO Event Monitor* has been modified in order to be able to transmit the detected event information to our main application .

First of all, a segment of shared memory has been established between the main application and the *IIO Event Monitor* sub-process to transmit the necessary event information and set up semaphores to synchronize sub-process and thread. One semaphore is used to force the AMS alarm thread to start *IIO Event Monitor*, and the other semaphore is used to indicate to the alarm thread that an event has been detected and the thread will handle it accordingly.

As the *IIO Event Monitor* does not provide us with the value of the magnitude that has triggered the alarm and does not differentiate rising and falling voltage events, the thread has been configured to take care of this.

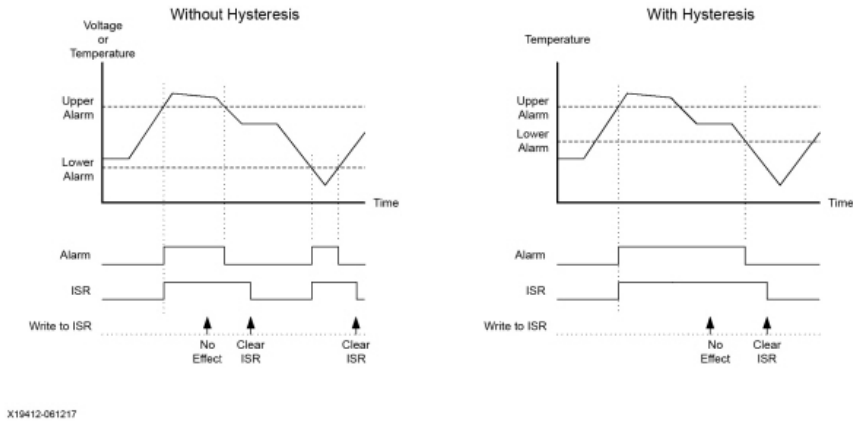
It should be noted that in the process of debugging this thread, several bugs have been detected



in the “xilinx-ams” driver that we have solved to ensure the correct functioning of our application.

The first encountered bug is due to the driver masking an alarm when it is triggered so that it is not detected again until it has been previously reset to normal values to avoid the same alarm going off constantly. However, the unmasking process proposed an impossible situation and did not allow an alarm to go off more than once despite a previous reset as it should work. By modifying several logical operations, the problem was solved without affecting its performance.

The other problem that was found was a lack of functionality of the driver since the AMS by default enables hysteresis for the temperature alarm and the lower limit places it at 0 degrees. The problem is that the driver does not allow you to modify the lower limit or disable hysteresis, so it only allowed you to enable temperature alarms once. We made the decision to disable hysteresis from the driver itself since it does not suit our application and with this, we find the desired operation of the AMS alarms



**Figure 5.1: Difference between AMS temperature alarm with hysteresis on and hysteresis off**

The figure 5.1 shows how AMS hysteresis alarm system works, where it is clearly seen that hysteresis does not provide any usefulness to our application if the driver does not allow changing the lower alarm threshold.

## 5.1 Configure shared memory segment and synchronization semaphores

For the communication of events between the *IIO Event Monitor* application and the main application, the use of shared memory functionality and proper synchronization between both applications is crucial.

The desired space in shared memory has been reserved using the shm library and is identified by a specific memory key in the main application. The reserved memory space matches the space occupied by a struct that acts as a wrapper for all the variables intended to be exchanged through shared memory, which includes event information and semaphores. After enabling access to shared memory for external processes, the variables included in the wrapper and the semaphores have been initialized, we now proceed to configure shared memory in the *IIO Event Monitor* application.

The access to the reserved memory segment has been set up upon application startup, and this application will release the semaphores as necessary for the alarm thread to operate with the data provided by the *IIO Event Monitor*.

Using this configuration, it is possible to collect data from an application external to the main application, respecting the execution flow initially proposed.

## 5.2 Detection and handling sensors alarms functions

When detecting alarms, as seen, the magnitudes related to the Xilinx chip are obtained through an external application, while alarms related to I<sup>2</sup>C or GPIO devices, or link status are based on flags located in registers or global variables. Therefore, the detection process relies on readings from relevant registers, similar to monitoring threads but with a much shorter period. That is why I have relied on the work done in the functions used in the monitoring thread to develop the alarm detection functions, taking into account the devices that use masks to avoid repeatedly reporting the same alarm based on the datasheets of the devices, and for those that do not have this mechanism, such as the SFPs, a mask has been declared in the application to mimic this mechanism.

Regarding the handling of alarms, based on the activated bit or the information from the *IIO Event Monitor* obtained by AMS, we can extract, according to the documentation of the devices, the type of event, its severity and the channel if the device has different channels. Then it depends on whether the event has been triggered by a change of state or a magnitude, since magnitude events also have direction depending on whether they have exceeded the upper threshold or are below the lower threshold.

