

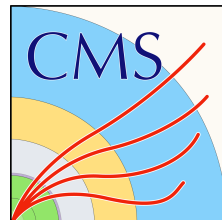


Aristotle University of Thessaloniki
Faculty of Engineering
School of Electrical and Computer Engineering
Department of Electronics and Computers

Master Thesis

An automated CI/CD framework for Zynq UltraScale+ MPSoC devices

Name: Vasileios Amoiridis
Student ID: 8772
University Supervisor: Ioannis Papaefstathiou
CERN Supervisors: Petr Žejdl & Marc Dobson



June 30, 2022
Geneva, Switzerland



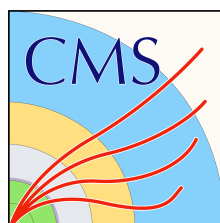


Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης
Πολυτεχνική Σχολή
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Ηλεκτρονικής και Υπολογιστών

Διπλωματική Εργασία

Ένα αυτόματο πλαίσιο για συνεχή ενσωμάτωση και συνεχή
παράδοση/ανάπτυξη για ολοκληρωμένα της οικογένειας Zynq
UltraScale+ MPSoC

Ονοματεπώνυμο: Βασίλειος Αμοιρίδης
ΑΕΜ: 8772
Υπεύθυνος Καθηγητής: Ιωάννης Παπαευσταθίου
Υπεύθυνοι CERN: Petr Žejdl & Marc Dobson



30 Ιουνίου, 2022
Γενεύη, Ελβετία

Abstract

CERN is working on the High-Luminosity LHC (HL-LHC) upgrade which will be installed in 2026. The CMS experiment and its Data Acquisition (DAQ) system will also be upgraded. The upgrade of the CMS DAQ system involves the installation of new electronics that will also host the Zynq UltraScale+ Multi-Processor System-on-Chip (MPSoC) from Xilinx. The Zynq UltraScale+ MPSoC will run control and monitoring software on a Linux Operating System (OS).

Building and testing software and firmware to boot hundreds of embedded system boards is quite demanding and can introduce possible failures due to its complexity. These failures can hold back the development and the operation of various projects using the Zynq UltraScale+ MPSoC. This thesis presents the research, design, implementation, and testing of an automated CI/CD workflow on GitLab, that can build and deploy the software and firmware of the board, then boot it, and monitor its boot process.

The automated CI/CD framework introduces eight pipelines that automate various parts of the design and test phase of a project for the Zynq UltraScale+ MPSoC. The pipelines have been designed in a way that they can build the Vivado and the PetaLinux projects. They can build the CentOS 8 root filesystem and also deploy the boot images to the TFTP and NFS servers. The pipelines are also able to bootstrap the SD Card of the board and boot it through the Network. Finally, the pipelines monitor the boot process and interact with the Zynq UltraScale+ MPSoC.

Further research areas have also been highlighted that can be implemented in a future project to improve the automated GitLab CI/CD framework. Such cases are the development of customized tests from the DAQ developers for the software and the firmware of the Zynq UltraScale+ MPSoC.

Περίληψη

Ο Ευρωπαϊκός Οργανισμός Πυρηνικών Ερευνών (CERN) εργάζεται στην αναβάθμιση του Μεγάλου Επιταχυντή Αδρονίων Υψηλής Φωτεινότητας (HL-LHC) ο οποίος θα εγκατασταθεί το 2026. Το περίγραμμα Συμπαγούς Μιονικού Σωληνοειδούς (CMS) και το Σύστημα Απόκτησης Δεδομένων (DAQ) επίσης θα αναβαθμιστούν. Η αναβάθμιση του CMS DAQ συστήματος περιλαμβάνει την εγκατάσταση νέων ηλεκτρονικών τα οποία θα φιλοξενήσουν το σύστημα πολλαπλών επεξεργασιών σε ολοκληρωμένο κύκλωμα (MPSoC) της οικογένειας Zynq UltraScale+ της Xilinx. Το Zynq UltraScale+ MPSoC θα εκτελεί λογισμικό ελέγχου και παρακολούθησης σε Λειτουργικό Σύστημα Linux.

Η δημιουργία και η δοκιμή λογισμικού και υλικολογισμικού για την εκκίνηση εκατοντάδων ενσωματωμένων συστημάτων είναι αρκετά απαιτητικές και μπορούν να παρουσιάσουν πιθανές αστοχίες λόγω της πολυπλοκότητάς τους. Αυτές οι αποτυχίες μπορούν να εμποδίσουν την ανάπτυξη και λειτουργία διαφόρων εργασιών που εκτελούνται στο Zynq UltraScale+ MPSoC. Η παρούσα διπλωματική εργασία παρουσιάζει την έρευνα, τον σχεδιασμό, την υλοποίηση, και τη δοκιμή ενός αυτοματοποιημένου πλαισίου εργασιών συνεχής ενσωμάτωσης και συνεχής παράδοσης/ανάπτυξης CI/CD, που είναι σε θέση να δημιουργήσει και να αναπτύξει το λογισμικό και το υλικολογισμικό της πλακέτας, ενώ επίσης είναι σε θέση να το εκκινήσει και να παρακολουθήσει τη διαδικασία εκκίνησης.

Το αυτοματοποιημένο πλαίσιο CI/CD εισάγει οκτώ αγωγούς που αυτοματοποιούν διάφορα κομμάτια της φάσης σχεδιασμού και δοκιμής ενός έργου για το Zynq UltraScale+ MPSoC. Οι αγωγοί έχουν σχεδιαστεί με τρόπο ώστε να μπορούν να δημιουργήσουν και να αναπτύξουν τα project του Vivado και του PetaLinux. Μπορούν να δημιουργήσουν το CentOS 8 ROOTFS και να αναπτύξουν τα εκτελέσιμα αρχεία εκκίνησης στους TFTP και NFS servers. Οι αγωγοί έχουν τη δυνατότητα να προετοιμάσουν την εκκίνηση την SD κάρτα της πλακέτας και να εκκινήσουν το Zynq UltraScale+ MPSoC μέσω του δικτύου. Τέλος, οι αγωγοί παρακολουθούν τη διαδικασία εκκίνησης και αλληλεπιδρούν με το Zynq UltraScale+ MPSoC.

Έχουν επίσης τονιστεί περαιτέρω ερευνητικοί τομείς που μπορούν να εφαρμοστούν σε ένα μελλοντικό έργο, με σκοπό τη βελτίωση του αυτοματοποιημένου πλαισίου CI/CD. Τέτοιες περιπτώσεις είναι η ανάπτυξη προσαρμοσμένων δοκιμών από τους προγραμματιστές της ομάδας του DAQ συστήματος, για το λογισμικό και το υλικολογισμικό του Zynq UltraScale+ MPSoC.

Acknowledgements

Hereby, I would like to express my gratitude to all those who contributed in their own way and supported me over the course of my studies.

I would like to thank Dr. Petr Žejdl for being my supervisor during my time at CERN. His kindness and willingness to support me not only during working hours but also during his free time are highly appreciated. His motivational advice and guidance over the course of my studentship during the global pandemic, helped me to overcome obstacles and surpass myself during the project.

I would like to thank Dr. Marc Dobson for mentoring me during the project at CERN. His knowledge and expertise of the CMS experiment and its Data Acquisition System helped me a lot to understand the fundamentals of the project and what I was working for. His eagerness and useful comments during our discussions in the SoC meetings helped me a lot to excel during the project.

I would like to thank Dominique Gigi, Dr. Emilio Meschi, and Dr. Frans Meijers along with the rest of the DAQ group for their trust, help, and kindness during my time at CERN. The friendly environment that they provided me, shaped an unforgettable experience.

I would like to thank my professor Dr. Ioannis Papaefstathiou for the trust, guidance, and freedom that he gave me during the course of my thesis.

I would like to thank the people that I met on the Aristurtle student team, for all the time that we spent together and for the unforgettable experiences that we shared.

I would like to thank my friends for the amazing experiences that we shared together and the countless discussions that helped me to shape as a person and as an engineer. Their support was of vital importance during the course of my studies.

Finally, I would like to thank my family for always supporting me and being by my side whenever I needed them. Their sacrifices and selfless love helped me to pursue my goals and my dreams.

Vasileios Amoiridis
Geneva, Switzerland
June 30, 2022

Contents

Introduction	1
1 The CERN Laboratory	3
1.1 Introduction to CERN	3
1.2 The CERN accelerator complex	3
1.3 The CMS Experiment	5
1.3.1 The CMS sub-detectors	5
1.3.2 The CMS Data Acquisition system	7
1.4 The High-Luminosity LHC	8
1.4.1 The LHC upgrade	8
1.4.2 The DAQ upgrade	8
2 Project Description	10
2.1 Background	10
2.2 Objective	11
2.3 Requirements & Preconditions	11
2.4 Automation Requirement	12
2.5 Final Product	12
3 Background Research	14
3.1 Zynq UltraScale+ MPSoC internals	14
3.1.1 Hardware overview	14
3.1.2 Application Processing Unit (APU)	14
3.1.3 I/O peripherals	15
3.1.4 Platform Management Unit (PMU)	16
3.1.5 Configuration Security Unit (CSU)	16
3.2 Zynq UltraScale+ MPSoC boot process	16
3.2.1 The First-Stage Bootloader	17
3.2.2 The ARM Trusted Firmware	17
3.2.3 The Second-Stage Bootloader	17
3.2.4 The Linux Kernel boot	18
3.2.5 Summary	19
3.3 Continuous Integration & Continuous Delivery/Deployment	20
3.3.1 Continuous Integration (CI)	20
3.3.2 Continuous Delivery/Deployment (CD)	20
3.3.3 GitLab CI/CD Pipelines	20
3.3.4 GitLab Runners	21
4 Research and analysis	22
4.1 Linux OS on Zynq UltraScale+ MPSoC	22
4.1.1 Overview	22
4.1.2 Hardware Platform Configuration	22

4.1.3	Software Configuration	24
4.1.4	Root Filesystem generation	26
4.1.5	Integration	28
4.1.6	Bootstrapping	29
4.1.7	Bootting	31
4.2	GitLab CI/CD pipelines' notable attributes	33
4.2.1	Job Keywords	33
4.2.2	Global Keywords	35
4.2.3	Multi-project pipelines	35
5	Top-Level System Design	38
5.1	Top-Level System	38
5.2	Boot System	39
5.2.1	SD Card set up and Boot Process	39
5.2.2	Power-Cycling and Boot Monitoring	40
5.2.3	Integration	40
6	Implementation	42
6.1	Vivado Pipeline	42
6.1.1	Structure	42
6.2	PetaLinux Pipeline	43
6.2.1	Structure	43
6.2.2	Configuration	44
6.2.3	Build	45
6.2.4	Package	45
6.3	Dedicated Boot Images Pipeline	46
6.3.1	Structure	46
6.4	Root Filesystem Pipelines	48
6.4.1	CentOS 8 Filesystem Pipeline	48
6.4.2	Board Specific Filesystem Pipeline	49
6.5	Bootstrap Pipeline	50
6.5.1	Structure	50
6.6	Boot Pipeline	50
6.6.1	Power-Cycling script	51
6.6.2	Boot and Monitoring scripts	52
6.6.3	Integration	52
6.7	Maestro Pipeline	53
7	Testing the infrastructure	55
7.1	The AXI Chip2Chip Project	55
7.2	Vivado Pipeline Modifications	56
7.3	PetaLinux Pipeline Modifications	57
7.4	Board Specific Filesystem Pipeline	57
7.5	Results	58

7.6 Summary	59
8 Conclusion	61
9 Future Work	64
List of Figures	65
List of Tables	68
Abbreviations	69
Bibliography	74
A Zynq UltraScale+ MPSoC block diagram	82
B PMU internals and workings	83
C CSU internals and workings	84
D PMU & CSU BootROM codes	85
D.1 The PMU BootROM code	85
D.2 The CSU BootROM code	85
E The FSBL working principle	86
F Boot flow example	87
G GitLab CI/CD pipeline keywords	88
G.1 Job keywords	88
G.2 Global keywords	89
H What is a Docker Container?	90
H.1 Vivado Docker Container	91
H.2 PetaLinux Docker Container	92
H.2.1 Accept EULA script	93
H.3 CentOS 8 ROOTFS Docker Container	93
I CentOS 8 Filesystem Generation	95
I.1 Input Configuration and Options	95
I.2 Additional Binary Formats	95
I.3 DNF input arguments	96
J Network Boot	97
J.1 MAC Address	97
J.2 Boot Images retrieval from TFTP server	98
J.3 Filesystem mounting over NFS	98

Introduction

This diploma thesis is being conducted at the European Organization for Nuclear Research, also known as Conseil Européen pour la Recherche Nucléaire (CERN). CERN is home to the largest particle physics laboratory in the world. It provides a range of particle accelerators, detectors, laboratories, equipment, and general infrastructure that are needed for High-Energy Physics (HEP) experiments. International cooperation between nations, research institutes, universities, and scientists is what drives CERN's research forward. Numerous institutes contribute to the design of the experiments while the analysis of the data produced by the experiments is being conducted by CERN scientists, universities, and research institutes from all around the world [1].



Figure 1: The Globe of Science and Innovation, together with the "Wandering of the Immeasurable" sculpture, at CERN [2].

The project was carried out in the Data Acquisition (DAQ) Group of the Compact Muon Solenoid (CMS) Experiment. For the next system upgrade, Xilinx's Zynq UltraScale+ Multi-Processor System-on-Chip (MPSoC) will be used by the DAQ group. The MPSoC will provide control and monitoring services for the timing and control distribution system and it will run on a Linux OS.

The purpose of this project is to create a framework that provides an automated CI/CD workflow for building, delivering, and testing the software and firmware running on Zynq UltraScale+ MPSoC devices using the GitLab CI/CD pipelines.

Chapter 1 introduces CERN and its accelerator complex. It also describes the CMS experiment with its sub-detectors gives a brief description of the DAQ system, and discusses the undergoing upgrades for the High-Luminosity LHC (HL-LHC).

Chapter 2 gives a thorough description of the project and its objectives. The requirements and the preconditions are stated along with the final product.

Chapter 3 describes the background research that was needed before the project initiates. Several procedures need to be understood to initiate the design of the automated framework.

Chapters 4, 5, and 6 present the research, design, and analysis of the automated CI/CD framework.

Chapter 7 presents and discusses tests and results of the automated CI/CD framework.

Chapter 8 draws a conclusion about the diploma thesis while Chapter 9 underlines improvements and future work that can be done in a follow-up project.

1 The CERN Laboratory

1.1 Introduction to CERN

After the end of the Second World War, science in Europe was not world-class anymore. A handful of visionary scientists from Europe and North America imagined the creation of a physics research facility. Their vision was to stop the brain drain to America but most importantly, to provide a force that would unite Europe. Eventually, CERN was founded in 1954 in the Franco-Swiss border near Geneva [3].

Since its birth in 1954, CERN is responsible for many achievements, such as the discovery of *W and Z bosons*¹ in 1983, the invention of the *World Wide Web* in 1989, the *Large Hadron Collider* in 2008, and the discovery of the *Higgs boson*² in 2012. CERN's contribution to society is not limited to High-Energy Physics. CERN is developing novel technologies that are applied in areas such as health, aerospace, and the environment [4].

1.2 The CERN accelerator complex

CERN operates the world's largest and most powerful particle accelerator. The Large Hadron Collider (LHC) is composed of a 27-kilometre ring of superconducting magnets with multiple accelerating structures that boost particles' energy [5,6].

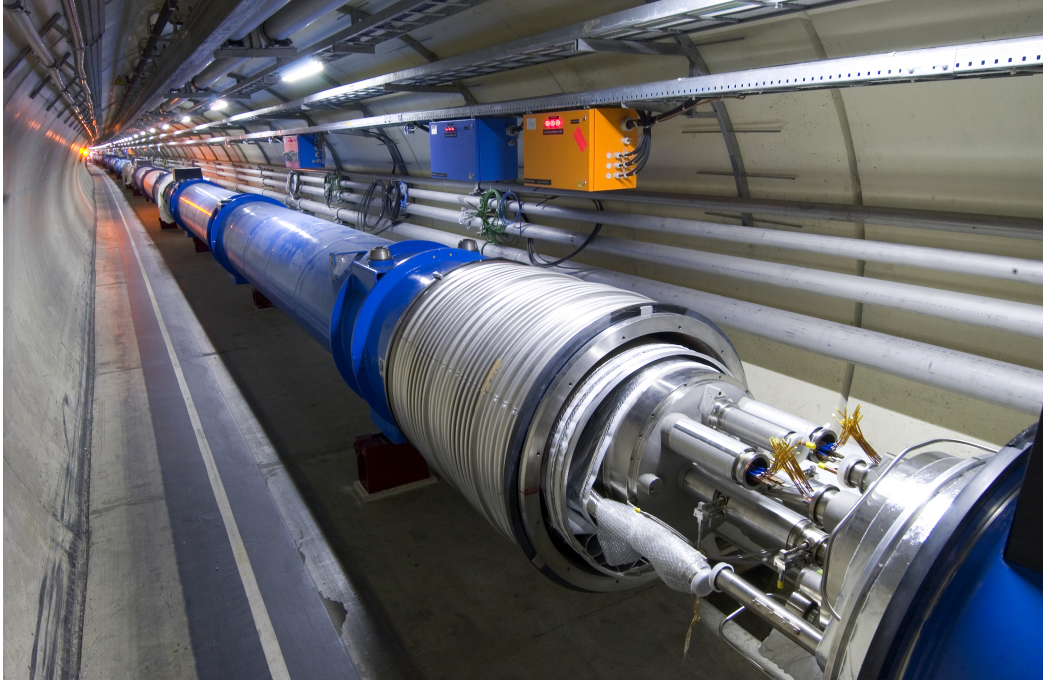


Figure 1.1: View of an open LHC interconnection [7].

The LHC accelerates two different particle beams at around the speed of light in two separate beam pipes. In order to minimize the collisions with gas molecules, the tubes are kept under an ultra-high vacuum³ in the order of 10^{-10} to 10^{-11} mbar [8].

¹W and Z bosons mediate the weak force. The weak force is responsible for the radioactive decay of atoms [4].

²Higgs boson is the quantum particle of the Higgs field. Particles that interact with the Higgs field get mass [4].

³The vacuum system of the LHC consists of 104 km of pipes and it's one of the largest in the world [8].

The trajectory of the particle beams is controlled by a magnetic field that is maintained by superconducting electromagnets. The cabling of their coils is made with superconductive materials that can carry a current of 11kA. This current generates a magnetic field of 8.3T ⁴ but the coil needs to be chilled at -271.3°C to stay in the superconductive state [9].

The acceleration of the particles is controlled by metallic chambers that produce an electromagnetic field known as RF cavities. In total, there are 16 RF cavities in the LHC that are contained in four cylindrical refrigerators, which allow them to operate in a superconducting state [10].

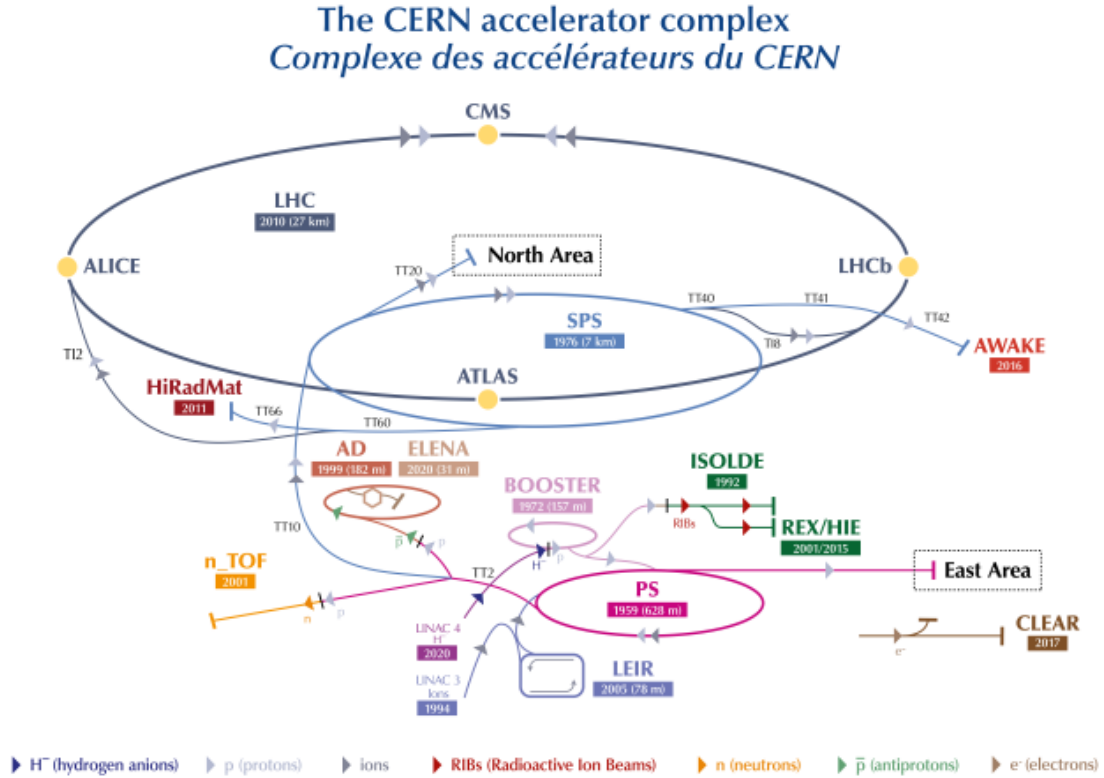


Figure 1.2: The CERN accelerator complex [11].

The CERN accelerator complex consists of interconnected machines that gradually increase the energy of the accelerated particles. The journey of the proton beams starts from the Linear accelerator 4 (Linac4). Linac4 is designed to accelerate negative hydrogen ions⁵ in order to develop energy of 160MeV and enter the Proton Synchrotron Booster (PSB). The ions lose their electrons during their transition from Linac4 to PSB while the protons are accelerated from 160 MeV to 2 GeV for injection to the Proton Synchrotron (PS). The PS propels the proton beams up to 26 GeV and sends them to the Super Proton Synchrotron (SPS) where they are pushed to 450 GeV.

The proton beams are finally pushed to the two pipes of the LHC. They circulate in opposite directions until the LHC ring is full of proton beams with an energy of 6.5 TeV. At this point, the two beams are ready for collisions inside the four detectors (ALICE, ATLAS, CMS and LHCb) where the total energy of the collision stretches out to 13 TeV [12].

⁴More than 100,000 times more powerful than Earth's magnetic field.

⁵hydrogen ions (H^-) are hydrogen atoms with an additional electron

1.3 The CMS Experiment

The Compact Muon Solenoid (CMS) is a general-purpose detector at the LHC. It is designed to observe phenomena that the collisions in the LHC might reveal. CMS can be described as a giant, high-speed camera which captures 3D "photographs" of particle collisions from all directions up to 40 million frames per second.

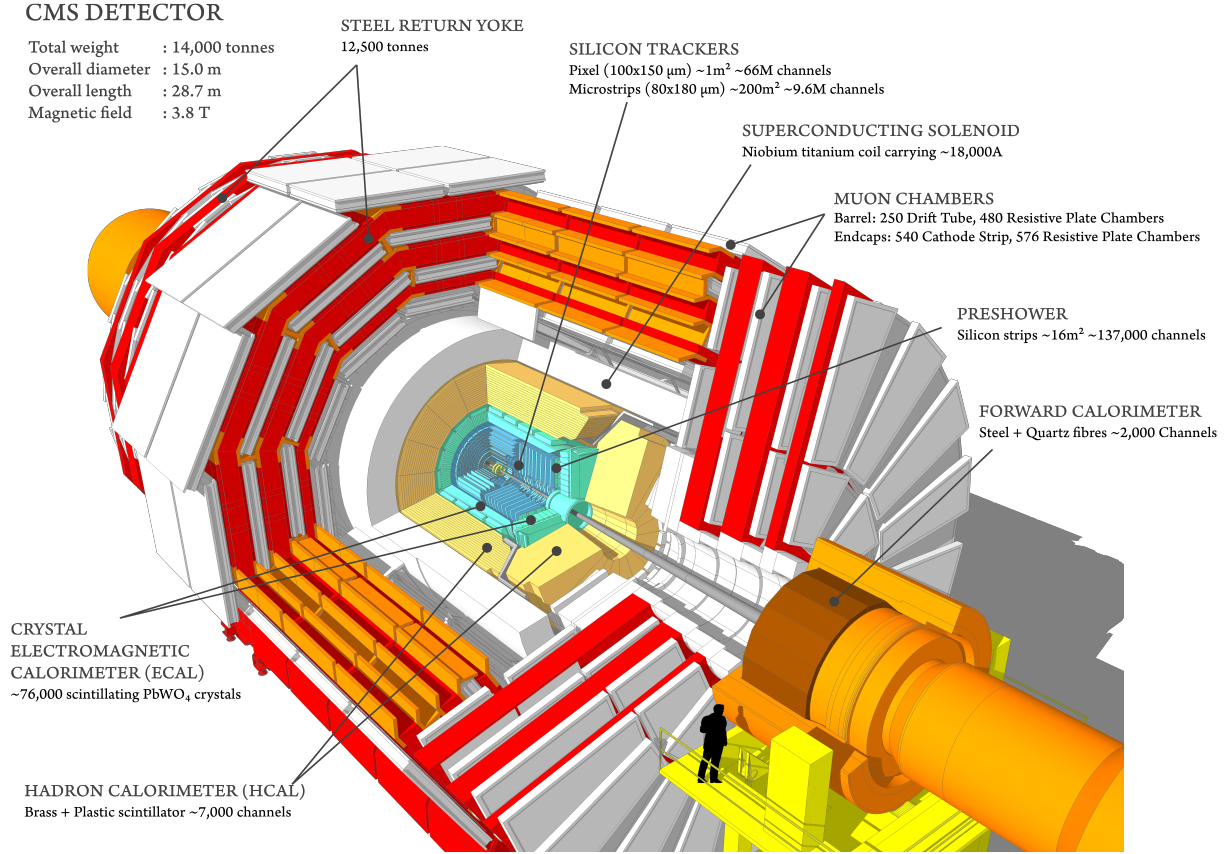


Figure 1.3: 3D model of the CMS detector and its internal components [13].

Even though the majority of the particles produced in the collision are in an "unstable" state, they rapidly turn into a stable state that can be detected by CMS. The identification of the stable particles and the measurement of their momentum and energies, when combined together, can recreate a 3D "image" of the collision.

The CMS can generate a magnetic field of 4 T. It achieves that with the circulation of around 18,000 A in its superconductive coils. The strength of the magnet is used to bend the charged particles as they move away from the collision point. When the trajectories of the particles are bent, the charge and the momentum can be identified since oppositely charged particles bend in opposite directions and high-momentum particles bend less compared to low-momentum ones [14].

1.3.1 The CMS sub-detectors

The CMS detector consists of different sub-detectors that serve different purposes: the Silicon Tracker, the Electromagnetic Calorimeter (ECAL), the Hadron Calorimeter (HCAL) and the Muon chambers.

The CMS Tracker is the innermost part of the detector and it is able to calculate the momentum of the particles. The path is tracked with a magnetic field and it can be reconstructed just from knowing the position in a number of key points. It can reconstruct paths of high-energy muons⁶, electrons, hadrons⁷, and paths deriving from the decay of short-lived particles [17].

The recreation of an event in CMS requires both the energy and the momentum of the particles involved. The ECAL is responsible to measure the energy of electrons and photons by stopping them. This is accomplished by the use of highly transparent crystals that scintillate when electrons and photons pass through. In that way, it generates a light proportional to the particle's energy that is measured using photodetectors [18].

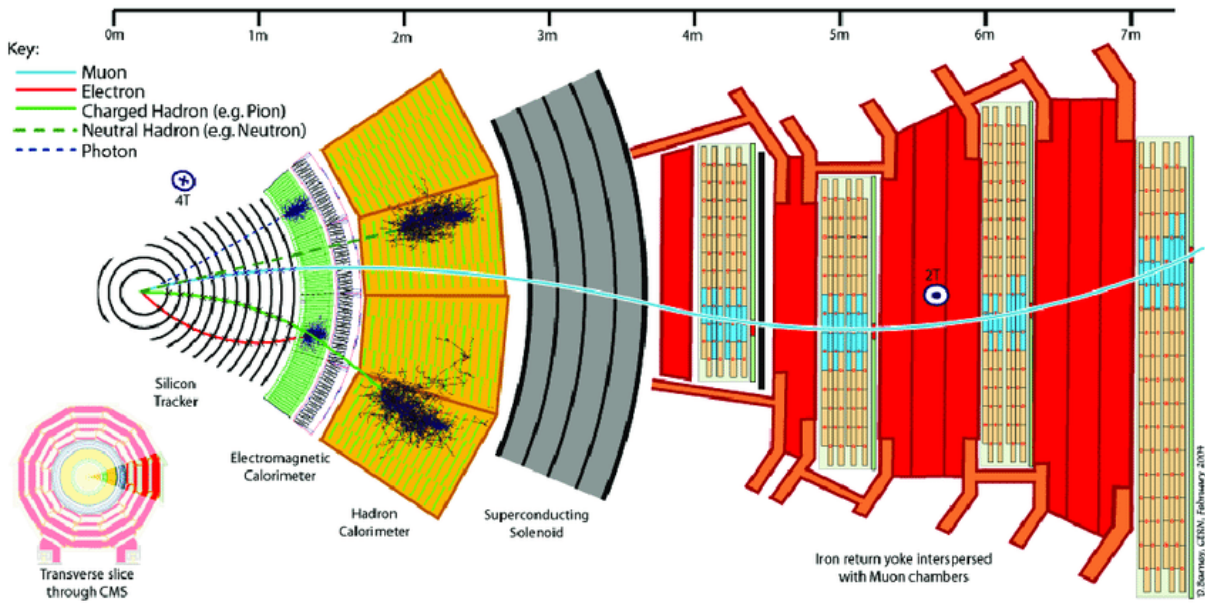


Figure 1.4: A transverse slice through one segment of the CMS detector indicating the responses of the various detecting systems to different types of particles [19].

