UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Telecommunications Engineering

Prototype elaboration of the data processing block in a SOM board with redundant boot, Aurora protocol and timing synchronization support for Hyper-Kamiokande detector

Master's Thesis

Master's Degree in Telecommunication Engineering

AUTHOR: Gómez Gambín, Alejandro

Tutor: Herrero Bosch, Vicente

Cotutor: Mora Mas, Francisco José

Cotutor: Ballester Merelo, Francisco José

ACADEMIC YEAR: 2023/2024

Greetings

This Master's Thesis wouldn't have been possible without the support received in the last year. I want to express my most sincere greetings to all people who have helped me.

To Vicente Herrero for always been so supportive and receiving me in his office whenever I made any advance or had any doubt. His technical knowledge about the analog electronics of the digitizer has been very valuable for this TFM.

To Jose Francisco Toledo for his knowledge about PCBs that has helped me to make the correct decisions when designing the firmware to use the hardware on the PCB correctly.

To Raul Esteve for helping me to catch up with the digitizer state and the Hyper-Kamiokande in general when I joined the project.

To Francisco Ballester for supporting me in the slow control tasks and doing all the paperwork to procure components that a project of this magnitude needs.

To Francisco José Mora for being an excellent academic and project manager, ensuring that I didn't lack any piece of hardware or software to continue with this TFM and beyond.

To Sergio Buedo and Gustavo Sutter, counting with two Xilinx FPGA experts that have given me very useful advises has been crucial whenever I got stuck.

To everyone who has accompanied me from the telecommunications degree to this master and have been with me in the process of learning in this beautiful career.

To all the newcomers to the UPV that I had the fortune to meet in this master. Being able to share time with you let be in the classroom or studying together has been a very enriching experience for me.

To the rest of the Hyper Kamiokande team who has provided me with the necessary documentation to understand their involvement and how their work is related to ours.

Resumen

El detector Hyper-Kamiokande (HK) está diseñado para estudiar los fenómenos relacionados con los neutrinos y la desintegración de protones. Para detectar estos sucesos, se utilizan fotomultiplicadores (PMT) que permiten capturar los fenómenos de Luz de Cherenkov. En el marco de este proyecto, la Universidad Politécnica de Valencia es la encargada de diseñar el DPB o Bloque de Procesado de Datos que funcionará de interfaz para transmitir los datos desde el front-end, donde se alojan los PMT, al exterior.

Este Trabajo de Fin del Máster Universitario en Ingeniería de Telecomunicación trata sobre el desarrollo de un prototipo para el DPB sobre una tarjeta System On Module(SOM) de Xilinx que dará soporte a las funciones que deberá realizar el DPB entre las que se encuentran comunicación mediante protocolo Aurora, sistema de arranque fiable y temporización y sincronización.

Resum

El detector Hyper-Kamiokande (HK) està dissenyat per a estudiar els fenòmens relacionats amb els neutrins i la desintegració de protons. Per a detectar estos successos, s'utilitzen fotomultiplicadors (PMT) que permeten capturar els fenòmens de Llum de Cherenkov. En el marc d'este projecte, la Universitat Politècnica de València és l'encarregada de dissenyar el DPB o Bloc de Processament de Dades que funcionarà d'interfície per a transmetre les dades des del front-end, on s'allotgen els PMT, a l'exterior.

Este Treball de Fi del Màster Universitari en Enginyeria de Telecomunicació tracta sobre el desenvolupament d'un prototip per al DPB sobre una targeta System On Module(SOM) de Xilinx que donarà suport a les funcions que haurà de realitzar el DPB entre les quals es troben comunicació mitjançant protocol Aurora, sistema d'arrancada fiable i temporització i sincronització.

Abstract

The Hyper-Kamiokande (HK) detector is designed to study phenomena related to neutrinos and proton decay. To detect these events, photomultipliers (PMT) are used to capture Cherenkov light phenomena. In the framework of this project, the Polytechnic University of Valencia oversees the designing of the DPB or Data Processing Block that will act as an interface to transmit the data from the front-end, where the PMTs are housed, to the outside.

This Final Project of the master's degree in Telecommunication Engineering deals with the development of a prototype for the DPB on a Xilinx System On Module (SOM) card that will support the functions to be performed by the DPB, among which are communication via Aurora protocol, reliable boot system and timing and synchronization.

Table of Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence.

APU Application Processing Unit.

ASIC Application Specific Integrated Circuit.

ATM Analog Timing Modules.

BSP Board Support Package.

CAN Controller Area Network.

CLB Configurable Logic Blocks.

CPU Central Processing Unit.

DAQ Data Adquisition Module.

DDR Double Data Rate.

DPB Data Processing Block.

ECC Error Correction Code.

EMAC Ethernet Media Access Controller.

EMIO External Multiplexed Input Output.

EvB Evaluation Board.

FMC FPGA Mezzanine Card.

FPGA Field Programmable Gate Array.

FSBL First Stage Boot Loader.

GEM Gigabit Ethernet MAC.

GPIO General Purpose Input Output.

GTH Gigabit Transceivers Type H.

GTY Gigabit Transceivers Type Y.

HD High Density.

HDL Hardware Description Language.

HKK Hyper-Kamiokande.

HP High Performance.

HV	High Voltage Module.
i3M	Institute for Molecular Imaging Technologies.
ID	Inner Detector.
IDE	Integrated Development Environment.
JTAG	Join Test Action Group.
KET	Key Enablement Technologies.
LUT	Look Up Table.
LV	Low Voltage Module.
LVDS	Low Voltage Differential Signal.
MDIO	Management Data Input Output.
MGT	Multi Gigabit Transceivers.
MIO	Multiplexed Input Output.
MoU	Memorandum of Understanding.
MPSoC	Multi Processing System on Chip.
MUISE	Master in Electronic Systems Engineering.
NAS	Network Access Storage.
NVM	Non Volatile Memory.
OD	Outer Detector.
OS	Operative System.
PC	Personal Computer.
PCB	Printed Circuit Board.
PCIe	Peripheral Component Interconnect Express.
PL	Programmable Logic.
PLL	Phase Locked Loop.
PMT	Photomultiplier tube.
PMU	Platform Management Unit.
PS	Processing System.
PXE	Preboot eXecution Environment.
QBEE	QTC Based Electronics over Ethernet.

QSPI Quad Serial Peripheral Interface.

QTC Charge to Time Converter.

Rn Radon.

RPU Real Time Processing Unit.

SATA Serial Advanced Technology Attachment.

SDK Source Development Kit.

SFP Small Form Factor Hot Pluggable.

SKK Super-Kamiokande.

SoC System on Chip.

SOM System On Module.

TDC Time-to-digit converter.

TFM Final Master Thesis.

TFTP Trivial File Transfer Protocol.

TRL Technology Readiness Level.

UAM Universidad Autónoma de Madrid.

UART Universal Asynchronous Receiver-Transmitter.

UPV Polythecnic University of Valencia.

UTC Universal Coordinated Time.

XSCT Xilinx Software Command Tool.

Part I

Introduction: The HKK Project

Chapter 1

Motivation

Electronics is one of the fundamental pillars where development of modern technologies stand, allowing also for unparalleled advances in several areas of knowledge providing new and more accurate ways to measure phenomena to assess the hypothesis that these areas of knowledge make.

One of those fields, often seen as a very theoretical one is Physics. Nothing further from the truth, this field needs a lot of experiments to prove their hypothesis true and the further we dive into the mysteries in our universe, the more difficult it becomes to measure the required magnitudes to perform the hypothesis validations.

Thus, this Final Master Thesis (TFM) of the Master in Telecommunications Engineering is a contribution in the development of a system that is able to help in this field, specifically in the High Energy Physics field. Taking advantage of what has been learnt in the master, the design of a Printed Circuit Board called the Data Processing Block (DPB) will be explained, focusing mainly on the special aspects that a physics experiment like this will need.

Please, take note that this TFM is a continuation of the same line of work than my previous TFM in the Master in Electronic Systems Engineering (MUISE) [1], so a comparison will be made in the following chapters with respect to the advances made from that time, which not only include software, even though a lot of focus is put in there, but also a change in the hardware, making it more similar in terms of form factor, power consumption, I/O ports, etc... to the final design used in the physics experiment

Hence, this TFM will be divided in the following parts:

- **Introduction:** including an explanation of the experiment, named Hyper-Kamiokande (HKK) and the current status in the development of the Data Processing Block (DPB) together with the objectives developed on this TFM to further advance to the final design of this board.
- **DPB Architecture:** In the previous TFM, an explanation of the DPB architecture had been given, in a single chapter. Here, more details, specially the ones that are useful to know for this project, will be given. Also, as we enter in the prototype and test phase, a budget must also be introduced accounting for the cost of developing such a device, where not only the manufacturing cost is considered but also other variables, like, as we will see, subcontracting a company to perform part of the job.
- **DPB in the electronics front-end:** the DPB must be able to communicate with several other boards to perform its functions, as it is like a hub, concentrating and routing the data from the whole electronics front-end to an external datacenter. This means that an explanation of the different protocols used to communicate with the boards is mandatory.
- **Methodology and Tasks:** this part focuses on the work done for this TFM. It can be summarised in a series of chapters where different procedures and results from the development are stated. It contains all the work done from January 2023 to June 2023, before receiving the next prototype, which is also explained in the last chapter of this part.

- **Conclusions and future work:** this project will be in the works until 2027, so there is a lot of time for improvements towards an ideal software design that fulfills all the requirements. This part states the conclusions drawn from this work and the future lines of work which involve the manufacturing of even more prototypes and finally, the mass manufacturing of the definitive design.

Concerning the project's organization, to be developed later in this document, this Master's thesis serves as a step towards designing a SOM based system. The SOM is a pivotal component within the electronics module developed by UPV for Hyper-Kamiokande (HKK). My role in the Institute for Molecular Imaging Technologies, where I am employed, involves collaboration with international research groups from Italy, such as the National Institute of Nuclear Physics, as well as institutions in Great Britain (Warwick University) and Japan (University of Tokyo). Additionally, collaborative efforts extend to private companies like Enclustra, responsible for designing the PCB for the final configuration of the DPB. Public institutions like the Universidad Autónoma de Madrid (UAM) provide essential technical support. Their expertise has been of great help for enabling me to understand the hardware and software architecture of the system to be developed and the tools used for that purpose within this year.

As a brief preview of this Master's thesis, it's worth noting that all the outlined objectives have been achieved, with detailed explanations to follow in subsequent chapters. These objectives are challenging, requiring a comprehensive understanding of the system from both hardware and software perspectives, essentially starting from scratch. Consequently, a thorough explanation for each phase of the development process is presented in English, as this thesis forms part of the official project documentation.

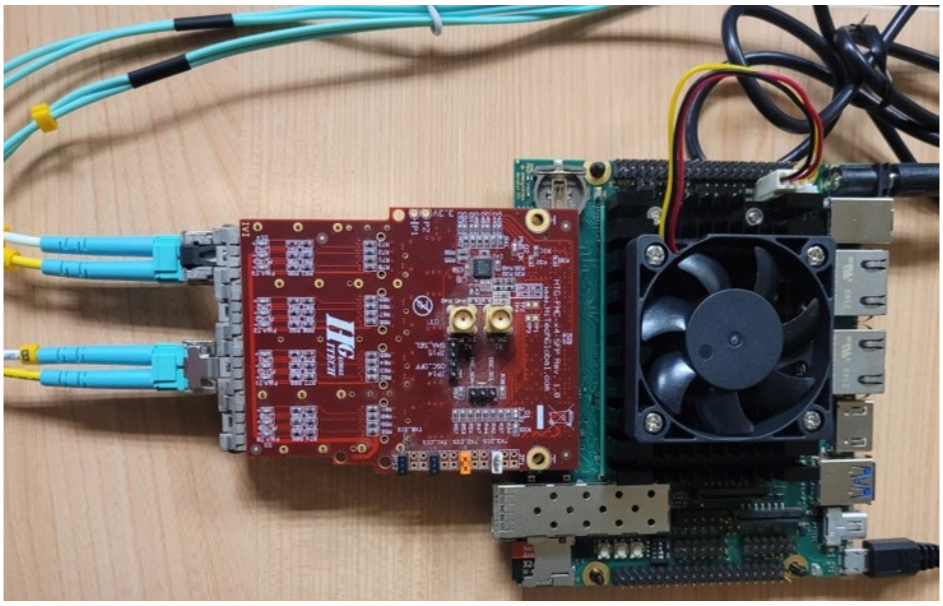


Figure 1.1.1: Electronic board where the DPB software has been designed

Chapter 2

Objectives

The outlined work in the chapter 4 corresponds to the second half of 2022. However, in this TFM, we will look further in time. Considering the time schedule of figure 1.2.1, the development had just started. Now we are looking to 2023 development, where the hardware must be designed an finalised, at least in some form of prototypes.

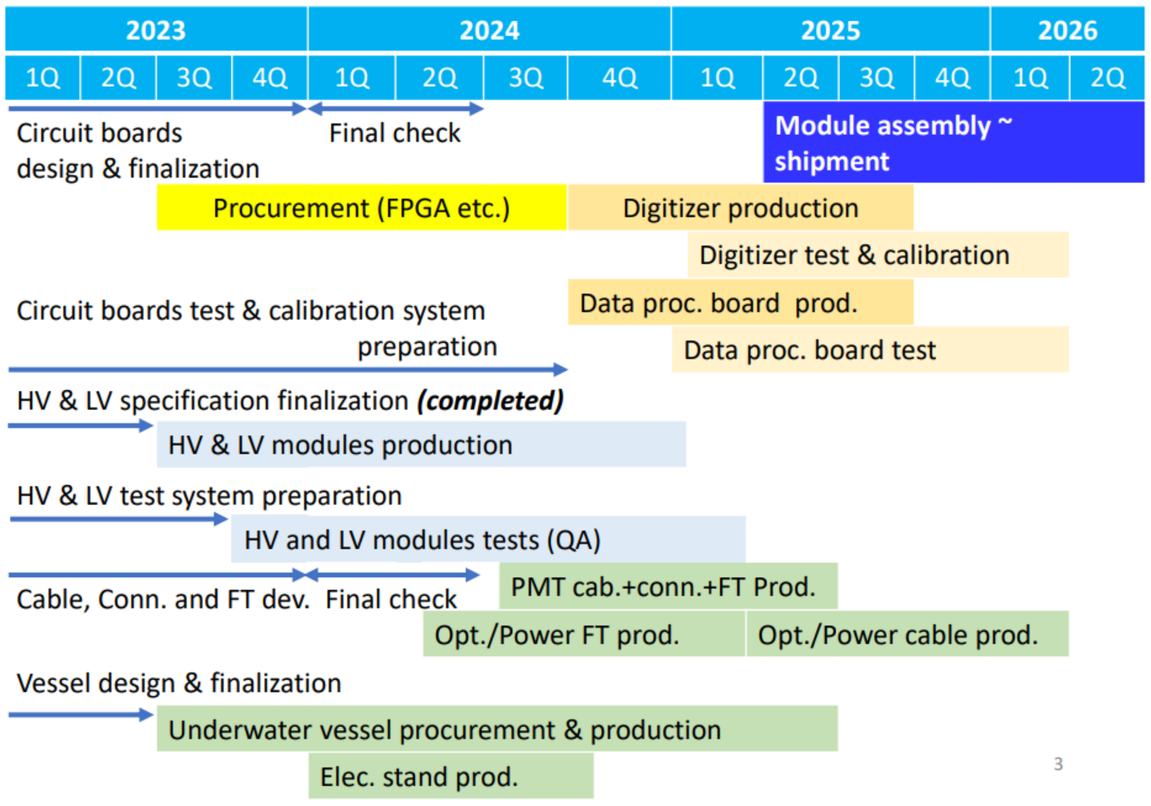


Figure 1.2.1: Time schedule for the design of the vessel, together with the front-end electronics and the PMT

This objectives chapter lists the goals to be achieved by this specific Final Master Thesis inside the HKK project, described in chapter 3. This project is immense, with participation from over 300 researchers across 75 institutes in 15 countries as of 2018. The numbers have since increased with the start of the HKK development. Such a vast collaboration presents coordination challenges. This means that management is crucial for this project to succeed and be ready in time. The responsibilities of UPV are integral to the experiment's electronics, attributed to the specialized module being crafted by the UPV. Specifically, the design is done by the Institute for Molecular Imaging Technologies (i3M), and the Memorandum of Understanding (MoU) signed by the Ministerio de Ciencia e Innovación reflects this dedication. This TFM serves as a foundational blueprint for the hardware and software development of the DPB board, leveraging contemporary tools.

As such, the objectives, seen as a continuation of the work already done over the ZCU102 part [1], can be summarised in the following way:

- **Learn and use the SOM form factor:** Jumping from an Evaluation Board, the System On Module (SOM) form factor will be the one used in the final DPB design. As such, some time must be invested into understanding which differences there are in this PCB with respect to a full fat Evaluation Board. These aspects will be discussed in Part II.
- **Understanding communication protocols with the rest of the electronics front-end:** the DPB has the main function of being the bridge that communicates the vessel with the outer world. As such, it must understand the *language* that every board in the vessel speaks. This *language* in computers is translated as some communication protocols with a specific data-frame format. In Part II, the protocols used by the timing, DAQ, HV, LV and the digitizer will be explained, to give a proposal of which kind of hardware or software must be integrated into the system to perform a correct communication.
- **Adding redundancy for the data links:** The first link where redundancy will be investigated is the data link, that is the link between the DPB and the DAQ. This link is a standard TCP/IP link so Linux should have mechanisms that allow to group interfaces and make redundancy easy to program and configure.
- **Debugging the operative system:** Making some code work in the context of an embedded OS is not trivial, as errors or unexpected behaviour can arise. To counter this, debugging exists, as a way to execute step by step the code of your software to spot possible bugs. Some debugging techniques in this context will be looked into to discuss in which cases it is better to use one or another.
- **Study the boot flow:** having several redundancy mechanisms that increase the reliability of the platform is desirable. However, if all of this mechanisms rely on having an embedded OS running, then a new weak point is created: what if the embedded OS cannot boot up due to the corruption of a memory zone where the OS image is stored? In this TFM the boot flow used to boot PetaLinux will also be investigated and some changes will be proposed to make the boot up more reliable.
- **Discuss improvements for incoming prototypes:** in the process of developing the SOM specific software, some issues will arise. These issues will be noted down to see if they can be fixed in an easier or cleaner way with the next hardware revisions of the DPB.

These objectives will be divided into the three remaining parts of the TFM:

- **Part II - DPB architecture:** where the Zynq Ultrascale+ architecture, SOM form factor and the communication protocols with the rest of the electronics front end.
- **Part III - Tasks and Methodology:** where debugging, redundant data links and the boot flow discussion will be included.
- **Part IV - Conclusions and future work:** as previously stated, discussion about future prototypes is required to establish a future line of work. This part will serve to that purpose, stating the potential issues that have happened or could happen in a future and take preventive measures against them.

Chapter 3

The next generation of Cherenkov Light Detectors: Hyper-Kamiokande (HKK)

3.1 The predecessor

The equipment utilized in physics to examine neutrinos is called a neutrino detector. It is designed to remain untouched by external factors such as cosmic rays or ambient radiation [2].

These detectors are massive entities and their operation is based on various neutrino detection methods available. Examples include scintillators (as seen in the Cowan-Reines neutrino study), radiochemical approaches, radio detectors, and Cherenkov light detectors.

The focus of this chapter is on the Cherenkov light detection method. These detectors are essentially vast tanks filled with water, supplemented with deuterium and gadolinium. This setting is optimal for neutrino interactions. When a neutrino interacts with water's electrons or nuclei, it can produce a charged particle that moves faster than light does in water. This results in a specific type of light emission known as *Cherenkov light*. This phenomenon is equivalent to the sonic boom in sound waves.

Surrounding the water tank are light-sensitive instruments called Phototubes. These can either be gas-filled or vacuum tubes that react to light. Among them, the Photomultiplier tube (PMT) stands out for its superior sensitivity. This PMT is tasked with detecting the Cherenkov light stemming from the neutrino interaction.

By analyzing the light pattern, various neutrino properties can be determined, including its direction, energy, and on occasion, its specific type. See figure 1.3.1 for reference.

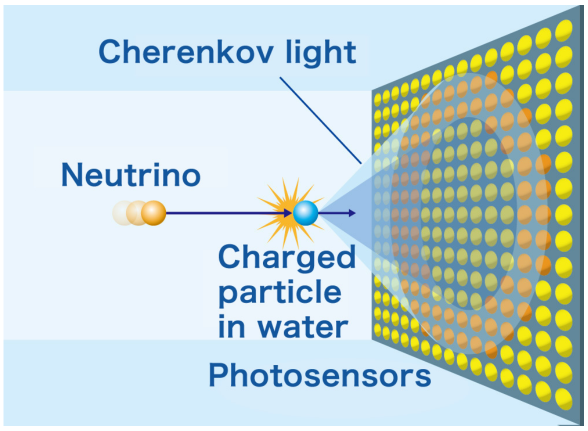


Figure 1.3.1: Cherenkov light phenomenon detected by a PMT [3]

Neutrino interactions with matter are exceptionally infrequent due to their low probability. Therefore, the larger the water tank and the greater the number of PMT, the higher the likelihood of detecting interactions within a given timeframe.

The currently operational largest neutrino detector is the Super-Kamiokande (SKK). The name "Kamiokande" is the result of joining the following words: KAMIOKA Neutrino Detection Experiment. Situated beneath Mount Ikeno near Hida city in Gifu Prefecture, Japan, it's managed by the facility known as Kamioka [4]. Over time, this detector has seen four major revisions due to various factors, such as cascade failures or the swapping of the 6000 PMT and electronic upgrades in its most recent version, Super-Kamiokande IV. The progression of this neutrino detector from its inception in 1996 to the present is documented in Table 3.1.

Phase		**SK-I**	**SK-II**	**SK-III**	**SK-IV**
Period	Start	1996 Apr	2002 Oct	2006 Jul	2008 Sep
	End	2001 Jul	2005 Oct	2008 Sep	2018 Jun
Number of PMTs	ID	11146 (40%)	5182 (19%)	11129 (40%)	11129 (40%)
	OD	1885			
Anti-implosion container		✗	✓	✓	✓
OD Segmentation		✗	✗	✓	✓
Front-end electronics		ATM (ID) and QTC (OD)			QBEE

Table 3.1: Main characteristics of the Super-Kamiokande iterations from 1996. The values in parentheses below the number of PMTs in the ID show percent photo-coverage of the surface [5]

In these versions, there has not been any alteration in the count of PMT or in the coverage percentage. Instead, modifications were primarily directed towards bolstering protective measures as mentioned earlier, and transitioning the front-end electronics. The technology shifted from using Analog Timing Modules (ATM) for Inner Detector (ID) and Charge to Time Converter (QTC) for Outer Detector (OD), to adopting a refined, unified QTC Based Electronics over Ethernet (QBEE) approach.

3.2 HKK structure

In May 2020, an extensive renovation of the SKK received approval. This would involve constructing a brand-new observatory from scratch, aiming to have a fiducial mass 10 times greater than the previous version, thereby becoming the world's largest underground water tank. When juxtaposing the specs of HKK with those of the SKK versions in table 3.1, the HKK boasts 40,000 PMT, equating to 40% photo-cathode coverage—identical to the SKK. A larger quantity of PMT is required due to the increased tank size. The primary ambition of this detector is to facilitate pristine proton decay searches through $p \rightarrow e^{+} + \pi^{0}$ and $p \rightarrow \overline{\nu} + K^{+}$, along with monitoring anti-neutrinos emanating from supernovas.

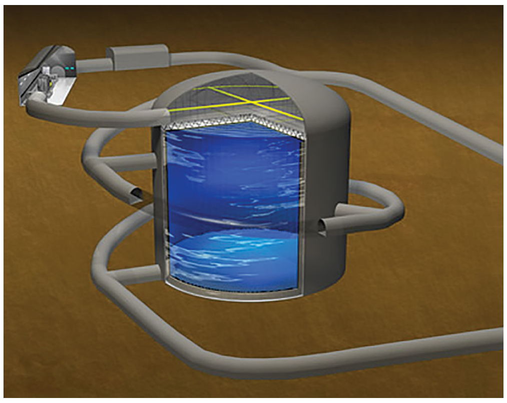


Figure 1.3.2: Hyper-Kamiokande water tank concept sketch. A megaton water tank used for gathering in 10 years data that SKK would take 100 years [3]

The design of the detector features a cylindrical tank with external dimensions of 60m in height and 74m in diameter. It is filled with 260,000 metric tons of ultrapure water, serving as a *water Cherenkov detector*. Encircling this tank are state-of-the-art photodetectors with 50% greater efficiency than those of SKK, offering enhanced accuracy in light intensity and detection timing. These PMT (Hamatsu R12860) will magnify events, particularly from neutrino interactions [6]. This design facilitates physicians in more precisely gauging the direction and velocity of neutrinos traversing the detector. An in-depth cross-sectional view of the primary tank is presented in figure 1.3.3.

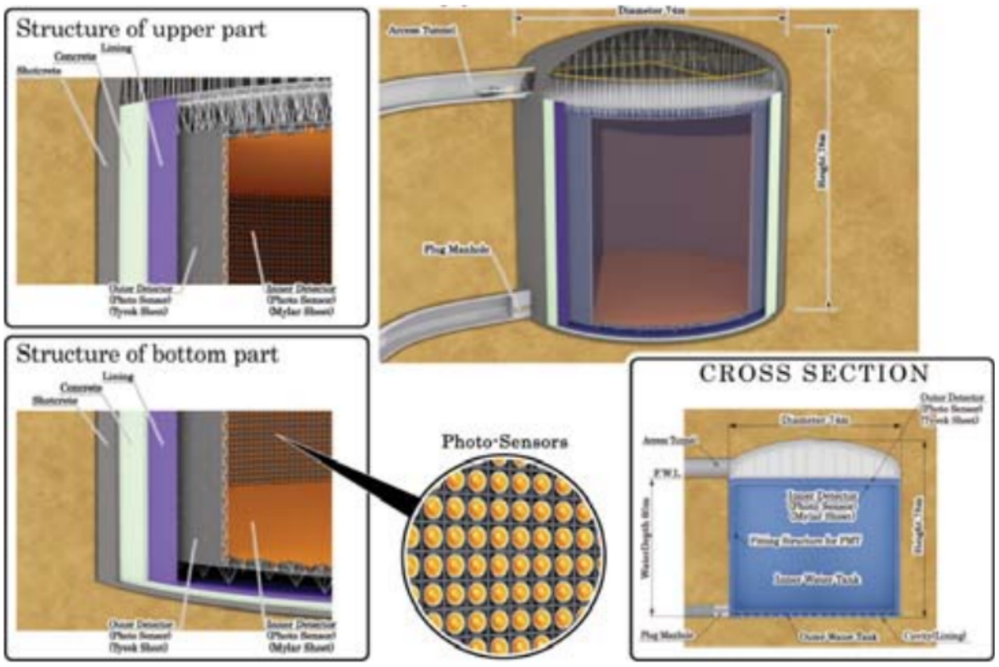


Figure 1.3.3: Hyper-Kamiokande Cross section for the first tank [6]

The detector is equipped with front-end electronics and is interconnected with a computer cluster dedicated to data processing. Its advanced data acquisition capabilities enable event detection within an average time frame of $2\mu s$. The structure of the OD is largely consistent with that of SKK, maintaining a layer thickness of 1 to 2m to limit external interference.

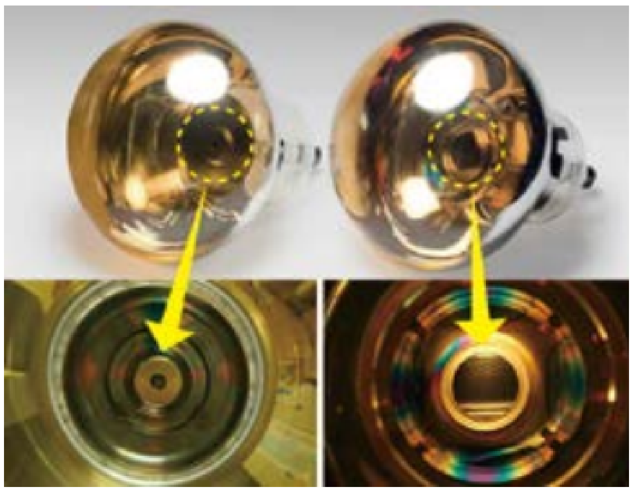


Figure 1.3.4: Ultrasensitive PMT that will be installed inside the HKK [6]

3.3 HKK goals

The substantial expansion in detector size facilitates the study of the tiniest constituents of matter in the Universe. These constituents comprise elementary particles, specifically quarks and leptons. For instance, a proton, which is constructed from three quarks, combined with an electron—a type of lepton—makes up a hydrogen atom. Neutrinos are a subset of leptons that carry no electric charge and exist in three varieties: electron neutrino, mu neutrino, and tau neutrino. Intriguingly, these three neutrino types can intermingle and transition between each other, a process termed ”Neutrino oscillation.” This phenomenon was first uncovered by Super-Kamiokande in 1998. In-depth examination of neutrino oscillation can shed light on the inherent properties of neutrinos.

While the discovery of the Higgs bosom particle seemed to complete the Standard Model—a framework describing the system of elementary particles—the data about the neutrino's mass and its mixing rates derived from past studies diverges considerably from that of quarks. This discrepancy suggests that a more foundational framework, surpassing the Standard Model, might be necessary. Neutrino oscillation experiments hold promise as pivotal tools in discerning the core structure of elementary particles [3].

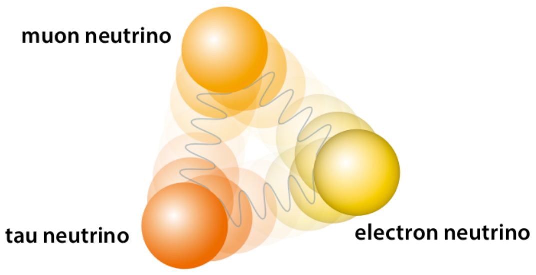


Figure 1.3.5: The existence of three flavors of neutrino and the mix of them cause the neutrino oscillations that make them change their type. [3]

Additionally, the detector does not solely rely on neutrinos originating from outer space, as depicted in figure 1.3.6. It also works in conjunction with neutrinos emitted from the J-PARC accelerator located in Tokai, Ibaraki. These neutrinos journey across Japan, from one end to the opposite where HKK is situated, facilitating the study of CP violation. A substantial number of neutrino detections is essential for examining CP violation. While SKK took a considerable duration to accumulate this volume, the HKK is projected to amplify the count of detected neutrinos by a factor of 30, thus expediting the research process. Concurrently, enhancements to the J-PARC neutrino beam are planned for HKK.

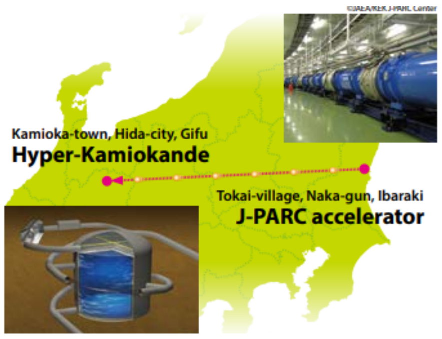

Kamioka-town, Hida-city, Gifu
Hyper-Kamiokande
Tokai-village, Naka-gun, Ibaraki
J-PARC accelerator

Figure 1.3.6: Travel of a neutrino from the JPARC accelerator to HKK[3]

The primary objective of this project is to delve deeper into the study of neutrino oscillations. Key phenomena targeted for investigation in the HKK, along with its descriptions, are shown in figure 1.3.7.

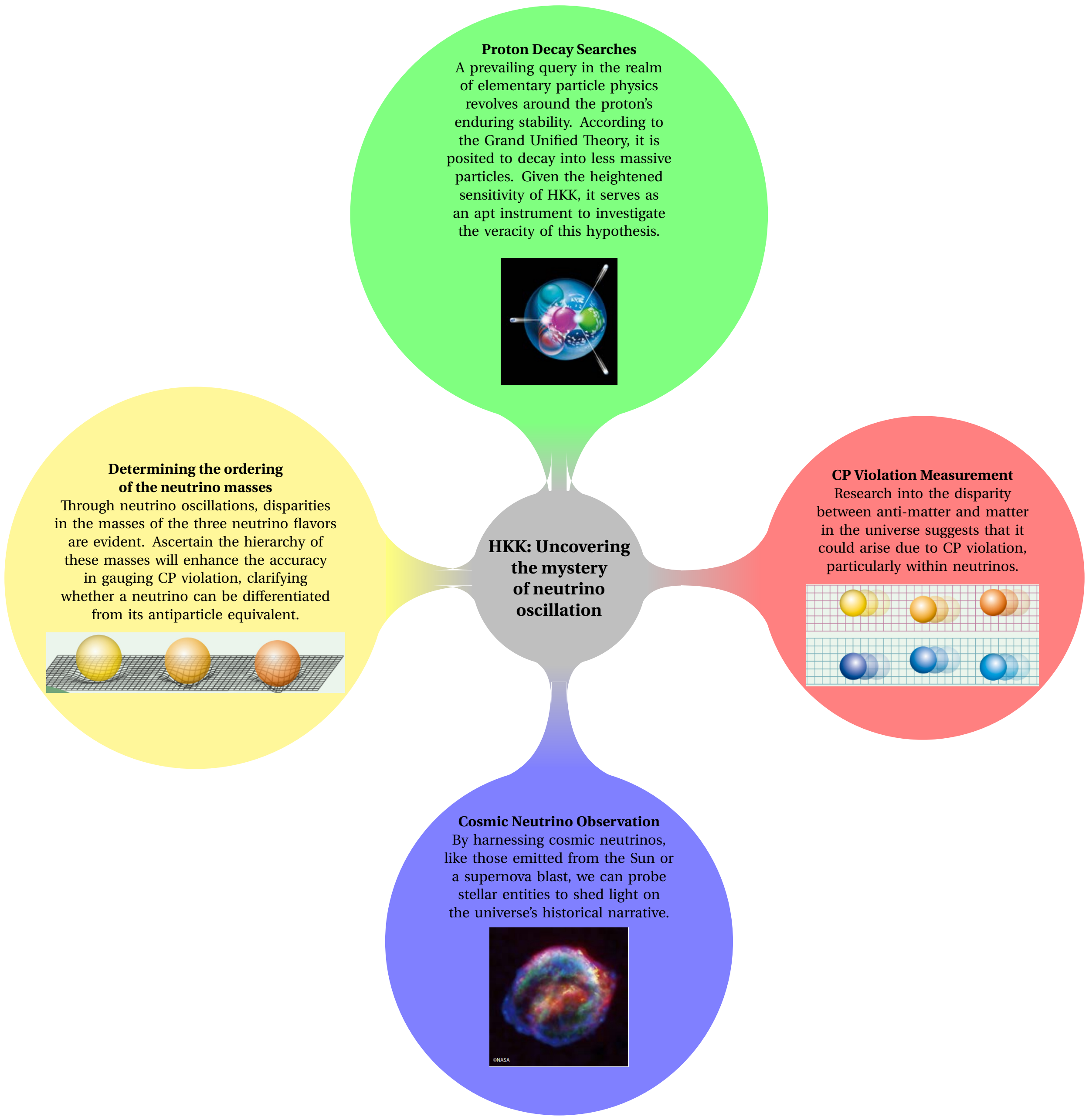

Proton Decay Searches
A prevailing query in the realm of elementary particle physics revolves around the proton's enduring stability. According to the Grand Unified Theory, it is posited to decay into less massive particles. Given the heightened sensitivity of HKK, it serves as an apt instrument to investigate the veracity of this hypothesis.
Determining the ordering of the neutrino masses
Through neutrino oscillations, disparities in the masses of the three neutrino flavors are evident. Ascertain the hierarchy of these masses will enhance the accuracy in gauging CP violation, clarifying whether a neutrino can be differentiated from its antiparticle equivalent.
HKK: Uncovering the mystery of neutrino oscillation
CP Violation Measurement
Research into the disparity between anti-matter and matter in the universe suggests that it could arise due to CP violation, particularly within neutrinos.
Cosmic Neutrino Observation
By harnessing cosmic neutrinos, like those emitted from the Sun or a supernova blast, we can probe stellar entities to shed light on the universe's historical narrative.
©NASA

Figure 1.3.7: Hyper-Kamiokande neutrino oscillation investigation fields [3]

The goals set out are at the forefront of elementary particle physics, primarily because the HKK has the potential to validate these four theories. Stepping away from theory, one must also recognize the future technological implications of these investigations. While they might currently seem like theoretical constructs aimed merely at a *better understanding of our universe*, history shows us the practical significance of foundational discoveries. Much like the exploration of electron behavior led to the birth of electronics, a staple in modern devices, and the study of photons paved the way for the development of fiber optics and photonics ensuring rapid communications, probing into neutrinos could usher in a novel realm of technology, tentatively termed *neutronics*. Such technology might be revolutionary for space communication, given neutrinos' unique property of minimal interaction and attenuation, ensuring low attenuation in long-distance communications.