In addition, I have considered it important to obtain the value of the trigger for activating the event. In the case of it being a magnitude alarm, it will be read as done in the monitoring thread, while if it is a status alarm the value will be “ON” or “OFF”.

## 5.3 Parse alarms data into JSON string and send it to the DAQ

Once the alarm event has been detected and the alarm thread gathered the necessary information of the event, the final handling step is to report the alarm to the DAQ so that they are aware of the triggered event and can act to correct or mitigate the consequences of the event.

The communication between the alarm threads and the receiving module has been established in the same way as in the monitoring thread. Both alarm threads become the Publishers in the communication meanwhile the DAQ or the corresponding receiving application act as Subscribers. The sockets used have been differentiated according to the information they carry by port number, so clearly a different socket is used, but with the same configuration in the alarm and monitoring threads.

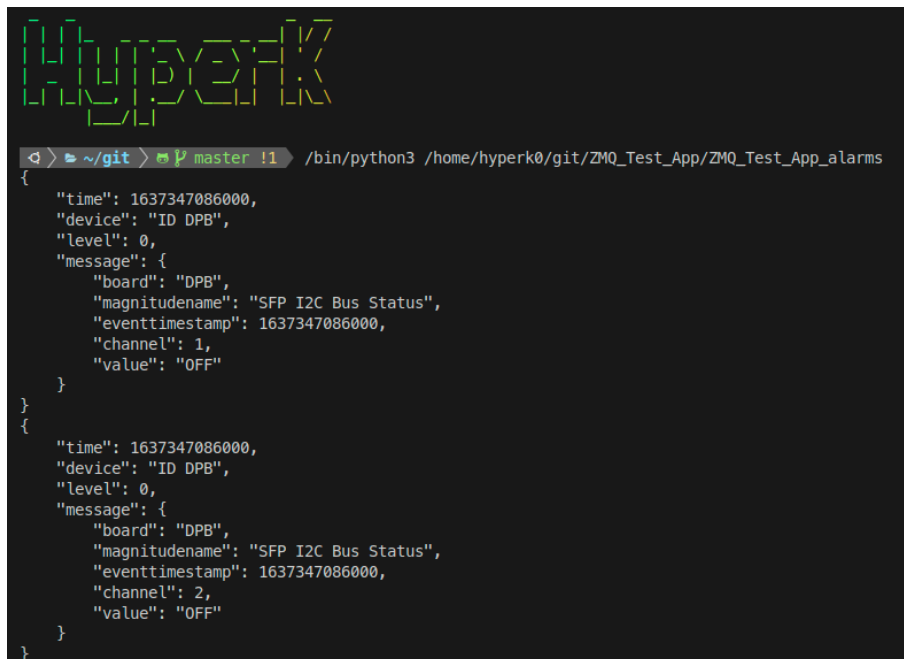
**Listing 5.1: Alarm JSON format example**

```
1 {  
2   "time": 1637343635012,  
3   "device": "ID DPB",  
4   "level": 1,  
5   "message": {
```

```
6   "board": "DPB",
7   "magnitudename": "SFP RX Power",
8   "eventtype": "falling",
9   "eventtimestamp": 1637343635000,
10  "channel": 0,
11  "value": 0
12 }
13 }
```

As it can be seen in the previous listing, the alarm JSON string format is different from the monitoring JSON format. The alarm JSON format contains two timestamps one to know when the message has been sent and the other to indicate when the event occurred. In addition, the message indicates the device that has triggered the alarm, in our case it will always be the ID DPB, the level which will be 0 if the event is critical or other than 0 if it is only a warning and the message which includes the captured and relevant information of the event. The event type and channel fields will appear in the message only when necessary.

The created message will be sent through the alarm socket after being successfully validated by the *json-schema-validate* application according to a defined schema, taking into account the desired message format.



```
Hyperk0
~/git > master !1 /bin/python3 /home/hyperk0/git/ZMQ_Test_App/ZMQ_Test_App_alarms
{
  "time": 1637347086000,
  "device": "ID DPB",
  "level": 0,
  "message": {
    "board": "DPB",
    "magnitudename": "SFP I2C Bus Status",
    "eventtimestamp": 1637347086000,
    "channel": 1,
    "value": "OFF"
  }
}
{
  "time": 1637347086000,
  "device": "ID DPB",
  "level": 0,
  "message": {
    "board": "DPB",
    "magnitudename": "SFP I2C Bus Status",
    "eventtimestamp": 1637347086000,
    "channel": 2,
    "value": "OFF"
  }
}
```

Figure 5.2: JSON strings received in Python application after triggering alarms

## Chapter 6

# Command handling thread development

The last thread to be developed is the command handling or *slow control* thread, which consists on receiving commands directly from the DAQ and verifying that it is an existing command. In that case, it acts accordingly with the received command and sends the corresponding response to the DAQ. If the command is invalid an “Invalid command” reply is sent to the DAQ servers.

