UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dept. of Electronic Engineering

Discussion and preparation of a FPGA-based hardware platform with embedded operative system for data processing tasks inside the neutrino detector Hyper-Kamiokande

Master's Thesis

Master's Degree in Electronic Systems Engineering

AUTHOR: Gómez Gambín, Alejandro

Tutor: Herrero Bosch, Vicente

Cotutor: Toledo Alarcón, José Francisco

Cotutor: Esteve Bosch, Raul

ACADEMIC YEAR: 2022/2023

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 advices has been crucial whenever I got stuck.

To the rest of the Hyper Kamiokande team who has provided me with the necessary documentation to understand their involvenment 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. El funcionamiento de este sistema depende de que la detección se realice en un tanque de agua enriquecida de Deuterio. Esto plantea un problema de acceso para el mantenimiento de la electrónica que digitaliza y procesa los datos para enviarlos a un servidor externo.

Este Trabajo de Fin del Máster Universitario en Ingeniería de Sistemas Electrónicos versa sobre la preparación de una plataforma de desarrollo basada en Zynq UltraScale+ MPSOC de Xilinx para alojar un sistema que se mantenga en funcionamiento durante 10 años sin necesidad de mantenimiento lo que implica un estudio de las partes que compondrán esta plataforma, así como una discusión del método más seguro de arranque y configuración de esta.

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 aquests successos, s'utilitzen fotomultiplicadors (PMT) que permeten capturar els fenòmens de Llum de Cherenkov. El funcionament d'aquest sistema depén del fet que la detecció es realitze en un tanc d'aigua enriquida de Deuteri. Això planteja un problema d'accés per al manteniment de l'electrònica que digitalitza i processa les dades per a enviar-los a un servidor extern

Aquest Treball de Fi del Màster Universitari en Enginyeria de Sistemes Electrònics versa sobre la preparació d'una plataforma de desenvolupament basada en Zynq UltraScale+ MPSOC de Xilinx per a allotjar un sistema que es mantinga en funcionament durant 10 anys sense necessitat de manteniment el que implica un estudi de les parts que compondran aquesta plataforma, així com una discussió del mètode més segur d'arrancada i configuració d'aquesta.

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. The operation of this system depends on the detection being carried out in a tank of Deuterium enriched water. This poses an access problem for the maintenance of the electronics that digitizes and processes the data to send it to an external server.

This Final Project of the masters degree in Electronic Systems Engineering deals with the preparation of a development platform based on Zynq UltraScale+ MPSOC from Xilinx to host a system that will remain in operation for 10 years without maintenance, which implies a study of the parts that will form this platform, as well as a discussion of the safest method of starting and configuring it.

Table of Contents

List of Figures

List of Tables

Acronyms

APU Application Processing Unit.

ASIC Application Specific Integrated Circuit.

ATM Analog Timing Modules.

BSP Board Support Package.

CAN Controller Area Network.

CERN European Organization for Nuclear Research.

CLB Configurable Logic Blocks.

CLI Command Line Interface.

CP Charge Parity.

CPU Central Processing Unit.

CRC Cyclic Redundancy Code.

DAQ Data Adquisition Module.

DHCP Dynamic Host Configuration Protocol.

DIPC Donostia lnternational Physics Center.

DMA Direct Memory Access.

DPB Data Processing Board.

DT Device Tree.

ECC Error Correction Code.

EMAC Ethernet Media Access Controller.

eMMC External Multi Media Card.

EvB Evaluation Board.

FD Far Detector.

FF Far Facilities.

FIT Flattened uImage Tree.

FMC FPGA Mezzanine Card.

FPGA Field Programmable Gate Array.

FSBL	First Stage Boot Loader.
GEM	Gigabit Ethernet MAC.
GPL	General Public License.
GPU	Graphics Processing Unit.
GTH	Gigabit Transceivers Type H.
GTY	Gigabit Transceivers Type Y.
HD	High Density.
HDL	Hardware Description Language.
HKK	Hyper-Kamiokande.
HLS	High Level Synthesis.
HP	High Performance.
HV	High Voltage Module.
i3M	Institute for Molecular Imaging Technologies.
ID	Inner Detector.
IDE	Integrated Development Environment.
ILA	Integrated Logical Analyzer.
In2p3	Institut national de physique nucléaire et de physique des particules.
IP Cores	Intellectual Property Cores.
JTAG	Join Test Action Group.
LUT	Look Up Table.
LV	Low Voltage Module.
MCIN	Ministerio de Ciencia e Innovación.
MDIO	Management Data Input/Output.
MGT	Multi Gigabit Transceivers.
MII	Media Independent Interface.
MMC	Multi-Media Card.
MoU	Memorandum of Understanding.
mPMT	multi-Photomultiplier tube.
MPSoC	Multi Processing System on Chip.
MTD	Memory Technology Devices.

NFS Network File System.

NP Non-Deterministic Polynomial.

NTP Network Time Protocol.

OD Outer Detector.

OS Operative System.

OTP One Time Programmable.

OUI Organizationally Unique Identifier.

PC Personal Computer.

PCB Printed Circuit Board.

PCIe Peripheral Component Interconnect Express.

PHY Physical Layer.

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.

RAM Random Access Memory.

Rn Radon.

ROM Read Only Memory.

RPU Real Time Processing Unit.

RTL Register Transfer Level.

RTOS Real Time Operating System.

SATA Serial Advanced Technology Attachment.

SD Secure Device.

SDK Source Development Kit.

SFP Small Form Factor Hot Pluggable.

SGMII	Serial Gigabit Media-Independent Interface.
SKK	Super-Kamiokande.
SMB	Server Message Block.
SoC	System on Chip.
SOM	System On Module.
TFM	Final Master Thesis.
TFTP	Trivial File Transfer Protocol.
UAM	Universidad Autónoma de Madrid.
UPV	Polythecnic University of Valencia.
USB	Universal Serial Bus.
VHDL	Very High (Speed Integrated Circuits) Hardware Description Language.
XDC	Xilinx Design Constraints.
XNF	Xilinx Netlist Format.
XSCT	Xilinx Software Command-Line Tool.
XSDB	Xilinx System Debugger.

Part I

Introduction: The HKK Project

Chapter 1

Introduction: objectives

Electronics has always been a main driving force for the development of modern society. It has influenced several fields, going from practical use cases, like control systems in a car to being the support force for theoretical studies, providing the much needed computational performance or the sensors used to measure nature phenomena.

This Final Master Thesis (TFM) of the Master in Electronic Systems is a deep dive in how electronics is able to help in the field of physics, which strives to provide explanations and evidence for the behavior of the universe in all of its aspects: matter, its motion through space and time and the energy and forces that drive matter to do so.

This objective section lists the goals to be achieved by this specific Final Master Thesis inside the Hyper-Kamiokande (HKK) project, described in the next chapters. It is a huge project that originated in Japan with a high degree of organizational complexity , including 300 researchers from 75 institutes in 15 countries as of 2018 and this number has grown since then. This makes it very difficult but very important to coordinate. The UPV duties are the core of the electronics of the experiment due to the nature of the module being designed by the UPV, specifically by the Institute for Molecular Imaging Technologies (i3M). This commitment is gathered in a Memorandum of Understanding (MoU) signed by the Ministerio de Ciencia e Innovación and the University of Tokyo. This TFM is conceived as a starting point for developing a Data Processing Board (DPB) from a hardware and software standpoint, making use of the available tools nowadays. As such, the objectives can be summarized in the following way:

- **Prepare a development platform** that allows to perform the creation from the ground up of an embedded OS and its corresponding hardware platform. As it will be explained in the next part, the chosen platform is Xilinx Zynq UltraScale+ MPSoC, a hybrid chip that contains both a multicore ARM CPU and a FPGA. The development platform must consist of the Vivado Design Suite for creating the hardware that will be instantiated in the FPGA and the Source Development Kit (SDK) for configuring and building a bootable image of the embedded OS.
- **Learn the functionality** of the SDK being used and the design workflow that an embedded OS engineer should follow to, departing from a fixed physical hardware platform, that is an evaluation board, know how to configure the different elements to prepare a system that can boot the embedded OS and how to interact with it.
- **Learn how to make changes** from the default configuration of the embedded OS to suit the needs of the project. For instance, adding more ethernet interfaces, add serial controllers for sensors, among other functionalities.
- **Study the boot flow:** going from complete shutdown system to the OS up and running requires a series of steps that initialize the CPU, prepare a environment for searching for the OS image and boots it. This flow is very critical for HKK project because one of the most important aspects for this platform to have is reliability so the boot process must be tightly controlled to ensure that it completes successfully and if it does not, activate backup mechanism.

- **Manage memory in the system:** booting up from different sources means that the boot components need to be stored in different locations in the board so the mechanism to flash any of these memory with the required information and accessing it from the boot stage to perform boot up or from the OS itself to perform configuration, update and maintenance tasks need to be investigated.

- **Application programming for embedded OS:** the architecture of the platform CPU is not the same as the one present in a standard computer and the tools and libraries available are different versions of the ones found on a standard PC. However, it will be seen that everything is standardized at this point so that the procedure for programming applications at high level is the same and the underneath libraries are in charge of managing the differences between architectures.

- **Network connection redundancy:** one of the specifications of the DPB board is to have redundant optical fiber links. This means having a lot of interfaces instantiated in the system and not only that, they need to coordinate so that each interface has a backup that performs exactly the same function and serves as fallback when the other one fails. The embedded OS needs to be aware of this so some configuration needs to be made.

- **Slow-control controllers:** sensors in the project will use serial protocols to communicate their measurements to the DPB. The DPB must have controllers to drive these sensors. So, the last objective of this TFM would be learning how to instantiate them, manage them and code software to perform the slow control tasks and package that data to send it to the DAQ.

To summarize, all the objectives fall under the same purpose: learn how to establish a development platform with the same architecture that the final DPB in the vessel will have. This can be achieved thanks to Xilinx providing general purpose PCB with Zynq chips and a good amount of I&O that can be useful for performing a lot of applications. Once the platform and software has been developed in this general-purpose board which, as it is made by Xilinx, there is a lot of documentation and example designs, it can be ported to a more specific hardware with a subset of the features of the evaluation board that are truly needed.

Regarding the organization of the project, which will be explained in more detail later, this TFM will be used as a starting point to develop the SOM that will be central to the electronics module that the UPV is developing for HKK. This is part of my job in the Institute for Molecular Imaging Technologies where I am hired and collaborating with international investigation groups from Italy like the National Institute of Nuclear Physics, Great Britain (Warwick University) or Japan (University of Tokyo). Furthermore, there is also collaboration with private companies such as Enclustra, which have been hired to design the PCB for the final design of the DPB and other public institutions such as the Universidad Autónoma de Madrid (UAM), which provides me with technical support. Their know-how has been crucial for me to learn in just a year the hardware and software architecture of the system to be developed and the needed tools.

As a little preview of this TFM, I can say that all the objectives listed here have been reached with more details coming in the next chapters. These objectives are not easy as they involve understanding the system from a hardware and software standpoint starting from scratch. Hence, it has been decided to make a comprehensive explanation for each part of the procedure for developing in this platform while doing it in English because this thesis will belong to the official project documentation.

Chapter 2

The history of neutrinos

Inside Physics, which is the general name given to this natural science, there exist several fields which study the different aspects of the universe. In this TFM, the focus will be placed in *Particle Physics* also known as high energy physics, which studies the fundamental particles that form matter [1]. Particle physics has demonstrated the existence of fermions or bosoms, for instance. This field provides a serious challenge because these particles cannot be detected easily even though their existence has been proved theoretically.

Inside these difficult-to-detect particles, the neutrino can be found: a subatomic particle that is formed when a radioactive and scattering process takes place. In this case, the neutrino is formed due to the beta decay, postulated in Fermi's theory, where one large neutral particle (n^0) decays into a proton(p^+), an electron(e^-) and a neutrino$(\overline{\nu}_e)$.

$$n^0 \rightarrow p^+ + e^- + \overline{\nu}_e$$

The first postulation of the neutrino as a particle was given by Wolfgang Pauli [2] in 1930 to explain how the beta decay fulfills the principles of conservation of energy, momentum and angular momentum (spin) and then unified by Enrico Fermi [3] with the positron model by Paul Dirac and the neutron-proton model by Werner Heisenberg. However, he faced fierce opposition due to the theory being *too remote from reality*.

Nonetheless, in 1942, Wang Ganchang [4] proposed using beta capture as a way to experimentally detect neutrinos, which would give Fermi's theory solid evidence.

This experiment was drive into reality by Clyde Cowan, Frederick Reines, Francis B. "Kiko" Harrison, Herald W. Kruse, and Austin D. McGuire by creating antineutrinos in a nuclear reactor by beta decay [5]. Then this antineutrinos reacted with protons and produced the expected neutrons and positrons, as stated by Fermi's theory:

$$\overline{\nu}_e + p^+ \rightarrow n^0 + e^+$$

As the positron is the antiparticle of the electron, when it finds one, they annihilate each other, releasing two gamma rays (γ), which can be detected. The neutron is detected by making a nucleus catpure it, emitting another gamma ray. If these two events coincide, then it means there has been an antineutrino interaction in the reactor. This experiment was awarded with the Physics Nobel Prize in 1995 [6].

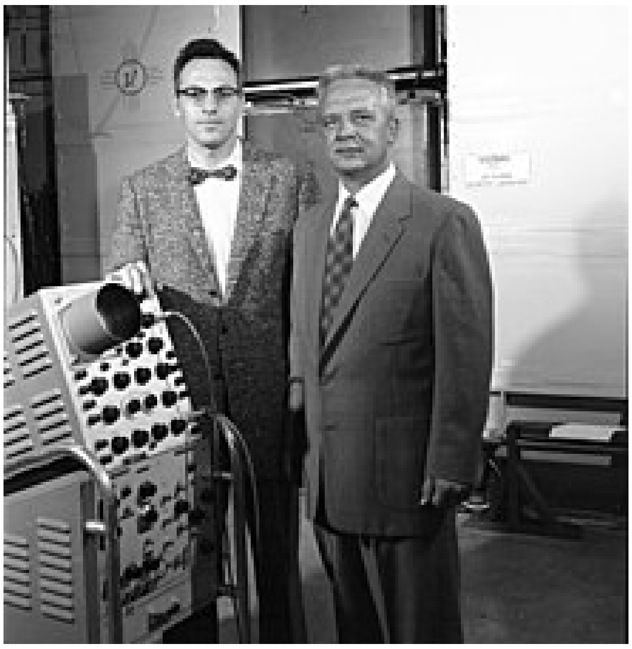


Figure 1.2.1: Clyde Cowan (right) and Frederick Reynes (left) after demonstrating the existence of the neutrino particle

Since then, many studies over the neutrinos have been done to discover its properties:

- **Neutrino flavor:** the discovery in 1962 of another type of neutrino called the muon neutrino and a third type called the tau neutrino in 1975.
- **Solar neutrino problem discovery:** The Standard Solar Model predicts a number of neutrinos arriving from the sun. However, as now we are able to measure the flux of electron neutrinos arriving from the sun, it has been discovered that only a third or a half of the predicted number arrive.
- **Oscillations:** investigating that neutrinos can change of flavor through time.
- **Cosmic neutrinos:** same as Sun neutrinos, it is expected for neutrinos to come from many other parts of the universe. The source of these neutrinos is theorized to be or the Big Bang or Supernova events.

These phenomena caused by neutrinos coming from the space is studied by neutrino astronomy. This science is still in its early stage due to the difficulty of detecting neutrinos as they don't interact with almost anything and when they do so, it is in a very weakly manner. In the next chapter, the setup to detect a decent amount of neutrinos will be explained. Precisely, this TFM has been developed in a project that aims to improve the neutrino detection capabilities available as of now with the most modern detectors.

Chapter 3

Hyper-Kamiokande (HKK)

3.1 The predecessor

The physics apparatus used to study neutrinos is referred to as a *neutrino detector*, built to be isolated from any other influence like cosmic rays or background radiation [7].

These neutrino detectors are huge structures that work following a neutrino detection technique of the existent ones let it be radiochemical methods, radio detectors or scintillators (like in the Cowan-Reines neutrino experiment) such as Cherenkov light detectors.

The experiment that gives name to this chapter is based on the latter: the Cherenkov light detection. This detectors are huge water-filled tanks enriched with deuterium and gadolinium. This medium is ideal for neutrino interaction as the interaction of one of these subatomic particles with the electrons or nuclei of water can produce a charged particle faster than the speed of light in water. This produces a cone of light called *Cherenkov light* and can be defined as the equivalent of light to a sonic boom in acoustic waves.

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

This PMT detects the Cherenkov light produced by the neutrino interaction. By sensing the pattern of light many information of the neutrino can be inferred, such as direction, energy and sometimes the flavor information of the incident neutrino. figure 1.3.1.

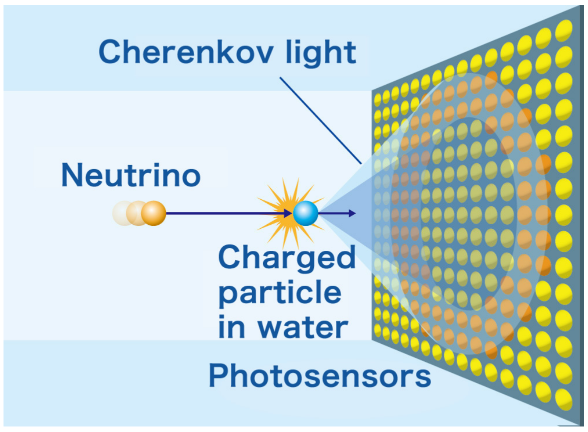


Figure 1.3.1: Cherenkov light phenomenon detected by a PMT [8]

These detections are very rare because the probability of a neutrino interacting with matter is very low so the bigger the water tank and the larger the number of PMT, the more interactions will be detected in the same amount of time.

The largest neutrino detector operating nowadays is the Super-Kamiokande (SKK). Kamiokande is a combination of several words: KAMIOKA Neutrino Detection Experiment. It is located under Mount Ikeno near the city of Hida in the Gifu Prefecture, Japan [9]. Kamioka is the name of the facility that manages this detector. This detector has undergone up to 4 revisions due to several reasons, such as a cascade failure or the replacement of the 6000 PMT and the upgrade of electronics in the latest iteration: Super-Kamiokande IV. Table 3.1, shows the evolution of the neutrino detector from its first version built in 1996 until now.

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 [10]

This versions have seen no change in the number of PMT or in the percentage of coverage. Changes were most in the line of protection measures as stated before and changing the technology used in the front-end electronics, going from a Analog Timing Modules (ATM) for Inner Detector (ID) and Charge to Time Converter (QTC) for Outer Detector (OD) to a unified technology of improved QTC Based Electronics over Ethernet (QBEE).

3.2 HKK structure

In May 2020, a complete overhaul of the SKK was green-lit. It would consist in a new observatory built from the ground up with the objective of being 10 times larger fiducial mass than its predecessor, making it the largest underground water tank in the world. Comparing the specifications of HKK to the SKK versions of table 3.1, in HKK there are 40,000 PMT which correspond to a 40% photo-cathode coverage, the same as SKK. In HKK, much more PMT are needed to cover all the area of the much bigger tank. The goal of this detector is to provide clean proton decay searches via $p \rightarrow e^{+} + \pi^{0}$ and $p \rightarrow \overline{\nu} + K^{+}$ and observation of anti-neutrinos coming from supernovas.

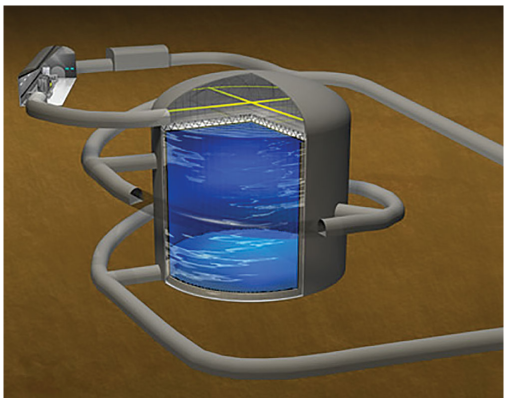


Figure 1.3.2: Hyper-Kamiokande water tank concept sketch. A megaton water tank used for gathering in 10 years data that SKK would take 100 years [8]

The detector design consists of a cylindrical tank with outer dimensions of 60m height × 74m diameter; this is filled with 260,000 metric tons of ultra-pure water to form a *water Cherenkov detector*. This tank would be surrounded by ultrasensitive photodetectors that exhibit 50% higher efficiency than the SKK ones allowing for higher precision in light intensity and detection time. These PMT (Hamatsu R12860) will amplify signatures such as those detected in neutrino interactions [11]. This will enable physicists to measure the direction and the speed of the neutrino going through the detector more accurately. A detailed cross section of the first tank is shown in figure 1.3.3.

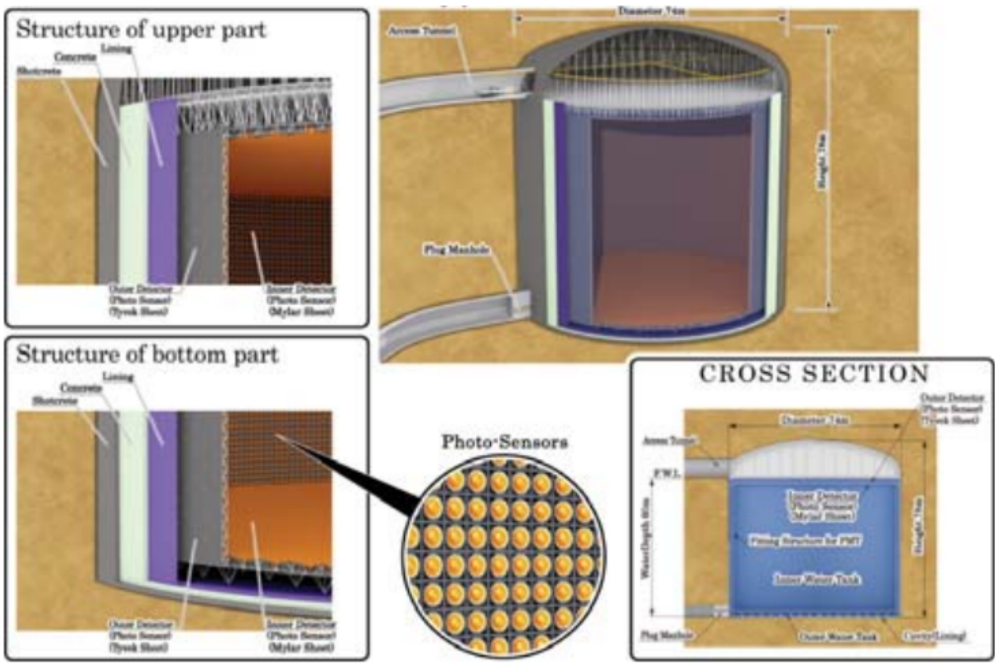


Figure 1.3.3: Hyper-Kamiokande Cross section for the first tank [11]

The detector includes its front-end electronics and a network that connects it with a cluster of computers for data processing. The high-efficiency data acquisition allows for detecting events in the interval of $2\mu s$ in average. The OD remains mostly unchanged in structure from SKK with a layer of 1 to 2m for constraining external background.

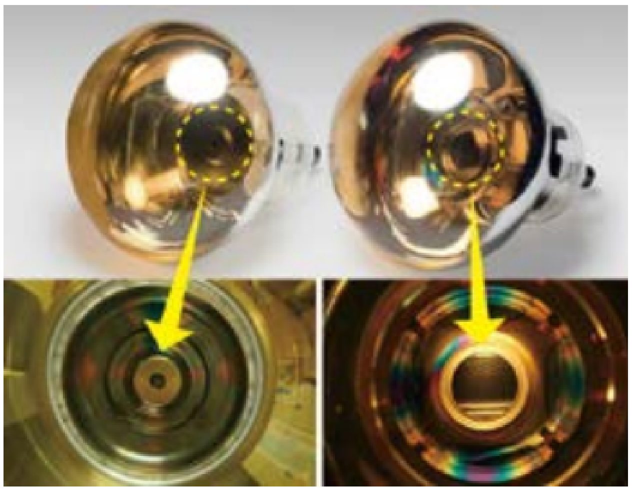


Figure 1.3.4: Ultrasensitive PMT that will be installed inside the HKK [11]

3.3 HKK goals

This huge increase in size will make the smallest units of matter in the Universe possible to study. The matter in question consist of elementary particles called quarks and leptons. For example, one proton, made of three quarks, and one electron, which is a kind of lepton, form a hydrogen atom. The neutrino is a kind of lepton without electric charge and exists in types; electron neutrino, mu neutrino and tau neutrino. The three types of neutrinos mix with each other and can change their type. This phenomenon is called Neutrino oscillation and was discovered by Super-Kamiokande in 1998. The detailed study of neutrino oscillation enables us to reveal the properties of neutrinos. The Standard Model, which describes the elementary particle system, seemed to be completed by the discovery of the Higgs bosom particle. However, the information about neutrino mass and its mixing rate obtained by previous studies have a large difference from those of quarks. It is considered that a more fundamental framework is needed beyond the Standard Model. Neutrino oscillation experiments are expected to be a key to address what the fundamental framework of elementary particles is [8].

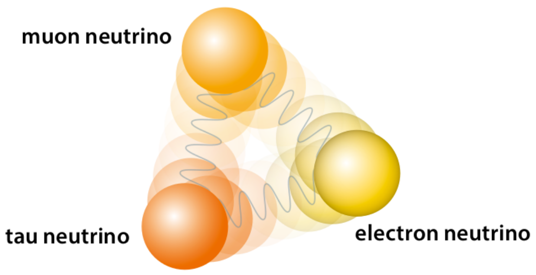


Figure 1.3.5: The existence of three flavors of neutrino and the mix of them cause the neutrino oscillations that make them change their type. [8]

The aim of this project is deepen in the investigation of the neutrino oscillations. The main phenomena to be investigated in the HKK as well as a short explanation of each one of them are shown in figure 1.3.7.

Moreover, this detector doesn't work alone, with just neutrinos coming from outer space as shown in figure 1.3.7, but also with neutrinos shot from the J-PARC accelerator in Tokai, Ibaraki. These neutrinos would travel from one side of Japan island to the opposite, where HKK so that the CP violation can be studied. Charge Parity (CP) violation requires a good amount of neutrino detections. SKK took long to make that amount. With HKK, the amount of detected neutrinos is predicted to increase 30 times, allowing for less time-consuming results. J-PARC neutrino beam will also be improved at the same time that HKK is being built.

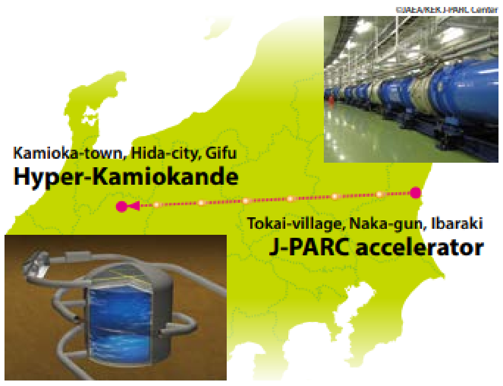

©JAEA/KEK J-PARC Center
Kamioka-town, Hida-city, Gifu
Hyper-Kamiokande
Tokai-village, Naka-gun, Ibaraki
J-PARC accelerator

Figure 1.3.6: Travel of a neutrino from the JPARC accelerator to HKK[8]

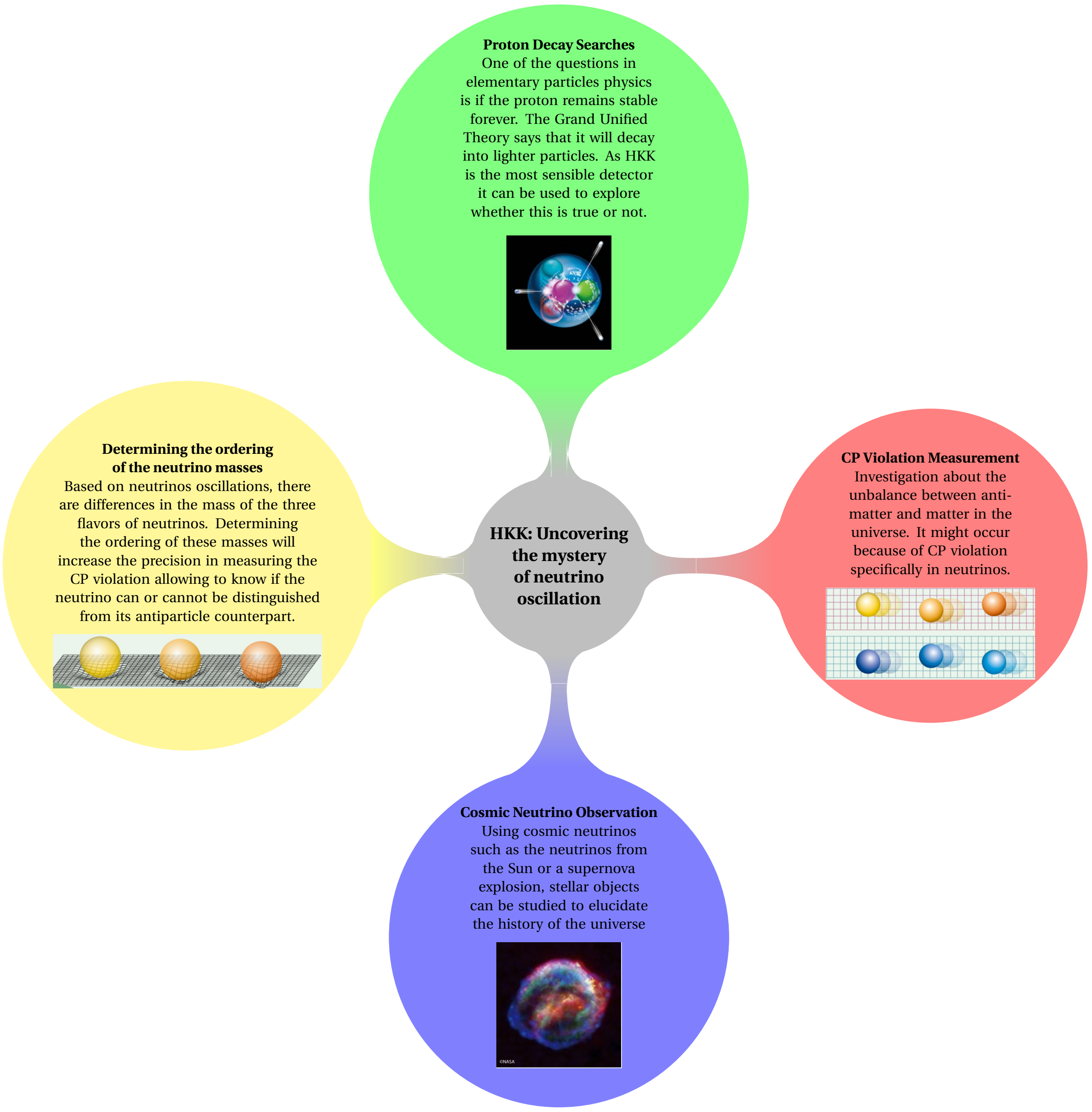

Proton Decay Searches
One of the questions in elementary particles physics is if the proton remains stable forever. The Grand Unified Theory says that it will decay into lighter particles. As HKK is the most sensible detector it can be used to explore whether this is true or not.
Determining the ordering of the neutrino masses
Based on neutrinos oscillations, there are differences in the mass of the three flavors of neutrinos. Determining the ordering of these masses will increase the precision in measuring the CP violation allowing to know if the neutrino can or cannot be distinguished from its antiparticle counterpart.
HKK: Uncovering the mystery of neutrino oscillation
CP Violation Measurement
Investigation about the unbalance between anti-matter and matter in the universe. It might occur because of CP violation specifically in neutrinos.
Cosmic Neutrino Observation
Using cosmic neutrinos such as the neutrinos from the Sun or a supernova explosion, stellar objects can be studied to elucidate the history of the universe
©NASA

Figure 1.3.7: Hyper-Kamiokande neutrino oscillation investigation fields [8]

This objectives are the bleeding edge of elementary particles physics because the HKK will be able to prove these four theories. Going out of the most theoretical point of view, the value that this will have in future technology also needs to be considered. This might seem now that it will stay as just physical theory for *better understanding of our universe*. Nothing further from the reality: as with every new technological advance, first the physical fundamentals of it had to be discovered, proved and measured so that it can be used in mankind's advantage. For instance, like the behaviour of the electron was studied and gave birth to the field of electronics, present in any device in our daily lives, like the photon was studied and then gave birth to the fiber optics and photonics field for blazing fast communications, investigating neutrinos could give birth to a new set of technologies that could be called *neutronics*, which could be used for example for communication in the space as the neutrinos are particles that, as they don't interact with almost any particle and do so very weakly, do not suffer of attenuation or huge losses, which allows long distance communications.

Chapter 4

Project organisational structure

The management structure of the HKK project must take into account the different components of the project (the far detector, the IWCD, the ND280 complex, and entire beam facility), and the differing responsibilities of the two host institutions (The University of Tokyo/NNSO/ICRR for the far detector, and KEK/J-PARC for the rest) and the international collaboration.

This chapter focuses on explaining the roles of the international collaboration in each part of the detector. The project has been divided into several levels depending on the component being worked on. Those levels are referred to as Far Detector (FD) followed by a number. There are in total 7 FD groups:

- **FD1:** Everything relative to the 50cm PMT in the detector. This group has task relative to the testing of the sensors, manufacturing the PMT covers (task done by Donostia lnternational Physics Center (DIPC)) and assembling together everything.
- **FD2:** This group is in charge of other PMT, called multi-Photomultiplier tube (mPMT). These sensors are much smaller than the FD1 ones and have several advantages that help get a better neutrino detection such as weaker sensitivity to Earths magnetic field, increased granularity and directional information with an almost isotropic field of view [12].
- **FD3:** OD photo-detection. HKK has two rings of PMT. This group is in charge of the outer one. From procuring the sensors, to testing them, assembling and cabling the whole unit, installation and the software that analyses the data coming from the PMT
- **FD4:** The front-end electronics group in charge of developing the electronics within the vessel that will gather the data from the PMT and send it to the computer network. This group is divided into three subgroups called FD4.1, FD4.2 and FD4.3 . FD4.1 part will be explained in far more detail in the next chapter. FD4.2 is involved in everything related to the underwater vessel mechanically speaking and FD4.3 is in charge of the module assembly.
- **FD5:** DAQ for receiving data from the DPB and monitoring the behaviour of the detector.
- **FD6:** Detector calibration to ensure that PMT deliver the expected output to a certain event. This includes Light injection process , photogrammetry, precalibration among other processes.
- **FD7:** This group procures the systems used to fill the tank with water. From the room drainage, to the piping system and the flow in the tank.

Apart from the FD , there is also the Far Facilities (FF) group, in charge of the facilities associated with the FD.

An approximate schedule of all the project from January 2022 onward is shown in figure 1.4.1 .

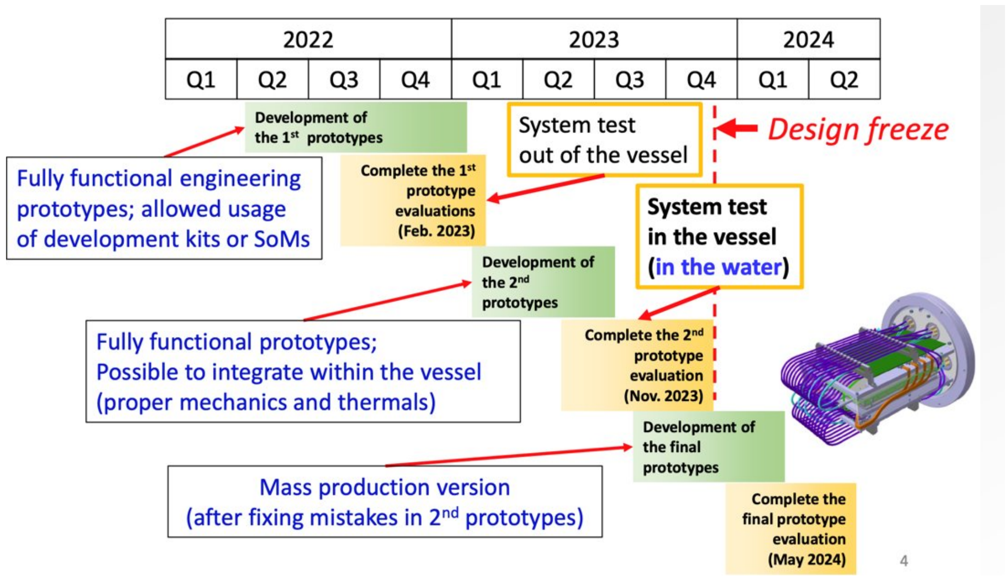

2022
2023
2024
Q1
Q2
Q3
Q4
Q1
Q2
Q3
Q4
Q1
Q2
Development of the 1st prototypes
System test out of the vessel
Design freeze
Fully functional engineering prototypes; allowed usage of development kits or SoMs
Complete the 1st prototype evaluations (Feb. 2023)
System test in the vessel (in the water)
Development of the 2nd prototypes
Fully functional prototypes; Possible to integrate within the vessel (proper mechanics and thermals)
Complete the 2nd prototype evaluation (Nov. 2023)
Development of the final prototypes
Mass production version (after fixing mistakes in 2nd prototypes)
Complete the final prototype evaluation (May 2024)
4

Figure 1.4.1: Time schedule for the design of the vessel, the front-end electronics and the PMT

The Polythecnic University of Valencia (UPV) belongs to FD4 group,involved in the development of the front-end electronics, explained in the next chapter. This front-end has a lot of elements so one university or institute doesn't design all of it. Instead, this is a collaboration between the following entities:

- **Italy:** making the digitizer board. The entity responsible of the design and test is the Istituto Nazionale di Fisica Nucleare (Rome, Italy).
- **France:** In charge of the timing and synchronization module that will synchronize the clocks and data capture of all the vessels of the tank for accurate timestamps in the experiment. The entity responsible is the Institut national de physique nucléaire et de physique des particules (In2p3).
- **Poland:** working to support the UPV with the DPB.
- **Spain:** it is involved in several stages of the project, but FD4 is the responsibility of the Polythecnic University of Valencia (UPV)
- **Switzerland:** Procurement of the High Voltage Module (HV) and Low Voltage Module (LV). Also the European Organization for Nuclear Research (CERN) also collaborates in this project by providing space in its laboratories for testing the prototypes once they are ready in Geneva.
- **Great Britain:** in charge of developing the DAQ software. It intervenes in two groups: FD3 where they are developing a digitizer board for the OD, and in the FD4 group, because the part of the framework coded by this team must be implemented in the DPB. One of the entities in charge is the Warwick University (Coventry, UK).