The HCAL can detect a hadron's energy, position, and arrival time. This is accomplished by using alternating layers of the absorber and fluorescent scintillator materials. When a hadronic particle interacts with a plate of the absorber, produces numerous secondary particles. These particles can also produce numerous particles when interacting with the absorber layers, resulting in a "shower" of particles. As the particles pass through the layers with the fluorescent scintillator materials, they emit rapid light pulses. Optic fibers collect this light and feed it to photodetectors. When the amount of light in a certain region is summed up over different layers, this total amount of light is a measure of the hadron's energy [20].

Detection of muons is one of CMS's most important tasks. Unlike most particles, muons are able to penetrate several meters of iron without interacting, which makes their detection challenging. Therefore, there are four Muon Stations (MS) that sit outside the solenoid where they are the only particles to interact. A particle is measured by fitting a curve to the hits in the muon stations. By tracking its position through the muon stations alongside the tracker measurements for the momentum, CMS can accurately trace a particle's path [21].

⁶Muons are elementary particles with the same charge as electrons but they are 200 times heavier [15].

⁷Hadrons are built from quarks and gluons and they react through the agency of the strong force [16].

1.3.2 The CMS Data Acquisition system

The CMS detector can be described as a 3D camera that captures pictures (events) of the collisions of particles with a frequency of 40MHz. The vast majority of these events are not useful to look at because they don't contain any sign of interesting physics. This is why the events undergo a sophisticated filtering process [22].

Trigger and filtration system

The filtration system consists of a two-level Trigger System. The first level is the *Level-1 Trigger (L1 Trigger)*. The L1 Trigger reduces automatically the event rate from 40MHz to 100kHz. It uses custom FPGA designs that run algorithms capable of distinguishing signs of interesting physics, e.g particles with a large amount of energy. The electronics of the L1 Trigger are located very close to the detector in the underground caverns.

The second level is the *High Level Trigger (HLT)*. The HLT synchronizes data from different parts of the detector in order to recreate a complete event and sends it to a farm of more than a thousand standard computers that are located above ground. The HLT farm PCs run very complex physics algorithms that look for very specific information. This way the event rate is further reduced from 100kHz to 100Hz.

Baseline data acquisition pipeline

The CMS DAQ system consists of various interconnected subsystems. These subsystems are located both on the surface and in underground caverns, and they are connected through optical fibers and links. A high-level diagram of the DAQ system can be seen in Figure 1.5.

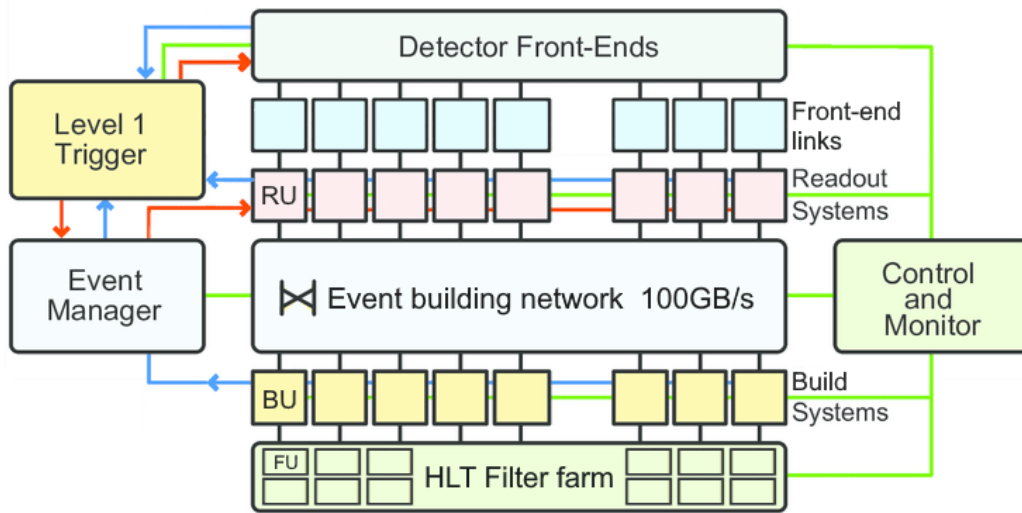


Figure 1.5: High-level diagram of the CMS DAQ system.

The data from the various sub-detectors arrive in the DAQ system from the Front-end driver (FED) and the FED Readout Optical Link (FEROL). The FEROLs collect and pack the raw data fragments from the FED into TCP/IP packets and feed them to the Readout Unit (RU) through the network. The RUs aggregate the data and forward them to the event-building network.

The event building network sends the data packets that were collected from the RUs, to a single Builder Unit (BU). The BU proceeds to assemble the packets into an event. Every BU is responsible for one event, so multiple events are processed in parallel. Finally, the assembled event from the BU is fed to the Filter Unit (FU), which forms the HLT farm.

Timing and Control Distribution System

The DAQ pipeline is controlled and monitored by the Timing and Control Distribution System (TCDS) boards. These boards are responsible to generate and distribute timing commands for the DAQ pipeline and also synchronize the input of data from the sub-detectors. TCDS is also responsible to distribute the LHC bunch clock to all CMS subsystems.

The DAQ and TCDS Hub

In 2026, the CMS DAQ system will initiate its upgrade operation to prepare for the first runs of the High-Luminosity LHC (HL-LHC) in the second half of 2027. The data aggregation, the FEROLs and the TCDS will be incorporated into the new *DAQ & Timing Hub (DTH)*. Some of the main tasks of the TCDS will be the translation of raw data to TCP/IP packets, the distribution of the timing and control signals to the subsystems, and their monitoring [23].

1.4 The High-Luminosity LHC

The operation of the LHC is divided into periods of runs where beams take place and technical stops where there is hardware maintenance and commissioning. The first three runs of the LHC were the *Phase 1*. In 2026, the LHC will move to *Phase 2* and major upgrades will take place in the LHC and its experiments [24].

1.4.1 The LHC upgrade

In 2026, the LHC will be upgraded to an accelerator with increased capabilities, the High-Luminosity LHC (HL-LHC). The upgraded accelerator will operate with higher luminosity in order to allow more particle collisions per bunch crossing⁸. Luminosity measures how capable a particle accelerator is to produce a certain amount of interactions and is proportional to the production cross-section σ_p of rare events and the number of events per second dR/dt [25, 26]:

$$\frac{dR}{dt} = L * \sigma_p$$

More luminosity results in more particle collisions which eventually will lead to more rare events taking place [26].

1.4.2 The DAQ upgrade

More particle collisions result in more data that need to be gathered and processed by the experiments. In order to manage the increased amount of data, the CMS DAQ group has already started designing and manufacturing hardware prototypes that fulfill the requirements of the new accelerator.

⁸A group of particles that travels in the beam pipe is called a particle bunch. When particle bunches collide, there is a bunch crossing.

The infrastructure that was chosen by the CMS experiment and the DAQ group in order to house the new system is the Advanced Telecommunications Computing Architecture (ATCA). The ATCA is an open architecture that provides robust system management and targets telecommunication applications. Its specifications range from mechanical structures to high-speed interconnect technologies and power supply management. The most notable components of the ATCA platform are the Front Panel, the Rear Transition Module (RTM), the Shelf Manager and the Intelligent Platform Management Controller (IPMC) [27].

Multiple boards can be connected on both sides of a shelf. The DTH board serves as the Front Panel board and incorporates CERN's implementation of the IPMC and numerous custom electronics. The RTM hosts the On-Board Controller (OBC) which is a Trenz Mezzanine card⁹ of the Zynq UltraScale+ MPSoC from Xilinx and all the electronics associated with it [28].

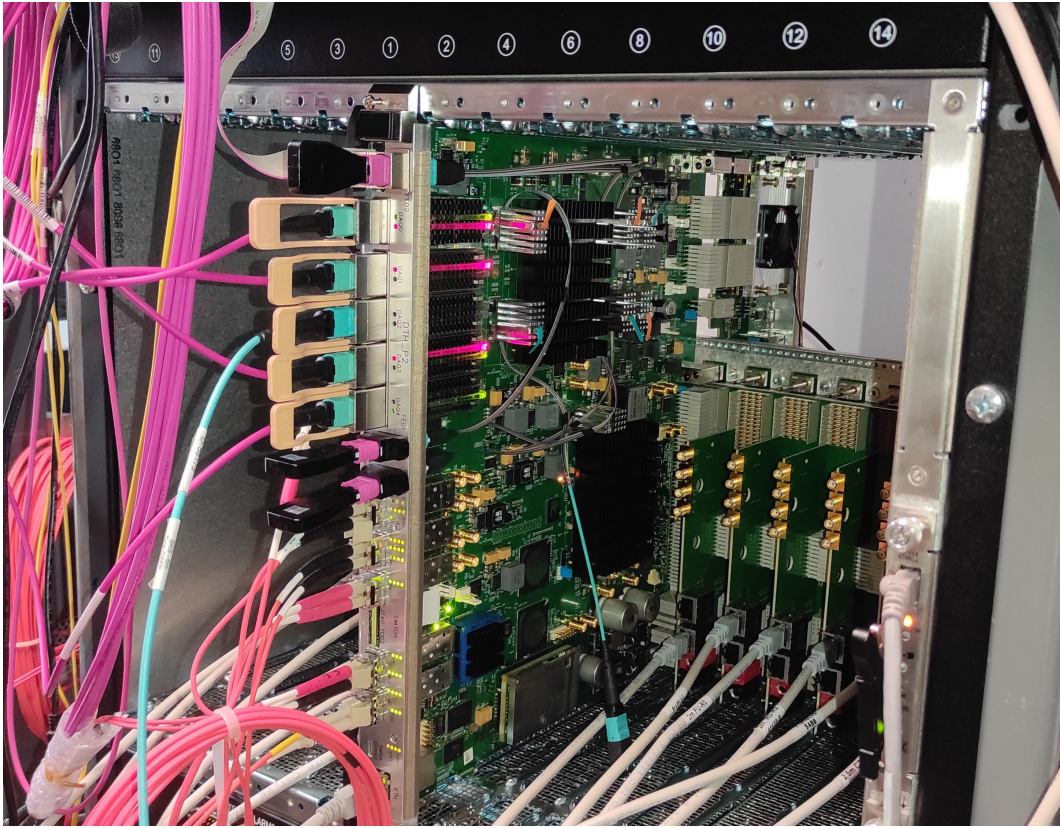


Figure 1.6: The ATCA crate located in the DAQ group's laboratory, with one DTH board sitting in the front panel while the RTM is visible in the back panel, connected with an industrial grey connector with the DTH. The Trenz Mezzanine of Zynq UltraScale+ MPSoC is sitting on the RTM, behind the grey connector, under the fan, and the cooler.

Even though the Zynq UltraScale+ MPSoC could have been placed on the DTH board, eliminating the need for an RTM, the DAQ group decided to place it on the RTM. In this way, the electronics that are directly associated with the Trenz Mezzanine card are all placed on the RTM. This means that in case there is a need to change from one Mezzanine card to another, the DTH is not affected.

⁹A mezzanine card is a printed circuit board that plugs directly into another board.

2 Project Description

2.1 Background

The current data acquisition system uses an abundance of Field-Programmable Gate Array (FPGA) devices and custom hardware for its various operations. This hardware is monitored by rack-based server PCs and is connected with them through PCI bridges¹⁰. The new ATCA-based design incorporates an embedded system that performs control and monitoring tasks on the custom hardware and FPGAs. The use of such an embedded system in this application, eliminates the use of additional rack-based server PCs thus, minimizing the complexity of the system.

The system that was chosen by the DAQ team for this application is the *Zynq UltraScale+ Multi-Processor System-on-Chip (MPSoC)* from Xilinx. A simplified block diagram of the MP-SoC can be seen in Figure 2.1.

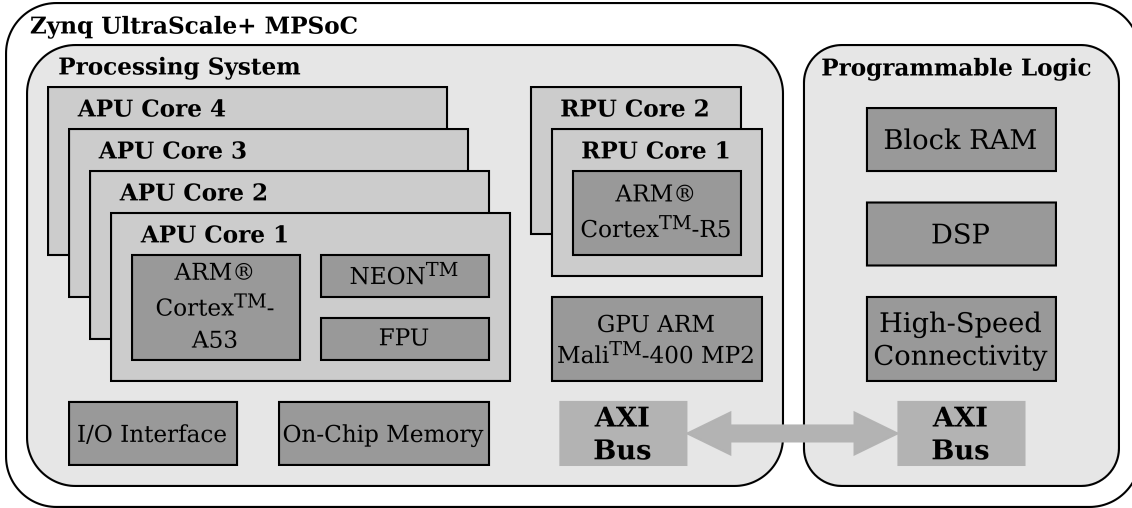


Figure 2.1: Block diagram of the Zynq UltraScale+ MPSoC [30].

This MPSoC contains both a Processing System (PS) and Programmable Logic (PL). The Processing System consists of a quad-core 64-bit ARM processor that is capable of running an embedded Linux distribution that is used by the DAQ group, with control and monitoring software. The PL contains hardware that facilitates the control and monitoring of the FPGAs and custom hardware components. This implementation allows the control and the monitoring to be very close to the hardware which results in better performance.

The Zynq UltraScale+ MPSoC will be placed on the RTM board in the ATCA crate, as it was explained in Section 1.4. There will be around 250 such boards in the DAQ pipeline. These boards consist of very specific components dedicated to their tasks, while the Zynq UltraScale+ MPSoC itself will run a Linux OS with software customized for its needs.

The Zynq UltraScale+ MPSoC from Xilinx is already used by other groups and experiments at CERN and more plan to use it. The High-Luminosity LHC (HL-LHC) upgrade will host thousands of embedded systems using the Zynq UltraScale+ MPSoC.

¹⁰PCI is a local bus standard for connecting hardware devices to each other. A PCI bridge allows connections between multiple PCI buses [29].

2.2 Objective

The DAQ software that will be running on the Zynq UltraScale+ MPSoC devices will need to be extensively tested on each board before its deployment. The feedback for the actual state of the board alongside the validation that the MPSoC booted correctly is of major importance. Hardware and software development for such embedded systems is rather complex since there are several parts in the MPSoC running different software and firmware. This adds extra complexity to the boot process as well, since each piece of software and firmware runs in a predefined order with different privileges and access to hardware resources. Booting and testing 250 boards, that are already complicated systems per se, can possibly introduce various failures. It can become even more difficult, when these boards are located 100 meters below the surface of the Earth, near the CMS Experiment.

The project aims to create an automated CI/CD framework that will use mechanisms to automate the aforementioned procedures. Automating such procedures can lead to more manageable and efficient software development, minimization of the time required to troubleshoot problems, along with more sophisticated and reliable testing. The automation requirement is further analyzed in Section 2.4.

2.3 Requirements & Preconditions

The requirements for the automated CI/CD framework are stated in Table 2.1. Their priority has been described using the MoSCoW analysis. The MoSCoW analysis basically divides the priorities into the following categories [31]:

- Must have:** These requirements are *critical* and *must be met*.
- Should have:** These requirements are *important* but *not necessary*.
- Could have:** These requirements are *desirable* but *not necessary*.
- Will not have:** These requirements are *not appropriate* for this project.

Table 2.1: Project requirements.

#	Requirement	MoSCoW
1	The CI/CD pipeline builds the Vivado project	<i>Must</i>
2	The CI/CD pipeline builds the PetaLinux project	<i>Must</i>
3	The CI/CD pipeline builds the CentOS 8 root filesystem	<i>Must</i>
4	The CI/CD pipeline builds the Board Specific root filesystem	<i>Must</i>
5	Additional services are automatically added to the filesystem	<i>Could</i>
6	Additional kernel modules are automatically added to the filesystem	<i>Could</i>
7	The firmware binaries are automatically deployed to the TFTP server	<i>Should</i>
8	The root filesystem is automatically deployed to the NFS server	<i>Should</i>
9	The CI/CD pipeline power cycles the Zynq UltraScale+ MPSoC	<i>Must</i>
10	The CI/CD pipeline boots the Zynq UltraScale+ MPSoC	<i>Must</i>
11	Dedicated U-Boot bootstraps the SD Card	<i>Could</i>
12	The CI/CD pipeline monitors the boot process	<i>Must</i>

Even though the development of the project took place in the ZCU102 Evaluation Kit that hosts Zynq UltraScale+ MPSoC, the final product can be used with different hardware architectures. The preconditions of the project are described in Table 2.2.

Table 2.2: Project preconditions.

#	Precondition
1	The project is developed on the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit
2	The project is developed using Xilinx Vivado and PetaLinux Tools
3	The Operating System (OS) distribution is CentOS 8 Linux
4	The Zynq UltraScale+ MPSoC will sit on the RTM in an ATCA crate
5	The Zynq UltraScale+ MPSoC boots from an SD Card
6	The Zynq UltraScale+ MPSoC boots through the network with TFTP and NFS server
7	The project is tracked using GitLab
8	The GitLab CI/CD framework is used
9	The project uses official development tools provided by CERN

2.4 Automation Requirement

This automated CI/CD framework will be responsible to automate the building of the software that will be running on the Zynq UltraScale+ MPSoC. This will ease the build process and it won't require recalling all the necessary steps to recreate the end software. The software will be automatically deployed to the board and the CI/CD will be also responsible to boot it. This will minimize the total required time to boot multiple boards and also accelerate the development.

The automated CI/CD framework will monitor the boot process and will verify that it is in the desired state. In that way, the board will be in a state where it can function properly and therefore be extensively tested. Additional testing can be done with more CI/CD pipelines, dedicated to testing specific functions of the MPSoC.

2.5 Final Product

The final product that will be delivered, is an automated CI/CD framework that is compliant with the initial requirements. All the requirements with a *must* priority have to be fulfilled.

The final products will be:

1. Scripts that are able to:
 - Generate the Vivado¹¹ output products
 - Generate the PetaLinux¹² output products
 - Build the Root Filesystem (ROOTFS)
 - Bootstrap the board
 - Boot the board through the network

¹¹Vivado is a development environment provided by Xilinx for hardware design on its chips [32].

¹²PetaLinux is an Embedded Linux development solution for Xilinx Zynq chips [33].

- Verify that the board was booted successfully
2. GitLab CI/CD pipelines that will incorporate the aforementioned scripts
 3. A GitLab CI/CD pipeline that will orchestrate the inputs and the outputs of each pipeline

3 Background Research

The background research refers to the research about the internals and workings of the Zynq UltraScale+ MPSoC as well as its boot process. It also concentrates on the GitLab CI/CD pipelines and their working principle.

3.1 Zynq UltraScale+ MPSoC internals

3.1.1 Hardware overview

The Zynq UltraScale+ MPSoC consists of several processing units, General Purpose I/O (GPIO) peripherals, communication interfaces, and an FPGA. The internals of the chip are grouped into two parts: the Processing System (PS) and the Programmable Logic (PL).

The PS hosts several processing units along with the communication interfaces and the I/O peripherals. It supports different power modes to facilitate integration in different environments. It has an on-chip RAM (OCM) and an interface for external DDR memory. The main components of the PS are:

- Arm Cortex-A Application Processing Unit (APU)
- Arm Cortex-R Real-Time Processing Unit (RPU)
- A Graphics Processing Unit (GPU)
- A MicroBlaze processor for the Platform Management Unit (PMU)
- A MicroBlaze processor for the Configuration Security Unit (CSU)

The PL architecture provides a vast set of functions and resources. As a configurable hardware resource, it comes with gates, clock structures, standard and high-range I/O, Digital Signal Processors (DSP), Look-Up Tables (LUT)¹³, and block RAMs. It also houses PCIe and Ethernet controllers for connectivity as well as several units for image and video processing.

The PS and PL can be connected using various interfaces and signals. This allows the processors to access functions and hardware-accelerators¹⁴ in the PL. The PL configuration happens during or after the PS is booted thus allowing a software-oriented approach for the manipulation of the PL [34].

The complete hardware architecture of the Zynq UltraScale+ MPSoC can be seen in Appendix A.

3.1.2 Application Processing Unit (APU)

The Application Processing Unit (APU) is responsible for executing the bootloaders and eventually the Linux OS. Some of the notable features of the APU is the Armv8-A architecture of the Quad-core Arm Cortex-A53 processing unit at 1.5GHz along with its Floating Point Unit (FPU). A block diagram of the APU is presented in Figure 3.1:

¹³The LUTs hold custom truth tables, which are loaded when the chip is powered on.

¹⁴Hardware-Accelerators are specialized hardware components that are tailor-made to compute specific tasks.

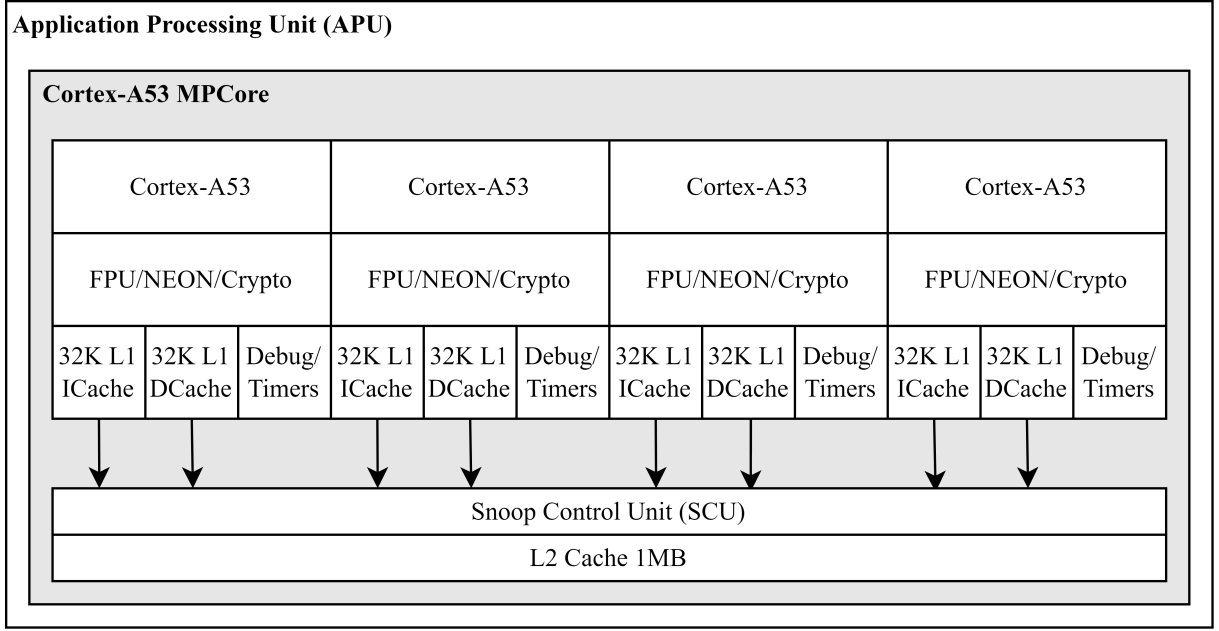


Figure 3.1: Application Processing Unit (APU) block diagram [35].

The APU is connected to the I/O peripherals through the Central Switch. Since the Central Switch is in the Full Power Domain (FPD), there is a Low Power Switch in-between to protect the circuitry and also give access to the Low Power Domain (LPD) peripherals to the I/O (See Appendix A).

3.1.3 I/O peripherals

There is a wide range of I/O peripherals that enable various interface capabilities. They provide connectivity, storage, and high-speed connections. I/O peripherals are essential for the operation of the Zynq UltraScale+ MPSoC since they are used during the boot process and the Linux OS. These I/O peripherals are presented in Table 3.1:

Table 3.1: I/O peripherals and interfaces.

#	Connectivity	External Storage	High Speed
1	SPI	NAND	USB 3.0
2	CAN	SD 3.0	SATA 3.0
3	I2C	eMMC 4.51	PCIe v2.1
4	UART	Quad-SPI	Display Port
5	GPIOs		SGMII Ethernet
6	USB 2.0		
7	Ethernet		

The controllers for the connectivity I/Os exist in the Low Power Domain (LPD) and are connected through the Multiplexed I/O (MIO) interface to external pins and the PL [36].

The controllers for the external storage and the high-speed connections reside in the Full Power Domain (FPD). PCIe and 100Gb/s Ethernet blocks reside in the PL [36].

3.1.4 Platform Management Unit (PMU)

The Platform Management Unit (PMU) contains a triple-redundant MicroBlaze processor. It initializes the system before it boots, and it handles the power and error management along with specific software tests. The PMU firmware that runs on a 128KB RAM is designed in a modular way to provide flexibility. Each feature that has a distinct functionality is incorporated in a respective module so it is loaded only when needed. [37].

3.1.5 Configuration Security Unit (CSU)

The Configuration Security Unit (CSU) is responsible for the secure operation of the Zynq UltraScale+ MPSoC. It incorporates a triple-redundant MicroBlaze processor that is not user-accessible. Some of the most important of its responsibilities is to secure the MPSoC's data, the secure boot, and the secure key storage and management [38].

3.2 Zynq UltraScale+ MPSoC boot process

The boot process of the Zynq UltraScale+ MPSoC involves several stages that use different pieces of software to initialize different parts of hardware [38]. Figure 3.2 shows a simple example of the boot flow:

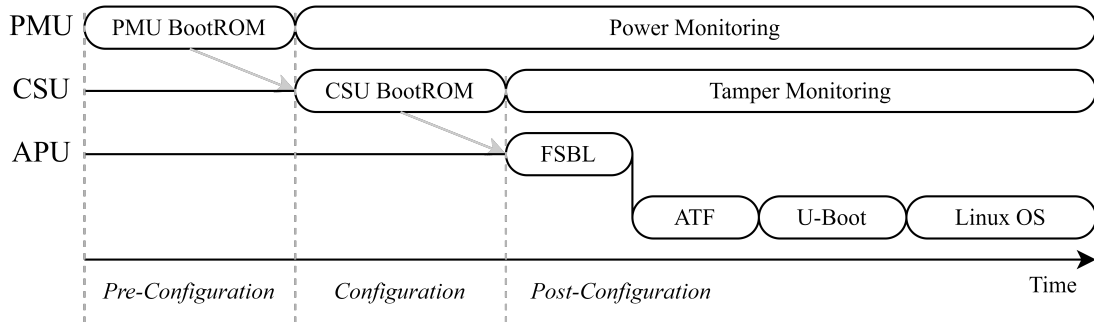


Figure 3.2: Simplified view of the boot process [39].

The boot flow pictured in Figure 3.2 can be described in three functional stages [38,39]:

1. **Pre-configuration stage:** This stage is dominated by the PMU. The PMU BootROM (PBR)¹⁵ starts execution to initialize the system and monitors all wake-up and reset-related processes.
2. **Configuration stage:** In this stage, the CSU BootROM (CBR) takes over, it interprets the boot image which contains the First-Stage Bootloader (FSBL), and loads it in the on-chip RAM (OCM) of the processor.
3. **Post-configuration stage:** This stage is entered when the FSBL starts its execution. The CBR at this stage enters the tamper monitoring and response mode and the system continues with the boot process.

¹⁵It is the first code that is executed after power-on and is stored in the ROM.

The boot image is a binary file that consists of the boot header and partitions for the various images along with a partition header. An example of such images is the FSBL, the PMU firmware, or an application for the RPU [40].

The pre-configuration and configuration stages execute code that is stored in the MPSoC's ROMs cannot be changed (see Appendix D). The post-configuration stage is of major importance since the user has more flexibility over design choices.

3.2.1 The First-Stage Bootloader

The primary objective of the FSBL is to load the Second-Stage Bootloader (SSBL) which is Universal Bootloader (U-Boot). Except for this, several tasks that are handled by the FSBL such as memory and device initialization, the configuration of the PS, or configuring the ARM Trusted Firmware (ATF). When all of the secondary tasks are completed, the FSBL moves on with updating the Program Counter and handing over control to U-Boot. A more detailed description of the FSBL is explained in Appendix E.

3.2.2 The ARM Trusted Firmware

The main objective of the ATF is to ensure that all the software running on the MPSoC has access only to necessary resources. For example, the Linux OS should not have access to the region where the public key in the MPSoC is stored [41].

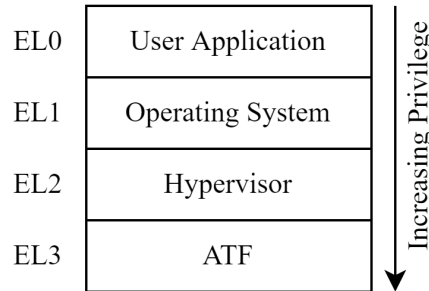


Figure 3.3: Exception levels [42].

Since the ATF is initialized by the FSBL, it is now ready to be used by U-Boot and the Linux. U-Boot starts immediately after the FSBL since the program counter is set to its starting address and runs at EL2 [43].

3.2.3 The Second-Stage Bootloader

The Second-Stage Bootloader (SSBL) is responsible for initiating the boot process of Linux on the Zynq UltraScale+ MPSoC. Universal Bootloader (U-Boot) is a universal bootloader and is used for this application.