Unlike the monitoring and alarm threads, the command thread has a different ZeroMQ communication configuration. This thread requires bi-directional communication in order to receive commands and send the corresponding response, so it is not possible to use the Publisher-Subscriber model previously used. This is why it has been decided to use a ZeroMQ Request-Reply communication model.

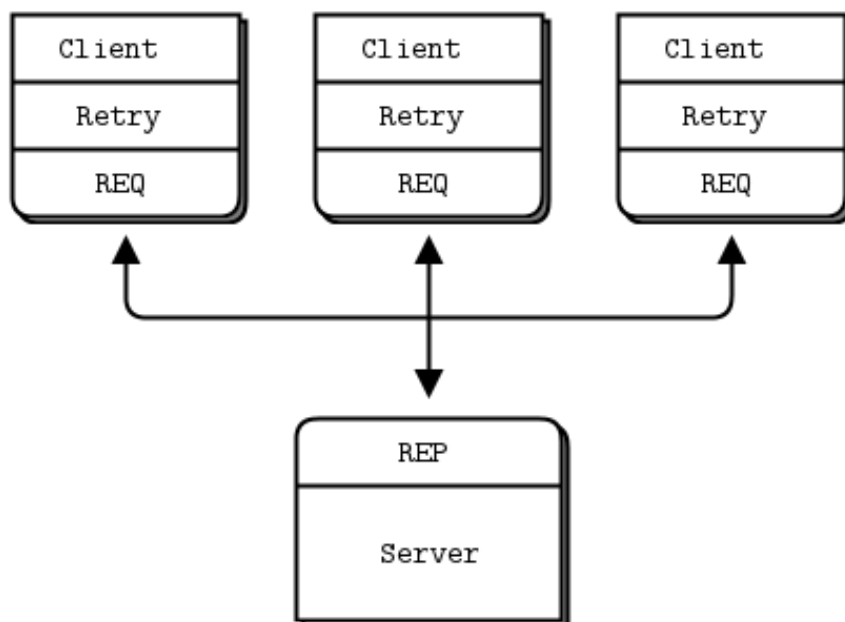


Figure 6.1: Request-Reply pattern [29]

This Request-Reply model consists of, as the name suggests, Request sockets requesting the Reply socket, and the requesting socket waits for a response. While the Reply socket is initially waiting for a message to be sent and once it receives it, it sends a response to the message sender socket and waits again for a request. A bidirectional communication that follows a Request-Reply flow suitable for command line operation. Furthermore, this model can be implemented with several clients on the same server since ZeroMQ itself implements a queuing system based on the Round-Robin algorithm to organize the requests and then directs the response to the corresponding requester [30].

In this case, I have defined the command thread socket as Reply and the test Python application used as Request as the function of the DAQ servers. With this, the command thread waits to receive a message to process it and answer as it should.

## 6.1 Parse commands from the DAQ into JSON string for processing

The message received by the slow control thread is a JSON string which among other keys includes the value of the message which is the command to be handled. So the command thread must extract the command string and once extracted the string is divided into the command parameters (see tables A.1 and A.2 for the list of commands implemented in the annex), which can range from 3 to 5 words, in order to treat each command parameter independently.

Once the command is segmented into different strings using the `strtok(3)` function, a JSON string is created with the fragments of the received command, to apply the *json-schema-validate* application and by means of another schema designed by my project colleague to verify that the command is valid. If the command is not correct, an “Invalid command” response is sent.

**Listing 6.1: Command request JSON format example**

```
1 {  
2   "msg_id": 0,  
3   "msg_time": "2021-11-19T17:54:30.691Z",  
4   "msg_type": "Command",  
5   "msg_value": "READ DPB TEMP PCB",  
6   "uuid": "931fbc9d-b2b3-c248-87d6ae33f9a62"  
7 }
```

It can be seen in the above listing the received JSON string format, where the command thread must retrieve “msg\_id” and “msg\_value” key values. The “msg\_id” value will be indicated in the command reply and “msg\_value” is the command string itself.

## 6.2 Define the command cases and develop functions to handle each case

As previously indicated, a list of commands to be implemented has been defined, so far only referring to the DPB, and a function has been declared to deal with the different commands depending on which board it is addressed to. As so far only the commands for the DPB are known, only this function has been developed.

The DPB handling function retrieves the received fragmented command and differentiates the received parameter between SFP and the rest, because SFPs have more measurement magnitudes, and the rest of the parameters allow to define general cases and save code. Then, cases are differentiated by the magnitude and finally differentiated by the desired operation, either READ or SET, so calls the corresponding function to carry out the received command.