Chapter 5

FD4 and the DPB front-end

This TFM is developed inside the FD4 group, in charge of the electronics front-end as stated in the previous chapter. A front-end is defined in electronics as the group of components of a system that serve as the interface that gathers the data from a given sensor and transforms so that it is readable by the processing system it must be sent to. The electronics front-end of the HKK is formed by the following components, each one of them in a separate PCB:

- **Digitizer board:** this board is directly connected to 12 PMT, and gathers the data from all of them and, as it names suggests, digitizes the data to be sent to a server. This board has 12 channels and a FPGA running a firmware that gathers the data from those 12 channels and sends it through a MiniSAS (figure 1.5.2) connector to the DPB. A drawing of the board of a digitizer prototype as of January 2023 can be seen in figure 1.5.1.

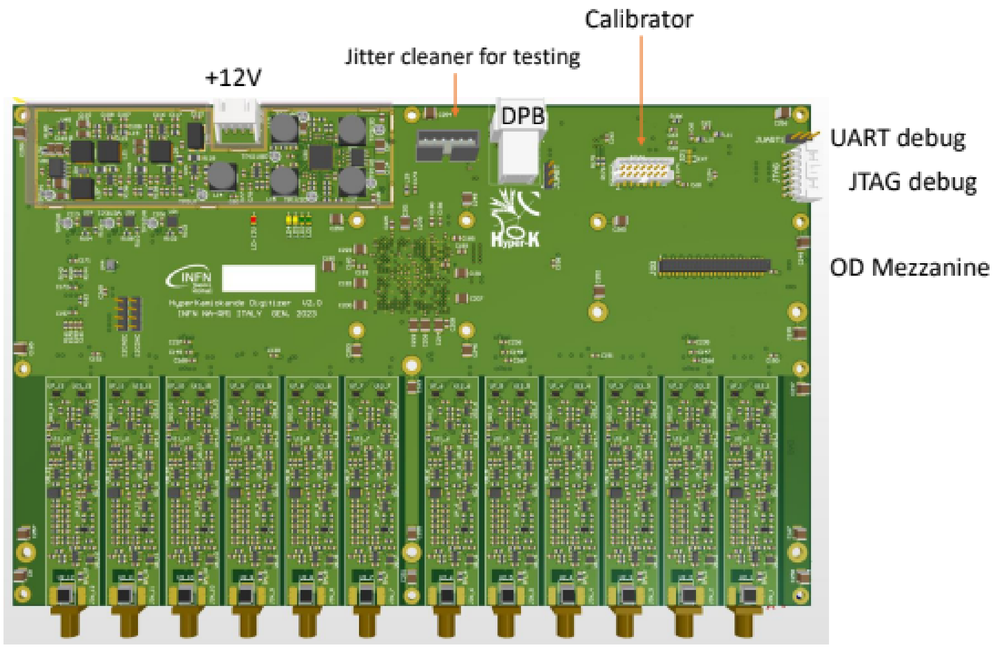


Figure 1.5.1: Digitizer board. Prototype from January 2023

Figure 1.5.2: MiniSAS connectors to transfer data from the Digitizer to the DPB

- **Timing and synchronization module:** This module uses a custom timing protocol, different from TCP/IP working over fiber optics that sends a reference clock so that the Phase Locked Loop (PLL) of the rest of the boards in the vessel can hook up their clocks and be synchronized. Each board, the DPB, the Digitizer and the Voltage supply modules need to be hooked to this clock so that all sensors' timestamps and sensor data are consistent with each other system and avoiding errors like having data coming from the future due to a bad synchronization. Figure 1.5.3 shows the clock distribution scheme from the timing module to the rest of the project.

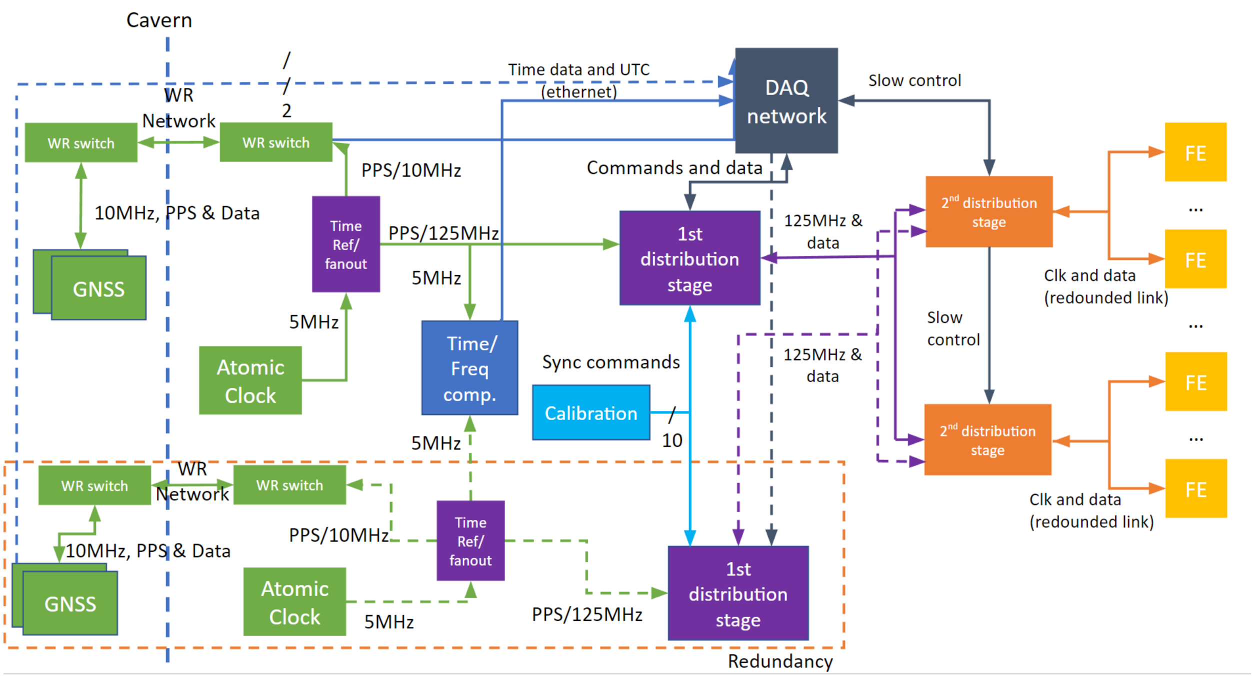


Figure 1.5.3: HK Clock distribution

- **Data Processing Board (DPB):** This board is the hub of the rest of the modules. Every board described previously end up connected to this module and it is in charge of gathering the PMT data connected to two digitizers and the slow control data from them like temperature, power, voltage and current sensors among others and package it all in standard TCP/IP frames and send them through a data channel to the DAQ. The physical layer will be 1Gbps BaseX optical fiber link.

- **Data Adquisition Module (DAQ):** This module is the frontier between FD4 and the data-center where all the data gets processed. It consists of an optical switching network that interconnects all the DPB, gathers the TCP/IP frames and sends them to the servers where they are stored for doing the corresponding computations that scientists driving the experiment demand. The DAQ side would be then standard x86 servers running an application that collects the data from the switching network. This data gets sent from the DPB, which run in their ARM CPUs a complementary application that packages the frame using the same library than the DAQ side. This library is called ZeroMQ and its installation will be detailed in following sections.

- **Low Voltage Module (LV):** receives 2A, 48V from the outside. Maximum current consumption is 3A, though average power dissipation should not exceed 100 W. Input voltage and current must be monitored. The power supply to each electronics module can be switched on/off independently from the slow controls. Output voltages and currents must be monitored. Digitizer modules need only a +12V supply.

- **High Voltage Module (HV):** must provide typ. 2 kV, max. 2.6 kV to PMTs. It has an RS-485 or RS-232 interface to the Slow Controls. It is probably the largest and thickest module in the FE box. There will be one design from CAEN and a second one from iSeg for testing.

Both the HV and LV modules are already manufactured by CAEN, with their corresponding datasheets. They also provide a RS485 port for slow control monitoring.

Figure 1.5.4 shows a summary of how all the enumerated modules are connected between them. The data links in figure 1.5.4 are labeled with their corresponding speed. For communicating between the digitizers and the DPB a link of 2.5 Gb/s will be used. However, for sending the data from the DPB to the DAQ 1Gb/s speed is used. This means that the DPB should have a memory buffer where it stores the data as it arrives because it cannot send it as fast as it arrives. Regarding the *timing link*, for simplicity in the DPB it is the same speed as the link with the DAQ referred to as the *data link*. For the digitizer, it is 125 Mb/s. Each one of the links coming from the DPB are repeated for redundancy so that it one fails it can fallback to the other. The reason for this can also be seen in figure 1.5.4.

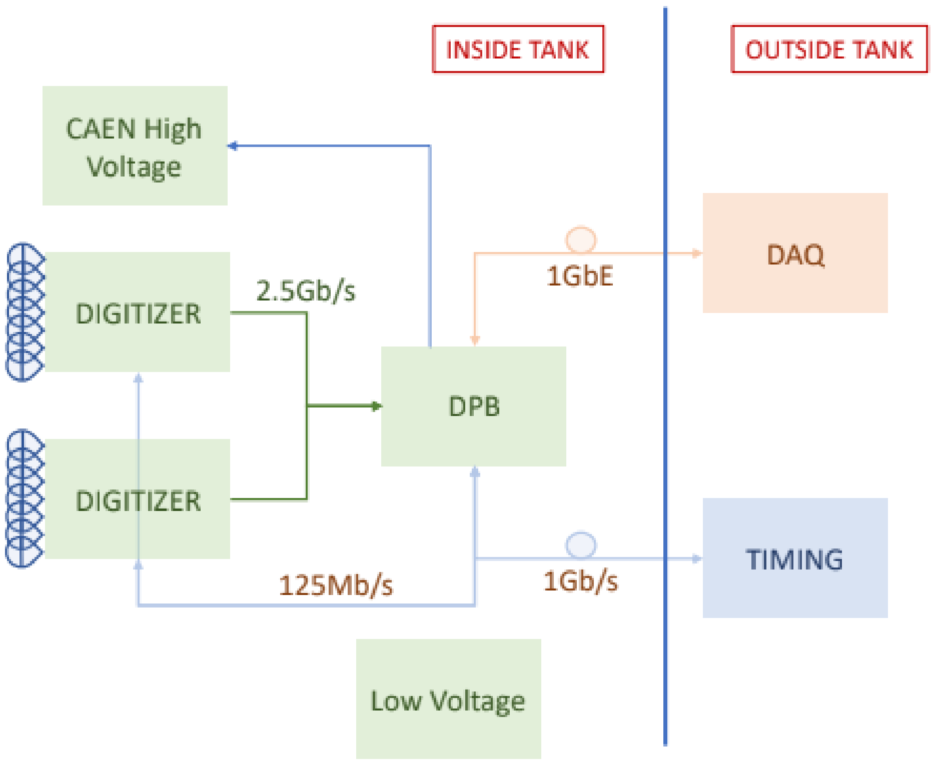


Figure 1.5.4: Block diagram of the functional connections between the elements of the electronic front-end

It is very important to note that there is a lot of the electronic system inside the water tank. This is one of the most important changes going from SKK to HKK. The reason for arranging photodetectors and electronics inside the pressure resistant vessel, as shown in figure 1.5.5, are the increment in granularity of the measurement, and thus enhanced event reconstruction, in particular for multi-ring events, that is, events that will stimulate more than one ring of PMT around the water tank.

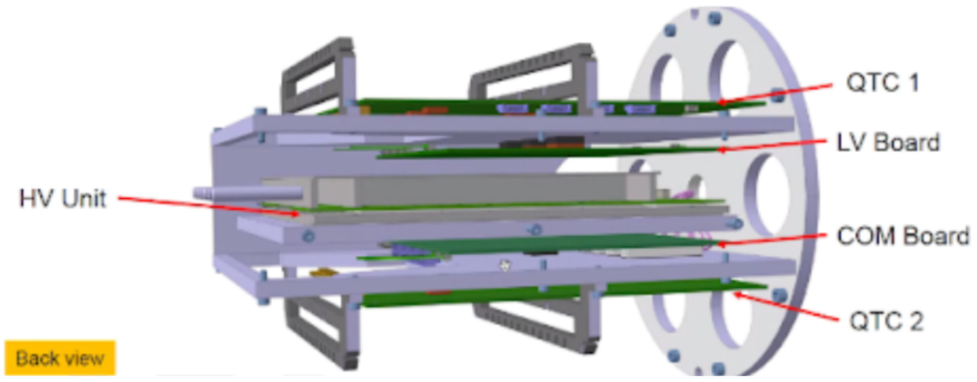


Figure 1.5.5: 3D sketch of one vessel that wild hold 24 PMTs, 2 digitizers (called QTC in the figure) and a DPB, (called COM in the figure) board together with their corresponding HV and LV modules

5.1 Redundancy and data merging: the UPV tasks in this project

The Polythecnic University of Valencia (UPV) is in charge of one of the parts of the electronics front-end: the DPB. This means that the tasks that the university must perform are directly related to the specs that the DPB board must fulfill together with a commitment with helping in the project as a whole as signed in the Memorandum of Understanding (MoU) [13] between the Ministerio de Ciencia e Innovación (MCIN) and the University of Tokio:

- MCIN is committed to participate significantly in the construction and exploitation of the Hyper-Kamiokande experiment, by encouraging and supporting in a proportionate manner, the involvement of the Spanish Institutes. The specific contributions by the Spanish Institutes will be defined in an independent document, separated from this MoU. *The UPV is in charge of the DPB but there are other Spanish universities in charge of other tasks in different FD groups.*
- MCIN will make their best effort to promote the funding to ensure the participation and the successful collaboration of the Spanish institutes within the HyperKamiokande experiment, in the whole experiment life. A list of the potential collaboration from the Spanish Institutes in the construction of the experiment is listed in Annex II, subject to the corresponding budget availability.
- MCIN will encourage the Spanish Institutes to join the operational phase of the HyperKamiokande experiment and to participate in the scientific exploitation of the data acquired. Any personnel, students and property of the Spanish Institutes located at the Hosts for the purposes of this MoU shall comply with the relevant regulations in force at the host site.

Regarding the specifications of the DPB, there are two crucial aspects as stated in the vessel modules description and all the specs must be defined around them to ensure the correct operation of the detector. This two aspects read as follows:

- **Gathering Data:** The Data Processing Board (DPB) addresses several functional blocks in the ID&OD front-end electronics box, mainly *Systems control and data transmission*, Optical I/F to the DAQ, and *Data buffering and handling*. This means that the DPB acts as a hub for all data to arrive in different format and provide a common frame format to package all this data to be sent to a server so that this server can easily decode the data from the experiment. This means that the DPB must be running an application to perform such tasks. The most convenient and powerful way is using an embedded Operative System (OS) running on a System on Chip (SoC). This SoC must be chosen so that it adapts to our needs which can be summarised as enough compute power to manage the TCP/IP stack, enough transceivers to hold all the optical fiber links connection, and enough memory for buffering the data as it arrives from the digitizers.

- **Reliability:** In our baseline design, the front-end electronic modules are installed in the water to avoid using long analog cables from PMT to the electronics located outside of the tank. Also, this makes the number of cables in the water to the outside of the tank much smaller, make the PMT support structure lighter and the construction of the detector much easier. The sheath of the cable is known to be one of the sources of Rn and thus, minimizing the length of the cable is expected to help in reducing the amount of Rn in the water. We can not access the electronics once the water is filled and thus, we can not fix components if they fail. Therefore, we require the total failure rate of the system under the water to be less than 1%/year. Of course, this solution requires many technical challenges in developing the system, such as water-tightness and durability. Also, it is not possible to upgrade the electronics until the tank is drained, which is not expected to happen for more than 10 years once the operation is started.

  From the DPB side there are many things that can be done to ensure reliability and that the electronics will not need to be change in 10 years time:

  - **Redundant physical links**: by adding more ports to the DPB that perform the same function, redundancy is achieved. This means that if one physical link fails, the data can still be sent through the other link, the redundant one. This redundant link is turned off until needed.
  - **Redundant memory:** data buffered and configuration is stored in the main memory of the system, more commonly known as Random Access Memory (RAM) due to the technology the main memory usually uses on PC. Memory can have soft and hard failures overtime. Soft failures can be solved just by writing again in the same cell, but hard failures mean that that cell is lost and nothing can be stored in it again. Memories have their own hardware-managed redundancy, called Error Correction Code (ECC), which consists of putting more memory chips, going over the advertised capability and keeping copies of the stored data in a transparent way to the user.
  - **Redundant bootup:** Using a embedded OS means that it needs to boot up from somewhere. That place must be an internal memory which has very good reliability. In the development of this TFM, it has been agreed upon the fact that Quad Serial Peripheral Interface (QSPI) Flash memory are more reliable than External Multi Media Card (eMMC) because of how both standards are thought. So it is a good idea to boot up from the QSPI memory and even better, having a lightweight operating system which allows a QSPI flash, with a lower density by design than eMMC, to hold several copies of the OS image. This means that if there is a hard failure in one of the QSPI cells, a SHA256 CRC will detect that failure and jump to the next image by adding an offset to the memory reading operation.
  - **Updating firmware:** bugs arise in the development of any software. The more iterations, the better the produced software is if the correct guidelines are followed. However, there is no test that makes a software flawless and errors will arise during normal operation. So, there needs to be a way to update the firmware once everything is installed in the vessel. This firmware can be provided through the data links and updated in the QSPI memory or even doing a remote Preboot eXecution Environment (PXE) boot so that the image and the file system is loaded from a remote server as a way of fallback if every local memory in the DPB fails so at least a debug operation can be performed to see where are the errors.

Part II

Platform. Xilinx IDE and Petalinux SDK

Chapter 1

Hardware selection

This chapter will give the justification to the hardware being used for initial tests and embedded OS development, as well as the features contained in the chosen architecture and how to communicate with the chip from a PC.

The modern approach to designing digital electronics has evolved a lot in the past years. As of this project, a high performance system needs to be assembled for high speed communication, data saving and configuration of ad-hoc boards. Designing such a system starting from zero would be way out of scope for this thesis and for the HK project in general. Right now, the procedure that provides the most optimal system for a given task in the least time possible is Field Programmable Gate Array (FPGA) design workflow.

FPGA are not unknown to electronic engineers. They are semiconductor devices that are based around a matrix of Configurable Logic Blocks (CLB) connected via programmable interconnects which enables them to be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGA from Application Specific Integrated Circuit (ASIC), which are custom manufactured for specific design tasks. They are an evolution of One Time Programmable (OTP) with SRAM that can be programmed several times. [14].

CLB become different logic functions depending on how FPGA is programmed. The way to program FPGA nowadays is loading a bitstream: a group of bits that establish the logical functions of the CLB, in which physical CLB they are implemented (known as placing) and how they are interconnected (known as routing) to perform as a whole the logical function indicated by the user.

1.1 ZCU102 evaluation board

Among the most popular FPGA manufacturers, Xilinx is the biggest one. Its popularity has lead to its architecture being chosen by the HKK Committee to be used in the DPB.

Diving deeper into what Xilinx offers for Embedded Applications platform development, its flagship, the Zynq UltraScale+ Multi Processing System on Chip (MPSoC) can be found. This behemoth chip provides 64-bit processor scalability while combining real-time control with soft and hard engines for graphics, video, waveform, and packet processing. Built on a common real-time processor and programmable logic equipped platform, three distinct variants include dual application processor (CG) devices, quad application processor and GPU (EG) devices, and video codec (EV) devices, creating unlimited possibilities for any embedded application [15].

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 2.1.2 depicts a simplified block diagram with all the parts that form this chip. At first glance, there are two main parts [16]:

- **Programmable Logic (PL):** the classic FPGA part. It contains all the blocks needed for instantiating your own hardware described through Hardware Description Language (HDL):
  - Storage and Signal Processing: FPGA have small memories integrated in them that can be used for generating memory hardware blocks for storing data. Xilinx calls their own FPGA integrated memories BlockRAM and its evolution of much higher density UltraRAM [17]. Then there are also DSP blocks, with special architectures that enable the instantiation of modules such as Multiply and Accumulate Units for filtering in a much more efficient way.
  - System Logic Cells: the basic logic units used for implementing logical functions. They are formed by a 6 input Look Up Table (LUT) together with a multiplexer and two flip flops in the case of UltraScale+ Architecture. Figure 2.1.1 depicts a simplified schematic of a Logic Cell:

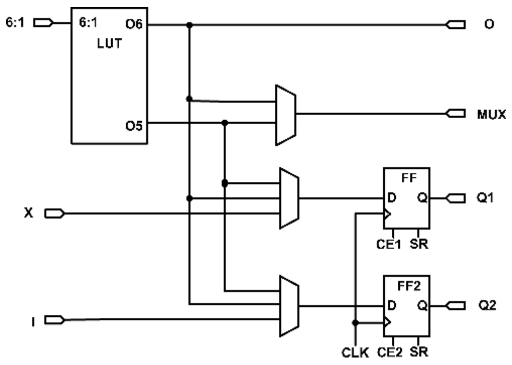


Figure 2.1.1: Zynq UltraScale+^{TM} Logic Cell [18]

  - General Purpose I/O. The input/output pins of the FPGA divided into two subgroups: High Performance (HP) for faster speeds and High Density (HD) with lower speeds but more density, which allows for a higher number of low speed communication buses.
  - High Speed Connectivity: The highest speeds buses are included in this category, from the Multi Gigabit Transceivers (MGT) such as Gigabit Transceivers Type H (GTH) and Gigabit Transceivers Type Y (GTY) to the 100G Ethernet Media Access Controller (EMAC) and Peripheral Component Interconnect Express (PCIe), which allow for tenths of Gigabits Per Second transfer speeds.

- **Processing System (PS):** FPGA usually contained just CLB but, as they were often used together with a hard CPU for accelerating certain tasks, the PS concept was born, with silicon space dedicated to specific functional units, like integrating ASIC next to the FPGA and communicating both worlds with a high speed interconnect fabric. The blocks in this part of the UltraScale+ MPSoC architecture are as follows:
  - Application Processing Unit (APU): The main cores of the PS in charge of running the embedded OS of choice.
  - Real Time Processing Unit (RPU): Two complementary cores that are specially dedicated to executing Real Time applications, that is, functions that need to be executed in a deterministic time.
  - Memory: The main memory of the CPU, equivalent to the RAM memory in a personal computer.
  - Platform Management Unit (PMU): for managing the system as a whole from a scheduling and power delivery point of view.

– Graphics Processing Unit (GPU): for geometry and image processing tasks. UltraScale+ delivers video encoding/decoding capabilities in embedded application thanks to this block.
– Voltage and Temperature Monitoring: crucial for HKK as these magnitudes need to be measured to ensure that the DPB operates in the optimal conditions in order to optimize its lifespan.
– Connectivity ranging from the most general one like USB, Controller Area Network (CAN), Ethernet to high speed like Display Ports, USB 3.0, Serial Advanced Technology Attachment (SATA) and HDMI.

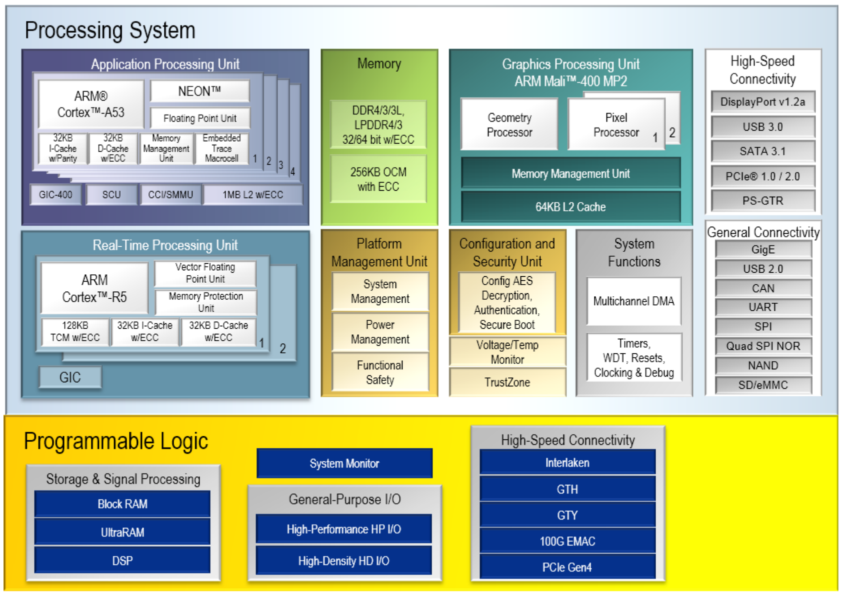


Figure 2.1.2: Zynq UltraScale+^{TM} Block Diagram [15]

This high performance system is sold for development in the form of Evaluation Boards. The one being used in this project is the ZCU102, an evaluation board sporting a Zynq UltraScale+ MPSoC optimized for quick application prototyping using the XCZU9EG-2FFVB1156E device. The ZU9EG contains many useful processor system (PS) hard block peripherals exposed through the Multi-use I/O (MIO) interface and a variety of FPGA PL, high-density (HD) and high-performance (HP) banks. Table 1.1 lists a brief summary of the resources available within the ZU9EG.

Feature	Resource Count
HD Banks	5 banks, 120 pins
HP Banks	4 banks, 208 pins
MIO Banks	3 banks, 78 pins
PS-Side GTR 6Gb/s	4 PS-GTR
PL-side GTH 16.3 Gb/s	24 GTHs
Logic Cells	599150
CLB flip flops	548160
Max distributed RAM	8.8 Mb
Total Block RAM	32.1 Mb
DSP Slices	2520

Table 1.1: ZCU9EG Features and Resources [19]

A picture of the board together with labels of its different components can be seen in figure 2.1.3.

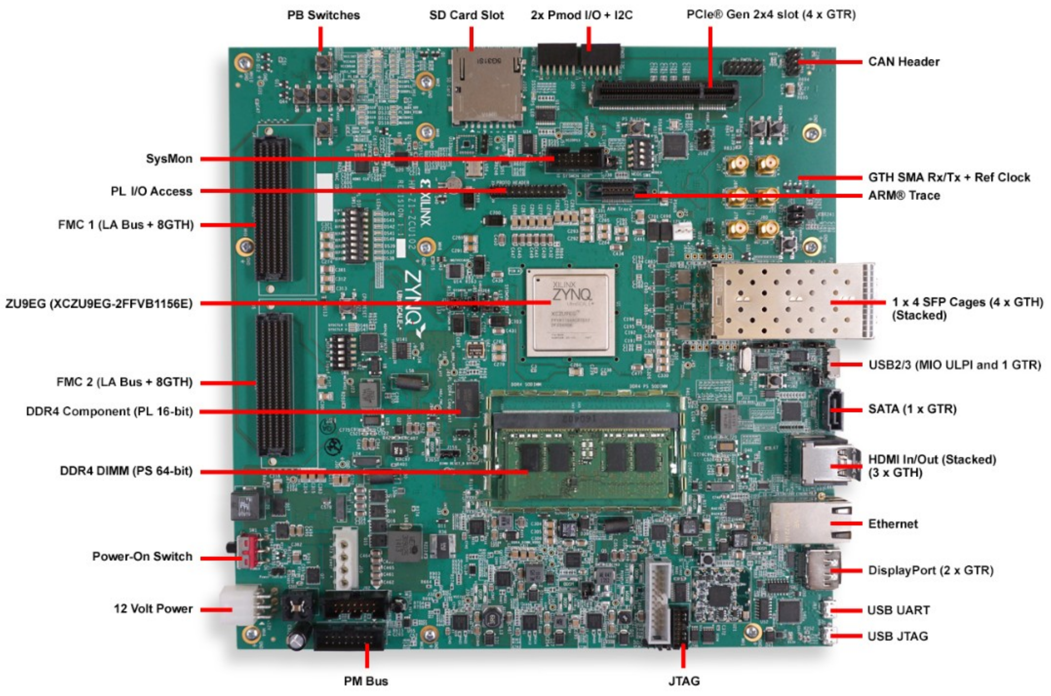


Figure 2.1.3: ZCU102 board features. All the I/O included in the board is depicted together with the FPGA model.

This board includes several of the features needed for the development of the different tasks:

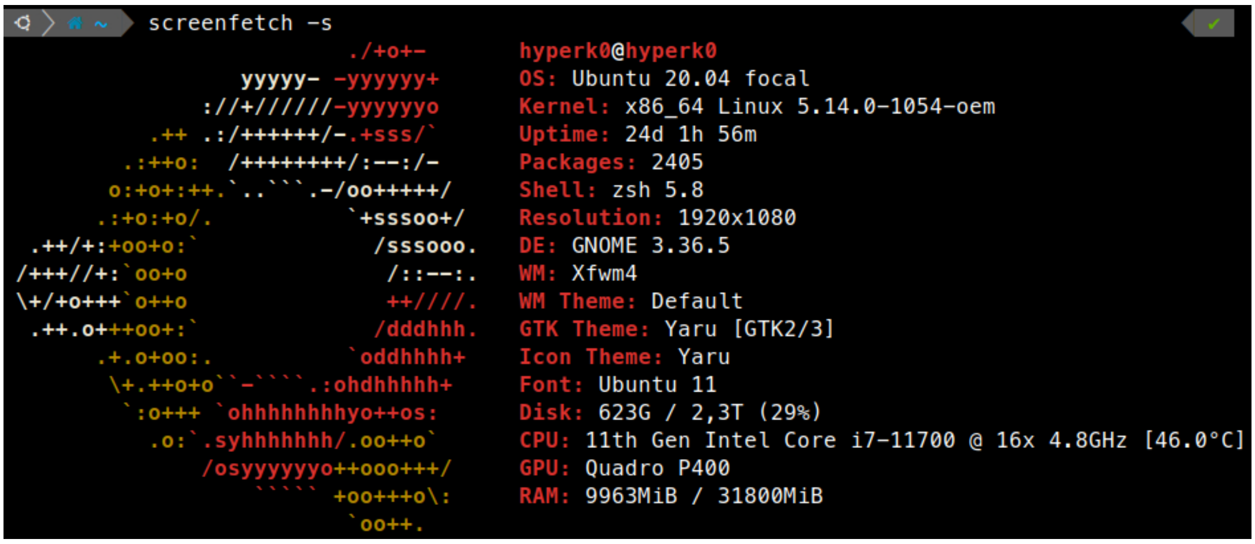
- **Enough memory for redundancy:** with SD Card slot and a 128 MB QSPI flash a good number of copies of the OS image can be stored.
- **SFP Cage:** essential for plugging Small Form Factor Hot Pluggable (SFP) modules. The 4x SFP Cage will allow to test redundant links, together with the RJ45 already included in the board.
- **Powerful CPU:** Thanks to the quad core ARM53 Cortex cores, the TCP/IP stack that needs to be run to send data to the DAQ can be run without issues.
- **FPGA with the most CLB**: so that debugging of the different cores forming the system can be done without worrying about running out of space and do a correct estimation of how much resources will it take to build the hardware in the final prototype.
- **On Chip sensors:** to monitor the temperature, voltage and other relevant magnitudes to ensure that the board is running within the expected conditions.
- **Pmod I/O:** for connecting external sensors to simulate the slow control sensors that will go into the final prototype.

1.2 Host PC for development

This Evaluation Board (EvB) is fully leveraged when used together with a powerful PC that has the correct tools. Thus, a dedicated PC for programming, debugging both hardware and software used in the EvB is the ideal partner for it. For this project, the Dell Precision 3650 Workstation PC has been used.

1.2.1 Hardware specifications

The Dell Precision 3650 Workstation can be configured in different ways. For this project, as this is not needed for serving multiple FPGAs, the specifications chosen among those available in the data-sheet [20] are shown in figure 2.1.4 :



~~~
screenfetch -s
                         ./+o+-       hyperk0@hyperk0
                 yyyyy- -yyyyyy+      OS: Ubuntu 20.04 focal
              ://+//////-yyyyyyo      Kernel: x86_64 Linux 5.14.0-1054-oem
          .++ .:/++++++/-.+sss/`      Uptime: 24d 1h 56m
        .:++o:  /++++++++/:--:/-      Packages: 2405
       o:+o+:++.`..```.-/oo+++++/     Shell: zsh 5.8
      .:+o:+o/.          `+sssoo+/    Resolution: 1920x1080
 .++/+:+oo+o:`             /sssooo.   DE: GNOME 3.36.5
/+++//+:`oo+o               /::--:.   WM: Xfwm4
\+/+o+++`o++o               ++////.   WM Theme: Default
 .++.o+++oo+:`             /dddhhh.   GTK Theme: Yaru [GTK2/3]
      .+.o+oo:.          `oddhhhh+    Icon Theme: Yaru
       \+.++o+o``-````.:ohdhhhhh+     Font: Ubuntu 11
        `:o+++ `ohhhhhhhhyo++os:      Disk: 623G / 2,3T (29%)
          .o:`.syhhhhhhh/.oo++o`      CPU: 11th Gen Intel Core i7-11700 @ 16x 4.8GHz [46.0°C]
              /osyyyyyyo++ooo+++/     GPU: Quadro P400
                  ````` +oo+++o\:     RAM: 9963MiB / 31800MiB
                         `oo++.
~~~

Figure 2.1.4: Workstation technical specifications listed through the command `screenfetch -s`

The main hardware highlights would be the 11th gen Intel Core i/ 11700 CPU together with 32GB RAM, which will allow large compilations and code building to take place without a decrease in system responsiveness. The GPU is a basic Nvidia Quadro P400 just for screen display tasks. Regarding storage, the software gives a sum of the capacity of all disks installed in the PC. The system has three drives:

- **OS drive:** 512 GB NVMe SSD, for storing the Operative System and the development tools
- **Data SSD:** 1 TB NVMe SSD, as the working directory for the projects being developed in this PC. This helps speed up code compilation because OS tasks take bandwidth in a separate drive so all bandwidth in this drive can be dedicated to the specific development software.
- **Data HDD:** 1TB HDD, the slowest drive in the system, it is just for storing backup images of the OS itself or projects backup which are not being worked on.

1.2.2 Expansion cards

This workstation PC has support for more I/O through the installation of PCIe expansion cards that add more ports to the rear part of the computer. The Dell computer itself only provides USB and one RJ45 Ethernet port which is enough for the peripherals normally connected to a computer. However, working with the ZCU102 requires more I/O so that its functioning can be properly assessed. Thus, the following PCIe cards were included in the PC:

- **DGE-528T Copper Gigabit PCI Card [21]:** Contains one Gigabit RJ45 Ethernet port. This extra Ethernet port is required for giving the ZCU102 a network connection directly to the computer. This network will be configured in a separated local subnet from the UPV network as a point to point network to the board.
- **Startech PEX1000SFP2 [22]:** As it was commented in the introduction about the DPB, the system will have several 1000BASEX SFP, which means that the final prototype will be communicating to the internal network of the detector through several Fiber Optics links. Only one link is active while the others serve as fallback in case the active one fails. Due to this constraint, one SFP port in the computer is not enough, thus the need of upgrading the network card to this one.

1.2.3 Software configuration

Regarding the software used for this project, Xilinx provides a Unified SDK called *Vitis Unified Software Platform* that allows to install the basic tools for hardware development and software deployment on that same hardware. This Platform is formed by three main pieces of software:

- **Vivado:** Delivers a SoC-strength, IP-centric and system-centric development environment that has been built from the ground up to address the productivity bottlenecks in system-level integration and implementation. The equivalent of this software in Intel FPGA counterpart would be Quartus, both are in charge of acting as an Integrated Development Environment (IDE) for, departing from a blank project where the development platform is chosen, act as a tool to develop your own hardware based on the constraints of your development platform. It is both easy to use for beginners and with a lot of functionalities for experts. For instance, it can be used both in a Graphical User Interface or directly through a command line using its own language called *tcl* or Tool Command Language.
- **Vitis HLS:** High Level Synthesis (HLS) is an automated design process that takes an abstract behavioral specification of a digital system and generates a register-transfer level structure that realizes the given behavior [23]. This is usually written in C/C++ to be built into a block which can be included in a Vivado project. This block can often be reused in multiple projects, and even potentially be loaded up in Vivado for manual optimization. Figure 2.1.5 shows the flow for hardware development if the starting point were HLS. It aim at coding the algorithm in a high level software language that is then translated into RTL and follows the same flow as HDL.