Chapter 4

Status of the DPB development

Before getting into the matter, this chapter will cover the current status of the electronics front-end, so that the point of departure in this TFM is clear. The UPV is in charge of designing both hardware and software of the Data Processing Block (DPB), an electronics module in charge of concentrating and routing all the data that comes from other parts of the frontend to an external datacenter, known as DAQ. A simplified block diagram of the electronics front-end can be seen in figure 1.4.1.

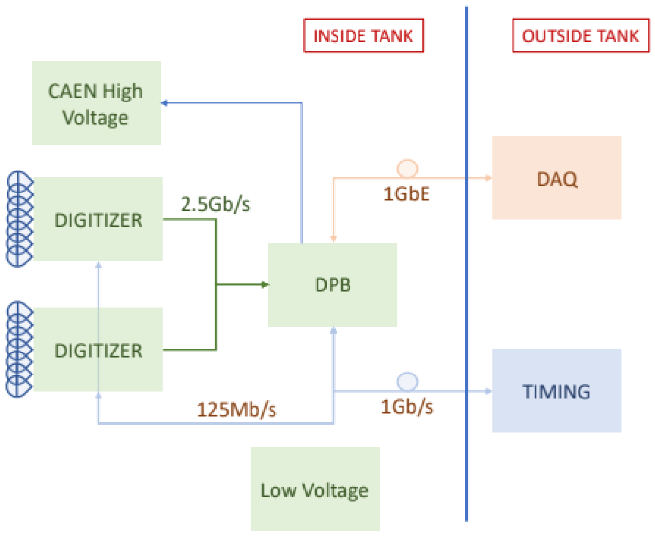


Figure 1.4.1: Block diagram of the electronics front-end. The digitizers are connected to the PMT, which send the data to the DPB. The DPB also controls the High Voltage Module (HV) and Low Voltage Module (LV) power supplies and receives information from a Timing module that must be sent to the digitizer

This electronics front-end has a special requirement regarding reliability and durability: one of the most important changes in HKK with respect to SKK is the fact that the PMT are much more sensible and accurate. This must be accompanied by the use of the shortest possible wires to avoid attenuation and signal degradation. This poses a challenge because it means that the digitizer board must be inside the vessel together with the PMT. So, the electronics front-end is underwater in the same vessel as the PMT. This makes impossible for the boards to be replaced or repaired in the case of a failure or broken module because it would imply to send a diver or empty the detector which is very expensive and time-consuming.

As for the tasks it must perform, they are typical to any data acquisition system: it must have links with the digitizer boards that will pass the information in digital format to the DPB, fiber optics link with standard TCP/IP stack to communicate with the DAQ and several serial interfaces to communicate with the power supply units (labelled as High Voltage and Low voltage in figure 1.4.1). There is also a timing link, which uses a custom protocol to ensure a clean clock regeneration for the digitizer and accurate timestamping for the events captured by the PMT.

So, the development began with the agreement upon using an architecture that fulfills the following requirements:

- **Data Integration:** The Data Processing Block (DPB) encompasses several functional units within the ID&OD front-end electronics box, notably *"Systems control and data transmission"*, "Optical I/F" connected to the DAQ, and *"Data organization and buffering"*. In essence, the DPB serves as a central hub that receives varied data formats and consolidates them into a uniform frame format, facilitating straightforward decoding by a server. Consequently, an application must run on the DPB to execute these operations. An efficient approach involves deploying an embedded Operative System (OS) on a System on Chip (SoC). The chosen SoC should meet the criteria of adequate computational power for the TCP/IP stack management, sufficient transceivers to accommodate all optical fiber link connections, and ample memory to buffer incoming data from the digitizers.

- **Dependability:** In our foundational design, the front-end electronic modules are submerged in water, obviating the need for extended analog cables running from the PMT to exterior electronics. This arrangement significantly reduces cable numbers within the tank and simplifies the PMT support framework and overall detector assembly. Notably, cable sheathing is a known Rn source. By minimizing cable lengths, we anticipate a reduction in Rn concentration. Once filled, the tank's electronics become inaccessible, making component repairs unfeasible. As a result, we aim for an underwater system failure rate of less than 1%/year. Naturally, this approach brings forth technical hurdles, including ensuring watertight integrity and system durability. Moreover, any upgrades to the electronics would necessitate draining the tank, an event not projected to occur for approximately a decade post-initiation of operations.

It was decided to use an architecture that combines power and flexibility: *Zynq Ultrascale+ MPSoC*. This architecture made by Xilinx Inc. is divided into two blocks, explained later that include a powerful ARM CPU that can run Linux and FPGA fabric, where hardware can be instantiated through HDL coding by loading a bitstream into the chip. This allows to have a species of mini-computer into each vessel, reliable and powerful enough to leave all the monitoring and data transferring to a trustworthy chip. Then, the users of the detector will just need to connect from the outside to each of the vessel in case they want to change any parameter to the OS. The access to these boards would be a very well known standard: a Linux terminal, which allows for customizing any aspect of the module.

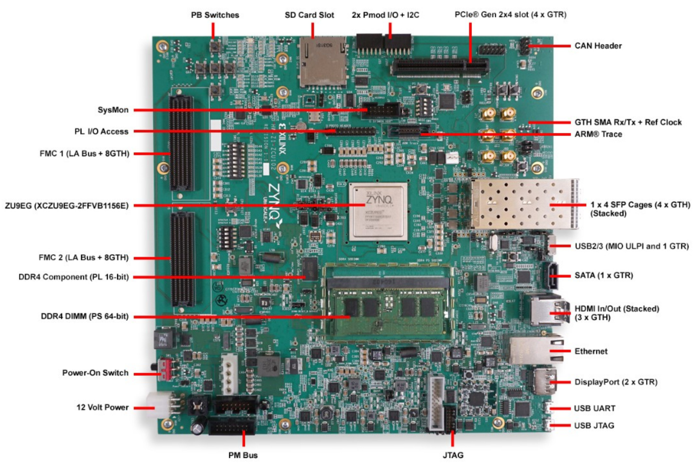


Figure 1.4.2: ZCU102 Evaluation Board used to begin the DPB design and development

As there was no fixed hardware decided, besides the architecture being used, it was decided to begin the software development in a Xilinx standard Evaluation Board (EvB) called the *ZCU102*. This board is dedicated to develop applications of almost any kind that can take advantage of a Zynq Ultrascale+ chip. As we can see in figure 1.4.2, the ZCU102 has a lot of I/O ports. However, only some of them were used, for example, PCI Express won't be present in the final design.

The ZCU102 development part was covered in my other TFM, here is the summary of the conclusions and what was learnt through the time spent with the ZCU102.

- **PetaLinux as the Chosen OS:** Xilinx offers a comprehensive SDK, facilitating the development process especially when bootstrapping an OS. While an OS differs from conventional applications, the Linux kernel has evolved over time to enhance its capabilities and support diverse CPU architectures. Xilinx's specialized Linux distribution for their specific hardware merges the familiarity of the standard Linux development environment with the benefits of on-board application building and sophisticated debugging tools.
- **Startup Procedures:** The Zynq Ultrascale+ integrates multiple hard microcontrollers in the PS designated for various memory types. Beyond the principal DDR4 memory, there are controllers for QSPI, eMMC, and SD flash. Upon initialization via the FSBL, these memory controllers become accessible, allowing the system to boot from their respective domains. This versatility boosts system reliability by enabling memory redundancy across multiple chips. Should one memory source fail, an alternative can take over. Expanding on this, PetaLinux even supports PXE boot [7], drawing the kernel image from a remote TFTP server.
- **Custom OS thanks to Yocto:** A standout feature of PetaLinux is its adaptability, aligning well with the Zynq Ultrascale+ architecture. Users can modify the PetaLinux image to include or exclude specific software components, optimizing the OS for the intended purpose. Yocto provides the environment for such precise customization. Grasping Yocto's concepts, including recipes, layers, and its toolchain, is paramount for efficient image construction, a critical step for creating the software stack for DPB functions like data linking and slow control.
- **Startup Scripts:** Ensuring maximum system autonomy, initial configurations post-OS boot must be automated. Linux inherently caters to this requirement, vital across platforms from PCs to embedded devices. In this TFM, both *systemv* and *systemd* were explored. Though the latter has seen its share of controversy within the Linux community, its structured approach to service initiation using *.service* files offers a more streamlined configuration method, making it the preferred choice for the DPB firmware.
- **Managing Slow Control Data:** Typically, sensors utilize serial protocols for configurations and data retrieval. In this TFM, the hard SPI controller was interfaced with a magnetometer sensor, and a register read test was conducted. This test validates the Zynq Ultrascale+'s ability, under Linux, to handle read/write operations with the various sensors set for the DPB. Regardless of the sensor's protocol—be it SPI or otherwise—the Zynq platform is versatile. With I2C hard controllers in the CPU and customizable cores in the FPGA like UART and RS232, diverse serial communications are supported, ensuring comprehensive slow control capabilities.
- **Incorporating Additional Network Interfaces:** Optical links are earmarked for data and timing link implementation, each using distinct protocols. For the data link, the standard TCP/IP stack is employed. Within this TFM, the steps to integrate additional network interfaces that are TCP/IP compatible have been detailed. Introducing more links via SFP modules connected to the FPGA fabric is straightforward as it simply requires an IP Core. A pre-existing Linux driver fully supports the required functionalities for link establishment. However, configuration within the device tree demands careful attention to ensure that the IP Core's configuration is done correctly by the Linux driver.

Part II

DPB Architecture

Chapter 1

Zynq Ultrascale+

In this chapter, the architecture used for the design of the DPB will be explained focusing on the parts that are relevant to the DPB requirements.

1.1 Core architecture to the DPB

The Data Processing Block (DPB) is a system that can be classified in the embedded applications category as it is a special combination of both hardware and software to perform a very specific function. If it had to be summarised into one phrase it would be *Being the intermediary with everything inside the vessel*, as seen in figure 1.4.1, where it is clearly connected with every module contained in the vessel and also to two modules (timing and DAQ) connected outside the vessel.

For running the custom protocols at optimal levels of performance, an FPGA, where hardware can be instantiated to perform specific tasks, besides having a general purpose CPU is ideal, so, as stated in my MUISE TFM, the ZCU102 by Xilinx had been chosen.

Looking into the offerings of Xilinx for the development of Embedded Applications platforms, the Zynq UltraScale+ Multi Processing System on Chip (MPSoC) architecture seems to be the most attractive. This formidable chip offers 64-bit processor scalability, seamlessly integrating real-time control with both soft and hard IPs dedicated to graphics, video, waveform, and packet processing. Constructed on a unified platform featuring a real-time processor and programmable logic, it manifests in three different variants: dual application processor (CG) devices, quad application processor and GPU (EG) devices, and video (EV) devices. This diversity opens up endless possibilities for various embedded applications [8].

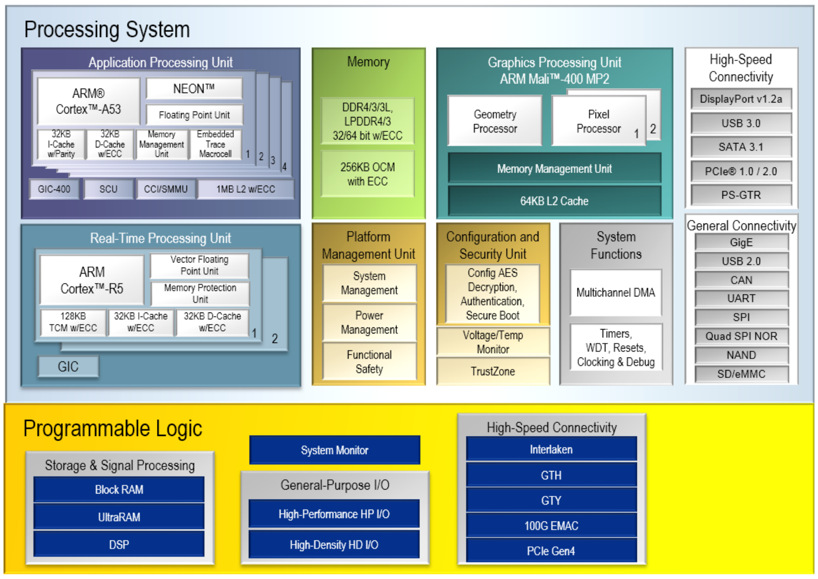


Figure 2.1.1: Zynq UltraScale+^{TM} Block Diagram [8]

This poses the fundamental advantage of having a powerful, fully customizable system that allows for fine tuning to suit the needs of any application, even the one in this project, where redundancy constraints imply the need of much more hardware than other embedded application. Figure 2.1.1 shows a block diagram with the parts that form the Zynq Ultrascale+ MPSoC architecture.

This block diagram can be divided into two parts: the light blue background which is the Processing System (PS) and the yellow one corresponding to the Programmable Logic (PL).

1.2 Programmable logic (PL): flexibility

FPGA have always consisted of a series of blocks in a microchip that can be programmed to become any hardware that can be coded in Hardware Description Language (HDL), opening the doors to a world of high flexibility when creating custom hardware without relying on long time custom ASIC designs. Xilinx divides the PL into the following components:

- Data Storage and Processing: Integrated memory within the FPGA facilitates the generation of memory hardware blocks for data retention. For instance, Xilinx labels their inbuilt FPGA memories as BlockRAM, and its advanced high-density version is known as UltraRAM [9]. Additionally, DSP blocks exist, boasting unique architectures which permit the development of units like Multiply and Accumulate, optimizing filtering processes.
- Fundamental Logic Cells: These serve as the foundational units to establish logical operations. Constituted by a 6-input Look Up Table (LUT), a multiplexer, and a pair of flip flops, especially in the UltraScale+ Architecture context. Figure 2.1.2 illustrates a basic Logic Cell schematic.

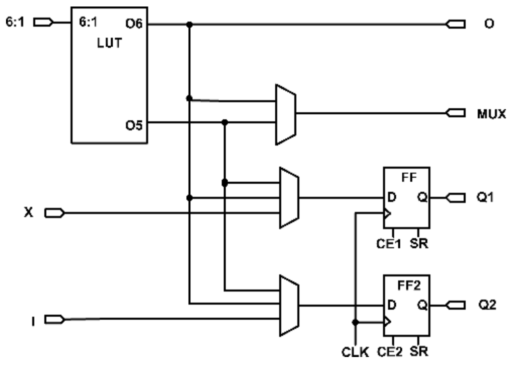


Figure 2.1.2: Zynq UltraScale+^{TM} Logic Cell [10]

- Universal I/O: These are the FPGA's input/output pins, segmented into two categories: High Performance (HP), designed for swift operations, and High Density (HD), which though slower, offers greater density, thereby enabling a plethora of low-speed communication channels.
- Advanced Speed Connectivity: This category houses the fastest buses, ranging from Multi Gigabit Transceivers (MGT) types like Gigabit Transceivers Type H (GTH) and Gigabit Transceivers Type Y (GTY), all the way to the 100G Ethernet Media Access Controller (EMAC) and Peripheral Component Interconnect Express (PCIe). These are capable of managing transfer rates in the vicinity of tens of Gigabits Per Second. Figure 2.1.3 shows a block diagram of an MGT where there are two clearly defined parts: transmission and reception where there is an interface that communicates with the corresponding block that configures communication protocol and it goes through a series of blocks to a differential buffer. This allows to have higher speeds than what would be achieved with a standard LVDS connection.

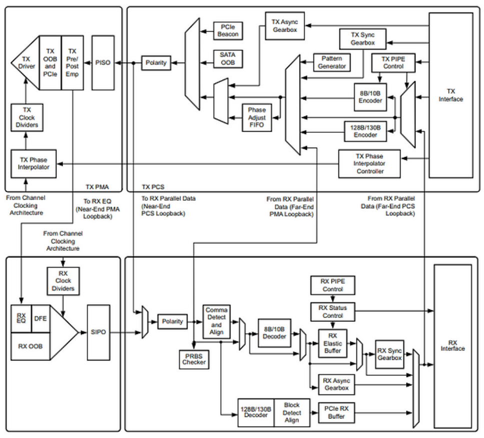


Figure 2.1.3: Multi Gigabit Transceivers (MGT) block diagram in a Xilinx FPGA

1.3 Processing system (PS): raw compute power

Historically, FPGA primarily comprised of CLB. However, as they frequently collaborated with a dedicated CPU for task acceleration, the notion of PS emerged. This concept devoted silicon space to distinct functional units, essentially integrating ASIC adjacent to the FPGA and bridging the two via a rapid interconnect fabric. The components of the UltraScale+ MPSoC architecture in this segment include:

- Application Processing Unit (APU): These represent the core units of the PS and are responsible for running the preferred embedded OS. The chip used in HKK for the DPB sports a dual core system that enables the much needed high performance.
- Real Time Processing Unit (RPU): A pair of auxiliary cores explicitly tailored for Real Time application execution. These applications demand deterministic time execution. These cores are not used yet, they are left in low power state but if any issue regarding computational power arises, they can be used, running a separate program from the APU.
- Memory: Acts as the primary memory for the CPU, mirroring the role of main memory, usually called just RAM on PCs.
- Platform Management Unit (PMU): Entrusted with overseeing the entire system's scheduling and power distribution.
- Voltage and Temperature Oversight: Vital for HKK, as these parameters must be closely monitored to guarantee that the DPB remains within ideal operational conditions, thus enhancing its longevity.
- Connectivity spans from standard options like USB, Controller Area Network (CAN), and Ethernet to high-speed counterparts such as Display Ports, USB 3.0, Serial Advanced Technology Attachment (SATA), and HDMI.

1.4 Embedded operative system: PetaLinux

Such a complex system would not be complete without a piece of software up to the task of managing this hardware. This is where the embedded Operative System (OS) comes in. Having an OS poses several advantages in managing hardware as it provides a standardised API for the software to use the hardware as efficiently as possible.

For this matter, Linux has been chosen as it is an open source solution, meaning that anything can be customised with enough expertise and it has all the required libraries by the rest of the teams at the front end electronics.

However, Linux is just the kernel, the core that interacts with the hardware and provides the abstraction layer for the software running over Linux to use. It is not a complete OS. When combined with certain software packets it is converted in what is known as a Linux distribution (or *distro* for short). Xilinx has a custom Linux distribution called **PetaLinux**, which has been chosen due to the following reasons:

- **Customizability:** Linux can be used in a big variety of device ranging from Raspberry Pi to 64 cores dual CPU servers. Depending on the use case, the operative system must include certain software packages. PetaLinux uses the Yocto project that, thanks to the recipe's system, allows to add and remove software packages very easily, tailoring the image to the user's needs.
- **Size:** This advantage is linked to the previous one: being able to remove any unneeded software package or kernel component like for example, PCIe drivers or USB drivers, will make the resulting OS image smaller. This allows the image to fit into smaller flash memories, have a smaller memory footprint regarding RAM usage and, in our case, allows to have more copies of the same OS, which means increased redundancy as there are more independent files in different memory zones that can be read to boot the system.
- **Easy updates:** the bootup process is relatively simple: it is formed by just three files as it will be seen in Part III, all of them written in an internal flash memory. This makes the update process trivial as it is just overwriting those files with new ones. Of course, one must be very careful when doing this because, if the new system does not boot up due to an error while making the update, the DPB will be unusable. For this reason, it is recommended to have a Golden Image where the system can fallback to ensure that it can always boot up, even after a failed update.

Chapter 2

Hardware design

In this chapter, the hardware used for the DPB prototype and final version will be discussed together with the reasons that made our team opting for the solution explained here.

2.1 The SOM form factor

The DPB development team had already decided which architecture was going to be used for developing an embedded system that will perform high speed data routing and monitoring tasks, with a certain degree of reliability due to the special constraints of the project. Now, this team is in the prototyping phase, where the team has already worked with that architecture and has developed some software on a standard board. This board is too big and expensive to directly being used as the prototype as it is too far from the final hardware that your budget and time constraints allow.

Also, as this is an university team, we do not count with enough expertise or manpower to dedicate several weeks or even months to design a board, route correctly a chip that has 900 pins! and with special requirements regarding PCB layers and signal integrity.

After thorough investigation, it was decided that having a complete in-house designed board was not possible and that is when the System On Module (SOM) solution was discovered.

SOM provides the core components of an embedded system, that is the FPGA chip of the desired architecture which has all the blocks explained in the previous chapter in a single production ready PCB. Figure 2.2.1 is an example of how a SOM looks.

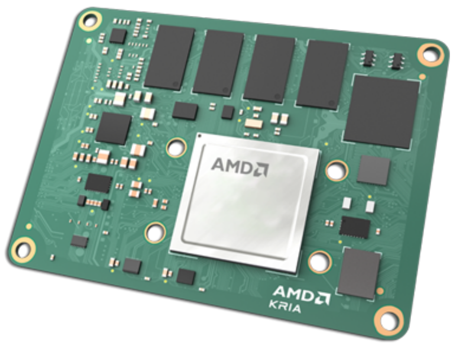


Figure 2.2.1: Picture of a SOM as a general overview

SOM has the following components:

- **Embedded processing system:** which in this case is a Zynq Ultrascale+ MPSoC chip just as in the ZCU102 but tailored to the needs of the experiment. For example, the ZCU102 chip had much more CLB than the final SOM decision as this experiment does not need that much custom hardware to work.

- **Main memory:** required for the embedded application to run in. They are usually DDR memory, with several chip soldered depending on the memory capacity ranging from 2 GB to up to 8GB of memory with the possibility of adding Error Correction Code (ECC) for increased reliability.
- **Flash memory:** non volatile memory to store the OS image, configuration files or data needed for the correct operation of the system. There are usually QSPI NOR flash or eMMC memories which use the same technology as standard SD cards but are soldered into the board.
- **Clocking:** the Zynq Ultrascale+ needs external clock sources to work so SOM also has some fixed clock sources on the PCB that are directly routed for the cleanest possible clocks to the pins of the FPGA.

2.2 SOM + Baseboard as a decoupled solution for I/O

Looking at figure 2.2.1, the reader may have thought *Where are the inputs and outputs of the system? It seems incomplete to me.* And it would be correct, because the SOM itself does not include any form of standard inputs and outputs like USB, ethernet ports, SATA, etc... However, if we turn around the PCB, we can see something of the likes of figure 2.2.2.

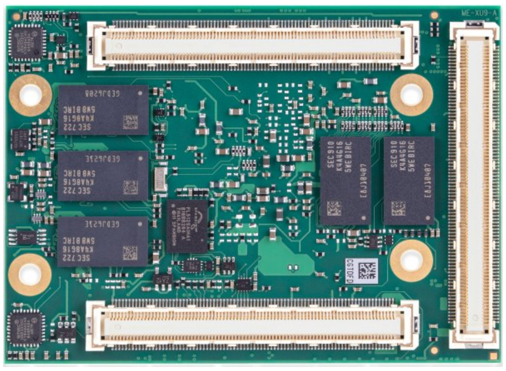


Figure 2.2.2: Picture of a SOM from the back, showing the three standard A B C module connector

SOM has standard connectors that can be inserted into a baseboard that provides the corresponding I/O for the application being developed. This is a huge advantage because it isolates the complexity of a computing system from the I/O requirements. SOM designers take care of all the complicated routing and placing of CPU, memories, clocks, etc... and route all the I/O pins of the chip to the module connectors. At the other side the baseboard engineers design another PCB where only I/O is involved so it is easier to design something from scratch in this project as the special requierements on the DPB involve having redundant ports which means lots of fiber optics SFPs and MiniSAS connectors, a combination which is hard to find in an *all in one board.*

Figure shows an example of a SOM base board. In the center, the three A B C connectors where the SOM is inserted can be seen.

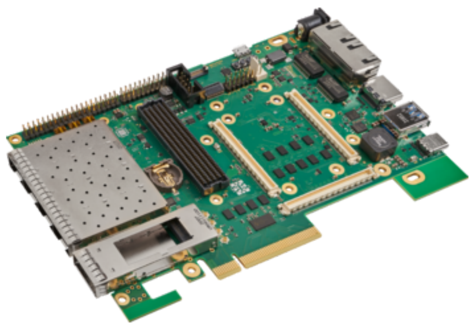


Figure 2.2.3: Picture of a SOM baseboard. This board includes for example FMC slot, USB and a SFP cages together with a PCIe interface to insert on a computer for versatility.

This combination proved to be very useful for the DPB design as it allowed us to decouple the SOM design (complex part) and the baseboard design (relatively simple part), which was crucial as it made us be in time for the deadlines that were imposed during this part of the project development.

2.3 Redundancy and reliability considerations

Having a SOM doesn't make reaching reliability conditions automatic, now it is easier thanks to the SOM + baseboard combination as the requirements can be assigned to one of the two parts and work on them independently. To summarize, the reliability mechanisms in hardware, which are completely based on redundancy are as follows:

- **Redundant links:** In figure 1.4.1, there are four different blocks connected to the DPB. Redundancy in each one of the links communicating the DPB with each block has been implemented as follows:
  - **Digitizer:** there are two digitizer connected to each DPB. Each digitizer is connected through a MiniSAS connector, which is a type of port used on Network Access Storage (NAS) with high speed and very good reliability. MiniSAS has 36 pins, equivalent to having 4 SATA connectors in parallel. There are enough pins for implementing all the communications protocols needed and even add redundancy to some of them. Figure 2.2.4 shows a schematic of the MiniSAS port with labels of each of the connections.

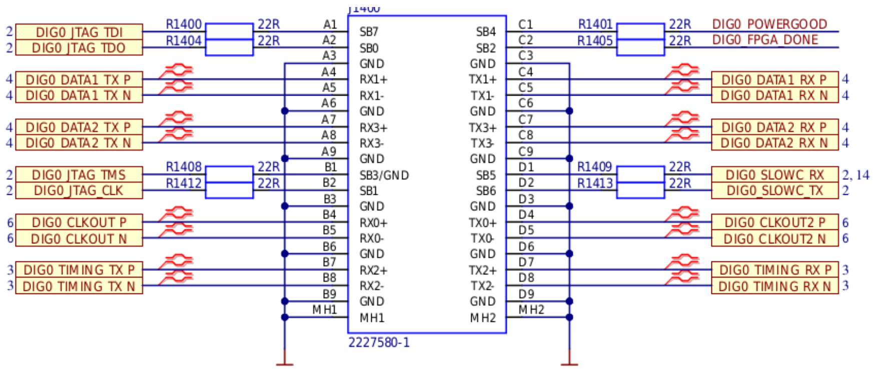


Figure 2.2.4: MiniSAS port schematic from the DPB2 schematic generated by Altium PCB Designer

Table 2.1 shows the different links to the digitizer through the MiniSAS connector and whether they have a redundant link or not, depending on how important they are for the experiment correct operation.

Link	Description	Redundant?	Justification for using redundancy
DIG0_DATA	Digitizer data link. Using this link the data from the PMT and events are transferred to the DPB using Aurora protocol (explained in the next chapter)	✓	Data link is crucial, the experiment would not work without having the data of the PMT being sent to the DAQ. Thus it is a priority to ensure that this link always work even after 10 years
DIG0_JTAG	JTAG programming. The digitizer use Kintex 7 FPGAs which need to be programmed with a bitstream	×	JTAG has too many pins to add redundancy without compromising the redundancy of other links
DIG0_TIMING	Timing link to the digitizer, used to provide timing information for accurate timestamping	×	Timing is very important but these links are not as stressed as the data links specially when receiving a supernova event where the data link is expected to work very close to its full capacity
DIG0_SLOWC	Slow control sensor monitoring from the digitizer	×	Slow control are just sensors for monitoring with a very low frequency so wearout will be much less than with the data link. Also, data link can be used to send slow control data by defining a special dataframe
DIG0_CLK	Low jitter phase alligned clock for accurate timestamping on the digitizer	✓	The phase alligned and low jitter clock is crucial for accurate timestamping and for the circuits on the digitizer to work correctly in detecting the events in the detector, thus two differential clocks are sent to each digitizer

Table 2.1: Digitizer links table listing which links have redundancy in hardware and the reasons.

– **DAQ:** Fiber optic links will be used. The DAQ is just an external server to the vessel which is communicated to all the DPB on the detector through an optical switching network. It uses the standard TCP/IP protocol stack where the physical layer would be 1000BaseX, that is fiber optics at 1Gbps speed. For each DPB, two fiber optics links are dedicated to communication with DAQ, often referred as *data links*. There will be one main interface, corresponding to a SFP port which the DPB uses under normal operating conditions. If that link fails it will automatically switch to the other data link.

– **Timing:** it uses the same physical layer as the data link. This link will be referred to as *timing link*. However, it is not a standard TCP/IP stack in the upper layer as with the data link, it is a completely custom timing protocol which requires an FPGA core instantiated in the PL to work. As with the data links, there will be two timing links: main and backup. Two FPGA cores will be instantiated. Each one of them will be routed to a MGT where a SFP module is routed from the baseboard so that two independent parallel timing links are available.

Figure 2.2.5: Avago SFP module used for both the timing link and the data link

– **High Voltage Module (HV) and Low Voltage Module (LV):** These boards are power supply units that are configured through RS485 protocol from the DPB. Figure 2.2.6 shows the schematic of the connector that goes to these boards. The redundancy here is applied in two different ways: first there is an independent RS485 link for each of the boards and the boards themselves have a main and a backup CPU that can be turned on using the signals labelled with *EN_CPU_(HV/LV)_(0/1)* in the connector on the same figure.

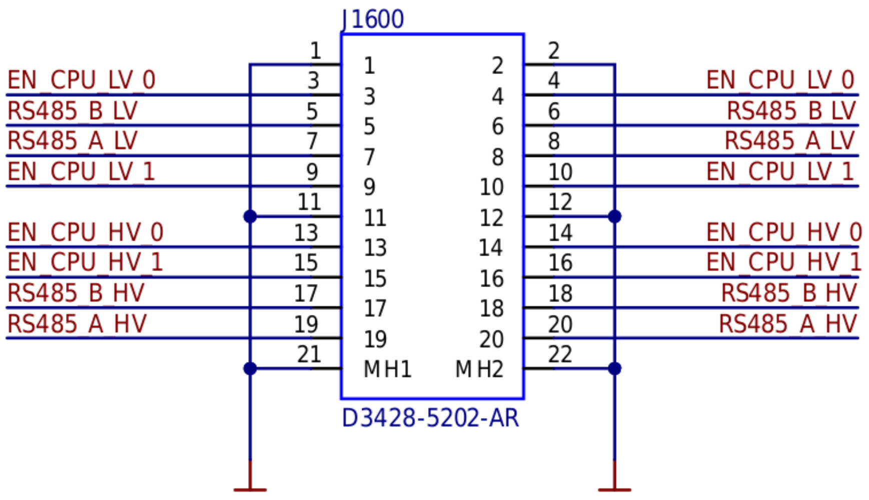


Figure 2.2.6: HV and LV port from the DPB2 schematic generated by Altium PCB Designer

Chapter 3

SOM-based commercial solutions

This chapter shows the available SOM commercial solutions and the logic followed for choosing to outsource the manufacturing of the DPB prototypes to an external company.

3.1 Xilinx commercial solutions

Xilinx, as the introducer of this form factor, has several commercial solutions available to the general public. Xilinx SOMs, as stated in the official announcement [11], are solutions for faster time to deployment in smart vision applications which enable software developers in their familiar design environment. SOM already includes the hardware needed on a credit card size PCB so that designers only need to focus on the board level design instead of chip level making the SOM directly pluggable into the end product for application deployment. This is a big advantage in our case because it allowed us to focus on the baseboard design with our specific requirements and then just plug in the SOM and boot it up.

Xilinx gives the following SOM portfolio in the Kria® line:

- **Kria K26 SOM:** for vision AI in smart cities and smart factories, using Zynq Ultrascale+ to bring a quad core Arm subsystem with 256K system logic cells and 4k60Hz video codec for smart cameras. It also has 240 dual pin connector with support for 15 cameras, 40G ethernet and USB.
- **Cost optimized SOM:** yet to be released to the public, will be oriented to electric drivers and other size and cost-constrained applications.
- **Highest AI compute SOM:** another yet to be released family of SOMs that deliver the highest performance/watt for edge AI applications.

3.2 Enclustra SOMs and base boards

Looking at Xilinx offerings it is clear that they are oriented to AI applications. HKK experiment requires the DPB to perform data gathering, routing and slow control monitoring, acting as a minicomputer inside each of the vessel in order to have total control over what is happening on it without needing to open the vessel physically (something impossible as previously discussed).

Enclustra Inc is one of the Xilinx partners regarding SOM design and manufacturing. In their web, there is a portfolio of their products where in the section System-On-Modules, there are 18 models of SOM that range from the architecture of the chip or FPGA used to the compute and memory capabilities. For our case, the ***Mercury XU8+*** SOM was chosen as it sports the Zynq Ultrascale+ MPSoC with enough FPGA blocks for the DPB application.

Figure 2.3.1 shows the front view of the Mercury XU8+ SOM, this SOM hasn't exactly fixed specs, but there are several tiers inside the same products depending on the memory or the FPGA size as shown in figure 2.3.2.

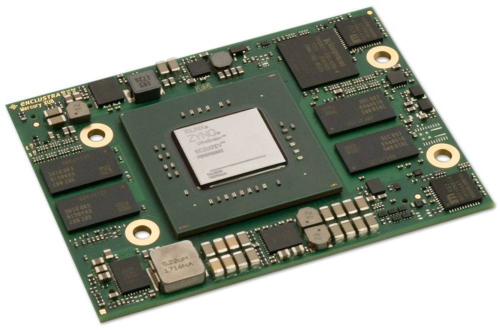
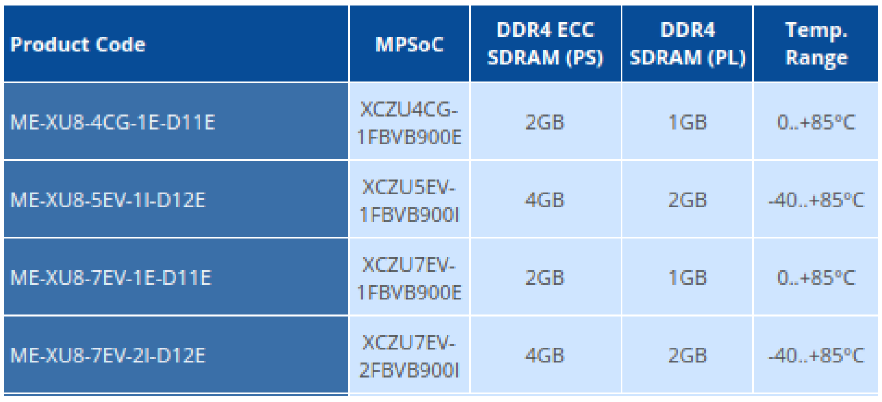


Figure 2.3.1: Mercury XU8+ SOM by Enclustra Gmbh

Specifications for this SOM vary from 2GB DDR4 ECC memory to up to 4GB. This is enough for prototype development but for final design, memory would need to be upgraded to 8GB.