In case of failure in the reading or setting operation, a JSON response will be sent to the DAQ indicating “ERROR: READ operation not successful” or “ERROR: SET operation not successful” respectively.

**Listing 6.2: Command reply JSON format example**

```

1 {
2   "msg_id": 0,
3   "msg_time": "2021-11-19T17:54:30.691Z",
4   "msg_type": "Command reply",
5   "msg_value": 38.5,
6   "uuid": "931fac9d-b2b3-c248-87d6ae33f9a62"
7 }
```

As can be seen in the previous listing, the command reply format is exactly the same as for the command request message. The “msg\_id” is retrieved from the corresponding command request, and the “msg\_type” value will always be “Command reply”. The message time is presented with the format: “<year>-<month>-<day>T<hour>:<min>:<second>.<ms>Z” and the UUID is generated randomly for every command reply message.

Regarding the message value key value, it depends on the command operation. If the command operation is SET and it is successful, the message value will be “OK” indicating everything went smoothly. Whereas if the command operation is READ and it was successful, the message value will be “ON” or “OFF” if a status has been read, or if a magnitude has been read, the message value will be the value of the magnitude read.

To carry out the READ commands I have used the read functions previously developed for the monitoring thread, while for the SET commands, I have developed new functions with a very similar structure to the read ones but following the writing flow either through I<sup>2</sup>C, GPIO, or writing to registers through the sysfs file system.



## Chapter 7

# Develop manufacturing test software

The software designed for the slow control applications of the DPB will not be used solely for the purpose of real-time component monitoring or alarming, as much of this software can be very useful for manufacturing tests for new prototypes and the final version of the DPB. This is why in this chapter we will deal with the adaptation of the software designed in C programming language to Python as a library to be able to use it from the Robot framework, and how we intend to design the test cases in Robot to be able to automate them.

### 7.1 Adapt previously developed software for manufacturing test

After including the Robot framework in the PetaLinux device tree, in order to define the desired test cases and automate them, it is necessary to adapt the functions designed in C to Python. For this, the Python ctypes library allows you to make use of C data structures and functions in Python. It also allows wrapping complete C libraries in Python.

Being aware of the functionalities of this library, I have used the library project wizard of Vitis IDE to encapsulate all the functions designed in the DBP2\_App application except for the declaration of the threads and the main in a library importable by other applications. It should be noted that minimal modifications have been made to the functions by eliminating any semaphore as it will not be necessary and parameterizing the path of the *IIO Event Monitor* application. Once the library has been successfully compiled, the Vitis IDE itself provides the .so file which comprises the shared library. We only need to insert this .so file as well as the headers of the external libraries used to develop the application functions in the “/usr/lib/” directory of the DPB OS so that the functions can be executed on the board and include these files in the corresponding path so that the compiler detects these functions. The last step in the adaptation is to transcribe the functions from C language to Python language, indicating the type of arguments that each function requires. To do this, ctypes provides us with a table of basic C data structures, their corresponding structure in ctypes, and how the Python Interpreter assimilates them.



| <b>ctypes type</b> | <b>C type</b>                          | <b>Python type</b>       |
|--------------------|--|--------------------------|
| c_bool             | _Bool                                  | bool (1)                 |
| c_char             | char                                   | 1-character bytes object |
| c_wchar            | wchar_t                                | 1-character string       |
| c_byte             | char                                   | int                      |
| c_ubyte            | unsigned char                          | int                      |
| c_short            | short                                  | int                      |
| c_ushort           | unsigned short                         | int                      |
| c_int              | int                                    | int                      |
| c_uint             | unsigned int                           | int                      |
| c_long             | long                                   | int                      |
| c_ulong            | unsigned long                          | int                      |
| c_longlong         | __int64 or long long                   | int                      |
| c_ulonglong        | unsigned __int64 or unsigned long long | int                      |
| c_size_t           | size_t                                 | int                      |
| c_ssize_t          | ssize_t or Py_ssize_t                  | int                      |
| c_time_t           | time_t                                 | int                      |
| c_float            | float                                  | float                    |
| c_double           | double                                 | float                    |
| c_longdouble       | long double                            | float                    |
| c_char_p           | char* (NULL terminated)                | bytes object or None     |
| c_wchar_p          | wchar_t* (NULL terminated)             | string or None           |
| c_void_p           | void*                                  | int or None              |
| POINTER()          | <data>*                                | class                    |
| Structure          | struct                                 | class                    |

**Table 7.1: Basic data structures in ctypes, C y Python**

Considering the conversion of data types from C to Python using ctypes, I have designed a Python script called *Init\_Robot*, which will be executed to start the Python library. This script defines the special data types such as I<sup>2</sup>C devices or JSON objects and then passes the C functions to Python by defining the argument and return types of the function. The transcription of C functions into Python is done automatically by using the function headers from the .h file of the shared library, detecting the return data type and arguments and assigning their equivalence in ctypes by looking at a dictionary that has been designed. In addition, in the processing of the arguments of each function it is taken into account whether the argument is a pointer to define it as such or not.