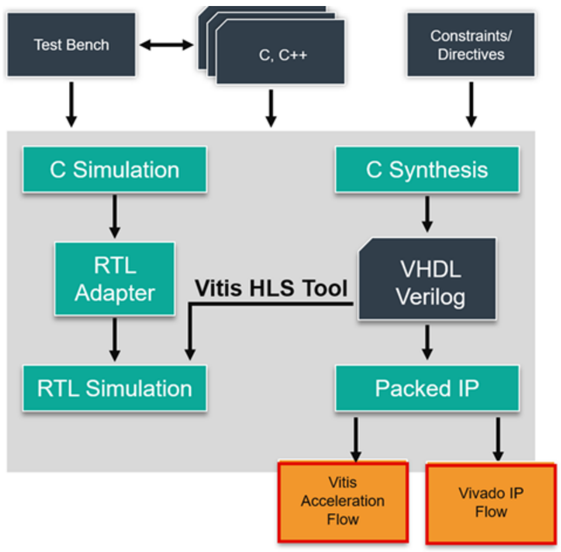


Figure 2.1.5: High Level Synthesis flow [23]

- **Vitis IDE:** Used for the software development part. After creating your hardware in Vivado, if the system contains a CPU, then it can run software. This software can be easily coded in Vitis, an Eclipse-based IDE that allows to code, run and debug the software for the platform of your choice, and optimize the platform itself for the best results.

The version of the tools installed on the development PC are 2020.1 and, as it will be explained in Part III, 2022.2 when upgrading the platform. However, these are not the only tools installed, there are two very important pieces of software left:

- **Embedded OS SDK:** the tools explained above make up for the needs of developing the hardware and the software running on the EvB, but in the software realm there are many types of programs and one of the most complex are Operative System. That is why its SDK is separated from the rest, because they do not follow the traditional software workflow of coding applications directly in code. OS are very complex pieces of software made of several components that provide the software platform for other applications to run on, from libraries to interfaces, they act as an abstraction layer to the hardware to ease the software development.

  Xilinx has support for installing several Embedded OS which range from the simple Real Time Operating System (RTOS) to Embeeded Linux, a lightweight version of the Linux kernel. For this project, Petalinux, an embedded Linux based on the Yocto Open Project will be used due to some advantages that will be explained in the corresponding section. Xilinx provides the PetaLinux Tools SDK as a workflow for configuring, building and packaging your own bootable image.

- **Xilinx DocNav:** acts as an interface to the gigantic documentation datacenter that Xilinx hosts. From DocNav, the user can search for any document published by Xilinx from Overviews of products, Technical Reference Manuals, User Guides to videos showcasing functionalities or courses for their specific hardware. It also offers a function to filter by the version of their tools so that the user has access to the matching documentation of the version of the tools installed on its system.

Chapter 2

Hardware side: Xilinx Vivado Design Suite

Vivado Design Suite is an integrated design environment released by FPGA manufacturer Xilinx in 2012 [24]. Including a highly integrated design environment and a new generation of tools from the system to the IC level, these are based on a shared scalable data model and a common debugging environment.

This suite replaces the ISE Design Suite and all of its point tools to integrate everything in the same environment: the project navigator, synthesis, implementation tools, IP Core generation, Timing constraints editor, Power analyzer and FPGA Editor.

This software suite will be used throughout the project to create the specific hardware needed to fulfill its duty. As such, this chapter presents a basic explanation of the workflow that will be followed for the ZCU102 hardware platform creation, the use of IP integrator for easy and simple hardware instantiation of very complex systems and the hardware debugging through Integrated Logical Analyzer (ILA). As the target for this thesis is more in the software side of learning how to configure the embedded OS, the explanations provided here are very basic, as the ZCU102 already has very complete and well-explained reference designs that contains almost everything that is needed for embedded OS to run and the configurations that will be changed are minimal, like adding a new Ethernet core or changing the configuration of the ZynqMP SoC system to enable controllers for additional I/O.

2.1 Basic Workflow

The workflow in this tool has the target of configuring the FPGA block, disconnected and without any logic function implemented when they are turned on. This process must consider the huge amount of heterogeneous resources such as logic elements, DSP blocks, block RAMs and many more. Configuring these resources to perform a given computational task requires several steps [25]. These steps are represented in Figure 2.2.4 and an example with a counter described in VHDL is provided [26]:

- **Design Entry:** As with any software, the first step is to provide the user-made inputs that will produce the desired outputs once the flow has finished. In this workflow, this is called *Design Entry*, and consists of giving in one of the following ways a description of your hardware to the tool:
  - *Hardware Description Language (HDL):* the most widely used solution of describing digital circuits is using HDL such as VHDL or Verilog. They allow for a higher-level functional description, without detailing the structure at the basic gate level. This significantly reduces the time required to describe complex systems and unless using device-specific libraries, it allows portability between FPGA devices.

```
----------------------------------------------------------------------------
-- ENTITY DECLARATION.
----------------------------------------------------------------------------
  ENTITY counter IS
    PORT(clk     : IN  STD_LOGIC;   -- Main clock
         reset   : IN  STD_LOGIC;   -- reset, active_high
         enable  : IN  STD_LOGIC;   -- enable the next counter
         trigger : OUT STD_LOGIC    -- trigger the next counter
        );
  END ENTITY;

----------------------------------------------------------------------------
-- ARCHITECTURE DECLARATION.
----------------------------------------------------------------------------
  ARCHITECTURE rtl OF counter IS

    -- INTERNAL SIGNALS DECLARATION --
    SIGNAL count     : UNSIGNED(3 DOWNTO 0) := (OTHERS => '0');
    SIGNAL trigger_o : STD_LOGIC := '0';

    -- ATTRIBUTE DECLARATION --
    ATTRIBUTE MARK_DEBUG : STRING;
    ATTRIBUTE MARK_DEBUG OF count : SIGNAL IS "true";

seq_proc: PROCESS (reset, clk)
BEGIN -- for seq_proc
  IF (reset = '1') THEN
    count     <= (OTHERS => '0');
    trigger_o <= '0';
  ELSIF rising_edge(clk) THEN
    IF (enable = '1') THEN
      count <= count + 1;
      IF (count > x"B" AND count <= x"F") THEN
        trigger_o <= '1';
      ELSE
        trigger_o <= '0';
      END IF;
    END IF;
  END IF;
END PROCESS;
```

Schematic editor: it is the tool inside the suite for describing the hardware using interconnected blocks like a hand-drawn schematic on a white board. This allows to use FPGA primitives (instantiated as blocks in this schematic editor) which provides a more optimized result but one must be very careful while managing large projects because the image can get cluttered. One way of avoiding this is to define a hierarchy where new blocks are created. This editor itself is not available in Vivado because of a change in the workflow.

– *IP Cores:* Systems have evolved exponentially in complexity. As such, making a competitive commercial system within a low Time To Market starting from scratch is impracticable. Nowadays, digital system share a lot of hardware blocks like CPU or specific controller for I/O. Hence, the workflow right now almost always starts with a search for blocks called Intellectual Property Cores (IP Cores) that are optimized and fine tuned to perform the desired functionality they were designed for. These cores are made either by Xilinx, Intel FPGA or by other companies specialised in other fields of electronics and the use of them (as will be seen in the IP integrator section) is as simple as a drag and drop, configure and connect input and outputs. For this project, this will be the approach taken and, unless something very specific needs to be done or modified, HDL will be avoided except for the top hierarchy file (which must always be HDL to continue with synthesis).

- **Synthesis:** It consists in the translating the HDL code into a netlist which represents a compact description of the circuit. It is basically a file where the system components, and the interconnection between them and the I/O pins are specified. The format used in Vivado is the Xilinx Netlist Format (XNF) and the result of this stage can be viewed in the Schematic View (figure 2.2.1 as an example) which shows the RTL.

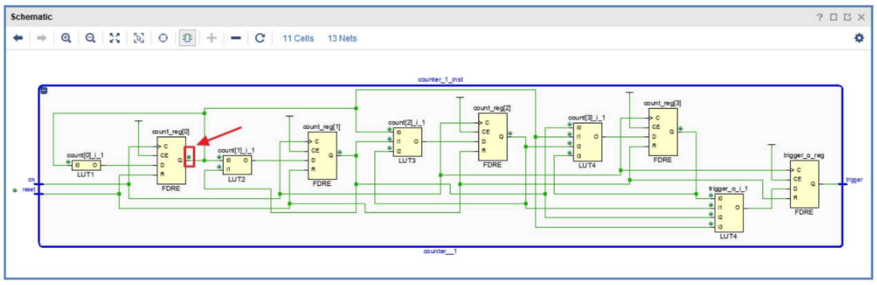
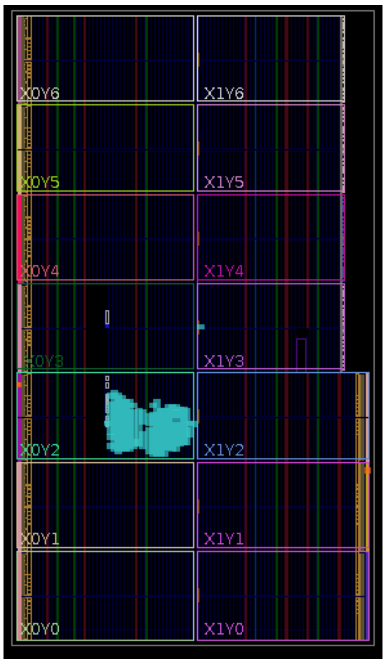
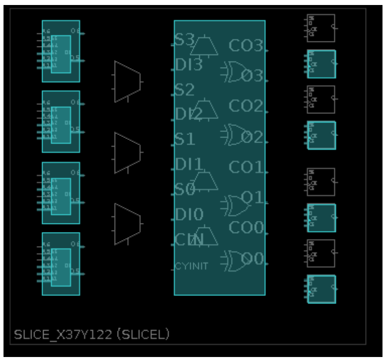


Figure 2.2.1: RTL View post synthesis schematic of a counter [26]

- **Simulation:** once the netlist is available, a simulation to assess the functionality can be run. This simulation is essential because it helps to catch errors before continuing to more time-consuming parts that, if a concept error is not caught before, would need a rework from scratch, after loosing a lot of time. That is why the IP Cores commented before include a test-bench in the same package, so that the user can see for itself how it runs.



(a) Device



(b) Enabled parts in a CLB

Figure 2.2.2: Device View after place and route. The blocks shown here are physically in the FPGA and will perform the function described in HDL. The implementation for two counters takes so much space is due to the instantiation of debug cores that will be explained in the ILA section.

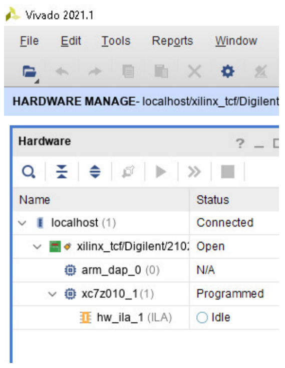
- **Mapping:** process the netlist to assign different parts of it to the available resources of the FPGA. This step involves adapting to the physical resources of the device, optimizations and checking design rules such as the available number of pins.

- **Place and Route:** It selects the modules obtained during mapping and assigns them to specific locations on the device. Once this process is completed, routing consists in interconnecting these blocks using the available routing resources. Both placing and routing are *NP-hard* problems, that is Non-Deterministic Polynomial.

  This means that their time to solve cannot be measured through a polynomial, which usually means that they are exponential or other kind of function that grows much faster than a polynomial. For placing and routing this comes to its heavily single-threaded nature. The placement cannot be made in parallel as the result of binding one mapped part of the netlist to a resource of the device makes it unavailable for the rest of the parts so if two threads were running at the same time they may decide to place two parts of the netlist in the same CLB, resulting in an error. Of course routing must come after placing components. The conclusion from this explanation is that placing and routing are very computationally intensive problems and take long time to solve, which means that, before testing in a FPGA one must be very sure, thanks to simulations, that its code works correctly, at least in the functional level and ideally also at gate level, giving the appropriate timing constraints that can be checked before placing and routing.

- **Timing Analysis:** As the placing and routing is completed, now there is a complete view of how the system will behave from a timing perspective. By knowing the interconnection delays and the propagation times of all the blocks in the FPGA (included in the Vivado database), an accurate timing simulation can be performed. In this project there will not be any timing simulation as the system comes from a reference design already tested by Xilinx. Timing (and other constraints) can be specified through constraints files called Xilinx Design Constraints (XDC) files. An example of code is shown here...

- **Programming:** At the end of placing and routing, a file is generated that contains all the necessary information for configuring the device: logic block configuration and interconnections. This information is stored under the form of *bitstream, .bit*, where each bit indicates the open or close state of a switch on the device. As, most FPGA devices are SRAM based (loose their configuration at power-down), the configuration bitstream is usually loaded into a non-volatile flash-memory and then loaded on power up. This case is not applied when loading through JTAG, where this memory is the computer itself and the .bit needs to be transferred again when the FPGA is powered down. This programming can be done also from the Vivado Hardware Manager (figure 2.2.3), a tool that is separated from the project itself, where given a *.bit*, the Hardware Manager looks for any USB connected boards by opening a local web server (usually in port TCP 5121), lists them (figure 2.2.3) and through a single click programs the FPGA with the desired hardware description.



Name	Status
localhost (1)	Connected
xilinx_tcf/Digilent/210:	Open
arm_dap_0 (0)	N/A
xc7z010_1 (1)	Programmed
hw_ila_1 (ILA)	○ Idle

Figure 2.2.3: Hardware Manager GUI. A list of the available devices is shown

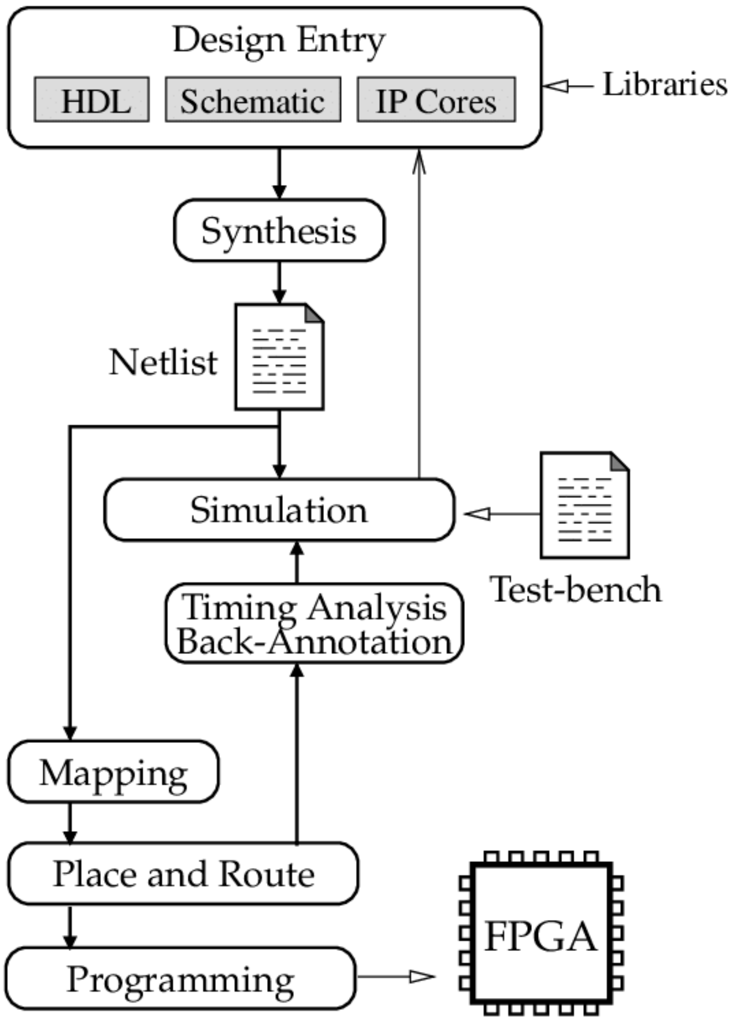
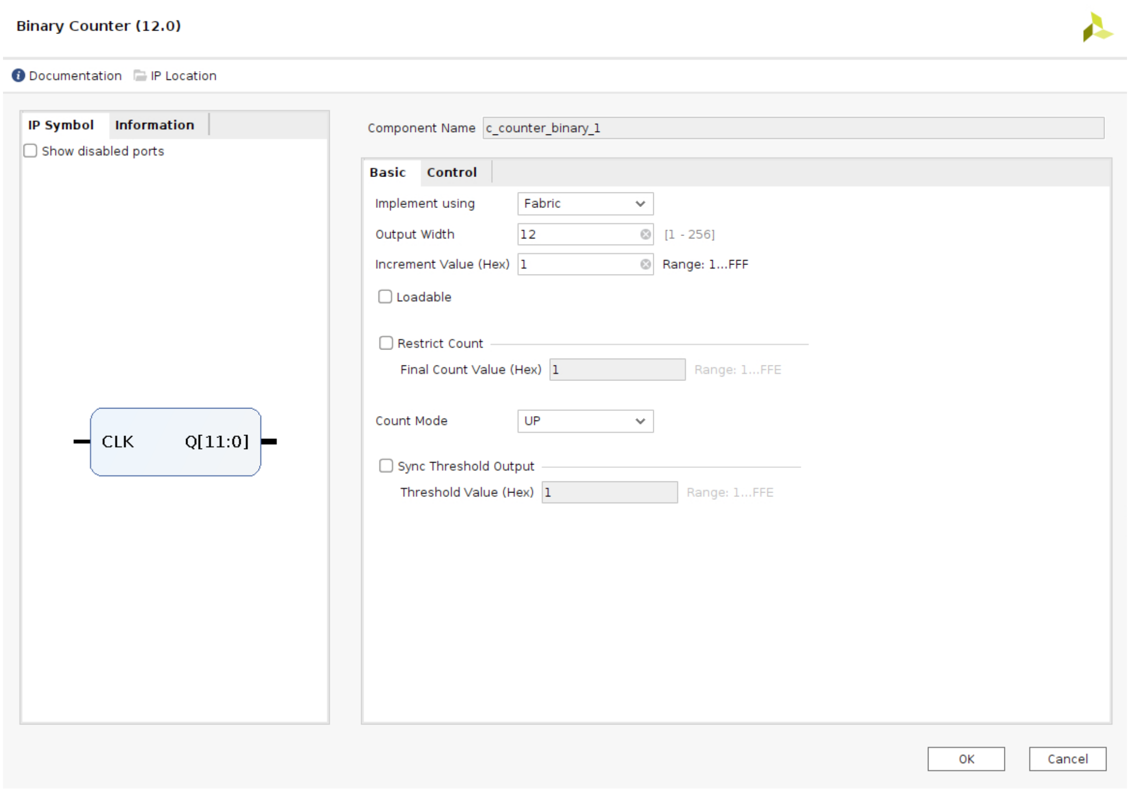


Figure 2.2.4: Classical FPGA design flow [25]

2.2 Instantiating and configuring the PS and other blocks in IP integrator

As FPGAs become larger and more complex, and as design schedules become shorter, use of third-party IP and design reuse is becoming mandatory. Xilinx recognizes the challenges designers face, and to aid designers with design and reuse issues, has created a powerful feature within the Vivado Design Suite called the Vivado IP Integrator. The Vivado IP integrator lets you create complex system designs by instantiating and interconnecting IP from the Vivado IP catalog on a design canvas. You can create designs interactively through the IP integrator canvas GUI or programmatically through a Tcl programming interface. Designs are typically constructed at the interface level (for enhanced productivity) but may also be manipulated at the port level (for precision design manipulation).

They can also be programmed to be configured through a user-created GUI. For example, if a parameter for the size of the output of a counter is created, this parameter can be then shown in this GUI (Figure 2.2.5) so that, when this counter is packaged into an IP Core, it can be configured for that specific design, allowing for more flexibility.


Binary Counter (12.0)

Documentation IP Location

IP Symbol | Information

Show disabled ports

Component Name: c_counter_binary_1

Basic | Control

Implement using: Fabric

Output Width: 12 [1 - 256]

Increment Value (Hex): 1 Range: 1...FFF

☐ Loadable

☐ Restrict Count

Final Count Value (Hex): 1 Range: 1...FFE

Count Mode: UP

☐ Sync Threshold Output

Threshold Value (Hex): 1 Range: 1...FFE

Figure 2.2.5: Xilinx Counter LogiCore IP configuration GUI

Then, this blocks can be interconnected in a schematic window. These blocks may also have interfaces, that is, standardised group of ports for a specific function, for instance, clock, reset, an AXI bus or a differential pair for SFP connection. IP Integrator can also act as a guidance for connecting these signals (figure 2.2.6) indicating which are the compatible interfaces. It goes even further than that, including two wizards called *Block Automation* and *Connection Automation* that instantiate any intermediate block need for two IP Cores to communicate between themselves and connects them following the interfaces and port definitions.

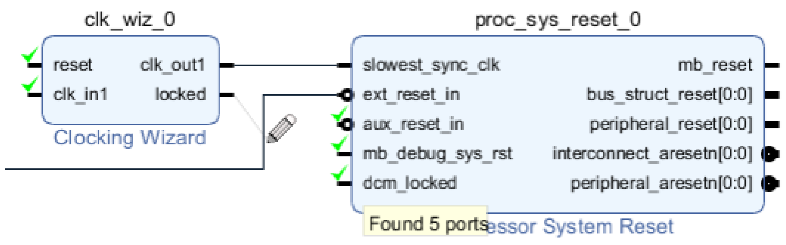


Figure 2.2.6: Connecting IP Cores. Green ticks indicate that the ports can be connected. Notice that for example, outputs cannot be connected to outputs [27].

For this project, the IP integrator will be a central part of the design process as it will be where all the changes for ensuring correct functionality will be made. It is that important because the PS part, which is not a FPGA itself needs to be instantiated as an IP Core for the tools to consider it a part of the system. Figure 2.2.7 shows the block that represents the *Zynq UltraScale+ MPSoC Processing System* and figure 2.2.8 represents the internal block diagram where, just by clicking into one part leads to another window where that specific part of the PS can be configured.

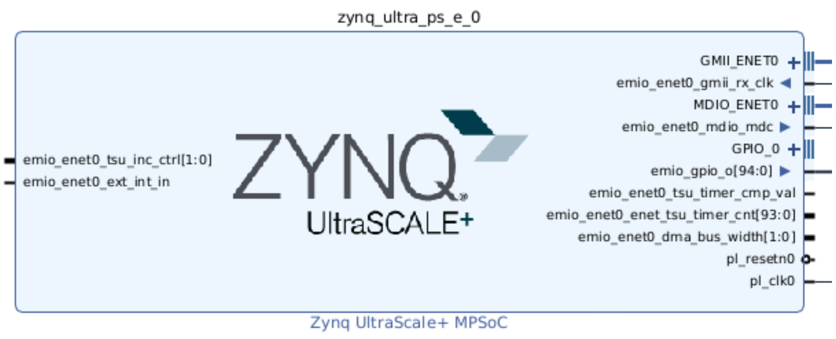


Figure 2.2.7: Zynq UltraScale+ MPSoC Processing System IP block in IP Integrator.

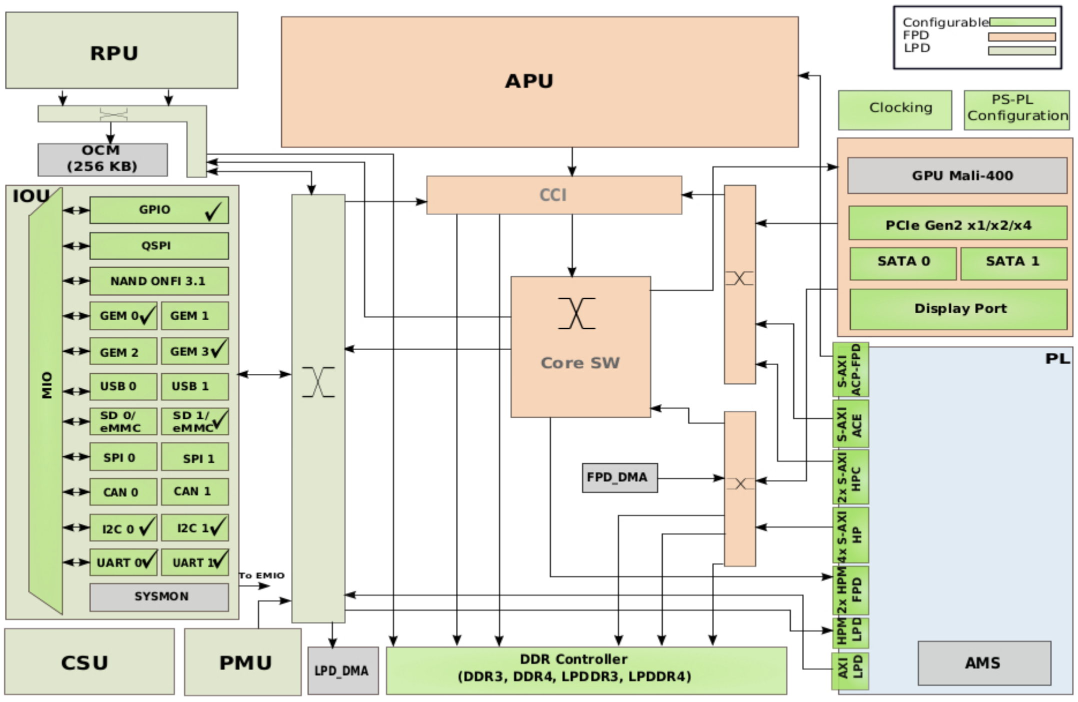


Figure 2.2.8: Zynq UltraScale+ MPSoC Processing System internal block diagram for configuring the different parts of the PS, enabling the capabilities that the user desires. For example, in this design two Gigabit Ethernet MAC (GEM) are enabled for two ethernet ports.

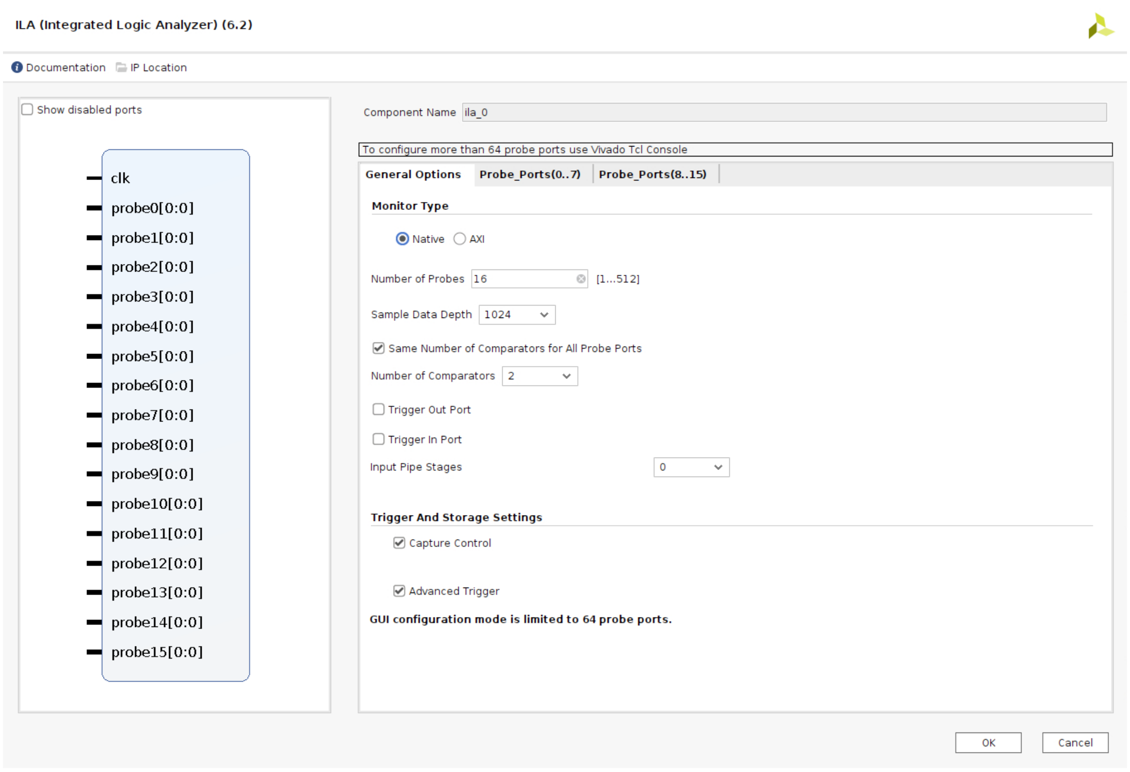
2.3 Vivado Logic Analyzer

Simulations have been mentioned in previous sections. However, these simulations are done on the computer, without using the target board where the hardware is going to run. Once the design is finished it is programmed into a board to test it. Is it possible to debug the system if it is already programmed on the board? The answer is *yes, absolutely*. Vivado offers a tool, available in the Hardware Manager, called the Logic Analyzer. It works as follows:

1. During the design flow, a core called Integrated Logical Analyzer (ILA) [28] can be added in the IP Integrator block design. This core takes up some resources of the FPGA to create an internal debug system that will have in its *probe* inputs some signals of the circuit under test.

   This core allows up to 64 probes from the GUI (figure 2.2.9) and it can be configured by double clicking it with the following options:

- *Monitor Type:* Native or AXI. This option specifies which type of interface ILA should be debugging, native is usually used unless an AXI interface is debugged. They are commonly used in complex IP Cores that need to output data in a standardised way. The ports listed here belong to the native option.
- *Number of probes:* number of signals to be probed for debugging.
- *Sample Data Depth:* The number of bits sampled. The more depth is chosen in the drop down menu, the more blockRAMs will be needed and longer signals can be visualized.
- *Number of comparators:* comparators are used to trigger the data sampling, for example, in the rising or falling edge of a signal or when some signal is equal to other.
- *Trigger In and Out ports:* toggles to include an optional input or output for triggering the data sampling.
- *Input Pipe Stages:* number of registers to add to the input probes.
- *Capture Control and Advanced Trigger:* toggles to enable capture control and the state machine-based trigger sequencing.



Component Name ila_0

To configure more than 64 probe ports use Vivado Tcl Console

Monitor Type

[✓ Native] AXI

Number of Probes 16 [1...512]

Sample Data Depth 1024

[✓] Same Number of Comparators for All Probe Ports

Number of Comparators 2

[ ] Trigger Out Port

[ ] Trigger In Port

Input Pipe Stages 0

Trigger And Storage Settings

[✓] Capture Control

[✓] Advanced Trigger

GUI configuration mode is limited to 64 probe ports.

Figure 2.2.9: Integrated Logical Analyzer IP Core configuration GUI

2. When the FPGA is programmed with the bitstream, the Hardware Manager will show a window with the Logic Analyzer, as it will detect that the design being tested has an ILA instantiated on it.

3. The tool waits for the trigger configured in the core to start sampling data. The ILA core is in charge of sampling the data in the FPGA BlockRAM and then sending it through JTAG to the Vivado Logic Analyzer.

4. When the sampling is finished, the Logic Analyzer will show the signals probed during that time (figure 2.2.10 for the example of the two counters explained in the workflow), allowing for debugging in a interface very similar to a verification done right after synthesis.

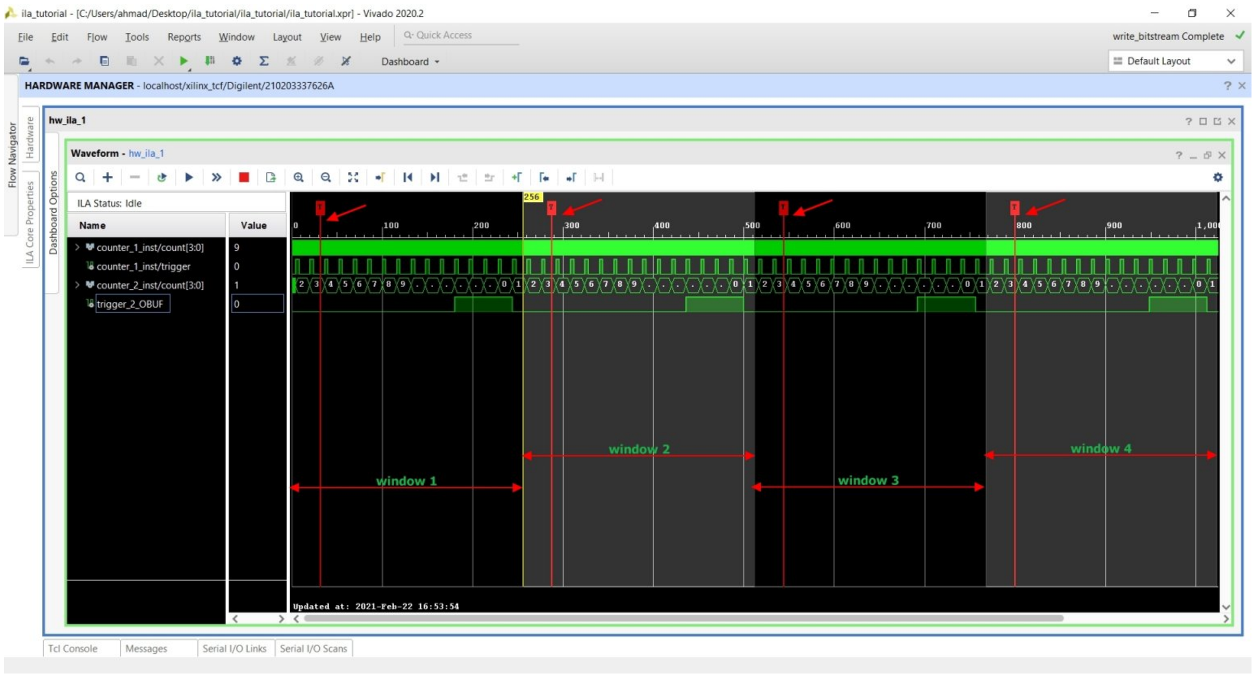

ila_tutorial - [C:/Users/ahmad/Desktop/ila_tutorial/ila_tutorial/ila_tutorial.xpr] - Vivado 2020.2
File Edit Flow Tools Reports Window Layout View Help
Quick Access
write_bitstream Complete
Dashboard
Default Layout
HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/210203337626A
hw_ila_1
Waveform - hw_ila_1
ILA Status: Idle
Name
Value
counter_1_inst/count[3:0]
9
counter_1_inst/trigger
0
counter_2_inst/count[3:0]
1
trigger_2_OBUF
0
window 1
window 2
window 3
window 4
Updated at: 2021-Feb-22 16:53:54
Tcl Console
Messages
Serial I/O Links
Serial I/O Scans

Figure 2.2.10: Vivado Logic Analyzer Waveform visualizer [26]

Through this interface, the user can perform tasks such as adding probes to the visualization, zoom in or out the signals, restart data sampling again and even do a manual trigger in case the data is not sampled due to a failure in the trigger signal. In figure 2.2.10, the red T's are the moments when the trigger was produced. Notice that the time window of the signals captured is the one after the trigger. This behaviour can be changed, the user can capture a time window that is completely before the trigger, completely after or with the trigger in the middle of it.

Chapter 3

Software side: Embedded OS

In this chapter, an introduction to the OS being used in the ZCU102 EvB will be given. The explanations given here will serve to understand the OS configuration while building it and the most important elements to consider to reach the desired specifications in the DPB.

3.1 The embedded Linux kernel

Hello everybody out there using minix - Im doing a (free) operating system (just a hobby, wont be big and professional like gnu) for 386(486) AT clones. [...] It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as thats all I have :-(.

Linus Torvalds, August 1991

This quote comes from the founder of Linux itself, heavily understating the importance that what he was developing in that time would have in the future. Today, more than 96.4% of the top 1 million server applications run on Linux. Across public clouds, 90% of the workloads use Linux as their OS. [29].

Linux owes its success to its open-source nature, allowing for full customization and optimization for specific applications. There is a lot of demand for that in servers, as they need to optimize its resources from a space and cost perspective and having a OS that fully uses all the resources the best way possible is ideal.

However, in this TFM the focus will be placed in embedded systems, which is the kind of system the DPB is: Embedded Linux systems appear in multiple shapes and forms. They vary from one another and are widespread across virtually every technology segment. Progress in semiconductor technology has substantially helped spur Linux's adoption in the embedded world, with many reasons accounting for such growth. System on Chip (SoC) hardware with low-power and small footprint is increasingly becoming developers premier processor choice. The ZCU102 can be considered a large memory embedded system, as it sports 4GB of RAM which is in the realm of normal computers, though, the CPU is ARM, adapted to the embedded systems environment and by default there are no hard drives in the ZCU102, only flash memories with 128MB to 16GB, another characteristic aspect of embedded systems.

In this piece of hardware, the linux kernel is run. The Linux kernel is a member of the family of Unix-like OS kernels, with AT&T Bell Labs devising the first version of Unix back in 1969. Being proprietary, Unix, the first portable OS, stimulated the inevitable development of free and open source alternatives like Linux and, among the many others, FreeBSD, NetBSD, and OpenBSD.

At system startup, the Linux kernel loads into RAM and stays in memory throughout the session duration. As the first program to load, it is the fundamental core of an OS in that it has complete control over everything occurring in the system. Similar to how user commands occur in the so-called user space, the kernel executes processes and handles interrupts in kernel space.

3.2 OS election: Petalinux or Ubuntu

Linux kernel is not the complete OS, just the layer that software used to request access to the underlying hardware. A complete embedded Linux system definition is that of a distribution (or distro). As an umbrella term, it usually denotes software packages, services, and a development framework on top of the OS. After boot, the Linux kernel mounts a root file system with system libraries, critical files, and loadable modules. Hence, even a tiny embedded Linux distribution comes with the necessary executable target binaries, tools, and utilities like glibc and busybox, populated in a directory structure on a root file system.