U-Boot comes with a Command-Line Interface (CLI) that is visible through the serial port of the Zynq UltraScale+ MPSoC. It provides several commands for R/W operations to the various memory types, network interface, manipulating the hardware, and much more. These operations are triggered by environmental variables that can store sequences of U-Boot specific commands [44–46].

```
1 U-Boot 2019.01 (Mar 18 2022 - 09:25:38 +0000)
2 Model: CUSTOM BOARD CERN
3 Board: Xilinx ZynqMP
4 DRAM: 4 GiB
5 EL Level: EL2
6 Chip ID: zu9eg
7 Loading Environment from SPI Flash...
8 In: serial@ff000000
9 Out: serial@ff000000
10 Err: serial@ff000000
11 Model: CUSTOM BOARD CERN
12 Board: Xilinx ZynqMP
13 Net: ZYNQ GEM: ff0e0000, phyaddr c, interface rgmii-id
14 Warning: ethernet@ff0e0000 using MAC address from ROM
15 eth0: ethernet@ff0e0000
16 U-BOOT for CERN CMS
17 Hit any key to stop autoboot:
18 ZynqMP>
```

Figure 3.4: U-Boot console output during booting.

The main objective of U-Boot is to load the Linux Kernel and the Device-Tree to the processor’s memory. There are several configurations in U-Boot that allow the booting of the Linux Kernel in different ways. The simplest way is to store the Linux Kernel Image, the Device-Tree Blob (DTB) and the root filesystem to a local storage media. However, U-Boot also supports networking interfaces. It can retrieve the images from a TFTP server while also using the Linux root filesystem on an NFS server [47, 48].

After U-Boot has retrieved the Kernel image and DTB, it will load them in the memory. In the end, U-Boot may pass the boot arguments for the kernel command line and let the Linux Kernel boot.

3.2.4 The Linux Kernel boot

Once the control is handed over to the Linux Kernel, it extracts itself from its compressed version. The Linux Kernel is provided with the address of the Device-Tree Blob (DTB) which can be described as a “map” of the underlying hardware. Since the Linux Kernel has access to the I/O peripherals that are outside of the MPSoC, the Device-Tree must contain information not only for the MPSoC’s hardware itself but also for the peripherals that are directly connected to it [49, 50].

The drivers and the modules that were configured during the building of the Kernel are being enabled while the Kernel is unfolding. Then, the Kernel moves on with the mounting of the root filesystem. It will mount the root filesystem according to the information provided in the command line as boot arguments.

When the root filesystem is mounted, the kernel is freeing all of its unused memory in the RAM, and the *init* process is started. It has a Process ID (PID) of 1 and it’s the parent of all the other processes. In CentOS 8, the *init* process is *systemd*. It handles the startup, the activation of the configured services, and the user login to the userspace [51, 52].

```

1 ## Flattened Device Tree blob at 10000000
2   Booting using the fdt blob at 0x10000000
3   Loading Device Tree to 0000000007ff4000, end 0000000007fff39a ...
   OK
4 Starting kernel ...
5 [ 0.000000] Booting Linux on physical CPU 0x000000000000 [0x410fd034]
6 [ 0.000000] Linux version 4.19.0-xilinx-v2019.2 (oe-user@oe-host) (
   gcc version 8.2.0 (GCC)) #1 SMP Mon Mar 14 09:20:10 UTC 2022
7 [ 0.000000] Machine model: CUSTOM BOARD CERN
8 ... (Kernel loading) ...
9 [ 9.966437] VFS: Mounted root (nfs4 filesystem) on device 0:17.
10 [ 9.972698] devtmpfs: mounted
11 [ 9.976180] Freeing unused kernel memory: 2112K
12 [ 9.999557] Run /sbin/init as init process
13 [ 10.010649] random: fast init done
14 [ 10.249051] systemd[1]: systemd 239 (239-51.el8_5.2) running in
   system mode.
15 [ 10.272170] systemd[1]: Detected architecture arm64.
16 Welcome to CentOS Linux 8!
17 ... (CentOS 8 root filesystem configuring) ...
18 CentOS Linux 8
19 Kernel 4.19.0-xilinx-v2019.2 on an aarch64

```

Figure 3.5: Kernel console output during booting.

3.2.5 Summary

The boot process of the Zynq UltraScale+ MPSoC with Linux as an OS is a complex procedure with several steps. An abstract view of the boot process is pictured in Table 3.2. The various processors along with the FPGA require a methodical boot process that will not only provide flexibility to the user demands but will also deliver a reliable solution to this complex process. The CMS DAQ group will use the in-house developed Reliable Booting System [53].

Table 3.2: Abstract boot process.

#	Step Description
1	Chip powers ON
2	PMU BootROM code executes and CSU reset is release
3	CSU reads the boot mode pins and the Boot Image is fetched
4	CSU reads the Boot Header from the Boot Image
5	CSU loads FSBL to the OCM
6	FSBL configures the PS and PL
7	FSBL loads ATF and U-Boot to the DDR memory
8	FSBL initializes the ATF and hands over control to U-Boot
9	U-Boot loads Kernel Image and Device-Tree blob in the DDR memory
10	Linux Kernel boots up using information from the Device-Tree
11	Linux Kernel mounts the root filesystem
12	The root filesystem executes systemd service manager

An example boot flow can be seen in Appendix F.

3.3 Continuous Integration & Continuous Delivery/Deployment

3.3.1 Continuous Integration (CI)

Continuous Integration (CI) is a technique that automates the integration of the code changes in a single software project. The software is built automatically and it is tested. CI usually refers to the build stage of the process and its goal is to find bugs quicker and minimize the time it takes to validate a project by automating the build process. An automated build process ensures that every time the software is built, it is done under the same preconditions and follows the same steps [54].

3.3.2 Continuous Delivery/Deployment (CD)

Continuous Delivery is a technique that expands upon CI and deploys all code changes into a testing environment after the build stage. If the code changes are pushed to the production environment then it can also be called Continuous Deployment. These two terms are sometimes used interchangeably as Continuous Delivery/Deployment (CD). CD lets the developer automate the testing to verify that the application is working correctly [55].

3.3.3 GitLab CI/CD Pipelines

GitLab is a tool that facilitates the development and maintenance of software. CERN uses its own instance of GitLab (gitlab.cern.ch) as its tool to develop and maintain software. GitLab offers its own CI/CD environment that is used for this project. CI/CD can also be described as numerous scripts that run in a predefined order with features that facilitate automation. The GitLab CI/CD is comprised of two main components [56]:

- **Jobs:** that clarify *what* to do. For example, a job could be the building of the FSBL.
- **Stages:** that clarify *when* to run the jobs. For example, the building of the boot images needs to be done after the images are built.

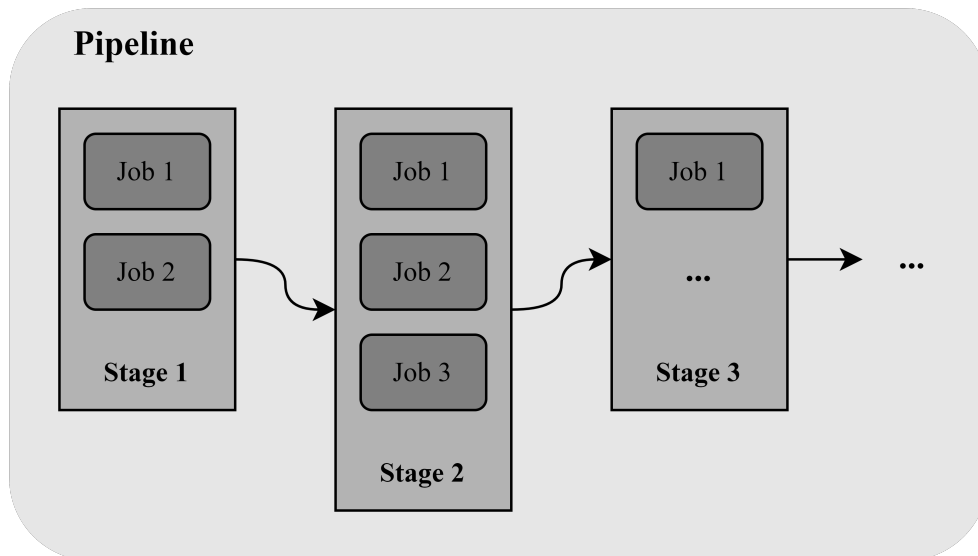


Figure 3.6: Example of the pipeline structure.

A stage can contain multiple jobs that can run either serially or in parallel. The most important rules that govern CI/CD pipelines are the following [56]:

- *Every* job in a particular stage needs to succeed in order to move on to the next stage.
- In case *any* of the jobs in a stage fails, the next stage is not executed and the pipeline ends prematurely unless it is configured differently.

A GitLab CI/CD pipeline is associated with its GitLab repository. It has direct access to all the files in the repository. The configuration of the CI/CD pipeline is done through the `.gitlab-ci.yml` file which has to be created. In this file, the pipeline is configured with respect to the requirements of the application. The jobs of the pipeline are configured in this file. Examples of such configurations are the script of the job, the artifacts when to run this job, the tag of the job, and the stage of the job [57].

3.3.4 GitLab Runners

The various jobs that are in a stage are executed by *Runners*. GitLab Runners are open-source applications that work with GitLab CI/CD to run the jobs of the pipeline. They can be installed on different machines with different OSs. GitLab Runners offer a range of different executors that can be used to run the jobs in different scenarios. For example, an GitLab Runner might use the Linux Shell as an executor while another GitLab Runner might use a Docker Container¹⁶ [59].

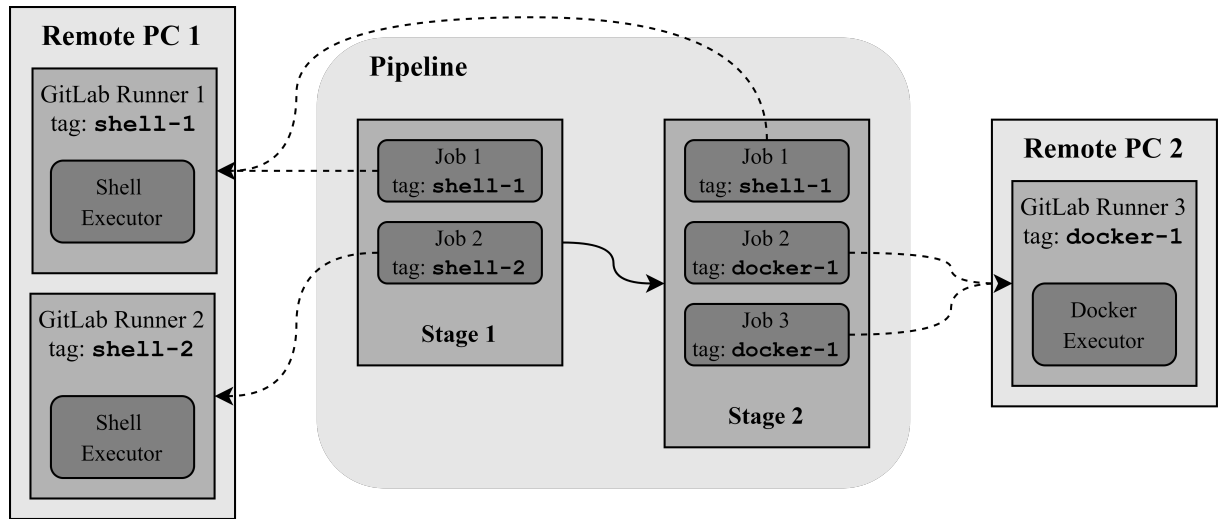


Figure 3.7: Example of GitLab Runner picking up jobs.

The GitLab Runner also offers the possibility to use individual tags. Jobs that use these tags can be executed by the corresponding Runners. This can be very useful for example, in case an application needs to run inside a Docker Container. A GitLab Runner with a Docker Executor will have a specific tag that this job can use in its configuration. When the GitLab Runner sees this job, it will pick it up [59].

¹⁶Docker Container is a standalone software that contains all the necessary equipment to run an application [58].

4 Research and analysis

4.1 Linux OS on Zynq UltraScale+ MPSoC

The previous Section 3.2 was focused on how the several steps of the Zynq UltraScale+ MPSoC boot process. This section focuses on identifying the various images and their build process.

4.1.1 Overview

The images used in the boot process and the operation of the Zynq UltraScale+ MPSoC concern not only different timestamps in the boot process but also different parts inside the MPSoC itself.

Starting from the PL, it is programmed as every other general-purpose FPGA. A *bitstream*¹⁷ or *bit file* is needed with the appropriate configuration for the internals of the FPGA. In order to generate such a file, the Vivado Design Suite from Xilinx is used.

Like the PL, the configuration of the PS is done through Vivado. Vivado is responsible for generating the Xilinx Shell Architecture (XSA) file which is also referred to as Hardware Description File. It contains information about the configuration of the PS as well as for the underlying hardware that is later used by PetaLinux to build the necessary images and the Device-Tree [60].

Moving towards the programming of the PS parts, there are several images that need to be built in order to boot and operate the board. PetaLinux tools from Xilinx are the development environment for the images that run both on the PMU (PMU firmware) and the APU (FSBL, ATF, U-Boot, Linux Kernel).

During the normal operation of the MPSoC, the kernel and the root filesystem are necessary. Both of them are handled initially by the PetaLinux Tools. For the root filesystem, several other in-house developed tools are used in order to increase its capabilities and make it useful.

4.1.2 Hardware Platform Configuration

The configuration of the PL and the PS is done through Vivado Design Suite. Vivado Design Suite is an Integrated Development Environment (IDE) provided by Xilinx. It provides tools for synthesis, analysis, and implementation of Hardware Description Language (HDL) designs and has features for configuring SoCs. The several tools of Vivado are written in native Tool Command Language (Tcl)¹⁸. This gives the users the flexibility to add their own Tcl scripts in order to modify the design or to add extra capabilities to the tools [61].

Vivado gives the opportunity to the developers to work both with the IDE and with Tcl commands. It includes a Tcl Console and a Tcl shell¹⁹ so users can interact with their designs either with the CLI or with scripts [63].

Vivado supports four different project types: RTL²⁰, Post-Synthesis, I/O Planning and Imported Projects. Each of these project types supports different input source files. This project focuses on RTL Projects [65].

¹⁷A bitstream is a file that contains the configuration information for an FPGA.

¹⁸Tcl is a high-level, general-purpose, interpreted dynamic programming language.

¹⁹Computer program which allows the system's services to be exposed to the user or other programs [62].

²⁰Register-Transfer Level (RTL) models a synchronous digital circuit [64].

Design flow

The Design Flow of an RTL project can vary between different users. An abstract RTL project workflow the following steps are followed (see Figure):

1. **PS Configuration:** The processing system is initialized. The several clocks and the peripherals of the PS that are used in the specified project are configured.
2. **RTL Design:** Source files written in an HDL are also used to create the project along with the Xilinx Design Constraints (XDC) files. The source files can be either written by the developer or can be IPs provided by Xilinx. The XDC file is responsible for informing the software which pins are used and how they are configured. Every pin that interacts with external devices needs to be configured in the XDC file [66].
3. **Synthesis:** During this process, an HDL design is converted into a netlist²¹. Every part of the HDL code is being translated into electrical interconnected components that exist in the FPGA, such as LUTs, BRAM, and Flip-Flops [67].
4. **Implementation:** This process is responsible for placing and routing the synthesized netlist generated in the previous step, on the device. This is done under all the timing, physical and logical constraints of the design [66].
5. **Hardware Generation:** This is the final stage where the bitstream and the XSA file are generated.

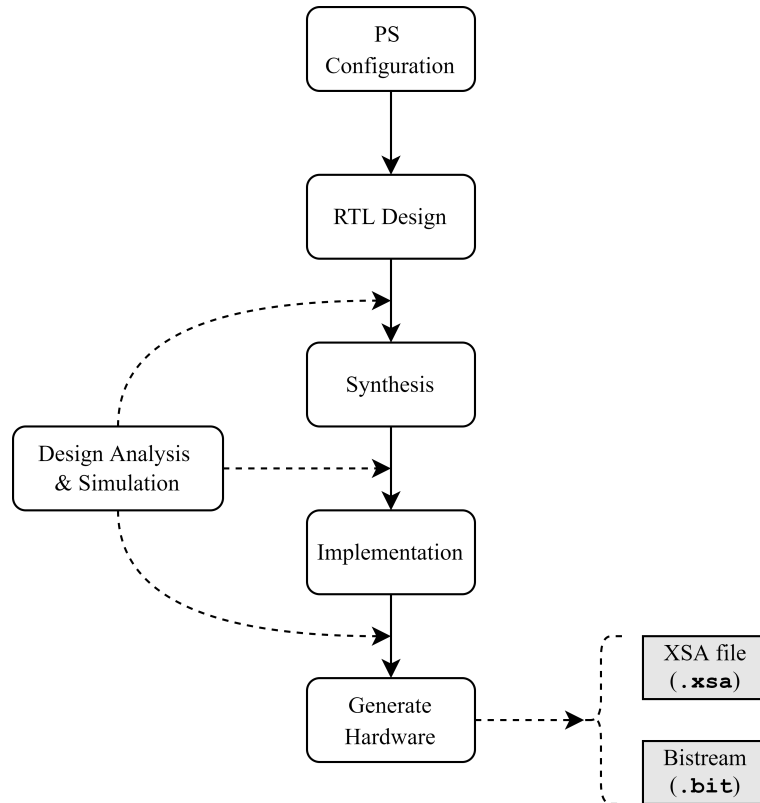


Figure 4.1: Abstract view of Vivado workflow [63, 66].

²¹Netlist is the description of an electronic circuit with gate levels and components [67].

There are other two important steps in the workflow for a Vivado RTL project. The Design Analysis and Simulation can be used to verify the design in terms of logic circuitry or timing or even power in order to improve the circuit performance. They can be executed at different times in between the aforementioned steps [66,68].

In summarizing, this step is an abstract Vivado workflow that generates the aforementioned bitstream and XSA files. The bitstream is a bit file that is streamed into the PL of the MPSoC to program it. The XSA file is used to build the software platform.

4.1.3 Software Configuration

The configuration of the software that boots and runs on the MPSoC is done through the PetaLinux tools. PetaLinux is provided by Xilinx and incorporates all the necessary tools to build, customize and deploy Embedded Linux solutions on Xilinx SoCs.

PetaLinux Tools are based on the Yocto Project. Yocto Project is an open-source set of tools that facilitates the creation of custom Embedded Linux distributions. PetaLinux utilizes the Yocto Project and provides a more simple solution that hides the complexity of directly interfacing with the Yocto Project [69].

A built embedded Linux system for the Zynq UltraScale+ MPSoC is composed of:

- PMU Firmware
- First-Stage Bootloader (FSBL)
- ARM Trusted Firmware (ATF)
- Device-Tree
- U-Boot
- Linux Kernel
- PetaLinux Root Filesystem (ROOTFS)

The aforementioned components are necessary for the boot process and the normal operation of a Linux OS on the Zynq UltraScale+ MPSoC. Their usage is further described in Section 3.2.

Project Creation

The XSA file was generated from Vivado in order to create a PetaLinux project. A Board Support Package (BSP) can also be used to create a PetaLinux project. A BSP is a template to create new PetaLinux projects and contains a pre-configured project and pre-built images. To create a BSP a PetaLinux project needs to be created using the XSA file for initialization [70].

The XSA file contains information about the configuration of the PS and the PL. Hence, it is used for the creation of the FSBL and the Device-Tree. The FSBL is responsible for the initialization of the PS while the Device-Tree is mapping the underlying hardware [70,71].

Design flow

A PetaLinux project workflow is divided into four different stages: *Creation*, *Configuration*, *Build*, *Package*. The stages are described below:

1. **Creation:** In this stage, the PetaLinux project is created. There are two separate workflows that allow the creation of the project either with a BSP or an XSA file.
2. **Configuration:** In this stage, the specified project is customized. Two workflows are provided in this stage. In the first one, the user is able to import a new XSA file in order to update the project to reflect the new hardware configuration. This can happen if the project is already created and after some time a new hardware configuration arrives. The latter one, allows the customization of the aforementioned components to fulfill the project needs i.e add a new variable to the U-Boot environment.
3. **Build:** This stage is responsible to build the embedded Linux system. It also allows to build just a specified component and not the whole system itself. In this stage, the Yocto Project is used underneath. The images that are built in this stage are shown in Table 4.1.

Table 4.1: Built images.

#	Component	Binary image
1	PMU Firmware	pmufw.elf
2	FSBL	fsbl.elf
3	ATF	bl31.elf
4	U-Boot	u-boot.elf
5	Device-Tree	system.dtb
6	Linux Kernel	Image
7	Compressed ROOTFS	rootfs.tar.gz

4. **Package:** In this stage, some of the images of the *Build* stage are packaged in a format that can be deployed. This file is also called the *boot file* or **BOOT.bin** because it is used to boot the MPSoC. Depending on the application the contents of the **BOOT.bin** vary. For example, if the bitstream is fetched from the network during the boot process or the application does not use one, it is not needed in the **BOOT.bin**. For this project, the **BOOT.bin** is composed of the following images:

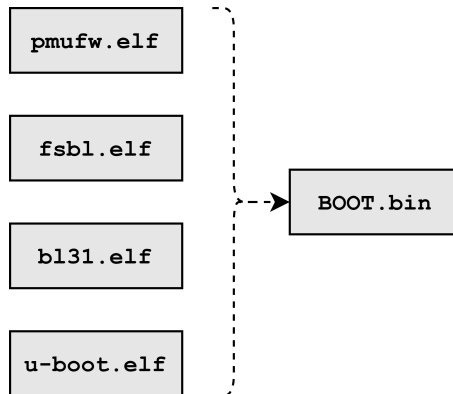


Figure 4.2: Packaged boot file.

At this point, the PetaLinux project has been created and all the necessary images to boot

and operate the Zynq UltraScale+ MPSoC are generated. A more abstract view of the PetaLinux project workflow is seen below in Figure 4.3

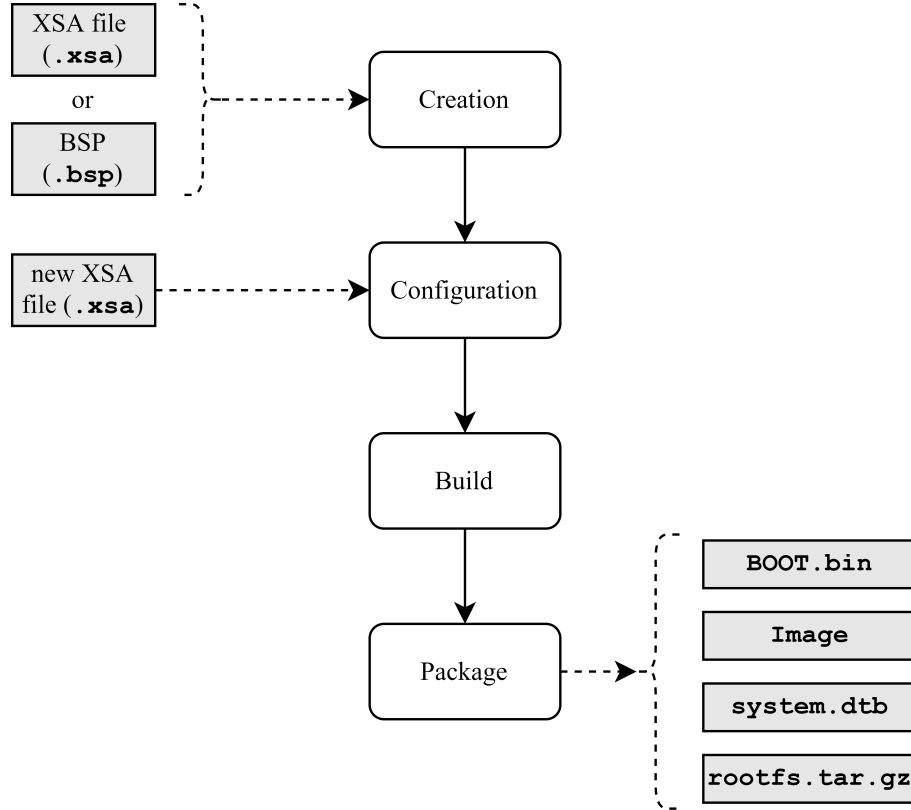


Figure 4.3: PetaLinux workflow [69].

The filesystem that is created in the Build stage of the PetaLinux workflow is later used to generate the ROOTFS of the Zynq UltraScale+ MPSoC.

The PetaLinux tools also have the ability to boot the board in JTAG mode using the Xilinx Software Command-line Tool (XSCT). The XSCT is an interactive CLI that is based on Tcl. It allows among other things to download and run applications on the hardware [72].

4.1.4 Root Filesystem generation

At the top of the hierarchical file-tree is located the ROOTFS. It contains the directories and files that are critical for booting and operating the system. It contains the device directory and all the necessary services that are needed to make the system usable [73].

The Linux OS distribution that will be running on the Zynq UltraScale+ MPSoC, is based on CentOS 8. CentOS is a free and open-source piece of software that is community-supported and it is aiming to be compatible with Red Hat Enterprise Linux (RHEL). Both CentOS and RHEL are usually used in server applications [74].

The ROOTFS that will be used on the Zynq UltraScale+ MPSoC will be CentOS 8 with a set of additional services and tools required by the DAQ group and the MPSoC itself. The additional services required by the DAQ group are downloaded during the creation of the ROOTFS while the tools required by the MPSoC in order to function properly are taken from the original ROOTFS created by the PetaLinux project. In this way, a Board Specific root filesystem

(BSROOTFS) is created that can run on the Zynq UltraScale+ MPSoC family.

Design flow

The computer architecture of the Zynq UltraScale+ MPSoC is `aarch64`. This results in a more complex creation of the ROOTFS since most commercial PCs use `x86` or `x86_64` architecture. There are several ways to overcome this problem. The most commonly used is the use of a *cross-compiler* or an *emulator*. The *cross-compiler* is a program that runs on one processor architecture but generates code for another processor architecture. The emulator allows to run and execute binaries of another architecture. This is done through built-in native compilers [75].

The solution that was chosen is to use QEMU. QEMU is an open-source machine and userspace emulator which allows emulating a different machine in software. More specifically, the `qemu-user-static` package allows running binaries of another architecture in the same operating system as the current one. QEMU is a very widespread emulator and it is already used by the DAQ group in other projects [76,77].

The workflow for the creation of the BSROOTFS consists of three different stages:

- **Create generic CentOS 8:** In this stage, the generic CentOS 8 ROOTFS is created. QEMU is used in order to be created for `aarch64` architecture.
- **Install DAQ services:** In this stage, the required packages by the DAQ group are installed using the package manager.
- **Create BSROOTFS:** In this stage, the tools from the PetaLinux ROOTFS are copied into the CentOS 8 ROOTFS that was created in the previous steps in order to create the BSROOTFS. Since the tools were created with the PetaLinux tools they are already compiled for `aarch64` architecture and no more specific actions are required.

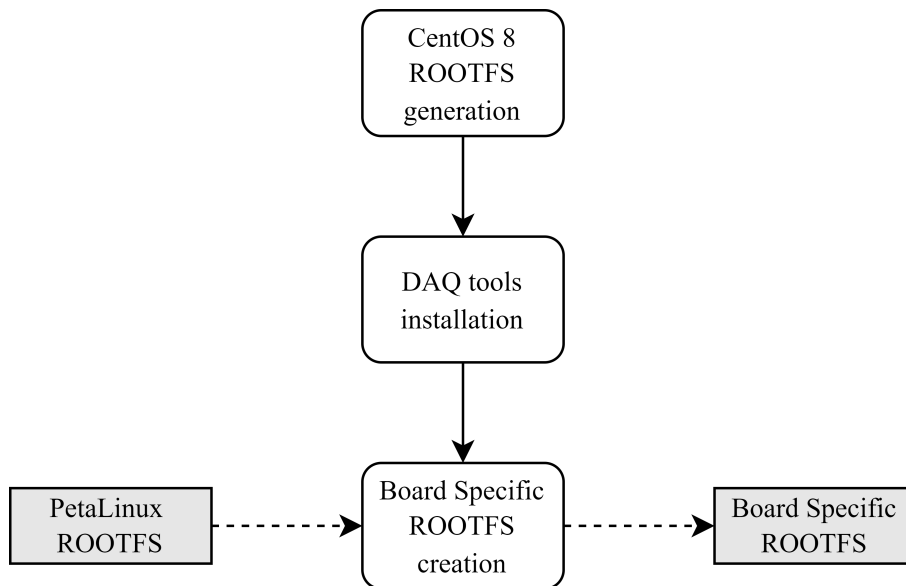


Figure 4.4: BSROOTFS creation workflow.

The BSROOTFS has now been created. All the required components to boot and operated have been analyzed. The next step is to integrate them all together.

4.1.5 Integration

The previous steps combined together create the complete workflow for the generation of all the required components to boot and operate the board. A graphical representation is shown in Figure 4.5

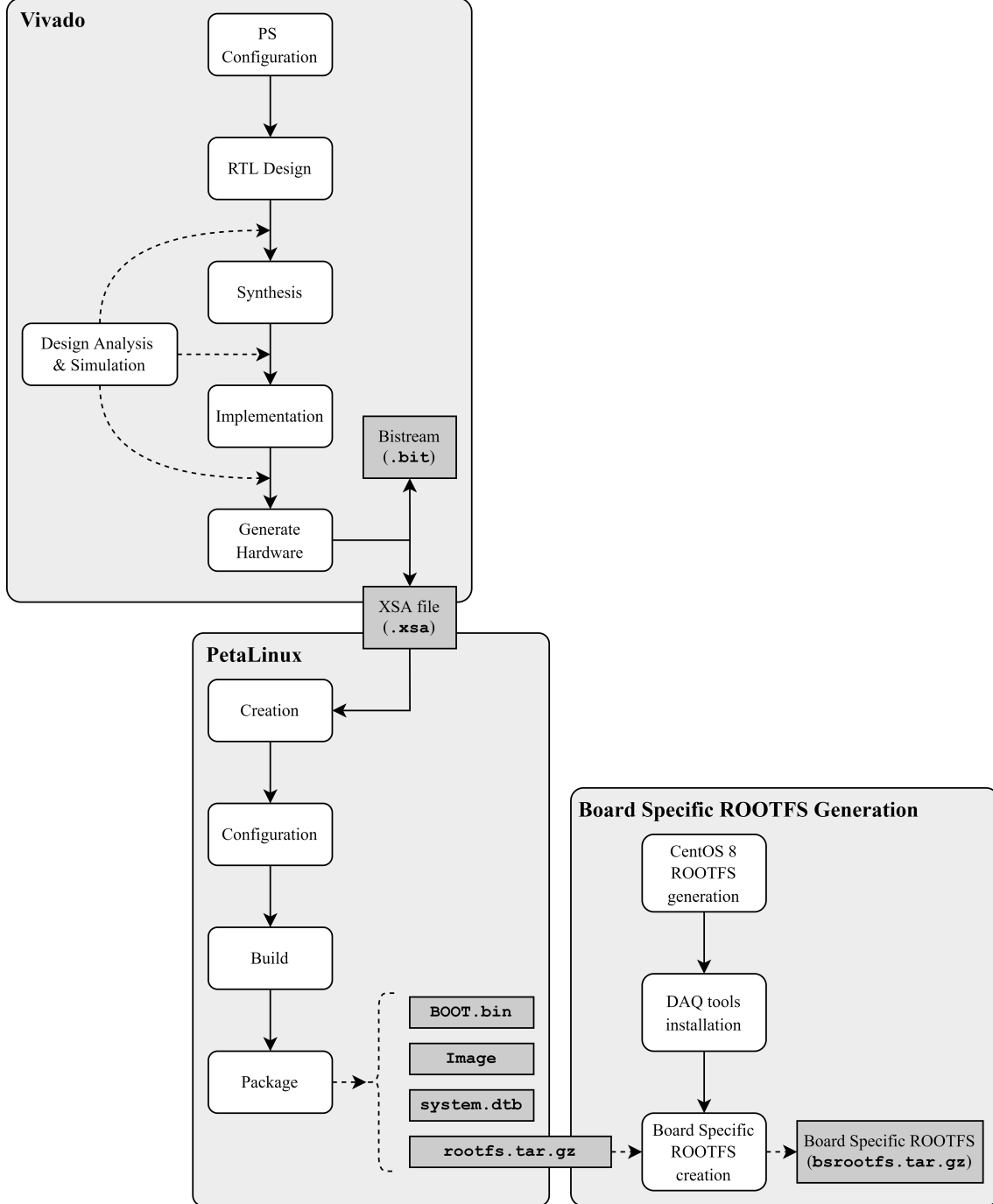


Figure 4.5: Integration.