**Listing 7.1: Ctypes dictionary of equivalences**

```

1 ctype_map = {
2     'int': ctypes.c_int,
3     'float': ctypes.c_float,
4     'char': ctypes.c_char,
5     'struct DPB_I2cSensors':DPB_I2cSensors,
6     'struct I2cDevice':I2cDevice,
7     'uint16_t': ctypes.c_uint16,
8     'uint8_t': ctypes.c_ubyte,
9     'uint64_t': ctypes.c_uint64,
10    'json_object':JsonObject,
11    'int ': ctypes.c_int,
12    'float ': ctypes.c_float,
13    'char ': ctypes.c_char,
14    'struct DPB_I2cSensors ':DPB_I2cSensors,
15    'struct I2cDevice ':I2cDevice,
16    'uint16_t ': ctypes.c_uint16,
17    'uint8_t ': ctypes.c_ubyte,
18    'uint64_t ': ctypes.c_uint64,
19    'json_object ':JsonObject,
20    'void':None,
21    # Add more types in case it is necessary
22 }
```

**Listing 7.2: Ctypes C function definition process**

```

1 <library_name>.<function_name>.argtypes = [ctypes.<argN_data_type> ...]
2 # N being the number of arguments of the function
3 <library_name>.<function_name>.restypes = ctypes.<return_data_type>
```

Once all the C functions have been defined in Python, the library can be considered initialized, but in addition to this, the script takes care of starting the sockets, devices, and processes to be used and obtains the GPIO base address just as the main thread of the application does. With this, I manage to start or obtain all the necessary resources to run the tests every time the library is initialized.

## 7.2 Automation of the tests using Robot framework

The definition of tests in the Robot framework is the most tedious task when testing a DUT with this platform, as it requires an initial analysis of the test cases we want to define. Afterward, we must decide whether to orient each case towards a data-driven test or a keyword-driven test, based on which type of test may be more intuitive for an arbitrary user.

Since I currently only have the DPB and I lack elements that could help me to develop test cases such as measurement probes, it has been decided to develop, for the time being, tests that can be performed only with the DPB. Therefore, tests have been implemented to verify the correct initialization of the library and its components, to check the correct value of any magnitude that has a stipulated value or range of values, and to probe the status of Ethernet interfaces, as well as being able to choose the active interface and perform connectivity tests. In addition, test cases have also been implemented to verify the correct operation of the GPIOs using the pins associated

with the SFPs to detect the SFPs connected and verify the correct reading by comparing via I<sup>2</sup>C the value of certain GPIO pins with their corresponding value stored in the SFP EEPROM.

Apart from all these test cases more focused on the correct functioning of the components, tests have also been implemented to force the activation of alarms and the use of commands to verify the operation of the software and communication through ZeroMQ.

The library containing all the functions for performing test cases has been designed by defining the functions with the keywords that should be associated with that function (using underscores as spaces), even though they can be modified with a label, and the number of arguments has been minimized to make the functions as intuitive and guided as possible. Then, when it comes to verifying the cases, I have designed specific functions for cases in which a closed result is expected, these will be the keyword-driven tests that will only require arguments about which component you want to perform the test on and the action to be performed. On the other hand, data-driven tests will mainly be those dedicated to reading values from a component and the user must be told what the value and expected performance of each component is to facilitate the test. Both cases will present an assertion in the event of a failure that will appear in the final Robot framework report.

```

Check Initialization
  Check ZMQ Initialization
  Check IIO Monitor
  Check GPIO Base Address
  Check I2C Devices Initialization
  Check SFP Presence
    
```

Figure 7.1: Example of keyword-driven test

|  |  |
|--|--|
| <pre> Bus voltages [Template] Bus Voltage Check Template SFP0      3.3 SFP1      3.3 SFP2      3.3 SFP3      3.3 SFP4      3.3 SFP5      3.3 12V       12 3V3       3.3 1V8       1.8           </pre> | <pre> Bus Voltage Check Template [Arguments]  \${device}  \${expected_voltage} Get Bus Voltage  \${device} Result Should Be Within Tolerance Range  \${expected_voltage}  10%           </pre> |
|--|--|

Figure 7.2: Example of data-driven test

As has been done with the code developed in C, it is also planned to attach to the submission of this TFG more detailed documentation of the functions developed in Python in HTML format extracted using the Sphinx tool. The main HTML file will be “index.html” .