Of course, not all Linux distributions are suitable for an embedded environment. Arguably, a distribution for embedded applications differs greatly from a desktop environments one. While both may include startup scripts to initialize network interfaces, a desktop Linux distro will likely support a full-blown GUI stack that may instead be lacking in an embedded environment.

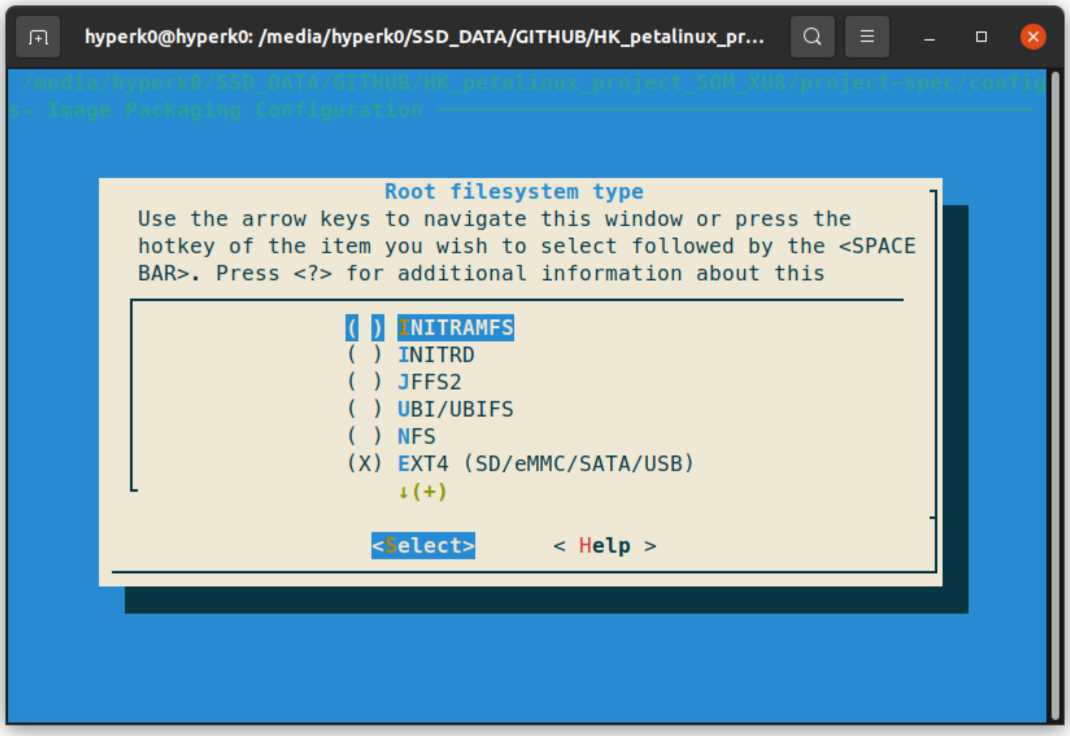
Here is where Xilinx offers two options for having Linux in its Zynq UltraScale+ boards: Petalinux and Ubuntu Desktop, not to be confused with Ubuntu Core, even though, being in a embedded system would suggest that Ubuntu Core would be a better choice. From my point of view, the decision of choosing as second option Ubuntu Desktop instead of Ubuntu Core is giving the choice of having two OS that differ a lot between each other so that it covers the needs of all of its customers.

For this project, **Petalinux** has been chosen due to the following reasons:

- **Customizability:** Petalinux provides a complete SDK that has the ability to customize the operating system to specific hardware and software requirements. This is critical in the DPB as hardware and software must serve a very specific purpose that is not usually the case in standard desktops: reliability. With Petalinux, it is possible to tailor the operating system to meet the specific requirements of the Zynq MPSoC, optimizing performance and minimizing resource usage.
- **Size:** Petalinux allows for a highly optimized, minimalist system image which is well-suited for use in embedded systems. This is important in the DPB as one of the measures considered to enhance reliability is to have redundancy of booting images. The smaller the image the more copies in memory can be placed. In contrast, Ubuntu Desktop is designed for use on desktop computers and typically includes a large number of applications and features that are not necessary for an embedded system, resulting in a larger image size.
- **Flexibility:** Petalinux provides a high degree of flexibility in terms of package selection, configuration, and customization of even the source code of the kernel, allowing for patches on the fly before compiling the boot image. This makes it well-suited for use in a system where the hardware is also designed together with the software, ensuring that any bugs specific to a device like SFP module or sensor can be patched without much trouble. Ubuntu Desktop, on the other hand, is a more general-purpose operating system that is not as flexible or customizable as the image that provides Xilinx is already pre-compiled.

3.3 Supported File Systems

One of the advantages regarding customizability in Petalinux is the choice of the file system to be mounted after booting the kernel. The file systems supported by Petalinux mainly differ on the kind of memory they are mounted one. Figure 2.3.1 shows the menu in `petalinux-config` with the main file systems available.



```
hyperk0@hyperk0: /media/hyperk0/SSD_DATA/GITHUB/HK_petalinux_pr...
 /media/hyperk0/SSD_DATA/GITHUB/HK_petalinux_project_SOM_XU8/project-spec/config
s→ Image Packaging Configuration ──────────────────────────────────────────

                          Root filesystem type
   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

                   ( ) INITRAMFS
                   ( ) INITRD
                   ( ) JFFS2
                   ( ) UBI/UBIFS
                   ( ) NFS
                   (X) EXT4 (SD/eMMC/SATA/USB)
                       ↓(+)

                 <Select>      < Help >
```

Figure 2.3.1: `petalinux-config` GUI showing the file systems supported by Petalinux

- *Initramfs*: it is a file system stored completely in the main memory of the OS, in this case DDR4 RAM. It is volatile, so the contents will be lost when the system is shutdown.
- *Initrd*: it is stored in the main memory also. The only difference between this system and *initramfs* is the bootup process. While *initrd* needs a block device called `ramdev` to unzip the contents it holds so it needs a driver. *initramfs* mounts a temporal directory in the `root` directory, which doesn't require a driver.
- JFFS2: used as the file systems for QSPI NOR Flash memories. When the file system is needed to be in a non-volatile memory and the storage options are these type of flash memory, this is the file system used.
- *UBI/UBIFS*: UBIFS is a new flash file system developed by Nokia engineers with help of the University of Szeged. In a way, UBIFS may be considered as the next generation of the JFFS2 file-system that works over different kind of memory volumes. While JFFS2 works over Memory Technology Devices (MTD), UBI does so over UBI subsystems which are higher level entities than MTD.
- *NFS*: Network File System, the file system will be mounted in a remote location and the board will need access to internet to access to its own file system.
- *EXT4*: the defacto Linux file system used in the most common Linux distributions. This extension allows to use almost any kind of device as the device to mount the *rootfs*: USB, SATA, SD, eMMC.

The chosen file system will be *initrd* or *initramfs*, as they work over the main memory of the system which, in this case, has ECC and has a longer lifecycle than MMC memories in the board which would fail in less than the requested 10 years duration if the OS is being very write-intensive on them.

3.4 The Yocto Project

Petalinux wouldn't exist without Yocto, a collaborative, open-source initiative that operates under the Linux Foundation, the nonprofit consortium dedicated to fostering the growth of Linux [30]. It comprises multiple projects related to embedded Linux software development under one umbrella. Yocto's family of open-source subprojects attempt to ease the process of configuring cross-compilation environments and retrieving, building and shipping Linux-based systems for various processor architectures. Developers interested in building a customised Linux distribution for their use case evaluate Yocto against common development-oriented Linux distros like Debian or Ubuntu. Yocto is **NOT** a Linux distro, it is a platform to develop your own one. It provides a flexible set of tools and a space where embedded developers worldwide can share technologies, software stacks, configurations, and best practices that can be used to create tailored Linux images for embedded and IOT devices, or anywhere a customized Linux OS is needed. The structure of this project can be summarised in figure 2.3.2.

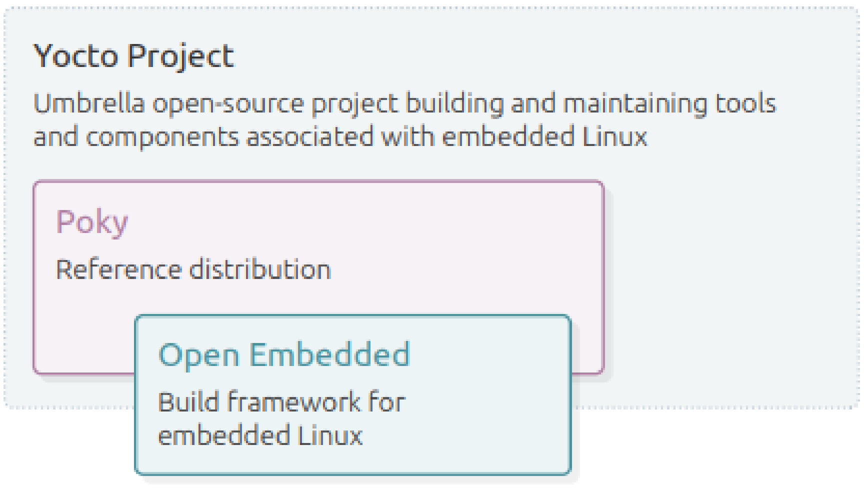

Yocto Project
Umbrella open-source project building and maintaining tools and components associated with embedded Linux
Poky
Reference distribution
Open Embedded
Build framework for embedded Linux

Figure 2.3.2: Yocto comprises multiple open-source projects hosted by the Linux Foundation. From the framework used to build the OS itself to Poky, the reference distribuiton as a barebones OS [30].

Yocto establishes a toolchain, that is a set of tools, to compile the kernel and file system to the target device together with a boot loader. To establish that toolchain, Yocto uses three configuration mechanisms together:

- **Metadata:** With Yocto, developers build from upstream source code and have virtually complete control over every stage of compiling and creating the target. The Yocto build process is extensively configurable due to the heavy use of metadata information exposed to the end developer. The three most important metadata parameter can be found in the *local.conf* file in the Yocto directory. Here is an example for the ZCU102 of the beginning of this file, showing the kind of machine, distribution and package format configuration:

```
MACHINE ??= "zynqmp-generic"
DISTRO ?= "petalinux"
PACKAGE_CLASSES ?= "package_rpm"
USER_CLASSES ?= "buildstats"
PATCHRESOLVE = "noop"
```

- **Recipes:** Recipes are metadata files specifying how to build a particular software package. A recipe standard layout includes descriptive metadata providing generic information about the recipe and the hosting location of the software package it builds. Recipes also provide licensing information, a section about package managers, package names and versions, build and runtime dependencies and how to create the packaged software bundle. Next to providing instructions for building packages, recipes describe where to download source code and patches, how to configure the source and how to compile it. Furthermore, recipes also describe dependencies between libraries or other recipes.

- **Layers and BSPs:** Layers are a collection of related recipes. With Yocto, developers can quickly build an image suitable for emulation and later customise their build for specific hardware by adding a hardware layer into their development environment. Furthermore, Yocto provides Board Support Package (BSP) as metadata layers tailored to a specific embedded board. BSP facilitate running a particular OS with different types of hardware devices. Xilinx takes advantage of this by building a reference BSP for the ZCU102 where the user can give its own ZCU102 .xsa and adjust the OS image to any new hardware added with respect to the reference design. Figure 2.3.3 shows the dependence structure of an application in Yocto.

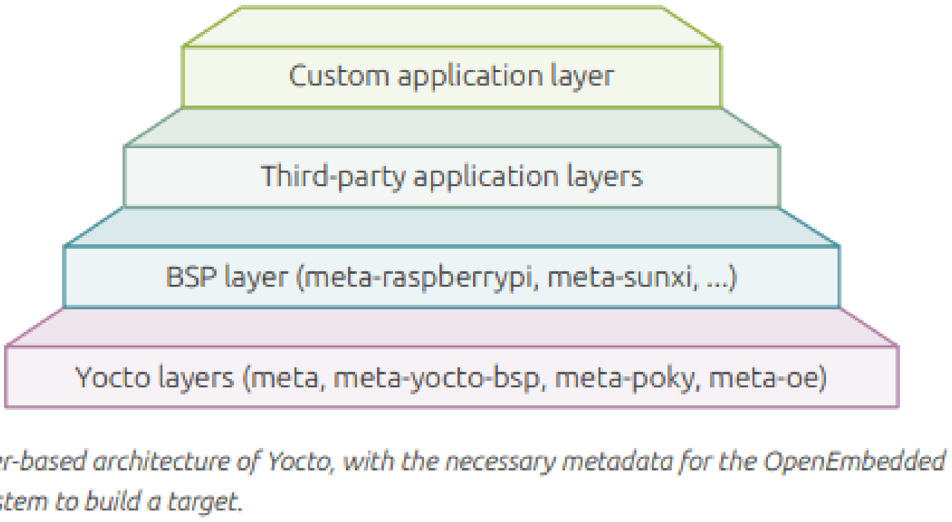

Custom application layer
Third-party application layers
BSP layer (meta-raspberrypi, meta-sunxi, ...)
Yocto layers (meta, meta-yocto-bsp, meta-poky, meta-oe)
The layer-based architecture of Yocto, with the necessary metadata for the OpenEmbedded build system to build a target.

Figure 2.3.3: The layer-based architecture of Yocto, with the necessary metadata forming a pyramid with several steps: embedded linux kernel, specific BSP files and third party applications that your application may depend on [30].

This layer system will allow to easily port applications to the Petalinux embedded system as one of the usual problems for running applications in the linux kernel are dependencies, which, in Ubuntu are solved downloading them when installing anything through `apt-get` but in Yocto, these dependencies must be told to the image compiler so that everything is installed correctly.

3.5 The Init System

When the OS boots, it must be prepared to develop the task it has been designed to. This means that an initial configuration of the services and applications that will perform these tasks must be done. For that, the Linux kernel has the *Init* function, called after the own kernel initialization that starts services running simultaneously in the background. For example, the application in charge of packaging data from the digitizer boards and send it to the DAQ or the NTP service are examples of this.

Petalinux supports two types of init systems:

- **system V:** it is the traditional Unix Init system thought to execute console scripts in the */etc/init.d* directory when booting up Petalinux. This method requires a file with all the commands for starting the service. In Petalinux, this scripts are installed through a Yocto recipe that also configures the *rc.d* directory and executes the `start` command for systemv.
- **systemD:** Petalinux got updated in 2022.1 version due to Yocto being updated to use this new init system as default in version 3.0 also codenamed Zeus. This system differs from the previous one in that it is not only a replacement to how init scripts and tasks are run in Linux. It is a much more complex and has a bigger scope for integrating several parts of the system as a super service that allows parallel execution, faster boot speed and replace services like journal, energy management and network management among others. SystemD is in charge of booting everything under the kernel allowing several parallel processes, achieved through the implementation of a dependency management service based on services control. Resources assigned to those services are optimized through cgroups or control groups, a Linux kernel feature that groups the processes in groups so that the resource usage by those processes can be assigned, accounted and isolated so that processes inside that group cannot exceed a determined memory usage, CPU utilization, I/O bandwidth, etc… These services are in fact the tasks that must be run in the background from the moment the system boots up. The management of these services is done through the following systemd utilities:
  - *systemctl:* Controls the administrator and system services. Used for starting, stopping, resetting and monitoring the services running under systemd.
  - *journalctl:* it keeps track of the rest of systemd utilities so that, in case of error, the service that has failed can be found easily.
  - *hostnamectl:* used to control the host name.
  - *localectl:* for controlling keyboard distribution and system language.
  - *timedatectl:* set the system date and time.
  - *systemd-clgs:* shows cgroup content.
  - *systemadm:* front-end for systemctl command.

  Services are created following the Linux philosophy where *everything is a file*, so even these services are represented as a file which is exposed to the file system. These files have extension *.service* in the folder `/etc/systemd` and has a similar form to a *.ini* file, where there are several sections with the name between square brackets [ ], each section with their corresponding options like shown in the example below.

```
[Section1]
Directive11=value
Directive12=value

[Section2]
Directive21=value
Directive22=value
.
.
.
```

  The most important sections together with their attributes and an explanation of each one of them are shown in table 3.1 and examples of services creation can be found in the Part III, chapter 7.

Section	Option	Explanation
Unit	*Description*	State the name and basic functionality of the unit
	Documentation	This directive provides a location for a list of URIs for documentation. These can be either internally available man pages or web accessible URLs.
	Before	The units listed in this directive will not be started until the current unit is marked as started if they are activated at the same time.
	After	The units listed in this directive will be started before starting the current unit
	Requires	This directive lists any units upon which this unit essentially depends. If the current unit is activated, the units listed here must successfully activate as well, else this unit will fail
	Wants	This directive is similar toăRequires, but less strict. The units listed here can be started in parallel with this unit instead of having to wait
	Conflicts	This can be used to list units that cannot be run at the same time as the current unit
Install	*Alias*	This directive allows the unit to be enabled under another name as well
	RequiredBy	States a unit that will fail if this unit fails to start
	Also	Enable and disable units together with this one when this is enabled or disabled
Service	*Type*	state the type of service being started
	ExecStart	This specifies the full path and the arguments of the command to be executed to start the process.
	ExecStop	This indicates the command needed to stop the service. If not give, the process will just be killed with kill command
	Restart	This indicates the circumstances under which systemd will attempt to automatically restart the service. This can be set to values like always, on-success, on-failure, on-abnormal, on-abort, or on-watchdog
	RemainAfterExit	This directive is commonly used with the oneshot type. It indicates that the service should be considered active even after the process exits

Table 3.1: Most common sections in a *.service* file [31]

3.6 The Device Tree

The Linux kernel acts as the interface between hardware and software. That means that the kernel itself must be aware of what is available in both worlds. The software part is already explained thanks to the configuration Yocto does, the OS knows which packages are installed, the dependencies between them and the services running through the init system of choice.

This section focuses on how the Linux kernel knows the available hardware resources in the platform. This information is stored in the *Device tree* [32], a data structure and language for describing the hardware in a platform. It is stored in a file with extension .dts and .dtsi and contains all the specific hardware of the platform so that the operating system doesn't need to hard code details of the machine.

Structurally, the DT is a tree, or acyclic graph with named nodes, and nodes may have an arbitrary number of named properties encapsulating arbitrary data. A mechanism also exists to create arbitrary links from one node to another outside of the natural tree structure.

Conceptually, a common set of usage conventions, called bindings, is defined for how data should appear in the tree to describe typical hardware characteristics including data busses, interrupt lines, GPIO connections, and peripheral devices. A basic device tree structure is depicted in figure 2.3.4.

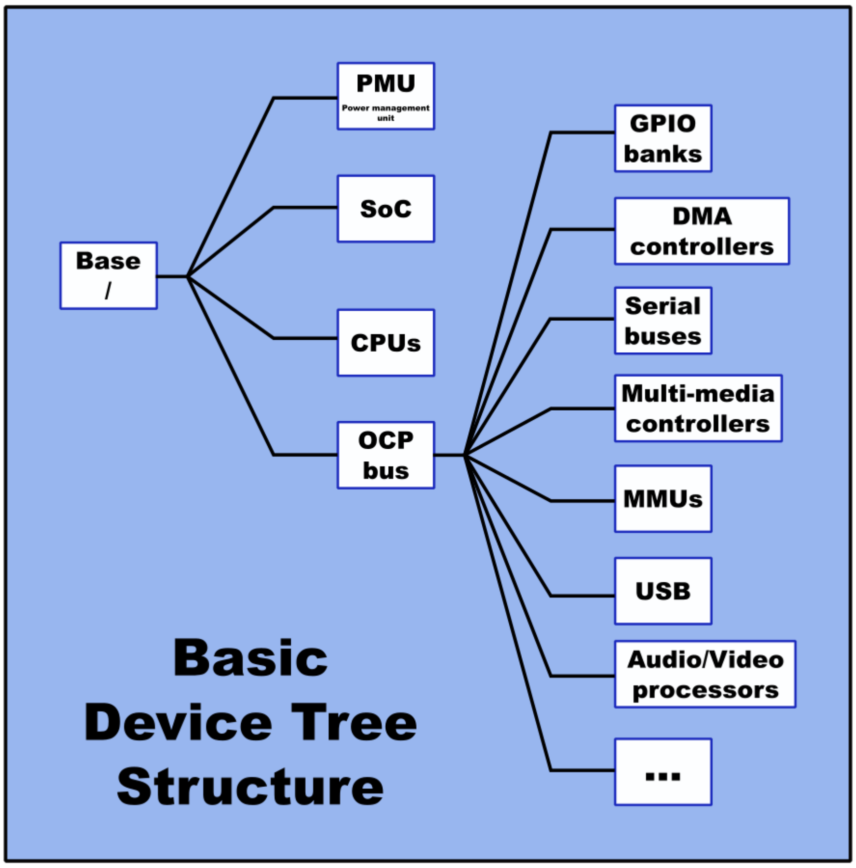


Figure 2.3.4: Basic Device Tree Structure

For Petalinux, focusing on the ZCU102 case there are several files to consider. These device tree files are already included in the ZCU102 BSP because the board has a known hardware.

As this is a hierarchy, several files, of extension .dtsi can be used in the same project with one of them including the rest, as if it was C code. In the ZCU102, this file is system-top.dts, that corresponds to the Xen Hypervisor when Petalinux wants to be run:

```
/dts-v1/;
#include "zynqmp.dtsi"
#include "zynqmp-clk-ccf.dtsi"
#include "zcu102-rev1.0.dtsi"
#include "pl.dtsi"
#include "pcw.dtsi"
/ {
chosen { //node name
bootargs = "earlycon"; //property name = property value
stdout-path = "serial0:115200n8";
};
aliases {
ethernet0 = &gem3; //When using &, it indicates a reference to another node
i2c0 = &i2c0;
i2c1 = &i2c1;
serial0 = &uart0;
serial1 = &uart1;
spi0 = &qspi;
};
memory {
device_type = "memory";
reg = <0x0 0x0 0x0 0x7ff00000>, <0x00000008 0x00000000 0x0 0x80000000>;
};
};
#include "system-user.dtsi"
/include/ "xen.dtsi"
```

This device tree can be divided in the following nodes:

- *Includes:* statements to call other files to have a cleaner and modular solution for the device tree. The includes in this case contain the files relative to the Zynq MPSOC chip, where there are several controllers for timers, ethernet, SD controllers, etc... It also includes the ZCU102 one for fixes specifically for the ZCU102 hardware and *pl.dtsi* for any device tree included in the PL (Programmable Logic) part and *pcw.dtsi* for the dynamic properties of the IPs inside the Processing System (PS).
- *Chosen node*: defines parameters defined by the system firmware at boot time. This is used to pass the kernel command line the command to establish the serial communication the moment the FSBL is loaded.
- *Aliases*: as alternative names to refer to a node. For instance, instead of using *uart0* when specifying the *stdout-path* property, it is used *serial0*, as it is an alias of *uart0* that already gives the idea that this node is a serial communication terminal.
- *Memory*: The main memory is defined here as a *memory* node with two ranges of memory of 2GB each accounting for the total of 4GB present in the PS. The property *reg* is a cell <>, with 4 32 bits values indicating the range of addresses reserved for each zone. This is an example of a node representing real hardware present in the EvB.

Chapter 4

Das U-boot Environment

In this chapter, the bootloader that will be used to load the OS through all the project will be explained together with the basic concepts of what a bootloader does and why it is essential in the system.

4.1 A bootloader's goal

Before executing any application the microprocessor must undergo the boot process which means going from a total powered-down state to having a fully functional OS that allows it to perform the tasks it was designed to. OS are very complex pieces of software that are usually stored in non-volatile large capacity devices. However, a CPU can only access its main memory when initialised (RAM or ROM) [33].

When the processor is powered on, the memory doesn't hold an operating system, so special software is needed to bring the OS into memory from a given Non-Volatile Memory. This software is normally a small piece of code called the *boot loader*.

On a desktop PC, the boot loader resides on the master boot record (MBR) of the hard drive and is executed after the PC's basic input output system (BIOS) performs system initialization tasks.

In an embedded system, the boot loaders role is more complicated because these systems rarely have a BIOS to perform initial system configuration which leaves the option of having the boot loader code in the memory itself loaded automatically after CPU initialization. Although the low-level initialization of the microprocessor, memory controllers, and other board-specific hardware varies from board to board and CPU to CPU, it must be performed before an OS can execute.

The functions that a boot loader should perform are as follows [33]:

- Initialize the hardware that will be used when the OS boots
- Provide an environment to perform commands that will boot the OS
- Issuing those commands to start the OS
- Read and write in arbitrary memory addresses for getting information about the OS and the configuration parameters.
- Transfer images from non volatile devices to the RAM and viceversa.

For the case of Xilinx Zynq UltraScale+ MPSoC, as an embedded system, must have a boot loader to perform those tasks. The chosen boot loader is called *U-Boot*, an open source project under the General Public License (GPL) license born as a combination of two small bootloaders PPCboot and ARMbootthese bootloaders were merged to create U-Boot [34].

The U-Boot project uses some portions of the Linux kernel code and has a similar source code structure. This fact along with other advantages such as stability, support for many processors and boards, easiness to adapt to other platforms, and active community of developers enhancing and supporting the project have contributed to make U-Boot, the most used Boot loader.

The features of U-boot are as follows [34]:

- **Bootstrap the hardware platform:** that is initialising the hardware in the machine, for example asserting the reset lines of the FPGA cores that are visible to U-boot thanks to the device tree configuration.
- **Load an OS** and transfer the control to the user of it.
- **Network capabilities** such as Dynamic Host Configuration Protocol (DHCP) for IP assignment or Trivial File Transfer Protocol (TFTP) and Network File System (NFS) for booting or maintaining the file system in a device not physically present in the machine but accessed through the internet.
- **Serial support**, used for printing the terminal in the development PC through a serial connection, usually USB.
- **Flash support and management:** NOR, NAND, and Multi-Media Card (MMC) or Secure Device (SD) cards, including their corresponding commands to copy content to main memory, check the CRC and test if they are working correctly.
- **Interactive shell** through which commands to configure the environment to perform the boot of an OS image are performed.

4.2 Execution flow in Xilinx devices

This section will focus on the U-boot execution flow, that is just one part of the four-stages boot of the system. This general boot procedure is depicted in figure 2.4.1, where U-boot is named as the second stage boot-loader. The First Stage Boot Loader (FSBL) is made by Xilinx and initialises the CPU and the memory together with the bitstream load so that U-boot is left with the hardware instantiated. We have to remember that the work is being done on a FPGA, meaning that the hardware being instantiated in it can take a lot of forms, it is not fixed so this step is required.

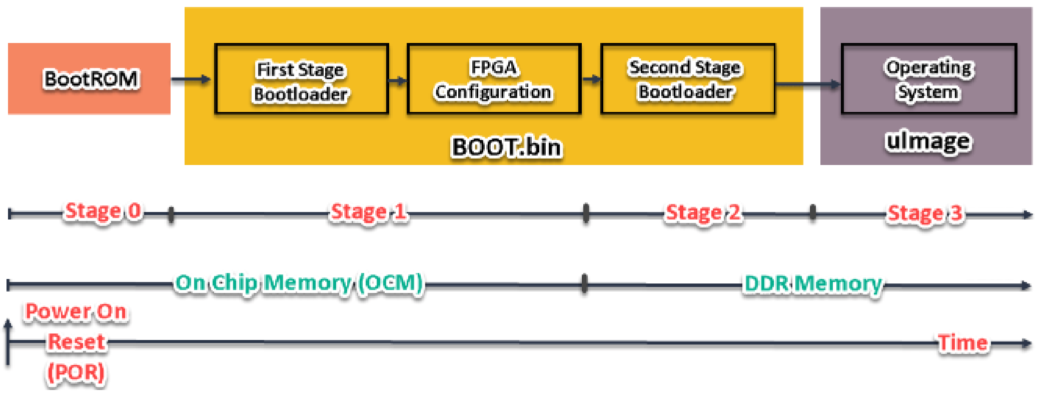


Figure 2.4.1: Zynq Boot Sequence [35]

After Power on Reset and FPGA configuration, the CPU does the following steps:

- **Execute the bootstrap** to configure interruption and exception vectors, clocks and SDRAM.
- **Decompress U-boot** from the BOOT.bin file that is located in a flash device to the main memory.
- **Execute U-boot.elf** file and pass the control to U-boot environment.

This procedure will present users a terminal printout very similar to the one displayed below. This example has been gathered from a default ZCU102 boot up. In this lines, information about the board is displayed, as well as the available amount of memory, the boot mode being used, which in this case is from a SD card and finally the configuration of two Zynq Gigabit Ethernet MAC (GEM) for Internet connection.

```
U-Boot 2022.01 (Sep 20 2022 - 06:35:33 +0000)

CPU:   ZynqMP
Silicon: v3
Model: ZynqMP ZCU102 Rev1.0
Board: Xilinx ZynqMP
DRAM:  4 GiB
PMUFW:  v1.1
PMUFW no permission to change config object
EL Level:       EL2
Chip ID:        zu9eg
NAND:  0 MiB
MMC:   mmc@ff170000: 0
Loading Environment from FAT... *** Error - No Valid Environment Area found
Warning - bad env area, using default environment

In:    serial
Out:   serial
Err:   serial
Bootmode: LVL_SHFT_SD_MODE1
Reset reason:   EXTERNAL
Net:   FEC: can't find phy-handle

ZYNQ GEM: ff0b0000, mdio bus ff0b0000, phyaddr 9, interface gmii
eth0: ethernet@ff0b0000FEC: can't find phy-handle

ZYNQ GEM: ff0e0000, mdio bus ff0e0000, phyaddr 12, interface rgmii-id
eth1: ethernet@ff0e0000
scanning bus for devices...
starting USB...
No working controllers found
Hit any key to stop autoboot:  0
ZynqMP>
```

The user is presented with the choice of autobooting, which runs another file called *boot.scr* with U-boot commands to perform an automatic OS boot. An example of this script is located in Appendix 1 and will explained in deeper while commenting how a redundant boot system can be configured, in Part III, chapter 6.

If the user chooses to stop autoboot, then an interactive shell will be displayed. This shell allows to perform a set of built-in commands for booting the system, managing memory, and updating an embedded systems firmware. A complete list of commands in U-boot is available in Annex 1, together with the explanation. That list is the result of typing `help` in the console. These commands can be divided in the following groups:

- **Information commands:** for getting information about the system, such as `bdinfo` for board information, `date` for system date and time or `fsinfo` for the file systems supported by the current configuration.
- **MII commands:** Media Independent Interface is used to interact with Ethernet PHY chips. These chips will be very important moving forward in the project as there will be a lot of interfaces using Ethernet protocols. A more detailed explanation of PHY chips readout can be found in Part III, chapter 9.
- **Network commands:** `dhcp` for automatic IP assignment or `ping` are the most usual commands for a normal system. However, in U-boot there is also the possibility of executing `tftpboot` command or `pxe get` to download the configuration for downloading an OS image from the Internet and boot the system from it.
- **USB commands:** Several commands for accessing the USB subsystem and reset the controller, read the USB device tree or directly reading blocks or complete OS images from the devices connected through USB.
- **Memory commands:** Used to manage RAM memory.
- **Serial port commands:** used to load files coming from the serial line.
- **I2C commands:** to interact with I2C devices. Some examples are `iloop` for reading a set of I2C address, `imd` to display the I2C memory of specific devices or `iprobe` to discover valid I2C chip addresses.
- **Environment variable commands:** U-boot gets most of its configuration parameter through environment variables, which act like variables in a piece of C code but this time the scope is bigger as they can be accessed by any software being executed in the environment. In this case, they are used for example, for knowing in which address the *boot.scr* is held in the main memory or for storing parameters such as the IP address, the board name or which set of commands should be executed to source the aforementioned *boot.scr*. This will be disclosed in more detail in Part III, chapter 6.

Chapter 5

Petalinux Tools for Embedded Linux Development

The biggest advantage of having such a well supported board that is used by so many developer is the effort that Xilinx puts to make every low level aspect of the development as easy as possible. One of the biggest challenges is the preparation of the required files for a correct boot up of the system. As it was commented in the previous section, in figure 2.4.1, there are 4 stages that require three files to go from one to the next. These three files (*BOOT.bin*, *boot.scr* and *image.ub*) must be generated somehow. For achieving that task, Xilinx makes available to the developers the *Petalinux Tools for Embedded Linux Development* [36].

5.1 SDK for creating a bootable image

PetaLinux is an embedded Linux Source Development Kit (SDK) targeting FPGA-based System on Chip (SoC) designs. It contains everything necessary to build, develop, test and deploy embedded Linux systems. This tool contains:

- **Yocto Extensible SDK:** Yocto itself has already been explained in the previous sections. One of the most glaring advantages is its customizability, that comes into life thanks to its Extensible SDK [37], which allows to easily add new applications and libraries to an image or modify the source of an already existing component and test the changes done. This SDK is integrated with the OpenEmbedded build system to automatize the building tasks of your newly added applications. Furthermore, OpenEmbedded also provides several collections of recipes packaged in different layers that make very easy to add the most common software packages to the custom OS image.
- **Xilinx Software Command-Line Tool (XSCT) and tool chains:** Petalinux SDK has a tight integration with other Xilinx tools because the firmware it generates is going to be installed in a system that is generated through these other Xilinx tools such as Vivado. Hence, it needs to support the same protocols to configure the board to run the firmware it generates. This means that the SDK itself can send XSCT commands for initializing the CPU, configure memory spaces and transfer files into the system's main memory. In the JTAG boot method, this will be explained further.
- **Petalinux CLI Tools:** This SDK doesn't have a GUI, meaning that instead of creating an image file in the same way as a .bit file in Vivado can be created by interacting with an User Interface, the command line is used, typing several commands to go through the workflow of the Petalinux SDK.

The purpose of this SDK is to provide several files ready to be flashed or transferred through different means to a Xilinx board and boot the system from them following all the steps depicted in figure 2.4.1. The most relevant files, written in the order in which they are used for booting up Linux are as follows:

- **BOOT.BIN** A binary file that can be defined as some kind of .ZIP file that holds in its interior several other files crucial to the first boot stages of the board. The files included inside *BOOT.BIN* when configuring the project for Zynq UltraScale+ MPSoC are listed below:
  - *pmufw.elf:* Contains the Platform Management Unit (PMU) firmware. PMU is in charge of performing the initialization of the system prior to boot, the power management, software test library execution and system error handling. It manages the state of the peripherals connected to the CPU and can receive requests from the latter to turn on or off the peripherals.
  - *zynqmp_fsbl.elf:* The First Stage Boot Loader (FSBL) loads the bitstream if any and the second stage boot loader from an external non-volatile memory or JTAG itself.
  - *bl31.elf:* it is part of the ARM Trusted Firmware platform, tthe equivalent to Secure Boot in PCs. BL31's purpose is to handle transitions between the normal and the secure world, that is, it is a monitor that ensures that the software being executed in the secure environment is also signed and marked as secure.
  - *system.dtb* It is a device tree blob file, that is, a file that contains the information about the device tree of the system. This file is generated using the .dtsi files whose syntax has already been explained in chapter 3.
  - *u-boot.elf* the second stage boot loader data. The FSBL loads this into memory and executes it. The result is having U-boot ready to execute the script that will lead Linux to boot up.
- **boot.scr:** a script file written in U-boot command syntax that is loaded together with *BOOT.BIN* in the main memory. When the load of U-boot is finished, an autoboot is performed, which is nothing more than looking for this file in the main memory of the system and execute it.
- **Kernel Image:** contains the built Linux kernel to be loaded into the board. U-boot targets this file when performing the boot-up of the system.
- **rootfs.tar.gz:** the root file system into a tar file. This root file system contains all the software packages selected during the configuration phase of the project.
- **image.ub:** it is another zipped file like *BOOT.BIN* that contains everything necessary to perform the second boot stage. It contains the Kernel Image, the root file system and the device tree blob into a single file. This images are usually called Flattened uImage Tree (FIT).

Figure 2.5.1 shows the order in which the predefined XSCT script loads the files into the memory and the order they are executed. Red squares show files that contain the files represented with green circles. First *BOOT.BIN* gets loaded, then *boot.scr* and finally *image.ub*, meaning that only three files need to get flashed by the developer to the board memory.

The usage of these tools requires having some previous knowledge about Linux and the command line together with how Yocto works in order to configure the project to generate an OS image that fits the specific needs of the project.

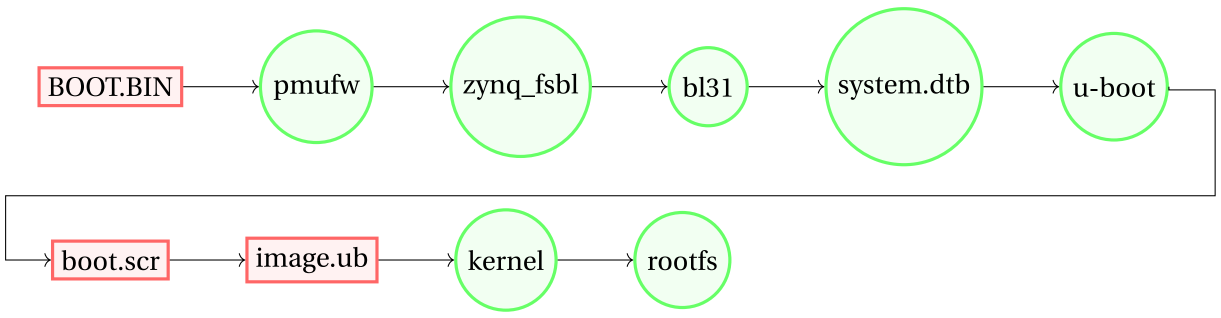


Figure 2.5.1: Files load order for performing a Linux Boot in the UltraScale+ MPSoC platform

5.2 Basic commands

In this section, a basic explanation of the commands available in the Petalinux SDK are shown. Table 5.1 shows a list of the commands together with their purpose inside the workflow of the tool.