At this point, all the images are ready to be deployed to their final destinations such that they can boot the board. Depending on the application this may vary.

4.1.6 Bootstrapping

As it was mentioned in preconditions No.5 and No.6 in Table 2.2, the Zynq UltraScale+ MPSoC boots from an SD Card and through the network. The `BOOT.bin` which is responsible for the first part of the boot process is stored in the SD Card. The boot mode of the board is set to SD Card boot mode so when it powers on, the CSU fetches the `BOOT.bin` image from the SD Card.

The Device-Tree, Kernel, PL bitstream, and the boot image (`system.dtb`, `Image`, `<name>.bit` and `BOOT.bin` respectively) are stored on a TFTP server. TFTP is a simple protocol for exchanging files through the network. It is typically used for downloading boot images to remote devices [48, 78].

The BSROOTFS is stored on an NFS server. The NFS server runs on a remote PC and it is accessed through the Network. This allows the system to run without a local disk. NFS servers are commonly used in applications with disk-less systems [79].

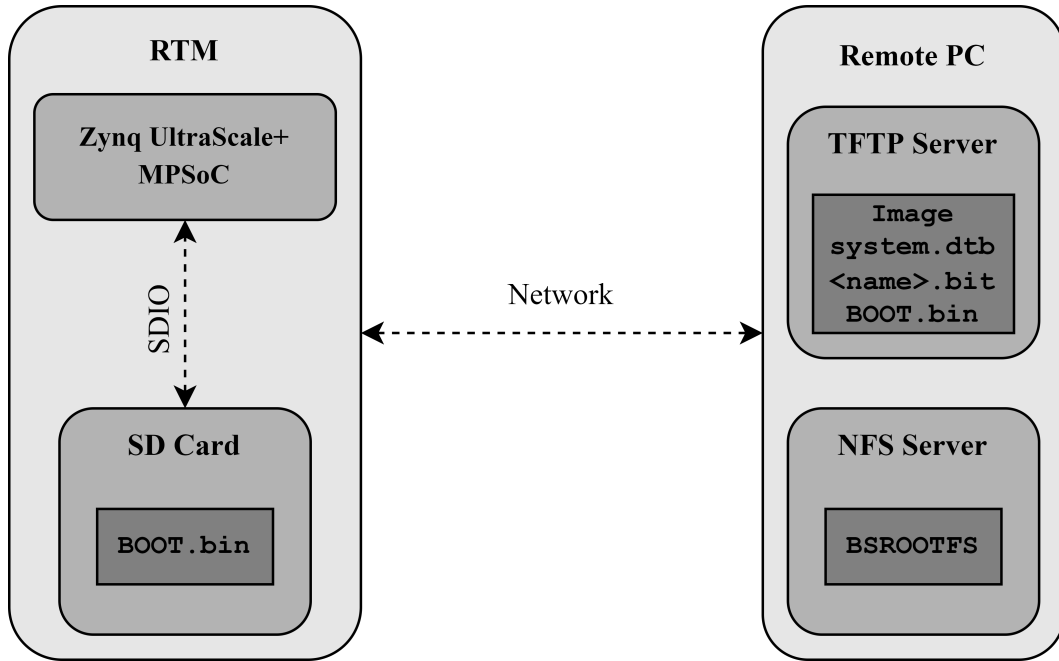


Figure 4.6: Bootstrapping of the Zynq UltraScale+ MPSoC for normal operation.

When the Zynq UltraScale+ MPSoC is placed on top of the RTM and everything is connected to the final system, the Intelligent Platform Management Controller (IPMC) should be responsible for the power cycling of the RTM and manage the boot mode of the Zynq UltraScale+ MPSoC [28].

During the development of the project, there are phases that neither IPMC nor RTM is available. This means that a solution should be proposed to make this system useful in all different stages of the development of a project meaning that it will substitute IPMC's control of the power cycling and management of the boot mode. This will be further discussed in Section 4.1.7.

SD Card Bootstrapping

The SD Card is plugged into the board during development. When it is plugged into the board, it is accessible only by the Zynq UltraScale+ MPSoC. If the SD Card is unplugged from the

board, the **BOOT.bin** can be manually copied into it with the help of a regular PC. If it stays plugged into the board, it is accessible only by the MPSoC itself.

In order to make this automated, the Zynq UltraScale+ MPSoC itself needs to bootstrap the SD Card. The **BOOT.bin** also needs to be stored somewhere temporarily, until it is fetched and stored in the SD Card. The storage problem can be solved quite easily by storing the **BOOT.bin** on the TFTP server along with the other images since it can be fetched in the same way that the other images are fetched.

About the fetching of the **BOOT.bin**, it can be done during the FSBL, during U-Boot, or during the normal Linux operation. This means that a dedicated PetaLinux project will exist with a dedicated U-Boot. The board will be booted through JTAG and will bootstrap the SD Card.

The Linux stage can introduce many problems since it requires a full boot. During the boot process, many failures can occur which would make the framework quite unreliable. On the other hand, the FSBL has a very specific objective which is mostly about initializing the essential hardware of the MPSoC and loading U-Boot while it provides much less flexibility about network interfacing than U-Boot. U-Boot is already used to fetch the Kernel Image and the Device-Tree as a step of the boot process. It requires fewer steps in the boot process than Linux, it provides more network interfaces than FSBL and it also provides tools to manipulate the SD Card. The boot overview that bootstraps the SD Card can be seen in Figure 4.7 [53,80].

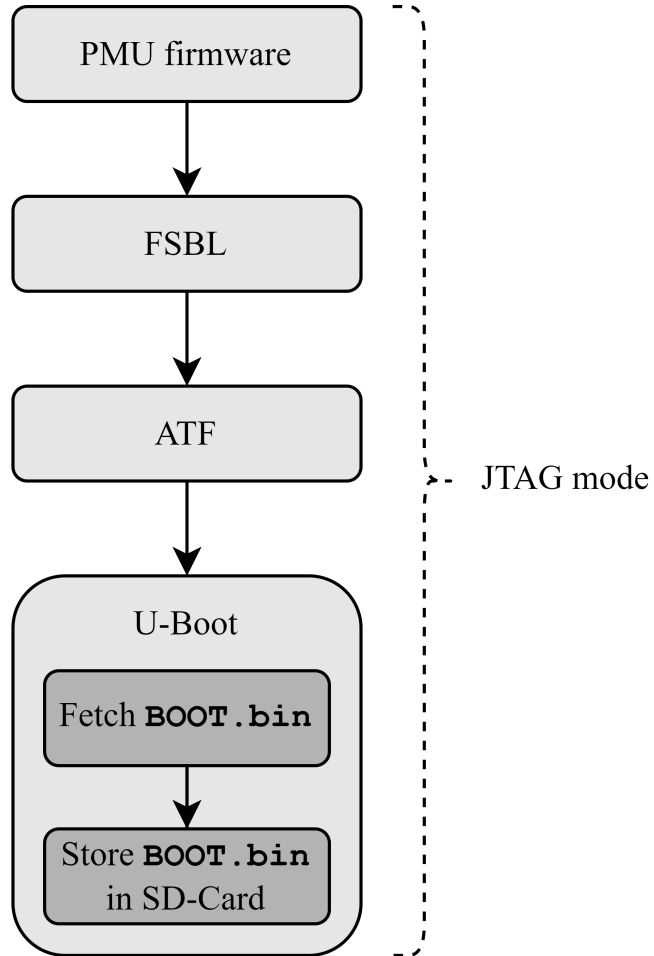


Figure 4.7: Bootstrapping of the SD Card with dedicated U-Boot. Boot with JTAG.

This leads to a system that uses a dedicated U-Boot image (which is different than the actual U-Boot that is used in the boot process) that its purpose is to fetch the `BOOT.bin` and store it in the SD Card. This dedicated U-Boot image along with the PMU firmware, ATF, and FSBL are downloaded in the MPSoC via a JTAG cable. This results in booting the board through JTAG with dedicated images to bootstrap the SD Card before the actual SD Card boot process starts. This is further analyzed in Section 4.1.7.

4.1.7 Booting

In order to boot the Zynq UltraScale+ MPSoC and bring it to a state that it can be tested, the board needs to be power-cycled and bootstrapped. In this way, the Zynq UltraScale+ MPSoC returns to its default initial state before it starts booting. Even though the IPMC is responsible for power-cycling the RTM and managing the boot mode of the Zynq UltraScale+ MPSoC, it was not available during the largest part of the development of the project.

Boot mode

As far as the boot mode is concerned, instead of the IPMC, a substitute solution is necessary for the development of the project. This can be done either with an external device or with the internal registers of the Zynq UltraScale+ MPSoC itself.

An external device would be directly connected to hardware boot mode pins on the board and change their state from JTAG to SD Card. After this, a power cycle would boot the board from the SD Card and the new executable files would start to run on the MPSoC.

The other approach would require access to the Zynq UltraScale+ MPSoC registers. The internal registers in the CRL_APB module allow the change of the boot mode through the software while also resetting the board. Since U-Boot is already used to bootstrap the SD Card and it also has access to the Zynq UltraScale+ MPSoC internal registers, it was considered the best solution.

This would result in a two-stage boot process:

1. **JTAG mode:** Boot the board through JTAG, fetch the `BOOT.bin` from the TFTP server, and store it in the SD Card. Change the boot mode of the board and reset it.
2. **SD Card mode:** After the board resets, it boots from the SD Card since the boot mode was changed in the previous step. The boot images are stored in the `BOOT.bin` in the SD Card.

Figure 4.8 explains the different steps in the process [81].

Power-cycle

As far as the power cycling is concerned, instead of the IPMC, an external Power Distribution Unit (PDU) that is accessed through the network was chosen as a solution to power-cycle the board. Since it is accessed through the network, a script can be implemented in order to automate the procedure of accessing and controlling the PDU.

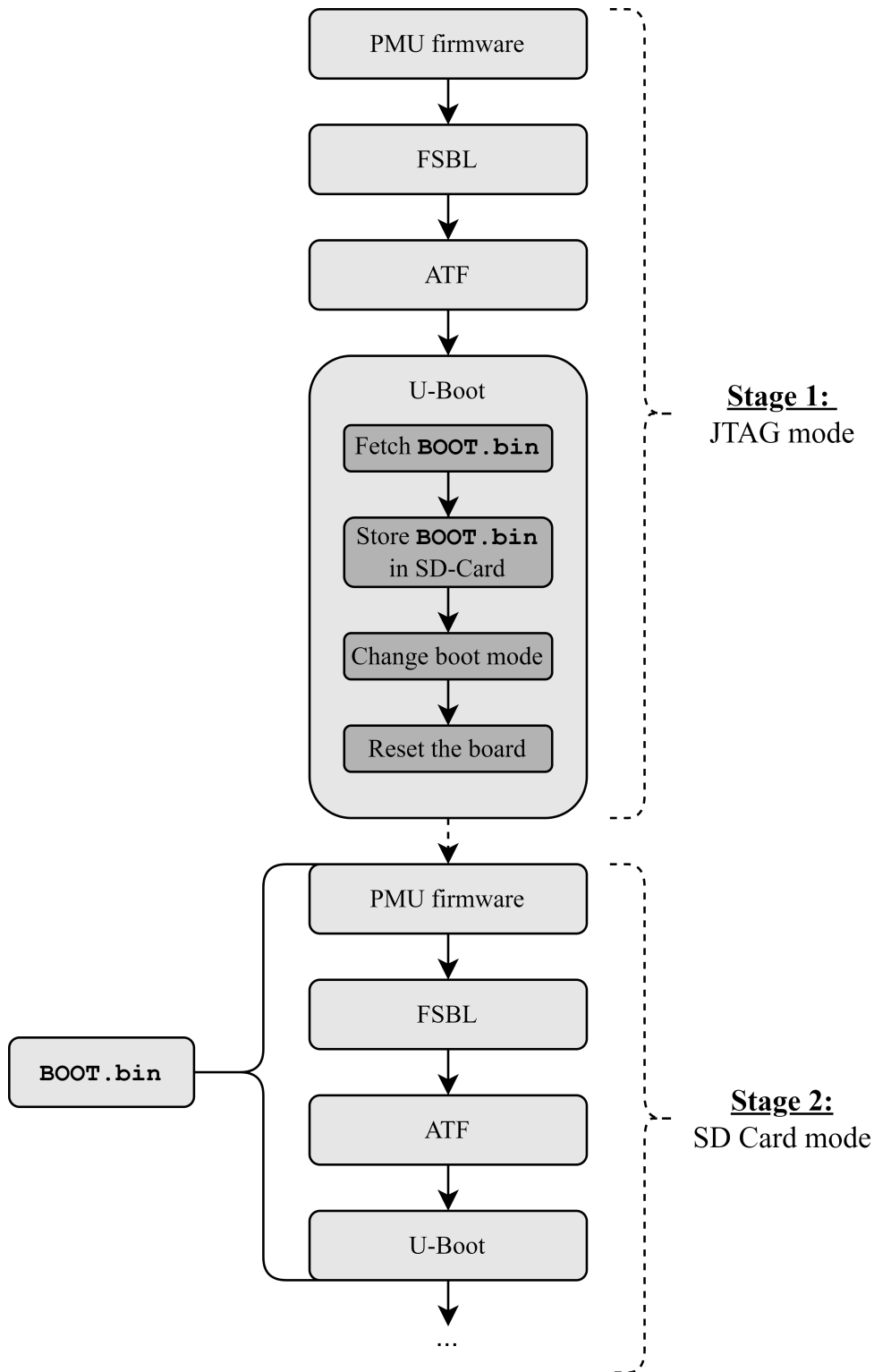


Figure 4.8: Bootstrapping of the SD Card with dedicated images and manipulation of the boot mode.

Verifying

According to Requirement No.12 in Table 2.1, the boot process needs to be monitored. In order to monitor the boot process, there needs to be feedback about the actual stage of the boot process. During booting, the Zynq UltraScale+ MPSoC prints out messages on its standard

output²². In this way, if a serial port of the Zynq UltraScale+ MPSoC is connected to a PC and the standard output is redirected through this serial port, the output can be captured via this serial port on the PC. A process running on the OS of the PC can capture the output of the MPSoC as an input to the PC. Figure 4.9 shows an example of such a system.

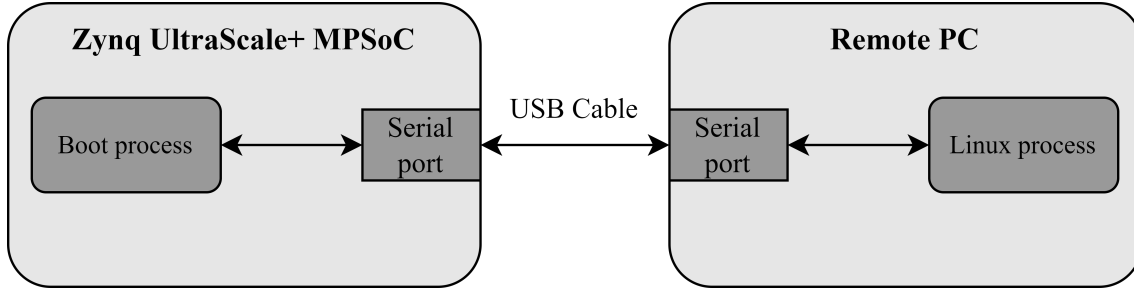


Figure 4.9: Abstract view of interaction with the Zynq UltraScale+ MPSoC boot process.

Since the communication is bidirectional, the process running on the remote PC can not only influence the booting process by interfacing with the CLI during the U-Boot stage as it was described in Section 3.2.3 but also after the Zynq UltraScale+ MPSoC has booted.

4.2 GitLab CI/CD pipelines' notable attributes

The GitLab CI/CD pipelines have a lot of characteristics that can be found useful in the implementation of this project. These characteristics can be configured in the `.gitlab-ci.yml` file. The GitLab CI/CD pipeline consists of various keywords that are categorized into *job keywords* and *global keywords*.

4.2.1 Job Keywords

Each job in a pipeline can be configured individually in its own way. This means that each job can have unique characteristics using these keywords and be tailored made to the application. Some of the most important attributes are described below.

Data sharing

One of the most important attributes of GitLab CI/CD pipelines is the data sharing between the jobs, stages, and pipelines.

In order to pass data between jobs in different stages, the keyword **artifacts** is used. The **artifacts** are a list of files and directories that are created after a successful run of the job. They can be specified either with the name of the file or the directory and they will be saved in a **.zip** archive. In order for a job to fetch the artifacts from other jobs in the same pipeline, the keyword **dependencies** is used. The **dependencies** keyword is basically a list of all the jobs that the specific job should fetch its **artifacts** from. This can be useful for example in case the result of a job is needed in a future job [82].

In order to pass data between consecutive runs of the pipeline, the keyword **cache** is used. The **cache** keyword basically defines the files and directories that need to be cached before the job exits. The next time the pipeline is run, the cache memory will be fetched and the data that

²²Stdout is a standard stream that can transfer output data from a command to the shell

were saved in the previous pipeline. This can be useful for example if a job is downloading a list of libraries or executable files that are needed. If these files are cached, there is no need to download them each time the job runs. Figure 4.10 presents the difference between cache and artifacts [82].

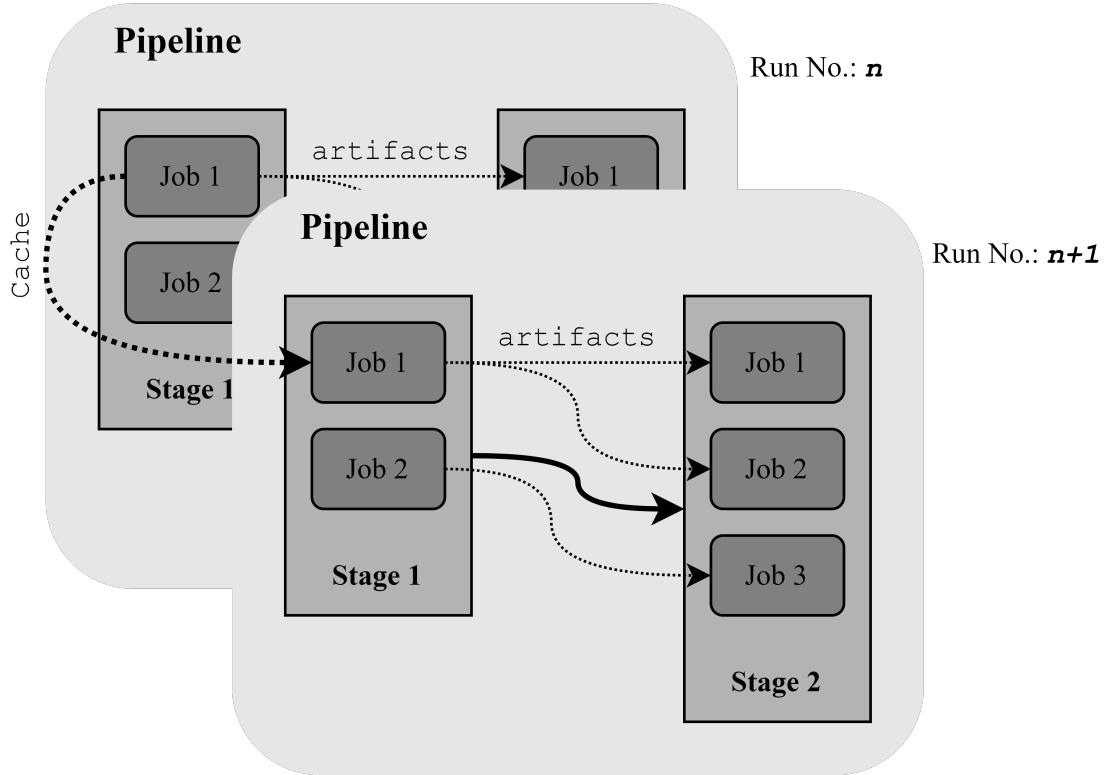


Figure 4.10: Differentiation between cache and artifacts.

In order to pass data between jobs in different pipelines, the keyword **needs** is used. This is further analyzed in Section 4.2.3 [82].

Commands

In order to execute commands or scripts, the GitLab CI/CD pipelines offer three keywords to facilitate their execution: **before_script**, **script**, and **after_script**. The way that they are used is described below:

- **before_script**: List of commands that run before the main script. This can be useful in case installation scripts need to take place or commands who set up a specific environment before the main script starts.
- **script**: List of commands that run as the main script.
- **after_script**: List of commands that run after the main script. This can be useful for example, in case log files from the main script need to be processed before the job exits.

The **trigger** keyword is used in a job to trigger another pipeline. It cannot be used together with the **script** keyword. This is further analyzed in Section 4.2.3 [82].

General

Some general characteristics that can be quite useful are the **tags** and **image** keywords. As it was described previously in Section 3.3.4 the **tags** keyword can be used in order to specify which GitLab Runner will pick up this job. The **image** keyword is used when this specific job runs in a GitLab Runner with a Docker executor in order to specify which Docker image to be used. These two keywords are usually used together [59, 82].

Finally, the **rules** keyword is used in order to include or exclude jobs in a pipeline. It can be quite useful in combination with the **changes** keyword which can detect changes in files or directories. An example of this combination can be a PetaLinux project that does not rebuild all the images every time that it is compiled, but it rebuilds only the images that had changes in their configuration files [82].

A complete list of all the job keywords that are available can be seen in Appendix G.1.

4.2.2 Global Keywords

The global keywords can be used at the beginning of the configuration file in order to define common behavior between the jobs.

The **stages** keyword is used in order to define the different stages of the pipeline along with their order. Figure 4.11 shows an example of how the **stages** keyword would be used in a pipeline with 3 stages named **configure**, **build**, **package** and run with this order [82].

```
1 stages:
2   - configure
3   - build
4   - package
```

Figure 4.11: Example use of **stages** global keyword.

The **default** keyword is used in order to define custom values for job keywords. If a job keyword has to be used in the same way among all the jobs in the pipeline, it is used in combination with the **default** global keyword. A very useful example is the use of the **cache** keyword in case all the jobs need to use the same configuration for their cache memory or the **before_script** in case there is a configuration that needs to be implemented before every job starts [82].

A complete list of all the global keywords that are available can be seen in Appendix G.2.

4.2.3 Multi-project pipelines

There are cases where different pipelines need to communicate, share data, or execute immediately one after the other. GitLab CI/CD pipelines offer this opportunity and provide tools to achieve it.

Data Sharing

In order to pass data between jobs in different pipelines, the keyword **needs** is used. Instead of listing the dependencies, the **needs** keyword is used with the following information, in order to fetch artifacts from a job from another pipeline [82].

- The *name* of the project that is being triggered.
- The *branch* of the project.
- The *job* of the project.
- If it wants to download the artifacts or not.

The following figure shows an example of a pipeline fetching artifacts from another pipeline. This can be useful for example when the PetaLinux project needs the hardware configuration XSA file in order to initiate a project.

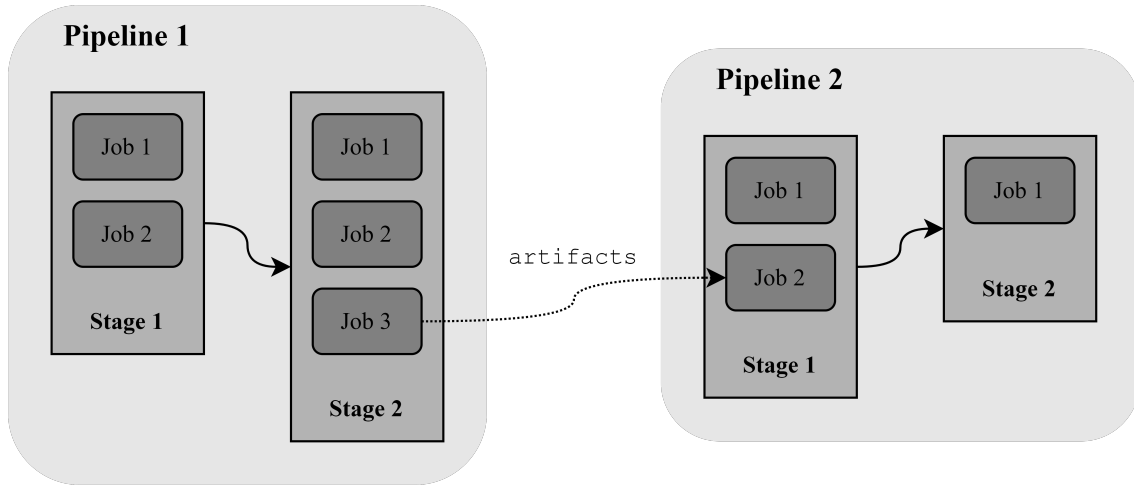


Figure 4.12: Data sharing between different pipelines.

Pipeline triggering

Apart from sharing data between each other, GitLab CI/CD pipelines can trigger each other as well. The **trigger** keyword can be used in this case in a job. It cannot be used in combination with the **script** keyword, meaning that only one of these two can be used in the same job [82].

The information required by the **trigger** keyword in order to execute another pipeline in a project are:

- The *name* of the project that is being triggered.
- The *branch* of the project.
- The execution *strategy* that is going to be used in the following jobs or stages. It, defines if the job after the one that triggered the new pipeline will wait for the pipeline to finish before starting or not.

The following figure shows an example of a pipeline triggering another pipeline. This can be very useful for example, when the Bootstrapping of the board is complete and the booting needs to start immediately.

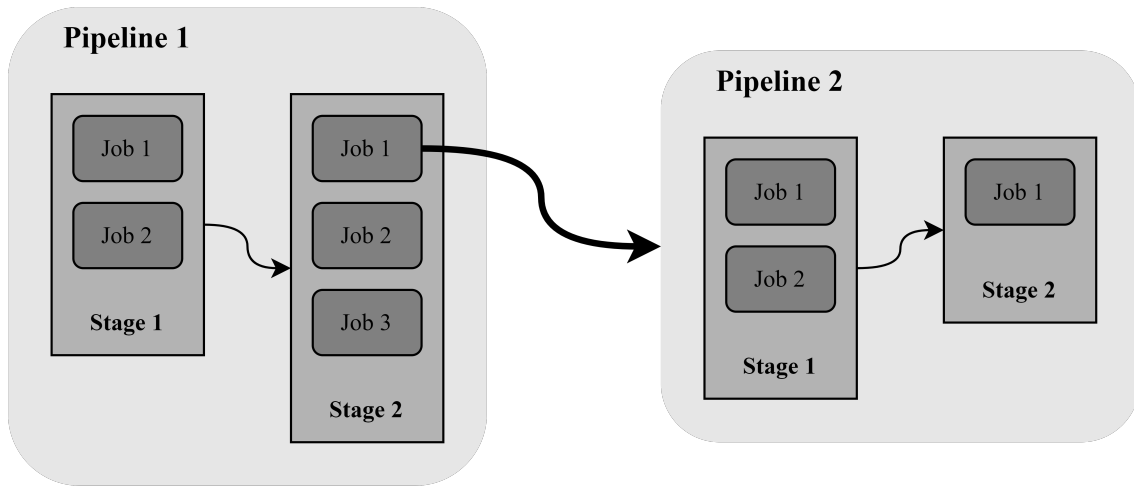


Figure 4.13: An example of a pipeline triggering another pipeline.

5 Top-Level System Design

This chapter explains the top-level system that was created for the automated CI/CD framework. The design is concentrated on how the system is built and how the Zynq UltraScale+ MPSoC boots.