It should be noted that although the test library will run on the DPB, Robot Framework will run from the device that is being used to test the board, in my case a computer, so to communicate both devices and run the test library on the board it must be included in the Robot Remote Server library so tests can be automated by initializing the library in the board and using the Robot framework Remote library on the computer to execute the relevant tests and obtain the execution report of

the same in a HTML format file. In addition, we have the possibility to customize the result of the report through the assertions implemented in the case verification functions, including images, graphs, or another type of data that can be used in the testing process.

The screenshot displays a Robot Framework Test Report with a green header. The report title is "Test Report" and it was generated on 20240520 at 13:44:22 UTC+02:00, 3 seconds ago. The "Summary Information" section shows that all tests passed, with documentation indicating a test suite running with the DPB Test Library. The start time is 20240520 13:44:17.996, the end time is 20240520 13:44:22.991, the elapsed time is 00:00:04.995, and the log file is log.html. The "Test Statistics" section contains three tables: "Total Statistics" for "All Tests" (1 total, 1 pass, 0 fail, 0 skip, 00:00:00 elapsed), "Statistics by Tag" for "No Tags", and "Statistics by Suite" for "Test" (1 total, 1 pass, 0 fail, 0 skip, 00:00:05 elapsed). The "Test Details" section includes filters for "All", "Tags", "Suites", and "Search", and input fields for "Suite:", "Test:", "Include:", and "Exclude:", along with "Search", "Clear", and "Help" buttons.

Figure 7.3: Basic example of successful Robot test report

In addition to the report, a log file is generated that provides much more detailed information on the execution of the tests and their results.



## **Part V**

# **Conclusions and future work**



# Chapter 1

## Lessons learned

The Hyper-Kamiokande project is a long-term effort to take a step forward in neutrino detection with the construction of the new high-performance detector. In this TFG I have sought to contribute to the development of this new infrastructure, specifically in the submerged electronics in the vessels of the ID of HKK, by learning about the tools available in the DPB and then using them to develop software that performs monitoring and slow control tasks on the board. In addition to reusing the software designed to lay the basis of manufacturing tests for the new prototypes of the DPB. Thus fulfilling the project-specific goals related to the monitoring of electronic systems status.

Meeting these objectives has involved encountering problems, among the most notable, I would mention finding the bug in the “xilinx-ams” driver and the debugging process of the application using Valgrind to identify and eliminate memory leaks that were causing segmentation faults.

After 4 month-time of work and fulfilling the objectives proposed at the beginning of the dissertation, I have been able to draw the following conclusions from each of them:

- **Develop DPB software :** One of the biggest challenges I initially faced when developing the application was the fact that I had to monitor each of the devices available in the DPB, as this implied using different communication protocols when acquiring or transmitting data through the application. The datasheet of the components and the schematics of the DPB prototype have been of great help in expanding my knowledge of the various types of communication used and being able to implement them in software for the desired applications.
- **Learn how to work in embedded environments :** Despite having basic knowledge of embedded systems, working on them is not as straightforward as working on any other system. Since I have worked with a lightweight version of Linux as an OS, I have been able to appreciate the differences compared to a more complete Linux image, especially in aspects such as debugging, which is much less detailed. Additionally, the DPB must reserve a significant portion of RAM and processing capacity for the data captured, which has led me to learn methods of memory profiling and optimization that are compatible with embedded systems, such as using the Valgrind tool to reduce application processing capacity consumption and detect and prevent memory leaks.



- **Program software in Linux** : As expected, programming on a Linux-based operating system, despite it being a lightweight version, has greatly aided me in developing the application. The drivers, libraries, and the hierarchical directory structure that Linux works with have simplified the implementation of desired functionalities, allowing me to focus solely on their implementation rather than internal function processes. However, this level of abstraction also entails the problem of encountering a bug or issue, as happened to me with the “xilinx-ams” driver, and debugging a driver is not a trivial task. Fortunately, my project partner and I managed to identify and fix the flaw through a rigorous debugging process.
- **Develop *slow control* system** : The development of software that performs *slow control* tasks has been a nice challenge for me when it comes to differentiating the different tasks and deciding the priority of each type using multi-threaded programming, in addition to needing to use all the available resources such as drivers, I<sup>2</sup>C buses and GPIO pins or Linux files to keep track of the status of the board. But above all, it has been a challenge for me to develop the command processing thread due to the precision required by the process from the moment you receive the command until you perform the corresponding action based on the command. Additionally, I have personally experienced the importance of synchronization using semaphores in a multi-threaded software when shared resources are used among different processes.
- **Create data structures** : Since the format of the JSON string had already been determined by the DAQ for each type of message, I only needed to correctly handle the json-c library tools to compose the JSON string while respecting the proposed format and hierarchy. Afterwards, I verify the message structure using model schemata. Emphasizing the verification of the message structure was decided upon, as mismatch from formats between message sender and receiver makes communication between the two impossible due to the absence of a common communication protocol.
- **Manage alarm systems** : Alarm systems are a key tool in slow control applications for reporting operating conditions outside the specified or critical ranges. Having defined two threads for alarms, one dependent on interruptions captured by the *IIO Event Monitor* and another dependent on readings from files or I<sup>2</sup>C buses or GPIO pins, allows for prioritizing this type of alarms and comfortably defining how they should be reported to the DAQ via the ZeroMQ alarm socket.
- **Test preparation and automation** : The development of automated test cases for the manufacture of prototypes of the DPB has been initially a bit tedious since the Robot platform required adapting the previously designed C software to Python using the ctypes library. However, this adaptation using ctypes, despite being somewhat tedious, has allowed me to reuse the functions developed in C and avoid having to redo them in Python. Furthermore, it presented the initial difficulty of knowing how to interpret the operation of the platform itself, since it works with keyword-driven and data-driven tests so that these tests are understandable in human language. This implies abstracting the test as much as possible on your part to facilitate the work of the person in charge of carrying out the manufacturing test and thus try to reduce possible human errors in this process and avoid unnecessary consumption of resources.