Product Code	MPSoC	DDR4 ECC SDRAM (PS)	DDR4 SDRAM (PL)	Temp. Range
ME-XU8-4CG-1E-D11E	XCZU4CG-1FBVB900E	2GB	1GB	0..+85°C
ME-XU8-5EV-1I-D12E	XCZU5EV-1FBVB900I	4GB	2GB	-40..+85°C
ME-XU8-7EV-1E-D11E	XCZU7EV-1FBVB900E	2GB	1GB	0..+85°C
ME-XU8-7EV-2I-D12E	XCZU7EV-2FBVB900I	4GB	2GB	-40..+85°C

Figure 2.3.2: Mercury XU8+ SOM variants

Regarding the baseboard, all Part III has been done over a commercial Enclustra baseboard called the Mercury ST1+, shown in figure 2.3.3.

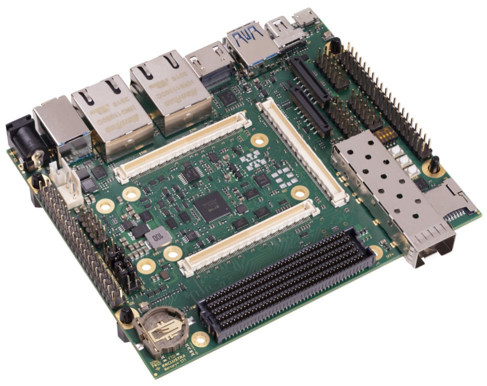


Figure 2.3.3: Mercury ST1+ baseboard by Enclustra Gmbh used for the first steps of software development in a System On Module

3.3 DPB prototype as a SOM compatible base board

When assigned with the task of designing and manufacturing the Data Processing Block by the HKK Directive Board, several challenges arise. In this section, the manufacturing will be treated. As with the production of any electronic product, then category to which the DPB belongs to, several options are presented:

- **Complete in-house solution:** this implies having the board designed both in hardware and software perspective, together with the generation of the required PCB files like Gerber files, drilling, etc... produced within the UPV. Then the UPV would also be in charge of procuring the components, the PCB materials and do the bringup tests to ensure that the board powers up correctly and receives correct supply from the electrical line.
- **Partial outsourcing:** the PCB design and the component procurement management would still be done within the UPV but the manufacturing and bringup tests would be outsourced to an external company.
- **Complete outsourcing:** hiring an external company to do the whole process from design, manufacturing to bringup tests and deliver us the finalized board with support over time.

First, complete in-house solution is completely discarded as the UPV does not have the manufacturing capacity to make such complex boards, neither the 12 first prototypes nor the final 1000 boards that will go into the vessels of HKK. Partial outsourcing seems manageable, however, due to time constraints and having enough budget to allow for a complete outsourcing, the latter was the path chosen. The structure within the investigation team at i3M makes the other two options unfeasible because of time and manpower, there are not enough resources. All the manpower is already busy looking into the most complicated and uncommon design requirements and firmware programming (the task I am developing). We simply do not have the structure of a decently-sized company to buy or to procure components.

That was when we decided to contact Enclustra with a proposition: completely outsource the design, manufacturing and bringup tests of a SOM baseboard that, complemented with an already available Marcury XU8+ SOM will be the first DPB prototype with hardware close enough to the final design to develop all the software functions.

The tasks assigned to Enclustra have been divided into three Work Packages (WP), each one with their corresponding sub-items:

- **WP1: Hardware Design:** DPB2 design in Altium, both in schematic and layout entry forms:
  - Detailed design and review of such design.
  - Library components entry and review.
  - Schematics entry and review.
  - Layout placement and review
  - Layout routing and review
  - Project Management
- **WP2: Board Production:** DPB2 production of 12 prototypes using the Altium layout and schematic (among others) files generated in the previous working package:
  - Production data generation and review (Gerber files, etc...)
  - Production support

- Production logistics
- PCB manufacturing specification document
- PCB assembly specification document
- Documentation review
- Project management

- **WP3: Board Bring-Up**: basic prototypes bring-up such as power, clock signals and JTAG programming.
  - Basic bring-up (power, clock, JTAG, 12 pcs).
  - Simple reference design (for JTAG test).
  - Bring-up report document.
  - Documentation review.
  - Handover and initial support.
  - Project management.

These are engineering costs, we need to also consider the external cost of board production. The summary of total cost from Enclustra outsourcing is shown in table 3.1.

Summary Position	Total Estimated Cost [EUR]
Development Cost	50.000,00 €
External Cost	10.000,00 €
Total Cost without VAT	60.000,00 €

Table 3.1: Summary of costs associated to Enclustra outsourcing

The costs per unit are very high but this is normal when manufacturing a prototype as all the engineering costs are included and few boards are manufactured. This cost will be much lower when manufacturing the 1000 final design boards as no engineering costs will be present or if they are, they will be much lower.

$$€/unit = \frac{60.000€}{12} = 5.000€$$

Furthermore, from the UPV there were also personnel involved, which has dedicated several hours for these boards to be produced in meetings with Enclustra defining the requirements and ensuring that the process went smoothly. Table shows the additional UPV personnel costs that should also be considering when speaking about the DPB manufacturing costs. The working hours stated here were done from January 2023 to June 2023.

Person's name	Time inverted [Hours]	Cost per hour [EUR/h]	Total Estimated Cost [Eur]
Francisco José Mora	100	60	6.000,00 €
José Francisco Toledo	100	60	6.000,00 €
Francisco José Ballester	100	60	6.000,00 €
Vicente Herrero Bosch	70	60	4.200,00 €
Raúl Esteve Bosch	30	60	1.800,00 €
Alejandro Gómez	800	40	32.000,00 €
Total Cost for the UPV	56.000,00 €		

Table 3.2: UPV personnel costs

3.4 TRL levels across the DPB development

As with any new technology development, it is crucial to enumerate the Technology Readiness Level (TRL) it undergoes through its development and associate them with the different stages in the development. As indicated by the official note from Spain Ministerio de Industria y Comercio [12], TRL is an accepted form of measuring the maturity of a technology from its proof of concept (TRL 1) to a completely tested system in a real environment (TRL 9).

Within these TRL not all technology fields are evaluated in the same way. There are some fields named the Key Enablement Technologies (KET), which the European Comission has decided that Europe would loose its competitiveness if these six technologies are not developed successfully:

- **Micro and nano electronics**
- **Advanced materials**
- **Industry biotechnology**
- **Photonics**
- **Nanotechnology**
- **Advanced manufacturing systems**

Table 3.3 shows an explanation of which condition should the technology fulfill to enter on that TRL and how each development stage of the DPB reach every TRL.

TRL Level	Technology Status	DPB status
1	Basic idea	Tasks of a data acquisition board
2	Concept and technology formulated	Which elements should a data acquisition board have.
3	Proof of Concept	Block diagram with all the I/O like in figure 1.4.1
4	Validated components in the laboratory	ZCU102 development
5	Validated components in relevant environment	DPB2 prototype manufacturing
6	Validated technology in relevant environment	DPB2 test inside a vessel at CERN
7	Validated technology in real environment	DPB2 test inside the vessel in the detector
8	Certified technology in real environment	Certification by experts commitee
9	Available technology in real environment	Hyper Kamiokande starts taking data successfully

Table 3.3: TRL levels table with respect to the DPB development

As can be seen in table 3.3, the DPB departs from a TRL3 because the concept is already invented and matured. It is from TRL 5 up to 9 where the special requirements are defined, specially the reliability specifications, which must be thoroughly tested throughout all the maturity process of the DPB. In this TFM, the TRL 5 will be developed, using the Mercury XU8+ SOM and ST1+ Baseboard provided by Enclustra to develop a firmware for the future DPB2 prototype.

Chapter 4

Communication protocols with other boards in the vessel

The DPB is in the center of the vessel when it comes to communications: it must talk to everything in the vessel and each one of these modules has a different protocol. This chapter defines the protocols used for the DPB to communicate with each board.

4.1 Digitizer: the Aurora protocol

Starting with the digitizer, it is a board which has a very small buffer. This is due to the usage of a Kintex 7 FPGA without any kind of DDR memory, leaving only the internal BlockRAMs available as memory for the firmware running on it. This means that a high speed protocol is needed. So Aurora was chosen. Aurora is a link layer communications protocol for use on point-to-point serial links, developed by Xilinx. It is available in two variants: *Aurora 8B/10B* [13] and *Aurora 64B/66B* [14]. The only difference between them is the frame size, which in the first we find 8 bits of data and in the latter 64 bits of data both accompanied with 2 bits to allow enough states for reasonable clock recovery and alignment of the data stream at the receiver.

There is much less overhead on the Aurora 64B/66B so this was the one decided to be used. Aurora is a parallel protocol, meaning that several lines can be used for doubling or tripling bandwidth, but here, as redundancy is the main focus only one data line will be used at the same time even though there are two data lines coming from each digitizer to the DPB.

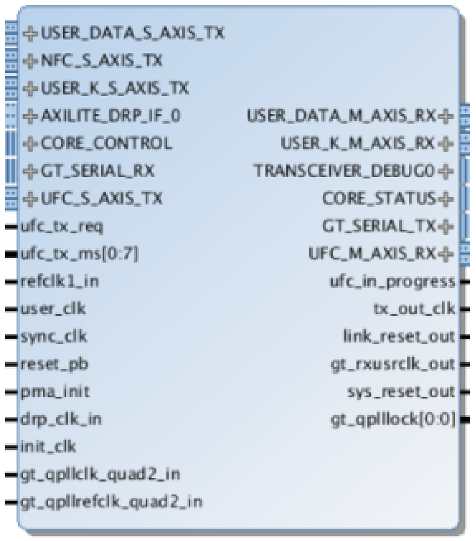


Figure 2.4.1: Aurora 64B/66B IP Symbol on Vivado [14]

Figure 2.4.1 shows an Aurora 64B/66B block with all of its port labelled. The most important ports are listed below:

- **USER_DATA_S_AXIS_TX:** Aurora uses AXI stream as the internal protocol to get or send the data it receives from another Aurora core to the rest of FPGA cores or to the PS. This interface is the transmission (TX) part, which means that through this input, all the data that we want the Aurora core to send arrives.
- **USER_DATA_M_AXIS_RX:** It is the reception (RX) part of the Aurora core, through this differential output, all the data that Aurora core receives is gathered and can be routed, same as with the TX, to a FPGA core or directly to the PS.
- **CORE_CONTROL:** contains all the control signals for the Aurora core such as resets or init signals for different parts of the core.
- **CORE_STATUS:** contains all the status signal for the Aurora core. This allows the user to monitor the link status, transceiver health and if there has been any error detected by the CRC and if it was corrected or not.
- **REFCLK_IN:** any FPGA core that uses transceivers requires a reference clock directly routed to the FPGA pins corresponding to that transceiver. For the FPGA cores case, GTH transceivers are used. These transceivers are divided into Quads, groups of four transceivers which also receive two differential clock inputs for the user to choose which one to use. Figure 2.4.2 shows a schematic of a GTH on the Mercury XU8+ SOM.

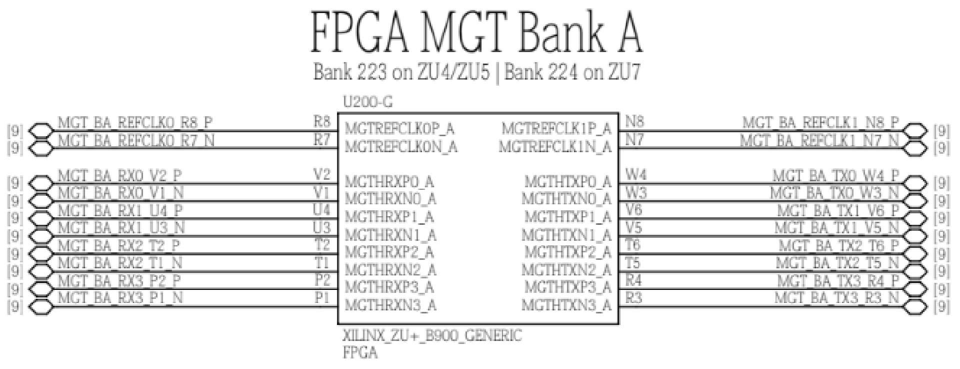


Figure 2.4.2: GTH Transceiver block on Zynq Ultrascale+ [15]

4.2 HV and LV: RS485 serial port

The High Voltage Module and Low Voltage Module use a standard RS485 protocol to encode ASCII characters that are sent and received to communicate with the electronics within these modules. The DPB has RS485 drivers to perform the encoding and decoding tasks of this protocol.

As they use RS485 [16], a Xilinx UARTLite Core [17] is enough to process the data that come from both ST3485 RS485 drivers: U1600 for HV and U1601 for LV in Enclustra schematic. This UART core uses an AXI4-Lite interface for register access and data transfers between the PS and the UART core for sending commands to the HV and LV modules and for reading the results stored in the registers. Figure 2.4.3 shows the schematic of both RS485 drivers.

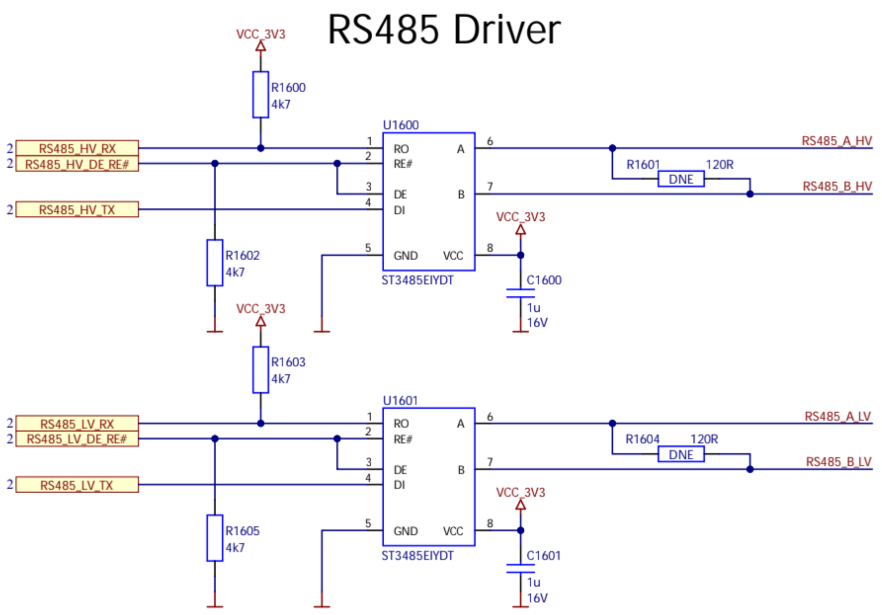


Figure 2.4.3: RS485 drivers for the HV and LV [16]. Each one of them is independently connected to an output pin of the connector in figure 2.2.6

4.3 DAQ: standard TCP/IP link

The communication with the DAQ, which is just a computer server is done with a standard network link. This link will use the standard TCP/IP protocol stack. A simplified view of the stack used in the communication between the DPB and the DAQ is shown in figure 2.4.4.

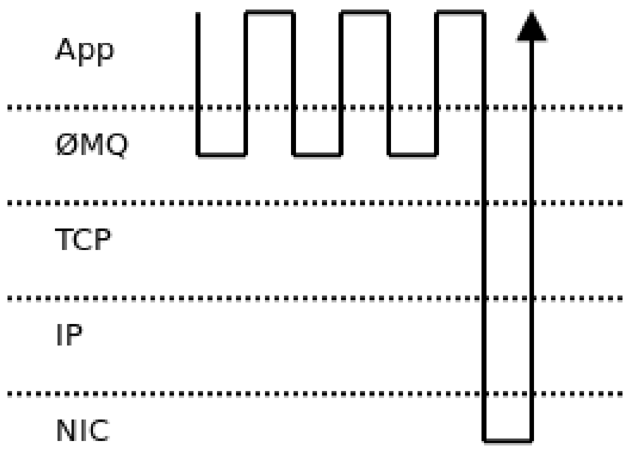


Figure 2.4.4: TCP/IP stack with ZeroMQ [18]. Note that the app can stack messages sending them one by one using successive calls to ZeroMQ layer. Then ZeroMQ will send them all at the same time

The libraries used to provide to the different applications the network stack is called ZeroMQ, which is a messaging system, or *message-oriented middleware*. When an application wants to communicate an event to other application or applications it just assembles the data and *sends* it as a message through the network.

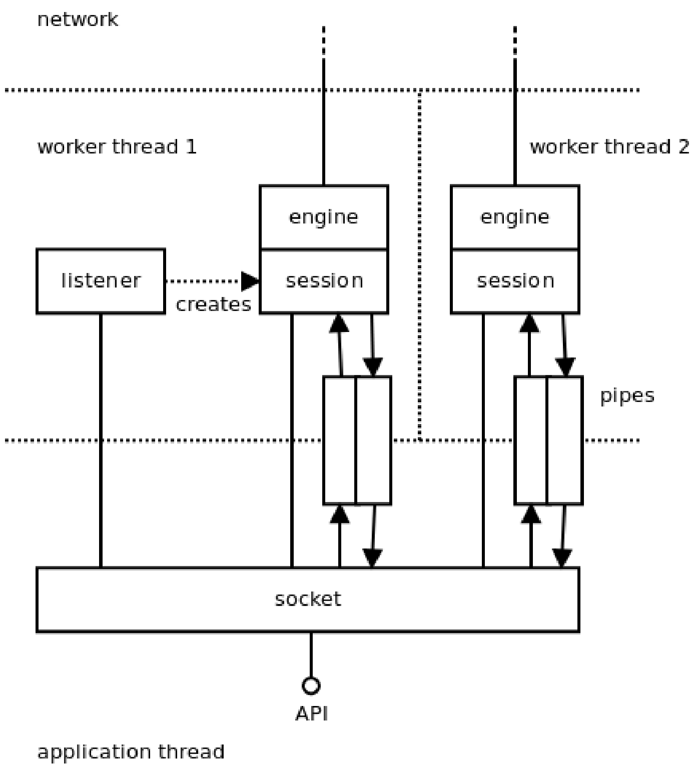


Figure 2.4.5: ZeroMQ architecture, divided into several worker threads for ultrafast parallel task processing [18]

The architecture of ZeroMQ uses sockets, very similar to the TCP ones. To this socket several worker threads are connected which can process different tasks like reading data in the network, queue messages, accept connections, etc...

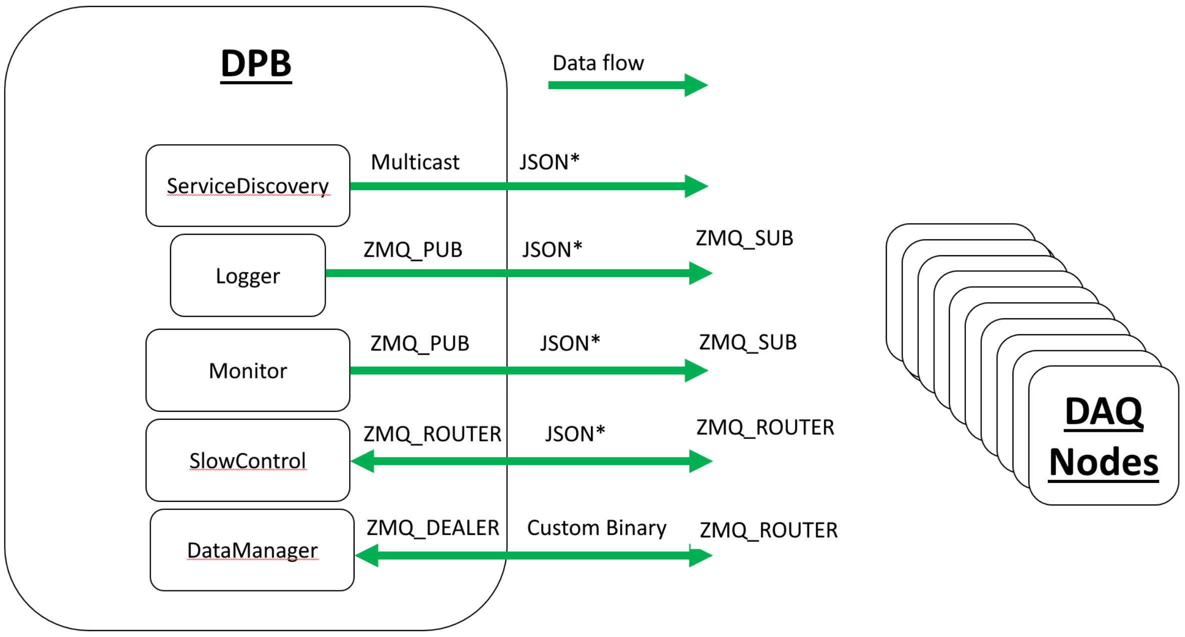


Figure 2.4.6: Different types of messages being sent from the DPB to the DAQ with data flows marked in green

For HKK DAQ, the messages shown in figure 2.4.6 have been defined. They include all the data that the DPB should deliver to the DAQ.

- **Service Discovery:** Used to connect to the DAQ nodes when the application to readout data starts.
- **Logger:** Any warning, error or information log of what is happening in the DPB, like doing printf to get information about the execution flow of a program.
- **Monitor:** these messages send periodic slow control messages, that is, the data sampled from the different sensors in the DPB or in any of the boards connected to them.
- **Slow Control:** These messages are only for slow control commands and configuration variables, for example RS485 commands for the HV or LV modules or I2C commands for the DPB sensors.
- **Data Manager:** this message holds all the data that comes from the PMTs like *adc charge*, the PMT identification and the timing information to give each event a timestamp.

4.4 Timing link: custom timing protocol

HKK is formed by several PMTs which allow to measure Cherenkov rings for studying neutrinos' behaviour. However, with just measuring the PMTs' charge it is not enough. To reconstruct the event, we need to know the arrival time of light emitted by neutrinos to the PMTs. The time synchronization is related to the accuracy of this event's reconstruction, so we need a very accurate system that keeps all the boards synchronized to a local time and synchronize that local time with the Universal Coordinated Time (UTC), so that the results of HKK can be compared to other neutrino experiments [19].

Figure 2.4.7 shows a block with the timing synchronization scheme, divided into three stages: A clock generated using GPS signals and a local time source in the form of an atomic clock and time quality measure to ensure that the generated clock is clean enough. Then a Time distribution board to send the information to all the vessels in the detector and finally the receptor in each vessel, what is called in figure 2.4.7 the *Time Distribution Endpoint*.

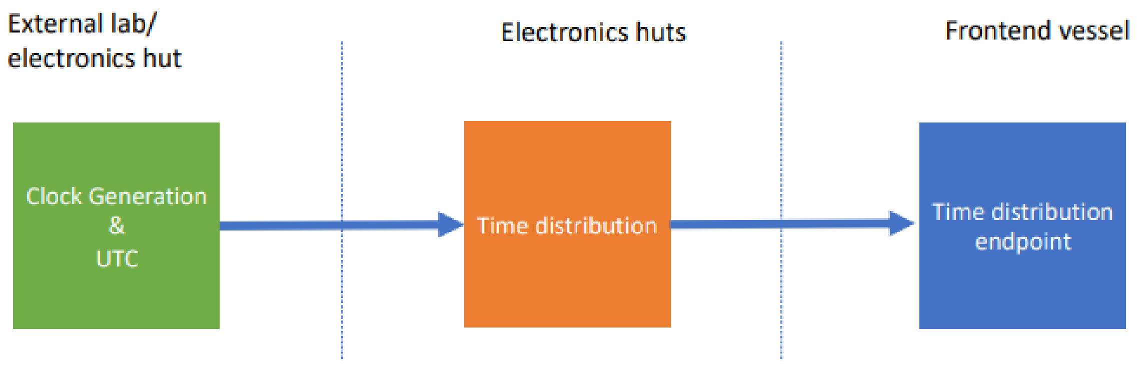


Figure 2.4.7: The block scheme that describes the 3 main time distribution sub-systems. [19]

The Time distribution endpoint in this case is the DPB itself. It is in charge of performing the following timing related tasks:

- **Clock Recovery and copying:** the DPB will get the clock from the time distribution system and generate two copies with minimal jitter and zero delay, one copy for each digitizer so that accurate timestamping can happen.
- **Synchronous commands:** the timestamping is computed through the use of two counters: a coarse and a fine counter. These counters are not infinite, they must be reset at some time. This is done through a synchronous reset that must be provided by the time distribution system.
- **Slow control transport:** data of slow control related to timing must also be sent to the DAQ.

These three tasks will be performed by a FPGA core which will be instantiated in the PL of the DPB Zynq Ultrascale+ MPSoC chip. This core is currently under development and will be a solution that will take advantage of two SFPs, completely dedicated to timing link, where timing information will be sent. Then, this core will recover the clock and all the synchronous commands like the TDC reset.

Figure 2.4.8 shows a proposal for the clock and data recovery FPGA core, using 8b10 coder/decoder, where data from the time distribution system is used to recover the clock and the data through a serializer-deserializer couple used in both sides of the link.

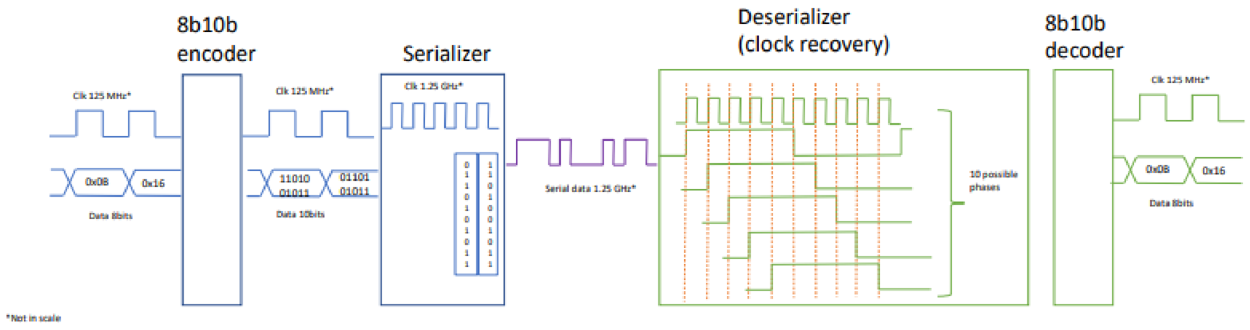


Figure 2.4.8: Simplified schematic view of the serializer-deserializer implementation for the DPB timing core [19]. This implementation is relatively simple, has low power consumption and small footprint on FPGA occupation.

Part III

Methodology and Tasks

Chapter 1

Methodology. Tools and platform

This part includes the explanation of all the tasks performed in the laboratory with the Zynq Ultrascale+ architecture. As it was commented in Part I, Chapter 4, this TFM is a continuation of other TFM written for the Master in Electronic Systems Engineering (MUISE) [1]. The MUISE TFM consisted on a first steps document where I began working not only on the Zynq Ultrascale+ architecture itself, but also all the Xilinx tools: Vivado, Vitis, PetaLinux Tools, etc...

This TFM will focus more specifically on a hardware form factor that will be very similar if not equal to the one introduced in the vessels in HKK. So the methodology applied to the previous TFM can also be applied to this one with the difference that now we can focus on hardware specific issues and not only on the chip itself. With the ZCU102 (figure 1.4.2) this was not possible as it had a completely different set of hardware, memory and I/O ports that didn't have any role on a DPB prototype.

The methodology followed to write each of this chapters is as follows:

1. **Requirements statement:** each chapter of Part III is written as the documentation for fulfilling a specific requirement of the DPB, so the first step is to state what we are trying to accomplish and how it will help to the DPB development.
2. **Modifications required:** through a step by step guide, the procedure for implementing the required changes in the DPB to fulfill the requirement is explained.
3. **Results and issues:** for future use, issues are also documented here. This TFM covers the electronics design part of a High Energy Physics Experiment. The architecture used in the DPB is not only applicable to processing and routing data, it can be tailored to many more kinds of applications and having a list of known issues together with a patch or other solution is ideal so that the developers don't stumble again with this problem in this project or in the future.

Regarding the tools used they have not changed neither:

- **Vivado ML Enterprise Edition 2022.2**: used for hardware generation, debugging and bitstream programming to the FPGA
- **Vitis 2022.2**: Integrated Development Environment (IDE) used to code applications for the Linux platform created in the SOM
- **PetaLinux Tools 2022.2**: Source Development Kit (SDK) that contains all the required tools to configure, customize and build a customizable Linux image. It covers all the steps regarding OS creation and has as a final output several files ready to load into the board to boot Linux.

1.1 The ST1 Base board + Mercury XU8 SOM

This part is mainly written as a documentation for the work done over a top-to-bottom SOM commercial solution by Enclustra Gmbh. That is, the SOM and the baseboard are available to the general public to purchase. This was the first step while in talks to manufacture the first DPB prototype to get used to the SOM form factor and start learning about the advantages and possible issues that can arise when using this kind of hardware instead of the ZCU102. The latter is an Evaluation Board (EvB). support from Xilinx and much more examples that made the learning curve much more approachable for a beginner.

The combination of hardware used is basically a Mercury XU8+ SOM (figure 2.3.1) with a ST1+ baseboard (figure 2.3.3). The SOM variant is the last one in figure 2.3.2: the *ME-XU8-7EV-2I-D12E*, with 4GB RAM available. Together with the used FMC card for extended functionality and the heatsink and fan for cooling, the hardware has the aspect seen in figure 1.1.1.

Chapter 2

PetaLinux project adaptation for Enclustra base board

The next chapters of part III provide all the aspects to consider when creating a PetaLinux embedded system in the System On Module (SOM) that will be used for the prototypes of the Hyper-Kamiokande vessels. Here, the steps for building the image are not listed as it has already been explained in ZCU102 but only the modifications needed to make the software work on the SOM.

As stated in the previous chapter the system used has the following parts:

- **SOM model:** Enclustra Mercury XU8+ (XU8-7EV-2I-D12E)
- **FPGA model:** XCZU7EV-2FBVB900I
- **Base board model:** Enclustra Mercury ST1+
- **PetaLinux version:** 2022.2

2.1 BSP provided. Adaption to 2022.2

The Board Support Package (BSP) for Enclustra boards can be downloaded from their GITHUB repositories. BSP provide a base project adapted to the platform to be used. Modifications need to be made to adapt it to the needs of each project.

There is also documentation for PetaLinux BSP and Boot management from different devices. However, these BSPs are adapted for version 2022.1, not 2022.2 so a careful procedure must be follow to compile a compatible version from this BSPs:

- **Upgrading the project**: Once unzipped by `petalinux-create`, the petalinux SDK supports upgrading project versions between minor versions, that is, versions that were released in the same year (2022.x for example). Just by doing `petalinux-build` or `petalinux-config` the user will get prompted to upgrade the Yocto project to the petalinux SDK version.
- **Adapting Enclustra patches**: Enclustra patches for u-boot and the kernel are included inside the BSP. This patches were made for a previous version of the kernel and the code has changed since then so there is some adaptations that need to be performed.
- **Detecting any bug**: changing version of PetaLinux implies changes not only in the base kernel version but also in the drivers included in the kernel. They get updates independently and then they get merged in a branch that is the one getting pulled for compilation of the PetaLinux image.

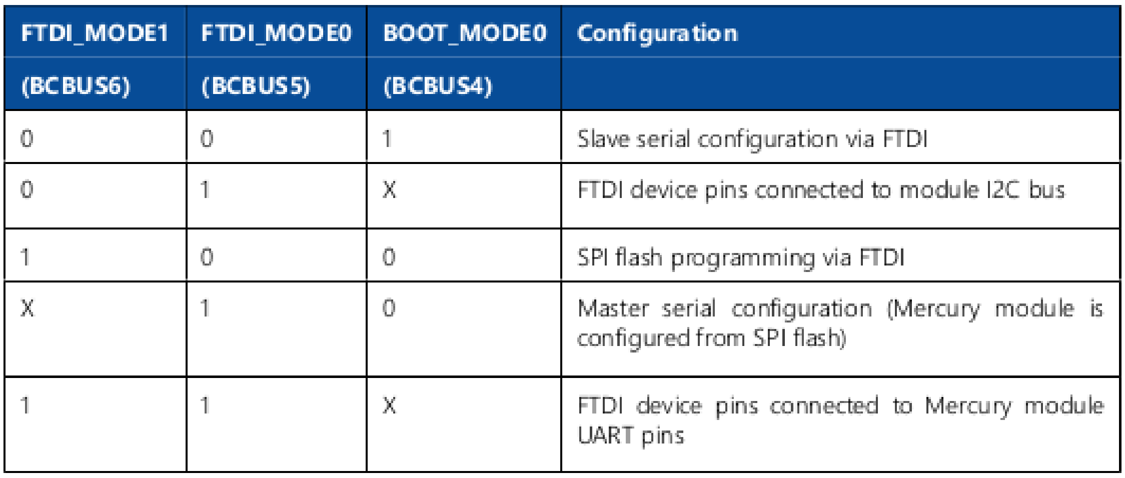
2.2 Preparations for boot process

Before being able to boot the system, there needs to be some previous configuration in the registers of the FTDI.

FTDI is a USB 2.0 Device Controller that allows FPGA serial configuration and SPI flash programming over USB without needing any additional hardware.

This FTDI is programmed through Enclustra Module Configuration Tool (MCT) to choose which of two ports A or B is multiplexed to the USB port:

- A: Xilinx JTAG implementation for programming FPGA
- B: Access to I2C and UART pins. Depending on FTDI_MODE1, FTDI_MODE0 and BOOT_MODE0, port B is configured as the table in figure 3.2.1.



FTDI_MODE1	FTDI_MODE0	BOOT_MODE0	Configuration
(BCBUS6)	(BCBUS5)	(BCBUS4)	
0	0	1	Slave serial configuration via FTDI
0	1	X	FTDI device pins connected to module I2C bus
1	0	0	SPI flash programming via FTDI
X	1	0	Master serial configuration (Mercury module is configured from SPI flash)
1	1	X	FTDI device pins connected to Mercury module UART pins

Figure 3.2.1: FTDI supported boot modes

The FTDI can easily be configured through the Enclustra MCT User Interface by plugging in the module to the PC. This tool is only available in Windows, so the module had to be configured in a PC different from the Ubuntu development PC.

The steps are as follows:

1. Download Enclustra MCT from the official webpage (`downloads.enclustra-.com`)
2. Connect the XU8 module to the ST1 motherboard and the micro USB port to the PC.
3. Make sure no other FTDI devices are connected to the computer, it can be done through a FTDI programming utility or just by checking that no other devices that have a FTDI chip are physically connected to any port which should be case if no other boards are connected to it.
4. Open the MCT and locate the Settings tab in the menu bar
5. Select Enable configuring any FTDI. (Note that when another FTDI device is attached, proceeding may brick it's pre-programmed functionality, that is why step 3 must be done).