5.1 Top-Level System

The results of the research that were described in Chapter 3 and Chapter 4 are combined in order to propose a Top-Level System Design for the automated CI/CD framework. The following figure shows the top-level system:

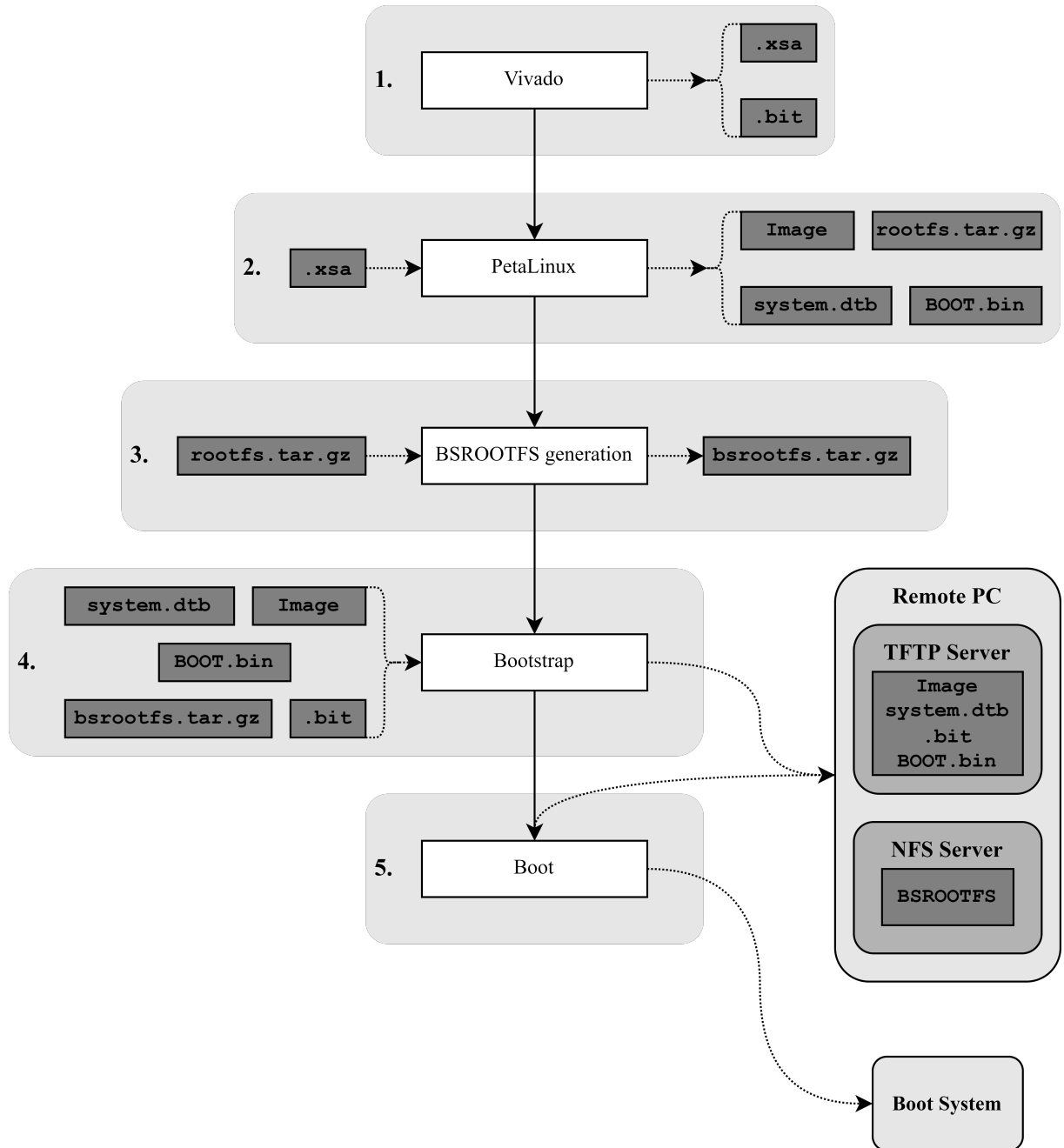


Figure 5.1: Top-Level System Design of the automated CI/CD framework.

1. A pipeline will be responsible for handling the compilation of the Vivado project. The output products will be the XSA file (`.xsa`) and the Bitstream file (`.bit`).
2. A pipeline will be responsible for building the PetaLinux project. It will be able to fetch the `.xsa` file in case a new hardware configuration is available and apply it to the current project. It will be able to build a complete embedded Linux system as it was discussed in Section 4.1.3.
3. The generation of the Board Specific root filesystem (BSROOTFS) requires several steps. In the beginning, a pipeline will generate a generic CentOS 8 ROOTFS without any PetaLinux tools. After the CentOS 8 ROOTFS generation, another pipeline will be responsible for adding the PetaLinux tools in the CentOS 8 ROOTFS in order to generate the BS-ROOTFS.
4. The bootstrapping of the board includes the bootstrapping of the TFTP and NFS servers. A pipeline will be responsible for fetching the images and the BSROOTFS from the previous pipelines and uploading them to the equivalent server.
5. The last step in the automated CI/CD framework is to boot the board and verify that it has booted successfully. The boot process is done through an SD Card, so the SD Card is bootstrapped before the boot process starts. Later, the board resets and the booting process starts while it is monitored in real-time. A pipeline will be responsible to trigger this procedure.

All the different pipelines that are created in order to serve the needs of the automated CI/CD framework will be orchestrated by a Master pipeline which will be responsible for the data sharing between the pipelines and their triggering.

5.2 Boot System

The part of the automated CI/CD framework that is responsible for the booting and the monitoring of the booting of the board is divided into two parts: The setup of the SD Card with the boot process and the power-cycling with the boot monitoring.

5.2.1 SD Card set up and Boot Process

The Zynq UltraScale+ MPSoC will boot from an SD Card and through the Network, so the SD Card needs to be bootstrapped with the `BOOT.bin` file that was created. The MPSoC boots initially through JTAG with the dedicated images and bootstraps the SD Card as it was described in Section 4.1.7. This is done with the help of the Xilinx Software Command-line Tool (XSCT) Console that was described in Section 4.1.3. A Tcl script will load the dedicated images to the MPSoC through JTAG and they will bootstrap the SD Card and reset the board. The board then will boot with the SD Card and the boot process with the newly created `BOOT.bin` will take place.

5.2.2 Power-Cycling and Boot Monitoring

The Zynq UltraScale+ MPSoC needs to power-cycle before booting, in order to reset all the configurations that might have happened in the previous session. A network-controlled PDU that controls the power of the Zynq UltraScale+ MPSoC board will be used. Apart from this, the boot process needs to be monitored in order to verify that the board actually booted. The serial port of the Zynq UltraScale+ MPSoC will be connected to the serial port of a PC. In this way, a service on this PC that will be responsible for reading the serial port will be able to monitor the boot process in real-time.

5.2.3 Integration

The integration of the aforementioned procedures forms the Boot System. A graphical representation of the Boot System is shown in Figure 5.2.

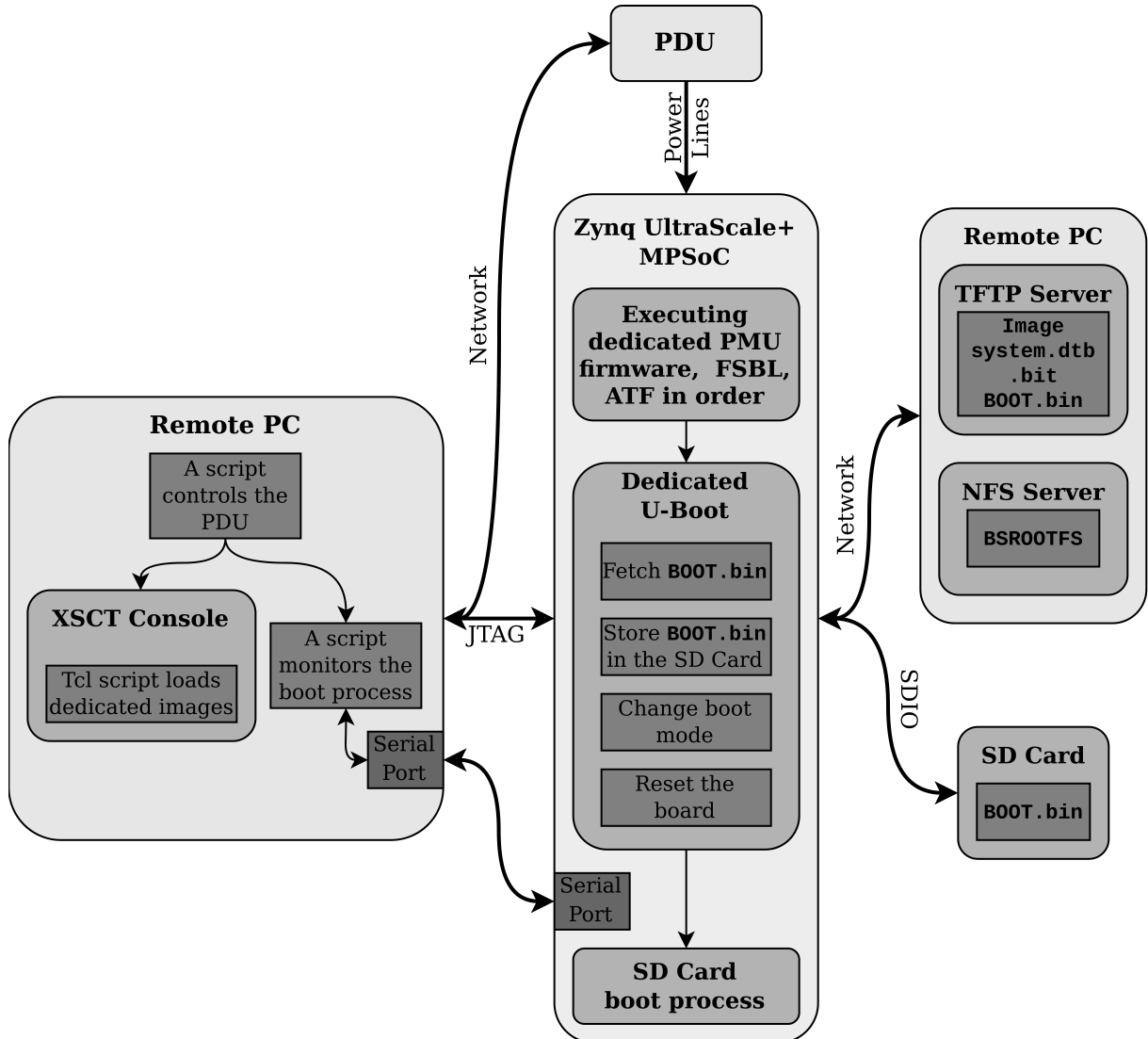


Figure 5.2: Boot System Design.

The system is executed in the following logical order:

1. The script that controls the PDU, commands the PDU to power cycle the board.

2. The Tcl script starts to load and execute the dedicated images through JTAG one-by-one as it was described in Section 4.1.7. The scripts that monitor the boot process runs in the background to check that the booting started. It verifies the boot process by reading the output of the serial port of the Zynq UltraScale+ MPSoC.
3. When *Stage 1* of the boot process is finished, *Stage 2* starts, and the Zynq UltraScale+ MPSoC boots from the SD Card with the `BOOT.bin` that was created in the previous steps of the framework.

6 Implementation

This chapter describes the integration of the solutions proposed in the previous chapters with the GitLab CI/CD pipelines and their implementation.

6.1 Vivado Pipeline

The Vivado Pipeline is responsible for compiling the Vivado Project and generating the bitstream and XSA files. The Vivado project is initiated and the first two steps that were described in Section 4.1.2 should be created already. The *PS Configuration* and *RTL Design* are created by the user. Design Analysis and Simulation are not foreseen to be implemented automatically by this project.

6.1.1 Structure

This project utilizes two GitLab Runners with different executors. One GitLab Runner uses a Shell Executor that is used in order to create a Docker Container that contains the Vivado project with all the necessary tools to compile it. The other GitLab Runner uses this container in order to run the compilation of the Vivado project. The picking up of the jobs is done with respect to their tags [59]. The contents of the GitLab repository are:

Table 6.1: Vivado project repository structure.

#	Name	Type	Description
1	.gitlab-ci.yml	File	Pipeline configuration file
2	Dockerfile	File	Configuration of the Docker Container
3	scripts/	Folder	Job scripts
4	<vivado_project_name>/	Folder	Vivado Project

The Vivado Pipeline is divided into four different stages with each stage composed of one job. Figure 6.1 shows an abstract view of the Vivado Pipeline with the generated artifacts.

The first stage runs in the GitLab Runner with Shell Executor. An already made Docker Container with the Vivado necessary tools is used as a base container. This Docker Container is described in Appendix. This means that the job only needs to copy the Vivado project inside the Docker Container and save it in the GitLab Container Registry²³ where it can be fetched from.

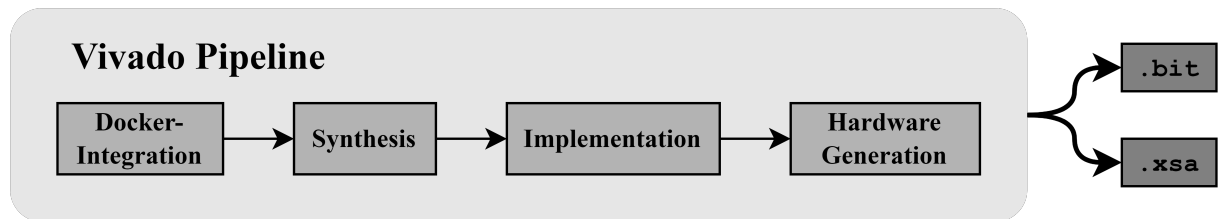


Figure 6.1: An abstract view of the Vivado Pipeline with the four stages and the generated artifacts.

²³The GitLab Container Registry is a secure and private registry for Docker images [83].

The latter three jobs run in the GitLab Runner with a Docker Executor which uses the Docker Container that was created in the first stage. Their responsibility is to run the rest of the three steps described in Section 4.1.2 which are *Synthesis*, *Implementation*, and *Hardware Generation*. Vivado is executed in Tcl mode and the required Tcl commands are used in form of scripts to synthesize, implement and generate the hardware platform of the project.

The jobs that run in Vivado, use the same cache memory so there is no need for passing artifacts between the jobs. The only job that uses artifacts is the last one since it has to store the Bitstream and the XSA file in order to be used in the next pipelines. Figure 6.2 shows a detailed representation of the Vivado Pipeline.

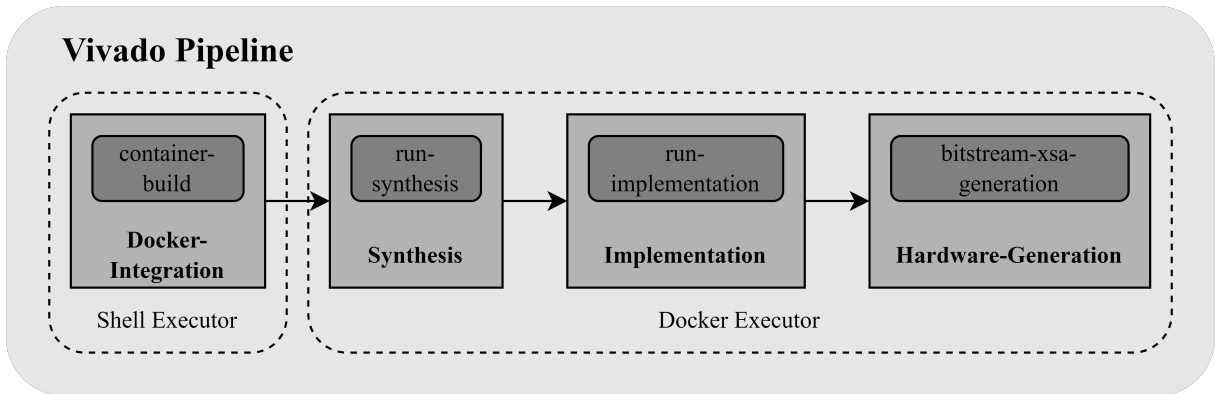


Figure 6.2: Detailed view of the Vivado Pipeline.

6.2 PetaLinux Pipeline

The PetaLinux Pipeline is responsible for compiling the PetaLinux Project and generating the boot images and the PetaLinux ROOTFS that are later used to boot the board and create the BSROOTFS. The PetaLinux Pipeline though is not creating the PetaLinux project, so the user should create it before importing it to the pipeline.

6.2.1 Structure

The PetaLinux project uses the same GitLab Runners as the Vivado project. The first GitLab Runner uses a Shell Executor in order to create a Docker Container that contains the PetaLinux project along with all the necessary tools to compile it while the second one uses this Docker Container with a Docker Executor. An abstract view of the PetaLinux Pipeline is shown in Figure 6.7.

Table 6.2: PetaLinux project repository structure.

#	Name	Type	Description
1	.gitlab-ci.yml	File	Pipeline configuration file
2	Dockerfile	File	Configuration of the Docker Container
3	.dockerignore	File	Files excluded from the Docker Container
4	new-xsa/	Folder	Directory of the new XSA file
5	<petalinux_project_name>/	Folder	PetaLinux Project

The PetaLinux Pipeline is divided into four different stages but in contrast to the Vivado Pipeline, it has multiple jobs per stage. The first stage serves the same role as the one in the Vivado Pipeline. It runs in a GitLab Runner with a Shell Executor and it uses an already made Docker Container with the PetaLinux tools already installed. The job copies the PetaLinux project inside the Docker Container and exits. This Docker Container is described in Appendix.

The latter three stages are responsible for the *Configuration*, *Build* and *Package* stages that were described in Section 4.1.3.

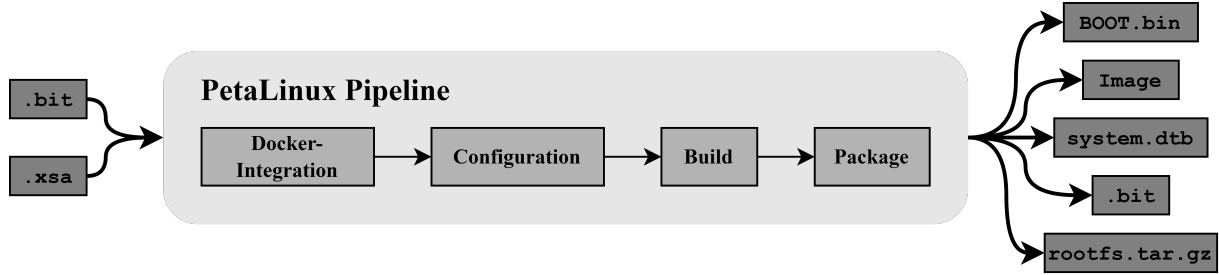


Figure 6.3: An abstract view of the PetaLinux Pipeline with its four stages and the input and output artifacts.

6.2.2 Configuration

The *Configuration* stage is responsible for the configuration of the project. There are three different jobs in this stage and only one of them runs. In order to make the pipeline more flexible and make it able to run without being dependent on the Vivado pipeline, additional jobs were added in the *Configuration* stage.

1. **Silent-Configuration:** This job runs with a manual trigger when there is no change in the hardware configuration (XSA) of the project. For example, if there is a new addition in the PMU firmware, there is no new hardware configuration file. The `--silentconfig` option is used and the pipeline is triggered manually by the user [84].
2. **Hardware-Configuration:** This job runs with a manual trigger when there is a change in the hardware configuration (XSA) of the project in the `new-xsa/` directory. An example is when there is a new XSA file that needs to get imported into the PetaLinux project without including the Vivado Pipeline. The `--get-hw-description=<path-to-xsa-file>` option is used. The `--silentconfig` flag is used in order to configure the project from the files and not from the menuconfig GUI. The pipeline is triggered manually by the user [84, 85].
3. **Hardware-Configuration-API:** This job runs automatically when the pipeline is triggered externally. It fetches the XSA file and the Bitstream from the artifacts of the Vivado Pipeline and uses the same command with the *Hardware-Configuration* job to import the new XSA file.

```

1 $ # Import new .xsa file without the menuconfig
2 $ petalinux-config --get-hw-description=<path-to-xsa> --silentconfig

```

Figure 6.4: Example use of `petalinux-config` command.

All three stages use the same cache memory in order to pick up from the last configuration in the next run of the pipeline regardless of which job runs. The generated configuration is stored as an artifact. The jobs in the build stage use these artifacts to accelerate the build process. These artifacts are stored from the `\$PROJECT_NAME/build` and `\$PROJECT_NAME/project-spec/meta-plnx-generated` directories.

6.2.3 Build

The next stage is responsible for building several components of the PetaLinux project. Each component is built individually using the `petalinux-build -c <component>` command. This command allows building a component individually by specifying it using the `-c` flag option. In order to accelerate the development, each job is triggered only when there is a change in the respective component. For example, if there is a change in the FSBL there is no need to rebuild the ATF.

```
1 $ # Build U-Boot
2 $ petalinux-build -c u-boot
```

Figure 6.5: Example use of `petalinux-build` command.

In order to optimize the speed of the compilation of the Device-Tree, Linux Kernel, and U-Boot, the same cache memory is used. The Linux Kernel and U-Boot use the Device-Tree in their compilation. A common cache memory that allows access to common resources accelerates their development since for example, the Device-Tree does not need to get compiled three times, but only once.

The generated boot images and the PetaLinux ROOTFS are stored in the `\$PROJECT_NAME/images/linux` directory. These artifacts are used in the package stage.

6.2.4 Package

At this stage, all the images created in the *Build* stage are fetched as artifacts. The PMU firmware, FSBL, ATF, and U-Boot are packaged together in order to create the `BOOT.bin` image. They are stored as artifacts to be available for the other pipelines. Figure 6.7 shows a detailed description of the PetaLinux Pipeline.

```
1 $ # Package boot images
2 $ petalinux-package --boot --format BIN --fsbl zynqmp_fsbl.elf --u-
   boot u-boot.elf --pmufw pmufw.elf --force
```

Figure 6.6: Example use of `petalinux-package` command.

A visual representation of the detailed description of the PetaLinux Pipeline is shown in Figure 6.7.

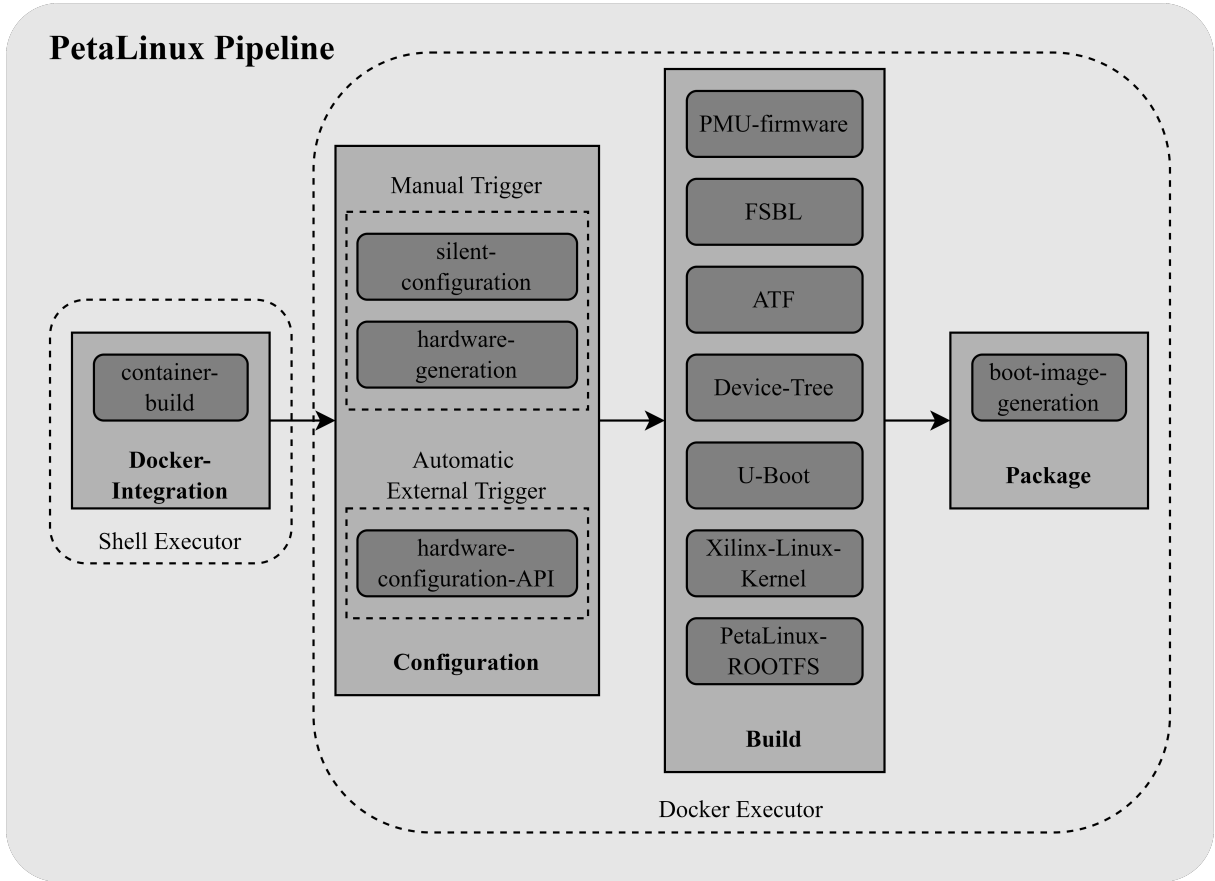


Figure 6.7: Detailed view of the PetaLinux Pipeline.

6.3 Dedicated Boot Images Pipeline

The Dedicated Boot Images Pipeline is responsible for creating the dedicated images that boot the Zynq UltraScale+ MPSoC in the first stage of the boot process as it was described in Section 4.1.7. These images boot the Zynq UltraScale+ MPSoC through JTAG, set up the SD Card, and generate a reset. The Pipeline is a simplified version of the PetaLinux Pipeline.

6.3.1 Structure

The first stage which includes the *Docker Integration* remains the same. The same Docker Container with the PetaLinux Pipeline is used since this pipeline compiles a PetaLinux project as well. The other stages have some significant changes which are described below:

- The *Configuration* stage contains only the **Hardware-Configuration-API** job. This ensures that the dedicated images will be built with the latest available hardware configuration without mixing different versions.
- The *Build* stage builds only the essential images which are the PMU firmware, FSBL, ATF, and U-Boot. These are the only images that are used in the first stage of the boot process so there is no need to build the rest of them.
- The *Package* stage does not package the file in a single `BOOT.bin` since the files are loaded

individually through JTAG. They are stored as artifacts and are ready to be fetched from the other pipelines.

The dedicated U-Boot is responsible for fetching the `BOOT.bin` file from the TFTP server, storing it in the SD Card, and resetting the Zynq UltraScale+ MPSoC. When the `BOOT.bin` is fetched from the TFTP server, its name contains also the parent directory, so before storing it in the SD Card, it needs to be renamed to `BOOT.bin`. Sometimes there will be no new `BOOT.bin` file in the application, but only a new test script. In order to avoid writing to the SD Card when the `BOOT.bin` is the same, the `BOOT.bin` of the SD Card is fetched and their CRC²⁴ values are checked. A flowchart of the dedicated U-Boot is shown in Figure 6.8.

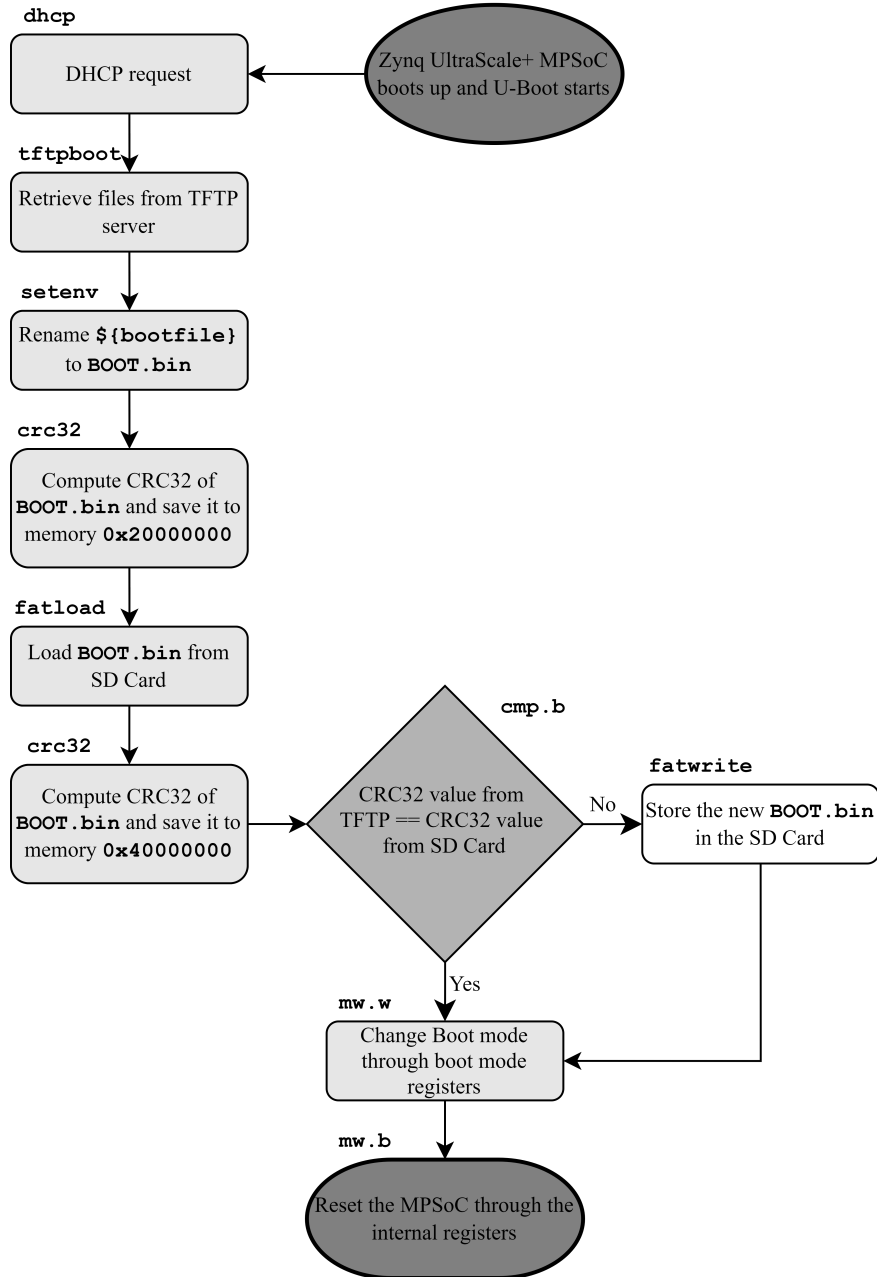


Figure 6.8: Flowchart of the dedicated U-Boot.

²⁴CRC is an error-detecting code added to data in order to detect changes to digital data during transmission, storage or retrieval.