## Chapter 2

### Future work

As previously mentioned, the HKK project is long-term, and there is still enough work in the electronics section that could take 3 years to complete. Regarding the slow control software on which my TFG is based, it cannot be considered definitive by any means, but a solid basis for the final software. This is due to the fact that during the development of this TFG, I only had the DPB of the modules that will be inside the vessel and the different methods of communication with the DAQ had not been stipulated.

However, in the near future, the DAQ has informed us that they will provide us with a library that will implement communication between DPB and their servers using a structure similar to the one used in the application developed in this TFG, based on ZeroMQ sockets and including Multicast for non-priority sockets. Additionally, we will have access to the digitizers and HV and LV boards to carry out slow control tasks of every module that will be inside the vessel and testing these tasks are carried out properly. Therefore, the following points can be followed to continue the work done in this TFG:

- **Redefine communication sockets** : Adapt the already established socket configuration to the slight modifications proposed by the DAQ to follow a common communication model commonly agreed by the project members.
- **Develop *slow control* software for the remaining modules in the vessel** : Expand the existing software to perform the slow control tasks, including monitoring the status of HV, LV boards and digitizers and controlling them by the DAQ via command. Extend the slow control functionalities already developed for the DPB to the remaining modules, since so far only the status monitoring of the Aurora links has been measured via GPIO due to the absence of the remaining modules.
- **Extend the use of the Robot Framework to all the system in the vessel** : The test cases designed to date in the Robot framework environment have been mainly focused on verifying the correct functioning of the internal communication protocols of the DPB and the SFPs, ensuring that the manufacturing has been correct, in addition to verifying the operation of part of the designed software. Due to the practicality of the Robot framework, it would be convenient to expand the catalogue of tests defined to verify the communication between different modules and even not simply limit Robot framework to manufacturing tests but

also use it to define tests to verify the functionality of the software and its compatibility with the other modules.

As we come to the end of this TFG, I simply have to give my most sincere thanks to the reader who has reached this point for taking the necessary time to read this thesis.

# Bibliography

- [1] I.C. Baianu et al. *Fundamentals of Physics and Nuclear Physics*. Free Software Foundation Inc., Nov. 2002.
- [2] Laurie M. Brown. “The idea of the neutrino”. In: *Phys. Today* 31.9 (1978), pp. 23–28. doi: 10.1063/1.2995181.
- [3] The Hyper-Kamiokande Collaboration. *Hyper-Kamiokande Project*. Source Link. 2015.
- [4] Hyper-Kamiokande Proto-Collaboration et al. *Hyper-Kamiokande Design Report*. 2018. arXiv: 1805.04163 [physics.ins-det].
- [5] Alejandro Gómez Gambín. “Discussion and preparation of a FPGA-based hardware platform with embedded operative system for data processing tasks inside the neutrino detector Hyper-Kamiokande”. MA thesis. 2023.
- [6] Alejandro Gómez Gambín. “Prototype elaboration of the data processing block in a SOM board with redundant boot, Aurora protocol and timing synchronization support for Hyper-Kamiokande detector”. MA thesis. 2024.
- [7] Tessolve. *Five Reasons to Use System on Modules (SoM) in Embedded System Design*. Source Link. n.d.
- [8] Proculus Technologies. *An In-Depth Guide to System on Modules (SoMs): Everything You Need to Know*. Source Link. n.d.
- [9] Advanced Micro Devices. *Ultrafast Embedded Design Methodology Guide*. Source Link. n.d.
- [10] Xilinx Inc. *UltraScale MPSoC Technology*. Source Link. n.d.
- [11] Texas Instruments. *Power Supply Design Seminar*. Source Link. n.d.
- [12] Advanced Micro Devices. *GPIO Module*. Source Link. n.d.
- [13] Michael K Johnson and Erik W Troan. *Linux application development*. Addison-Wesley Longman Publishing Co., Inc., 1998.
- [14] Gene Sally. *Pro Linux embedded systems*. Apress, 2010.
- [15] Patrick Mochel. “The sysfs filesystem”. In: *Linux Symposium*. Vol. 1. The Linux Foundation San Francisco, CA, USA. 2005, pp. 313–326.
- [16] Advanced Micro Devices. *Vivado Design Suite Overview*. Source Link. n.d.
- [17] Advanced Micro Devices. *Vitis™ Embedded Development: Domain Overview Page*. Source Link. 2022.