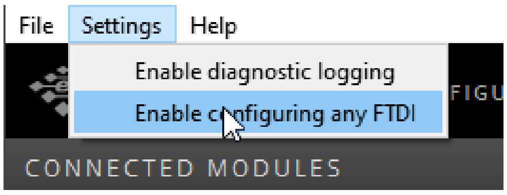


Figure 3.2.2: Enabling any FTDI configuration

6. After that click the Enumerate button

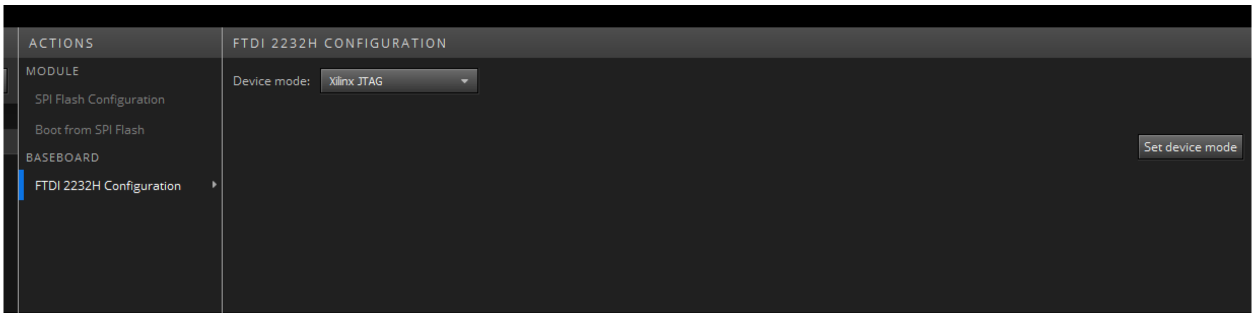
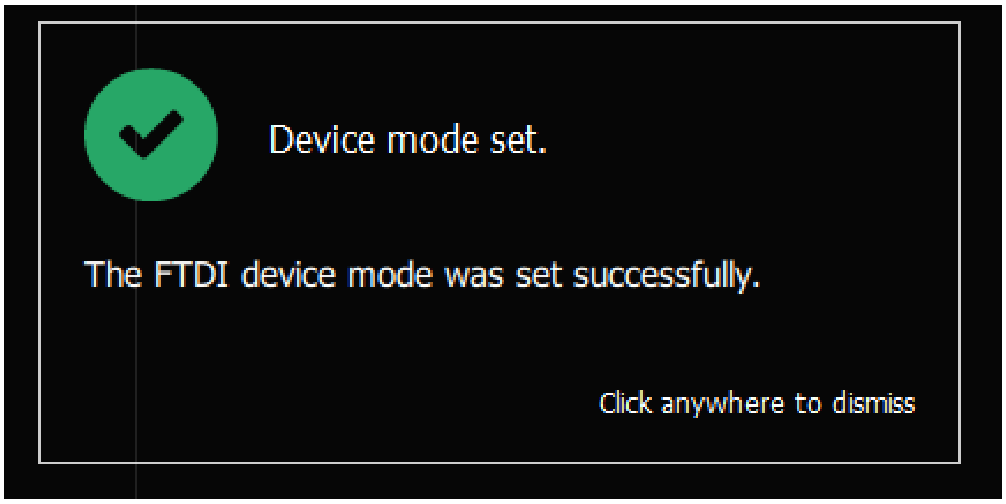


Figure 3.2.3: Enumerated board

7. In the Action pane, navigate to the FTDI Configuration section.
8. For the Device mode, select Xilinx JTAG
9. Press the Set device mode button and wait the process to complete, with the following message showing up in the software. The message of figure 3.2.4 appears indicating a correct configuration in JTAG mode. This will allow to program the board with a bitstream like the ZCU102.



Device mode set.

The FTDI device mode was set successfully.

Figure 3.2.4: Success message for configuring the module in JTAG mode

2.3 Boot from SD Card

Now that the FPGA has been correctly configured, a PetaLinux image needs to be loaded into it. Like in the ZCU102, there are several boot modes. In this section, the SD card method is the one used because the ST1+ baseboard has a microSD card slot that makes the process of swapping the kernel image very easy as only three files need to be copied.

The prerequisites for this boot mode are having the XU8 module attached to its ST1 base board and correctly configured as explained before.

The image to be mounted has been compiled in the Enclustra example project for this combination of SOM + Base board, available in its Github repository and has different configuration from the ZCU102 reference design reflected in the following files:

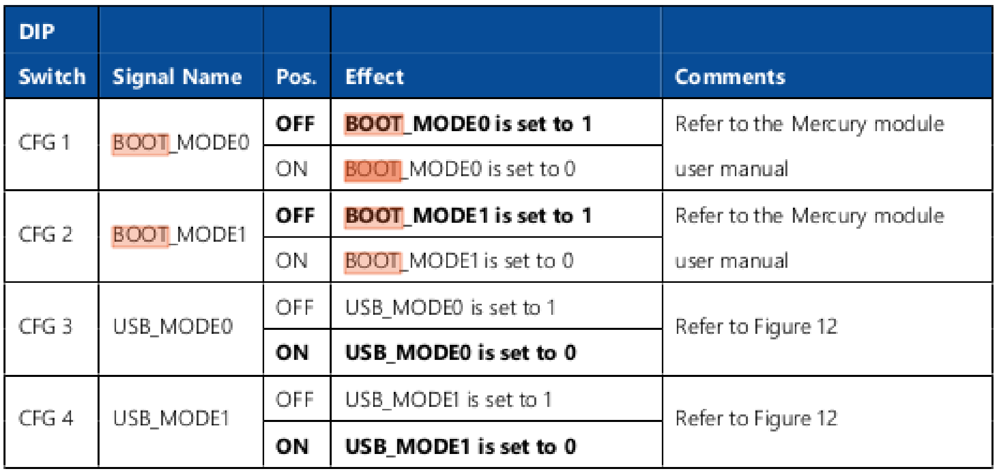
- **Vivado xsa:** two different platforms have completely different xsa files because everything in the hardware is different, from the FPGA Model to the peripherals attached to it. The Mercury+ XU8 and ST1 base board combination has much less ports than the ZCU102 and a different ping arrangement.
- **Device tree:** obviously, two different .xsa lead to two different device trees because the nodes in this file also change, going from the memory to the peripherals such as SFP, ethernet ports, SD card port, etc... may be configured in different ways.
- **File System:** the reference design has its rootfs (Root Filesystem) set to ext4 by default which means that the file system is stored in a external memory device such as a SD card. The ZCU102 rootfs type was initrd, which is stored in the main memory of the system, in this case the RAM connected to the PL.

For more information about what these files contain and their functions, refer to the MUISE TFM [1].

The steps to generate the image after configuring it with `petalinux-config` and building it with `petalinux-build` are depicted in the SD Boot Mode adjustments section in the PetaLinux Documentation of Enclustra repository based on the PetaLinux tools reference guide UG1144 from Xilinx:

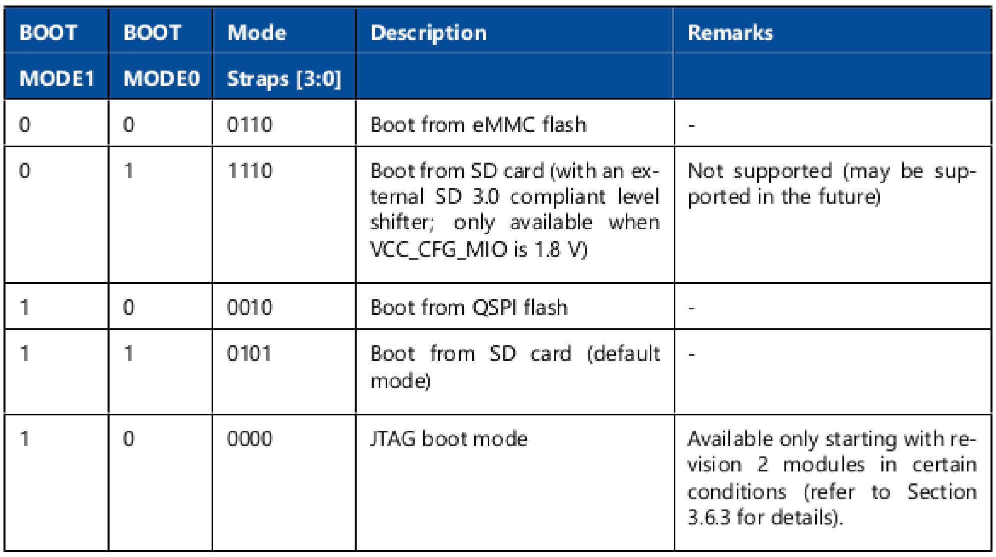
- Files needed for SD boot: (files are in PETALINUX_PROJECT_FOLDER -> images -> linux)
  - BOOT.BIN
  - boot.scr
  - system.dtb
  - Image
  - rootfs.tar.gz
- Create two partitions on the SD Card. The first partition is **FAT32** formatted, the second **EXT4**. UG1144 [20] describes how to create these partitions.
- Copy the files **BOOT.BIN, boot.scr, system.dtb and Image** to the first (FAT32) partition.
- Extract the file **rootfs.tar.gz** to the second partition: `sudo 'tar -xvf rootfs.tar.gz -C /path/to/second/partition'`
- Unmount all partitions and remove the SD card.

- Prepare the module and baseboard for SD boot mode by setting accordingly the DIP switches in the ST1 board. These two images represent the behaviour of the XU8 module related to the DIP switches in the ST1 base board. The DIP switches are connected to BOOT_MODE0 and BOOT_MODE1 pins in the module, which select the different boot modes available.



DIP Switch	Signal Name	Pos.	Effect	Comments
CFG 1	BOOT_MODE0	**OFF**	**BOOT_MODE0 is set to 1**	Refer to the Mercury module user manual
		ON	BOOT_MODE0 is set to 0	
CFG 2	BOOT_MODE1	**OFF**	**BOOT_MODE1 is set to 1**	Refer to the Mercury module user manual
		ON	BOOT_MODE1 is set to 0	
CFG 3	USB_MODE0	OFF	USB_MODE0 is set to 1	Refer to Figure 12
		ON	**USB_MODE0 is set to 0**	
CFG 4	USB_MODE1	OFF	USB_MODE1 is set to 1	Refer to Figure 12
		ON	**USB_MODE1 is set to 0**	

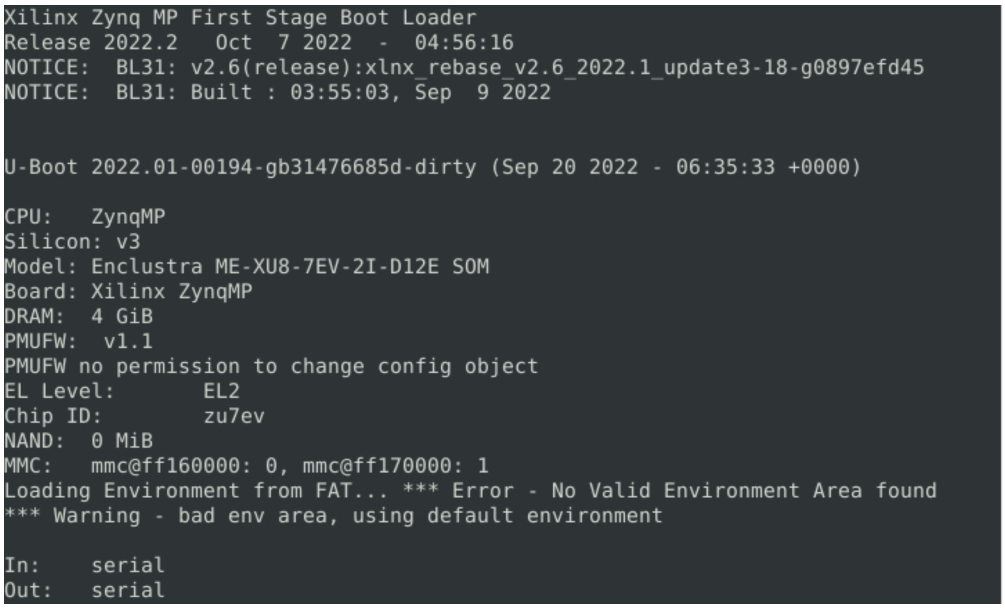
Figure 3.2.5: DIP switches behaviour in the ST1 base board



BOOT MODE1	BOOT MODE0	Mode Straps [3:0]	Description	Remarks
0	0	0110	Boot from eMMC flash	-
0	1	1110	Boot from SD card (with an external SD 3.0 compliant level shifter; only available when VCC_CFG_MIO is 1.8 V)	Not supported (may be supported in the future)
1	0	0010	Boot from QSPI flash	-
1	1	0101	Boot from SD card (default mode)	-
1	0	0000	JTAG boot mode	Available only starting with revision 2 modules in certain conditions (refer to Section 3.6.3 for details).

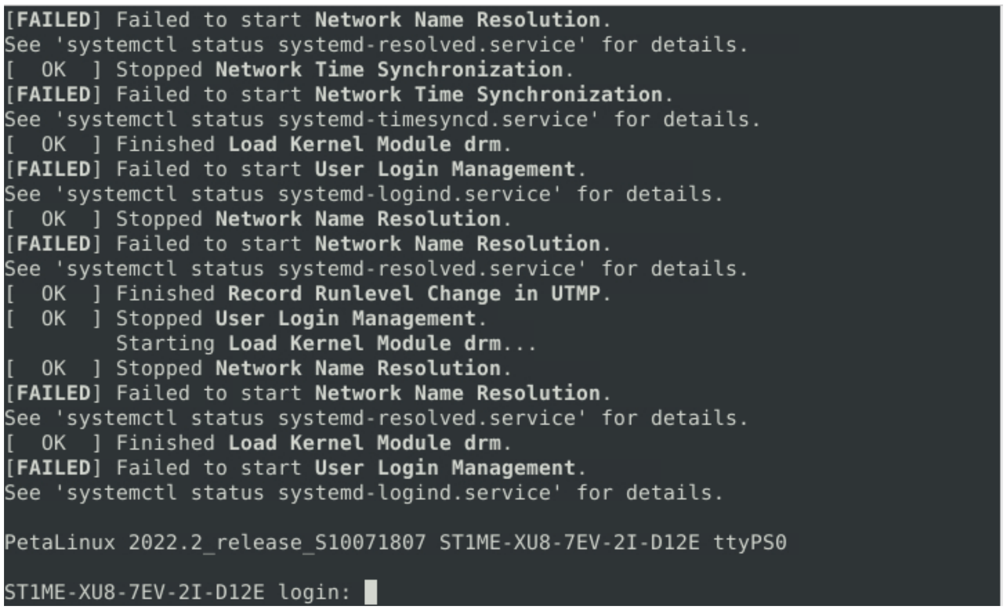
Figure 3.2.6: Mercury+ XU8 BOOT_MODE pins behaviour

- Insert the SD card.
- Plug in the USB cable and connect a serial terminal.
- Plug in the power jack.
- Open the serial terminal and configure it to listen to the correct USB. In this case it is *ttyUSB1*, but it can vary depending on the devices connected to the PC. Figures 3.2.7 and 3.2.8 show the log of the booting U-Boot and Linux respectively. The default login for a PetaLinux project is:
  - **Username: root**
  - **Password: root**



```
Xilinx Zynq MP First Stage Boot Loader
Release 2022.2   Oct  7 2022  -  04:56:16
NOTICE:  BL31: v2.6(release):xlnx_rebase_v2.6_2022.1_update3-18-g0897efd45
NOTICE:  BL31: Built : 03:55:03, Sep  9 2022


U-Boot 2022.01-00194-gb31476685d-dirty (Sep 20 2022 - 06:35:33 +0000)

CPU:   ZynqMP
Silicon: v3
Model: Enclustra ME-XU8-7EV-2I-D12E SOM
Board: Xilinx ZynqMP
DRAM:  4 GiB
PMUFW:  v1.1
PMUFW no permission to change config object
EL Level:       EL2
Chip ID:        zu7ev
NAND:  0 MiB
MMC:   mmc@ff160000: 0, mmc@ff170000: 1
Loading Environment from FAT... *** Error - No Valid Environment Area found
*** Warning - bad env area, using default environment

In:    serial
Out:   serial
```

Figure 3.2.7: FSBL loading with the correct parameters. Notice the board model.



```
[FAILED] Failed to start Network Name Resolution.
See 'systemctl status systemd-resolved.service' for details.
[  OK  ] Stopped Network Time Synchronization.
[FAILED] Failed to start Network Time Synchronization.
See 'systemctl status systemd-timesyncd.service' for details.
[  OK  ] Finished Load Kernel Module drm.
[FAILED] Failed to start User Login Management.
See 'systemctl status systemd-logind.service' for details.
[  OK  ] Stopped Network Name Resolution.
[FAILED] Failed to start Network Name Resolution.
See 'systemctl status systemd-resolved.service' for details.
[  OK  ] Finished Record Runlevel Change in UTMP.
[  OK  ] Stopped User Login Management.
         Starting Load Kernel Module drm...
[  OK  ] Stopped Network Name Resolution.
[FAILED] Failed to start Network Name Resolution.
See 'systemctl status systemd-resolved.service' for details.
[  OK  ] Finished Load Kernel Module drm.
[FAILED] Failed to start User Login Management.
See 'systemctl status systemd-logind.service' for details.

PetaLinux 2022.2_release_S10071807 ST1ME-XU8-7EV-2I-D12E ttyPS0

ST1ME-XU8-7EV-2I-D12E login:
```

Figure 3.2.8: Login screen. Fails in the initialization of the service are due to the board not being connected to the Internet and thus not having DNS, DHCP, NTP and other services.

Notice that in PetaLinux 2022.2 the recommended log in is the username *petalinux* with a password to be established in the first boot. However, in this image, root is enabled by default which is useful for testing but should be disabled in the final build for security reasons. For a project of these features it must be evaluated whether it is worth to leave root enabled or not as this board can only be accessed through a local network built in the DAQ side.

2.4 Patching up the kernel. Bugfixing

The first problem arisen from updating from PetaLinux 2022.1 to 2022.2 is located in the Cadence Network Drivers, also known as macb. The problem was in an update from 2022.1 (the latest BSP version Enculstra has available in Github) to 2022.2. The ST1+ baseboard has two Ethernet interfaces, each one of it connected to a PHY chip which is in charge of performing the auto-negotiation and establishing the physical link with the other side of the Ethernet link. At the same time, these PHY chips are connected to the Gigabit Ethernet MAC (GEM) inside the PS, which are the controllers that provide ethernet functionality to Linux. The Gigabit Ethernet PHY chips used in the Mercury+ XU8 share the same MIO interface (76...77) for their MDIOs and is distributed to both PHYs through a level shifter. This poses a problem because, the MDIO is associated to one of the GEM and thus it is only up when that interface is up, so the order in which the interfaces go up is very important.

The code had to be modified by removing that conditional and generating a patch for this change to take effect during compilation:

```
From 1f35d45498fda8169daedbecb810b79fb354cf92 Mon Sep 17 00:00:00 2001
From: Alejandro Gomez <algogam@teleco.upv.es>
Date: Tue, 7 Feb 2023 21:39:39 +0100
Subject: [PATCH] arm:zynqmp:macb:mdio_probe

Change for the Enclustra SOM in 2022.2

Signed-off-by: Alejandro Gomez <algogam@teleco.upv.es>
---
drivers/net/ethernet/cadence/macb_main.c | 8 +-------
1 file changed, 1 insertion(+), 7 deletions(-)

diff --git a/drivers/net/ethernet/cadence/macb_main.c
 ↪  b/drivers/net/ethernet/cadence/macb_main.c
index 34bfcb991109..63ea96e13731 100644
--- a/drivers/net/ethernet/cadence/macb_main.c
+++ b/drivers/net/ethernet/cadence/macb_main.c
@@ -952,13 +952,7 @@ static int macb_mdiobus_register(struct macb bp)

of_node_put(dev_np);

- / Check if the MDIO producer device is probed */
- if (mdio_pdev && !dev_get_drvdata(&mdio_pdev->dev)) {
- platform_device_put(mdio_pdev);
- netdev_info(bp->dev, "Defer probe as mdio producer %s is not probed\n",
- dev_name(&mdio_pdev->dev));
- return -EPROBE_DEFER;
- }
+
platform_device_put(mdio_pdev);
return mdiobus_register(bp->mii_bus);
--
2.25.1
```

The reason for this error happening is as follows: In the Cadence Ethernet driver, called *macb* and located in `drivers/net/cadence/macb_main.c` within the kernel source three a new conditional if check was added in 2022.2. This conditional checked that, given an ethernet interface, the corresponding MDIO associated was already probed. This is the case of shared MDIO because if another ethernet interface sharing the same MDIO had been probed before, then its corresponding MDIO should have been probed. This poses a problem because, if that is the case, the driver is coded to report an error and not initialize that ethernet interface. The error printed in the console as follows:

```
macb ff0e0000.ethernet eth1: Defer probe as mdio producer axi is not probed
```

and then returns from the function *macb_mdiobus_register* with an error instead of returning the device to initialise eth1. I removed this if with a self-made patch reverting the change. If you go to the linux-xlnx repository and compare 2022.1 branch with 2022.2 branch you will see that the if conditional is not present in the previous version.

The Linux kernel updates are pushed to the official kernel source tree through email request which contains patches with the new features and bug fixes. This helps keeping track of when, how and who made these changes in a very detailed way. The lines of code that caused this error were added in the following patch:

`https://lore.kernel.org/lkml/YsSVqknDQxdWqfds@lunn.ch/t/`

For more information about how a MDIO bus works, refer to Part III, chapter 6.

Chapter 3

FMC expansion card for additional SFP slots

The ST1 base board only has one SFP. For this project there is a requirement of having redundant links through Fiber Optics so there is the need for more. The ST1 base board has a FMC expansion slot for a FMC card (standard pin out) so there is the possibility to introduce more ports of the desired type depending on the application.

For this project, there is a lack of SFP slots in the base board so a FMC card with SFP slots is ideal. Searching through the Xilinx Catalogue, the best FMC is one manufactured by Hi-Tech Global called the *HTG-FMC-X4SFP+* that includes 4 SFPs slots and all the circuitry for generating the clocks. This chapter will explain how it was plugged in and configured to another two extra ethernet interfaces through SFP

For making this FMC work with the SOM in PetaLinux there are several aspects to consider:

- **Routing of the pinouts:** Having the correct IC and ports in the FMC is just the beginning of the design of a FMC card. This interface has standard pinouts so that devices from different manufacturers can communicate. This means that there is the need to follow some rules in the routing of your extra peripherals located in the FMC board to the FMC slot. The standard defines several columns with letters from A to K where there are groups of pins that make up for differential pairs of data and differential pairs of clocks besides the Vcc supply, the power good LEDs and other indicators of FMC correct operations.
- **Speed required for the SFPs:** SFP have 1Gbps speed and SFP+ 10Gbps. From the signal integrity point of view, this might require special mechanisms in the FPGA in the form of MGT (Transceivers) for supporting these speeds. This decision must be made by the designer of the FPGA side because in the FMC standard there is no word about MGT or standard LVDS because these are FPGA-related concepts. FMC can be used in PCBs that don't have FPGA.
- **Configuration of the SFPs:** The FMC card is routed in some way to the FPGA. Depending whether the pins are routed to the PL or to the PS is crucial to know how our system should be programmed in Vivado. If it is possible to add IP Cores because it is PL connected or just use the PS GEM, which would need a manual configuration.

3.1 Resources of the FPGA for SFP connection

SFP is an adapter for connecting network cables to the board. Network interfaces are very high speed, being right now 1Gbps the defacto standard but arriving up to 20 Gbps even 40Gbps in the latest beta-state technology. These speeds require special devices in the FPGA to handle such a high speed channel. For Xilinx, those are MGT (MultiGigabit Transceivers), which receive both different pairs for reception and transmission and a clock. MGTs include mechanisms to equalize the channel by measuring delays and recovering the clock so that a high speed communication can be performed.

Mercury XU8+ FPGA, the ZCU7ev, includes 4 MGT banks with 4 MGT each in the PL and another MGT bank with another 4 MGT for the PS, so 20 Transceivers in total. This project requires a lot of these transceivers because there is going to be 6 SFPs in the final prototype and 2 MiniSAS connectors with two channels each, accounting for 10 high speed interfaces.

3.2 Pinout of all the implied boards

The FMC card is shown in figure 3.3.1.

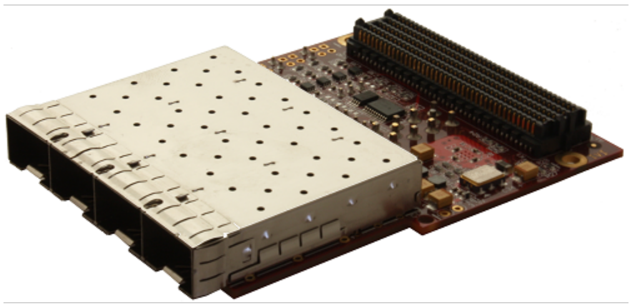


Figure 3.3.1: HTG-FMC-x4SFP+ Mezzanine card. This card has a standard connector to provide the FPGA with extra functionality

This design is formed by three different boards put together so a lot of care must be put into configuring the correct I/O pins in Vivado and ensure that the required inputs and outputs are correctly routed from a hardware perspective. Else, even if software is correct, the FO connection will not work because physical connections are not made the way they should be.

Fortunately, FPGA Mezzanine Card (FMC) is a standard connector with 400 pins in which each pin has a given function, ranging from high speed serial interfaces, clocks to slow control I2c or SPI buses. There is also room for user defined signals in case some special functionality is wanted to be leveraged with FMC of the same manufacturer as the main board. This means that all manufacturers are aware of in which pins they should expect high speed interfaces or clocks so that they route their own chips to those pins to be compatible.

Considering this and just by looking at the schematics of the three boards (HTG-FMC-x4SFP+, ST1+ base board and Mercury XU8+) the traceability from the SFP port to the FPGA is very clear. There are two interfaces that need to go from the FMC to the FPGA for SFPs to work and both are related to MGT (MultiGigabit Transceivers), because these are the modules within the FPGA that receive the SFP signals:

- Gigabit Transceiver Transmission and Reception differential pairs: Taking SFP3 as an example. Looking at the three schematics, the following path is routed:

  ```
  SFP3 -> DP0_M2C (FMC) -> Module Connector B in ST1+ -> MGT_BD_0 (Mercury
  XU8+)
  ```

  So, there is a direct connection between the SFP3 to a MGT in the FPGA, specifically in MGT Bank D. If the same procedure is followed for the other SFPs, all of them arrive to the same MGT bank, occupying all the 4 slots available in the bank, which is to be expected.

- Gigabit Transceiver Clock: MGTs need a clock to work. These clocks must be fed to the corresponding input of the corresponding MGT bank that the SFP are connected to. In any MGT there are two differential pairs for clock input. If the routing is done from the FMC Reference clock 0, the following path is obtained:

  ```
  FMC_REF_CLK0 -> GBTCLK0_M2C (FMC) -> Module connector B(FMC_GCLK0_M2C) ->
  MGT_BD_REFCLK0
  ```

  This means that the FMC outputs a reference clock to input CLK0 from Bank D of MGT.

The schematics made by the manufacturers of the FMC card, the ST1+ baseboard and the Mercury XU8+ SOM depict the trace followed below to connect the SFP slots to the MGTs and the clock. These schematics are included as additional deliverables in this TFM submission.

3.3 PS or PL configuration?

SFPs can be routed to MGT located in the PS or in the PL and, depending on that, the configuration of PetaLinux is different:

- **If routed directly to the Processing System (PS):** this would imply that the MGT has a direct connection through its TX and RX differential pairs to one of the GEM in the PS. As this is a direct connection with no FPGA in between, no new hardware acting as a bridge can be instantiated to act as interface. Then it is crucial that the physical layer sported in the SFP is supported by the GEM. This is the case: reading through the Zynq Ultrascale+ Technical Reference Manual, it is listed that the 1000 Base-X standard is supported. Some additional configuration like establishing a fixed 1Gbps link needs to be done on the device tree. Further comments will be made when working on the DPB2 prototype.

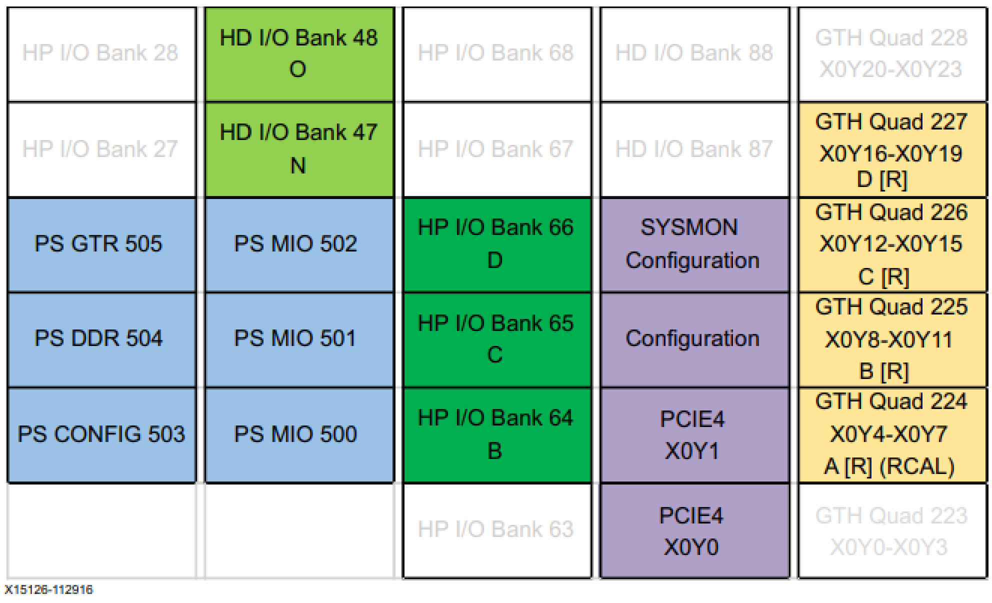
- **If routed to the Programmable Logic (PL):** if MGT Bank belongs to the PL, then an ethernet core can be instantiated to act as an interface between GEM in RGMII mode and the 1000Base-X SFP, allowing for autonegotiation thanks to the core acting as a PHY chip. The core used for this case is *1G/2.5G Ethernet PCS/PMA or SGMII v16.2 LogiCORE IP*, the same used for the ZCU102, because it provides a monolithic linux driver for ease of use with one of the GEM in the PS.

The PL case is the one treated here as MGT Bank D is a MGT bank connected to the PL. The procedure for configuring both Vivado (hardware) and PetaLinux (Software) is described in the following sections.

3.4 Vivado configuration

Departing from the Enclustra Reference Design, the *1G/2.5G Ethernet PCS/PMA or SGMII v16.2 LogiCORE IP* has been instantiated and configured as follows:

- **Ethernet MAC:** Zynq PS Gigabit Ethernet Controller, for using it with the PS thanks to a PetaLinux driver.
- **Standard:** 1000BASEX. Used for fiber optics. The SFP used is a Cisco GLC-X compliant with this standard.
- **Core Functionality**:
  - Physical Interface: Device Specific Transceiver, to use the MGT that the FPGA has.
  - Reference Clock Frequency: 156.25 MHz, the same as in the ZCU102 project.
  - Transceiver Location: This is a very sneaky option because it is not easy to find in the documentation. Xilinx FPGAs are divided into tiles given a X and Y position and depending on the part number and packaging these numbers change. This data can be extracted from UG1075, page 52, figure 1-19 which corresponds to the XCZU7 in FBVB900 packaging, which is the chip included in the Mercury XU8+.
  - DRP Clock Frequency: 50 MHz, extracted from a PS PLL.



HP I/O Bank 28	HD I/O Bank 48 O	HP I/O Bank 68	HD I/O Bank 88	GTH Quad 228 X0Y20-X0Y23
HP I/O Bank 27	HD I/O Bank 47 N	HP I/O Bank 67	HD I/O Bank 87	GTH Quad 227 X0Y16-X0Y19 D [R]
PS GTR 505	PS MIO 502	HP I/O Bank 66 D	SYSMON Configuration	GTH Quad 226 X0Y12-X0Y15 C [R]
PS DDR 504	PS MIO 501	HP I/O Bank 65 C	Configuration	GTH Quad 225 X0Y8-X0Y11 B [R]
PS CONFIG 503	PS MIO 500	HP I/O Bank 64 B	PCIE4 X0Y1	GTH Quad 224 X0Y4-X0Y7 A [R] (RCAL)
		HP I/O Bank 63	PCIE4 X0Y0	GTH Quad 223 X0Y0-X0Y3

Figure 3.3.2: XCZU7 and XAZU7 Banks in FBVB900 Package and XQZU7 Banks in FFRB900 Package

- **Shared Logic:** Include Shared Logic in Core. The rest of the options are left in its default state.

The IP ports (figure 3.3.3) can be divided into three big groups:

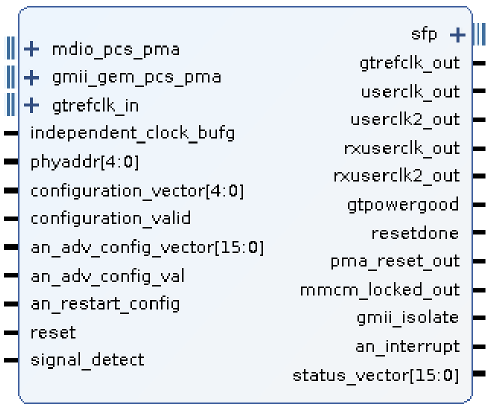
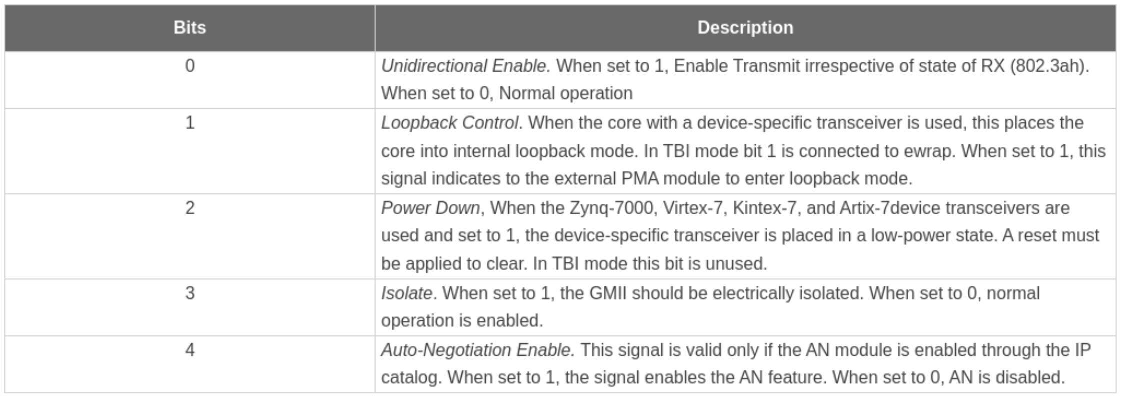


Figure 3.3.3: 1G/2.5G Ethernet PCS/PMA Core using Device Transceivers and including Shared Logic

- **Transceiver I/O:** they are connected directly to the MGTs. Thus, they have very strict placing and routing constraints to ensure the promised performance. Thanks to Transceiver location option in the configuration, these interfaces are automatically assigned to the corresponding pins of the MGT. In this case, as transceiver X0Y16 was chosen the pins assigned where the ones corresponding to MGT 0 from Bank D:
- **Zynq PS I/O:** The communication with the GEM1, configured to EMIO mode:
  - *mdio_pcs_pma:* MDIO interface, connected to the equivalent MDIO_ENET1
  - *gmii_gem_pcs_pma:* GMII interface for data transfer, conneceted to its equivalent interface *GMII_ENET1* in the ZynqMP PS.
  - *independent_clock_bufg:* conneceted to one of the Low power clocks of the ZynqMP PS that is outputed to the PL. In this case PL1 clock has been configured to output 50 MHz. The name of the output is *pl_clk1*
- **Configuration I/O:** this core is configured through several inputs: -
  - *phyaddr[4:0]*: corresponding to the address that this core, behaving as a PHY chip will have. It is very important to indicate the same address to the PetaLinux driver, so that it can be detected.
  - *configuration vector[4:0]*: 5 bits for configuring many things shown in the table in figure 3.3.4.