Design Flow Step	Tool/Workflow
Hardware platform creation	Vivado Design Suite
Create PetaLinux project	`petalinux-create -t project`
Initialize PetaLinux project	`petalinux-config -get-hw-description`
Configure system-level options	`petalinux-config`
Create user components	`petalinux-create -t COMPONENT`
Configure the Linux kernel	`petalinux-config -c kernel`
Configure the root file system	`petalinux-config -c rootfs`
Build the system	`petalinux-build`
Test the system on qemu	`petalinux-boot -qemu`
Deploy the system	`petalinux-package -boot`
Update the PetaLinux tool system software components	`petalinux-upgrade -url/-file`

Table 5.1: Design Flow Overview. Basic commands to perform the design flow [38]

- `petalinux-create`: used to create objects that belong to a Petalinux project or the project itself depending on the options used in the command. Followed by letter `-t`, one can create a `project`, `apps` or kernel `modules`. For creating a project the user can use a predefined Xilinx template depending on the kind of CPU in the system (zynqMP, zynq or microblaze) or directly use a Board Support Package (BSP), which is a zipped file that can contain much more information about the specific peripherals and the routing of the chip in a specific EvB.

- `petalinux-config`: customize the existing project by using an auto-generated configuration GUI. When the project is already created the first step in the workflow is to specify to the tool which platform will be used. This platform file (.xsa) is generated by Vivado and its location must be given to Petalinux tools through the parameter `get-hw-description`.

  After that, the user can configure through parameter `-c` the different components of the system: the drivers included in the kernel, software packages in rootfs, u-boot parameters, device-tree nodes, etc... This options will be further explained in Part III, chapter 1.

- `petalinux-build:` Once the project has been configured, the user must use this command to generate the boot images. This command performs all the necessary actions to compile the kernel from pulling the source files from Github repositories of the kernel, u-boot and other software packages to managing dependencies and the build order so that there are no errors.

- `petalinux-boot`: used to boot petalinux through JTAG or by using QEMU emulator. This command is used when the user doesn't want to flash the OS image into any external memory and prefers to run the OS directly through JTAG like loading a bitstream in Vivado or use an emulator if they have no board available. The arguments for `petalinux-boot` should be either `-jtag` or `-qemu`.

  For JTAG, the SDK calls a XSDB that has the corresponding commands to load everything in the board. The usage of this custom XSDB file will be explained in Part III, chapter 2.

- `petalinux-package`: `petalinux-build` only generates the kernel image together with the rootfs. As explained in previous section, a *BOOT.BIN* file is needed with the files for the first stage of booting. This command performs the packaging of several files that were compiled individually by `petalinux-build` and produces the desired `BOOT.BIN` format. This gives a lot of customizability as any of this files can be changed before packaging by others not being built by the Petalinux Tools itself. However, this is not a recommended practice.

- `petalinux-upgrade`: Xilinx has a very strict compatibility matrix regarding its software products. Petalinux Tools share the same version numbering as Vivado and Vitis with the format *Year.Revision*. The Petalinux Tools guide [36] explains that it is crucial to use the Vivado and Petalinux Tools labelled with the same version. Otherwise, they are considered incompatible. However, as the revisions within the same year usually contain minor changes, Xilinx provides the `petalinux-upgrade` command that updates a project from a previous version to the version of the tools that the developer has installed in his PC so they can use the latest version of both Vivado and Petalinux tools released in that year if they contain fixes that are relevant to the project being worked on.

  NOTE: The upgrade of a Petalinux project requires to download the new version of the tools from Xilinx official webpage. Then, both the old Petalinux tools installed in the PC and the project should be upgraded using the following commands:

```
#Upgrade the Petalinux Tools
petalinux-upgrade -f /path/to/the/downloaded/new/tools

# Upgrade the Petalinux project just by doing petalinux-build.
# Yocto itself will ask whether you want to upgrade the project to a new version
petalinux-build
WARNING: Your Yocto SDK was changed in tool.
Please input "y" to proceed the installing SDK into project, "n" to exit:y
```

5.3 Petalinux project: File Hierarchy

As there is no user interface available for coding inside the Petalinux project, an external code editor (Visual Studio Code) must be used. This kind of editors are not aware of the file structure of the project as they are not specific for working in Petalinux so the developer must know where they should look for finding the file or set of files they need to edit in order to add a specific feature they need for its use case. This section enumerates the different folders that form a Petalinux Project with the most important subdirectories where the user should look. The file hierarchy shown in figure 2.5.2 belongs to Petalinux version 2022.2.

In this directory tree there are several folders:

- **build:** temporary files for building Petalinux image. This folder is created by `petalinux-build` command and contains all the sources, patches and intermediate steps in the toolchain to perform a correct image creation.

- **components:** folder containing the source files of yocto and petalinux project configuration. It contains the recipes of the software packages that are available in the OpenEmbeedded repositories.
- **images:** contains the FIT image, rootfs, etc... that result from `petalinux-build` command.
- **hardware:** this folder is usually used to store different .xsa files that contain the hardware information.
- **project-spec:** the most important folder from a user perspective as it contains everything that can be modified by the user to customize the Petalinux image. The other folders, save for some exception are not recommended to be modified because they are auto-generated by the tool itself. Inside this folder the user can find the meta-user folder which is a yocto layer created by the petalinux tools where all the user defined packages should go. This folder has the following subdirectories:
  - *conf*: text files to configure which predefined software packages from other Yocto layers should be added to the system.
  - *recipes-apps*: each folder inside this directory contains a Yocto recipe together with the source code of an self-made application to install into the petalinux rootfs. If any self-made user package wants to be installed, this is the folder where it should be copied to. Each of the application folder contains a *.bb* file containing the recipe and a *files* folder with the source code.
  - *recipes-kernel*: it is used to contain patches for the Linux kernel. Sometimes, not all the functionalities that the user wants are in the mainline kernel or there is some bug to be fixed. Petalinux tools provide the usual Linux patching system where a differential file is created and applied to the source code before compiling it.
  - *recipes-bsp*: contains two folders: u-boot and device-tree. u-boot folder contains the patches for u-boot in the same way as recipes-kernel contains Linux patches. device-tree folder contains the *system-user.dtsi* with device tree nodes added by the user who must make aware of the newly instantiated hardware in the .xsa file provided to Petalinux to the Linux kernel itself.

```
build
├── components
│   ├── yocto
│   └── plnx-workspace
├── hardware
├── images
├── project-spec
│   ├── configs
│   ├── hw-description
│   └── meta-user
│       ├── conf
│       ├── recipes-apps
│       ├── recipes-bsp
│       │   ├── device-tree
│       │   └── u-boot
│       └── recipes-kernel
└── README
```

Figure 2.5.2: Default directory tree of a Petalinux 2022.2 project

Chapter 6

Migration from version 2020.1 to 2022.2

Through the development of the project, the Petalinux version was updated from 2020.1 to 2022.2. The change has been decided because u-boot options are more integrated into the SDK, adding changes in the GUI to configure parameters that before could only be set up in the text files inside the project, which could lead to unexpected behaviour due to not being explicitly supported by Petalinux SDK. Moreover, as it will be seen in Part III chapter 10, there were some bugs that needed patches in a specific driver of the Linux kernel. This patch was already merged in the mainline kernel in 2022.2 version, meaning that the bug was no longer present and thus the patch wasn't needed.

This TFM has been written with the workflow used for 2022.2 version. However, I find interesting to see which are the most noticeable changes when porting from 2020.1 to 2022.2 so that there is a journal of the development process, which sometimes includes required updates of the SDK that can give developers a hard time.

The changes that must be made to the project are listed below among the ones found in the Appendix A (Migration) of the Petalinux 2022.2 reference guide.

- **Yocto version** the yocto version used to compile the Petalinux image with all the Bitbake recipes has changed to a more recent one which brings some of the below listed changes. The version of yocto used can be checked in the petalinux project folder: `components/yocto/layers/core/meta-poky/conf/poky.conf`. This is an example of the 2022.2 file where version 3.4.1 codenamed *honister* is used:

```
;poky.conf file for Petalinux 2022.2
DISTRO = {``poky''}
DISTRO\_NAME = {``Poky (Yocto Project Reference Distro)''}
DISTRO\_VERSION = {``3.4.1''}
DISTRO\_CODENAME = {``honister''}
SDK\_VENDOR = {``-pokysdk''}
```

However, in Petalinux 2020.1, version 3.0.1 codenamed *zeus* is used as seen in its corresponding file. This means that all the migration notes going from 3.1 to 3.4 must be considered in the migration.

```
;poky.conf file for Petalinux 2020.1
{DISTRO} = {``poky''}
{DISTRO\_NAME} = {``Poky (Yocto Project Reference Distro)''}
{DISTRO\_VERSION} = {``3.0.1''}
{DISTRO\_CODENAME} = {``zeus''}
{SDK\_VENDOR} = {``-pokysdk''}
```

- **Tiny rootfs:** Section 2, the image comes configured with a tiny rootfs made specially for small memories so that it takes less space. This tiny rootfs is just a linux environment to load a full fat rootfs aftwerwards. This behaviour can be changed by modifying the name of the image from `petalinux-initramfs-image` to `petalinux-image-minimal` in `petalinux-config -> Image packaging configuration -> INITRAMFS/INITRD image name`.

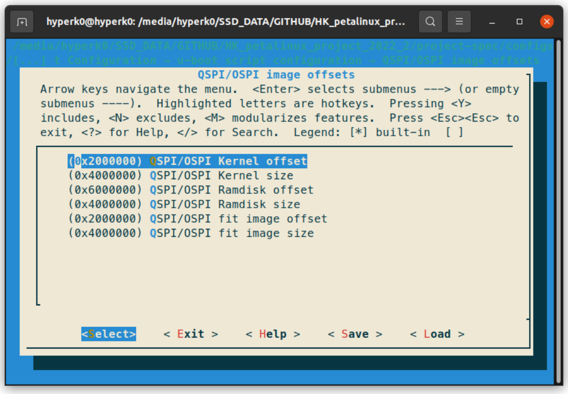
- **Syntax change in the bitbake recipe files:** Yocto override syntax has been changed so the code needed to be adapted to the new recipe syntax. One example of this is the removal of wildcards, special characters that can stand in for unknown characters in a text value. In the case of Yocto recipes these characters were * and ?.

- **New init system:** Previously, systemv was used as the init system of Petalinux. However, now the systemd system is the one used by default as it is more recent and the one used by modern Linux systems. Ubuntu uses it, that's why the `systemctl` command is used to enable services. Petalinux SDK offers the possibility to switch between both of them in `petalinux-config -c rootfs Image Features Init-manager`. This causes a change in what is explained in section 10. Now, that recipe must be modified so that it installs the init script so that it works with systemd. The new recipe file is as follows:

- **User management and root access:** Now the default user created inside the image is not *root*, but a user called *petalinux* and the password is created the moment you log in. You can change the petalinux password to a more permanent one by editing in `petalinux-config -c rootfs PetaLinux RootFS Settings (root:root;petalinux:petalinux:passwd-expire;) Add Extra Users` and setting up a password different to `passwd-expire` which is a Yocto command that enforces the user to set a new password at first login. The text file that will be edited by this action is the *plnxtool.conf* in `<project-folder>/build/conf`. For enabling root access, *debug tweaks* must be enabled in `petalinux-config -c rootfs Image Features`.

- **U-boot script template:** in Part III, chapter 6, the procedure to make modifications to the *boot.scr* file was explained. In 2022.2, the directory where this template is stored has changed from `<project-folder>/project-spec/meta-user/recipes-bsp/u-boot/u-boot-zynq-scr` to `<project-folder>/components/yocto/layers/meta-xilinx/meta-xilinx-core/recipes-bsp/u-boot/-boot-zynq-scr` and is now called *boot.cmd.generic*. Modifications must now be made here with the same procedure.

- **QSPI Flash offset settings:** the file for manually configuring QSPI offset for the different boot files no longer exists. That is due to a change on how Petalinux now configures the offsets for the different memory types. This allows an easier configuration of the offsets directly through the KConfig GUI in the following submenu: `petalinux-config U-boot configuration u-boot script configuration QSPI/OSPI image offsets`. Here there are several options to configure and no matter the type of memory present in the evaluation board. One of the problems of editing a text file that presents QSPI settings for several types of platforms is that, given the different size of the QSPI memory depending on the platform it was difficult to know which values should be changed. This procedure will be more detailed in Part III, chapter 3. Figure shows the configuration menu for changing QSPI offset settings in Petalinux 2022.2.



```
hyperk0@hyperk0: /media/hyperk0/SSD_DATA/GITHUB/HK_petalinux_pr...
/media/hyperk0/SSD_DATA/GITHUB/HK_petalinux_project_2022_2/project-spec/configs
/[...] t Configuration → u-boot script configuration → QSPI/OSPI image offsets
                      QSPI/OSPI image offsets
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

      (0x2000000) QSPI/OSPI Kernel offset
      (0x4000000) QSPI/OSPI Kernel size
      (0x6000000) QSPI/OSPI Ramdisk offset
      (0x4000000) QSPI/OSPI Ramdisk size
      (0x2000000) QSPI/OSPI fit image offset
      (0x4000000) QSPI/OSPI fit image size

      <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 2.6.1: QSPI offset settings in Petalinux 2022.2

Part III

Tasks Development and Results

Chapter 1

Project basic worflow

This chapter contains the basic workflow to start from scratch a Petalinux project going from downloading the tools and creating a project, going through all the basics in the customization process up to the point where a fully functional OS image has been created. Departing from the process explained in this chapter, the rest of the chapters in Part III will treat the procedure to add specific functionalities to the reference image.

1.1 Project creation

First, both the Petalinux SDK must be installed together with a baseline project creation.

1. Access Xilinx website to download the Petalinux Tools SDK of the desired version. In this case *2022.2*. The SDK is a zipped file with extension .run as it behaves as an installer of the tools. Issue the following command to install the tools:

   ```
   # To install the tools in a custom directory different
   # from the location of the .run file
   ```

   ```
   hyperk0@hyperk0:~$ ./petalinux-v2022.2-final-installer.run
   ```

   ```
   # To install the tools in the same directory as the .run file
   ```

   ```
   hyperk0@hyperk0:~$ ./petalinux-v2022.2-final-installer.run --dir
                       /custom/path/to/install/the/tools
   ```

2. Once executed the command, accept the terms of agreement by first reading them, press q to exit from the text reader and then pressing y as requested by the installer.

3. Once the tools are installed, the system is ready to create a Petalinux project. As everything is done from a terminal, just opening it and typing `petalinux-create` will not recognise the command because the environment needs to be configured first. To configure it inside the Petalinux tools directory there is a file called settings.sh that does so, so by issuing the command

   ```
   # To configure the terminal for Petalinux tools
   source <path-to-petalinuxtools>/settings.sh
   # You can also set an alias in the .bashrc for a more convinient setup
   alias petalinux_conf_2022="source <path-to-petalinuxtools>/settings.sh"
   ```

4. To have a reference design that will allow to easily compile a Petalinux image for the ZCU102, Xilinx offers in its webpage a link to download a Board Support Package (BSP), which, as explained before, contains a reference design for the ZCU102 together with a simple configuration that takes into account every peripheral connected to the PS in the ZCU102. From that reference, the developer can add its own cores to the PL, do the routing in Vivado, and finally configure everything in the reference Petalinux project.

5. Create a project using the downloaded BSP as a reference using the following command:

```
hyperk0@hyperk0:~$ petalinux-create -t project -s <path-to-bsp>
```

   This will result in a folder with the name of the BSP file with the project inside.

1.2 Project configuration

Now that the project has been created, it must be configured. This list of steps provides the procedure to do a basic configuration of the Petalinux image:

1. Move to the folder of the project using cd.

2. `petalinux_build` : do this the first time to download and create the yocto folder with all the recipes stored in the directory `<path-to-the-project>/components/yocto/layers`.

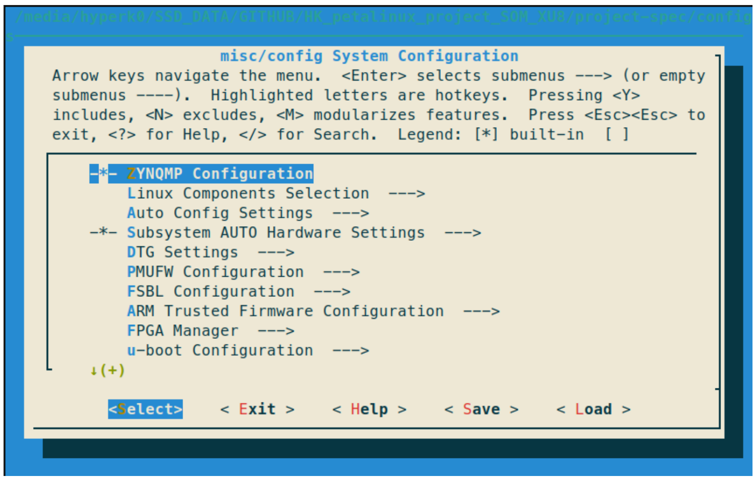
3. To select what to install go to `<path-to-the-project>/project-spec/meta-user/conf` and open the `user-rootfsconfig` with a text editor

4. There add `CONFIG_<name of the folder with the recipe you want to include>`. For example: *CONFIG_nano* to install the text editor nano.

5. Then set up you petalinux image accessing the `petalinux_config` with different arguments. In the figure below, there is the generated screen of the terminal after issuing that command:

```
# General configuration
```

```
hyperk0@hyperk0:~$ petalinux-config
```

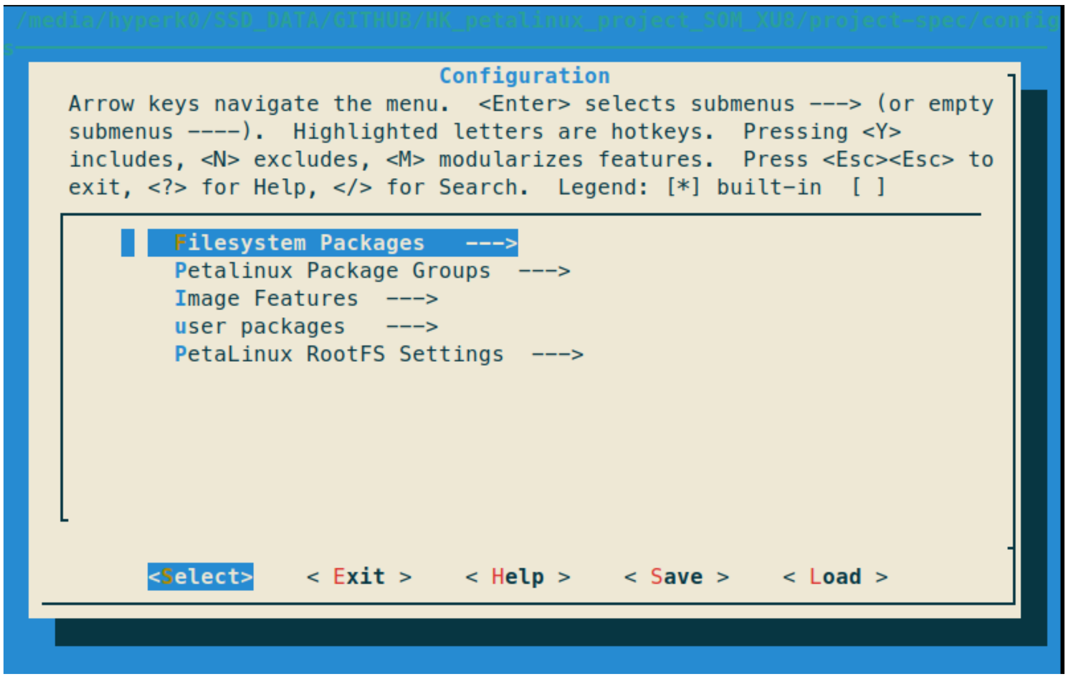


```
/media/hyperk0/SSD_DATA/GITHUB/HK_petalinux_project_SOM_XU8/project-spec/config
s
                    misc/config System Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

      -*- ZYNQMP Configuration
          Linux Components Selection  --->
          Auto Config Settings  --->
      -*- Subsystem AUTO Hardware Settings  --->
          DTG Settings  --->
          PMUFW Configuration  --->
          FSBL Configuration  --->
          ARM Trusted Firmware Configuration  --->
          FPGA Manager  --->
          u-boot Configuration  --->
      ↓(+)

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 3.1.1: Main Petalinux configuration screen

```
# Packages configuration
```

```
hyperk0@hyperk0:~$ petalinux-config -c rootfs
```

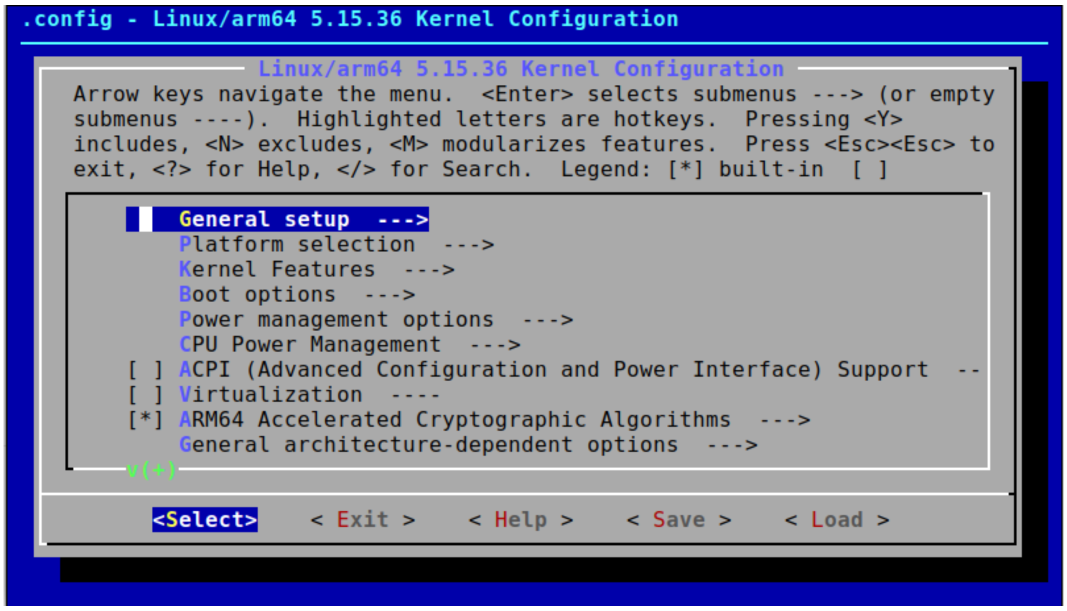


```
/media/hyperk0/SSD_DATA/GITHUB/HK_petalinux_project_SOM_XU8/project-spec/config
s
                              Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

          Filesystem Packages  --->
          Petalinux Package Groups  --->
          Image Features  --->
          user packages  --->
          PetaLinux RootFS Settings  --->

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 3.1.2: Root File System configuration screen

```
# Kernel configuration
```

```
hyperk0@hyperk0:~$ petalinux-config -c kernel
```



```
.config - Linux/arm64 5.15.36 Kernel Configuration
                 Linux/arm64 5.15.36 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

          General setup  --->
          Platform selection  --->
          Kernel Features  --->
          Boot options  --->
          Power management options  --->
          CPU Power Management  --->
      [ ] ACPI (Advanced Configuration and Power Interface) Support  --
      [ ] Virtualization  ----
      [*] ARM64 Accelerated Cryptographic Algorithms  --->
          General architecture-dependent options  --->
      v(+)

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 3.1.3: Linux Kernel configuration screen

6. In the rootfs configuration go to user packages and mark all the packages you want to install. Then use the right arrow key to select the Save command to exit saving the changes or press two times ESC to exit the configuration of rootfs without saving.

7. To enable the development version of the packages, that is, also including header files to perform application building in the board itself, you need to go to `<path-to-the-project>/project-spec/meta-user/conf` and open *petalinuxbsp.conf* file. There, add the following line:

```
IMAGE_FEATURES_append = " dev-pkgs"
```

NOTE: It is VERY important to leave a blank space between the first double quotes and dev-pkgs.

1.3 Image compilation and packaging

Once the project has been configured, the steps followed to build the petalinux image are the next ones:

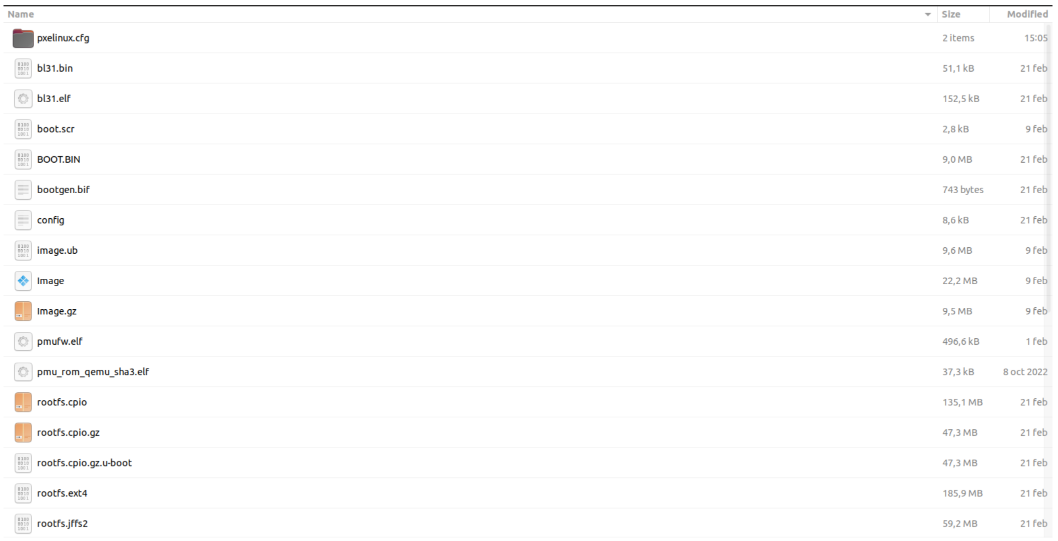
1. Build the image. This step will take a lot of time in the first try because it needs to download all the source code and compile it from scratch.

```
hyperk0@hyperk0:~$ petalinux-build
```

2. Package the FSBL among the other files required to boot in the first stage in a BIN file.

```
hyperk0@hyperk0:~$ petalinux-package --fsbl images/linux/zynqmp_fsbl.elf
--u-boot images/linux/u-boot.elf --pmufw images/linux/pmufw.elf
--fpga images/linux/system.bit --boot --force
```

3. The result after the two previous steps should be a directory in `<path-to-the-project>/images/linux` with the content shown in figure 3.1.4 .



Name	Size	Modified
pxelinux.cfg	2 items	15:05
bl31.bin	51,1 kB	21 feb
bl31.elf	152,5 kB	21 feb
boot.scr	2,8 kB	9 feb
BOOT.BIN	9,0 MB	21 feb
bootgen.bif	743 bytes	21 feb
config	8,6 kB	21 feb
image.ub	9,6 MB	9 feb
Image	22,2 MB	9 feb
Image.gz	9,5 MB	9 feb
pmufw.elf	496,6 kB	1 feb
pmu_rom_qemu_sha3.elf	37,3 kB	8 oct 2022
rootfs.cpio	135,1 MB	21 feb
rootfs.cpio.gz	47,3 MB	21 feb
rootfs.cpio.gz.u-boot	47,3 MB	21 feb
rootfs.ext4	185,9 MB	21 feb
rootfs.jffs2	59,2 MB	21 feb

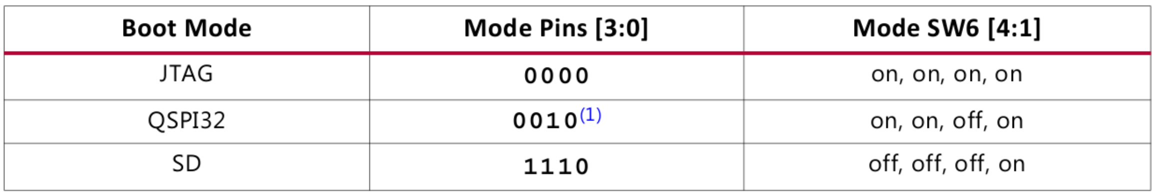
Figure 3.1.4: Linux images directory after completing the workflow

Chapter 2

System boot-up

Now that the OS image is ready, it needs to be loaded into the board. In this chapter, two very different ways for loading the image are presented: JTAG for loading it directly from the PC into the board and from an SD Card which represents using an external non-volatile memory.

Before booting the board, the DIP switches labelled SW6 need to be changed to a specific position depending where the BOOT.BIN file is. These positions are shown in figure 3.2.1 .



Boot Mode	Mode Pins [3:0]	Mode SW6 [4:1]
JTAG	0000	on, on, on, on
QSPI32	0010(1)	on, on, off, on
SD	1110	off, off, off, on

Figure 3.2.1: SW6 position depending on the boot mode [19]

For this DIP switch, in relation to the arrow, moving the switch toward the label ON is a 0. DIP switch labels 1 through 4 are equivalent to Mode pins 0 through 3 as per the Ultrascale+ technical guide [16].

2.1 Load image from JTAG

1. Change the switches group SW6 to boot from the JTAG and power on the board.
2. Connect UART and JTAG ports through USB to the computer.
3. Open Vivado to do the AutoConnect in the Hardware Manager. This is needed to enable the JTAG port and open the Hardware server. Close it afterwards.
4. Execute the following commands in two separate bash sessions with the baud rate in the serial port set to 115200:

```
hyperk0@hyperk0:~$ sudo minicom
hyperk0@hyperk0:~$ petalinux-boot --jtag --prebuilt 3
--hw_server-url TCP:localhost:3121
```

5. Petalinux should autoboot, leaving the user with the prompt to input username and password. By default: User: *root* and Password:*root*.

```
PetaLinux 2022.2_release_S10071807 xilinx-zcu102-20222 ttyPS0

xilinx-zcu102-20222 login: root
Password:
root@xilinx-zcu102-20222:~#
```

The command `petalinux-boot -jtag` has limited options due to its arguments, that don't allow to select specifically which files to load before booting. Depending on the arguments used, the following files get transferred to the main memory of the board (in this case the DDR4 ECC RAM): - `prebuilt <level>`: it picks the files from the `<project-folder>/pre-built/linux/images`. There are several levels, each one loads a subset of the files needed to boot up petalinux. -

- Level 1: loads the FSBL, PMU and the DTB (device tree blobbed). It doesn't load any environment to work on. This level is used when a boot loader different from u-boot wants to be used.
- Level 2: loads all the files from level 1 and the u-boot.elf, which includes the u-boot environment. Allows to use the u-boot terminal. It is very important to deny autoboot because there is no script loaded in the main memory so it will throw an error and fallback to the Distro boot method.
- Level 3: loads all the files from level 2 and the boot.scr, kernel image and rootfs. This can take a long time as JTAG has low transfer speeds. This method must be used if a full kernel wants to be booted using JTAG only.

The prebuilt levels can also be specified as keywords:

- `--u-boot`: Same as `prebuilt 2` but with the files from `<project-folder>/images/linux`.
- `--kernel`: Same as `prebuilt 3` but with the files from `<project-folder>/images/linux`.

If the user wants more customizability about what is being sent to JTAG before boot, a custom XSDB (Xilinx System Debugger) script is needed. The `petalinux-boot --jtag` command has another argument named `--xsdb-conn`, that allows to execute a custom command in xsdb terminal instead of the default one. By combining the `--tcl` to dump the commands used for the default JTAG boot and adding a new line with the `dow` command that copies the file to the desired memory address a new script called `custom_jtag_boot.xsdb` was created. So, executing this command boots the system with the custom instructions:

```
hyperk0@hyperk0:~$ petalinux-boot --jtag --hw_server-url localhost:3121
              --xsdb-conn source ../../scripts/custom_jtag_boot.xsdb --prebuilt 2
```

2.2 Load image from SD Card

NOTE: You don't need to open Vivado to do AutoConnect. This method is the easisest one for faster test but requires to have physical access to the board:

1. Connect UART port through USB to the computer.
2. Power on the board.
3. `sudo minicom` (with the baud rate set to 115200).
4. Power off the board.
5. Copy to a FAT32 SD CARD the following files: *"BOOT.BIN" "image.ub"* and *"boot.scr"*.
6. Connect the SD CARD to the development board.
7. Change the switches group SW6 to boot from the SD CARD.
8. Power on the board.

Chapter 3

Programming internal memory

As the DPB will be installed inside a vessel there won't be any human contact under normal conditions. This means that there will not be any JTAG or SD card slot to introduce the image of the OS. For commercial embedded systems it is fairly common to include a basic OS image in a flash memory to have a local way to boot. This chapter explains how to establish a partition system for the QSPI memory to store the OS image and boot from there, just as it will be done when the final prototype in manufactured.

3.1 QSPI partitioning for bootup

The ZCU102 includes a 128MB QSPI Flash memory that can be programmed with the same files used to boot from the SD Card. The procedure to set up the image to boot from the QSPI Flash was made easier in 2021.2 version onwards but it is also possible in 2020.1:

1. Run `petalinux-config`
2. Under `Subsystem AUTO Hardware settings:`
   - `Flash Settings`: create all partitions as needed and set the size needed for each one. **NOTE:** Set the kernel size to a size higher than your image.ub because that is the partition where it will be flashed.
   - Under `Advanced bootable images storage Settings:`
     - `boot image settings -> images storage media`: set to `primary flash`
     - `kernel image settings -> image storage media`: set to `primary flash`
   - Some adjustments can be made to kernel configuration and features to get a reduced kernel configuration that fits into the QSPI memory.
3. Run `petalinux-config -c u-boot` and set `ARM architecture -> Boot script offset` corresponding to your offset address. The offset address can be found in `components/plnx_workspace/device-tree/device-tree/system-conf.dtsi` under the dts entry for the `bootscr` partition. For the above example configuration the correct offset is `0x1f800000`.
4. Run `petalinux-build`
5. After the build is finished use `petalinux-package` command with the according flags to generate the boot components (assuming console sesion is open in Petalinux project root directory):

```
hyperk0@hyperk0:~$ petalinux-package --boot --force --fsbl --fpga --u-
```

6. Program the flash memory with BOOT.bin, boot.scr and image.ub by located in `/images/linux` directory. Remember to rename all the extensions of the files to `.bin`. Connect the JTAG port to the computer, turn on the board and run the following script, that should be located in the same directory as the three files to be flashed:

   ```
   #!/bin/bash -ex
   default_parameter="-flash_type qspi-x8-dual_parallel -fsbl zynqmp_fsbl.elf
                      -blank_check -url tcp:localhost:3121"
   program_flash -f BOOT.BIN $default_parameter
   program_flash -f boot.scr -offset 0x1f80000 $default_parameter
   program_flash -f image.ub -offset 0x2000000 $default_parameter
   ```

7. Set the boot mode in the board to QSPI mode and switch it on.

3.2 QSPI for storing configuration files

The ZCU102 includes a QSPI Flash that can be used as an storage device mounted in Petalinux. The steps followed here use the jffs2 file system in Petalinux:

1. List the MTD partitions present and select a partition with `cat /proc/mtd`. The listed partitions will be the ones that where previously configured in `petalinux-config -> Subsystem AUTO Hardware settings -> Flash Settings`. By default the BSP project that Xilinx provides has the following partitions:

   ```
   dev:    size   erasesize  name
   mtd0: 01e00000 00002000 "boot"
   mtd1: 00040000 00002000 "bootenv"
   mtd2: 02400000 00002000 "kernel"
   ```

2. Create a directory that will serve as the mounting point of the memory block corresponding to a QSPI flash partition:

   ```
   root@xilinx-zcu102-20222:~# mkdir qspi_flash0
   ```

3. Mount the partition to spi_flash0:

   ```
   root@xilinx-zcu102-20222:~# mount -t jffs2 /dev/mtdblock2 /qspi_flash0
   ```

   mtdblock2 means that the `kernel` partition will be mounted in the folder because it is the block associated with `mtd2`.

4. Once mounted you can `cd` to the folder and create files there. Upon shutdown every file stored there will remain because it is a non-volatile NOR flash device.