- [18] Advanced Micro Devices. *Vitis™ Application Acceleration Development: Platform Types*. Source Link. n.d.
- [19] Robot Framework. *Robot Framework User Guide*. Source Link. n.d.
- [20] NRE Labs. *Using Robot Framework for Automated Testing*. Source Link. 2018.
- [21] Texas Instruments. *INA3221 Triple-Channel, High-Side Measurement, Shunt and Bus Voltage Monitor with I2C Interface Datasheet*. Source link. n.d.
- [22] Microchip Technology Inc. *PIC32 Family Reference Manual*. Source link. n.d.
- [23] Broadcom. *AV02\_3012EN\_DS\_AFBR\_571xZ*. Source link. Aug. 2018.
- [24] AMD. *PL System Monitor Specifications*. Source Link. n.d.
- [25] The Linux Kernel Organization. *IIO Subsystem Documentation*. Source link. n.d.
- [26] AMD. *DC Characteristics Over Recommended Operating Conditions*. Source Link. n.d.
- [27] JSON Schema Organization. *JSON Schema Specification*. Source link. n.d.
- [28] ZeroMQ. *ZeroMQ Guide*. Source Link. n.d.
- [29] ZeroMQ. *ZeroMQ Guide*. Source Link. n.d.
- [30] ZeroMQ. *ZeroMQ Socket API*. Source Link. n.d.

**Part VI**

**Annexes**



## Annex A

### Additional Listings

| Command | Board | Magnitude | Parameter                                     | Write-Value |
|---------|-------|-----------|---|-------------|
| SET     | DPB   | STATUS    | SFP0 / SFP1 /<br>SFP2 / SFP3 /<br>SFP4 / SFP5 | ON/OFF      |
| SET     | DPB   | STATUS    | ETH0 / ETH1                                   | ON/OFF      |
| SET     | DPB   | TEMP      | PCB   | number      |
| SET     | DPB   | TEMP      | FPGA  | number      |
| SET     | DPB   | TEMP      | FPDCPU  | number      |
| SET     | DPB   | TEMP      | LPDCPU  | number      |
| SET     | DPB   | VOLT      | FPDCPU  | number      |
| SET     | DPB   | VOLT      | LPDCPU  | number      |
| SET     | DPB   | CURR      | SFP0 / SFP1 /<br>SFP2 / SFP3 /<br>SFP4 / SFP5 | number      |
| SET     | DPB   | CURR      | 12V   | number      |
| SET     | DPB   | CURR      | 3V3   | number      |
| SET     | DPB   | CURR      | 1V8   | number      |

**Table A.1: Setting DPB command list**



| <b>Command</b> | <b>Board</b> | <b>Magnitude</b> | <b>Parameter</b>                              |
|----------------|--------------|------------------|---|
| READ           | DPB          | STATUS           | SFP0 / SFP1 /<br>SFP2 / SFP3 /<br>SFP4 / SFP5 |
| READ           | DPB          | STATUS           | ETH0 / ETH1                                   |
| READ           | DPB          | TEMP             | SFP0 / SFP1 /<br>SFP2 / SFP3 /<br>SFP4 / SFP5 |
| READ           | DPB          | TEMP             | PCB   |
| READ           | DPB          | TEMP             | FPGA  |
| READ           | DPB          | TEMP             | FPDCPU  |
| READ           | DPB          | TEMP             | LPDCPU  |
| READ           | DPB          | VOLT             | FPDCPU  |
| READ           | DPB          | VOLT             | LPDCPU  |
| READ           | DPB          | CURR             | SFP0 / SFP1 /<br>SFP2 / SFP3 /<br>SFP4 / SFP5 |
| READ           | DPB          | CURR             | 12V   |
| READ           | DPB          | CURR             | 3V3   |
| READ           | DPB          | CURR             | 1V8   |
| READ           | DPB          | TXPWR            | SFP0 / SFP1 /<br>SFP2 / SFP3 /<br>SFP4 / SFP5 |
| READ           | DPB          | RXPWR            | SFP0 / SFP1 /<br>SFP2 / SFP3 /<br>SFP4 / SFP5 |

**Table A.2: Reading DPB command list**