6.4 Root Filesystem Pipelines

The Root Filesystem Pipelines are responsible for generating the Board Specific root filesystem (BSROOTFS). This is done with a combination of 2 separate pipelines. The first pipeline is responsible for creating a bare CentOS8 ROOTFS while the second one will be responsible for adding all the necessary PetaLinux tools.

6.4.1 CentOS 8 Filesystem Pipeline

The CentOS 8 Filesystem Pipeline is responsible to create a bare CentOS 8 Filesystem. It is divided into two different stages with one job per stage. The first stage serves the same role as the first stages in the previous pipelines. It runs in a GitLab Runner with a Shell Executor in order to create a Docker Container. This Docker Container has all the necessary tools to build the CentOS 8 Filesystem.

The second stage is responsible for actually generating the CentOS 8 Filesystem inside the Docker Container. This happens by using a GitLab Runner with a Docker Executor that uses the Docker Container created in the previous stage. A Python script named `mkrootfs.py` uses the DNF²⁵ Package Manager²⁶ in combination with QEMU to create a CentOS 8 Filesystem with `aarch64` architecture.

Table 6.3: CentOS 8 Filesystem project repository structure.

#	Name	Type	Description
1	<code>.gitlab-ci.yml</code>	File	Pipeline configuration file
2	<code>Dockerfile</code>	File	Configuration of the Docker Container
4	<code>configs/</code>	Folder	Directory of extra config files for the ROOTFS
5	<code>mkrootfs.py</code>	File	Script which generates the ROOTFS

The DNF Package Manager uses groups of packages that can be installed altogether. This can be specified by the option `group` along with the name of the specified group. The option `Minimal Install` includes the minimum amount of packages in order to have a basic CentOS 8 distribution [88].

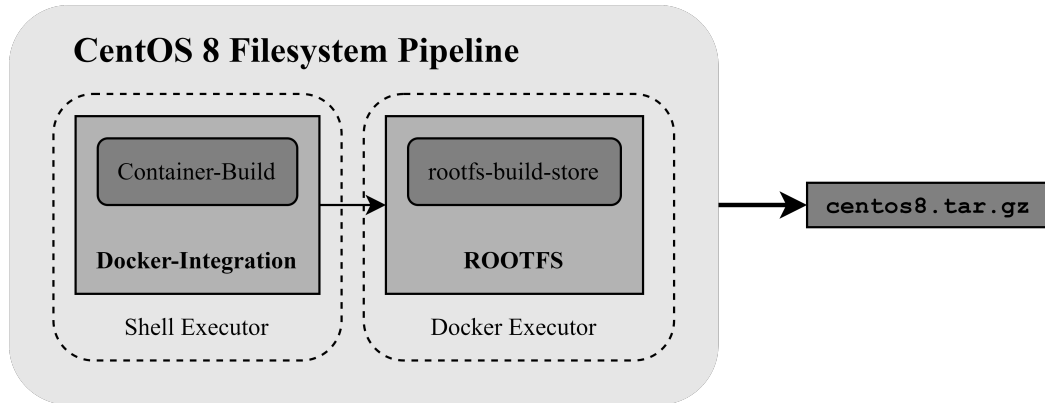


Figure 6.9: Detailed view of the CentOS 8 Filesystem Pipeline.

²⁵DNF is a Package Manager for rpm-based Linux distributions, such as CentOS 8, Fedora, and RedHat [86].

²⁶Package Manager is a software toolbox that automates the installation, configuring, updating, and removing of software packages in a filesystem [87].

The DNF Package Manager uses an option to force a specific architecture for the packages that are downloaded. The `--forcearch` option is used to manage packages of a specific architecture. If the specific architecture is configured inside QEMU, when the DNF command runs with this option, QEMU will pick it up and run it with emulation [86].

6.4.2 Board Specific Filesystem Pipeline

The Board Specific Filesystem Pipeline is responsible to create the BSROOTFS for the Zynq UltraScale+ MPSoC. It is divided into two stages. The first stage has two jobs which are responsible to fetch the PetaLinux ROOTFS from the PetaLinux Pipeline and the CentOS 8 ROOTFS from the CentOS 8 Filesystem Pipeline. The second stage has one job and is responsible to create the BSROOTFS. The jobs do not need to run inside any specific Docker or Shell Executor. To keep the project as independent as possible from the host PC a Docker Executor is used.

The PetaLinux ROOTFS contains Kernel modules that are necessary by the kernel to operate on the Zynq UltraScale+ MPSoC. They are located in the `/lib/modules/<kernel-version>-xilinx-v<vivado-version>/` folder. Since these files are created by PetaLinux, their architecture is `aarch64` so they can be copied inside the CentOS 8 ROOTFS in the same directory, to create the BSROOTFS.

```
1 $ # Copy kernel modules in the CentOS 8 Filesystem
2 $ cp -R petalinux-rootfs/lib/modules/4.19.0-xilinx-v2019.2/ bsrootfs
   /lib/modules/
```

Figure 6.10: Example of copying the PetaLinux kernel modules in the CentOS 8 Filesystem.

The jobs in the first stage use the job keyword `needs` as it was described in Section 4.2.3. The two jobs store the PetaLinux and CentOS 8 ROOTFS as artifacts in order to be fetched from the job in the next stage. A complete implementation of the Board Specific Filesystem Pipeline is shown in Figure 6.11.

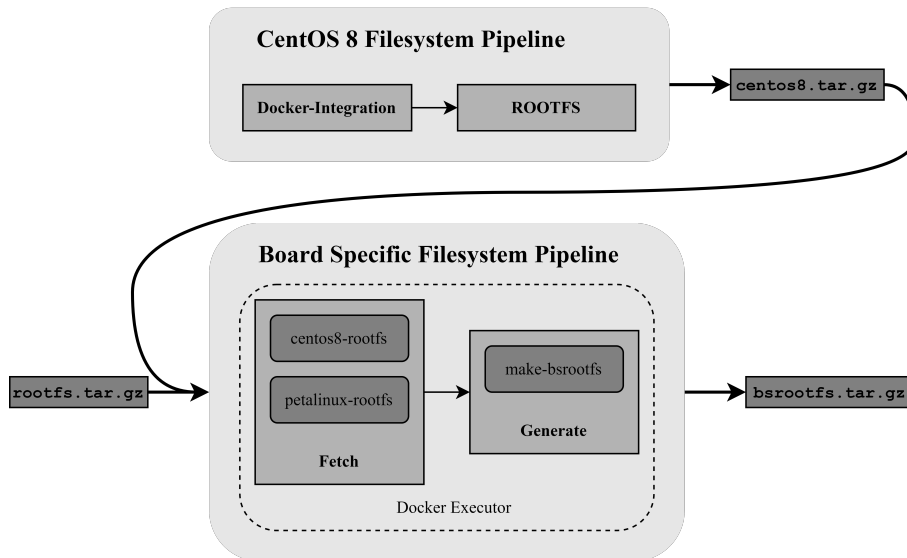


Figure 6.11: Detailed view of the Board Specific Filesystem Pipeline and its cooperation with the PetaLinux and CentOS 8 Filesystem Pipelines.

The repository structure of the Board Specific Filesystem pipeline contains the `.gitlab-ci.yml` for the configuration of the pipeline and a `specific-files/` folder which contains all the extra files for the BSROOTFS. There is no need for a Dockerfile because a shared GitLab Runner from the CERN network is used. This is possible since the pipeline does not require any specific tools and the shared available runners are enough.

6.5 Bootstrap Pipeline

The Bootstrap Pipeline is responsible for bootstrapping the board. It is responsible to store the boot images and the BSROOTFS to their final destinations. Then, the Zynq UltraScale+ MPSoC will fetch them and boot with them.

6.5.1 Structure

The Bootstrap Pipeline is divided into 2 different stages. The first stage consists of two different jobs which are responsible for fetching the boot images and the BSROOTFS respectively. The second stage consists of two different jobs that are responsible to store the boot images in the TFTP server and the BSROOTFS in the NFS server respectively. Even though the pipeline can run with either a Shell or a Docker Executor, the Docker Executor will provide more independence from the host PC.

In order to access the TFTP and NFS servers, the `rsync` command is used. The `rsync` command is a versatile copying tool that is used in order to copy or synchronize files to remote locations. It provides a very fast execution since it transfers only the difference between the two files, so the amount of data that is transferred through the Network is minimal. In order to transfer the files, it needs the username and hostname of the remote location and also the paths of the source and destination directories [89].

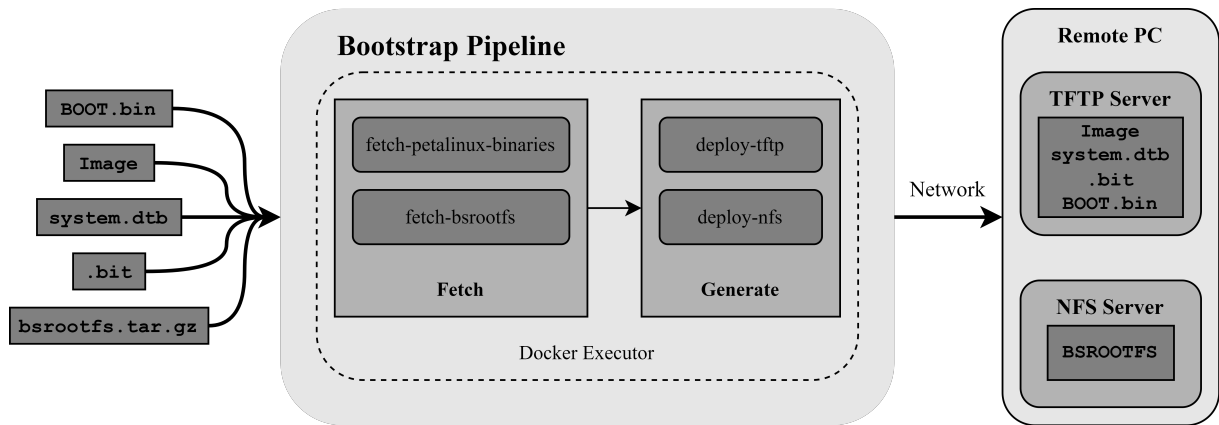


Figure 6.12: Detailed view of the Bootstrap Pipeline.

6.6 Boot Pipeline

The Boot Pipeline was thoroughly described in Section 5.2. It is responsible to bootstrap the SD Card, power-cycle the Zynq UltraScale+ MPSoC, boot it, and monitor its boot process.

This is done by scripts that handle these requirements. The scripts that were implemented are the following:

Table 6.4: Scripts for the Boot Pipeline

#	Name	Description
1	pdu.py	Communicate with the PDU and power-cycle the board
2	jtag.boot.tcl	Download the dedicated images through JTAG to boot the board
3	serial_line.py	Read the Serial Console to monitor the boot process
4	runme.sh	Manage the execution of the other scripts

The Boot Pipeline consists of one stage with one job. This job uses the `rsync` command to synchronize the dedicated images and the scripts to the PC that the Zynq UltraScale+ MPSoC is connected to. The Zynq UltraScale+ MPSoC is connected to the PC through JTAG. When the files are synchronized, it executes the `runme.sh` script using the `ssh`²⁷ command.

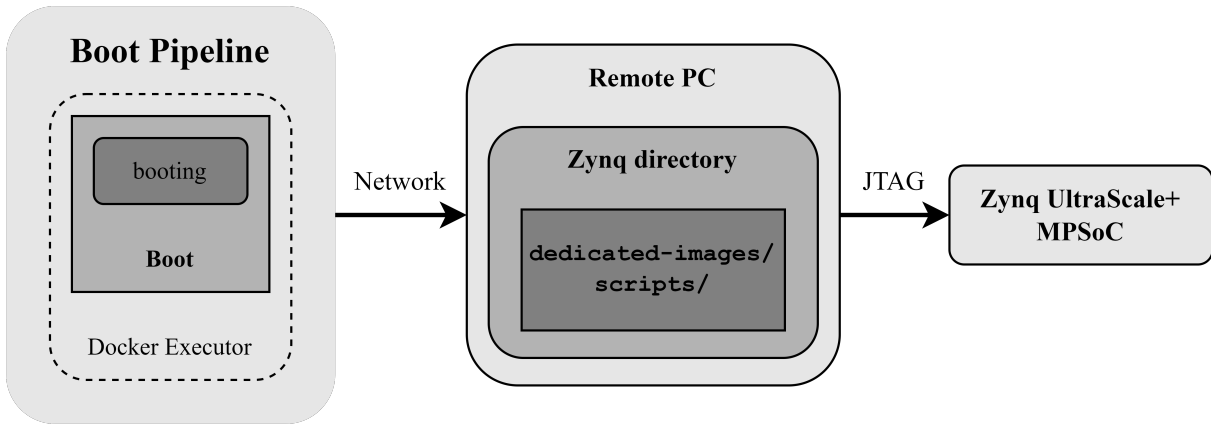


Figure 6.13: Detailed view of the Boot Pipeline.

6.6.1 Power-Cycling script

This is a python script that communicates with the PDU to power-cycle the Zynq UltraScale+ MPSoC. It connects to the PDU through SSH and interacts with it using the `pexpect`²⁸ package of python.

```

1 [user@host ~]$ ssh pdulab40@pdu-lab40-r02-10
2 pdulab40@pdu-lab40-r02-10's password:
3 Welcome to PX3 CLI!
4 Last login: 2022-06-12 14:30:49 EDT
5 [My PDU] > power outlets 1 cycle
6 Do you wish to cycle outlet 1? [y/n] y
7 [My PDU] >

```

Figure 6.14: PDU's CLI interaction.

As it can be seen in Figure 6.14, the interaction with the PDU happens in lines No. 1, 2, 5, 6. This means that expected patterns in these lines, such as "]" in line No. 1, "`password:`"

²⁷SSH is a network protocol that provides a secure way to access a computer over an unsecured network [90].

²⁸Pexpect is a tool that allows spawning and control of child applications and responds to expected patterns [91].

in line No.2, " >" in line No. 5, and "n]" in line No. can be used to send the appropriate patterns. A file holding the password for the PDU is kept in the PC that is connected to the Zynq UltraScale+ MPSoC since the scripts are running there, in order to not be visible in the repository.

6.6.2 Boot and Monitoring scripts

The boot script is a Tcl script that uses Vivado-compatible Tcl commands to boot the board. The dedicated images (PMU firmware, FSBL, ATF, and U-Boot) are downloaded, one by one, through JTAG to the Zynq UltraScale+ MPSoC. More information can be found in Appendix.

The boot monitoring script is a python script that utilizes the `pexpect` package of python to interact with the board. By using predefined strings that are printed by Zynq UltraScale+ MPSoC during booting, the script can verify the state of the boot process. If the boot process is stuck at some point, nothing will be printed in its output so the timeout will terminate the script and the pipeline will fail.

The first time interaction with the Zynq UltraScale+ MPSoC happens in the U-Boot stage. Before executing its commands, U-Boot allows using the CLI. This can be triggered by pressing any key when U-Boot prompts so. At this point, `pexpect` can be used to send commands to U-Boot. Later in the boot process, when the Linux OS is running, the Zynq UltraScale+ MPSoC can be also accessed through SSH.

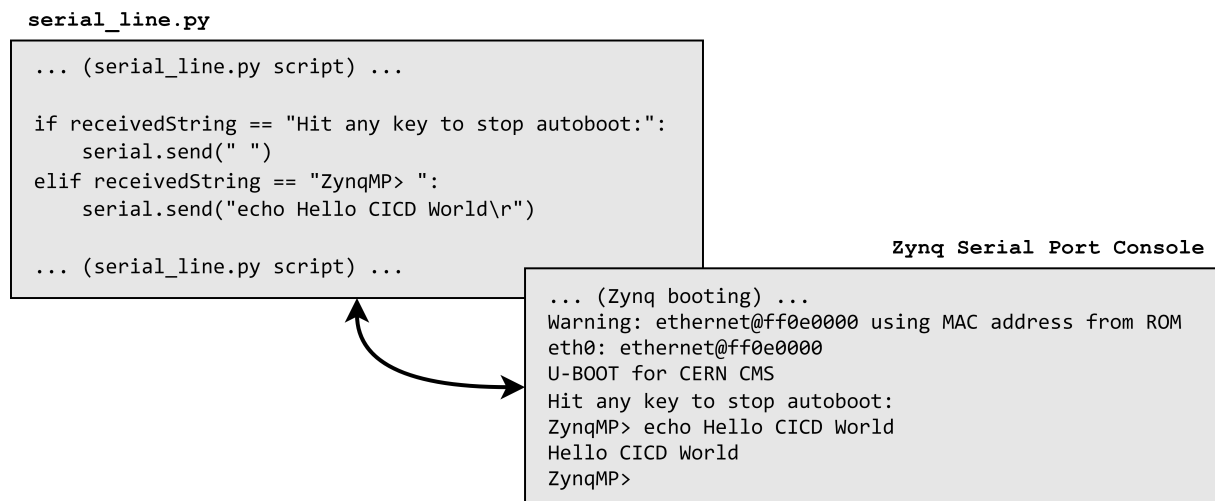


Figure 6.15: Example of interacting with U-Boot through the `serial_line.py` script during booting.

Figure 6.16 shows an example of interacting with U-Boot. When the specified pattern of a string is caught by the `serial_line.py` script, it responds. In this example, the U-Boot boot process is interrupted and U-Boot will drop to its CLI where commands can be executed by catching the `ZynqMP>` pattern.

6.6.3 Integration

The integration of the above scripts happens in the `runme.sh` script. The PDU script needs to run first to power-cycle the board. The boot and monitoring scripts need to run in parallel since

the dedicated images are loaded and executed immediately. The boot script will exit when the last dedicated image (U-Boot) of the first stage in the boot process is loaded. The monitoring script will continue until the Zynq UltraScale+ MPSoC is fully booted.

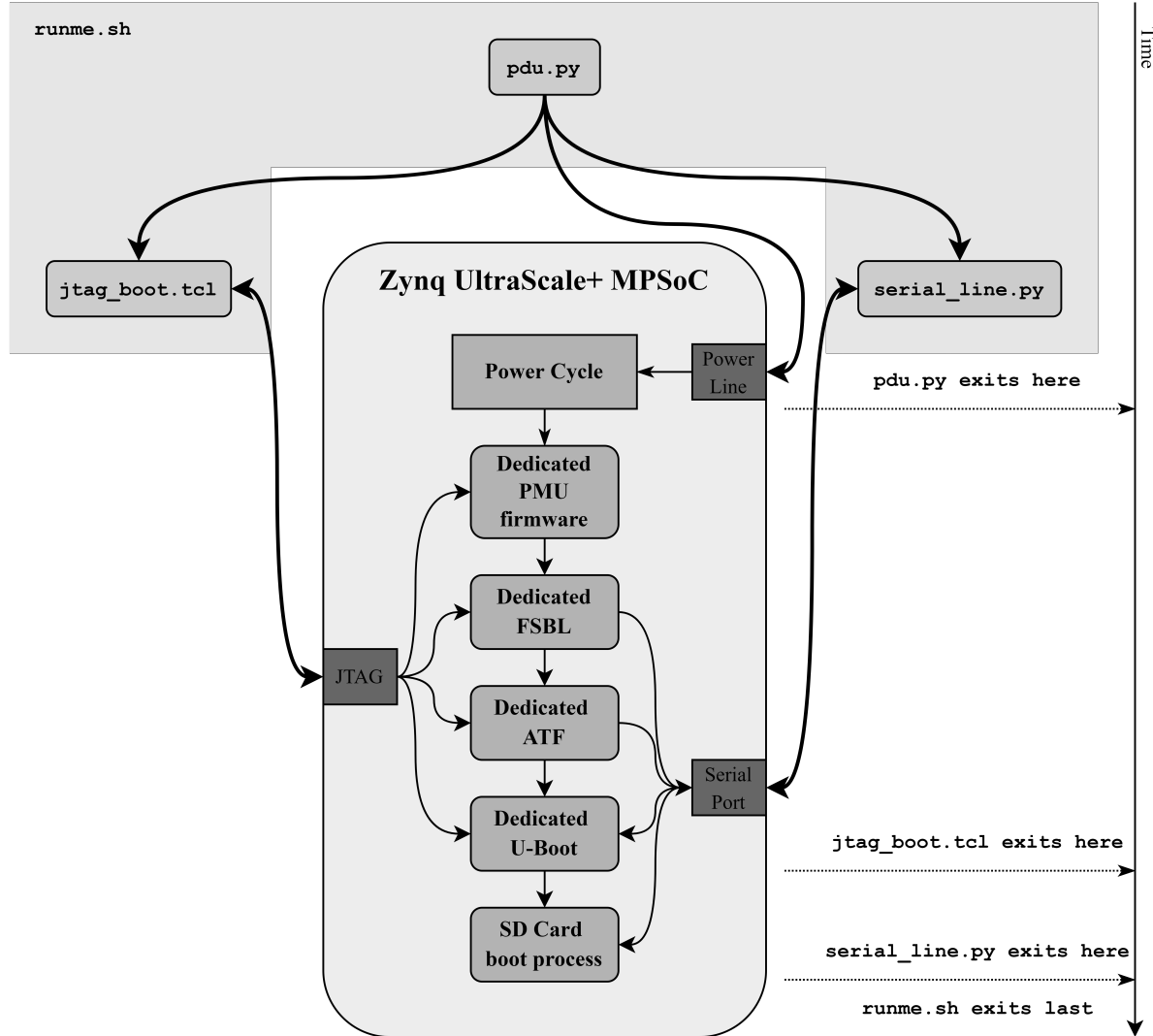


Figure 6.16: Management of the scripts during booting.

6.7 Maestro Pipeline

The Maestro Pipeline is the Orchestrator of all the previous pipelines. All these pipelines need to run in a predefined order without any timing implications. The Maestro pipeline does not run any script itself. Instead of scripts, the Maestro pipeline triggers other pipelines using the **trigger** keyword as was described in Section 4.2.3.

Each stage triggers one pipeline except for the PetaLinux stage which triggers both the PetaLinux and Dedicated Boot Images Pipelines. The jobs are configured such that the next stage cannot be started if the pipeline which was triggered in the previous stage, is not finished.

The Maestro Pipeline is also responsible to pass the environmental variables of the pipeline. These variables contain project-specific information such as project name, project id, etc... A more thorough explanation of the use of these environmental variables is given in Section 7.

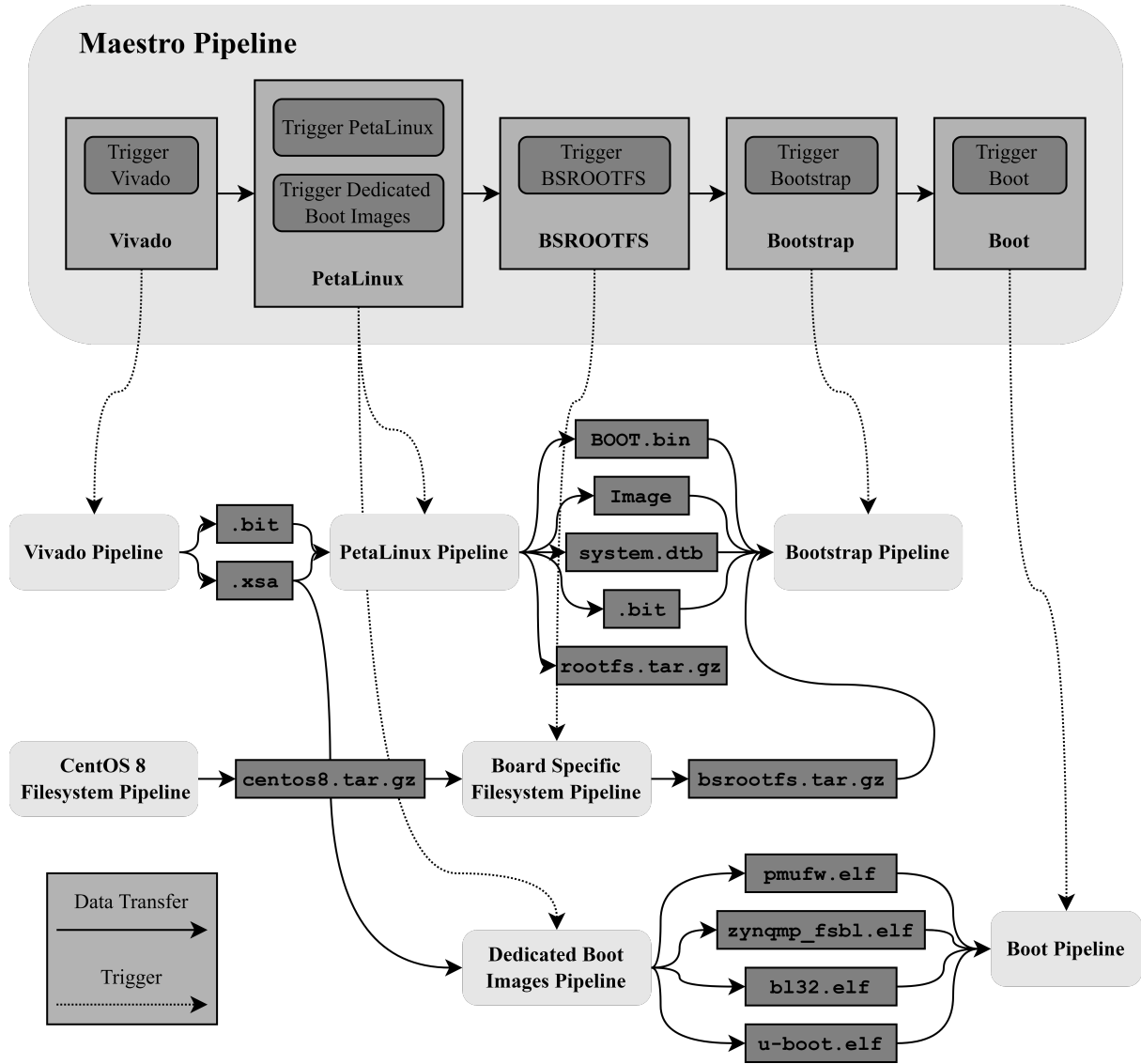


Figure 6.17: The Maestro Pipeline.

7 Testing the infrastructure

The automated CI/CD framework is tested by applying a working project to its infrastructure. The modifications to the automated CI/CD framework are categorized by the pipelines themselves. The automated CI/CD framework should be able to build and deploy the firmware and the software of the board while also being able to boot and monitor its boot process.

Table 7.1: Project modifications.

#	Pipeline	Implementation
1	Vivado Pipeline	The Vivado Project needs to be copied in the place of the <code>vivado_project/</code> folder.
2	PetaLinux Pipeline	The PetaLinux Project needs to be copied in the place of the <code>petalinux_project/</code> folder.
3	Dedicated Boot Images Pipeline	-
4	CentOS 8 Filesystem Pipeline	-
5	Board Specific Filesystem Pipeline	Extra configuration files are added in the <code>files/</code> folder.
6	Bootstrap Pipeline	-
7	Boot Pipeline	-
8	Maestro Pipeline	The configuration variables need to change.

The CentOS 8 Filesystem Pipeline does not require any specific implementation since its objective is to build the CentOS 8 ROOTFS with the DAQ tools included.

The Bootstrap Pipeline is responsible to deploy the boot images to the TFTP server and the BSROOTFS to the NFS server. Since the TFTP and NFS servers do not change for every new project, the pipeline does not need any extra modifications.

The Boot Pipeline is responsible to Boot the board and monitoring its boot process. The boot process is standardized as the Zynq UltraScale+ MPSoC needs to boot from an SD Card and through the Network. No further modifications are required.

The Maestro Pipeline does not need to undergo any major changes. As the orchestrator of the other pipelines, the changes that are needed include the update of the variables that hold the project names and the jobs that provide artifacts to other pipelines.

7.1 The AXI Chip2Chip Project

A fundamental project that checks one of the most important utilities of the Zynq UltraScale+ MPSoC is the AXI Chip2Chip Project. The Advanced eXtensible Interface (AXI) is an on-chip communication bus protocol and it can be used in order to communicate between FPGAs. Since it is the native bus on ARM processors for SoC devices, it can be used for communication between the Processing System (PS) and the Programmable Logic (PL) [92].

The Zynq UltraScale+ MPSoC will run control and monitoring software on the Linux OS,

for the FPGA chips that are placed on the DTH board. This is thoroughly described in Section 1.3.2. The AXI Chip2Chip will be used for communication with the FPGAs. A project has been developed to test this functionality.

The concept of the project is that an AXI Chip2Chip communication will be established between the FPGA part of the Zynq UltraScale+ MPSoC and another FPGA. The Zynq UltraScale+ MPSoC FPGA will receive data from the other FPGA and it will store the data in a memory location that is accessible by the PS. This register will be accessed by the userspace of the Linux OS.

7.2 Vivado Pipeline Modifications

The Vivado project needs to get integrated into the pipeline. The Vivado project gets under version control and the new structure of the pipeline are:

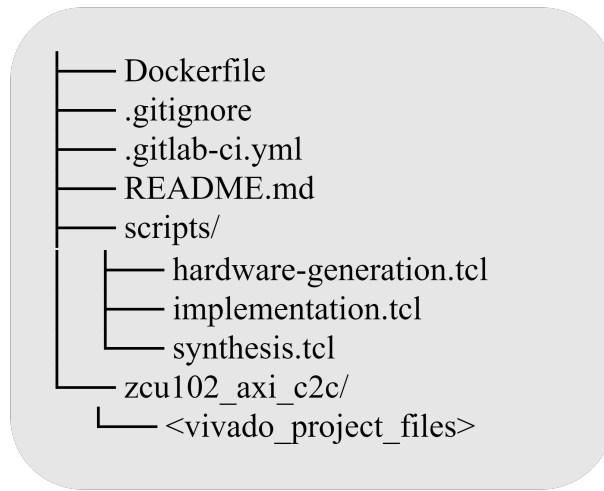


Figure 7.1: AXI Chip2Chip Vivado project GitLab structure.

The global variables that hold the project name and the root folder name need to be updated with the new values. When the pipeline is triggered from the Maestro Pipeline, it can fetch those values from the Maestro Pipeline.

```

1 variables:
2     FOLDER_NAME: zcu102_axi_c2c
3     PROJECT_NAME: zcu102_ipi

```

Figure 7.2: Variables for Vivado Project.

Since the PS configuration and the PS and the RTL analysis are completed already (see Section 6.1), the project is ready to move to the Synthesis stage.