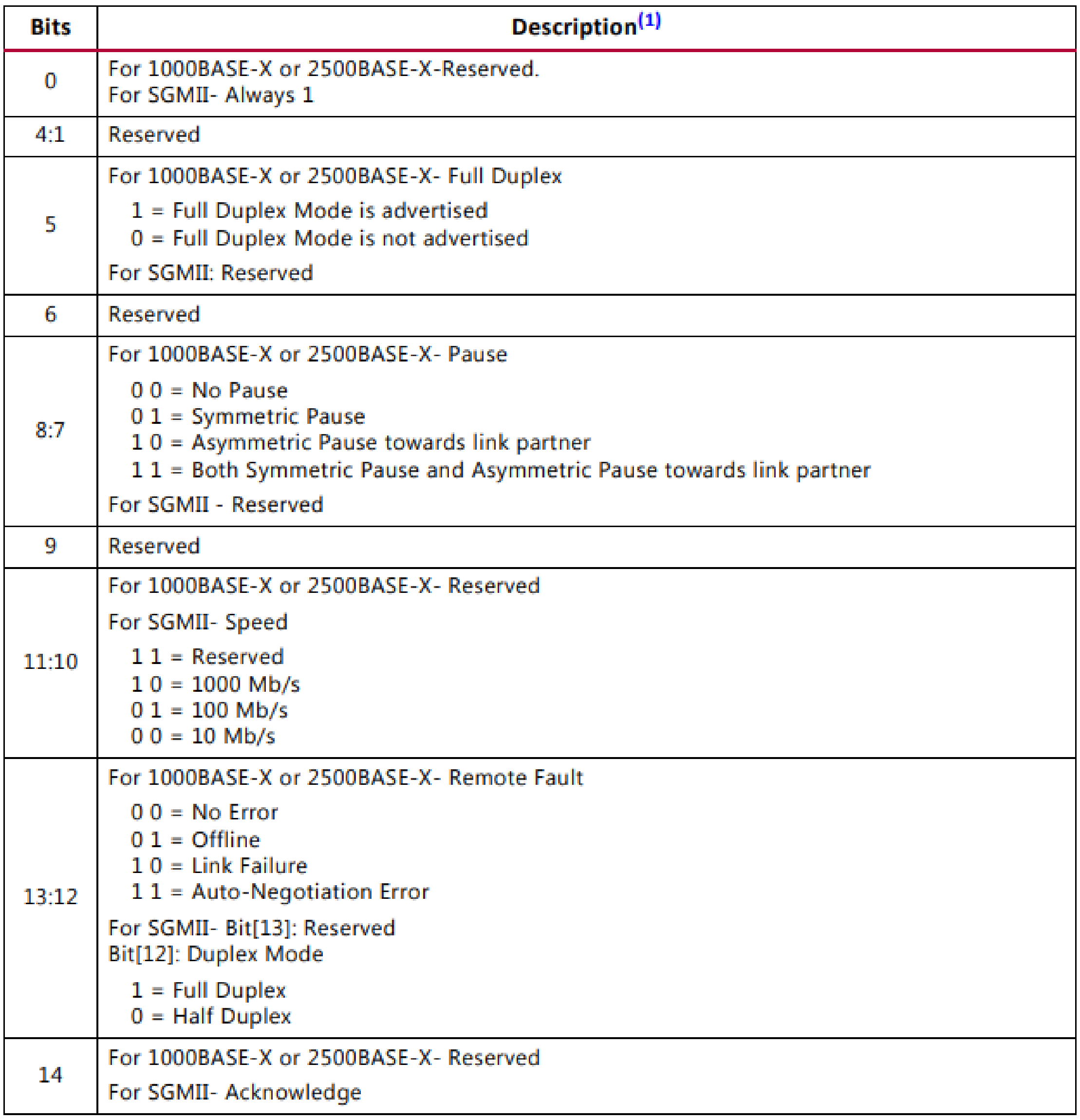


Bits	Description
0	*Unidirectional Enable.* When set to 1, Enable Transmit irrespective of state of RX (802.3ah). When set to 0, Normal operation
1	*Loopback Control*. When the core with a device-specific transceiver is used, this places the core into internal loopback mode. In TBI mode bit 1 is connected to ewrap. When set to 1, this signal indicates to the external PMA module to enter loopback mode.
2	*Power Down*, When the Zynq-7000, Virtex-7, Kintex-7, and Artix-7device transceivers are used and set to 1, the device-specific transceiver is placed in a low-power state. A reset must be applied to clear. In TBI mode this bit is unused.
3	*Isolate*. When set to 1, the GMII should be electrically isolated. When set to 0, normal operation is enabled.
4	*Auto-Negotiation Enable.* This signal is valid only if the AN module is enabled through the IP catalog. When set to 1, the signal enables the AN feature. When set to 0, AN is disabled.

Figure 3.3.4: Configuration Vector of the 1G/2.5G PCS/PMA IP Core

– *configuration_valid* signal to indicate that the configuration given to the device is valid and can be read.

– *an_adv_config_vector[15:0]*: another configuration vector with aspects about auto negotiation. The meaning of each bit is show in table in figure 3.3.5.



Bits	Description[(1)]
0	For 1000BASE-X or 2500BASE-X-Reserved. For SGMII- Always 1
4:1	Reserved
5	For 1000BASE-X or 2500BASE-X- Full Duplex 1 = Full Duplex Mode is advertised 0 = Full Duplex Mode is not advertised For SGMII: Reserved
6	Reserved
8:7	For 1000BASE-X or 2500BASE-X- Pause 0 0 = No Pause 0 1 = Symmetric Pause 1 0 = Asymmetric Pause towards link partner 1 1 = Both Symmetric Pause and Asymmetric Pause towards link partner For SGMII - Reserved
9	Reserved
11:10	For 1000BASE-X or 2500BASE-X- Reserved For SGMII- Speed 1 1 = Reserved 1 0 = 1000 Mb/s 0 1 = 100 Mb/s 0 0 = 10 Mb/s
13:12	For 1000BASE-X or 2500BASE-X- Remote Fault 0 0 = No Error 0 1 = Offline 1 0 = Link Failure 1 1 = Auto-Negotiation Error For SGMII- Bit[13]: Reserved Bit[12]: Duplex Mode 1 = Full Duplex 0 = Half Duplex
14	For 1000BASE-X or 2500BASE-X- Reserved For SGMII- Acknowledge

Figure 3.3.5: Auto Negotiation vector of the 1G/2.5G PCS/PMA IP Core

– *an_adv_config_val*: equivalent to *configuration_valid* but for the *an_adv_config_vector*.

– *reset*: the typical reset signal at system startup (low level).

– *signal_detect*: tied to 1 as explained in the constants table.

Regarding the configuration signals, they can be fixed to some values, considering how the core is configured. This is done in the same block diagram through the constant block, depicted in figure 3.3.6.

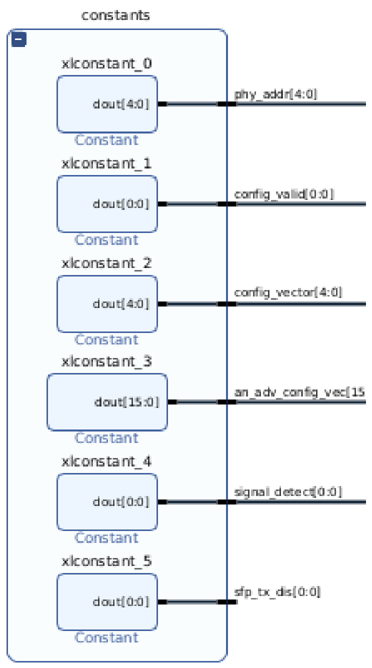


Figure 3.3.6: Constants Cores instantiated for each of the configuration inputs of the PCS/PMA Ehternet Core

Constants are configured as shown in table 3.2.

Configuration input	Value	Description of the functionality
phy_addr[4:0]	9	The PHY address. This address is used to identify a chip in the MDIO bus. As this core resembles a PHY chip, it needs to have a phy address
config_valid	0	With a rising edge of this signal, the configuration inside the core is overwritten However, this signal is not used so it will be left at 0
config_vector[4:0]	0	When the 5 bits are set to zero it means that the core is under Normal operation, no loopback mode powered up, not isolated and no autonegotiation. Autonegotiation has been disabled because we already know that the link speed will be 1 Gbps.
an_adv_config_vector	55297	This value is irrelevant because Autonegotiation is disabled. This is the default value that the Product Guide of the core recommends
signal_detect	1	This signal should be connected to the signal_detect of the SFP. However, it isn't available in any pin of the FPGA so, it is tied to 1 to indicate that light is always being detected as recommended by the Product Guide

Table 3.2: Constants configuration and description for the Ethernet FPGA core

The rest of the signals are optional and just serve for debug purposes or for chaining several cores that share the same MGT bank thanks to the *gtrefclk_out* port that would be connected to the next *gtrefclk_in* to, for example, connecting the 4 SFPs that with this routing go to MGT bank D.

```
-- Module declaration
FMC_DP0_M2C_P                  : in   std_logic;
FMC_DP0_M2C_N                  : in   std_logic;
FMC_DP0_C2M_P                  : out   std_logic;
FMC_DP0_C2M_N                  : out   std_logic;
FMC_GCLK0_M2C_N                : in   std_logic;
FMC_GCLK0_M2C_P                : in   std_logic;

-- Block diagram instantiation
sfp_rxn : in STD_LOGIC;
sfp_rxp : in STD_LOGIC;
sfp_txn : out STD_LOGIC;
sfp_txp : out STD_LOGIC;
gtrefclk_in_clk_n : in STD_LOGIC;
gtrefclk_in_clk_p : in STD_LOGIC;

-- Pin assignment

sfp_rxn              => FMC_DP0_M2C_N,
sfp_rxp              => FMC_DP0_M2C_P,
sfp_txn              => FMC_DP0_C2M_N,
sfp_txp              => FMC_DP0_C2M_P,
gtrefclk_in_clk_n    => FMC_GCLK0_M2C_N,
gtrefclk_in_clk_p    => FMC_GCLK0_M2C_P,
```

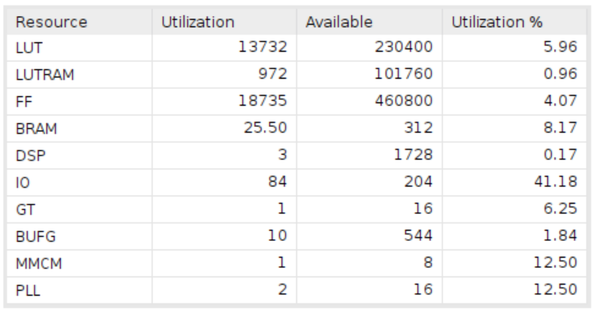
Else, Vivado will not be able to assign the properties for constraining the location of the pins because, they do not match the direction of the MGT ports, giving the following critical warning that will result in an error during the placement stage.

```
[Vivado 12-2285] Cannot set LOC property of instance
 ↪ 'Mercury_XU8_i/gig_ethernet_pcs_pma_0/U0/pcs_pma_block_i/transceiver_inst/
Mercury_XU8_gig_ethernet_pcs_pma_0_0_gt_i/inst/gen_gtwizard_gthe4_top.
Mercury_XU8_gig_ethernet_pcs_pma_0_0_gt_gtwizard_gthe4_inst/
gen_gtwizard_gthe4.gen_channel_container[4].gen_enabled_channel.gthe4_channel_wrapper_inst/
channel_inst/gthe4_channel_gen.gen_gthe4_channel_inst[0].GTHE4_CHANNEL_PRIM_INST'...
The pin direction of site GTHE4_CHANNEL_X0Y16 with package pin D2 does not match
 ↪ the given terminal FMC_DP0_M2C_P
```

This is the XDC script that is autogenerated when the core is instantiated and does the placement and routing automatically without user intervention.

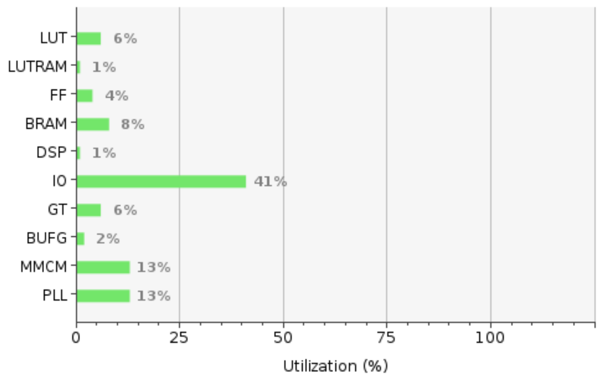
```
# Channel primitive location constraint
set_property LOC GTHE4_CHANNEL_X0Y16 [get_cells -hierarchical -filter {NAME =~
 ↪ *gen_channel_container[4].*gen_gthe4_channel_inst[0].GTHE4_CHANNEL_PRIM_INST}]

# Channel primitive serial data pin location constraints
# (Provided as comments for your reference. The channel primitive location
 ↪ constraint is sufficient.)
#set_property package_pin D1 [get_ports gthrxn_in[0]]
#set_property package_pin D2 [get_ports gthrxp_in[0]]
#set_property package_pin D5 [get_ports gthtxn_out[0]]
#set_property package_pin D6 [get_ports gthtxp_out[0]]
```

The final design resources usage is depicted in figures 3.3.7a and 3.3.7b. These numbers come from the Vivado report after placing and routing the design:



Resource	Utilization	Available	Utilization %
LUT	13732	230400	5.96
LUTRAM	972	101760	0.96
FF	18735	460800	4.07
BRAM	25.50	312	8.17
DSP	3	1728	0.17
IO	84	204	41.18
GT	1	16	6.25
BUFG	10	544	1.84
MMCM	1	8	12.50
PLL	2	16	12.50

(a) Resources usage table



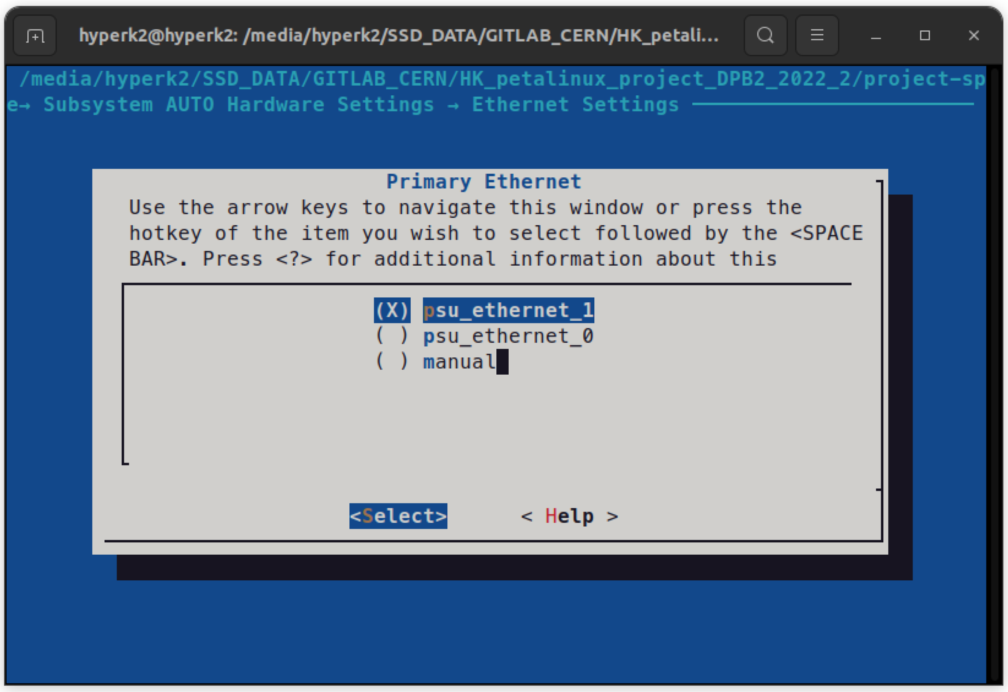
(b) Occupation percentage of the FPGA resources

Finally, the bitstream was generated and the hardware was exported as .xsa file to be read by PetaLinux.

3.5 PetaLinux configuration

The configuration in PetaLinux is rather simple, as there is no change from the ZCU102:

1. `petalinux-config --get-hw-description hardware/SFP_1`. This command changes the platform of the project to the one of the .xsa contained in SFP_1 folder.
2. Check inside the configuration if psu_ethernet_1 is detected.



```
hyperk2@hyperk2: /media/hyperk2/SSD_DATA/GITLAB_CERN/HK_petali...
 /media/hyperk2/SSD_DATA/GITLAB_CERN/HK_petalinux_project_DPB2_2022_2/project-sp
e→ Subsystem AUTO Hardware Settings → Ethernet Settings ──────────────────────

                              Primary Ethernet
      Use the arrow keys to navigate this window or press the
      hotkey of the item you wish to select followed by the <SPACE
      BAR>. Press <?> for additional information about this

                         (X) psu_ethernet_1
                         ( ) psu_ethernet_0
                         ( ) manual

                      <Select>      < Help >
```

Figure 3.3.8: Selecting GEM1 Ethernet in `petalinux-config` menu

3. Modify the device tree to include gem1. An example device tree is included in the annex. as a node for configuring the driver to enable the network interface.

4. Execute `petalinux-build` and `petalinux-package` to create the bootable image.

3.6 Results and issues

When booting PetaLinux, however, the results were far from the expected. In the U-Boot screen, there was an error message saying `Could not get PHY for eth3: addr 9`, which means that the phy was not detected. In U-boot there are other commands such as `mii info` or `mdio` that give information about the devices connected that make use of these interfaces. No MDIO was detected neither for address 9, so it was assumed that a failure in the hardware bootup occured. By using the debug signal *resetdone* and *gtpowergood* in two of the board's LEDs, it was discovered that while the latter lit up indicating that the MGT were working, the *resetdone* didn't, which meant that either the reset signal was failing to be asserted or there was no clock at all in the device, disabling it to perform any change in the *resetdone* signal. The reset was checked and the GPIOs were delivering the reset signal as the set up for those was the same as in the ZCU102, so the problem was that the clock was not being generated.

The clock generation was done directly in the FMC through a Si570, a low jitter clock generator IC designed by Skyworks Inc. This IC is also included in the ZCU102 and configured through I2C by the driver included in the Linux Kernel. This means that there is a reference design that can be departed off.

To perform the Si570 configuration through the I2C bus, the following steps were done:

1. **Routing of the FMC:** looking in the ST1 user manual, there is a block diagram that shows that the I2C1 controller included in the PS is connected to several I2C slaves through the PL, as indicated in figure 3.3.9.

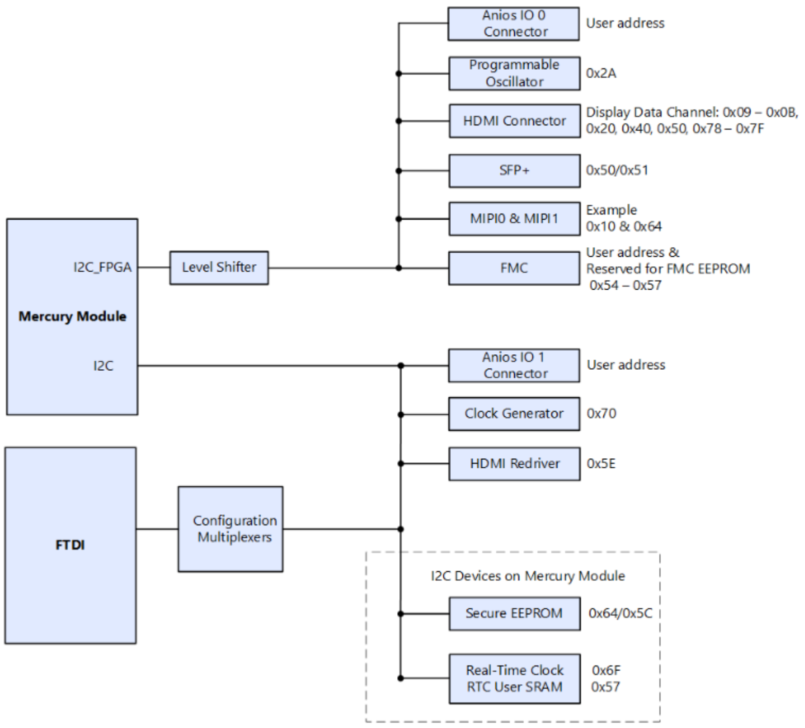


Figure 3.3.9: I2C devices connnected to the SOM and the FTDI module

One of the devices that have I2C slave connection is the FMC card, with addresses 0x54 to 0x57 reserved for any I2C slaves that can be plugged through FMC. So the first step would be to check in Vivado that the I2C1 is connected through EMIO to the corresponding pins of the FPGA for SCL and SDA and connecting them in the vhdl top file to the corresponding *inout* ports, that use pins K14 for I2C_SDA_FPGA and K15 for I2C_SCL_FPGA.

```
 -- I2C ports declaration
 I2C_SCL_FPGA                   : inout   std_logic;
 I2C_SDA_FPGA                   : inout   std_logic;
-- I2C from the block diagram instantiation
 IIC_FPGA_sda_i      : in     std_logic;
   IIC_FPGA_sda_o      : out    std_logic;
   IIC_FPGA_sda_t      : out    std_logic;
   IIC_FPGA_scl_i      : in     std_logic;
   IIC_FPGA_scl_o      : out    std_logic;
   IIC_FPGA_scl_t      : out    std_logic;
-- Buffering of the IIC pins to create the inout of the ports
component IOBUF is
 port (
   I : in STD_LOGIC;
   O : out STD_LOGIC;
   T : in STD_LOGIC;
   IO : inout STD_LOGIC
 );
  end component IOBUF;
```

This changes must be included in Vivado and do the workflow again to obtain another .xsa file.

2. **PetaLinux driver:** now that the hardware has been exported, PetaLinux needs to know which device is being connected. This is specified in the device tree. In `project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi`, the following node was added:

```
&i2c1 {
    status = "okay";
    clock-frequency = <400000>;
    si570: clock-generator@55 {
        #clock-cells = <0>;
        compatible = "silabs,si570";
        reg = <0x55>;
        temperature-stability = <20>;
        factory-fout = <156250000>;
        clock-frequency = <156250000>;
    };
};
```

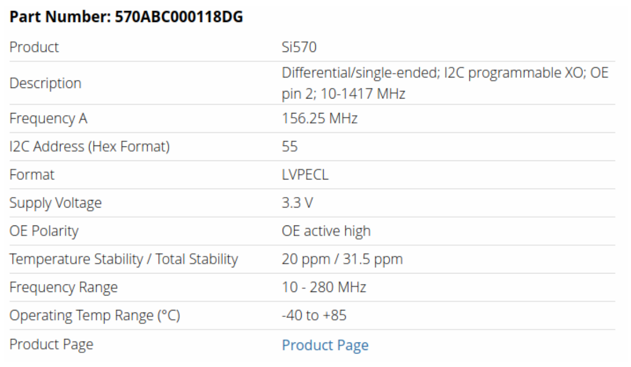
The meaning of this node is that, now, i2c1 master controller is enabled with a clock frequency for communicating with the slaves of 400 kbps. Its only child node, that is, the only i2c1 device we want PetaLinux to see is the si570, a clock-generator with its properties detailed. The documentation for filling this properties is located in the Linux source code in

`Documentation/devicetree/bindings/clock/silabs,si570.txt`

Where the following information about the nodes is presented:

- compatible: Shall be one of “silabs,si570”, “silabs,si571”, “silabs,si598”, “silabs,si599”
- reg: I2C device address.
- #clock-cells: From common clock bindings: Shall be 0.
- factory-fout: Factory set default frequency. This frequency is part specific. The correct frequency for the part used has to be provided in order to generate the correct output frequencies. For more details, please refer to the data sheet. - temperature-stability: Temperature stability of the device in PPM. Should be one of: 7, 20, 50 or 100.
- clock-frequency: Output frequency to generate. This defines the output frequency set during boot. It can be reprogrammed during runtime through the common clock framework.

Here, we can find three specifications that are factory set: *factory-fout*, which is the reference startup frequency when the oscillator powers up, *reg*, as the 7 bit I2C address of the device and the *temperature-stability*. For knowing these three specs, Skyworks provides a webpage in which, given the exact part number of your chip, you get the specs. In this case, the chip had the following designation: *570ABC000118DG* which provided the table seen in figure 3.3.10.


Part Number: 570ABC000118DG

Product	Si570
Description	Differential/single-ended; I2C programmable XO; OE pin 2; 10-1417 MHz
Frequency A	156.25 MHz
I2C Address (Hex Format)	55
Format	LVPECL
Supply Voltage	3.3 V
OE Polarity	OE active high
Temperature Stability / Total Stability	20 ppm / 31.5 ppm
Frequency Range	10 - 280 MHz
Operating Temp Range (°C)	-40 to +85
Product Page	Product Page

Figure 3.3.10: Si570 specs included in the HiTech Global FMC

So, the I2C address is 0x55, the temperature stability is 20 and the factory frequency is 156.25 MHz. A more detailed meaning of each of the digits in the part number is depicted in the Si570 datasheet, shown in figure 5.1.1 in the annex.

When loading the new OS version, the same error happened. The Si570 wasn’t outputing any clock even though it was being configured to do so. The reason for this was in the FMC card Schematic:

- **Routing of the FMC:** The pins assigned to the I2C interface for the FMC were not connected to the Si570, but to a EEPROM Memory in the FMC. This wasn’t very apparent because, as shown in this figure, the Si570 had two SCL and two SDA buses called *OSC_SCL*, *OSC_SDA* and *SCL* and *SDA* (the ones used for I2C configuration and shared with the EEPROM memory). Inspecting the schematic closer, the *SCL* and *SDA* pins, which were the ones being used for I2C configuration had a NS label in its nets. This means *No stuff*, that is, the connection wasn’t made in the PCB so *SCL* and *SDA*, the pins used for I2C configuration from the SOM weren’t the same as the ones used in the FMC for configuring the Si570.

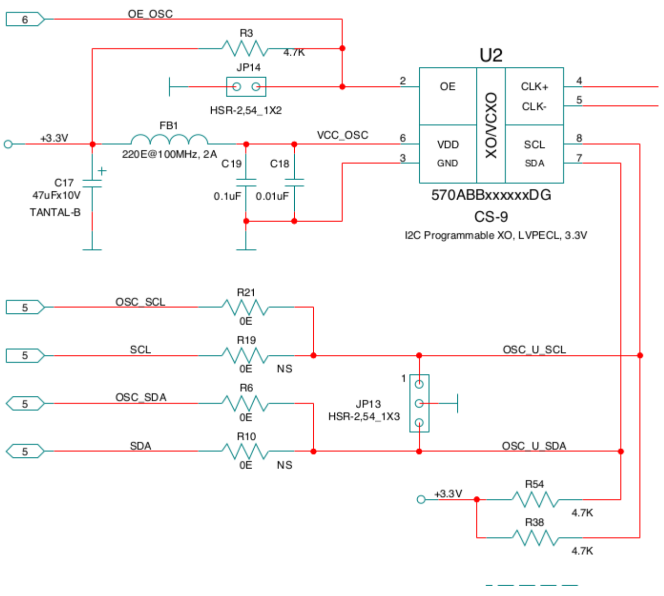


Figure 3.3.11: Si570 No Stuff connection of SCL and SDA

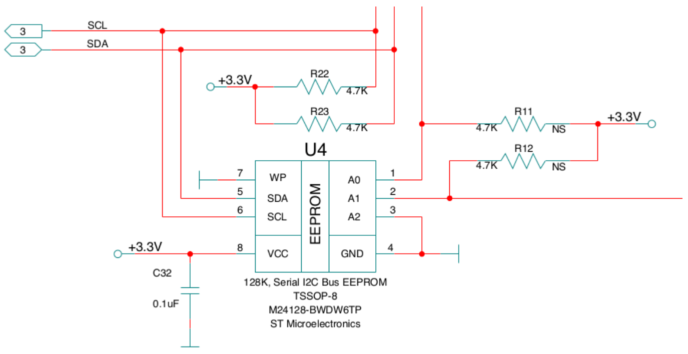


Figure 3.3.12: Notice that EEPROM has the SCL and SDA pins that were though to be "shared" with Si570

- **Voltage problems:** Having tracked down the issue, the solution was simple: just use the other pins named *OSC_SCL* and *OSC_SDA* to perform the I2C configuration. This wasn't possible because the pins in the FMC routed to these ports were LA_05_P and LA_05_N, which were routed through the ST1+ Base board to the Mercury XU8+ ZynqMP chip and connected to the FPGA Bank HP 66, which is a bank that has a max voltage of 1.8V. The Si570 in the FMC needs at least 0.75 x Vdd in SCL or SDA to consider them a high level (denoted in the datasheet as ViH) so this approach is not valid either.

- **Use alternative clock:** not being able to use that MGT clock input because there is no way to configure the oscillator in the FMC side leaves us with only one option: use the other clock input of MGT Bank D: *MGT_BD_REFCLK1*. This input is connected to another clock generator inside the ST1+ Base board: the SI5338B-B-GMR, depicted in figure 3.4.1.

This is a feasible solution because the clock is no longer generated inside the FMC. However, this specific chip does not have driver support in the Linux kernel, so a new driver that programs the registers inside the chip through I2C to output the desired frequency is needed. Coding a driver is not trivial, thus the first option was to search for a driver in the kernel.

Chapter 4

Adding a non-standard driver to mainline kernel

This chapter is a continuation in solving the problem laid out in chapter 3 towards getting a redundant optical fiber link working thanks to the extra SFP provided by the FMC in chapter 5. In this chapter, the problem of having a clock routed to the correct MGT is tackled: instead of using the Si570 of the FMC whose i2c lines are not routed to the FPGA, a PLL already included in the ST1+ baseboard will be used. (schematic in figure 3.4.1). However a suitable driver for this PLL is not included by default in the Linux kernel.

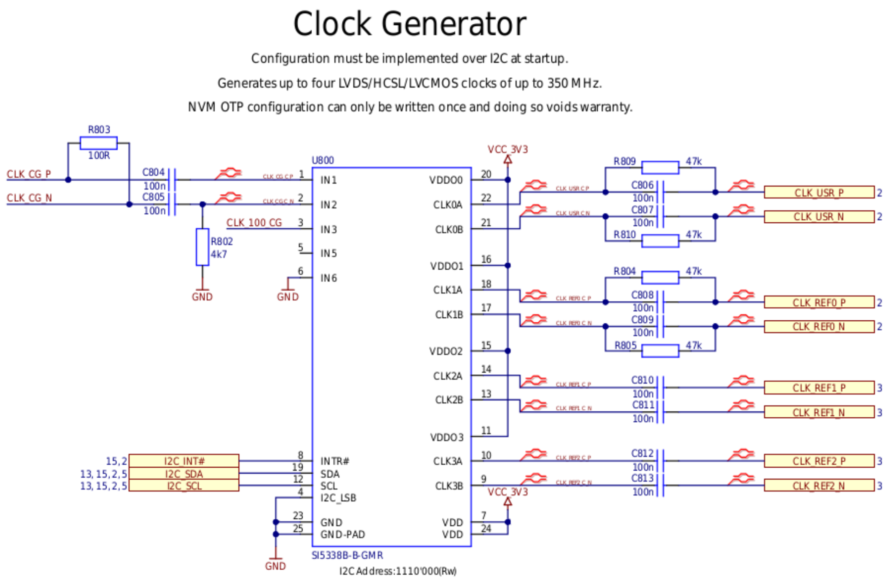


Figure 3.4.1: Si5338 Clock Generator Schematic. CLK_REF0 is routed to the MGT Bank D

Xilinx mantains their own kernel version of linux in their Github repository called linux-xlnx , where the kernel is pulled for PetaLinux compilation. Of course there are a lot of versions of PetaLinux and each one is associated to a specific version of the linux kernel. To know which version (or state of the kernel in the repository) is PetaLinux 2022.2 using one can go to the PetaLinux 2022.2 - Product Update Release Notes and Known Issues and search for the associated Git Tag of the component called *linux-xlnx*. In the PetaLinux 2022.2, the kernel being used is marked with tag *xlnx_rebase_v5.15_LTS_2022.2*.

Looking in the `/drivers/clk` which is the folder where the drivers for clock generators are, there was no driver for the Si5338 as there was not any source file with this name and no other driver for other Skyworks clock generators was compatible with this specific ASIC. One could think that the search is over, as there is no driver and thus no configuration that can be done to set an output frequency from Linux itself. However, the Xilinx Linux kernel isn't the only place where a driver can be looked for, as it might be available in other branches that did not make it to the Git Tag that PetaLinux is using for the Linux Kernel.

This was confirmed when looking in Enclustra Github repository, specifically in an I2C application note, which includes instructions for controlling several I2C devices that Enclustra usually includes in their devices. Here, a Si5338 reference was found, so, the driver existed, it was coded at some point of time and then removed for unknown reasons as far as I know from the mainline kernel before 5.15_LTS was released.

The files needed to build the drivers were found in a linux kernel commit in their own repositories that Enclustra uses for their own Development Environment tool.

The driver is formed by the following files:

- *clk-si5338.c*: the source file of the driver containing all the functions to probe the chip, configure it, enable the corresponding clock outputs and for debugging and reporting functionalities to the linux kernel.
- *clk-si5338.h*: the header file that accompanies every .c file when coding in C language. It has the same name as the .c file and contains multiple define that associate easy-to-remember names for different configuration parameters of the driver.
- *si5338.h*: any driver in the linux kernel needs to contain a header file in the `platform_data` folder that contains the struct variables for the device tree configuration.
- *KConfig:* this file is located inside each folder that contains source code to enable different options in the linux kernel. This files in each directory are then used to generate the Configuration GUI shown to the user when typing `petalinux-config -c kernel`. So, some lines need to be added here to say to the configuration in the PetaLinux SDK that a new driver must be configured.
- *Makefile:* if we are talking about a compilation in C language, the Makefile is always present as the list of source code that must be compiled and which output it produces. To compile the driver, the .c file needs to be added to the MakeFile present in the `drivers/clk` folder.