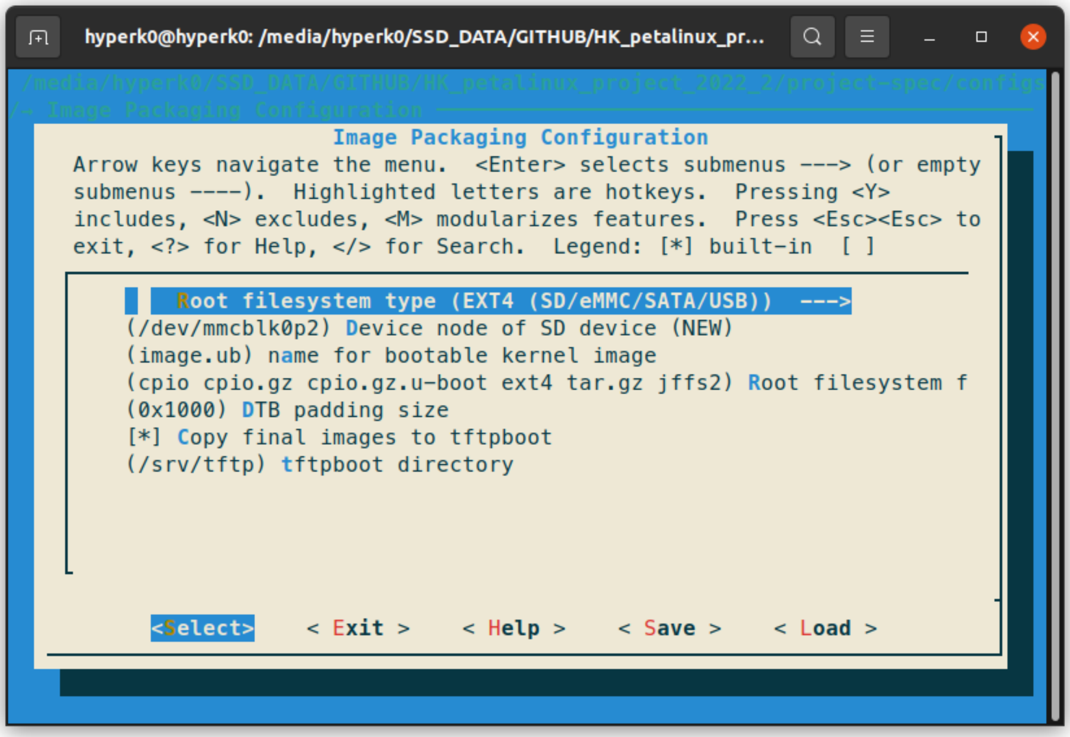
3.3 SD: Root File System in Ext4

Booting from an EXT4 rootfs consists in having your rootfs stored in a non-volatile memory instead of the main memory. This poses the advantage of having a non-volatile rootfs that will not be wiped when the system shuts down which is very useful to save any configuration files or data that needs to stay unchanged between power cycles. Hence, another memory device is needed. There is a lot of alternatives: SD, eMMC, SATA, USB, etc... For this example in the ZCU102, the same SD Card that is used to boot Petalinux will be used, so there needs to be a distribution of the available space between two partitions:

- **Boot partition**: in FAT32 format, holding the files needed for boot: *BOOT.BIN*. *boot.scr* and *Image* This last file is changed, now, *image.ub* is not needed because the rootfs doesn't need to be unpacked in the main memory.
- **Rootfs partition**: it must be formatted in a Linux system with ext4 format. This is the partition that will server as the main "drive" for petalinux.

The procedure for changing the rootfs is very straightfoward:

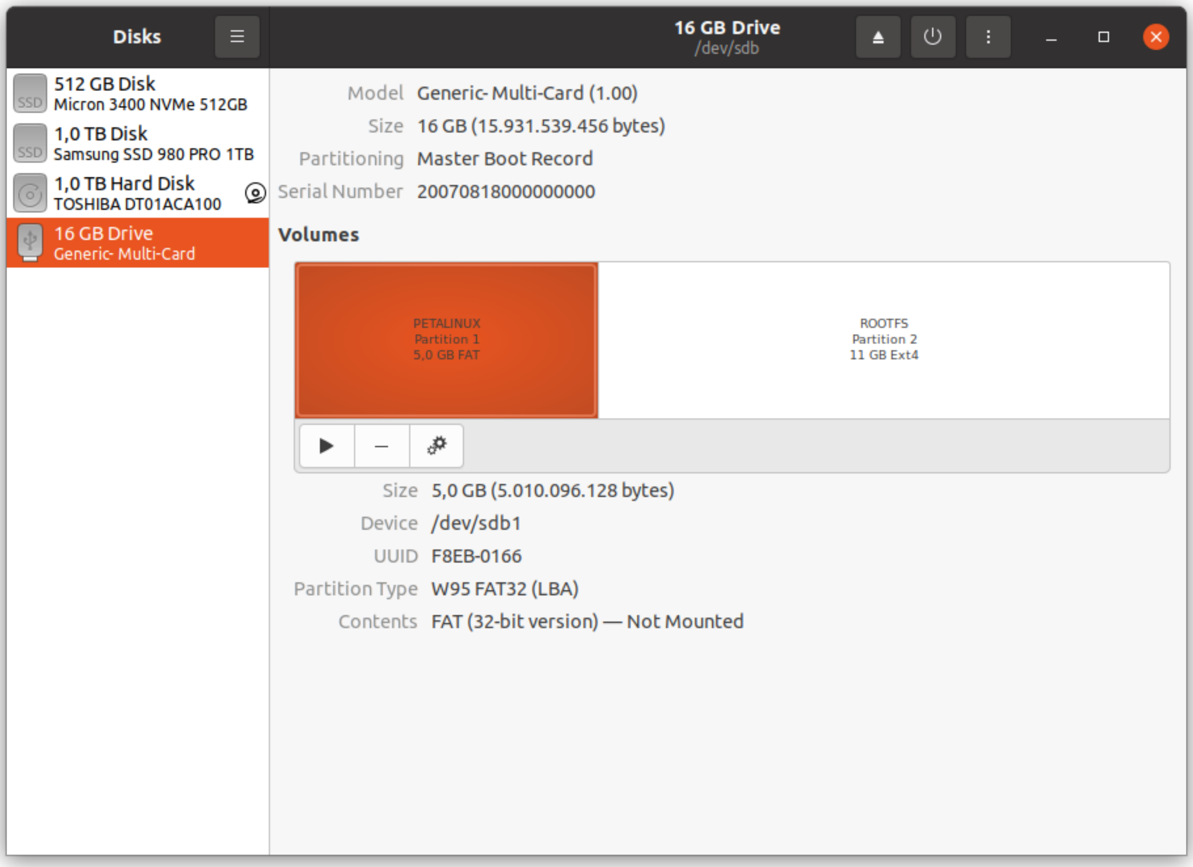
1. Go to `petalinux-config -> Image Packaging Configuration`
2. Go to `Root File System Type` and select EXT4.
3. Now the node must be selected. The node is a name given to each partition of each device to be univocally referred to inside the petalinux file system. This case would be the mmc connected to SD controller 0, so it would begin with *mmcblk0*. The first partition will be for booting and the second for rootfs, so a *p2* (partition 2) is added resulting in *mmcblk0p2* as is shown in figure 3.3.1.


hyperk0@hyperk0: /media/hyperk0/SSD_DATA/GITHUB/HK_petalinux_pr...

```
/media/hyperk0/SSD_DATA/GITHUB/HK_petalinux_project_2022_2/project-spec/configs
/→ Image Packaging Configuration
                      Image Packaging Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

      Root filesystem type (EXT4 (SD/eMMC/SATA/USB))  --->
      (/dev/mmcblk0p2) Device node of SD device (NEW)
      (image.ub) name for bootable kernel image
      (cpio cpio.gz cpio.gz.u-boot ext4 tar.gz jffs2) Root filesystem f
      (0x1000) DTB padding size
      [*] Copy final images to tftpboot
      (/srv/tftp) tftpboot directory

      <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 3.3.1: Selecting EXT4 as the filesystem shows new options in the Image Packaging Configuration menu

4. Save the changes and do `petalinux-build` and `petalinux-package` like in previous sections.
5. Create the two partitions using any Disks management Utility. The final result should be the one depicted in figure 3.3.2.



512 GB Disk
Micron 3400 NVMe 512GB

1,0 TB Disk
Samsung SSD 980 PRO 1TB

1,0 TB Hard Disk
TOSHIBA DT01ACA100

16 GB Drive
Generic- Multi-Card

Model	Generic- Multi-Card (1.00)
Size	16 GB (15.931.539.456 bytes)
Partitioning	Master Boot Record
Serial Number	20070818000000000

Size	5,0 GB (5.010.096.128 bytes)
Device	/dev/sdb1
UUID	F8EB-0166
Partition Type	W95 FAT32 (LBA)
Contents	FAT (32-bit version) — Not Mounted

Figure 3.3.2: Partition structure in the SD Card

6. Unzip the contents of the rootfs.tar.gz file located in `images/linux` to the ROOTFS partition by issuing the following command:

```
hyperk0@hyperk0:~$ sudo tar -xvf images/linux/rootfs.tar.gz
                   -C /media/hyperk0/ROOTFS/
```

This command is very important and cannot be performed through the user interface because it will leave important files without copying to the SD card, causing an error when booting and must be done with sudo because the file *sudoers.d* must be owned by root and it if the rootfs was uncompressed as a common user, the following error would show when trying to do `sudo` in petalinux:

```
sudo:/usr/local/bin/sudo must be owned by uid 0 and have the setuid bit set
```

where uid 0 is the user id corresponding to root and the setuid bit is changed with `chmod` command. However, doing the unzipping of rootfs as `sudo` already avoids this error.

Chapter 4

Adding premade software to the OS (NTP client)

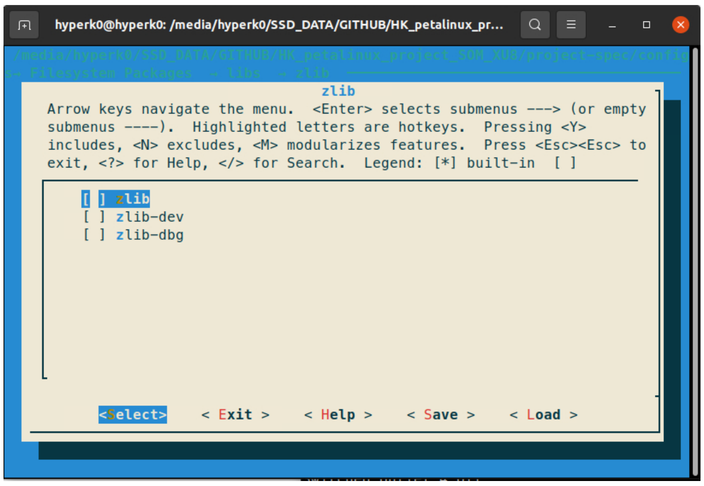
Software packages have a very easy install procedure. In Yocto, the developer can indicate which software packages are needed to be included in the OS image. This can be done in two possible ways:

1. **Use Petalinux Predefined Packages:** By typing `petalinux-config -c rootfs`. The user is presented with the GUI in figure 3.1.2. In this menu there are several sections:

   - `Filesystem Packages` which includes the most common software packages to be added to a Linux system.
   - `Petalinux Package Groups` which are bigger groups made by the developers of the SDK that contain several software packages grouped in the functionalities that they provide to the OS for example network packages or development utilities (commands like `gcc`, `make`,etc...).

   Each one of the packages contained in both of these menus has three different versions: the `standard`, which is the standard software to be executed, `debug`, used for debugging purposes and `dev`, which also includes header files in case that the software packages are needed to be used as libraries to build other applications directly in Petalinux. An example of the menu with *zlib* is shown in figure 3.4.1 .

2. **User defined Packages:** For a much more modular approach, the developer doesn't need to look for specific packages in the aforementioned menus. As explained in Part I, chapter 1, there is a file called `user-rootfsconfig` stored in a project directory: `<path-to-the-project>/project-spec/meta-user/conf` where any software package, even the ones that have not been developed by the user itself, so that they are not in the *meta-user* layer can be included.



```
hyperk0@hyperk0: /media/hyperk0/SSD_DATA/GITHUB/HK_petalinux_pr...
 /media/hyperk0/SSD_DATA/GITHUB/HK_petalinux_project_SOM_XU8/project-spec/config
s→ Filesystem Packages  → libs  → zlib
                              zlib
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

      [ ] zlib
      [ ] zlib-dev
      [ ] zlib-dbg

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 3.4.1: Zlib as an example of package install when using the Filesystem Packages menu

To explain how to install a package, a very common application will be chosen as an example. This application will also be mandatory to be included in the final prototype of the DPB, as it is the Network Time Protocol (NTP) client, which will synchronize time and date with the ones in the DAQ so that timestamps are accurate.

The steps described here serve to install the server side and the client side of a NTP communication:

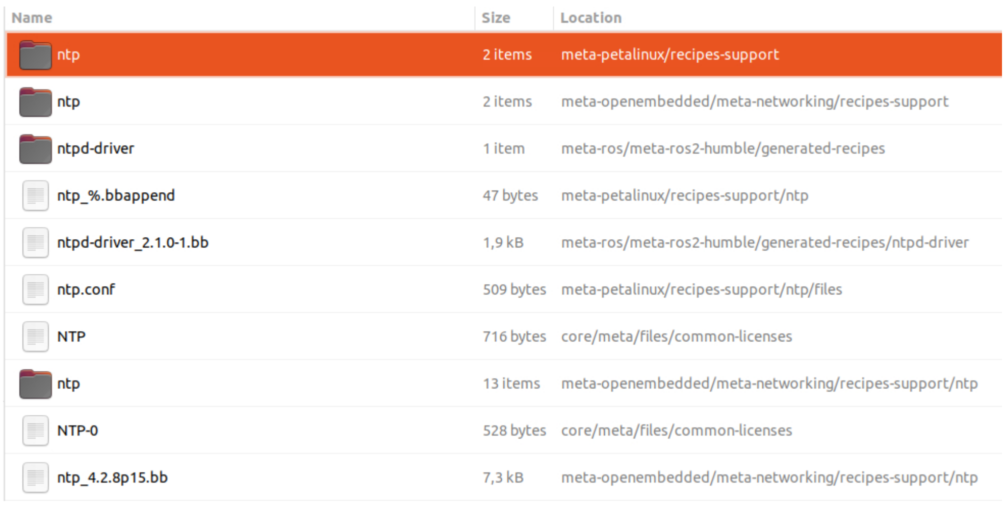
1. Install ntpdate in the machine that will act as server:

```
hyperk0@hyperk0:~$ sudo apt-get install ntpdate
```

2. Add in the hosts file of the client the IP of the server and its name:

```
hyperk0@hyperk0:~$ sudo nano /etc/hosts
```

3. Install NTP Daemon (ntpd) in the client machine. In this case, as it is a Petalinux machine, it needs to be included by adding *CONFIG_ntp* to the `user-rootfsconfig`. To generalize a bit more, if the developer wants to include any software package in the OS image, there is a general procedure they can follow : Yocto saves all the standard software packages recipes inside layers and these layers are represented as folders stored in `<path-to-the-project>/components/yocto/layers`. Doing a quick search in the file explorer with the name of the package the user desires will return a result of a folder with the name structure `<package>` or `<package>_<version>` if the software package exists in any of the layers repositories included in that Yocto release. An example of this with ntp is shown in figure 3.4.2 . So, the developer shall write in the `user-rootfsconfig` file a new line with *CONFIG_<package>*.



Name	Size	Location
ntp	2 items	meta-petalinux/recipes-support
ntp	2 items	meta-openembedded/meta-networking/recipes-support
ntpd-driver	1 item	meta-ros/meta-ros2-humble/generated-recipes
ntp_%.bbappend	47 bytes	meta-petalinux/recipes-support/ntp
ntpd-driver_2.1.0-1.bb	1,9 kB	meta-ros/meta-ros2-humble/generated-recipes/ntpd-driver
ntp.conf	509 bytes	meta-petalinux/recipes-support/ntp/files
NTP	716 bytes	core/meta/files/common-licenses
ntp	13 items	meta-openembedded/meta-networking/recipes-support/ntp
NTP-0	528 bytes	core/meta/files/common-licenses
ntp_4.2.8p15.bb	7,3 kB	meta-openembedded/meta-networking/recipes-support/ntp

Figure 3.4.2: Searching the layers directory for a software package, in this case, ntp

4. Go into `petalinux-config -c rootfs`, select `user packages`, move to the *ntp* option and press Y. An asterisk will appear next to the name of the package, indicated that it will be downloaded and included in the OS image.

5. Search for `ntp.conf` in the following directory: `<path-to-the-project>/components/yocto/layers/meta-petalinux/recipes-support/ntp/files` and add the IP address (or the hostname) of the ntp server. This file will be the `ntp.conf` that will be stored in the client.

6. Open the `/etc/ntp.conf` of the server and add a restrict clause with the IP address or the hostname of the client. You will also need to add ntp servers from your country where you are told to do so:

   ```
   # Use servers from the NTP Pool Project
   server 0.es.pool.ntp.org
   server 1.es.pool.ntp.org
   server 2.es.pool.ntp.org
   server 3.es.pool.ntp.org

   # Local users may interrogate the ntp server more closely
   restrict 127.0.0.1
   restrict ::1
   restrict 10.0.0.2  # ZCU102 IP address
   ```

7. Restart the ntp service to save changes.

   ```
   hyperk0@hyperk0:~$ sudo ntp restart
   ```

8. Build the Petalinux project and boot the OS in the client. To force a synchronization, something possible after adding the `restrict` clause in the server *ntp.conf*, issue the following command:

   ```
   root@xilinx-zcu102-20222:~# ntpd -g
   ```

Chapter 5

Building demo code for data transfer simulation

The data link in the DPB is used to transfer data between two points: the vessels inside the detector and the datacenter (also called DAQ in this project). For this matter, a common framework, that is a set of APIs and functions has been set by the people in charge of the DAQ. This framework is open-source and is stored in GITHUB. Inside this repository in the *mpmt* folder, there is an example of how to code an application that uses this framework to work as the DPB side of communication. To test that this framework is compatible with the Zynq Ultrascale+ architecture, a sample application has been built for the ZCU102.

This section also explains the two procedures to add a custom application together with its recipe for Yocto to build it and add to the rootfs. This is a step up with respect to the previous section, where the NTP application was already built and added to Yocto.

5.1 Using crosscompilation

The first way is to use the SDK itself to add the application as it was done with the NTP server. However, this case requires several steps as the application is not available in Yocto repositories because it was coded specifically for this project, it is not a widely used software package.

Basically, the steps that will be described consists on performing crosscompilation, defined as the act of building a specific software for a target that has a different cpu architecture to the computer that builds the software, also known as host. For this case, the host is a x86 machine and the target is ARM64 so, the libraries used for x86 building will produce a binary for x86 that cannot be run in the ZCU102. Yocto has a specific procedure to use libraries for ARM64 and compile a compatible binary in a x86 machine. Moreover, it can transfer that binary to the rootfs and the application can be run through a simple command.

The steps are as follows:

1. Create the application inside the petalinux project folder. For this example, I have called the application *tbdaq*, which means the Test Bench DAQ.

```
hyperk0@hyperk0:~$ petalinux-create -t apps -n tbdaq
```

2. Go to *petalinux-project>/project-spec/meta-user/recipes-apps/tbdaq* folder. There create a *files* folder if it does not exist.

3. Copy the contents of the mpmt folder in the root of HKMPMTDAQ repository to that *files* folder.

4. Open the Makefile and replace its content with the following.

```
ToolDAQPath=ToolDAQ
ZMQInclude= -I $(ToolDAQPath)/zeromq-4.0.7/include/
CXXFLAGS='-D_GLIBCXX_USE_CXX11_ABI=0'

all:
$(CC) $(LDFLAGS) mpmt/src/*.cpp -O -o tbdaq -lzmq -lboost_date_time
-lboost_serialization -lboost_iostreams -I mpmt/include $(ZMQInclude) -lstdc++
clean:
-rm -f tbdaq *.elf *.gdb *.o
```

The makefile needs to be modified to make it use the Yocto toolchain instead of the host building tools, as this would produce a x86 compatible binary, which is an undesired result. The changes made to the Makefile are as follows:

- DIRECTORY TO BUILD: The mpmt folder contains the Makefile given to the compiler
- The variable *${CC}* must be used in the makefile as the compilation command in order for the app to be compiled in the aarch64-arm ISA.
- If there is an error about the std libraries not being found, just add to the makefile compilation directive `-lstdc++`.
- It is FUNDAMENTAL to add $(LDFLAGS) to the compilation directive because it contains linker flags needed for the packaging and management by petalinux builder like loading a GNU Hash.
- CC VARIABLE VALUE: is the crosscompiler being used. In Petalinux tools: aarch64-xilinx-linux-gcc -march=armv8-a+crc -mtune=cortex-a72.cortex-a53 -fstack-protector-strong -D_FORTIFY_SOURCE=2 -Wformat -Wformat-security -Werror=format-security –sysroot=/home/hyperk0/Xilinx_WORK/xilinx-zcu102-2020.1/build/tmp/work/aarch64-xilinx-linux/tbdaq/1.0-r0/recipe-sysroot

1. Open the recipe file *tbdaq.bb* and copy the following lines:

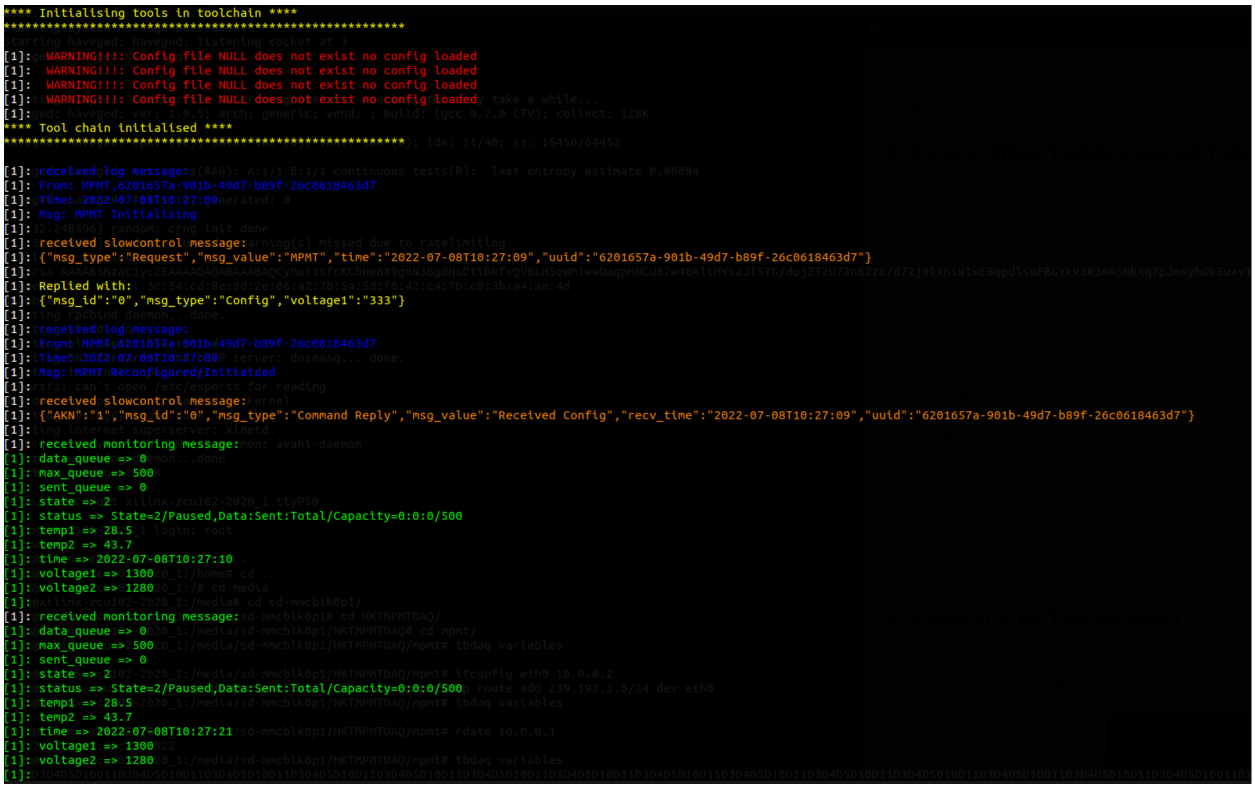
```
# This file is the tbdaq recipe.

SUMMARY = "DAQ Testbench SOC side"
SECTION = "PETALINUX/apps"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;
                    md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://Makefile file://mpmt file://ToolDAQ/zeromq-4.0.7 "

S = "${WORKDIR}"
DEPENDS = " zeromq boost"

do_compile() {
    oe_runmake
}

do_install() {
        install -d ${D}${bindir}
        install -m 0755 ${S}/tbdaq ${D}${bindir}
        install -m 0755 ${S}/mpmt/variables ${D}${bindir}
}
```

The most remarkable line is the DEPENDS variable. It must be added to the .bb file to include the libraries that the app depends on. In this case, it depends on boost and zeromq. The first library provides extended functionality to the C language and the second one manages all the networking that the application needs.

2. The rest of the steps are the same as with NTP, add it to the *user-rootfsconfig*, tick it in the *petalinux-config -c rootfs* and do *petalinux-build*. If the developer is unsure that the application is well coded, the compilation time can be reduced by just compiling the app by using the following command:

3. Once the application is built and the Petalinux system is booted, the application can be run by typing *tbdaq* in the terminal. However, before doing that, the system must be configured so that the app knows which one is the network interface to be used. For that, a static ip route must be set because the application is designed to send all the traffic to that destination ip. This can be done with the following command, where eth0 is the network interface that is connected to the x86 computer running the DAQ side of the communication.

```
hyperk0@hyperk0:~$ ip route add 239.192.1.0/24 dev eth0
```



```
**** Initialising tools in toolchain ****
********************************************************
[1]:  WARNING!!!: Config file NULL does not exist no config loaded
[1]:  WARNING!!!: Config file NULL does not exist no config loaded
[1]:  WARNING!!!: Config file NULL does not exist no config loaded
[1]:  WARNING!!!: Config file NULL does not exist no config loaded
[1]:
**** Tool chain initialised ****
********************************************************

[1]: received log message:
[1]: From: MPMT,6201657a-901b-49d7-b89f-26c0618463d7
[1]: Time: 2022-07-08T10:27:09
[1]: Msg: MPMT Initialising
[1]:
[1]: received slowcontrol message:
[1]: {"msg_type":"Request","msg_value":"MPMT","time":"2022-07-08T10:27:09","uuid":"6201657a-901b-49d7-b89f-26c0618463d7"}
[1]:
[1]: Replied with:
[1]: {"msg_id":"0","msg_type":"Config","voltage1":"333"}
[1]:
[1]: received log message:
[1]: From: MPMT,6201657a-901b-49d7-b89f-26c0618463d7
[1]: Time: 2022-07-08T10:27:09
[1]: Msg: MPMT Reconfigured/Initiatsed
[1]:
[1]: received slowcontrol message:
[1]: {"AKN":"1","msg_id":"0","msg_type":"Command Reply","msg_value":"Received Config","recv_time":"2022-07-08T10:27:09","uuid":"6201657a-901b-49d7-b89f-26c0618463d7"}
[1]:
[1]: received monitoring message:
[1]: data_queue => 0
[1]: max_queue => 500
[1]: sent_queue => 0
[1]: state => 2
[1]: status => State=2/Paused,Data:Sent:Total/Capacity=0:0:0/500
[1]: temp1 => 28.5
[1]: temp2 => 43.7
[1]: time => 2022-07-08T10:27:10
[1]: voltage1 => 1300
[1]: voltage2 => 1280
[1]:
[1]: received monitoring message:
[1]: data_queue => 0
[1]: max_queue => 500
[1]: sent_queue => 0
[1]: state => 2
[1]: status => State=2/Paused,Data:Sent:Total/Capacity=0:0:0/500
[1]: temp1 => 28.5
[1]: temp2 => 43.7
[1]: time => 2022-07-08T10:27:21
[1]: voltage1 => 1300
[1]: voltage2 => 1280
[1]:
```

Figure 3.5.1: DAQ communication working and receiving data to the x86 machine

5.2 Building applications in the evaluation board

Another way that can bypass crosscompilation is compiling the application in the board itself, meaning that the target and the host are the same. Petalinux can also be used as a build environment for applications running on ARM. This means that all the development tools and libraries must be installed in the rootfs. This will produce a huge OS image, 8 times bigger at least than a normal image so it is unfeasible to flash this into the definitive system, it should be used only for development purposes:

1. Add the following packages to the petalinux image:
   - *packagegroup-core-buildessential* located in File System Packages > misc. This contains the *make* command together with all the developer tools to compile C code.
   - zeromq and boost through the *user-rootfsconfig* (CONFIG_zeromq and CONFIG_boost)
   - Enable the *dev-pkgs* in the *petalinuxbsp.conf* file.
   - Point to the lib and the include directory in the Makefile.
   - Copy *zmp.hpp* to the include file through the following command:

   ```
   hyperk0@hyperk0:~$ cp /media/sd-mmcblk0p1/HKTMPMTDAQ/ToolDAQ/zeromq-4.0.7
                        /include/zmq.hpp /usr/include
   ```

2. Build the image and boot Petalinux, including in the image the *mpmt* folder so that it appears in the rootfs. It can also be directly copied to the sd card and access to it directly in Petalinux.
3. Do *make* command in mpmt folder.
4. An *a.out* file will be produced, this is equivalent to the *tbdaq* command when the crosscompilation method is used.

Chapter 6

Redundant boot configuration

This chapter explains the procedure to create some redundancy to boot the system in case the kernel stored in the local memory gets corrupted. 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.

6.1 Solution discussion

As a part of the project, it is very useful to have a backup boot image that it is not stored locally in the board in the case that the memory gets corrupted. The ZynqMP allows several boot modes as stated in previous sections. One of them allows to load the kernel image from a TFTP server while having only the minimal BOOT.bin and the boot.scr files stored locally. This allows to set up the minimal U-Boot environment and prepare for a pxe boot which looks for a server in a specific IP address and downloads and unpacks the image from that server. This is very useful, both for the development process, as the image can be easily replaced without the need of a SD Card, thus allowing remote swapping of the image, without accessing physically to the ZCU102. From the final user perspective it allows to have a backup image that can be easily updated as it is stored in an external server and then being sent at bootup to the board in case the local copy of the kernel image fails in the CRC check.

There are three files that play a role in booting up the system and each one 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 U-boot. 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 [16], chapter 11: Boot and configuration.
- **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 is 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.

6.2 TFTP Server setup

Trivial File Transfer Protocol (TFTP) is a very simple protocol to transfer files from one device to another. U-boot has the needed tools to use this protocol and the Preboot eXecution Environment (PXE) to boot a Linux image from the internet. This requires to have a TFTP server running in a server connected to one of the interfaces of the ZCU102 so As a prerequisite, a TFTP server must be hosted in the server. For this workstation the package `tftpd-hpa` has been installed. To prepare this setup, the following steps were applied:

1. Access the configuration file of the tftp server by doing `sudo nano /etc/default/tftpd-hpa` and specify the directory in *TFTP_DIRECTORY* of the server and the flag –create in *TFTP_OPTIONS*
2. Ensure that the ownership of that folder belongs to the username specified in *TFTP_USERNAME* and that there are write permissions for the rest of the users so that the Petalinux SDK can copy files to the TFTP server.
3. Go to the project folder and type `petalinux-config`
4. Go to *Image Packaging configuration*
5. Tick the *Copy final images to tftp boot* and specify the tftp server directory below as the same value that was used in *TFTP_DIRECTORY*
6. Build the image and package it following the usual instructions, already described.
7. Open the hardware manager in Vivado, by using the GUI or the terminal. To open it through tcl, type the following commands:

```
hyperk0@hyperk0:~$ xilinx_conf
hyperk0@hyperk0:~$ vivado -mode tcl
```

8. Turn on the board, open the serial terminal and type

```
hyperk0@hyperk0:~$ petalinux-boot --jtag --prebuilt 2
--hw_server-url TCP:localhost:3121
```

9. When the line, *Press any key to stop autoboot shows*, press any key to enter into the U-Boot terminal.
10. Now, it is needed to configure the environment to do a network boot.

```
Zynq-MP> setenv ipaddr 10.0.0.2
Zynq-MP> setenv serverip 10.0.0.1
Zynq-MP> pxe get
```

After the last command, a configuration file should have been downloaded to the board.

11. Finally, do `pxe boot` and the system should first download three files and after a CRC check, boot up:

    - *rootfs.cpio* containing the root file system in offset *0x2100000*
    - *Image* containing the kernel image in offset *0x18000000*
    - *system.dtb* containing a flattened device tree in offset *0x40000000*

This procedure didn't come out without issues. After doing step 11 and downloading the three files to the RAM, the system didn't boot up, throwing the following error message:

```
ERROR: Did not find a cmdline Flattened Device Tree
```

After investigating thoroughly, it was discovered that when flashing to the RAM the three files, it was done using fixed offset addresses so that, after the flash, the boot script could find them to boot the system up. This fact constrained the size of the three files because if one of the files was bigger than the offset left for the next file then, it would be overwritten with the next file to be written. This happened between the *Image* and the *system.dtb* files because the offset of the image was *0x18000000* and the device tree was *0x40000000*. If these to numbers are subtracted, a space of *0x28000000*, which correspond to 671 MB (671 088 640 bytes). The image was bigger than that, so the system could not boot up because of the image being partialy overlaped with the device tree. This image is that big because the file system type was *initramfs*, which includes inside the image file the kernel and the rootfs. However, in this type of boot this is not needed, as the rootfs is flashed separately. Then using an *initrd*, which only includes the kernel in the image file reduced the size to 20MB, leaving the rootfs with the rest of the size. Fortunately, the offset of the image allows the rootfs to weigh up to 1GB, so there is no problem there.

6.3 Coding U-boot script for autoboot

Using the JTAG custom script explained in Part III, chapter 2, the boot.scr script can be loaded without having to load the whole kernel image and rootfs. That part would be loaded through TFTP and then booted through pxe.

From Petalinux 2020.1 onwards, the default boot method is executing an u-boot script that is previously loaded in the main memory at address by default *0x20000000*. This script by default contains the boot functions to execute if the user does a JTAG, QSPI or SD Card boot. However, for this project, a fast remote way to load the image is also wanted to serve as a backup in case the local flash memory fails. This has already been explained in the TFTP boot section but the changes need to be reflected in the script so that the boot process is automatic. The steps to edit the script are as follows:

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 do

   ```
   hyperk0@hyperk0:~$ petalinux-build -c -u-boot -x cleansstate
   hyperk0@hyperk0:~$ 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.

An example for a u-boot script that uses the *pxe boot* command can be found in Annex 1. Notice the following parts:

- *boot_targets* variable at the beginning, with both qspi and pxe on it. This variable is a default environment variable from u-boot that contains a list of all the possible boot methods ordered so that the first one is the one with highest priority and so on. Here, only two possible boot method are desired so *qspi* and *pxe*.
- *The for loop:* it runs through all the boot_targets array picking each string and comparing it to all the possibilities. For example the first string in the array is *qspi0* which means that it will enter into the if clause that matches with this name and read from the QSPI flash in the established memory addresses.
- An *if* clause with the instructions for pxe is added to set the environment variable that prevents overlapping of the device tree file with the FIT image and the ip addresses of the board and the TFTP server to enable `pxe boot`. The `pxe get` command is inside an if because if it is successful in downloading a .cfg file it returns a 1 and that means that `pxe boot` is possible.

Chapter 7

Changing init behaviour (connect to SMB drive)

Another aspect of the Linux kernel that could be very useful to the project is the *init*, which is the initial configuration done in the OS just after booting up the kernel and before transferring control to the user. This can be used to configure storage, applications and interfaces automatically on boot. In order to know how init behaviour can be changed, this chapter presents an example of how a Server Message Block (SMB) client can be automatically configured in the ZCU102 board to connect to a SMB server located in the host computer.

7.1 PC Setup

1. `sudo nano /etc/samba/smb.conf`
2. Configure the interfaces by uncommenting interfaces and assigning enp4s0.
3. Configure the shared folder as by adding this section to the .conf file:

```
[HKShare]
  comment = Shared folder for ZCU102
  path = /home/hyperk0/ZCU102FILES
  read only = no
  writable = yes
  guest ok = yes
```

4. /etc/init.d/smbd restart # Restart Samba server to apply changes
5. Open the following firewall ports:

```
sudo ufw allow 137/udp
sudo ufw allow 138/udp
sudo ufw allow 139/tcp
sudo ufw allow 445/tcp
```

6. Check the state of the Samba server:

```
hyperk0@hyperk0:~$ sudo systemctl status smbd
```

7.2 Petalinux setup

In the petalinux image you need to set the following options to enable SMB support in the OS:

1. Go to kernel configuration:

```
hyperk0@hyperk0:~$ petalinux-build -c kernel
```

2. Go to `File Systems -> Network File Systems` and enable `SMB3 and CIFS support`

3. Create a new application called *myapp-init*.

```
hyperk0@hyperk0:~$ petalinux-create -t apps --template install -n myapp-init
```

4. In this step, depending on the init system available in your OS, you must follow one of the two guides:

   - For Systemv (Petalinux 2020.1): This system is based on the execution of script at several boot stages. To create a init script that uses systemv do the following:

     (a) Copy the following recipe instructions in the .bb file. It installs two files: the myapp-init script inside the init.d directory so that Linux executes it when it boots.

```
# This file is the myapp-init recipe.
SUMMARY = "Simple myapp-init application"
SECTION = "PETALINUX/apps"
LICENSE = "MIT"

SRC_URI = "file://myapp-init"
S = "${WORKDIR}"
FILESEXTRAPATHS:prepend = "${THISDIR}/files:"
inherit update-rc.d
INITSCRIPT_NAME = "myapp-init"
INITSCRIPT_PARAMS = "start 99 S ."

do_install() {
    install -d ${D}${sysconfdir}/init.d
    install -m 0755 ${S}/myapp-init ${D}${sysconfdir}/init.d/myapp-init
}
FILES:${PN} += "${sysconfdir}/*"
FILES:${PN} += "/home/*"
```

     (b) Copy inside the files folder the bash script that will be executed. This script can also be a Daemon with its *start()*, *stop()* and *restart()* functions but it is simpler to use a script named *myapp-init*. This script creates a folder and mounts the 10.0.0.1 SMB server inside that folder. The server is password protected so a password must be given in plain text or in SHA format.