The memory location that the AXI Chip2Chip IP has been mapped in the 0x00.B010.0000 address. The data that the FPGA of the Zynq UltraScale+ MPSoC is going to receive is going to be stored in the memory location with an offset of 0x10 at 0x00.B010.0010. The offset at 0x8 at 0x00.B010.0008 is a read only register and will be used in the final project to read the status registers of the IP.

7.3 PetaLinux Pipeline Modifications

The PetaLinux project needs to get integrated into the pipeline as well. The PetaLinux project gets under version control and the new structure of the pipeline is shown in Figure 7.3. The PetaLinux tools already provide a `.gitignore` file that lets the unnecessary files out of version control.

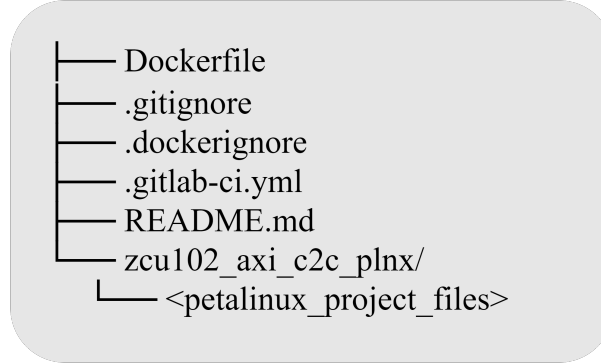


Figure 7.3: AXI Chip2Chip PetaLinux project GitLab structure.

The global variables that hold the Vivado project information need to be updated with the new values as well. This information is needed in order to be able to fetch the artifacts (XSA file and Bitstream) from the Vivado Pipeline. These variables can be fetched from the Maestro Pipeline as well. In case the PetaLinux project needs to be compiled alone, the variables need to be set also in the `.gitlab-ci.yml` of the PetaLinux project.

```

1 variables:
2   PETALINUX_PROJECT_NAME: petalinux_axi_c2c
3   VIVADO_PROJECT_GITLAB_PATH: <project_path_on_gitlab/>
4   VIVADO_PROJECT_NAME: zcu102_axi_c2c
5   VIVADO_PROJECT_GITLAB_ID: <vivado_project_gitlab_id>
6   VIVADO_PROJECT_GIT_BRANCH: master
7   VIVADO_PROJECT_JOB: bitstream-xsa-generation

```

Figure 7.4: Variables for PetaLinux project.

For this specific project, a *crashkernel* was also needed. A *crashkernel* is a feature of the CentOS 8 distribution that allows the booting of a second kernel on top of the kernel that has crashed [93].

There is a long procedure on how to enable the crashkernel for the Zynq UltraScale+ MP-SoC which requires modifications in the `/project-spec/meta-user/recipes-kernel/linux/linux-xlnx/devtool-fragment.cfg` in the boot arguments of the kernel and the configuration file of the crashkernel's `kdump` service [53].

The modifications are included in the PetaLinux project so the pipeline configuration is not affected.

7.4 Board Specific Filesystem Pipeline

The Board Specific Filesystem Pipeline is responsible to generate the BSROOTFS that is used by the Zynq UltraScale+ MPSoC. This is done by adding any extra configuration files for the

filesystem in the `specific-files/` folder as it is described in Section 6.4.2.

For the AXI Chip2Chip project, since a *crashkernel* was necessary, there are specific configuration files that must be copied into the filesystem. These files are responsible for the configuration and the behavior of the *crashkernel*. The files are named `kdump` and `kdump.conf` and they need to be copied in the `/etc/sysconfig/` and `/etc/` respectively. In order to do that, the copy needs to happen manually in the `script` of the CI/CD pipeline. Another prerequisite of the crashkernel is its `Image`, which is the same `Image` as the one with the ordinary Kernel. For the crashkernel to work, this `Image` needs to be copied in the `/boot/` directory with the name `vmlinuz-<kernel-version>-xilinx-v<petalinux-version>` [93].

The new script of the `make-bsrootfs` job of the Bootstrap Pipeline looks like:

```
1 $ cp -r petalinux-rootfs/lib/modules/4.19.0-xilinx-v2019.2/ bsrootfs
   /lib/modules/
2
3 $ # Additional commands
4 $ mv images/linux/Image bsrootfs/boot/vmlinuz-4.19.0-xilinx-v2019.2
5 $ cp specific-files/kdump bsrootfs/etc/sysconfig/kdump
6 $ cp specific-files/kdump.conf bsrootfs/etc/kdump.conf
```

Figure 7.5: Script of the `make-bsrootfs` job.

7.5 Results

All the pipelines exited successfully which means that there was no problem with the compilation of the projects. In order to check that everything ran as expected, several actions are needed.

1. Since the board is accessible through SSH it means that it is booted, the PetaLinux project is working and it has a network connection.

```
1 $ ssh root@zcu102-lab40-r01-33
2 Password: *****
3
4 [root@zcu102-lab40-r01-33 ~]$
```

Figure 7.6: Successful SSH connection to the Zynq UltraScale+ MPSoC.

2. The `mount` command can show if the board is using NFS for its root filesystem. The `systemctl` command can show the status of services in the filesystem. This will show if the crashkernel is configured correctly if the `kdump` service was configured and started successfully and simultaneously if the NFS filesystem is used, since the crashkernel is in the BSROOTFS on the NFS server [94,95].

```
1 kdump.service - Crash recovery kernel arming
2   Loaded: loaded (/usr/lib/systemd/system/kdump.service; enabled;>
3   Active: active since Sat 2022-06-18 16:46:47 CEST;
```

Figure 7.7: Successful crashkernel `kdump` service initialization.

3. The last step that is going to show if the Vivado project was properly compiled and that the hardware exists, the AXI Chip2Chip IP along with the other IPs need to be visible and accessible from the userspace of the Linux OS. The IPs that are implemented in the PL exist in the `/proc/device-tree/amba_pl00/` directory.

```

1 [root@zcu102-lab40-r01-33 /]$ cd /proc/device-tree/amba_pl00/
2 [root@zcu102-lab40-r01-33 amba_pl00]$ ls
3 '#address-cells'          gpio@0008000
4 '#size-cells'             i2c@a000000
5 axi_bram_ctrl@b000000     i2c@a000100
6 axi_chip2chip@b010000    i2c@a001500
7 compatible                i2c@a001600
8 gpio@a0002000             misc_clk_0
9 gpio@a0003000             misc_clk_1
10 gpio@a0004000             name
11 gpio@a0005000             ranges
12 gpio@a0006000             serial@a002000
13 gpio@a0007000             system_management_wiz@a001a00
14 [root@zcu102-lab40-r01-33 amba_pl00]$

```

Figure 7.8: PL IPs visible from userspace.

4. The last step checks if the configured registers can be manipulated. According to the specifications of the project, the register with address `0x00\_B010\_0010` should be a read-write register, while the register at `0x00\_B010\_0008` should be read-only. The access to the registers is done with the use of the `devmem` command [96].

```

1 [root@zcu102-lab40-r01-33 /]$ devmem 0x00b0100010 32 0x12345678
2 [root@zcu102-lab40-r01-33 /]$ devmem 0x00b0100010 32
3 0x12345678
4 [root@zcu102-lab40-r01-33 /]$ devmem 0x00b0100008 32
5 0x12345678

```

Figure 7.9: Register manipulation.

In line no.1 the register with address `0xb0100010` is being written and in line no.2 its value is being read. At line no.4 the register with address `0xb0100008` is being read. Lines no.3 and no.5 print the output of the read commands.

7.6 Summary

Each pipeline was tested by implementing various different projects needed by the DAQ group. Different projects raised different challenges for the project. The pipelines were built in a way such they fulfill the requirements of the project stated in Table 2.1.

During the testing phase of the project it was noticed that sometimes after U-Boot would soft reset the MPSoC, the MPSoC would stuck. The Boot Pipeline is able to capture this failure and print it to the CI/CD console. The reason behind this problem was investigated thoroughly. However, the cause of this problem was not discovered.

Overall, it can be concluded that the automated CI/CD infrastructure can successfully build

and deploy the software and the firmware of the Zynq UltraScale+ MPSoC, then boot it and monitor its boot process.

8 Conclusion

An automated CI/CD framework was researched, analyzed, designed, and implemented in order to resolve the problem of handling hundreds of boards containing the Zynq UltraScale+ MPSoC. The experience of the DAQ group on previous upgrades of the DAQ system and the research during this project, found out that building and testing software and firmware for hundreds of boards can not only be quite demanding but it can also introduce possible failures due to its complexity.

During the High-Luminosity LHC (HL-LHC) upgrade, the Compact Muon Solenoid (CMS) experiment and its Data Acquisition (DAQ) system will be upgraded. The upgrade will introduce electronics that host the Zynq UltraScale+ MPSoC. This embedded system will be included in an ATCA-based design and it will perform control and monitoring tasks for the electronics. It will run on a Linux OS based on the CentOS distribution. Hundreds of these embedded systems will be used by the DAQ group.

The automated CI/CD framework builds and delivers software and firmware for the Zynq UltraScale+ MPSoC, then it also boots and monitors its boot process until the device is fully booted and accessible over the network. The automated CI/CD framework incorporates eight pipelines that are able to build the software and the firmware for the board, deploy the images required for booting, boot it and also monitor its boot process in an interactive way. These pipelines are responsible for different aspects of the whole process:

1. The Vivado Pipeline is responsible for compiling the Vivado Project and generating the Bitstream and Hardware Description (XSA) files.
2. The PetaLinux Pipeline is responsible for compiling the PetaLinux Project and generating the boot images and the PetaLinux Root Filesystem (ROOTFS).
3. The CentOS 8 Filesystem Pipeline is responsible for creating a bare CentOS 8 Filesystem.
4. The Board Specific Filesystem Pipeline is responsible for creating the Board Specific root filesystem (BSROOTFS).
5. The Bootstrap Pipeline is responsible for bootstrapping the board by storing the boot images in the TFTP server and the BSROOTFS in the NFS server.
6. The Dedicated Boot Images Pipeline is responsible for creating the dedicated images that boot the board through JTAG in order to bootstrap the SD Card with the boot image.
7. The Boot Pipeline is responsible for booting the board and monitoring its boot process.
8. The Maestro Pipeline is responsible for orchestrating and managing the execution of all the other pipelines.

The automated CI/CD successfully bootstraps the SD Card with the required boot images in order to boot the Zynq UltraScale+ MPSoC through the Network while monitoring its boot process.

All the software and firmware pieces are built inside Docker containers in order to be more portable and more compatible in different environments. The pipelines are designed in a way that all possible jobs (e.g. all software and firmware builds) are run in a containerized environment in order to provide more independence over the underlying software and hardware of the host PC.

Table 8.1: Evaluation of requirements.

#	MoSCoW	Requirement	Solution
1	<i>Must</i>	The CI/CD pipeline builds the Vivado project	Vivado Pipeline
2	<i>Must</i>	The CI/CD pipeline builds the PetaLinux project	PetaLinux Pipeline
3	<i>Must</i>	The CI/CD pipeline builds the CentOS 8 root filesystem	CentOS 8 Root Filesystem Pipeline
4	<i>Must</i>	The CI/CD pipeline builds the Board Specific root filesystem	Board Specific Filesystem Pipeline
5	<i>Could</i>	Additional services are added to the filesystem automatically	-
6	<i>Could</i>	Additional kernel modules are added to the filesystem automatically	-
7	<i>Should</i>	The firmware binaries are deployed to the TFTP server automatically	Bootstrap Pipeline
8	<i>Should</i>	The root filesystem is deployed to the NFS server automatically	Bootstrap Pipeline
9	<i>Must</i>	The CI/CD pipeline power-cycles the Zynq UltraScale+ MPSoC	Boot Pipeline
10	<i>Must</i>	The CI/CD pipeline boots the Zynq UltraScale+ MPSoC	Boot Pipeline
11	<i>Could</i>	Dedicated U-Boot image bootstraps the SD Card	Dedicated Boot Images Pipeline
12	<i>Must</i>	The CI/CD pipeline monitors the boot process	Boot Pipeline

The evaluation of the requirements' solutions is shown in Table 8.1. All the requirements with a *Must* and *Should* priority have been met. The Bootstrap Pipeline can deploy the boot images and the BSROOTFS in the TFTP and NFS servers. A dedicated U-Boot image can bootstrap the SD Card automatically. From the research conducted in this thesis, it became apparent that it might be useful to add services and kernel modules to the filesystem automatically. However, due to time constraints, their implementation remains a topic for future work.

The research, development, and implementation of the project were presented and shared with engineers and scientists at CERN and other institutes. Particularly, a dedicated presentation has been conducted in an SoC Interest Group Meeting at CERN²⁹. A lot of discussions

²⁹<https://indico.cern.ch/event/1139381/>

took place regarding the Zynq UltraScale+ MPSoC with beneficial feedback.

The automated GitLab CI/CD framework was tested with the ZCU102 board and it successfully builds and deploys the software and the firmware for the Zynq UltraScale+ MPSoC, then boot it and monitor its boot process.

Familiarity with different topics, ranging from hardware development in FPGAs to the software and embedded Linux development in ARM processors is essential. The internals and the boot process of the Zynq UltraScale+ MPSoC need to be well understood. Xilinx documentation is spread across several documents and it is a time-consuming process to identify the document that contains the information that someone is looking for. The tools that are available hide a lot of complexity and accelerate the development of new projects. Nevertheless, the integration of the various parts and the time to develop applications should not be underestimated.

9 Future Work

It has been concluded that the automated CI/CD framework is able to build the software and the firmware for the Zynq UltraScale+ MPSoC, bootstrap it, boot it and monitor its boot process. During the development of the project, a number of topics were discovered that needed further research but were not implemented due to timing constraints. Table 9.1 shows a list of these topics.

Table 9.1: List of future work.

#	Improvement / Addition / Research Topic
1	Extend the current CI/CD framework with customized tests.
2	PetaLinux Pipeline to create the project from the BSP.
3	Additional services and modules are added automatically to the filesystem.
4	Every single job to run inside a Docker container.

The current GitLab CI/CD framework could be extended with testing capabilities. After the Boot Pipeline exits with success, automated customized tests can be executed using the GitLab CI/CD framework, to test certain functions in hundreds of Zynq UltraScale+ MPSoC boards. This automated procedure could accelerate the testing of the boards while also providing more reliable testing.

The PetaLinux Pipeline could be able to create a PetaLinux project from scratch instead of being required by the user to upload an already created PetaLinux project. An already made BSP could be used that boots the Zynq UltraScale+ MPSoC through the Network. Any other addition could be added later by the user in the meta-user layer. This could help accelerate the testing of different bitstream files since it does not affect the software and it can also help to automate even more the infrastructure.

Additional services and kernel modules can be added automatically to the filesystem. This can be done either in the PetaLinux Pipeline when the PetaLinux ROOTFS is built or in the Board Specific Filesystem Pipeline when the BSROOTFS is created. It is quite simple if the extra modules and services just need specific files to be copied into the filesystem. For more complicated actions, a configuration management tool such as *puppet*³⁰ might be needed.

Even though all the jobs that build software and firmware run inside Docker containers, jobs that create containers such as the one for the CentOS 8 ROOTFS do not run in containers. In theory, a Docker container can be created in a Docker container however it would require additional research. Hence, it is suggested as future work for the project.

³⁰<https://puppet.com/>

List of Figures

1	The Globe of Science and Innovation, together with the "Wandering of the Immeasurable" sculpture, at CERN [2].	1
1.1	View of an open LHC interconnection [7].	3
1.2	The CERN accelerator complex [11].	4
1.3	3D model of the CMS detector and its internal components [13].	5
1.4	A transverse slice through one segment of the CMS detector indicating the responses of the various detecting systems to different types of particles [19].	6
1.5	High-level diagram of the CMS DAQ system.	7
1.6	The ATCA crate located in the DAQ group's laboratory, with one DTH board sitting in the front panel while the RTM is visible in the back panel, connected with an industrial grey connector with the DTH. The Trenz Mezzanine of Zynq UltraScale+ MPSoC is sitting on the RTM, behind the grey connector, under the fan, and the cooler.	9
2.1	Block diagram of the Zynq UltraScale+ MPSoC [30].	10
3.1	Application Processing Unit (APU) block diagram [35].	15
3.2	Simplified view of the boot process [39].	16
3.3	Exception levels [42].	17
3.4	U-Boot console output during booting.	18
3.5	Kernel console output during booting.	19
3.6	Example of the pipeline structure.	20
3.7	Example of GitLab Runner picking up jobs.	21
4.1	Abstract view of Vivado workflow [63,66].	23
4.2	Packaged boot file.	25
4.3	PetaLinux workflow [69].	26
4.4	BSROOTFS creation workflow.	27
4.5	Integration.	28
4.6	Bootstrapping of the Zynq UltraScale+ MPSoC for normal operation.	29
4.7	Bootstrapping of the SD Card with dedicated U-Boot. Boot with JTAG.	30
4.8	Bootstrapping of the SD Card with dedicated images and manipulation of the boot mode.	32
4.9	Abstract view of interaction with the Zynq UltraScale+ MPSoC boot process.	33
4.10	Differentiation between cache and artifacts.	34
4.11	Example use of stages global keyword.	35
4.12	Data sharing between different pipelines.	36
4.13	An example of a pipeline triggering another pipeline.	37
5.1	Top-Level System Design of the automated CI/CD framework.	38
5.2	Boot System Design.	40
6.1	An abstract view of the Vivado Pipeline with the four stages and the generated artifacts.	42
6.2	Detailed view of the Vivado Pipeline.	43

6.3	An abstract view of the PetaLinux Pipeline with its four stages and the input and output artifacts.	44
6.4	Example use of <code>petalinux-config</code> command.	44
6.5	Example use of <code>petalinux-build</code> command.	45
6.6	Example use of <code>petalinux-package</code> command.	45
6.7	Detailed view of the PetaLinux Pipeline.	46
6.8	Flowchart of the dedicated U-Boot.	47
6.9	Detailed view of the CentOS 8 Filesystem Pipeline.	48
6.10	Example of copying the PetaLinux kernel modules in the CentOS 8 Filesystem. .	49
6.11	Detailed view of the Board Specific Filesystem Pipeline and its cooperation with the PetaLinux and CentOS 8 Filesystem Pipelines.	49
6.12	Detailed view of the Bootstrap Pipeline.	50
6.13	Detailed view of the Boot Pipeline.	51
6.14	PDU's CLI interaction.	51
6.15	Example of interacting with U-Boot through the <code>serial_line.py</code> script during booting.	52
6.16	Management of the scripts during booting.	53
6.17	The Maestro Pipeline.	54
7.1	AXI Chip2Chip Vivado project GitLab structure.	56
7.2	Variables for Vivado Project.	56
7.3	AXI Chip2Chip PetaLinux project GitLab structure.	57
7.4	Variables for PetaLinux project.	57
7.5	Script of the <code>make-bsrootfs</code> job.	58
7.6	Successful SSH connection to the Zynq UltraScale+ MPSoC.	58
7.7	Successful crashkernel <code>kdump</code> service initialization.	58
7.8	PL IPs visible from userspace.	59
7.9	Register manipulation.	59
A.1	Hardware architecture of the Zynq UltraScale+ MPSoC [97].	82
B.1	PMU firmware architecture [98].	83
C.1	CSU block diagram [38].	84
E.1	Stages of FSBL operation. [39].	86
F.1	Boot flow example of the Zynq UltraScale+ MPSoC [39].	87
H.1	Relationship between a Dockerfile, Docker Image, and Docker container [99]. .	90
H.2	Container vs Virtual Machine [58].	90
H.3	Vivado Dockerfile.	91
H.4	PetaLinux Dockerfile.	92
H.5	Accept EULA script.	93
H.6	CentOS 8 ROOTFS Dockerfile.	94
I.1	Input arguments of the <code>mkrootfs.py</code> file.	95
I.2	Form of the register file of the <code>binfmt_misc</code> service.	95
I.3	Register file of the <code>binfmt_misc</code> service in order to execute <code>aarch64</code> binaries with QEMU.	96
J.1	Configuration options in U-Boot to retrieve MAC address from EEPROM.	97

J.2	EEPROM node in the Device-Tree.	97
J.3	Boot arguments of the Linux Kernel. EEPROM link to the root note of the Device-Tree using a <i>phandle</i>	98

List of Tables

2.1	Project requirements.	11
2.2	Project preconditions.	12
3.1	I/O peripherals and interfaces.	15
3.2	Abstract boot process.	19
4.1	Built images.	25
6.1	Vivado project repository structure.	42
6.2	PetaLinux project repository structure.	43
6.3	CentOS 8 Filesystem project repository structure.	48
6.4	Scripts for the Boot Pipeline	51
7.1	Project modifications.	55
8.1	Evaluation of requirements.	62
9.1	List of future work.	64
D.1	PMU BootROM code operations [40].	85
D.2	CSU BootROM code operations [39].	85
G.1	Job keywords [82].	88
G.2	Global keywords [82].	89
I.1	DNF command input arguments.	96

Abbreviations

ALICE A Large Ion Collider Experiment.

API Application Programming Interface.

APU Application Processing Unit.

ATCA Advanced Telecommunications Computing Architecture.

ATF ARM Trusted Firmware.

ATLAS A Toroidal LHC Apparatus.

AXI Advanced eXtensible Interface.

BRAM Block-RAM.

BSP Board Support Package.

BSROOTFS Board Specific root filesystem.

BU Builder Unit.

CAN Controller Area Network.

CBR CSU BootROM.

CD Continuous Delivery/Deployment.

CERN Conseil Européen pour la Recherche Nucléaire.

CI Continuous Integration.

CI/CD Continuous Integration & Continuous Delivery/Deployment.

CIB Crypto Interface Block.

CLI Command-Line Interface.

CMS Compact Muon Solenoid.

CRC Cyclic Redundancy Check.

CSU Configuration Security Unit.

DAQ Data Acquisition.

DAST Dynamic Application Security Testing.

DHCP Dynamic Host Configuration Protocol.

DNF Dandified YUM.

DSP Digital Signal Processors.

DTB Device-Tree Blob.

DTH DAQ & Timing Hub.

ECAL Electromagnetic Calorimeter.

EEPROM Electrically Erasable Programmable ROM.

eMMC embedded Multi-Media Card.

EULA End-User License Agreement.

FED Front-end driver.

FEROL FED Readout Optical Link.

FPD Full Power Domain.

FPGA Field-Programmable Gate Array.

FPU Floating Point Unit.

FSBL First-Stage Bootloader.

FU Filter Unit.

GPIO General Purpose I/O.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HCAL Hadron Calorimeter.

HDL Hardware Description Language.

HEP High-Energy Physics.

HL-LHC High-Luminosity LHC.

HLT High Level Trigger.

I/O Input/Output.

I2C Inter-Integrated Circuit.

IDE Integrated Development Environment.

IP Intellectual Property.

IP Internet Protocol.

IPI Inter Processor Interrupt.

IPMC Intelligent Platform Management Controller.

JTAG Joint Test Action Group.

L1 Trigger Level-1 Trigger.

LHC Large Hadron Collider.

LHCb LHC beauty.

Linac4 Linear accelerator 4.

LPD Low Power Domain.

LUT Look-Up Tables.

MAC Media Access Control.

MIO Multiplexed I/O.

MPSoC Multi-Processor System-on-Chip.

MS Muon Stations.

NFS Network File System.

OBC On-Board Controller.

OCM on-chip RAM.

OS Operating System.

PBR PMU BootROM.

PC Personal Computer.

PCI Peripheral Component Interconnect.

PCIe PCI express.

PDU Power Distribution Unit.

PID Process ID.

PL Programmable Logic.

PLL Phased-Locked Loop.

PMU Platform Management Unit.

PS Proton Synchrotron.

PS Processing System.

PSB Proton Synchrotron Booster.

QEMU Quick Emulator.

RAM Random Access Memory.

RF Radiofrequency.

RHEL Red Hat Enterprise Linux.

ROM Read Only Memory.

ROOTFS Root Filesystem.

RPU Real-Time Processing Unit.

RTL Register-Transfer Level.

RTM Rear Transition Module.

RU Readout Unit.

SATA Serial Advanced Technology Attachment.

SGMII Serial Gigabit Media-Independent Interface.

SoC System-on-Chip.

SPB Secure Processor Block.

SPI Serial Peripheral Interface.

SPS Super Proton Synchrotron.

SSBL Second-Stage Bootloader.

SSH Secure Shell.

TCDS Timing and Control Distribution System.

Tcl Tool Command Language.

TCP/IP Transmission Control Protocol/Internet Protocol.

TFTP Trivial File Transfer Protocol.

UART Universal Asynchronous Receiver Transmitter.

U-Boot Universal Bootloader.

USB Universal Serial Bus.

VM Virtual Machine.

WDT Watchdog Timer.

XDC Xilinx Design Constraints.

XSA Xilinx Shell Architecture.

XSCT Xilinx Software Command-line Tool.

YAML YAML ain't markup language.

Bibliography

- [1] CERN, “Our people.” <https://home.cern/about/who-we-are/our-people>, Jan. 2022. Accessed: 21/01/2022.
- [2] Guillaume, Jeanneret, “Wandering the Immeasurable.” <https://cds.cern.ch/record/1957174>, Oct. 2014. Accessed: 21/01/2022.
- [3] CERN, “Our history.” <https://home.cern/about/who-we-are/our-history>, Jan. 2022. Accessed: 25/01/2022.
- [4] CERN, “Our IMPACT.” <https://home.cern/about/what-we-do/our-impact>, Jan. 2022. Accessed: 25/01/2022.
- [5] CERN, “Accelerators.” <https://home.cern/science/accelerators>, Jan. 2022. Accessed: 28/01/2022.
- [6] CERN, “The Large Hadron Collider.” <https://home.cern/science/accelerators/large-hadron-collider>, Jan. 2022. Accessed: 28/01/2022.
- [7] Brice, Maximilien, “View of an open LHC connection.” <https://cds.cern.ch/record/905940?ln=en>, Oct. 2005. Accessed: 28/01/2022.
- [8] CERN, “A vacuum as empty as Interstellar space.” <https://home.cern/science/engineering/vacuum-empty-interstellar-space>, Jan. 2022. Accessed: 28/01/2022.
- [9] CERN, “Pulling together: Superconducting electromagnets.” <https://home.cern/science/engineering/pulling-together-superconducting-electromagnets>, Jan. 2022. Accessed: 28/01/2022.
- [10] CERN, “Accelerating: Radiofrequency cavities.” <https://home.cern/science/engineering/accelerating-radiofrequency-cavities>, Feb. 2022. Accessed: 03/02/2022.
- [11] Esma Mobs, “The CERN accelerator complex,” July 2019. Accessed: 28/01/2022.
- [12] CERN, “CERN’s accelerator complex.” <https://home.cern/science/accelerators/accelerator-complex>, Feb. 2022. Accessed: 03/02/2022.
- [13] CERN and CMS, “3D model of the CMS detector.” https://cmsexperiment.web.cern.ch/sites/default/files/cms_160312_02.png, Feb. 2022. Accessed: 03/02/2022.
- [14] CMS, “CMS Detector.” <https://cms.cern/detector> Accessed: 03/02/2022, Feb. 2022. Accessed: 03/02/2022.
- [15] Britannica, “muon.” <https://www.britannica.com/science/muon>, Feb. 2019. Accessed: 04/02/2022.
- [16] Britannica, “hadron.” <https://www.britannica.com/science/hadron>, June 2011. Accessed: 04/02/2022.

- [17] CMS, “CMS Tracker.” <https://cms.cern/detector/identifying-tracks>, Feb. 2022. Accessed: 04/02/2022.
- [18] CMS, “Energy of Electrons and Photons (ECAL).” <https://cms.cern/detector/measuring-energy/energy-electrons-and-photons-ecal>, Feb. 2022. Accessed: 04/02/2022.
- [19] D. d’Enterria, M. Ballintijn, M. Bedjidian, D. Hoffman, O. Kodolova, C. Loizides, I. P. Lokhtin, C. Lourenco, C. Mironov, S. V. Petrushanko, C. Roland, G. Roland, F. Sikler, and G. Veres, *CMS Physics: Technical Design Report v.2: Addendum on High Density QCD with Heavy Ions*. Mar 2007. <http://cds.cern.ch/record/1019832> Accessed: 04/02/2022.
- [20] CMS, “Energy of Hadrons (HCAL).” <https://cms.cern/detector/measuring-energy/energy-hadrons-hcal>, Feb. 2022. Accessed: 04/02/2022.
- [21] CMS, “Detecting Muons.” <https://cms.cern/detector/detecting-muons>, Feb. 2022. Accessed: 04/02/2022.
- [22] CMS, “Triggering and Data Acquisition.” <https://cms.cern/detector/triggering-and-data-acquisition>, Feb. 2022. Accessed: 07/02/2022.
- [23] V. Amoiridis *et al.*, “CMS phase-2 DAQ and timing hub prototyping results and perspectives,” *JINST*, vol. 17, no. 05, p. C05003, 2022.
- [24] Mike Lamont, “LHC Commissioning: Longer term plan.” <https://lhccommissioning.web.cern.ch/schedule/LHC-long-term.htm>, May 2022. Accessed: 23/05/2022.
- [25] CERN, “High Luminosity LHC Project.” <https://hilumilhc.web.cern.ch/>, Feb. 2022. Accessed: 15/02/2022.
- [26] W. Herr and B. Muratori, “Concept of luminosity,” 2006. Accessed: 21/05/2022.
- [27] PICMG, “AdvancedTCA® Overview.” <https://www.picmg.org/openstandards/advancedtca/>, May 2022. Accessed: 21/05/2022.
- [28] CERN, “CERN IPMC.” <https://cern-ipmc.web.cern.ch/>, June 2022. Accessed: 01/06/2022.
- [29] Oracle, “PCI Local Bus.” <https://docs.oracle.com/cd/E19455-01/805-7378/6j6un038j/index.html>, Feb. 2022. Accessed: 15/02/2022.
- [30] Xilinx, “Zynq UltraScale+ MPSoC block diagram.” <https://www.xilinx.com/content/dam/xilinx/imgs/products/zynq/zynq-eg-block.PNG>, Feb. 2022. Accessed: 15/02/2022.
- [31] J. Stapleton, *DSDM: Dynamic Systems Development Method: The Method in Practice*. Addison-Wesley, 1 ed., 1986.