Once the driver was found, as it wasn't pulled from the Github branch by the SDK automatically, it needs to be added as a patch. Patches have already been discussed to fix some bugs present in other drivers, but they can also be used to add complete files to the kernel source code, the git format-patch command doesn't make any distinctions, if it needs to create the file because it doesn't exist, it will do so. So, following the git flow [1], the driver for Si5338 was added:

1. Download the drivers from the 2020 linux branch in Enclustra repositories.
2. Copy the new files and add the corresponding lines in the KConfig and MakeFile.
3. Do `git add -A` and `git commit --signoff`
4. A text editor will open for the user to type the description of the patch.
5. Do `git format-patch < branch base >`. A file with the patch will be generated.
6. Copy the file into the PetaLinux project in the following directory: *project-spec/meta-user/recipes-kernel/linux*.
7. Add the corresponding lines to the linux.bbappend file to include the patch into the compilation flow when performing petalinux-build'
8. Add the corresponding device tree node to `system-user.dtsi`, which calls this driver and configures the clock outputs. An example of this device tree node is present in Annex 2.

The configuration parameters in this driver are very detailed because this chip offers a lot possibilities. The generator has 6 clock inputs, where In1/In2 and In5/In6 behave as differential clock inputs and IN3 and IN6 are single ended. The figure below shows an ASCII drawing of how the different clock inputs arrive to each of the outputs.

```
   IN1/IN2   IN3         IN4 IN5/IN6
        |     |           |     |
  ------|     |           |     |
  |     |     |           |     |
  |     \     /           \     /
  |      \   /             \   /
  |       \ /               \ /
XTAL     REFCLK            FBCLK
  |       |  \             /   |
  |       |   \           /    |
  |       | DIVREFCLK DIVFBCLK |
  |       |     \       /      |
  |       |      \     /       |
  |       |       \   /        |
  |       |        PLL         |
  |       |       / | | \      |
  |       |      /  / \  \     |
  |       |     /  /   \  \    |
  |       |    /   |   |   \   |
  |       |    |   |   |   |   |
  |       |   MS0 MS1 MS2 MS3  |
  |       |    |   |   |   |   |
              O0  O1  O2  O3
```

Then, this clock generates internal clocks that lead to a PLL where a frequency is generated and afterwards, using MultiSynth, a proprietary frequency generator in each of the four differential outputs of the chip, a different frequency can be generated.

To configure all of this, there are a lot of options to adjust:

- **General clock generator options**: They are used to configure the chip itself:
  - #clock-cells: indicates whether the generator has one or several outputs. It must be set to 1 when the generator has multiple outputs which is the case being studied.
  - #address-cells: must be set to 1.
  - #size-cells: must be set to 0.
  - compatible: indicates the model of the chip being used. This is used to assign the correct driver to the device when petalinux probes it. In this case, this value must be set to `silabs,si5338`.
  - reg: i2c device address. It depends on the address assigned to it in the factory. For ST1 base board, the address is 0x70.
  - clocks: list of parent clocks in the order of < xtal > , < in1/2 >, < in3 >, < in4 >, < in5/6 >. In this vector, a node representing a clock must be given. For the ST1 a 100MHz single ended clock is generated and routed through IN3. Note, xtal and in1/2 are mutually exclusive. Only one can be set.
  - clock-names: just the name of the clocks described previously in that same order.

- **clkoutx nodes**, where each differential clock output is configured separately:
  - reg: the number of output, for *clkoutx* node where x ranges from 0 to 3, the reg must have the value x.
  - silabs, drive-config: the voltage of the generated clock. Here both the magnitude and the standard can be set. All the options are listed below together with their corresponding configuration codes:
  - clock-source: it refers to where the clock is coming from:

    ```
    #define SI5338_OUT_MUX_FBCLK        0
    #define SI5338_OUT_MUX_REFCLK       1
    #define SI5338_OUT_MUX_DIVFBCLK     2
    #define SI5338_OUT_MUX_DIVREFCLK    3
    #define SI5338_OUT_MUX_XOCLK        4
    #define SI5338_OUT_MUX_MS0          5
    #define SI5338_OUT_MUX_MSN          6 /* MS0/1/2/3 */
    #define SI5338_OUT_MUX_NOCLK        7
    ```

    It usually refers to the MultiSynth chip connected at each output so the usual configuration would be SI5338_OUT_MUX_MSN where N is the number of multiplexer as shown in the ASCII drawing.
  - disable-state: this chip is able to generate a clock only when requested to do so, so the driver must have a setting to indicate which logical state is considered to be disabled. Normally, it would be SI5338_OUT_DIS_HIZ* (High Impedance), but it will be seen later that it needed to be changed to other setting.

    ```
    #define SI5338_OUT_DIS_HIZ 0
    #define SI5338_OUT_DIS_LOW 1
    #define SI5338_OUT_DIS_HI 2
    #define SI5338_OUT_DIS_ALWAYS_ON 3
    ```

  - clock-frequency: as in any clock generator, the frequency in HZ is indicated in this node. For this case, *clkout0* was configured to have a 5MHz output for testing that an output was actually being produced and *clkout1*, the clock output that goes to MGT Bank D 156.25MHz.

4.1 Debugging the generated clock frequency

After adding the driver and compiling Linux, the system did not work. Using the Integrated Logic Analyzer in the *clkout0* testing output, there was no clock generated. Why? After looking in the documentation, it seems that there is flag called *enabled* in each of the *clkout* that is needed to say to the driver that we want to enable that output. After doing this to *clkout0* and *clkout1* the clock was generated! as shown below:

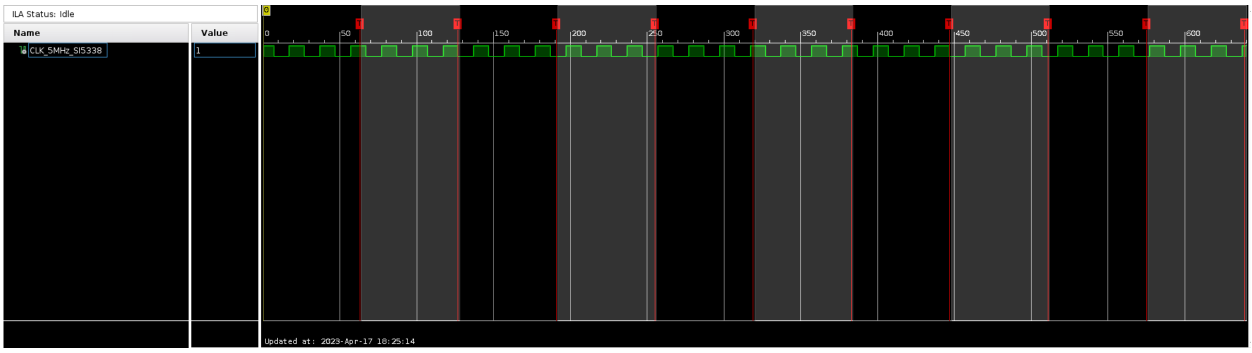


Figure 3.4.2: clkout0 signal being generated as a 5MHz clock

The frequency of the clock being shown is very easy to deduce. The Sample depth used for sampling the signal was 1024, the clock input for the ILA core was 100 MHz and the signal (the generated clock in the SI5338) was 5MHz. So, the samples taken per Si5338 clock cycle is $\frac{100}{5} = 20$. If 1024 is divided by 20, it gives a result of 51.2, meaning that in that time, 51 cycles of the SI5338 clock should be sampled. Counting the number of cycles it was checked that there were 51 cycles so the clock is 5 MHz, the correct frequency set up in the driver.

4.2 Testing the Ethernet link

Returning to our Ethernet Core, even though the clock was being generated, the SFP link wouldn't work. This has a simple explanation that could be deduced by outputting the signal *reset_done* to a LED in the ST1 base board. Thanks to this it was discovered that the reset was not performed when turning on the board which was pretty normal before because a clock was not being generated but, what about now? The clock was not available until the Linux system had booted, meaning that the reset was performed before the clock got generated so, nothing works. The solution found was rebooting the system without turning off the board but using the `reboot` command or the Power On Reset button. This does not erase the clock generator data but reboots the system so that it must initialize everything again. Asserting the reset again this time with a valid clock input yielded the expected results: macb detected a new link and configured it properly to 1Gbps/Full duplex as it can be seen in the following terminal messages:

UBOOT CONSOLE

```
ZYNQ GEM: ff0c0000, mdio bus ff0c0000, phyaddr 9, interface gmii

Warning: ethernet@ff0c0000 (eth4) using random MAC address - ca:32:17:cf:08:00

ZYNQ GEM: ff0d0000, mdio bus ff0d0000, phyaddr 11, interface gmii
```

PETALINUX CONSOLE

```
[   12.822833] macb ff0b0000.ethernet eth2: Link is Up - 1Gbps/Full - flow control off
[   12.830520] IPv6: ADDRCONF(NETDEV_CHANGE): eth2: link becomes ready
```

Chapter 5

DAQ Network Interface card analysis. The Intel XL710

Now that one side of the communication has been solved, the other side, that is the PC with a PCIe network card must be correctly configured. For testing the SOM with several SFP modules, the best way was to use a Network Expansion card that could allocate more than one SFP. For the initial stages of the project, while working with the ZCU102, a PEX1000SFP from Startech was used. This card included one SFP Gigabit port, making it ideal for testing one fiber optics link. As the network controller chip, it had a RTL8168B from Realtek that made possible the SFP connection to the computer.

Figure 3.5.1: Startech PEX1000SFP with the RTL8168B chip

However, now that more than one SFP port is necessary, an upgrade in the card is a must. Startech also offers another network card called the PEX10GSFP4I, that offers 4 SFP+ ports, meaning a total bandwidth of 40 Gbps. This bandwidth was supported thanks to the inclusion of the Intel XL710. As this chip is different from the one used in the previous NIC, the behaviour may vary and that was exactly the case.

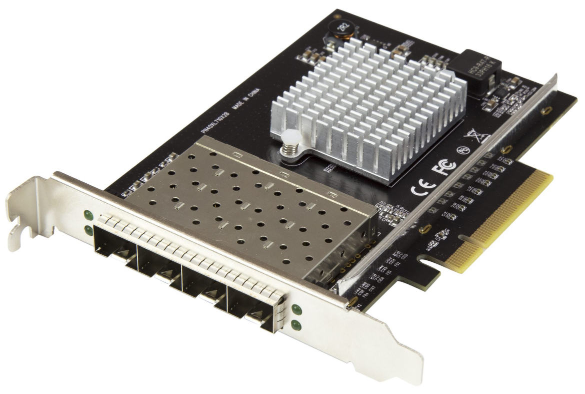
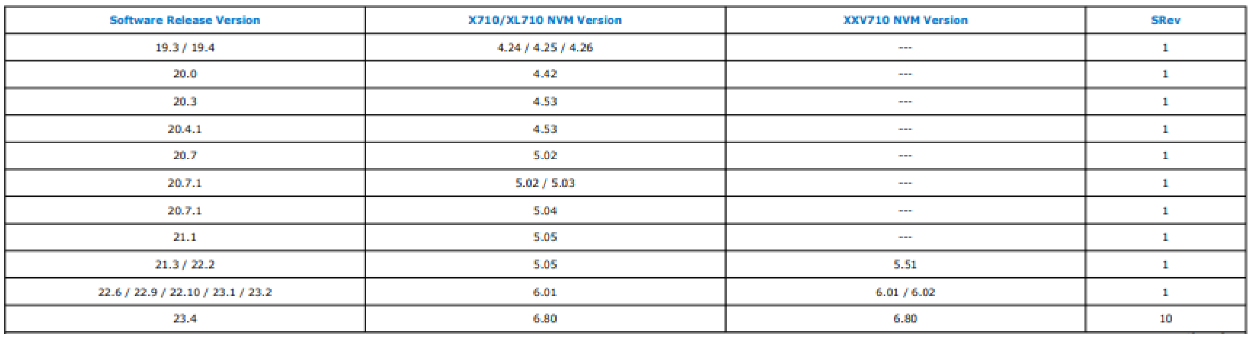


Figure 3.5.2: Startech PEX10GSFP4I with Intel XL710 chip

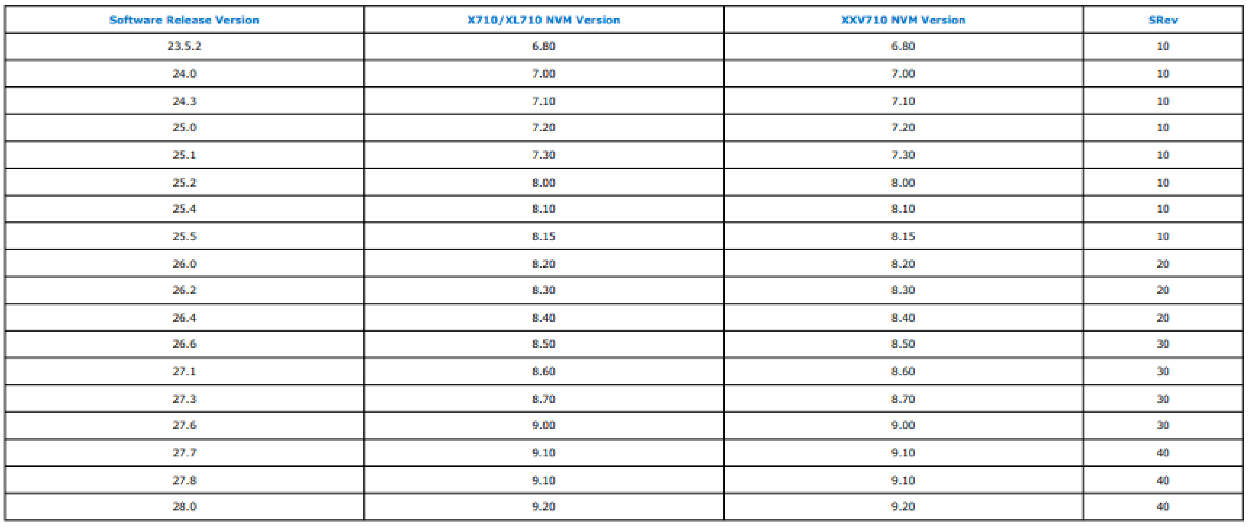
After ensuring that the SOM side of the communication worked connecting it to the Realtek NIC (known to be already working with other links), the Intel NIC was tested. At first there was no link established with the PC side not even detecting a SFP module in the corresponding cage.

Looking through the internet, a solution was found: it seems that this network cards are so complex that they have a firmware running on them that is updated to fix bugs and add support for other standards. This firmware is stored in a Non Volatile Memory (NVM) so that it can be easily accessed and updated through the use of the Intel NVM Update Utility. The reason for the communication not being established was due to the NIC having a very old firmware version that didn't have support for 1000BaseLX/SX SFP. This was checked by looking at the Intel XL710 Feature Support Matrix[21]. In this document there are two tables that helped to reach this conclusion: Each NVM version, that is, each firmware version, requires a minimum driver version as shown in both figures 3.5.3 and 3.5.4.



Software Release Version	X710/XL710 NVM Version	XXV710 NVM Version	SRev
19.3 / 19.4	4.24 / 4.25 / 4.26	---	1
20.0	4.42	---	1
20.3	4.53	---	1
20.4.1	4.53	---	1
20.7	5.02	---	1
20.7.1	5.02 / 5.03	---	1
20.7.1	5.04	---	1
21.1	5.05	---	1
21.3 / 22.2	5.05	5.51	1
22.6 / 22.9 / 22.10 / 23.1 / 23.2	6.01	6.01 / 6.02	1
23.4	6.80	6.80	10

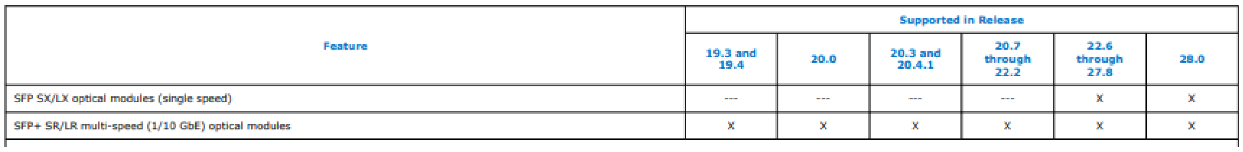
Figure 3.5.3: NVM version compatibility with the drivers of Intel XL710 (i40e)



Software Release Version	X710/XL710 NVM Version	XXV710 NVM Version	SRev
23.5.2	6.80	6.80	10
24.0	7.00	7.00	10
24.3	7.10	7.10	10
25.0	7.20	7.20	10
25.1	7.30	7.30	10
25.2	8.00	8.00	10
25.4	8.10	8.10	10
25.5	8.15	8.15	10
26.0	8.20	8.20	20
26.2	8.30	8.30	20
26.4	8.40	8.40	20
26.6	8.50	8.50	30
27.1	8.60	8.60	30
27.3	8.70	8.70	30
27.6	9.00	9.00	30
27.7	9.10	9.10	40
27.8	9.10	9.10	40
28.0	9.20	9.20	40

Figure 3.5.4: NVM version compatibility with the drivers of Intel XL710 (i40e) Contd.

Using the `ethtool -i enp2s0f3`, it was checked that the version of NVM installed was 5.04, meaning that the card required at a minimum the 20.7.1 driver. The driver version wasn't an issue as it was already updated to the latest version 28.0 even though the card wasn't able to operate at its full potential with all of the capabilities of this driver due to the firmware being obsolete. It was checked that in NVM 5.04 the 1000BaseLX/SX wasn't supported by inspecting another table shown in figure 3.5.5.



Feature	Supported in Release					
	19.3 and 19.4	20.0	20.3 and 20.4.1	20.7 through 22.2	22.6 through 27.8	28.0
SFP SX/LX optical modules (single speed)	---	---	---	---	X	X
SFP+ SR/LR multi-speed (1/10 GbE) optical modules	X	X	X	X	X	X

Figure 3.5.5: Standard compatibility with each firmware version in the Intel XL710 (i40e)

The relevant entry here for us is the SFP SX/LX optical modules (single speed), which is the case of the Avago and Cisco modules being used here (they only support a single speed, 1Gbps). This feature was not supported until 22.6 driver meaning that the firmware update that started to support these modules was, looking at the previous figures, 6.01, so an update is necessary.

This process was very simple, as Intel has a NVM Update Utility that just requires to be unzipped and run the script to update it. The script used is called `nvmupdate64e` and is used as follows:

```
nvmupdate64e -u -l -o results.xml -b -c nvmupdate.cfg
```

This command checks the file nvmupdate.cfg which contains a list of compatible devices and their current firmware versions together with to which version they can be updated to. If there is a match with a device connected to the PC then it will perform an update and save the results into an xml.

After performing this step, the *results.xml* file was generated. An extract of this file is shown below:

```
<Instance vendor="8086" device="1572" subdevice="0000" subvendor="8086" bus="2"
          dev="0" func="0" PBA="003400-000" port_id="Port 1 of 4"
          display="Intel(R) Ethernet Converged Network Adapter X710">
    <Module type="NVM" version="8000D8BC" previous_version="800024DA">
        <Status result="Success" id="0">All operations completed successfully.</Status>
    </Module>
    <VPD>
        <VPDField type="String">Example VPD</VPDField>
        <VPDField type="Readable" key="V0"></VPDField>
        <VPDField type="Checksum" key="RV">D7</VPDField>
    </VPD>
    <MACAddresses>
    <MAC address="E8EA6A27CE51">
    </MAC>
    <SAN address="000000000200">
    </SAN>
    </MACAddresses>
</Instance>
```

In this XML file there is a label called module with several attributes. This Module represents the NVM memory and is indicating that the firmware has been updated from previous_version to version, so from 800024DA corresponding to 5.04 to 8000D8BC corresponding to 9.20. Now by running the same `ethtool -i enp2s0f3` command the output showed the new version of the NVM:

```
hyperk1@hyperk1:/$ ethtool -i enp2s0f3
driver: i40e
version: 2.22.18
firmware-version: 9.20 0x8000d8bc 0.0.0
expansion-rom-version:
bus-info: 0000:02:00.3
supports-statistics: yes
supports-test: yes
supports-eeprom-access: yes
supports-register-dump: yes
supports-priv-flags: yes
```

After that update, the SFP was recognised and the link was established so the FMC card in the SOM was checked to be working and the NIC card which is a crucial part of the testbench that will be used for the DPB2 prototype has also been tested. Also, the speed tests were successful. By using `iperf3`, the performance was measured, giving very good results, close to 1Gbps. Using Jumbo frames this throughput could be increased until topping the whole 1Gbps bandwidth.

```
root@ST1ME-XU8-4CG-1E-D11E:~# iperf3 -c 10.0.0.1 -t 25
Connecting to host 10.0.0.1, port 5201
[  5] local 10.0.0.2 port 54852 connected to 10.0.0.1 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   114 MBytes   953 Mbits/sec    0    267 KBytes
[  5]   1.00-2.00   sec   112 MBytes   938 Mbits/sec    0    267 KBytes
[  5]   2.00-3.00   sec   113 MBytes   947 Mbits/sec    0    279 KBytes
[  5]   3.00-4.00   sec   112 MBytes   939 Mbits/sec    0    279 KBytes
[  5]   4.00-5.00   sec   112 MBytes   944 Mbits/sec    0    279 KBytes
[  5]   5.00-6.00   sec   112 MBytes   938 Mbits/sec    0    291 KBytes
[  5]   6.00-7.00   sec   112 MBytes   938 Mbits/sec    0    291 KBytes
[  5]   7.00-8.00   sec   112 MBytes   944 Mbits/sec    0    291 KBytes
[  5]   8.00-9.00   sec   112 MBytes   944 Mbits/sec    0    291 KBytes
[  5]   9.00-10.00  sec   112 MBytes   938 Mbits/sec    0    291 KBytes
[  5]  10.00-11.00  sec   112 MBytes   944 Mbits/sec    0    291 KBytes
[  5]  11.00-12.00  sec   112 MBytes   943 Mbits/sec    0    291 KBytes
[  5]  12.00-13.00  sec   112 MBytes   938 Mbits/sec    0    291 KBytes
[  5]  13.00-14.00  sec   112 MBytes   944 Mbits/sec    0    291 KBytes
[  5]  14.00-15.00  sec   112 MBytes   938 Mbits/sec    0    291 KBytes
[  5]  15.00-16.00  sec   113 MBytes   944 Mbits/sec    0    291 KBytes
[  5]  16.00-17.00  sec   112 MBytes   938 Mbits/sec    0    291 KBytes
[  5]  17.00-18.00  sec   112 MBytes   944 Mbits/sec    0    291 KBytes
[  5]  18.00-19.00  sec   112 MBytes   944 Mbits/sec    0    291 KBytes
[  5]  19.00-20.00  sec   112 MBytes   938 Mbits/sec    0    291 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-20.00  sec  2.24 GBytes   942 Mbits/sec    0             sender
[  5]   0.00-20.05  sec  2.24 GBytes   940 Mbits/sec                  receiver

iperf Done.
```

Chapter 6

Redundant data link creation

6.1 Adding an extra Ethernet Core

The DPB2 prototype will not have just one SFP for data link, it will have at least 2, for redundancy. In this section, the procedure to instantiate another network interface with another SFP of the same features as in the previous section will be described.

The steps, departing from the same reference project are as follows: first, Open Vivado and instantiate two Gigabit Ethernet PCS/PMA cores, then, Configure one of them with Shared Logic and the same parameters as in Part III, chapter 3. The other one must be configured with the same parameters as the first one but instead of including the Shared Logic in the core, select Shared Logic in the Example Design. This will create an IP Core without the shared logic. This shared logic is a part of the IP Core required for it to work but, as it is logic that is common to all the IP Cores of the same type, it is enough to instantiate this shared logic in one of the cores and then connect the rest of the cores to this core with the shared logic. If the user changes the Shared Logic option, the core preview will change a lot as shown in figure 3.6.1.

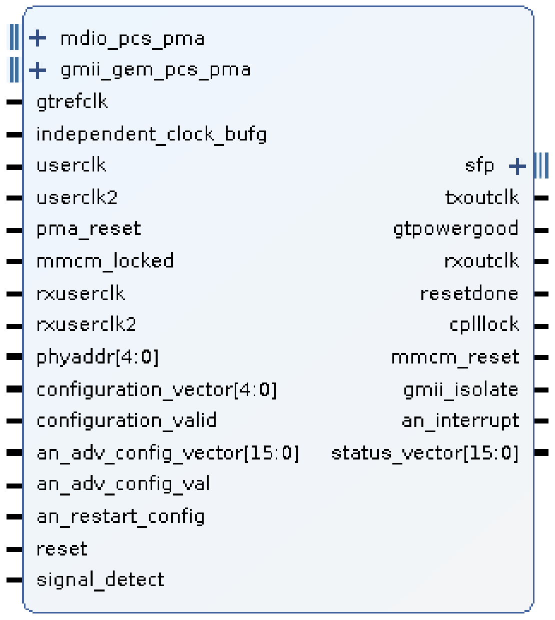


Figure 3.6.1: 1G/2.5G Ethernet PCS/PMA Core without Shared Logic

If comparing both blocks, one can instantly notice that the non-shared logic core has a lot of inputs that correspond to the outputs that were left unconnected in the shared logic core. All this outputs must be connected by name to its corresponding inputs. So, connect the corresponding inputs to the corresponding outputs of the Core with Shared Logic as stated in table 6.2.

From Instance Using Shared Logic in Core	To Instance Using Shared Logic in Example Design	Description of the functionality
gtrefclk_out	gtrefclk	Transceiver clock. It is converted inside the Shared logic core from a differential pair to a single ended clock and that connection is passed to Non Shared Logic Core
userclk_out	userclk	62.5 MHz clock at 1Gb/s
userclk2_out	userclk2	125 MHz clock at 1Gb/s
rxuserclk_out	rxuserclk	62.5 MHz clock at 1Gb/s
rxuserclk2_out	rxuserclk2	125 MHz clock at 1Gb/s
pma_reset_out	pma_reset	Physical Medium Attachement reset
mmcm_locked_out	mmcm_locked	Flag to indicate that the mmcm (a clock generator block) is producing a stable and reliable clock

Table 6.2: Port list in the Ethernet core used

1. Create another constants block with the same aspect as in the first core but changing the phy address to other value that no other phy chip has. In this case the ST1 base board has two PHY chips for the RJ45 ports that have addresses 3 and 7 assigned. Also, the Ethernet Core with Shared Logic has phy address 9. So, it was decided to use address 11 for this device.

2. Open the ZynqMP IP configuration and enable one GEM together with its MDIO interface and set them up as EMIO.

3. Create the corresponding inputs and outputs port of this new block that will go outside the FPGA. In this case it would be only the SFP interface because the other ones are connected to other blocks in the FPGA.

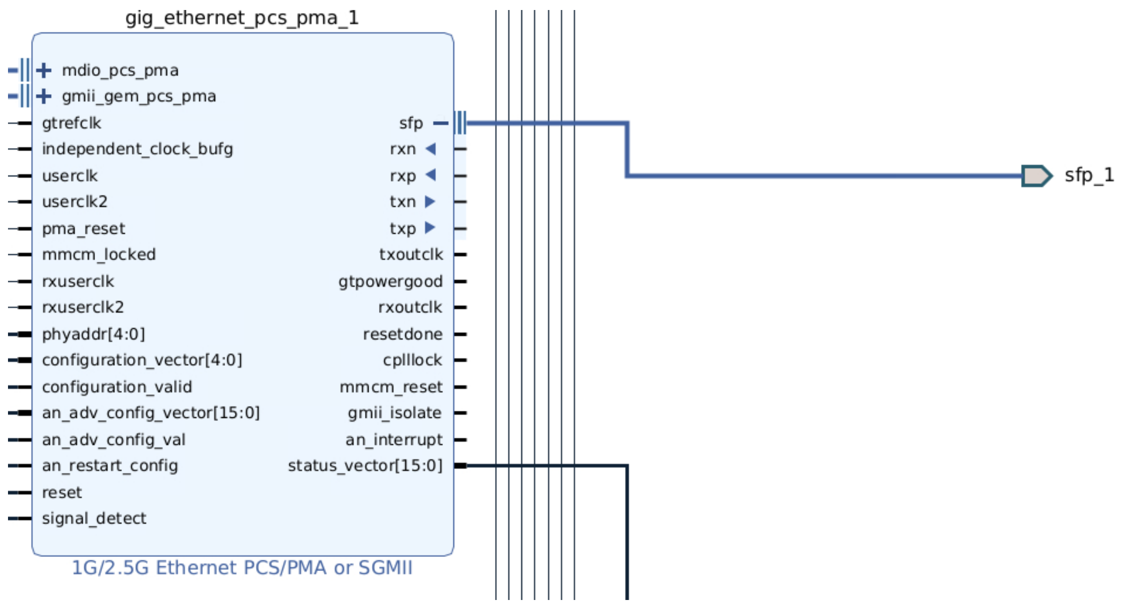


Figure 3.6.2: SFP output interface, by right click Create Interface Port

4. Do the Synthesis, Implementation and Bitstream Generation in Vivado

5. Export the hardware to a .xsa including the Bitstream.

6. Enable the driver through the configuration command:

```
hyperk1@hyperk1:/$ petalinux-config -c kernel
```

7. Add to the device tree the new node for the GEM.

```
&gem1 {                                         &gem2 {
    status = "okay";                                status = "okay";
    phy-handle = <&phy9>;                           phy-handle = <&phyb>;
    phy9: phy@9 {                                   phyb: phy@b {
      reg = <0x9>;                                  reg = <0xb>;
      xlnx,phy-type = <0x5>;                        xlnx,phy-type = <0x5>;
      reset-gpios = <&gpio 78 0>;                   reset-gpios = <&gpio 79 0>;
    };                                              };
};                                              };
```

8. Build the image and package it:

```
hyperk1@hyperk1:/$ petalinux-build
```

```
hyperk1@hyperk1:/$ petalinux-package --boot --force  --fsbl --fpga --u-boot
```

9. Test in the board. As it can be seen from the logs both from Uboot (Second Stage Boot loader) and PetaLinux, GEM1 and GEM2 get detected and correctly configured.

U-BOOT CONSOLE:

```
ZYNQ GEM: ff0c0000, mdio bus ff0c0000, phyaddr 9, interface gmii
Warning: ethernet@ff0c0000 (eth4) using random MAC address - ca:32:17:cf:08:00
, eth4: ethernet@ff0c0000FEC: can't find phy-handle

ZYNQ GEM: ff0d0000, mdio bus ff0d0000, phyaddr 11, interface gmii
, eth2: ethernet@ff0d0000FEC: can't find phy-handle
```

PETALINUX CONSOLE:

```
[6.334967] macb ff0d0000.ethernet eth2: Cadence GEM rev 0x50070106 at 0xff0d0000 irq 39
            (00:0a:35:00:09:89)

[6.233972] macb ff0c0000.ethernet eth1: Cadence GEM rev 0x50070106 at 0xff0c0000 irq 38
            (96:ee:35:90:7a:03)


[12.822833] macb ff0b0000.ethernet eth2: Link is Up - 1Gbps/Full - flow control off
[12.830520] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
```

6.2 The MDIO bus

Management Data Input Output (MDIO) is a standardized interface for accessing the configuration and status registers of Ethernet PHY devices in IEEE 802.3. It is commonly used to configure PHYsical layer chips like USB, PCIe or Ethernet. One MDIO bus can have one master and up to 32 slaves. It works as a tristate bus. An example of this system is shown in figure 3.6.3.

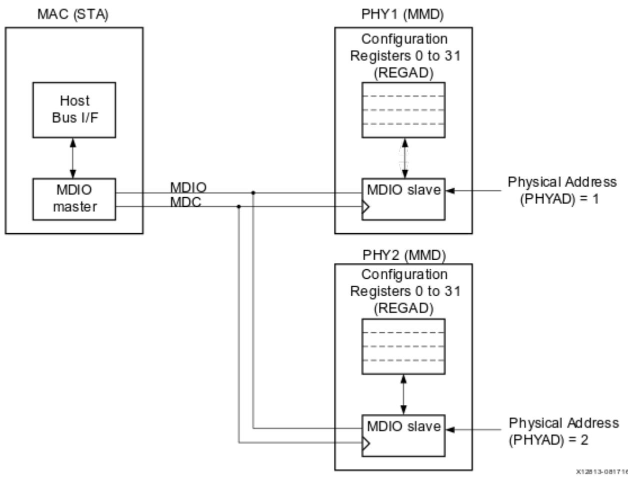

MAC (STA)
Host Bus I/F
MDIO master
MDIO
MDC
PHY1 (MMD)
Configuration Registers 0 to 31 (REGAD)
MDIO slave
Physical Address (PHYAD) = 1
PHY2 (MMD)
Configuration Registers 0 to 31 (REGAD)
MDIO slave
Physical Address (PHYAD) = 2
X12813-081716

Figure 3.6.3: A typical MDIO-managed System

1. One of the cores needs to be adapted because MDIO is a tristate bus, meaning that it has a third input labelled as *mdio_t* which puts the bus in High impedance (Z). Sharing this bus would require to instantiate IOBUFs blocks. However, these buffers can only be used when driving I/O signals that are connected directly to I/O ports not to internal connections as modern FPGAs don't have internal tristates. These cores have a special option called *MDIO for external PHY management* that produces an output to the core of type MDIO master. This output is normally used to output the MDIO bus to the outside of the FPGA to drive a PHY ASIC chip but it can also be used for connection with internal cores.

2. Once the option has been selected, two new ports will appear: the MDIO out interface that will be connected to the MDIO port of the other core, and the mdio_t signal which is connected directly to the corresponding mdio_t signal of the MDIO master in the GEM (ZynqMP IP Core).

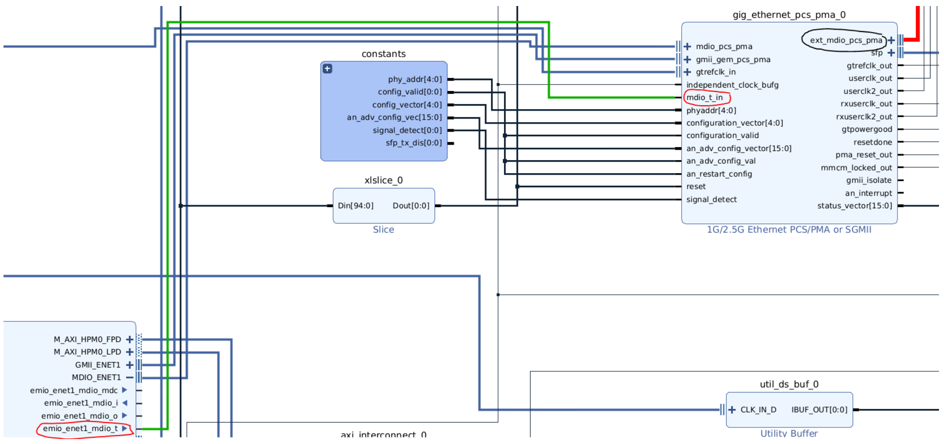


Figure 3.6.4: Connection of the mdio_t signal to the mdio_t input (green wire) and connection of the MDIO output (red wire) of the core enabled through Enable MDIO for External PHY Management option

3. Do the whole Vivado flow and export the .xsa

4. In the device tree, now both phy nodes should be inside the same gem node as they are now connected both to the same MDIO bus. In this case, both cores where connected to GEM1 MDIO, so the phy nodes shall be written under &gem1 node.

```
&gem1 {
    status = "okay";
    phy-handle = <&phy9>;
    phy9: phy@9 {
      reg = <0x9>;
      xlnx,phy-type = <0x5>;
      reset-gpios = <&gpio 78 0>;
    };
    phyb: phy@b {
      reg = <0xb>;
      xlnx,phy-type = <0x5>;
      reset-gpios = <&gpio 79 0>;
    };
};

&gem2 {
    status = "okay";
    phy-handle = <&phyb>;
};
```

5. Build and package the Linux image.

```
hyperk1@hyperk1:/$ petalinux-build
```

```
hyperk1@hyperk1:/$ petalinux-package --boot --force  --fsbl --fpga --u-boot
```

6. Test that MDIO bus are probed.

For this approach to work, the phy cores must be connected to an MDIO bus such that they are probed before turning on the GEM corresponding to that PHY core. Probing means registering the MDIO device in the MDIO bus. This is a crucial step before turning on the Ethernet interface.