```
#!/bin/sh
# Mount PC Shared Folder
mkdir /mnt/SharedFolder
sudo mount.cifs //10.0.0.1/HKShare /mnt/SharedFolder -o password=*******
```

- For Systemd (Petalinux 2022.2): This system is based on the creation of services with *.services* files. The steps to add them to the rootfs are as follows:
  (a) Copy the following recipe instructions in the .bb file. It installs

```
# This file is the myapp-init recipe.
SUMMARY = "Simple myapp-init application"
SECTION = "PETALINUX/apps"
LICENSE = "MIT"

SRC_URI = "file://myapp-init \
        file://myapp-init.service"
S = "${WORKDIR}"
FILESEXTRAPATHS:prepend = "${THISDIR}/files:"
inherit update-rc.d systemd
INITSCRIPT_NAME = "myapp-init"
INITSCRIPT_PARAMS = "start 99 S ."
SYSTEMD_PACKAGES = "${PN}"
# Enables Systemd service called myapp-init
SYSTEMD_SERVICE:${PN} = "myapp-init.service"
SYSTEMD_AUTO_ENABLE:${PN} = "enable"

# The first if checks which system init is enabled and if it is sysvinit,
# then it proceeds with the old method
do_install() {
    if ${@bb.utils.contains('DISTRO_FEATURES', 'sysvinit', 'true', 'false', d)};
        then
        install -d ${D}${sysconfdir}/init.d/
        install -m 0755 ${WORKDIR}/myapp-init ${D}${sysconfdir}/init.d/
    fi
    install -d ${D}${bindir}
    install -m 0755 ${WORKDIR}/myapp-init ${D}${bindir}/
    install -d ${D}${systemd_system_unitdir}
    install -m 0644 ${WORKDIR}/myapp-init.service ${D}${systemd_system_unitdir}
}
FILES:${PN} += "${sysconfdir}/*"
FILES:${PN}+="${@bb.utils.contains('DISTRO_FEATURES','sysvinit',
                '${sysconfdir}/*', '', d)}"
```

  (b) Copy the following *.service* file in the *files* folder of *myapp-init*. It creates the mounting service taking as input parameter the same script as the one with Systemv.

```
[Unit]
Description=StartScript
[Service]
ExecStart=/usr/bin/myapp-init
StandardOutput=journal+console
[Install]
WantedBy=multi-user.target
```

5. Boot the image after following the usual image building instructions.

6. Go to the */mnt/SharedFolder* directory. That folder is the mount point configured in the script (systemv) or service (systemd) and every change made there should also be reflected in the SMB server or vice versa.

Chapter 8

SPI communication through integrated controller

This chapter covers the basic on how to enable one of the modules included in the PS for slow control, that is for sensor monitoring because in the DPB there will be a lot of sensors to be read in order to monitor the status of the system at all times. This chapter shows an example on how to enable the integrated SPI controller to read a sensor.

For this example, the SPI controller inside the MPSOC has been configured to communicate with a magnetometer connected to the PMOD connector shown in figure 3.8.1. The ad-hoc board connected to the PMOD is the Steval MKI137v1 [39] with the LIS3MDL [40] magnetometer integrated.

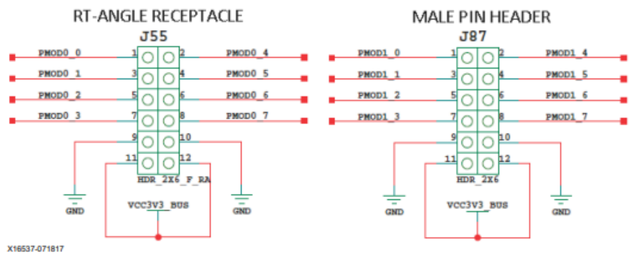


Figure 3.8.1: PMOD headers in the ZCU102 [19]

8.1 Enabling PS SPI controller

First, the PS SPI controller needs to be enabled. This is very simple. Just by going to Vivado to the Zynq UltraScale+ MPSoC Processing System IP block and double clicking on it, the internal block diagram in figure 2.2.8 will be displayed. Here, the user can find two controllers, SPI0 and SPI1. Enable one of them and select the EMIO option as the pins that the sensor will be connected to will be routed to the PL.

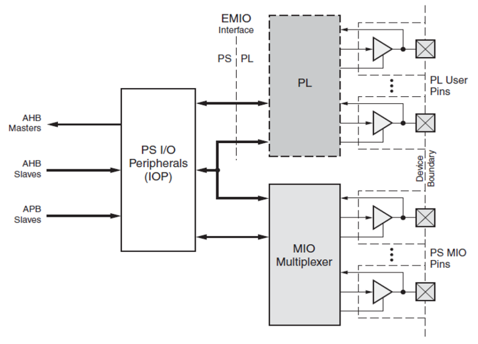


Figure 3.8.2: MIO and EMIO multiplexing for PS or PL routing respectively

In this architecture, all the controllers integrated on the PS have the option of MIO or EMIO. This is done to give the user more flexibility as MIO pins are fixed pins routed directly to the PS that cannot have any instantiated logic as they are not related to the PL in any way. However, if the user wants to instantiate IP cores or directly use peripherals that, by the design of the board are routed to the PL, the EMIO option exists. This is possible through the use of "wires" that interconnect the PS and the PL. The result of selecting EMIO for the SPI controller is that it will generate as two new outputs of the block a MISO, MOSI, a SCL and a SS (Slave Select), the four usual signals in any SPI communication.

8.2 Project configuration

The project configuration is done in two phases:

- **Vivado:** Once the SPI controller has been configured as EMIO, the four generated ports must be routed outside the PL. Using Vivado I/O ports assignments or directly the *xdc* constraint file, the pins assigned to the PMOD port of the ZCU102 have been configured following the table in page 75 of the ZCU102 reference guide [19].
- **Petalinux:** For SPI to work, the SPI driver must be enabled. This configuration is already done by default in the ZCU102 as well as the corresponding device tree node is already added in the *zynqmp.dtsi*. This device tree file is common for every ZynqMP device and contains the device tree node for both *spi0* and *spi1*. This file is available in the official Xilinx Github repository.

8.3 Test with SPI sensor

For this demonstration, a test tool called *spidev_test.c* already included in the Linux kernel has been used to send data to the SPI device in order to read the register highlighted in the register address map of figure 3.8.3.

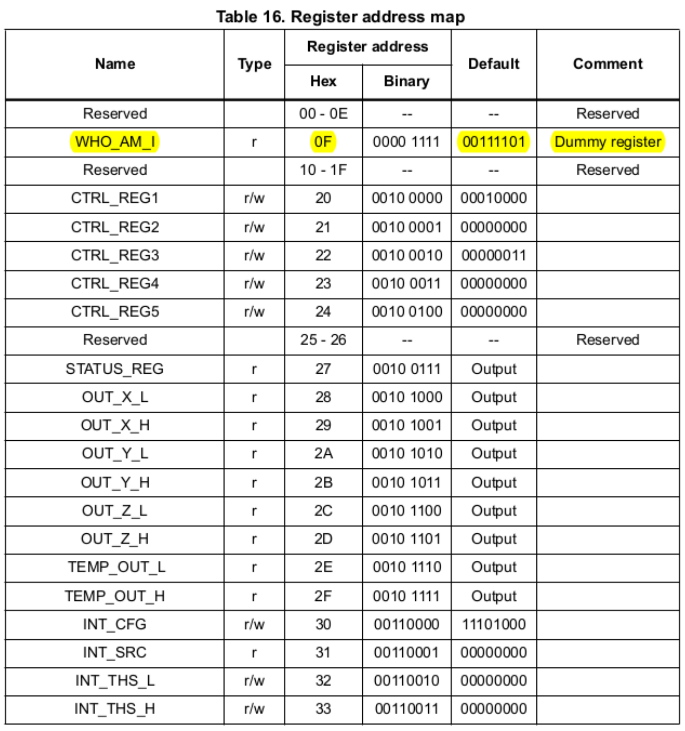


Table 16. Register address map

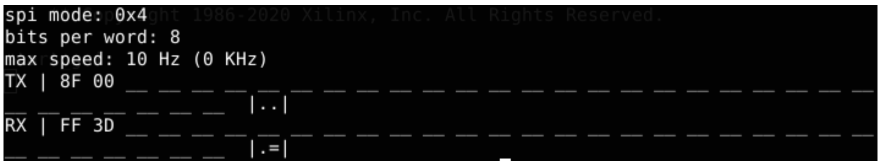
Name	Type	Register address		Default	Comment
		Hex	Binary		
Reserved		00 - 0E	--	--	Reserved
WHO_AM_I	r	0F	0000 1111	00111101	Dummy register
Reserved		10 - 1F	--	--	Reserved
CTRL_REG1	r/w	20	0010 0000	00010000	
CTRL_REG2	r/w	21	0010 0001	00000000	
CTRL_REG3	r/w	22	0010 0010	00000011	
CTRL_REG4	r/w	23	0010 0011	00000000	
CTRL_REG5	r/w	24	0010 0100	00000000	
Reserved		25 - 26	--	--	Reserved
STATUS_REG	r	27	0010 0111	Output	
OUT_X_L	r	28	0010 1000	Output	
OUT_X_H	r	29	0010 1001	Output	
OUT_Y_L	r	2A	0010 1010	Output	
OUT_Y_H	r	2B	0010 1011	Output	
OUT_Z_L	r	2C	0010 1100	Output	
OUT_Z_H	r	2D	0010 1101	Output	
TEMP_OUT_L	r	2E	0010 1110	Output	
TEMP_OUT_H	r	2F	0010 1111	Output	
INT_CFG	r/w	30	00110000	11101000	
INT_SRC	r	31	00110001	00000000	
INT_THS_L	r/w	32	00110010	00000000	
INT_THS_H	r/w	33	00110011	00000000	

Figure 3.8.3: Register map for the LIS3MDL magnetometer

Once the image was compiled and the board was booted the test was configured. By looking in the datasheet for the register table and the chronograph that shows the order in which the bits must be sent for a SPI read command, taking into account a 10 Hz speed, the command to be issued is the following:

```
root@xilinx-zcu102-20222:~# ./spidev_test -D /dev/spidev1.0
                            --speed 10 --cs-high -v -p "x8Fx00"
```

This reads the WHO_AM_I register which should have a fixed value of *00111101* = *0x3D*. This is the case as shown in the terminal in figure 3.8.4.



```
spi mode: 0x4ight 1986-2020 Xilinx, Inc. All Rights Reserved.
bits per word: 8
max speed: 10 Hz (0 KHz)
TX | 8F 00 __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __
__ __ __ __ __ __ __ __  |..|
RX | FF 3D __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __
__ __ __ __ __ __ __ __  |.=|
```

Figure 3.8.4: Terminal image after executing the command of spidev_test

To give more proof an ILA core was also instantiated connected to the MISO, MOSI, SCL and SS pins. Figure 3.8.5 shows the sequence in which the bits arrive. First the read request is sent through *io0_o* (MOSI) to the SPI slave. This takes 8 SPI CLK cycles to send the 8 bits in a 10 Hz clock. Then the data arrives to *io1_i* which is the MISO in this SPI communication. The bits here are received sequentially from the most significant to the less significant one, resulting in 8 bits stored in the WHO_AM_I register *0x3D* between the *66,700* and *67,6000* time intervals.

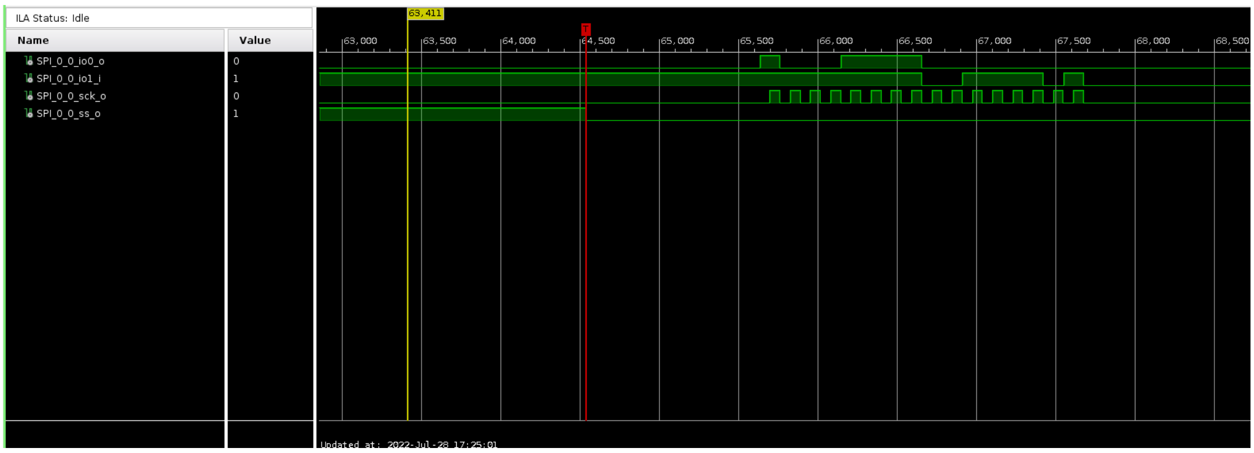


Figure 3.8.5: Logic Analyzer showing the SPI signals gathered in the FPGA

Chapter 9

Extra Network Interfaces

The data link in the DPB will use standard TCP/IP connections. In the final design there will be up to 4 data links so 4 physical network interfaces must be configured to work in Petalinux. This chapter covers the basics on how to use one of the Xilinx reference designs to instantiate an Ethernet IP Core that is connected to the SFP cage of the ZCU102 so that a fiber optic link can be established between a standard PC and the board. This is the perfect starting point for one data link in the DPB as the hardware in the final design will be very similar both in the DPB side, which is represented here by the ZCU102 and the DAQ side, represented by the computer whose specs were defined in Part II, chapter 1.

9.1 Xilinx Reference Designs

The aim of this chapter is to test the compatibility of the SFP modules with the board and with the OS itself and how easy it is to add more network interfaces through these SFP modules.

The IP Core being used here is the *1G/2.5G Ethernet PCS/PMA or SGMII LogiCORE IP* [41] which interconnects the Gigabit Ethernet MAC (GEM) with the SFP module acting as an interface so that the GEM can configure the SFP. As with any IP Core, this one tries to resemble to some hardware that could be perfectly implemented in an ASIC. In this case this is a PHYsical layer chip or PHY for short. This ASIC is in charge of managing the physical layer in the OSI protocol stack with functions like signalisation, auto-negotiation, carrier generation, among other things.

Luckily, Xilinx has an ample number of reference designs for its IP Cores and the Ethernet Core is no different, with a Github repository called ZCU102-Ethernet that has the following examples:

- *ps_emio_eth_1g:* PS 1000BASE-X design utilizing the GEM (Gigabit Ethernet MAC) over EMIO to a 1G/2.5G Ethernet PCS/PMA or SGMII IP.
- *pl_eth_1g:* PL 1000BASE-X design utilizing the AXI Ethernet 1G/2.5G Subsystem.
- *pl_eth_sgmii:* PL SGMII design utilizing the AXI Ethernet 1G/2.5G Subsystem.
- *ps_emio_eth_sgmii:* PS SGMII design utilizing the GEM over EMIO to a 1G/2.5G Ethernet PCS/PMA or SGMII IP.

For this chapter, the *ps_emio_eth_1g* and the *ps_emio_eth_sgmii* are the ones being used as they are the ones that use the *1G/2.5G Ethernet PCS/PMA or SGMII LogiCORE IP* in its two variants: Serial Gigabit Media-Independent Interface (SGMII), which is media independent, meaning it can be used with copper twisted pair and 1000Base-X which is for fiber optics links only.

9.2 SGMII SFP Modules test

Using the *ps_emio_eth_sgmii* project, that is, using the GEM3 of the PS, connected to the 1G/2.5G Ethernet PCS/PMA or SGMII IP as an interface with the SFP, several SFP modules have been tested:

- **Avago ACBU-5700RZ SFP module:** it was the one available at the beggining of the project. However, due to its age it doesn't offer compatibility with the SGMII protocol (see the Mouser EOL document for that module). BASE-X cannot be used due to the difference in the signalization process between a copper pair and an optic fiber. If BASE-T is the desired choice, then SGMII must be used as it is a media independent interface.
- **Finisar FCLF8522P2BTL:** This RJ45 SFP supports SGMII, so it should work out of the box. However, this proved to be impossible to do because when the system boots up it fails to do autonegotiation from the SFP side, even though the IP core states it to be completed through the status LEDs. The SFP needs to be unplugged and plugged again or connect it after the OS has booted up. however, before reconnecting it, you need to do ethtool and disable autonegotiation so that it gets a fixed advertised speed to negotiate with the other side.
- **Mikrotik S-RJ01:** Another SFP to RJ45 module that also supports SGMII. This one worked out of the box, did autonegotiation with the other host and established a link at system boot without the need to issue any command.

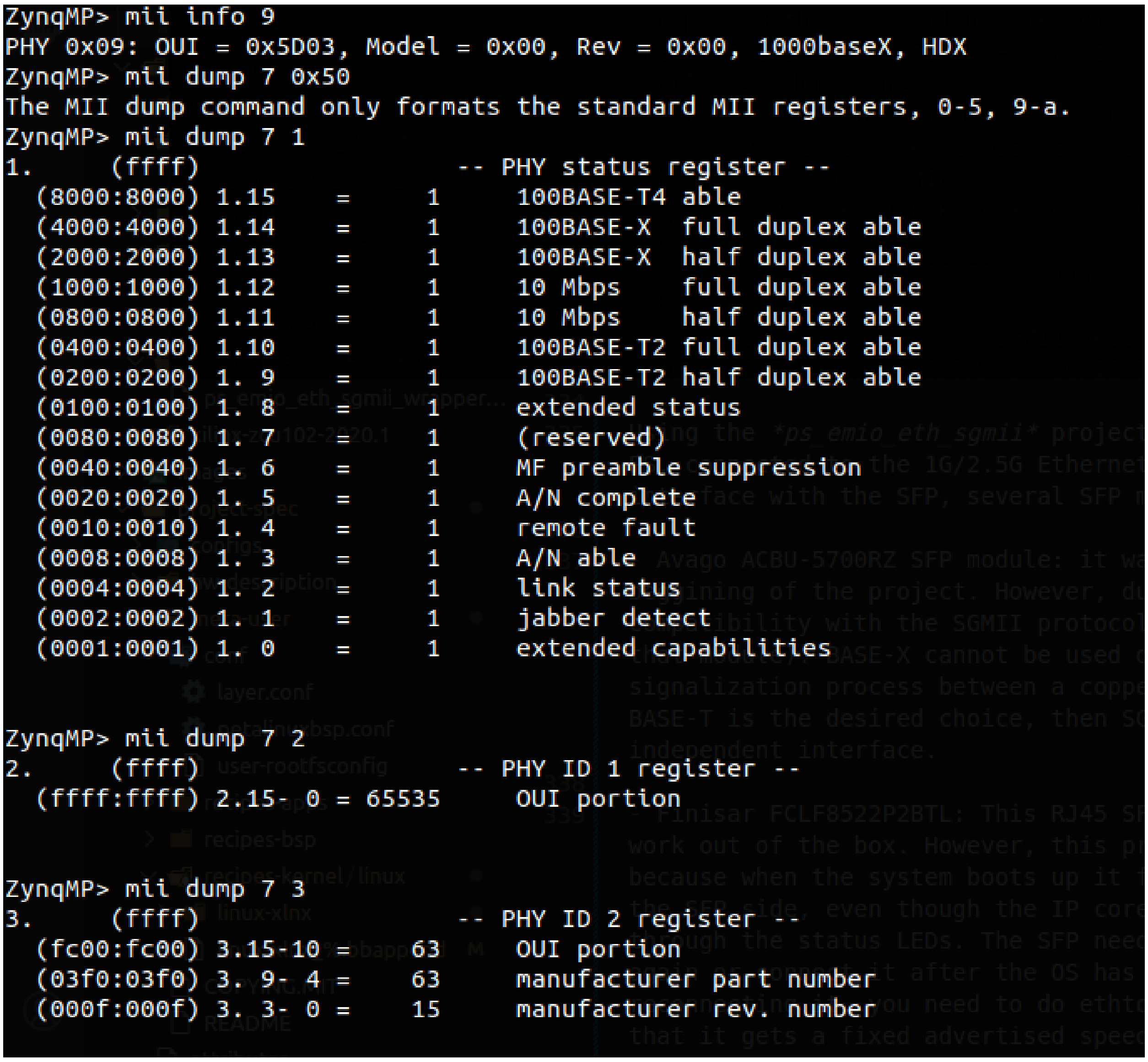
To investigate this difference in behaviour between the Finisar and the Mikrotik SFP, there are some ways to communicate with the module by using the 2 wire serial interface compliant with i2C protocol:

- The first way is, before loading the kernel image, one can access to the **U-Boot console**, that is, the first stage boot loader console, before the OS loads. This allows to use the *mii* command. This command can extract information from the Media Independent Interfaces connected to the PS. In this case there is only one mii connected: the SFP one which is allocated in address 9, (this should also be indicated in the device tree).

  Using the following U-boot command it is possible to read the register map with an output that gives the value and the meaning of each group of bits in an ordered list:

  ```
  ZynqMP> mii dump 9 register_number
  ```

  However, this command doesn't get the state of the physical PHY chip in the SFP, but a virtual one that the IP in the PL creates. This can be deduced because of the retrieved Organizationally Unique Identifier (OUI). The OUI is 00-5D-03, which doesn't belong to Mikrotik or Finisar, but to Xilinx. Then what the mii command is detecting is the IP core. This is normal, as the GEM output is GMII and is the IP core the one in charge of translating SGMII instructions to GMII so that the PS GEM can work. Figure 3.9.1 shows proof of the `mii` command returning the OUI of Xilinx.



```
ZynqMP> mii info 9
PHY 0x09: OUI = 0x5D03, Model = 0x00, Rev = 0x00, 1000baseX, HDX
ZynqMP> mii dump 7 0x50
The MII dump command only formats the standard MII registers, 0-5, 9-a.
ZynqMP> mii dump 7 1
1.      (ffff)                 -- PHY status register --
  (8000:8000) 1.15    =     1     100BASE-T4 able
  (4000:4000) 1.14    =     1     100BASE-X  full duplex able
  (2000:2000) 1.13    =     1     100BASE-X  half duplex able
  (1000:1000) 1.12    =     1     10 Mbps    full duplex able
  (0800:0800) 1.11    =     1     10 Mbps    half duplex able
  (0400:0400) 1.10    =     1     100BASE-T2 full duplex able
  (0200:0200) 1. 9    =     1     100BASE-T2 half duplex able
  (0100:0100) 1. 8    =     1     extended status
  (0080:0080) 1. 7    =     1     (reserved)
  (0040:0040) 1. 6    =     1     MF preamble suppression
  (0020:0020) 1. 5    =     1     A/N complete
  (0010:0010) 1. 4    =     1     remote fault
  (0008:0008) 1. 3    =     1     A/N able
  (0004:0004) 1. 2    =     1     link status
  (0002:0002) 1. 1    =     1     jabber detect
  (0001:0001) 1. 0    =     1     extended capabilities

ZynqMP> mii dump 7 2
2.      (ffff)                 -- PHY ID 1 register --
  (ffff:ffff) 2.15- 0 = 65535     OUI portion

ZynqMP> mii dump 7 3
3.      (ffff)                 -- PHY ID 2 register --
  (fc00:fc00) 3.15-10 =    63     OUI portion
  (03f0:03f0) 3. 9- 4 =    63     manufacturer part number
  (000f:000f) 3. 3- 0 =    15     manufacturer rev. number
```

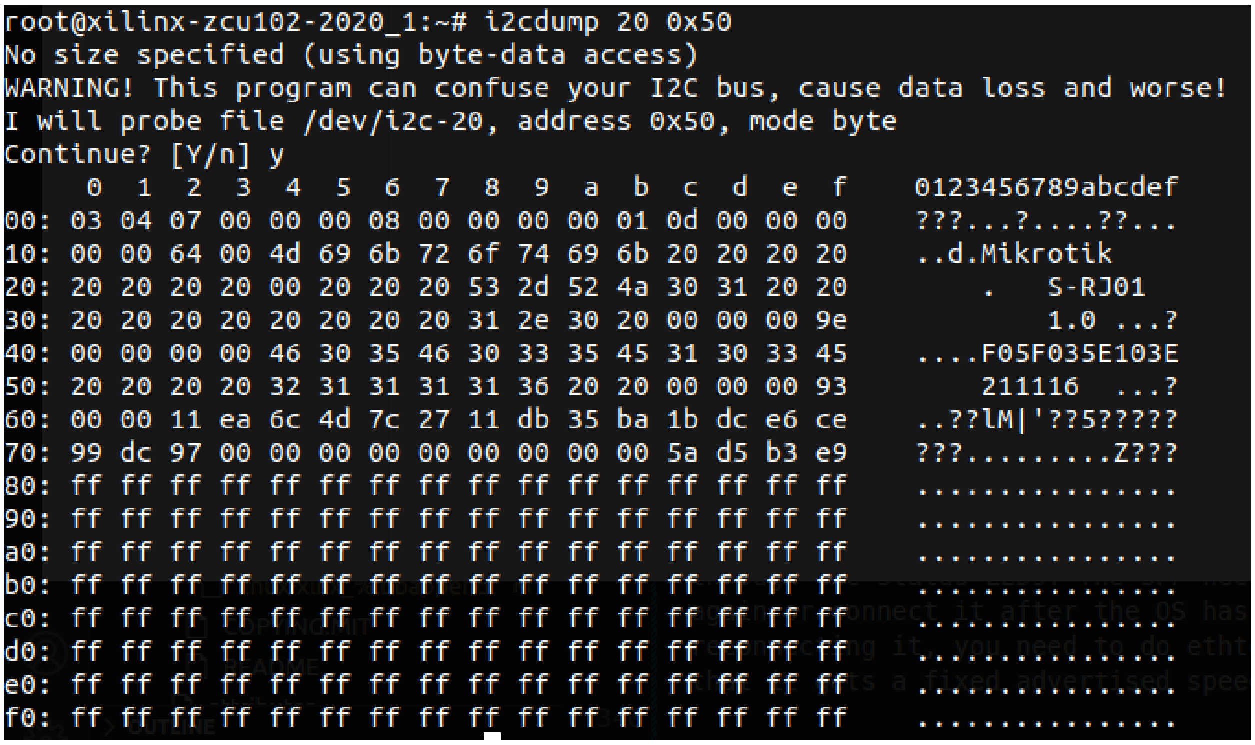
Figure 3.9.1: Uboot terminal showing mii dump command that gets the value of registers in Xilinx IP

- The second way is by directly using **i2ctools**, a set of commands in linux that allows to detect the i2c buses present in the system and read and write on the devices connected to them. To achieve this, the following steps must be followed:

  1. Issue *i2cdetect -l* command to list the i2c buses available in the system. The corresponding bus to SFP1 should have the SFP device connected to address 0x50 as stated by the ZCU102 guide. The bus that fulfilled this condition was bus 20 because when *i2cdetect 20* was used, it detected a device in address 0x50.
  2. Issue *i2cdump 20 0x50* command. This command reads the PHY Address 0xAC, standard to all SFP modules, which is a part of the chip where configuration can be written. Here SGMII mode can be enabled, advertise different speed and modes and do the software reset to apply changes. Everything should be done automatically through the IP core so that the user needn't touch anything.
  3. *i2cset* command can be used to modify the settings but it is not recommended unless the IP wasn't writing the configuration well. With this method, the physical PHY chip in the SFP module could be configured because it is directly connected through i2c. This could be

checked because in the register it is also stored in ASCII character the name of the manufacturer (Mikrotik for instance).

Figure 3.9.2 shows the result of the `i2cdump` command. As expected, the ASCII translation of the data stored in several registers forms the word *Mikrotik S-RJ01* which means that the configuration being read belongs to the connected SFP.



```
root@xilinx-zcu102-2020_1:~# i2cdump 20 0x50
No size specified (using byte-data access)
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-20, address 0x50, mode byte
Continue? [Y/n] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 03 04 07 00 00 00 08 00 00 00 00 01 0d 00 00 00    ???...?....??...
10: 00 00 64 00 4d 69 6b 72 6f 74 69 6b 20 20 20 20    ..d.Mikrotik
20: 20 20 20 20 00 20 20 20 53 2d 52 4a 30 31 20 20        .   S-RJ01
30: 20 20 20 20 20 20 20 20 31 2e 30 20 00 00 00 9e            1.0 ...?
40: 00 00 00 00 46 30 35 46 30 33 35 45 31 30 33 45    ....F05F035E103E
50: 20 20 20 20 32 31 31 31 31 36 20 20 00 00 00 93        211116  ...?
60: 00 00 11 ea 6c 4d 7c 27 11 db 35 ba 1b dc e6 ce    ..??lM|'??5?????
70: 99 dc 97 00 00 00 00 00 00 00 00 00 5a d5 b3 e9    ???.........Z???
80: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
90: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
a0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
b0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
c0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
d0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
e0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
f0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
```

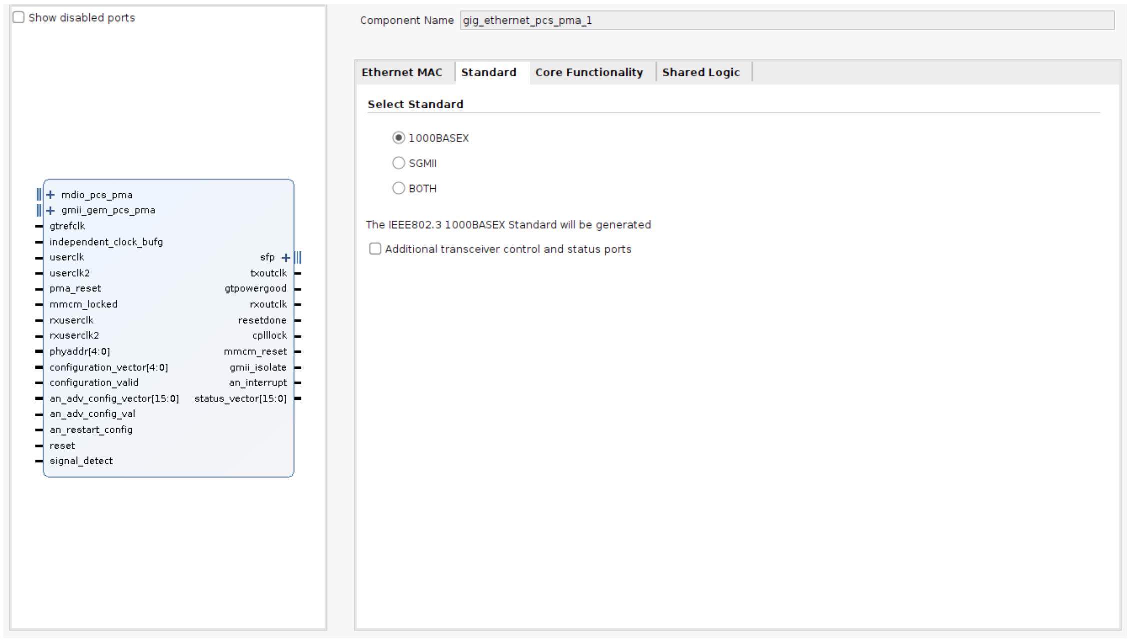
Figure 3.9.2: I2C dump of the PHY address visible through the 2-wire serial terminal

The main difference between the two methods is the protocol they use to read the registers. While the first one uses the MDIO interface, which is the standard protocol for the PHY chips, the second one uses I2C. In this case this is a crucial difference as the MDIO interface associated to GEM0 is connected to the Xilinx IP Core and thus, the configuration read is the one of the core instantiated in the PL. However, as the ZCU102 Manual [19] reads, the SFP I2C interfaces are connected directly to the hard I2C controllers located in the PS, so when using *i2ctools*, the information being read belongs directly to the SFP module, bypassing the Xilinx IP Core.

9.3 1000BaseX SFP Module

In this section a Avago AFBR-5715ALZ is used. This module, which is very similar to the one used in the final design of the DPB, does not include a SGMII SFP but optical ones, using BaseX. This IP core also supports optical SFP, the steps to follow if a change from SGMII to 1000BaseX is desired are only two:

- In **Vivado**, change the IP core settings to use 1000Base-X. This has been done by using the example project *ps_emio_eth_1g* provided in the same repository as the previous one. Figure 3.9.3 shows the GUI for configuring the Standard used in the IP Core.



☐ Show disabled ports

Component Name gig_ethernet_pcs_pma_1

Select Standard

[✓ 1000BASEX]
SGMII
BOTH

The IEEE802.3 1000BASEX Standard will be generated

☐ Additional transceiver control and status ports

Figure 3.9.3: IP Core configuration to 1000BaseX

- In **Petalinux SDK**, open the device tree configuration (*system-user.dtsi*) and change the 0x04 in the <phy-type> variable to 0x05. This indicates the driver that the core IP is configured to be used as 1000Base-X so that it interprets the auto negotiation well. It works with any of these two values but with 0x04, the link needs to be configured manually after booting.

Figure 3.9.4 shows the ZCU102 with the optical SFP mounted in the SFP cage and a OM3 cable connected to it. IP address assignment has been done manually in Petalinux assigning 20.0.0.1 to the Realtek Card in the PC and 20.0.0.2 to the ZCU102. A ping and a iperf3 test has been done and it worked correctly with a peak 1 Gbps speed which is to be expected.

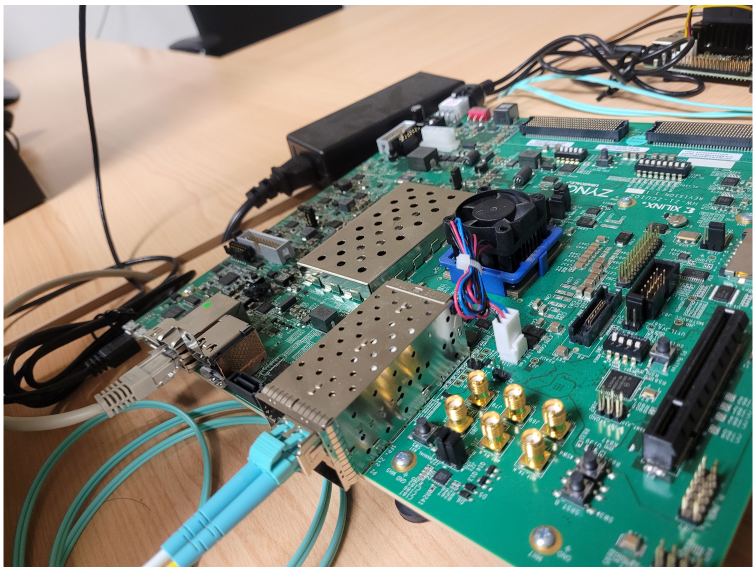


Figure 3.9.4: ZCU102 with the SFP module connected to the SFP cage and with an OM3 (aquamarine color) cable to the PC Realtek Network card

9.4 Combining solutions for redundancy support

Now that the IP Core that interconnects the PS GEM with an SFP module that sports 1000BaseX standard has been tested, the next step for implementing the needed hardware for the project can be done. The final project will have redundant SFPs modules to be able to fallback to another interface.

This section will combine the integrated RJ45 ethernet port used in the beginning of the ZCU102 development phase with the optical SFP used in the last section to have two adapters with the following specs:

- **Eth0:** This will be the SFP adapter with the exact same parameters as when one adapter was being used. This adapter will be the primary ethernet adapter configured at boot time with the listed parameters that will be introduced in: `petalinux-config` → `Subsystem AUTO Hardware Settings` → `Ethernet Settings`:
  - Primary Ethernet (psu_ethernet_0) which corresponds to GEM0 (SFP)
  - randomised or leave it default in the range assigned to these devices *00:0a:35:00:...*
  - IP Address: *10.0.0.2*
  - Subnet mask: *255.255.255.0*
  - Default Gateway: *10.0.0.1*
- **Eth1:** this will be the Ethernet adapter connected to GEM3 which uses the MIO pins going to the ZCU102 integrated Ethernet RJ45. The setup steps to configure this are as follows:
  1. Make the Vivado project with both the Gigabit Ethernet MAC IP Core and enabling inside the IP Configuration of the ZynqMPSOC both GEM 0 and 3 with the parameters shown in figures 3.9.5 and 3.9.6.

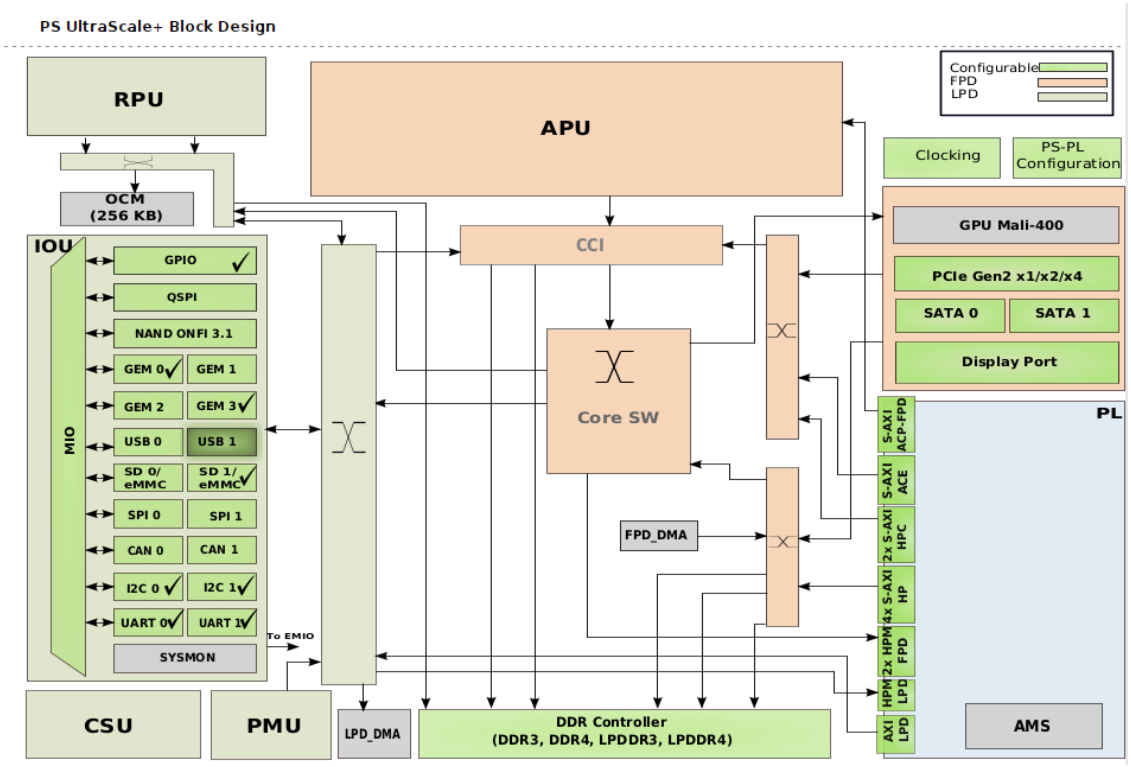
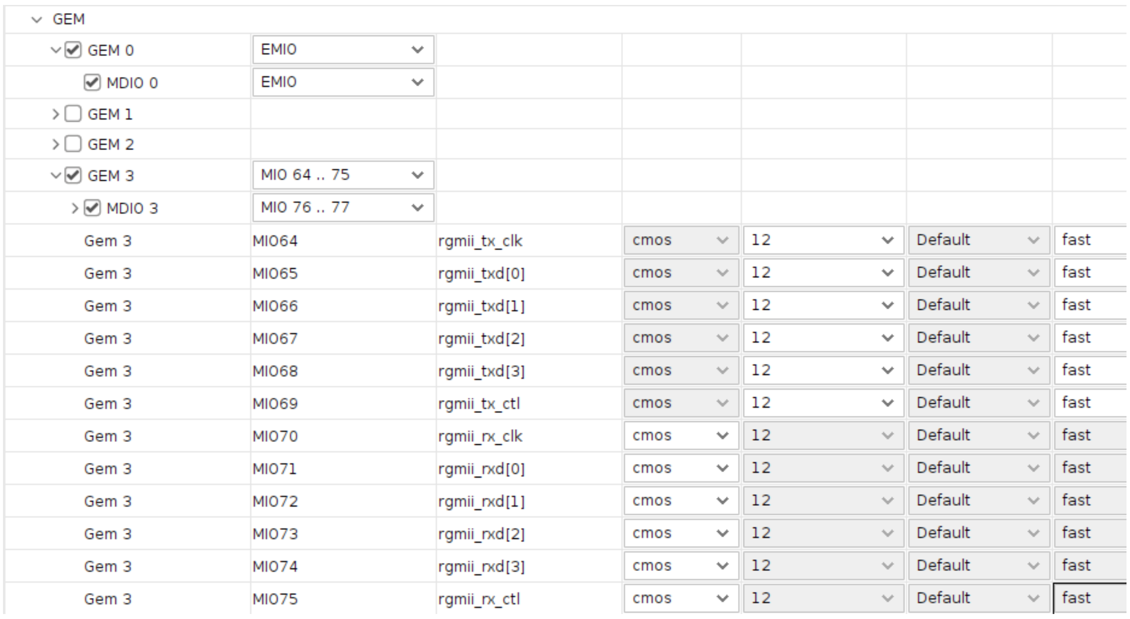


Figure 3.9.5: Block diagram of the UltraScale+ system (GEM0 and GEM3 are ticked)

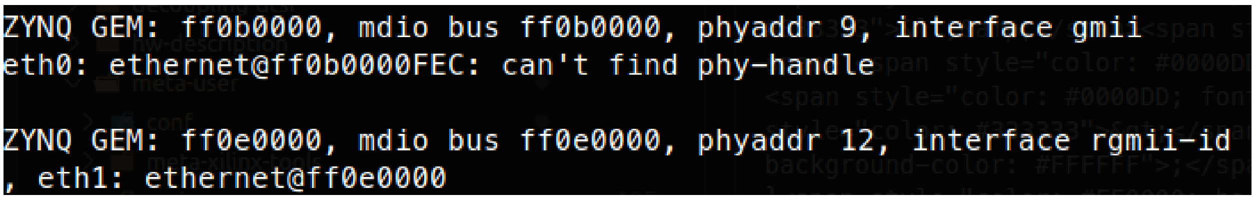


GEM						
☑ GEM 0	EMIO					
☑ MDIO 0	EMIO					
☐ GEM 1						
☐ GEM 2						
☑ GEM 3	MIO 64 .. 75					
☑ MDIO 3	MIO 76 .. 77					
Gem 3	MIO64	rgmii_tx_clk	cmos	12	Default	fast
Gem 3	MIO65	rgmii_txd[0]	cmos	12	Default	fast
Gem 3	MIO66	rgmii_txd[1]	cmos	12	Default	fast
Gem 3	MIO67	rgmii_txd[2]	cmos	12	Default	fast
Gem 3	MIO68	rgmii_txd[3]	cmos	12	Default	fast
Gem 3	MIO69	rgmii_tx_ctl	cmos	12	Default	fast
Gem 3	MIO70	rgmii_rx_clk	cmos	12	Default	fast
Gem 3	MIO71	rgmii_rxd[0]	cmos	12	Default	fast
Gem 3	MIO72	rgmii_rxd[1]	cmos	12	Default	fast
Gem 3	MIO73	rgmii_rxd[2]	cmos	12	Default	fast
Gem 3	MIO74	rgmii_rxd[3]	cmos	12	Default	fast
Gem 3	MIO75	rgmii_rx_ctl	cmos	12	Default	fast

Figure 3.9.6: Table with the HW pign assignments of GEM0(EMIO) and GEM3 (MIO)

2. Run the Vivado workflow for synthesis, implementation and bitstream generation
3. Export XSA including the bitstream.
4. Run `petalinux-config --get-hw-description /path/to/xsa`. This command will use the XSA as the system.xsa by renaming and copying it to the `project-spec/hw-description` folder.
5. Inside `petalinux-config` go to `Subsystem AUTO Hardware Settings` → `Ethernet Settings` and set up the parameters listed above.
6. In `project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi` and ensure that the GEM0 is being created with the same parameters as in section 9.3. GEM3 does not need to be added to the device tree because, as it is already a fixed Ethernet port in the ZCU102 what is being connected to that and it is included in the .dtsi of the ZCU102 included in the BSP that was used to create the project.
7. Build the image through `petalinux-build` and the *BOOT.BIN* through `petalinux-package --boot --force --fsbl --fpga --u-boot`
8. Copy the files to the SD card and boot the board.

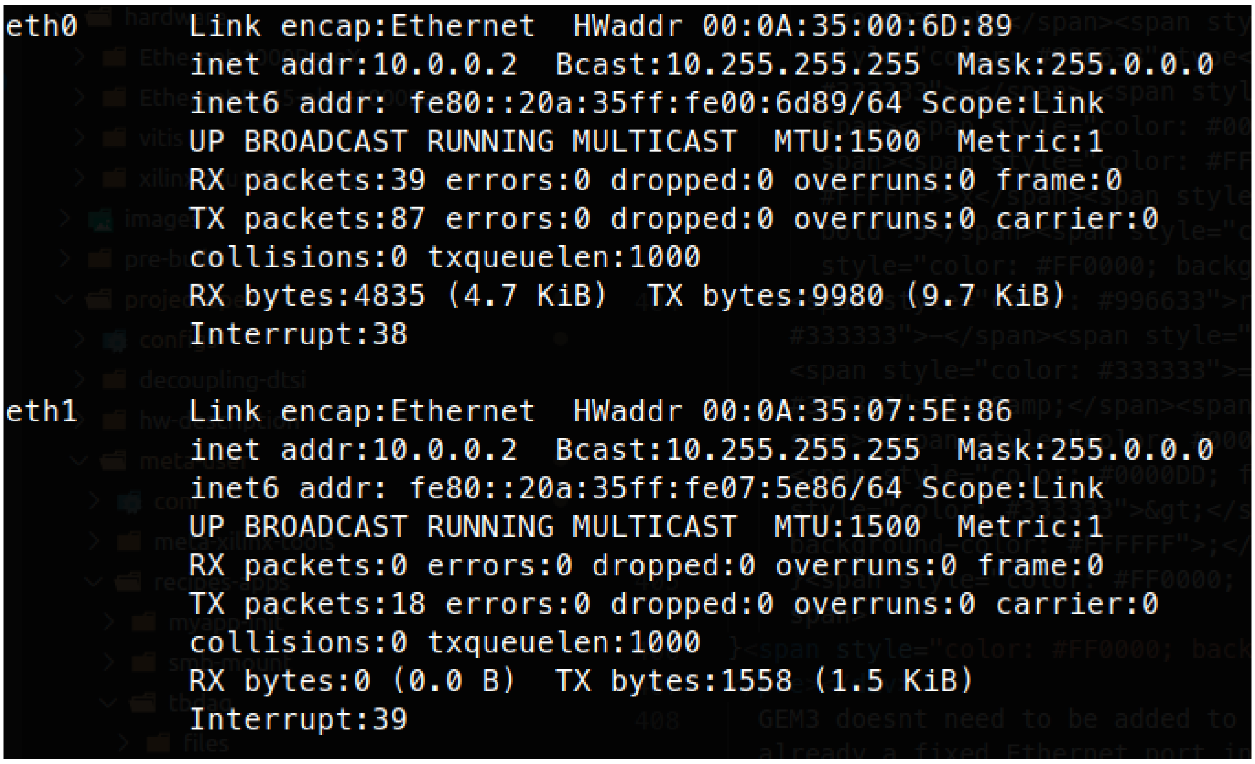
When the board boots, once U-boot is running, the output in the console should be similar to what is shown in figure 3.9.7. It detects the two ZYNQ GEM with their corresponding PHY addresses. 9 is the PHY address of the SFP, manually selected in the IP Core and 12 is the hardcoded PHY address of the PHY chip included in the ZCU102 that is routed to its RJ45 port. The phy-handle error can be ignored, it will not affect normal operation of the interface.



```
ZYNQ GEM: ff0b0000, mdio bus ff0b0000, phyaddr 9, interface gmii
eth0: ethernet@ff0b0000FEC: can't find phy-handle

ZYNQ GEM: ff0e0000, mdio bus ff0e0000, phyaddr 12, interface rgmii-id
, eth1: ethernet@ff0e0000
```

Figure 3.9.7: Ethernet interfaces detected with their corresponding phy addresses (9 for SFP, 12 for RJ45)

If the `ifconfig` command is executed the output in figure 3.9.8 will be shown:



```
eth0      Link encap:Ethernet  HWaddr 00:0A:35:00:6D:89
          inet addr:10.0.0.2  Bcast:10.255.255.255  Mask:255.0.0.0
          inet6 addr: fe80::20a:35ff:fe00:6d89/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:39 errors:0 dropped:0 overruns:0 frame:0
          TX packets:87 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4835 (4.7 KiB)  TX bytes:9980 (9.7 KiB)
          Interrupt:38

eth1      Link encap:Ethernet  HWaddr 00:0A:35:07:5E:86
          inet addr:10.0.0.2  Bcast:10.255.255.255  Mask:255.0.0.0
          inet6 addr: fe80::20a:35ff:fe07:5e86/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:18 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:1558 (1.5 KiB)
          Interrupt:39
```

Figure 3.9.8: eth0 and eth1 interfaces detected by Petalinux. IP address of eth1 needs to be modified in boot stage

In figure 3.9.8 the output states that there are two ethernet adapters *eth0* and *eth1* which have their respective MAC addresses. The only issue here is that due to the SDK not considering that there could be more than one ethernet adapters connected it has assigned the same IP addresses. This can be sorted out by changing the initial configuration creating a systemv event that will be executed after the *network.service* which initializes all the network links.

After applying the pertinent modifications to the eth1 interface, assigning it an IP within a different subnetwork, for instance 20.0.0.2 the *ping* and the *iperf3* tests were ran, giving a successful result in both tests meaning that an Ethernet controller instantiated in the PL can perform the desired tasks for having more Ethernet interfaces available in Linux. This will help in the final version of the design to have more than enough interfaces to allow for redundancy, if one interface fails we have another one or two to relieve it in its duties.

Chapter 10

Applying patches to Petalinux

Through the development process, there are some instances where a bug that is already solved in a more recent version of the petalinux kernel. This poses a big problem for people who started working and developed their software to work on a embedded platform with an old petalinux version. However, linux kernel offers the possibility to apply patches, which are text files with a special syntax, similar to the ones found in git that remove, add or replace lines of code in the source file during the compilation process so that the original source file is left unchanged with the bug as it was in that version but it compiles the kernel with the fix included. Petalinux has a very straightforward way to apply patches. In the project folder the user can go to `project-spec/meta-user/recipes-kernel/linux` and copy the patch files. There should be a folder and a *.bbappened* file. Inside the folder copy the .patch files and the in the *.bbaappend* file add them in the corresponding environment variable as in the following example:

```
SRC_URI += "file://bsp.cfg"
SRC_URI += "file://Patch001.patch"
SRC_URI += "file://Patch002.patch"
...
```

After adding these files, you need to issue the command `petalinux-build -c kernel` to build the kernel again. However, if you go to the source files of the linux kernel stored in `components/yocto/workspace/sources/linux-xlnx`, there won't be any change in the file specified by the patch because the old source files belonging to previous compilations remain. You need to erase the corresponding source files for the `petalinux-build` command to regenerate them with the patch applied. It is recommended to erase all the build results from previous run by running the following command:

```
hyperk0@hyperk0:~$ petalinux-build -x mrproper -f
```

In this section, an example is provided to explain the procedure to apply a patch. This example is done using a patch made by Xilinx to solve a bug related to the Gigabit Ethernet MAC and the 125MHz clock used in the transceiver.

The two patches are stated in AR-72806:

- *Patch 0001* affects the xilinx_phy.c file which includes the driver for the Xilinx IP itself. This enables a flag that makes a reset after clock enable. This ensures that the reset is made in the correct moment and the IP is correctly reconfigured so that the link is up again.
- *Patch 0002* affects the macb.c driver which is the Cadence GEM driver inside the PS. This patch schedules the needed restart of the PHY by sending in the correct instant the reset flag enabled in patch 0001. When the reset is successful, the function in this patch returns a 1, which clears the way to reinitialize the PHY.

In the 2020.1 case, it was only needed to apply *Patch 0001* because the other one had already been applied to the macb driver in 2020.1 version. However, the part of the fix that belongs to the `xilinx_phy.c` driver was applied in April 2021, so when the SDK does the pull from the repository, as it downloads tag 2020.1, it doesn't receive the patched code.

10.1 Patch programming

As it was said before, drivers of the different peripherals that can be used in petalinux get updates together with the OS update.This can bring new functionality, bugfixes among other things. However, there is the possibility that new bugs or unexpected behaviour is generated due to recent changes. To avoid waiting for this to be officially fixed and having to update to a more recent linux version, patches were created. This are defined as `diff` commands, that compare two files: the original that suffers from the one and another modified from this original fire that applies the fixes. This command generates a text file with a special syntax that contains the insertions and deletions of code that should be applied over the original file. For generating a patch the procedure is detailed in the Xilinx wiki:

1. When issuing `petalinux-build` command, the SDK has a step where the kernel is compiled from its source code. This source code is pulled from the Xilinx Github repository that contains linux kernel, called linux-xlnx, so the first step would be to pull from the repository all the source code.
2. Check which branch is your Petalinux SDK using. This can be easily checked in the corresponding Petalinux Release Notes. In this case, as 2022.2 table states, the repository *linux-xlnx* is pulling branch *xlnx_rebase_v5.15_LTS*.
3. Create a new branch through `git checkout -b <branch-name> <original branch>` where the original branch is *xlnx_rebase_v5.15_LTS*.
4. Modify the required source codes and perform the needed testing to ensure the fix works well.
5. Perform `git add` and `git commit` to add changes to the local repository.
6. If the command `git show` command is issued, the commits done will be displayed. Each commit will generate a separate patch file that can be applied independently.
7. Generate the patch by doing `git format-patch <branch base>` and a .patch file with the standard git diff format will be generated by comparing the files modified in the commit with the ones given in the branch named `<branch base>`.
8. Copy the patch to the `project-spec/meta-user/recipes-kernel/linux` directory and add it to the recipe file using the `file://` as shown previously.
9. Follow the normal compilation flow explained previously. If you check the source code located in the `build` folder, the new source code should have the patch applied.

10.2 Patch structure

This last section shows an example of a patch I had to program to delete a driver checking that was causing the Ethernet interface not being brought up. This structure is auto generated by git and basically states, which branch is the base, that is, where this source code patch should be applied, the name of the author, date and name of the patch. Then it uses the `diff` command with `---` and `+++` to say which lines should be excluded or included with respect to the base code. In this example as it was just deleting code, there are only `---`.

```
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
```

Part IV

Conclusions and Future Work

Chapter 1

Lessons learned

This TFM has been a comprehensive explanation of several aspects of the Xilinx Zynq Ultrascale+ architecture, how to take advantage of it in a EvB and how it can be used together with an embedded OS to fulfill the special requirements of such a sensible experiment like Hyper-Kamiokande. The knowledge acquired while learning about this extensive development platforms and how they helped finding the correct way to proceed in the implementation of the required functions for the DPB fulfilling reliability and speed specifications are listed below:

- **PS + PL Architecture:** The PS and PL division has been an architecture I have never worked with before. Traditional FPGA architecture include Logic elements based on LUTs to implement any logic function and then, through routing resources, it joins all these Logic Elements to implement the hardware described in HDL. However, for most of the applications a CPU is needed so the Zynq Ultrascale+ architecture also includes the PS which allows to run a full fledged Linux in four A53 ARM cores and, if that is not enough there is a dual Arm Cortex-R5F to run real time applications. These cores are hard cores, included in the silicon and are interconnected to the PL through EMIO and through high speed AXI interfaces, so that the PL can be used to instantiate IP cores like peripherals and then connect them and control them through a Linux driver like a conventional PC. This provides the advantages of flexibility to do hardware acceleration on the FPGA together with powerful processing power for OS running and general purpose applications in a well-supported platform like ARM.
- **Petalinux as embedded OS:** Xilinx provides a complete SDK which helps a lot during the development process in compiling and running an OS from scratch. An OS is not a conventional app and the Linux kernel has undergone many revisions to make it as powerful as possible while adding compatibility with more CPU architectures. This Linux distribution made by Xilinx specifically for their chips provides the standard development environment for any Linux developer with the ability of even building the applications in the evaluation board, with advanced debugging possibilities.
- **Bootup methods:** The Zynq Ultrascale+ has several hard microcontrollers in the PS for different types of memories. Besides the DDR4 memory that acts as the main memory of the system there are other controllers for QSPI, eMMC and SD flash, which means that, when the PS is initialised through the FSBL, these memories get available to be read from their corresponding microcontrollers and boot the system from there. This provides a huge difference in reliability because we can even have memory redundancy with different chips and if one memory dies completely, another one can be used. Going further, Petalinux also supports PXE boot thanks to the ability of downloading a kernel image from a TFTP server hosted in the Internet.

- **Image customization thanks to Yocto:** One of the key points of Petalinux is its flexibility and high customizability which makes it a perfect match for a chip with Zynq Ultrascale+ architecture. The Petalinux image can be customised with any software package, adding or stripping out any part of the OS which is not needed in order to make it as light as possible or as powerful as needed. This is possible thanks to Yocto, a project explained in Part II that aims to make an environment to fully customise a Linux image to suit the user needs. Understanding Yocto, the recipes, layers and how the toolchain works has been essential to perform an efficient image building and this knowledge is essential to build the DPB with the requested software that manages the data link, slow control, etc...

- **Initialization scripts:** As the system must be as autonomous as possible, the initial configuration after boot of the OS should be done automatically. Linux already supports this as it is an aspect that it is crucial from PC to phones or embedded systems. In this TFM, both *systemv* and *systemd* have been studied. The latter is an evolution of the first one, with some disagreement among the Linux community but it is true that providing services with more control of the order in which scripts get executed through *.service* files is, in my opinion, a cleaner way to implement init configuration and will be the chosen one for implementing the DPB firmware.

- **Slow control data sampling:** sensors are usually controlled through a serial protocol to configure the sensor and read the data registers. In this TFM, the hard SPI controller has been programmed with a magnetometer sensor and a test has been done by reading one of its registers. This has been made to prove that the Zynq Ultrascal+ can manage through Linux the writing and reading from the different sensors that will be present in the DPB. If they are not SPI sensors, it doesn't matter, as I2C hard controllers are also available in the PS and you can even instantiate any core in the FPGA such as an UART, RS232, etc... to enable any kind of serial communication so slow control is very well covered no matter the protocol it is decided to use further in the development of the DPB.

- **Adding extra network interfaces:** For implementing data and timing links, optical links will be used. These links will be using two different protocols. Regarding data link, the standard TCP/IP stack will be used. In this TFM the procedure to add extra network interfaces that support this stack has been explained. It is fairly simple to add more links through SFP modules routed to the PL as an IP Core is all that is needed. There is already a Linux driver that supports all the functions needed to establish a link with another device. One must be very careful of the configuration in the device tree as the IP Core is configured through a Linux driver and the data about the core is known to the driver through its device tree node. For example, if the core is configured in SGMII mode but the developer tells the driver that it is 1000BaseX, auto-negotiation will fail, as the signalisation is different in both cases.

Chapter 2

Development of the future prototype inside the vessel

This TFM is like a diary of my first steps in the HKK project, departing from almost zero knowledge on Linux development and about the Xilinx environment itself, I have been writing and documenting every step I have followed to create a fully functional Linux image with some of the hardware support required by the DPB functions.

This TFM was written while the project did not have a fixed hardware platform to work on. It was decided to work on Zynq Ultrascale+ architecture but not in which specific chip or board, so the ZCU102 was the better approach to start learning. However, moving forward, a hardware platform has been defined. This platform will be the one mounted in the vessel prototypes to be tested in the CERN institute. The name of the platform is SOM, or System On Module and provides a lot of flexibility when it comes to I/O ports as the configuration of the DPB has an uncommon combination of ports.

In this SOM, the following parts must be developed:

- **Implementing DAQ communication:** the software for DAQ communication tested in this TFM is just a demo of the capabilities of the framework. The real application must be coded following the same instructions as this sample demo.
- **Bonding driver:** Now that several network interfaces can be brought up, it is needed to establish a way to coordinate them so that they act as backup interfaces where only one of them is working at the same time and if that one fails the other one will be enabled. This has already been investigated theoretically. It is called bonding and Linux has native drivers that support it.
- **Slow control implementation:** The DPB prototype has all the required sensors by the project with their well known protocols and registers. In this TFM an example has been given about how to use SPI to control and monitor a sensor but there are much more possibilities. For instance, the HV and LV boards can be read and configured through RS485 protocol so, a new procedure must be defined to enable RS485 communication from Petalinux.
- **Timing link implementation:** Timing is crucial for the digitizer boards as they need to provide very accurate timestamps for the recorded events of the PMT. This timing information is given through a second stage timing distribution system that is connected to each DPB of the observatory. Then, the DPB through a custom IP Core decodes the data and sends it directly to the digitizer. This cores once defined and described in HDL must be instantiated in the PL to route the data from the SFP timing links to the corresponding MiniSAS pins.

- **Digitizer communication:** The digitizer link to the DPB is 2.5Gbps, so a protocol supporting such a high speed needs to be used. After several meetings, the protocol that will be used is the Aurora protocol, a communications standard made by CERN. Once a digitizer prototype is available, an Aurora core will be instantiated in the FPGA of both the digitizer and the DPB. Then, data should be transmitted to the DPB and read afterwards by the CPU through a DMA.

This parts will be extensively covered in the followup of this TFM. I am studying two masters so, I need to write two TFM. This one is the Electronics Systems Engineering TFM and the next will be Telecommunications Engineering one where I will cover the prototypes to be tested in the CERN together with the procedure and changes with respect to the general development aspects explained here with the ZCU102 as the reference platform. To give an insight of what the state of development at the moment of writing these lines, figure 4.2.1 shows the SOM with a FMC card connected to it. This card has two SFP modules with the same optical cables used in the ZCU102. These two links will be used together with bonding to enable data link redundancy through the standard TCP/IP stack.

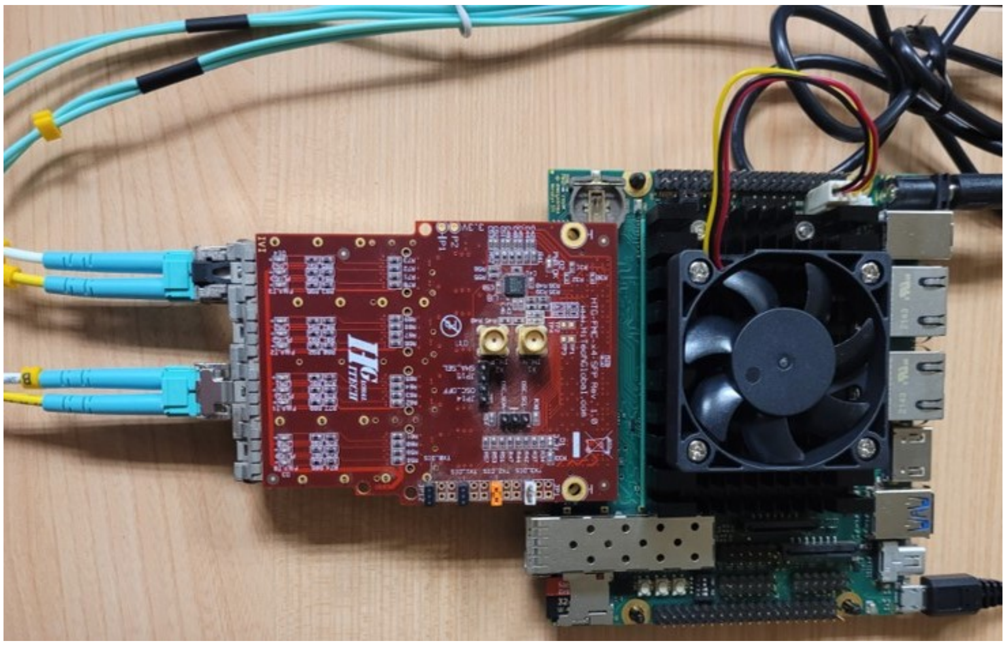


Figure 4.2.1: SOM current state of development. Future work will focus on this platform

Bibliography

[1] Baianu IC et al. *Fundamentals of Physics and Nuclear Physics.* Free Software Foundation Inc., Nov. 2002.

[2] Laurie M. Brown. “The idea of the neutrino”. In: *Phys. Today* 31N9 (1978), pp. 23–28. DOI: 10.1063/1.2995181.

[3] Enrico Fermi. “Versuch einer Theorie der β-Strahlen. I”. In: *Zeitschrift für Physik* 88 (1934), pp. 161–177.

[4] Kan Chang Wang. “A Suggestion on the Detection of the Neutrino”. In: *Phys. Rev.* 61.1-2 (1942), pp. 97–97. DOI: 10.1103/physrev.61.97.

[5] C. L. Cowan, F. Reines, F. B. Harrison, H. W. Kruse, and A. D. McGuire. “Detection of the Free Neutrino: a Confirmation”. In: *Science* 124.3212 (1956), pp. 103–104. DOI: 10.1126/science.124.3212.103. eprint: https://www.science.org/doi/pdf/10.1126/science.124.3212.103. URL: https://www.science.org/doi/abs/10.1126/science.124.3212.103.

[6] The Nobel Prize. *The Nobel Prize in Physics 1995.* TheNobelPrize. 1995.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] Gianfranca De Rosa. “A multi-PMT photodetector system for the Hyper-Kamiokande experiment”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 958 (2020). Proceedings of the Vienna Conference on Instrumentation 2019, p. 163033. ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2019.163033. URL: https://www.sciencedirect.com/science/article/pii/S0168900219313968.

[13] University of Tokyo, The High Energy Accelerator Research Organization, The Ministry of Science, and Innovation of the Kingdom of Spain. *Memorandum of Understanding on the Spanish Participation in the Hyper-Kamiokande Experiment.* https://www.ciencia.gob.es/ca/Noticias/2022/Septiembre/Espa-a-y-Japon-firman-Tratado-de-entendimiento.html. 2022.

[14] Advanced Micro Devices (AMD). *What is an FPGA.* https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html#:~:text=Field%20Programmable%20Gate%20Array%20(FPGA). 2023.

[15] 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.

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

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

[18] 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).

[19] *ZCU102 Evaluation Board. User Guide.* 1.7. https://docs.xilinx.com/v/u/en-US/ug1182-zcu102-eval-bd. AMD Xilinx. Feb. 2023.

[20] *Precision 3650. Ready to build the next big thing.* https://www.delltechnologies.com/asset/en-us/products/workstations/technical-support/precision-3650-spec-sheet.pdf. Dell Technologies. Aug. 2019.

[21] *DGE-528T PCI Gigabit Ethernet Adapter. Product Highlights.* https://eu.dlink.com/es/es/-/media/business_products/dge/dge-528t/datasheet/dge_528t_c1_datasheet_en.pdf. D-Link. 2014.

[22] *PCI Express Gigabit Ethernet Fiber Network Card w/ Open SFP. PEX1000SFP2.* https://media.startech.com/cms/pdfs/pex1000sfp2_datasheet.pdf. StarTech. Mar. 2023.

[23] *UG1339. Vitis High-Level Synthesis User Guide.* 2022.2. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/. AMD Xilinx. Dec. 2022.

[24] FPGAKey. *Vivado. Wiki Encyclopedia.* https://www.fpgakey.com/wiki/details/4. 2023.

[25] Bogdan Pasca. "High-performance floating-point computing on reconfigurable circuits". In: (Sept. 2011).

[26] VHDL Whiz. *Using Integrated Logic Analyzer (ILA) and Virtual Input/Output (VIO).* https://vhdlwhiz.com/using-ila-and-vio. Aug. 2021.

[27] *UG994. Designing IP Subsystems Using IP Integrator.* 2022.2. https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2022_2/ug994-vivado-ip-subsystems.pdf. AMD Xilinx. Oct. 2022.

[28] *PG172. Integrated Logic Analyzer v6.2 LogiCORE IP Product Guide.* 6.2. https://docs.xilinx.com/v/u/en-US/pg172-ila. AMD Xilinx. Oct. 2016.

[29] Canonical. "A guide to Linux for embedded applications". In: (Feb. 2022).

[30] Canonical. "Embedded Linux: Yocto or Ubuntu Core?" In: (Nov. 2022).

[31] Justin Ellingwood. *Understanding Systemd Units and Unit Files.* https://www.digitalocean.com/community/tutorials/understanding-systemd-units-and-unit-files. Feb. 2015.

[32] Thomas Petazzoni. *Device Tree for Dummies.* https://elinux.org/images/f/f9/Petazzoni-device-tree-dummies_0.pdf. Nov. 2013.

[33] *U-boot Reference Manual.* 90000852_K. https://hub.digi.com/dp/path=/support/asset/u-boot-reference-manual/. Digi. 2011.

[34] *U-Boot for i.MX51 Based Designs Source Code Overview and Customization.* AN4173. https://www.nxp.com/docs/en/application-note/AN4173.pdf. Freescale Semiconductor. July 2010.

[35] Mathworks Inc. *Author a Xilinx Zynq Linux Image for a Custom Zynq Board by Using MathWorks Buildroot.* https://es.mathworks.com/help/hdlcoder/ug/xilinx-zynq-linux-image-for-custom-boards.html. 2023.

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

[37] Henry Bruce. *Extensible SDK.* https://wiki.yoctoproject.org/wiki/Extensible_SDK. 2023.

[38] *PetaLinux Tools Documentation. Command Line Reference Guide. UG1157.* 2020.1. https://docs.xilinx.com/r/en-US/ug1144-petalinux-tools-reference-guide. AMD Xilinx. June 2020.

[39] *LIS3MDL adapter board for standard DIL24 socket.* https://www.mouser.es/datasheet/2/389/steval-mki137v1-1848614.pdf. STMicroelectronics. Sept. 2016.

[40] *Digital output magnetic sensor: ultra-low-power, high-performance 3-axis magnetometer.* https://www.mouser.es/pdfdocs/lis3mdl.pdf. STMicroelectronics. May 2017.

[41] *1G/2.5G Ethernet PCS/PMA or SGMII LogiCore IP Product Guide. PG047.* 16.2. https://docs.xilinx.com/r/en-US/ug1144-petalinux-tools-reference-guide. AMD Xilinx. Nov. 2022.

Part V

Annexes

Annex 1

Additional Listings

U-boot help command:

```
  # help
autoscr - run script from memory
base - print or set address offset
bdinfo - print Board Info structure
boot - boot default, i.e., run 'bootcmd'
bootd - boot default, i.e., run 'bootcmd'
bootelf - Boot from an ELF image in memory
bootm - boot application image from memory
bootp - boot image via network using BootP/TFTP protocol
bootvx - Boot vxWorks from an ELF image
clock - Set Processor Clock
cmp - memory compare
coninfo - print console devices and information
cp - memory copy
crc32 - checksum calculation
date - get/set/reset date & time
dboot - Digi ConnectCore modules boot commands
dcache - enable or disable data cache
dhcp - invoke DHCP client to obtain IP/boot params
echo - echo args to console
envreset- Sets environment variables to default setting
fatinfo - print information about filesystem
fatload - load binary file from a dos filesystem
fatls - list files in a directory (default /)
flpart - displays or modifies the partition table.
fsinfo - print information about filesystems
fsload - load binary file from a filesystem image
go - start application at address 'addr'
help - print online help
icache - enable or disable instruction cache
icrc32 - checksum calculation
iloop - infinite loop on address range
imd - i2c memory display
iminfo - print header information for application image
imm - i2c memory modify (auto-incrementing)
imw - memory write (fill)
inm - memory modify (constant address)
intnvram- displays or modifies NVRAM contents like IP or partition table
iprobe - probe to discover valid I2C chip addresses
itest - return true/false on integer compare
loadb - load binary file over serial line (kermit mode)
```

```
loads - load S-Record file over serial line
loady - load binary file over serial line (ymodem mode)
loop - infinite loop on address range
ls - list files in a directory (default /)
md - memory display
mm - memory modify (auto-incrementing)
mtest - simple RAM test
mw - memory write (fill)
nand - NAND sub-system
nboot - boot from NAND device
nfs - boot image via network using NFS protocol
nm - memory modify (constant address)
ping - send ICMP ECHO_REQUEST to network host
printenv- print environment variables
printenv_dynamic- Prints all dynamic variables
rarpboot- boot image via network using RARP/TFTP protocol
reset - Perform RESET of the CPU
run - run commands in an environment variable
saveenv - save environment variables to persistent storage
setenv - set environment variables
sleep - delay execution for some time
sntp - synchronize RTC via network
tftpboot- boot image via network using TFTP protocol
update - Digi ConnectCore modules update commands
usb - USB sub-system
usbboot - boot from USB device
version - print monitor version
```

U-Boot example script for pxe boot:

```
setenv boot_targets "qspi0 pxe"
for boot_target in ${boot_targets};
do
    if test "${boot_target}" = "jtag" ; then
        booti 0x00200000 0x04000000 0x00100000
        exit;
    fi
    if test "${boot_target}" = "mmc0" || test "${boot_target}" = "mmc1" ; then
        if test -e ${devtype} ${devnum}:${distro_bootpart} /image.ub; then
            fatload ${devtype} ${devnum}:${distro_bootpart} 0x10000000 image.ub;
            bootm 0x10000000;
            exit;
        fi
        if test -e ${devtype} ${devnum}:${distro_bootpart} /Image; then
            fatload ${devtype} ${devnum}:${distro_bootpart} 0x00200000 Image;;
        fi
        if test -e ${devtype} ${devnum}:${distro_bootpart} /system.dtb; then
            fatload ${devtype} ${devnum}:${distro_bootpart} 0x00100000 system.dtb;
        fi
        if test -e ${devtype} ${devnum}:${distro_bootpart} /rootfs.cpio.gz.u-boot; then
```

```
            fatload ${devtype} ${devnum}:${distro_bootpart}
            0x04000000 rootfs.cpio.gz.u-boot;
            booti 0x00200000 0x04000000 0x00100000
            exit;
        fi
        booti 0x00200000 - 0x00100000
        exit;
    fi
    if test "${boot_target}" = "xspi0" || test "${boot_target}" = "qspi"
    || test "${boot_target}" = "qspi0"; then
        sf probe 0 0 0;
        if test "image.ub" = "image.ub"; then
            sf read 0x10000000 0xF00000 0x6400000;
            bootm 0x10000000;
            exit;
        fi
        if test "image.ub" = "Image"; then
            sf read 0x00200000 0xF00000 0x1D00000;
            sf read 0x04000000 0x4000000 0x4000000
            booti 0x00200000 0x04000000 0x00100000
            exit;
            fi
        exit;
    fi
    if test "${boot_target}" = "nand" || test "${boot_target}" = "nand0"; then
        nand info
        if test "image.ub" = "image.ub"; then
            nand read 0x10000000 0x4100000 0x6400000;
            bootm 0x10000000;
            exit;
        fi
        if test "image.ub" = "Image"; then
            nand read 0x00200000 0x4100000 0x3200000;
            nand read 0x04000000 0x7800000  0x3200000;
            booti 0x00200000 0x04000000 0x00100000
            exit;
        fi
    fi
    if test "${boot_target}" = "pxe" ; then
        setenv fdt_addr_r 0x50000000
        setenv ipaddr 10.0.0.2
        setenv serverip 10.0.0.1
        if pxe get ; then
            pxe boot
        fi
    fi
done
```