The order in which the Macb probes the phychip is by the gem number, so: first gem0, then gem1, gem2 and finally gem3. In this case, as it can be seen in the device tree, both phy nodes are in GEM1: the phy corresponding to GEM2 is phyb and as it is probed in the GEM1 node it will be probed when GEM2 interface is turned on afterwards. Then, in each MDIO bus, devices are recognised through its Phy address. Phy addresses can also be changed. They can go from 0x01 to 0x1f. 0x00 is excluded because that address acts as a broadcast for all the devices in the bus. The change of phy address of a specific ethernet core must be done in two places and it is crucial that the same number is introduced in each:

- **In the core itself:** by using the phy_addr input. A constant must be connected there indicating the address value
- **The reg variable in the device tree**: each phy node has a property called reg which is a number representing the phy address of that phy node.

6.3 Bonding driver in PetaLinux

Once the SFP interfaces are connected and usable from Linux, a controlled way to decide which interface will be active at a given time and how to perform the switching when that interface fails must be developed.

Linux offers a very easy way to do it where the OS handles the switching called bonding which consists on creating a virtual interface called bond-master that aggregates several physical interfaces as their bond-slaves. This bond-master interface is given an IP address and communicates with the outside world as a normal interface. The packages that are delivered to that address are received through the physical interface/s depending on the configuration of the bonding. So, the other devices are just given one destination address in their routing tables, which is the bond-master one but they have configured that this interface is a bonding interface so there are more than one physical interface through which they can send the data to arrive to its destination.

There are several configuration modes in bonding, created depending on the advantage that benefits your application:

- **Round Robin policy:** Packets are transmitted in sequential order from the first slave to the last. Provides both load balancing and fault tolerance because any interface that is down is eliminated from the round robin.
- **Active Backup:** Only one slave active. A different slave becomes active if the currently activated fails. Only the virtual bond MAC address is visible from the outside to avoid confusing the switch.
- **Balance XOR policy:** Transmit to a certain slave based on a selectable hashing algorithm. Provides load balancing and fault tolerance
- **Broadcast:** transmits everything on all slaves at the same time. Provides fault tolerance without losing packets for switching from one interface to another but also uses a lot of energy.
- **IEEE 802.3ad:** It is an IEEE standard for dynamic link aggregation, with slaves that share the same speed and duplex settings. If this mode is the preferred one, the switches connected to the device must support the standard.
- **Adaptive transmit load balancing:** The outgoing traffic is distributed depending on the current load. The incoming traffic is received by the active slave at that time.
- **Adaptive load balancing:** It distributes both the outgoing and ingoing traffic depending on the current load. The receive load balancing is achieved by ARP negotiation. The bonding driver intercepts the ARP Replies sent by the local system on their way out and overwrites the source hardware address with the unique hardware address of one of the slaves in the bond such that different peers use different hardware addresses for the server.

The ideal mode to use in this project would be the Active Backup one as fault tolerance is needed but not all the SFPs will be turned on at the same time so, when the slave is switched by the bonding driver, the new SFP should be turned on.

The bonding driver can be interacted with and configured in two different ways. These two methods are explained below and they differ in which files must be modified to enable and configure bonding.

- **Using *ifenslave*:** A software package that allows to create bonding interfaces by configuring them in the */etc/network/interfaces*. This file is a standard Linux file that includes the configuration of the network interfaces. This file is read when the command `ifup` is executed and applies the configuration that to each interface when they are brought up (or down with `ifdown`.

  To include the interfaces file in PetaLinux, the workflow is just the typical Yocto recipe creation. Refer to [1] for more details about how Yocto works.

  Below, there is an example of the interfaces file. This file must be placed in */etc/network/* directory. As we can see we configure 4 interfaces: the loopback, the bond itself and the two physical interfaces that will join the bonding. These interfaces are linked through the *bond-slaves* and *bond-master* property which define which is the virtual bonding interface. The *bond-mode* is configured as active-backup. It is very important to note that neither *eth1* nor *eth2* have IP addresses assigned because the visible interface from the outside will be the *bond0* interface that will make use of *eth1* and *eth2* the hood.

```
auto lo
iface lo inet loopback

 auto bond0
 iface bond0 inet static
    address 20.0.0.2
    netmask 255.255.255.0
    gateway 20.0.0.1
    bond-mode active-backup
    bond-miimon 100
    bond-slaves eth1 eth2
    bond-primary eth1 eth2

 auto eth1
 iface eth1 inet manual
    bond-master bond0

 auto eth2
 iface eth2 inet manual
    bond-master bond0
```

  When booting the us. We perform the `ifup -a` command that enables all the interfaces present in the *etc/network/interfaces* file.

  Now, when disconnecting any of the interface the following message appears:

```
[46.997] macb ff0c0000.ethernet eth1: Link is Down
[47.021] bond0: (slave eth1): link status definitely down, disabling slave
[47.028] bond0: (slave eth2): making interface the new active one
```

  This message indicates that the system works correctly, as the eth1 was the primary slave (the first position in the bond-primary list) and when it was brought down by disconnecting the optical link, the eth2 interface was activated, so that the connection is not lost.

  The *interfaces* file should be edited in both ends of the communication if a direct connection to another PC is done as the PC that will receive the data also must have the information that two of its physical interfaces will be grouped in a bonding interface. If the other side is a switch, no extra steps are needed.

- **Using *networkd-service*** Another way is to use one of the systemD services included in the PetaLinux image. This is just another way to configure the interfaces with some initial parameters. In this case, the files that need to be created correspond to the networkd service. The steps to configure bonding through this method are as follows:

  1. Ensure that PetaLinux has the networkd service running in the background:

```
root@ST1ME-XU8-7EV-2I-D12E:~$ systemctl status systemd-networkd
* systemd-networkd.service - Network Configuration
     Loaded: loaded (8;;file://ST1ME-XU8-7EV-2I-D12E/lib/systemd/
     system/systemd-networkd.service/lib/systemd/system/
     systemd-networkd.service8;;; enabled; vendor preset: enabled)
     Active: active (running) since Fri 2021-11-19 17:19:30 UTC;
TriggeredBy: * systemd-networkd.socket
       Docs: 8;;man:systemd-networkd.service(8)man:
       systemd-networkd.service(8)8;;
   Main PID: 339 (systemd-network)
     Status: "Processing requests..."
      Tasks: 1 (limit: 4377)
     Memory: 972.0K
     CGroup: /system.slice/systemd-networkd.service
             `-339 /lib/systemd/systemd-networkd
```

  2. Create a new application in PetaLinux:

```
hyperk1@hyperk1:/$ petalinux-create -t apps -n networkd-bond
```

  3. Go to the folder created by the SDK:

```
hyperk1@hyperk1:/$ cd project-spec/meta-user/recipes-apps/networkd-bond
```

  4. Paste inside the .bb file the following recipe:

```
SUMMARY = "Networkd bonding system"
SECTION = "PETALINUX/apps"
LICENSE = "MIT"

SRC_URI = "file://myapp-init.service file://10-nd-bond.network \
       file://20-eth1.network file://20-nd-bond.netdev \
       file://30-eth2.network file://99-dhcp.network"
do_install() {
      install -d ${D}/etc/systemd/network/
      install -m 0644 ${WORKDIR}/myapp-init.service ${D}${systemd_system_unitdir}
      install -m 0644 ${WORKDIR}/10-nd-bond.network ${D}/etc/systemd/network/
      install -m 0644 ${WORKDIR}/20-eth1.network ${D}/etc/systemd/network/
      install -m 0644 ${WORKDIR}/20-nd-bond.netdev ${D}/etc/systemd/network/
      install -m 0644 ${WORKDIR}/30-eth2.network ${D}/etc/systemd/network/
      install -m 0644 ${WORKDIR}/99-dhcp.network ${D}/etc/systemd/network/
}
```

5. The recipe described in the previous step copies several files to the */etc/systemd/network* which are read by the networkd service to configure the different interfaces. The files to be created must be included in the *files* folder inside the `project-spec/meta-user/recipes-apps/networkd-bond`:
   - 10-nd-bond.network: creates the bonding virtual interface called *nd-bond* and assigns the network parameters.

```
[Match]
Name=nd-bond
[Network]
Address=20.0.0.2/24
Gateway=20.0.0.1
```

   - 20-eth1.network: creates the eth1 interface and associates it as a primary slave to the bonding *nd-bond*

```
[Match]
Name=eth1
[Network]
Bond=nd-bond
PrimarySlave=true
```

   - 20-nd-bond.netdev: Configures the bonding interface as active-backup with a polling period of 1 second. *PrimaryReselectPolicy* option is set as always so that at any time, if the primary slave is available, it should be the active slave. This means that if eth1 is down, eth2 will be the active slave, but, in the instant that eth1 comes up again, the active slave will be changed to eth1 even though eth2 is fine.

```
[NetDev]
Name=nd-bond
Kind=bond
[Bond]
Mode=active-backup
PrimaryReselectPolicy=always
MIIMonitorSec=1s
```

   - 30-eth2.network: creates the eth2 interface and associates it with *nd-bond*

```
[Match]
Name=eth2
[Network]
Bond=nd-bond
```

   - 99-dhcp.network: For the rest of ethernet interfaces that are not specified in other files, enable DHCP. eth* is a wildcard that means "any interface that starts with eth".

```
[Match]
Name=eth*
[Network]
DHCP=yes
```

The number prefix in these files mean the execution order, so the first file to be read will be *10-nd-bond.network* and so on until the file *99-dhcp.network*. This order is crucial as the bonding interface needs to be created before assigning the physical interfaces to it.

6. Go to the *user-rootfsconfig* in `project-spec/meta-user/conf` to add the line CONFIG_networkd-bond.
7. Enable the corresponding user package in the rootfs menu:

```
hyperk1@hyperk1:/$ petalinux-config -c rootfs
```

8. Do `petalinux-build` and `petalinux-package` and load the image in the SOM.
9. When booting the board, the interfaces should be displayed when issuing `ifconfig`. If eth1 is disconnected, it should happen the same as with ifenslave and the same messages should be displayed. This time there is no need to issue any command because petalinux configures the interfaces by using networkd-service by default. This service does not use the aforementioned interfaces file, so a command was needed to read it.

Chapter 7

Debugging PetaLinux

Making some device work in Linux implies the development of both hardware up to the task and a compatible driver that is able to communicate with the device, configure it and send or receive the relevant information from it. For example, the SFP module is driven by the PCS/PMA or SGMII Gigabit Ethernet Core. This core acts as a PHY chip, configured through a MDIO interface and thus it needs to be initialised appropriately. These initialisation and management are usually done through a driver, a piece of code that provides several functions to the OS to leverage the capabilities of the hardware that the driver is developed for.

However, issues with how the driver interacts with the device can arise, provoking a malfunction. This was the case with the macb driver, made by Cadence and compatible with the GEM present in many ARM SoC like the ZynqMP Ultrascale+. At first there were some issues with configuring the second ethernet interface but the hardware seemed to be working perfectly as the clock was being generated and the reset signal was asserted.

In this section, three methods to debug the kernel code will be explained. The first method was the one used to find the bug and thus will be explained together with the solution to the bug. The other two are alternatives that I find very interesting to debug other failures.

7.1 Using kernel messages

The easiest way from a coding perspective is adding messages to be printed in the middle of the code execution flow. This is the least flexible method as it needs to recompile the kernel source code each time a new message wants to be added either to indicate that a part of the code has been executed or to display a new variable.

In Linux, the function used to display messages in the terminal is not *printf()*, it is *printk()*. All *printk* messages are printed to the kernel log buffer, which can be displayed by using the *dmesg* command.

printk has the main difference with *printf* that it can specify a log level which is specifies the importance of a message. The kernel decides whether to show the message immediately (printing it to the current console) depending on its priority level (Priority column in the below table) and the current *loglevel* (a kernel variable). If the message priority value in the below table is lower than the *loglevel* the message will be printed to the console. The *loglevel* variable is specified when booting in the *bootargs* string. This string can be modified by going to:

```
petalinux-config -> DTG settings -> Add extra bootargs
```

The different log levels that exist in the linux kernel are defined in the *include/linux/kern_levels.h* file in the source code and are described in the following table:

Name	Priority	Alias function
KERN_EMERG	"0"	**pr_emerg()**
KERN_ALERT	"1"	**pr_alert()**
KERN_CRIT	"2"	**pr_crit()**
KERN_ERR	"3"	**pr_error()**
KERN_WARNING	"4"	**pr_warn()**
KERN_NOTICE	"5"	**pr_notice()**
KERN_INFO	"6"	**pr_info()**
KERN_DEBUG	"7"	**pr_debug() and pr_devel() if DEBUG is defined**
KERN_DEFAULT	""	
KERN_CONT	"c"	**pr_cont()**

Alias functions are defined in the */include/linux/printk.h* and perform the same function as printk() but including the log level implicitly in the name of the function. For example, the function used for debugging in this case: *pr_warn()* will implicitly send a KERN_WARNING through the terminal. The alias is described as a macro as follows:

```
/**
 * pr_warn - Print a warning-level message
 * @fmt: format string
 * @...: arguments for the format string
 *
 * This macro expands to a printk with KERN_WARNING loglevel. It uses pr_fmt()
 * to generate the format string.
 */
#define pr_warn(fmt, ...) \
    printk(KERN_WARNING pr_fmt(fmt), ##__VA_ARGS__)
```

It is just a simple printk starting with a KERN_WARNING.

This function was included in several parts of the macb driver to check what was going on with the interface initialization on boot. The inclusion of these *pr_warn()* sentences was done by patching the linux kernel with the procedure explained in my other TFM.

When the system was booted and the turn to configure the second SFP came, the macb driver gave the following messages:

```
[6.248004] macb: Beginning probe!
[6.248031] macb: Success macb_config!
[6.251498] macb: Success clk_init!
[6.258751] macb ff0d0000.ethernet: Not enabling partial store and forward
[6.269881] macb: Success irq!
[6.269888] macb: Success get_mac_address!
[6.273349] macb: Success of register_netdev!
[6.277439] macb: Success macb_writel!
[6.281787] macb: It is NOT MDIO!
[6.285528] macb: It is not fixed_link!
[6.288832] macb: There is MDIO node!
[6.292857] libphy: Success device_register!
[6.296522] libphy: Success reset_GPIO 1!
[6.300786] libphy: Success reset_GPIO 2!
[6.304788] libphy: Success of MDIO probed!
[6.308789] of_MDIO_register Success!
[6.312964] of_MDIO_register: address: 11 !
[6.322909] of_mdiobus_register_phy 0
[6.332964] of_MDIO_register: address: 11 !
[6.342909] of_mdiobus_register_phy -16
[6.352909] of_mdiobus_register final error: -16
[6.352909] macb: probe of ff0d0000.ethernet failed with error -16
```

This means that the procedure of initialising the interface there was an error when executing the *mdiobus_register_phy* function. This function registers the phy chip that has the address given as an input to the function. Using *pr_warn* the address being passed to the function was printed and then the result of the function was printed.

In Linux, if a function returns a 0 it means that it completed its tasks successfully and if it returns any other number it is taken as an error code. Linux has standardized error codes that the functions of the drivers coded in the kernel must follow. In this case, the returned error is -16, which by looking at a table that shows the correspondence of the error codes with their meanings shows that a -16 means EEBUSY which means that the device was busy when tried to be probed.

The reason is very simple if proper attention is paid to the printed messages in the terminal: the function *of_mdiobus_register* is called TWO times for address 11 and the first one the function executes successfully returning a 0 but the second one it returns EEBUSY error which is expected because the device was already probed.

To find the error, one must know how probing a device works: 1. A struct representing a device driver must be passed to the kernel. In this struct, one of the sections is the device_probe which includes the name of the function to probe the device associated to the driver. In the case of macb this function is called *macb_probe* and is called one time per activated GEM in the device tree 2. In the execution flow of *macb_probe* it must register the PHY chip or chips in the MDIO bus it belongs to. Per each GEM node in the device node there can be one, several or none phy nodes and the *of_mdiobus_register* is executed as many times as phy child nodes the GEM node has. 3. Looking at the generated device tree that was transferred to the board was the next step. This can be done by using the device tree converter command on the system.dtb (device tree blob) located in `<project-folder>/images/linux*` to convert it to dtsi that can be human read:

```
hyperk1@hyperk1:/$ dtc -O dts -o SOMtree.dts system.dtb
```

When looking at the GEM1 node the following was found:

```
ethernet@ff0d0000 {
      compatible = "xlnx,zynqmp-gem\0cdns,zynqmp-gem\0cdns,gem";
      status = "okay";
      interrupt-parent = <0x04>;
      interrupts = <0x00 0x3d 0x04 0x00 0x3d 0x04>;
      reg = <0x00 0xff0d0000 0x00 0x1000>;
      clock-names = "pclk\0hclk\0tx_clk\0rx_clk\0tsu_clk";
      #address-cells = <0x01>;
      #size-cells = <0x00>;
      iommus = <0x0d 0x876>;
      power-domains = <0x0c 0x1f>;
      resets = <0x0e 0x1f>;
      clocks = <0x03 0x1f 0x03 0x6a 0x03 0x2f 0x03 0x33 0x03 0x2c>;
      phy-handle = <0x12>;
      phy-mode = "gmii";
      xlnx,ptp-enet-clock = <0x00>;
      local-mac-address = [00 0a 35 00 3b 27];
      phandle = <0x56>;

      phy@b {
         reg = <0x0b>;
         xlnx,phy-type = <0x05>;
         phandle = <0x57>;
      };

      phy@B {
         reg = <0x0b>;
         xlnx,phy-type = <0x05>;
         reset-gpios = <0x11 0x4e 0x00>;
         phandle = <0x12>;
      };
};
```

This means that there are two phy nodes that share the same phy address (reg variable) but with different names: one is phy@b and the other one is phy@B so there are two phy nodes when there should be one as it was defined in the *system_user.dtsi*.

The reason for this was that, while doing tests, the compiler didn't remove the other node from previous test from the device tree compilation so, it ended up with two phy nodes.

Cleaning the project build folder by doing `petalinux-build -x mrproper` solved the problem, showing just one message for registering the phy in address 11. However, now the other SFP was giving an error of EEBUSY. This was easier to solve: it was due to the reset-gpios property. This property is used for the driver to know which GPIO it must assert to reset the core. Both cores had a shared reset line meaning that, when the first was probed everything went fine but when the second one (address 11) was probed the same reset line is asserted meaning that the first core (address 9) got reset when it was ready. This means that when the kernel tried to create the ethernet interface it couldn't attach to that PHY because its configuration was wiped by the second reset.

The solution is using different reset lines for all cores that represent peripherals directly connected to the PS because the drivers in Linux use GPIOs as reset lines individually to initialise the devices.

7.2 Using Vitis

Another way to debug the linux kernel is Vitis. Thanks to Vitis debug capabilities, the linux kernel can be debugged and introduce breakpoints in the code. This procedure has been gathered from the blog entry from Xilinx support webpage PetaLinux Image Debug Series: Debugging the Linux Kernel in Vitis:

1. Configuring the Linux Image:

   - Enable some features in the kernel to allow debugging:

   ```
   hyperk1@hyperk1:/$ petalinux-config -c kernel
       -> kernel Hacking
       -> kernel debug -> Compile time checks and
       compiler options -> [*] Compile the kernel with debugging info

       CPU Power Management -> CPU Idle -> [] CPU idle PM support
   ```

   - Build the image

   ```
   hyperk1@hyperk1:/$ petalinux-build
   ```

   - Package the image into BOOT.BIN

   ```
   hyperk1@hyperk1:/$ petalinux-package --boot --force  --fsbl --fpga --u-boot
   ```

2. Debugging in Vitis: Load the boot image onto your SD/QSPI and debug on the running target.

3. Launch Vitis, and close the welcome screen.

4. Create a new Debug Configuration.

5. Double Click on Single Application Debug:

6. Select Apply and Debug. This will open the debug perspective. You should see the Cortex A53 running (our boot image).

7. Expand the Xilinx Software Command Tool (XSCT) window. Here, we want to pass the symbol files to the Cortex A53 #0 for the Linux kernel. This is the *vmlinux* file in the images/linux folder in the petalinux project directory.

8. We set the symbol files by using the *memmap* command in XSCT:

9. You can add a breakpoint by any of the following options:

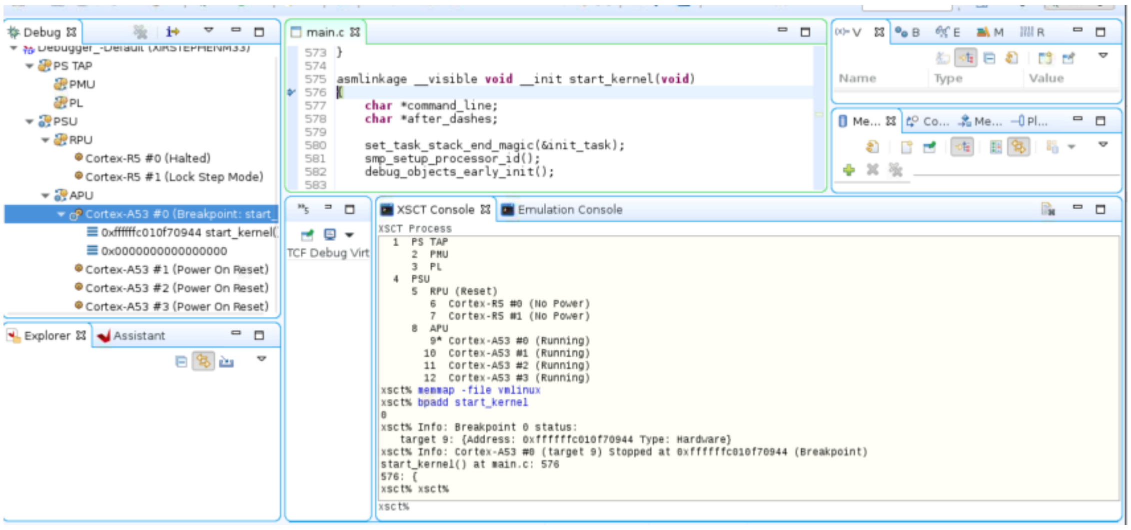
   - *bpadd function_name*
   - bpadd -file *"name_of_the_kernel_source_file"* -line *number_of_line* To find which functions names are valid, the following command can be used:

   ```
   aarch64-linux-gnu-objdump -t vmlinux | grep function_name
   ```

   It this line returns any result it means that the dump of the memory map of the kernel includes that name as a function_name and a breakpoint will be set when called from Vitis.

10. Once the breakpoints have been set, power cycle the board. The user can then use the debug functionality to step through the code.

    Figure 3.7.1 shows an example in the *kernel_start* function.


Debug
Debugger_-Default (XIRSTEPHENM33)
PS TAP
PMU
PL
PSU
RPU
Cortex-R5 #0 (Halted)
Cortex-R5 #1 (Lock Step Mode)
APU
Cortex-A53 #0 (Breakpoint: start_
0xffffffc010f70944 start_kernel(
0x0000000000000000
Cortex-A53 #1 (Power On Reset)
Cortex-A53 #2 (Power On Reset)
Cortex-A53 #3 (Power On Reset)
Explorer
Assistant
main.c
TCF Debug Virt
XSCT Console
Emulation Console
Name
Type
Value

```
573 }
574
575 asmlinkage __visible void __init start_kernel(void)
576 {
577     char *command_line;
578     char *after_dashes;
579
580     set_task_stack_end_magic(&init_task);
581     smp_setup_processor_id();
582     debug_objects_early_init();
583
```

```
XSCT Process
  1  PS TAP
     2  PMU
     3  PL
  4  PSU
     5  RPU (Reset)
        6  Cortex-R5 #0 (No Power)
        7  Cortex-R5 #1 (No Power)
     8  APU
        9* Cortex-A53 #0 (Running)
       10  Cortex-A53 #1 (Running)
       11  Cortex-A53 #2 (Running)
       12  Cortex-A53 #3 (Running)
xsct% memmap -file vmlinux
xsct% bpadd start_kernel
0
xsct% Info: Breakpoint 0 status:
   target 9: {Address: 0xffffffc010f70944 Type: Hardware}
xsct% Info: Cortex-A53 #0 (target 9) Stopped at 0xffffffc010f70944 (Breakpoint)
start_kernel() at main.c: 576
576: {
xsct% xsct%
xsct%
```

Figure 3.7.1: Debugging the kernel start function. This is the first function executed when Linux kernel boots [22]

7.3 Using debugfs

debugfs is another way for acquiring debug information. This is a file system that makes information about a process available to user space. It contains a family of function which are used to create a directory, a file, write on it,etc... like a normal file but in a special directory that is only meant to be modified by these functions, not by the user itself. This directory is usually */sys/kernel/debug*.

In the XU8 + ST1 reference PetaLinux image, if we go to that directory we can find several folders:

```
root@ST1ME-XU8-7EV-2I-D12E: ~$ /sys/kernel/debug# ls
asoc                dma_buf             iio                 pmbus
bdi                 dma_pools           memblock            pwm
block               dmaengine           mmc0                ramdisk_pages
bluetooth           dri                 mmc1                ras
clear_warn_once     dynamic_debug       mtd                 regmap
clk                 extfrag             opp                 regulator
debug_enabled       fault_around_bytes  output_status       remoteproc
device_component    gpio                pinctrl             sleep_time
devices_deferred    hid                 pm_genpd            split_huge_pages
```

For example, there is one folder named mmc1. This folder corresponds to the debug information of the mmc1 driver, which controls the Micro SD Card in the ST1 base board. Entering in this folder, we can find several text files. One of them is called clock, and shows the frequency of the clock that is used in the mmc1 controller which in this case, reads 50000000 which is equivalent to 50 MHz. Thanks to this file system, there is an ordered way to gather information about which parameters the driver has used to initialise the device.

Chapter 8

GPIO lines for peripherals control

The PS in the Zynq Ultrascale+ MPSoC includes General Purpose Input/Output pins that can be directly accessed from PetaLinux to trigger signals on FPGA IP Cores or to gather data from them.

The GPIO in this architecture is divided into several banks, as the Ultrascale+ Reference Manual states:

- Bank 0: 26-bit bank connected to MIO pins [0:25]
- Bank 1: 26-bit bank connected to MIO pins [26:51]
- Bank 2: 26-bit bank connected to MIO pins [52:77]
- Bank 3: 32-bit bank connected to EMIO signal sets [0:31]
- Bank 4: 32-bit bank connected to EMIO signal sets [32:63]
- Bank 5: 32-bit bank connected to EMIO signal sets [64:95]

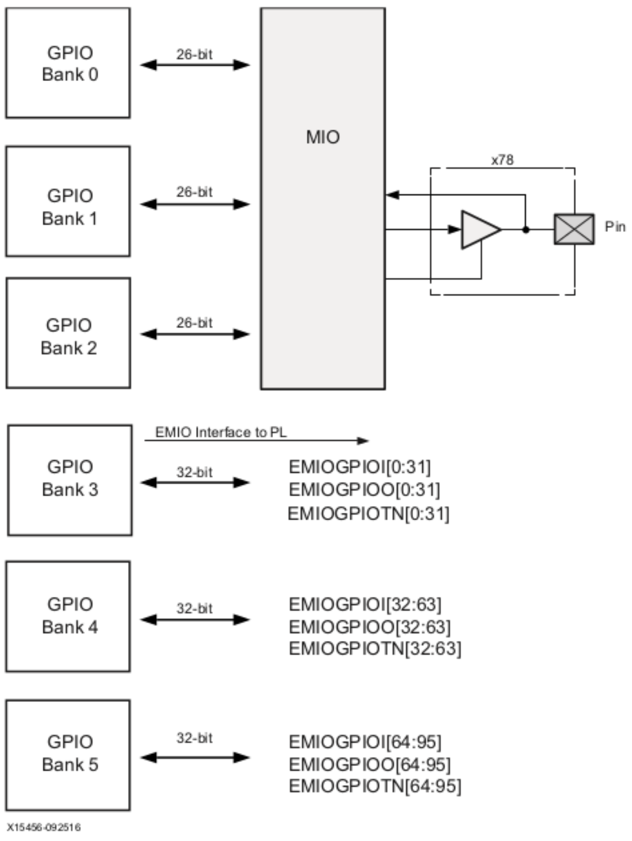
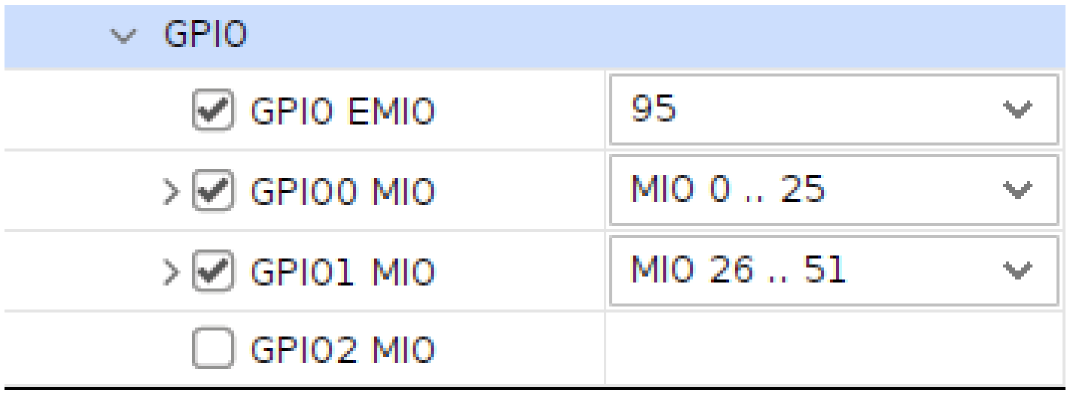


Figure 3.8.1: GPIO Block Diagram in Zynq UltraScale+[23]

These banks are grouped into two sets: the ones that control MIO pins and the ones that control EMIO signals. In the PS Core configuration, the user can select which GPIO banks are enabled. For the GPIO MIO banks, they are labelled with the number of the bank and put in a different row in the GUI while the GPIO EMIO there is a dropdown list in which a number ranging from 1 to 96 must be selected. This number is the width in bits that will be shown in the PS Core in IP integrator. If the maximum number in the GPIO EMIO list is not 96 but a smaller number it means that some of these lines are being used as PL Fabric Reset, which can also be configured. Every PL Fabric Reset line added takes one GPIO EMIO line, starting from the highest bit.



GPIO	
[✓] GPIO EMIO	95
[✓] GPIO0 MIO	MIO 0 .. 25
[✓] GPIO1 MIO	MIO 26 .. 51
[ ] GPIO2 MIO	

Figure 3.8.2: PS Configuration screen for GPIO selection[23]

After activating the GPIO and creating the XSA in Vivado that describes the system where the GPIOs are connected to, for example the reset lines, they can be firmware-controlled. As we already know, there are two environments that are aware of the hardware instantiated in the FPGA. Those are U-boot, as the booting environment can make good use of them to, for example, initialise some modules needed for next boot stages and PetaLinux itself.

8.1 From Uboot

GPIOs can be used freely in U-boot thanks to the command environment that it provides. The command of interest for controlling GPIOs is *GPIO*. This command has the utility to query and control GPIO pins with the following options followed by the desired pin to be controlled.

- input/output: sets the GPIO direction whether it is a pin where some signal is expected to be read or to be written from the CPU perspective.
- set: set to high level
- clear: set to low level
- toggle: change the state to the complementary one (high -> low) (low -> high).

The pin number can be easily deduced by typing *gpio status -a*. This command will return the state of all the GPIO pins detected by U-boot. Each line has the following structure:

```
gpio@ff0a0000173: output: 0 [ ]
      ^          ^       ^    ^
      |          |       |    |
gpio node     pin     I/O  state
reference   number   type
```

The pin number is listed in each line and they follow a numbering convention that matches with what is being described in the Ultrascale+ Reference Manual: first, the 3 GPIO MIO banks, which hold the first 78 numbers [0:77] and then the 3 GPIO EMIO banks, going just after them [78:173]. So, in the device tree and in the system itself, when referencing to a GPIO pin, the number of reference will always be the one decribed above. For the device-tree, the node representing the GPIO banks should also be referenced. In the Enclustra Reference design this is usually called *&gpio*.

```
gpio: gpio@ff0a0000 {
            compatible = "xlnx,zynqmp-gpio-1.0";
            status = "disabled";
            #gpio-cells = <0x2>;
            gpio-controller;
            interrupt-parent = <&gic>;
            interrupts = <0 16 4>;
            interrupt-controller;
            #interrupt-cells = <2>;
            reg = <0x0 0xff0a0000 0x0 0x1000>;
            power-domains = <&zynqmp_firmware PD_GPIO>;
        };
```

8.2 From PetaLinux

PetaLinux, following the philosophy of the linux kernel where “Everything is a file”, has the GPIO pins directly accessible from the user file system. For accessing peripherals, Linux uses the /sys directory, usually referred to as *sysfs*. The steps to, for example, toggle GPIO number 78 are as follows: Go to /sys/class/gpio directory. There is a file called export, which includes a list of which GPIO pins are controlled by the OS. By writing a number in that file, a GPIO pin can be “unlocked” to be controlled from the OS with the following command:

```
echo GPIO_number > /sys/class/gpio/export
```

where GPIO_number is computed in the following way: inside the */sys/class/gpio* directory there will be one or more folders named *gpiochip* followed by a number. Inside each of these folders there is a text file called *label*. The *label* that identifies the GPIO being discussed in this section are the *zynqmp_gpio*. Once the correct folder is known, pick the number that the *gpiochip* folder ends on. Then add the GPIO pin number in the convention explained previously. For example, if our folder is named *gpiochip334* and you want to use GPIO pin 78, you add 334 + 78 = 412, so the command would be *echo 412 > /sys/class/gpio/export*. This will create a new folder called *gpio412*. Inside *gpio412* by writting to several text files one can change the value of the GPIO and the I/O type just like in u-boot, by using the following commands:

```
echo "out" > /sys/class/gpio/gpio412/direction  # Set direction to output
echo "in" > /sys/class/gpio/gpio412/direction   # Set direction to input
echo 1 > /sys/class/gpio/gpio412/value         # Set value to high level
echo 0 > /sys/class/gpio/gpio412/value         # Set value to low level
```

This philosophy of editing files to change hardware state means that by using any linux library that allows to open, edit and close files like *fstream* can be used. However, there are some practices that should be followed when accessing *sysfs*. These guidelines are explained in the *sysfs* rules in the kernel documentation [24].

Chapter 9

QSPI redundant boot through TFTP server

This section covers the creation of the boot method that will be used in the DPB2 prototype. This method can already be implemented in the ST1 base board as it only involves the integrated Quad Serial Peripheral Interface (QSPI) memory in the XU8+ SOM, which is the same one used in the DPB2 prototype albeit with upgraded memory capacities.

9.1 Image size analysis

The OS image to perform the tests in the ST1 base board is an incomplete beta version of the OS image that will be loaded in the QSPI flash. This image has the following features, oriented to test a basic functionality of the OS together with the data link through TCP/IP:

- Init scripts to configure bonding
- Drivers for all the devices needed such as clock generator, PHY chips and the Gigabit Ethernet IP Core.
- Debugging tools and flags such as ethtool, phytool, mdiotool, etc…
- Software packages like the DAQ reference application, ifupdown to bring interfaces up and down
- SystemD init system to initialize the network services, NTP, among others.

The files that need to be loaded in the QSPI flash, their content and size are displayed in the following table:

File Name	Content	Estimated size
1. BOOT.BIN	FSBL, PMU, U-Boot	10MB
2. boot.scr	Booting command options	10kB (negligible)
3. image.ub	Kernel and rootfs	35MB

The Mercury XU8+ has a Spansion 64 MB QSPI flash. For the first tests (ST1 + XU8) this poses a significant disadvantage to testing redundant boot because there is not enough space to fit a copy of the image.ub due to its size. Also, there is just enough room for two BOOT.BIN and one image.ub, while the boot script doesn't pose any problem, there can be many copies but for the sake of simplicity, two are going to be introduced.

9.2 Boot script modifications

In more recent versions (from 2020.1) [1], the procedure to bootup the Linux kernel is using a script written with U-boot instructions. This script is transferred from non-volatile memory to main memory at a default address. This script is just made of the commands that Uboot uses to boot up Linux. Depending on the source where the kernel image is (JTAG, QSPI or SD Card) the commands will be different. As an addition, a fast remote way to load the image is also wanted to serve as a backup in case the local flash memory fails.

A crucial step for this project is getting redundancy in the boot process due to the long time the memories allocating the OS image must endure. To prevent that a hard failure on some memory bits causes a boot error, several copies of the same image can be included inside the QSPI memory.

However, not only the image is used for a correct boot up process, there are three files and each one of them has a different procedure of redundancy when it comes to its role in the booting process. These files are *BOOT.BIN*, *boot.scr* and *image.ub* and how booting process makes redundancy work for each stage is described as follows:

- **BOOT.BIN**: Corresponding to the first boot stage, once the device is powered on, the MPSoc reads 4 pins called the MODE pins and decides where is this file that contains the FSBL, Device tree, PMU firmware, bitstream and u-boot.elf to load the SSBL. Before loading this file, the device does a CRC check and if it fails then it will look for another *BOOT.BIN* file that can be a copy of the first one with an offset of 32 KB, so there can be several files in the same local memory let it be QSPI or eMMC and if there is an error loading the first then the second one can be used an so on. This procedure can be found in the Xilinx UG1085, chapter 11: Boot and configuration. This document has all the technical information about Zynq Ultrascale+ architecture.
- **boot.scr**: the boot script is the file searched after booting u-boot as SSBL and contains the U-boot command needed for loading the PetaLinux image. To look for this file, the environment variable *boot_targets* is used, where there is a list of all the possible places that the device can use to boot from. The first one will always be the one related to the MODE pins selection because the chip itself appends that name in the *boot_targets* variable. Each of the entries on this environment variable has another environment variable associated called *bootcmd_<device>*, which a script (all of it written inside the environment variable) that sources the *boot.scr* file. When sourcing the script, first it does a CRC check comparing it with a code that is generated in the header by `mkimage` command when generating the script file. If it fails it jumps to the next device in *boot_targets* variable until it finds a valid *boot.scr* file.
- **image.ub**: the FIT formatted image with linux kernel and rootfs in a single file. The redundancy for this is having several images which can be in the same local memory as *BOOT.BIN* and *boot.scr* or in a different one as the *boot_targets* variable is now used to iterate in the for loop inside *boot.scr* to search for a valid image.

The boot script can be edited by following this procedure as it is treated just as another Yocto recipe with some template files that are filled at building time with some environment variables:

1. Go to *<project-folder>/components/yocto/layers/meta-xilinx/meta-xilinx-core/recipes-bsp/u-boot/u-boot-zynq-scr* directory.
2. There will several files, each one a script for a specific hardware or case. Through a quick `petalinux-build` it was deduced that the template used to create the script was `boot.cmd.generic`
3. Open the corresponding file, in this case `boot.cmd.generic`.

4. Make the changes. Remember this is an u-boot script, so the commands there must belong to that environment.

5. To refresh the changes issue the following commands. They will clean the u-boot recipe and recompile it again with the changes made to the script.

```
hyperk1@hyperk1:/$ petalinux-build -c -u-boot -x cleansstate
```

```
hyperk1@hyperk1:/$ petalinux-build -c -u-boot
```

WARNING: There may be a bug in which the u-boot won't compile giving an error about *bb.ui*. This can be solved by executing `petalinux-build -x disttclean`

6. Open *boot.scr* and check that the changes have been made.

9.3 Memory map and boot flow proposal

For redundant boot up, two aspects need to be defined:

- **Memory map:** The QSPI memory will be the memory to be read when the board turns on. This means that all the files to do at least a basic load up of the second stage boot loader (U-boot) must be located there. Also, the memory layout is not as simple as loading one copy of each file and booting up. Figure 3.9.1 shows a memory map proposition for the 64MB QSPI flash, where due to space constraints, only the redundancy of the two first stages (BOOT.BIN and boot.scr) are tested.

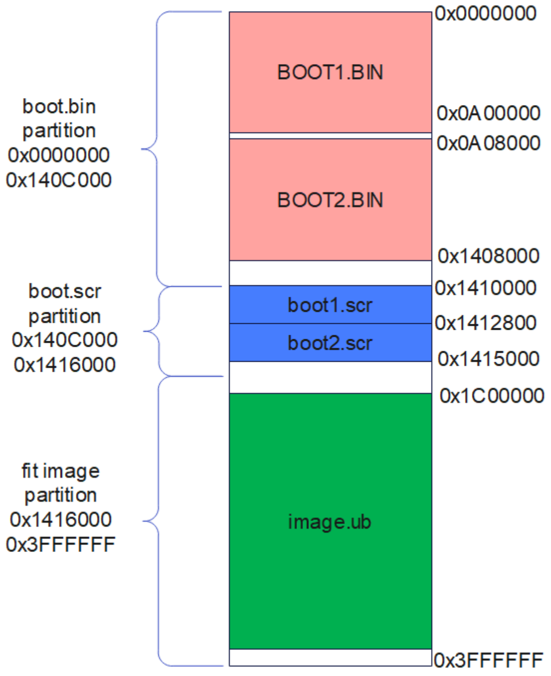


Figure 3.9.1: Spansion QSPI 64MB memory map. Each memory address points to a standard 1 byte cell

Having just one image.ub file (the linux kernel) is not critical because as long as U-boot loads correctly, another local memories can be read or even performing a PXE bootup by downloading an image from a TFTP server and boot from there.

- **Boot up flow:** It is mandatory to have a controlled boot up flow that takes advantage of having several copies of the same file. Considering how Zynq Ultrascale+ does the search for the files it needs to boot up, the boot flow shown in figure 3.9.2 has been made.

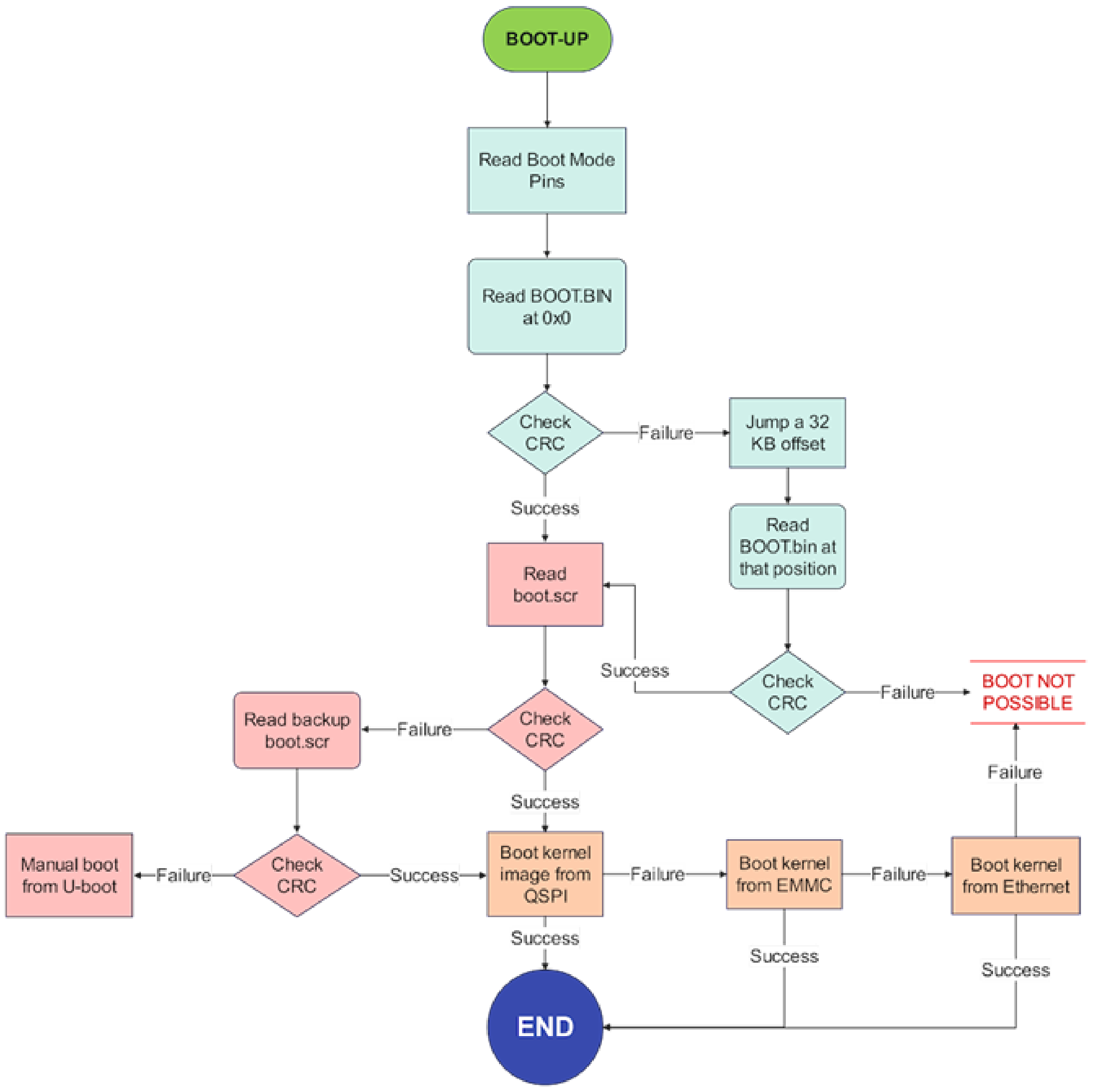


Figure 3.9.2: Boot up flow. The flow is divided into color-coded blocks where the blocks of the same color correspond to the same boot up phase, just reading different file or performing different operations to get the file from other sources

The most critical step is being able to find a valid *BOOT.BIN* file as it contains the first stage boot loader that initialises the hardware in the board. This is very similar to how a standard computer boots up: it has a ROM memory where it stores the essential bootup code and knows that is always going to be there at bootup. If some failure occurs due to component degradation, BOOT.BIN might not be correctly read and we would have a dead board just from the start. To avoid this, Zynq Ultrascale+ performs a CRC check and if BOOT.BIN is not valid, it will jump a 32KB offset and look for the next valid file there. That is why in figure 3.9.1, both *BOOT.BIN* are separated an offset of 32KB.

Part IV

Conclusions and Future Work

Chapter 1

Prototype tests and modifications for mass manufacturing

Now that a form factor has been decided, a prototype following it can be manufactured. However, what comes after that? The answer is, of course, a mass manufacturing of DPB that contain fixes made to sort out issues found during the prototypes' test.

The prototypes were received on July 2023 and there has been some time to perform tests on them. In figure 4.1.1, the DPB board (left) can be seen with many other boards like the Digitizer (right) and the Low Voltage board (left up).

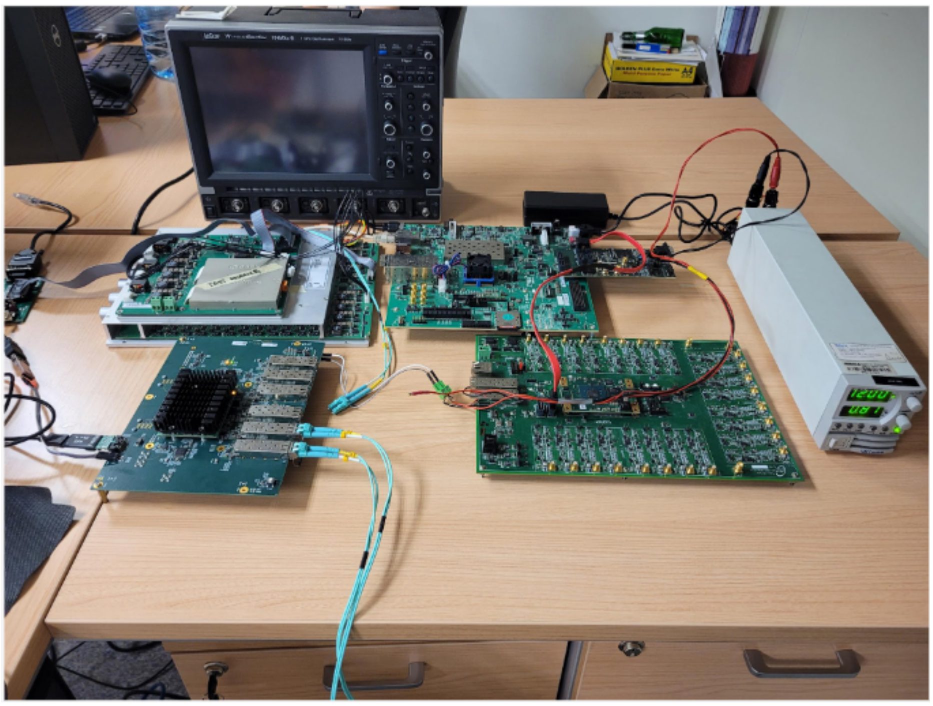


Figure 4.1.1: Testbench for the DPB assembled in the UPV i3M labs

Thus, a list with the changes to be made for the re-spin could be elaborated. The changes are minor and they do not involve changing the architecture, the form factor, the size of the board or the communication protocols to be used:

- **Clock routing:** There is some clocks that make the FPGA work that are routed in a complex way through a PLL. This can be simplified by just using a fixed oscillator as we only require a 125MHz clock.
- **SFP cages:** The SFP cages are not correctly aligned with the border, they should protrude a little bit beyond the board so that the flap used to insert the SFP module does not deteriorate when pulling in or out the module.

- **Assigning unique MAC Address:** this board has Ethernet interfaces and each one must have a unique MAC Address to identify itself in the physical layer. However, this board includes an EEPROM memory where an unique MAC address is stored. Normally, EEPROM memories are sold that only hold that MAC Address.
- **I2C bus slaves collision:** due to the use of multiplexers in the I2C bus, there is some times when the SFP modules, who share the same I2C address, might be connected at the same time to the I2C bus. So, when the I2C master wants to read one of them, both of the SFPs reply at the same time causing a bus collision. This can be solved by using only one I2C multiplexer instead of two, which is the current design.
- **Current leakage to the SFPs:** the power switches to turn off the SFPs individually to save power seem to not lock down the power totally as there are some SFPs that turn on when connected even if the power switch is disabled.
- **Fix the mounting holes:** there are some mounting holes that are not in the correct position with respect to the mechanical structure that will hold the DPB in place when mounted in the vessel. This should be reviewed together with the vessel team to find a solution.

After manufacturing a prototype with these changes, we will proceed with mass manufacturing, as the design is already done and only minor changes that will take from 4 to 6 weeks to be included in the schematics, now the unitary cost is much lower, given also that 900 boards will be manufactured instead of just 12. Table shows the manufacturing cost of the mass manufacturing of the DPB.

DPB Mass Manufacturing Item	**Total Estimated Cost [EUR]**
One time NRE development for upgrading the SOM	7.400,00 €
Manufacturing cost for 900 units	724.500,00 €
Total Cost without VAT	731.900,00 €

Table 1.1: DPB Mass manufacturing costs

As these 900 boards will be the ones mounted in HKK they need to be tested, so an automated test procedure should be developed together with a list of items to be fulfilled to assert that the board is ready to work in the detector. These tests, assembly and packaging of the boards to be sent to Japan will be done at CERN (Geneva) as a collaborative effort of all the institutions involved in this project.

Chapter 2

Lessons learned

The Hyper-Kamiokande project is a long-term effort to build the next generation of neutrino detectors. As I said in the first chapter of this TFM, this is translated into a several year effort that gave birth to two TFM, the MUISE one [1] and this one, being the latter the continuation of the first. In the first TFM, most of the time went into learning to develop software and use the tools provided by Xilinx in their standard evaluation boards. In this case, the goals are more project-specific as the form factor, architecture and specifications are fixed.

At the end of this 6 month-time work, the following conclusions can be enumerated:

- **The SOM form factor:** learning about the existence of this brand new type of PCB has been very revealing to me personally as it adds even more flexibility if it is possible to an already flexible platform. FPGAs were born with this concept in mind; use generic re-programmable logic cells to implement your own combinational and sequential functions which, at the end of the day, is the same as creating hardware from an ASIC. The SOM form factor will allow to have all the complexity inside an already commercially available and thoroughly tested product while the base board is the perfect place for our I/O needs. Remember that the DPB needs to have 6 SFP port together with MiniSAS and even a custom connector for interfacing with the power supply board (LV).

- **PetaLinux versions and patching:** One of the problems found during the setup for using the SOM was the fact that Enclustra had BSP templates that used version 2022.1 and the version used in HKK is 2022.2 so an upgrade was needed using that template as a departure point. The process went smoothly until the two ethernet interfaces of the ST1+ baseboard were tested, seeing that one of them did not work. This issue was a very good lesson to learn how to patch PetaLinux and specially how to inspect the codebase looking for bugs using the procedures stated in Part III, chapter 7.

- **Bonding:** This special addon driver for the network capabilities in Linux proved to be very useful as it configures several physical interfaces to be seen as just one virtual interface from the outside, making the redundancy work much more simple and better handled as Linux takes care of everything. Your application only needs to point to that virtual interface and Linux will decide through which physical interface the data will be sent.

- **Fiber optics standard:** Working with ethernet FPGA cores that implement the Physical Coding Sublayer and the Physical Medium access also helped to learn which standards at the physical layer exist. Normally, we are used to have 1000-BaseT in our homes (The standard copper twisted pair Rj45 Gigabit connection). However, as we are talking about fiber optics here, we have 1000-BaseX which can be 1000-BaseLX or 1000-BaseSX depending if the fiber is single mode or multi-mode.

- **Booting with fallback measures:** we are used to systems were only one source can be used for booting normally. For computers, just the main hard drive, for smartphones, the main flash storage. Why not combine several of them? From NOR flash, going through emmc, even downloading the OS from the internet on the spot, figure 3.9.2 shows how a redundant bootup can be built easily. HKK project demands this kind of mechanism for the DPB because of the inability to repair it or replace it without having to drain a whole tank of water, which is both time consuming and expensive.
- **The DPB next prototypes:** there is still a lot of work to do. The first DPB baseboard prototype is already in our laboratories undergoing extensive testing as shown in figure 4.1.1. As of November 2023, the integration tests are still undergoing to connect the digitizer, the timing link, the DAQ and the DPB together to have a Vertical Slice Test (VST) in January to be able to detect errors for the next prototype.

As the last paragraph of this TFM I just have to say, even after writing two Final Master Thesis based on the same line of work, there is still so much work to do and so much time left (3 years at a minimum) that I also decided to do my PhD Thesis in this group, as I see it as a grand opportunity to work in an international prestigious project performing my tasks using the latest advances in electronics, which is something I personally enjoy. For the reader who has reached this point, receive my sincerest gratitude for spending part of your time to read this. Thank you.

Bibliography

[1] 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”. In: (Sept. 2023).

[2] Chang Kenneth. “Tiny, plentiful and really hard to catch”. In: *New York Times* (June 2011).

[3] University of Tokyo and Institute for Cosmic Ray Research. *Hyper-Kamiokande. Peering into the Universe and its elementary particles from underground.* `https://www.hyperk.org/wp-content/uploads/2015/01/HKen-low.pdf`. 2015.

[4] Scoles Sarah. “Physicists Go Deep in Search of Dark Matter”. In: *Scientific American* (July 2017).

[5] K. Abe, Y. Hayato, T. Iida, K. Iyogi, and J. Kameda et al. “Calibration of the Super-Kamiokande detector”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 737 (Feb. 2014), pp. 253–272. DOI: `10.1016/j.nima.2013.11.081`. URL: `https://doi.org/10.1016%2Fj.nima.2013.11.081`.

[6] Francesca Di Lodovico and on behalf of the Hyper-Kamiokande Collaboration. “The Hyper-Kamiokande Experiment”. In: *Journal of Physics: Conference Series* 888.1 (Sept. 2017), p. 012020. DOI: `10.1088/1742-6596/888/1/012020`. URL: `https://dx.doi.org/10.1088/1742-6596/888/1/012020`.

[7] STMicroelectronics. *Uboot PXE documentation.* `https://github.com/STMicroelectronics/u-boot/blob/v2022.10-stm32mp/doc/README.pxe`. 2022.

[8] Advanced Micro Devices (AMD). *Zynq UltraScale+^{TM} MPSoC. Heterogeneous Multiprocessing Platform for Broad Range of Embedded Applications.* `https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html`. 2023.

[9] *UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices.* 1.0. `https://docs.xilinx.com/v/u/en-US/wp477-ultraram`. AMD Xilinx. June 2016.

[10] Shant Chandrakar, Dinesh D. Gaitonde, and Trevor Bauer. “Enhancements in UltraScale CLB Architecture”. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2015).

[11] Evan Leal (AMD) and Chetan Khona (AMD). *Xilinx System-on-Module Product Announcement.* `https://www.xilinx.com/publications/presentations/xilinx-kria-som-product-presentation.pdf`. 2021.

[12] Juan Miguel Ibáñez de Aldecoa Quintana and Ministerio de Industria y Comercio. Gobierno de España. *NIVELES DE MADUREZ DE LA TECNOLOGÍA TECHNOLOGY READINESS LEVELS.TRLS. UNA INTRODUCCIÓN.* `https://www.mincotur.gob.es/publicaciones/publicacionesperiodicas/economiaindustrial/revistaeconomiaindustrial/393/notas.pdf`. Note about the TRL from the Spanish Government. 2020.

[13] *PG046. Aurora 8B/10B LogiCORE IP Product Guide.* 11.1. `https://docs.xilinx.com/r/en-US/pg046-aurora-8b10b`. AMD Xilinx. Oct. 2023.

[14] *PG074. Aurora 64B/66B LogiCORE IP Product Guide*. 12.0. `https://docs.xilinx.com/r/en-US/pg074-aurora-64b66b`. AMD Xilinx. May 2023.

[15] Enclustra Gmbh. *Mercury XU8+ SOM schematics*. `https://download.enclustra.com/public_files/SoC_Modules/Mercury+_XU8/Mercury_XU8-R2-1_User_Schematics_V3.pdf`. Generated Altium Schematics for the Mercury XU8+ SOM. 2020.

[16] Enclustra Gmbh and Jose Francisco Toledo. *DPB2 prototype base board schematics*. Generated Altium Schematics for the DPB2 prototype. Private document. 2023.

[17] *PG142. AXI UART Lite LogiCORE IP Product Guide*. 2.0. `https://docs.xilinx.com/v/u/en-US/pg142-axi-uartlite`. AMD Xilinx. Apr. 2017.

[18] A. Brown and G. Wilson. *The Architecture of Open Source Applications, Volume II*. The Achrictecture of Open Source Applications. CreativeCommons, 2012. ISBN: 9781105571817. URL: `https://books.google.es/books?id=wQ--AwAAQBAJ`.

[19] LPNHE Paris, INFN Roma, and CEA IRFU Saclay. *Time Distribution System Proposal for the Hyper Kamiokande Experiment's Far Detector*. Time distribution information document for the collaboration evaluation. Private document. Oct. 2023.

[20] *PetaLinux Tools Documentation. Reference Guide. UG1144*. 2022.2. `https://docs.xilinx.com/r/en-US/ug1144-petalinux-tools-reference-guide`. AMD Xilinx. Oct. 2022.

[21] *Intel Ethernet Controller X710/XXV710/XL710 Feature Support Matrix*. 5.5. Intel Corporation. Aug. 2023.

[22] AMD Xilinx. *PetaLinux Image Debug Series: Debugging the Linux Kernel in Vitis*. `https://support.xilinx.com/s/article/1156809?language=en_US`. Article post from Xilinx about different debugging methods for Linux in Vitis. 2023.

[23] *Zynq UltraScale+ Device Technical Reference Manual*. 2.3.1. `https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm`. AMD Xilinx. Jan. 2023.

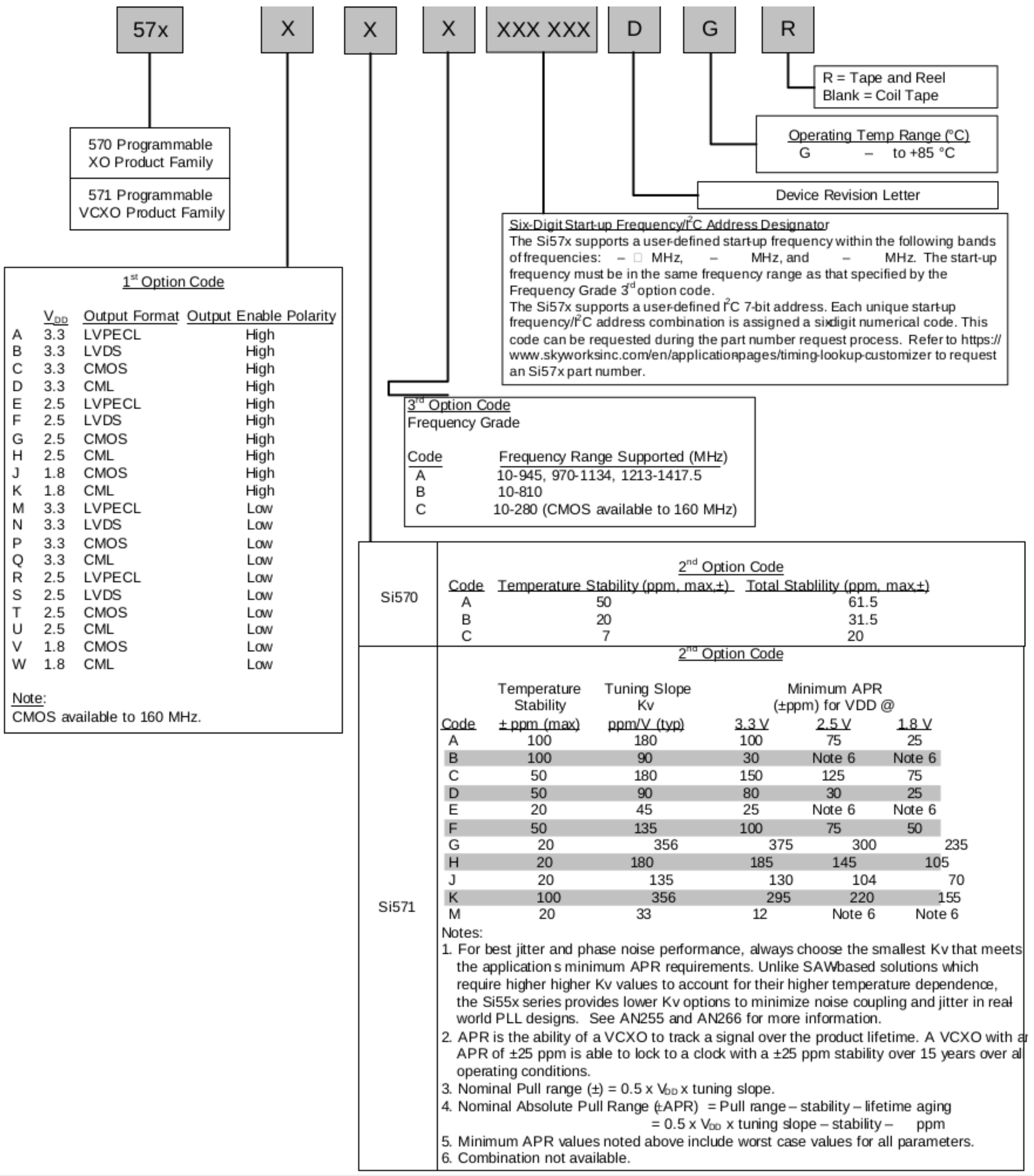
[24] The Linux Kernel Development Community. *Rules on how to access information in sysfs*. `https://docs.kernel.org/admin-guide/sysfs-rules.html`. 2023.

Part V

Annexes

Annex 1

Additional figures



R = Tape and Reel
Blank = Coil Tape

Operating Temp Range (°C)
G – to +85 °C

Device Revision Letter

570 Programmable XO Product Family

571 Programmable VCXO Product Family

Six-Digit Start-up Frequency/I^{2}C Address Designator
The Si57x supports a user-defined start-up frequency within the following bands of frequencies: – MHz, – MHz, and – MHz. The start-up frequency must be in the same frequency range as that specified by the Frequency Grade 3rd option code.
The Si57x supports a user-defined I^{2}C 7-bit address. Each unique start-up frequency/I^{2}C address combination is assigned a six-digit numerical code. This code can be requested during the part number request process. Refer to https://www.skyworksinc.com/en/application-pages/timing-lookup-customizer to request an Si57x part number.

1st Option Code

	V_{DD}	Output Format	Output Enable Polarity
A	3.3	LVPECL	High
B	3.3	LVDS	High
C	3.3	CMOS	High
D	3.3	CML	High
E	2.5	LVPECL	High
F	2.5	LVDS	High
G	2.5	CMOS	High
H	2.5	CML	High
J	1.8	CMOS	High
K	1.8	CML	High
M	3.3	LVPECL	Low
N	3.3	LVDS	Low
P	3.3	CMOS	Low
Q	3.3	CML	Low
R	2.5	LVPECL	Low
S	2.5	LVDS	Low
T	2.5	CMOS	Low
U	2.5	CML	Low
V	1.8	CMOS	Low
W	1.8	CML	Low

Note:
CMOS available to 160 MHz.

3rd Option Code
Frequency Grade

Code	Frequency Range Supported (MHz)
A	10-945, 970-1134, 1213-1417.5
B	10-810
C	10-280 (CMOS available to 160 MHz)

Si570 — 2nd Option Code

Code	Temperature Stability (ppm, max,±)	Total Stablility (ppm, max,±)
A	50	61.5
B	20	31.5
C	7	20

Si571 — 2nd Option Code

	Temperature Stability	Tuning Slope Kv	Minimum APR (±ppm) for VDD @		
Code	± ppm (max)	ppm/V (typ)	3.3 V	2.5 V	1.8 V
A	100	180	100	75	25
B	100	90	30	Note 6	Note 6
C	50	180	150	125	75
D	50	90	80	30	25
E	20	45	25	Note 6	Note 6
F	50	135	100	75	50
G	20	356	375	300	235
H	20	180	185	145	105
J	20	135	130	104	70
K	100	356	295	220	155
M	20	33	12	Note 6	Note 6

Notes:
1. For best jitter and phase noise performance, always choose the smallest Kv that meets the application s minimum APR requirements. Unlike SAW-based solutions which require higher higher Kv values to account for their higher temperature dependence, the Si55x series provides lower Kv options to minimize noise coupling and jitter in real-world PLL designs. See AN255 and AN266 for more information.
2. APR is the ability of a VCXO to track a signal over the product lifetime. A VCXO with an APR of ±25 ppm is able to lock to a clock with a ±25 ppm stability over 15 years over all operating conditions.
3. Nominal Pull range (±) = 0.5 x V_{DD} x tuning slope.
4. Nominal Absolute Pull Range (±APR) = Pull range – stability – lifetime aging
= 0.5 x V_{DD} x tuning slope – stability – ppm
5. Minimum APR values noted above include worst case values for all parameters.
6. Combination not available.

Figure 5.1.1: Si570 specs included in the HiTech Global FMC

Annex 2

Additional Listings

Gigabit Ethernet MAC device tree configuration with two phy chips:

```
&gem0 {
    status = "okay";
    /delete-property/ local-mac-address;
    phy-mode = "rgmii-id";
    phy-handle = <&phy0>;
    phy0: phy@3 {
        reg = <3>;
        txc-skew-ps = <1800>;
        txen-skew-ps = <420>;
        txd0-skew-ps = <420>;
        txd1-skew-ps = <420>;
        txd2-skew-ps = <420>;
        txd3-skew-ps = <420>;
        rxc-skew-ps =  <900>;
        rxdv-skew-ps = <420>;
        rxd0-skew-ps = <420>;
        rxd1-skew-ps = <420>;
        rxd2-skew-ps = <420>;
        rxd3-skew-ps = <420>;
    };
    phy1: phy@7 {
        reg = <7>;
        txc-skew-ps = <1800>;
        txen-skew-ps = <420>;
        txd0-skew-ps = <420>;
        txd1-skew-ps = <420>;
        txd2-skew-ps = <420>;
        txd3-skew-ps = <420>;
        rxc-skew-ps =  <900>;
        rxdv-skew-ps = <420>;
        rxd0-skew-ps = <420>;
        rxd1-skew-ps = <420>;
        rxd2-skew-ps = <420>;
        rxd3-skew-ps = <420>;
    };
};

&gem3 {
    status = "okay";
    /delete-property/ local-mac-address;
    phy-handle = <&phy1>;
    phy-mode = "rgmii-id";
};
```

Si5338 PLL Device tree configuration:

```
#include <dt-bindings/clock/clk-si5338.h>
&i2c0 {
    si5338@70 {
        compatible = "silabs,si5338";
        reg = <0x70>;
        #address-cells = <1>;
        #size-cells = <0>;
        #clock-cells = <1>;
        /* connect xtal to 25MHz, in5/in6 to 100MHz */
        clocks = <0>, <0>, <&ref100>, <0>, <0>;
        clock-names = "xtal", "in12", "in3", "in4", "in56";
        /* connect xtal as source of refclk */
        silab,ref-source = <SI5338_REF_SRC_CLKIN3>;
        /* connect in5/in6 as source of fbclk */
        silab,fb-source = <SI5338_FB_SRC_NOCLK>;
        /* connect divrefclk as source of pll */
        silab,pll-source = <SI5338_PFD_IN_REF_REFCLK>;
        /* Choose one MS for pll master */
        silabs,pll-master = <0>;
        /* Specify pll-vco frequency. pll-master is ignored. */
        silabs,pll-vco = <2450000000>;
        /* output list */
        clkout0 {
            reg = <0>;
            silabs,drive-config = "3V3_LVDS";
            silabs,clock-source = <SI5338_OUT_MUX_MSN>;
            silabs,disable-state = <SI5338_OUT_DIS_ALWAYS_ON>;
            clock-frequency = <5000000>;
            enabled;
        };
        clkout1 {
            reg = <1>;
            silabs,drive-config = "3V3_LVDS";
            silabs,clock-source = <SI5338_OUT_MUX_MSN>;
            silabs,disable-state = <SI5338_OUT_DIS_ALWAYS_ON>;
            clock-frequency = <156250000>;
            enabled;
        };
        clkout2 {
            reg = <2>;
            silabs,drive-config = "3V3_LVDS";
            silabs,clock-source = <SI5338_OUT_MUX_MSN>;
            silabs,disable-state = <SI5338_OUT_DIS_HIZ>;
            clock-frequency = <156250000>;
        };
        clkout3 {
            reg = <3>;
            silabs,drive-config = "3V3_LVDS";
            silabs,clock-source = <SI5338_OUT_MUX_MSN>;
            silabs,disable-state = <SI5338_OUT_DIS_HIZ>;
            clock-frequency = <156250000>;
        };
    };
};
```