- [32] Xilinx, “Vivado.” <https://www.xilinx.com/products/design-tools/vivado.html>, May 2022. Accessed: 23/05/2022.
- [33] Xilinx, “PetaLinux Tools.” <https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html>, May 2022. Accessed: 23/05/2022.
- [34] Xilinx, *Zynq UltraScale+ Device Technical Reference Manual, Chapter 35: PS-PL AXI Interfaces*, Dec. 2020. UG1085, v2.2.
- [35] Xilinx, *Zynq UltraScale+ Device Technical Reference Manual, Chapter 4: Application Processing Unit*, Dec. 2020. UG1085, v2.2.
- [36] Xilinx, *Zynq UltraScale+ Device Technical Reference Manual, Chapter 28: Multiplexed I/O*, Dec. 2020. UG1085, v2.2.
- [37] Xilinx, “Zynq UltraScale+ Device Technical Reference Manual, Chapter 6: Platform Management Unit,” Dec. 2020. UG1085, v2.2.
- [38] Xilinx, *Zynq UltraScale+ Device Technical Reference Manual, Chapter 12: Security*, Dec. 2020. UG1085, v2.2.
- [39] Xilinx, *Zynq UltraScale+ MPSoC Software Developer Guide, Chapter 7: System Boot and Configuration*, July 2021. UG1137, v2021.1.
- [40] Xilinx, *Zynq UltraScale+ Device Technical Reference Manual, Chapter 11: Boot and Configuration*, Dec. 2020. UG1085, v2.2.
- [41] Xilinx, *Zynq UltraScale+ MPSoC Software Developer Guide, Chapter 8: Security Features*, July 2021. UG1137, v2021.1.
- [42] ARM, *AArch64 Exception model*, Apr. 2022. v1.2.
- [43] Javier Garcia, “SoC Course with Reference Designs: ARM Trusted Firmware (ATF).” [ohwr.org/project/soc-course/wikis/ARM-Trusted-Firmware-\(ATF\)](https://ohwr.org/project/soc-course/wikis/ARM-Trusted-Firmware-(ATF)), June 2020. Accessed: 15/05/2022.
- [44] Simon Glass, “U-Boot Command Line Interface.” <http://www.denx.de/wiki/Knowhow/DULG/UBootCommandLineInterface>, June 2020. Accessed: 15/05/2022.
- [45] DENX, “Environment Variables Commands.” <http://www.denx.de/wiki/Knowhow/DULG/UBootCmdGroupEnvironment>, May 2008. Accessed: 15/05/2022.
- [46] Henry Beberman, “U-Boot Environment Variables.” <http://www.denx.de/wiki/Knowhow/DULG/UBootEnvVariables>, June 2018. Accessed: 15/05/2022.
- [47] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, “Design and implementation of the sun network filesystem,” 1985. Accessed: 15/05/2022.
- [48] D. K. R. Sollins, “The TFTP Protocol (Revision 2).” RFC 1350, July 1992.

- [49] Grant Likely, “Open Firmware and Devicetree: Linux and the Devicetree.” <https://www.kernel.org/doc/html/latest/devicetree/usage-model.html>, May 2022. Accessed: 16/05/2022.
- [50] M. Tim Jone, “Inside the Linux boot process.” <https://web.archive.org/web/20080531150425/http://www-128.ibm.com/developerworks/library/l-linuxboot/index.html>, May 2006. Accessed: 16/05/2022.
- [51] FreeBSD, “BSD System Manager’s Manual: init.” <https://www.freebsd.org/cgi/man.cgi?query=init&sektion=8&manpath=FreeBSD+13.0-RELEASE+and+Ports>, Aug. 2019. Accessed: 16/05/2022.
- [52] RedHat, “RHEL 8: Chapter 13. Introduction to systemd.” https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_basic_system_settings/introduction-to-systemd_configuring-basic-system-settings, May 2022. Accessed: 24/05/2022.
- [53] N. Dzemaili, “A reliable booting system for Zynq Ultrascale+ MPSoC devices,” Feb 2021. Presented 17 Mar 2021.
- [54] Amazon, “What is Continuous Integration?.” <https://aws.amazon.com/devops/continuous-integration/>, June 2022. Accessed: 03/06/2022.
- [55] Amazon, “What is Continuous Delivery?.” <https://aws.amazon.com/devops/continuous-delivery/>, June 2022. Accessed: 03/06/2022.
- [56] GitLab, “CI/CD pipelines.” <https://docs.gitlab.com/ee/ci/pipelines/>, June 2022. Accessed: 03/06/2022.
- [57] GitLab, “The .gitlab-ci.yml file.” https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html, June 2022. Accessed: 03/06/2022.
- [58] Docker, “What is a Container?.” <https://www.docker.com/resources/what-container/>, June 2022. Accessed: 03/06/2022.
- [59] GitLab, “GitLab Runner.” <https://docs.gitlab.com/runner/>, June 2022. Accessed: 03/06/2022.
- [60] Javier Garcia, “SoC Course with Reference Designs: Reverse engineering the xsa file.” <https://ohwr.org/project/soc-course/wikis/Reverse-Engineering-the-XSA-File>, June 2021. Accessed: 23/05/2022.
- [61] Xilinx, *Vivado Design Suite User Guide: Getting Started, Chapter 1: Vivado Design Suite Overview*, June 2021. UG910, v2021.1.
- [62] Eric S. Raymond, “The Jargon File: shell.” <http://www.catb.org/jargon/html/S/shell.html>, Apr. 2015. Accessed: 26/05/2022.
- [63] Xilinx, “Vivado Design Suite User Guide: Design Flows Overview, Chapter 2: Understanding Use Models,” Apr. 2022. UG892, v2022.1.

- [64] R. Cofer and B. F. Harding, “Chapter 7 - design implementation,” in *Rapid System Prototyping with FPGAs* (R. Cofer and B. F. Harding, eds.), Embedded Technology, pp. 103–125, Burlington: Newnes, 2006.
- [65] Xilinx, *Vivado Design Suite User Guide: System-Level Design Entry, Chapter 2: Working with Projects*, May 2022. UG895, v2022.1.
- [66] Xilinx, “Vivado Design Suite User Guide: Design Flows Overview, Chapter 1: Vivado System-Level Design Flows,” Apr. 2022. UG892, v2022.1.
- [67] VHDLWIZ, “Terminology, netlist.” <https://vhdlwhiz.com/terminology/netlist/>, May 2022. Accessed: 26/05/2022.
- [68] Xilinx, *Vivado Design Suite User Guide: Design Analysis and Closure Techniques*, Oct. 2021. UG906, v2021.2.
- [69] Xilinx, *PetaLinux Tools Documentation Reference Guide, Chapter 1: Overview*, Apr. 2022. UG1144, v2022.1.
- [70] Xilinx, *PetaLinux Tools Documentation Reference Guide, Chapter 3: Creating a Project*, Apr. 2022. UG1144, v2022.1.
- [71] Xilinx, “Xilinx Wiki: HSI debugging and optimization techniques.” <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841693/HSI+debugging+and+optimization+techniques>, Sept. 2020. Accessed: 28/05/2022.
- [72] Xilinx, *Xilinx Software Command-Line Tools (XSCT): Reference Guide (UG1208)*, June 2016. UG1208, v2016.2.
- [73] IBM, “Root file system.” <https://www.ibm.com/docs/pl/aix/7.1?topic=tree-root-file-system>, Dec. 2021. Accessed: 28/05/2022.
- [74] CentOS, “About CentOS: CentOS Linux.” <https://www.centos.org/about/>, Feb. 2022. Accessed: 15/02/2022.
- [75] Debian, “Raspberry Pi: qemu-user-static.” <https://wiki.debian.org/RaspberryPi/qemu-user-static>, Jan. 2022. Accessed: 28/05/2022.
- [76] QEMU, “QEMU.” <https://gitlab.com/qemu-project/qemu>, May 2022. Accessed: 28/05/2022.
- [77] Guillem Jover, “QEMU User Emulator (static version).” <http://manpages.ubuntu.com/manpages/jammy/man1/qemu-user-static.1.html>, May 2022. Accessed: 28/05/2022.
- [78] ARM, “TFTP server.” https://www.keil.com/pack/doc/mw/Network/html/group__net_t_f_t_ps___func.html, May 2022. Accessed: 29/05/2022.
- [79] ArchLinux, “Diskless system.” https://wiki.archlinux.org/title/diskless_system, Apr. 2022. Accessed: 29/05/2022.

- [80] Wolfgang Denk, “MMC Devices.” <https://www.denx.de/wiki/Knowhow/DULG/UBootCmdGroupMMC>, Nov. 2019. Accessed: 02/06/2022.
- [81] Xilinx, *Zynq UltraScale+ Devices Register Reference: Module Summary - CRL_APB Module*, May 2022. UG1087, v1.9.
- [82] GitLab, “.gitlab-ci.yml keyword reference.” <https://docs.gitlab.com/ee/ci/yaml/>, June 2022. Accessed: 04/06/2022.
- [83] Mark Pundsack, “GitLab Container Registry.” <https://about.gitlab.com/blog/2016/05/23/gitlab-container-registry/>, May 2016. Accessed: 09/06/2022.
- [84] Xilinx, *PetaLinux Tools Documentation Reference Guide, Appendix K: PetaLinux Commands*, Apr. 2022. UG1144, v2022.1.
- [85] Xilinx, *PetaLinux Tools Documentation Reference Guide, Chapter 10: Advanced Configurations*, Apr. 2022. UG1144, v2022.1.
- [86] Read the docs, “DNF Command Reference.” https://dnf.readthedocs.io/en/latest/command_ref.html, June 2022. Accessed: 11/06/2022.
- [87] Debian, “What is a package manager?” <https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html>, June 2022. Accessed: 11/06/2022.
- [88] CentOS, “Software Selection.” <https://docs.centos.org/en-US/centos/install-guide/SoftwareSpoke-x86/>, June 2022. Accessed: 11/06/2022.
- [89] Andrew Tridgell, Paul Mackerras, “rsync(1) - Linux man page.” <https://linux.die.net/man/1/rsync>, June 2022. Accessed: 12/06/2022.
- [90] SSH, “SSH Protocol – Secure Remote Login and File Transfer.” <https://www.ssh.com/academy/ssh/protocol>, June 2022. Accessed: 12/06/2022.
- [91] Noah Spurrier, “Pexpect version 4.8.” <https://pexpect.readthedocs.io/en/stable/>, June 2022. Accessed: 12/06/2022.
- [92] Xilinx, “AXI Chip2Chip.” <https://www.xilinx.com/products/intellectual-property/axi-chip2chip.html>, June 2022. Accessed: 16/06/2022.
- [93] RedHat, “Kernel crash dump guide.” https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/kernel_administration_guide/kernel_crash_dump_guide, June 2022. Accessed: 17/06/2022.
- [94] Linux, “mount(8) — Linux manual page.” <https://man7.org/linux/man-pages/man8/mount.8.html>, June 2022. Accessed: 18/06/2022.
- [95] Linux, “systemctl(1) — Linux manual page.” <https://man7.org/linux/man-pages/man1/systemctl.1.html>, June 2022. Accessed: 18/06/2022.
- [96] BoxMatrix, “Property:devmem.” <https://boxmatrix.info/wiki/Property:devmem>, June 2022. Accessed: 18/06/2022.

- [97] Xilinx, *Zynq UltraScale+ MPSoC Software Developer Guide, Chapter 2: Programming View of Zynq UltraScale+ MPSoC Devices*, July 2021. UG1137, v2021.1.
- [98] Xilinx, *Zynq UltraScale+ MPSoC Software Developer Guide, Chapter 10: Platform Management Unit Firmware*, July 2021. UG1137, v2021.1.
- [99] Docker, “Sample Application.” https://docs.docker.com/get-started/02_our_app/, June 2022. Accessed: 14/06/2022.
- [100] GitLab, “Releases.” <https://docs.gitlab.com/ee/user/project/releases/index.html>, June 2022. Accessed: 04/06/2022.
- [101] IBM, “Docker.” <https://www.ibm.com/cloud/learn/docker>, June 2022. Accessed: 14/06/2022.
- [102] Read the docs, “Kernel Support for miscellaneous Binary Formats (binfmt_misc).” <https://www.kernel.org/doc/html/latest/admin-guide/binfmt-misc.html>, June 2022. Accessed: 14/06/2022.
- [103] Xilinx, “Zynq UltraScale+ MPSoC: How to get MAC address from EEPROM on ZCU102 Board using PetaLinux?.” https://support.xilinx.com/s/article/70176?language=en_US, Apr. 2021. Accessed: 14/06/2022.
- [104] Kernel, “Pinctrl-bindings.” <https://www.kernel.org/doc/Documentation/devicetree/bindings/pinctrl/pinctrl-bindings.txt>, June 2022. Accessed: 14/06/2022.
- [105] eLinux, “Power.org™ Standard for Embedded Power Architecture™ Platform Requirements (ePAPR).” https://elinux.org/images/c/cf/Power_ePAPR_APPROVED_v1.1.pdf, June 2022. Accessed: 17/06/2022.
- [106] Xilinx, *ZCU102 Evaluation Board User Guide, Chapter 3: Board Component Description*, June 2019. UG1182, v1.6.
- [107] ARM, “TFTP (remote/network kernel) using U-Boot.” <https://community.arm.com/oss-platforms/w/docs/495/tftp-remote-network-kernel-using-u-boot>, Apr. 2020. Accessed: 14/06/2022.
- [108] Xilinx, “u-boot-xlnx.” <https://github.com/Xilinx/u-boot-xlnx/blob/master/README>, Apr. 2022. Accessed: 14/06/2022.

Appendices

A Zynq UltraScale+ MPSoC block diagram

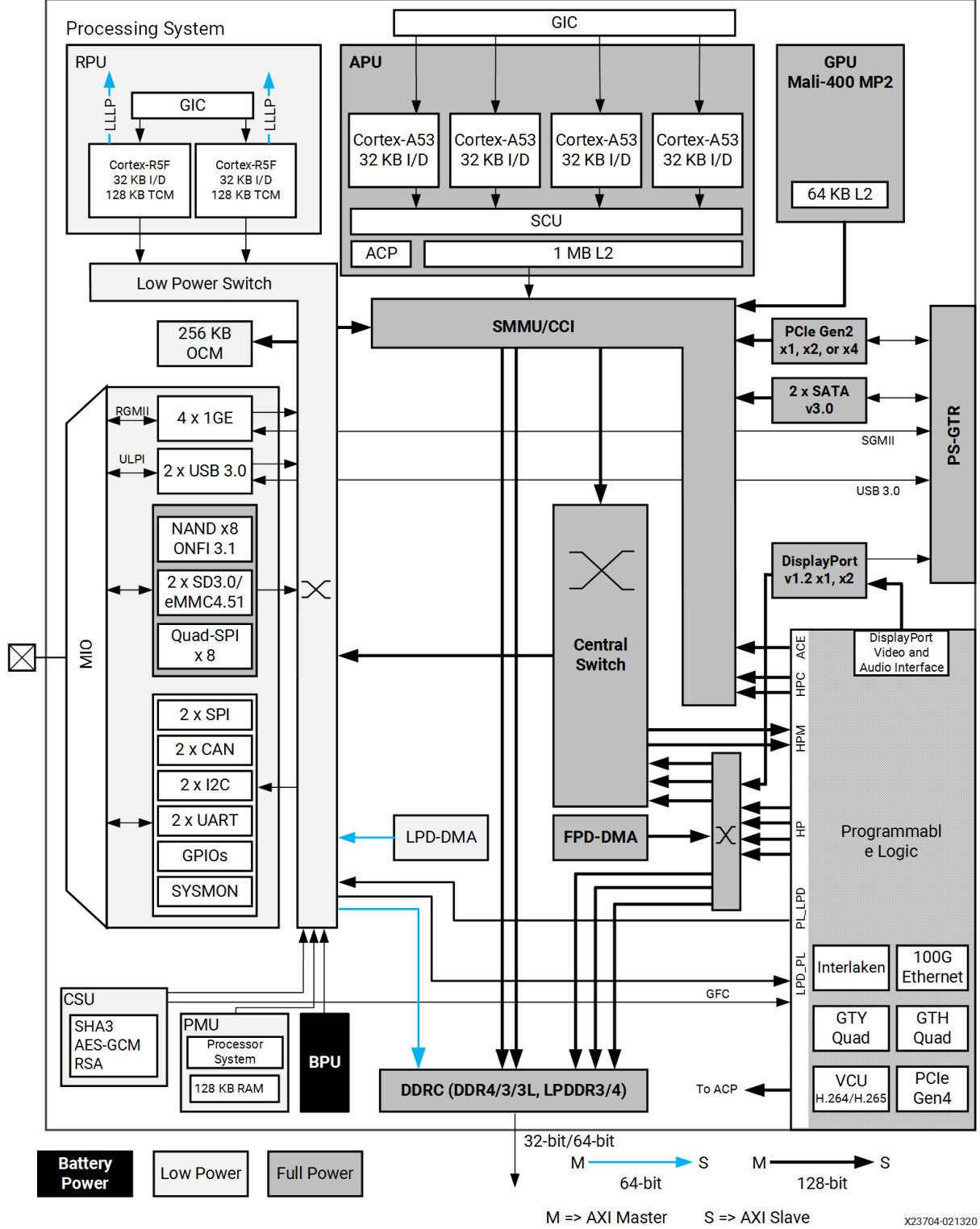


Figure A.1: Hardware architecture of the Zynq UltraScale+ MPSoC [97].

B PMU internals and workings

The Platform Management Unit (PMU) is a triple-redundant MicroBlaze processor. It is responsible for the initialization of the system before boot, power and error management, and execution of software tests. The code that is stored in the 32KB PMU ROM is responsible for the initialization of the system. The rest of the PMU functionality is handled by the PMU firmware which runs on a separate 128KB RAM [37].

The PMU firmware architecture is designed in a modular way. This provides the flexibility to add new functionality as a module. Each feature that has a distinct functionality is designed in a separate module so the PMU can load only the required modules for a specific application. The PMU firmware architecture block diagram is shown below in Figure B.1:

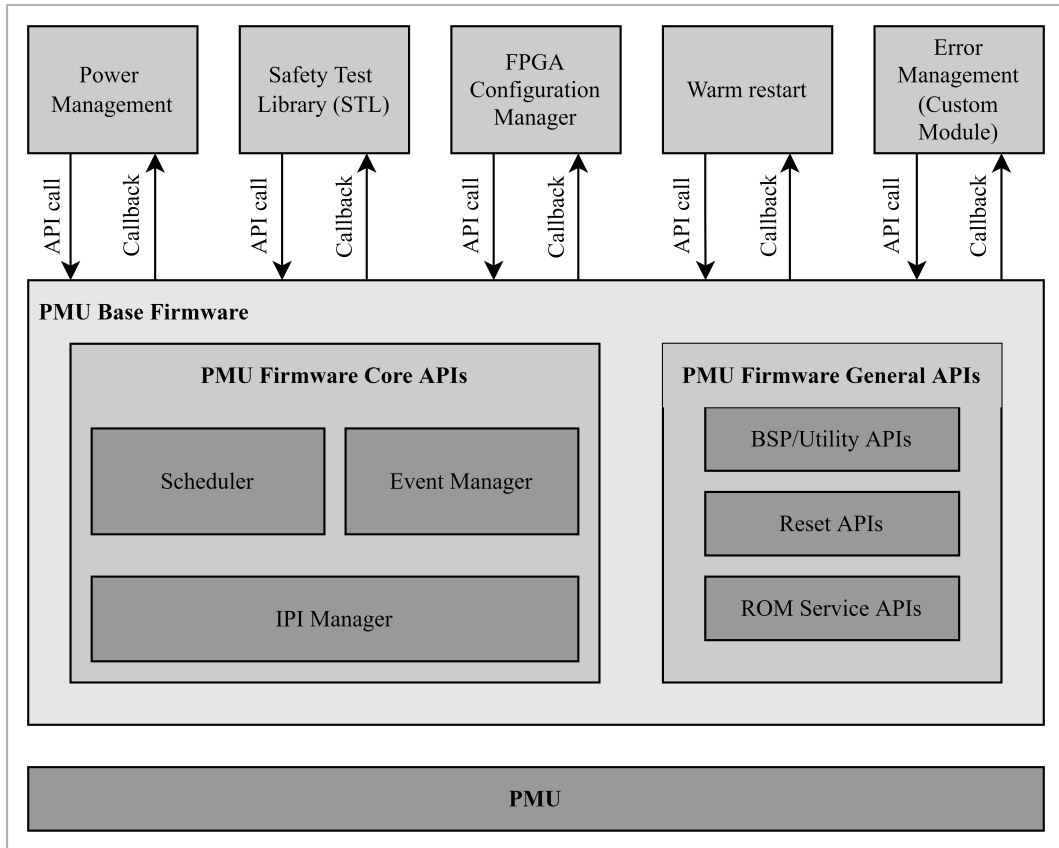


Figure B.1: PMU firmware architecture [98].

The PMU base firmware is responsible for initializing, configuring, and servicing the modules. The functions that are provided by the base firmware to the modules are called APIs and can be used by the modules in order to perform the required actions.

The PMU firmware core APIs are necessary for the operation of the PMU. They provide access to the scheduler, the event manager, and the IPI manager who is responsible for handling the IPI interrupts (interface for communication between the processing units).

There is a handful of modules in the PMU firmware. As an example, the power management module is responsible for powering on and off the different parts of the Zynq UltraScale+ MPSoC, among other things [98].

D PMU & CSU BootROM codes

D.1 The PMU BootROM code

The PMU BootROM (PBR) code is stored in the ROM of the PMU and is the first piece of software that is executed. The PBR code executes several mandatory operations that are required for the initialization of the system. If all the operations are exited successfully, the PBR moves forward with releasing the CSU and entering the PMU service mode. If errors have occurred, it generates an error and enters an error state. A list of the most important tasks is shown in Table D.1.

Table D.1: PMU BootROM code operations [40].

#	Task
1	Initialize the MicroBlaze processor and the System Monitor (PS SYSMON)
2	Initialize and validate the Phased-Locked Loop (PLL)
3	Clear the PMU and CSU RAM
4	Clear the LPD and FPD
5	Release the CSU reset or enter an error state

D.2 The CSU BootROM code

The CSU BootROM (CBR) code is stored in the ROM of the CSU and it is executed as long as the CSU is released by the PMU. The main objective of the CBR code is to load the FSBL to the OCM. Table D.2 shows a list of the most important operations of the CBR.

Table D.2: CSU BootROM code operations [39].

#	Task
1	Initialize the on-chip RAM (OCM)
2	Determine the boot mode by reading the value of the <i>boot-mode register</i>
3	Search for the boot image
4	Load the FSBL in the OCM
5	Load the PMU firmware in the PMU RAM (<i>Optional</i>)

The Zynq UltraScale+ MPSoC supports booting from QSPI Flash, SD Card, USB, eMMC and JTAG. The boot mode is determined by the value of the boot mode register. The boot-mode register is set either by the value of bootstrapping pins on the board or from software with access to the registers [40].

E The FSBL working principle

The main objective of the FSBL is to load the Second-Stage Bootloader (SSBL), U-Boot. Apart from this, there are multiple other tasks and operations that are handled by the FSBL. This process can be split up into four different stages (Figure E.1) and are described below:

1. **Initialization:** The FSBL initializes the PS, the DDR memory, and the PL.
2. **Boot device initialization:** The primary boot device is identified by reading the value of the boot mode register. The FSBL proceeds to initialize the boot device using the corresponding drivers. The boot header is validated at this point and a hand-off structure of necessary data is prepared for the ATF.
3. **Partition loading:** Every partition of the boot image is validated along with its partition header. Then, the FSBL continues by copying the partitions to the corresponding memory locations. The PMU firmware is copied in the PMU RAM, the ATF is copied in the OCM, and U-Boot is copied in the DDR.
4. **Hand-off:** Configure the ATF and set the program counter for the execution of U-Boot.

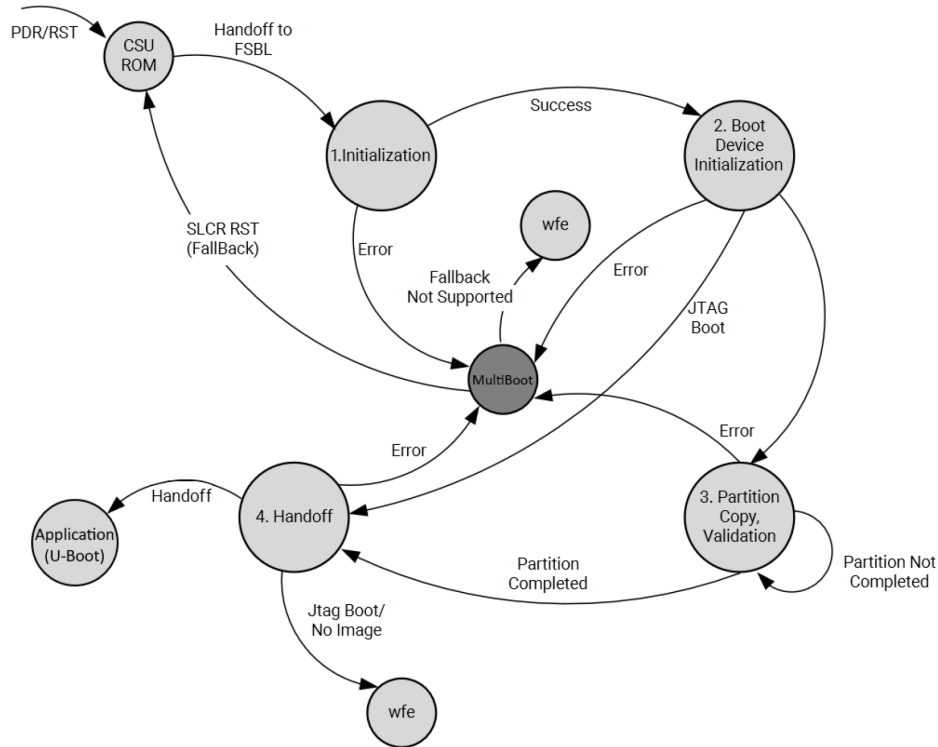
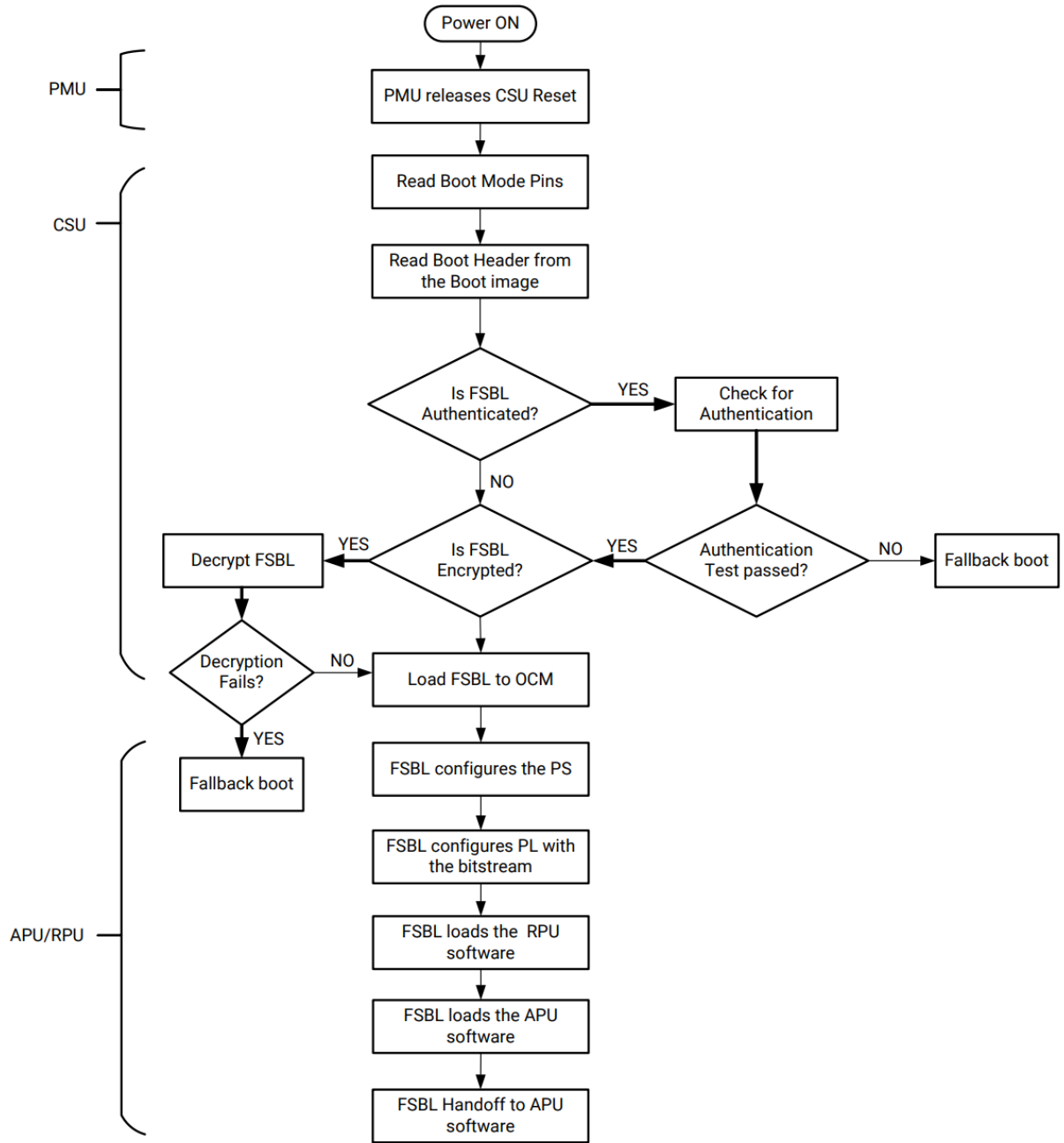


Figure E.1: Stages of FSBL operation. [39].

In case an error occurs during the execution of the FSBL and fallbacks are supported, the MultiBoot offset register is updated and a Watchdog Timer (WDT) reset is expected to occur. After the reset, the CSU BootROM and the FSBL fetch the new boot image from the newly calculated address in the MultiBoot register [39, 40].

F Boot flow example



X14935-070717

Figure F.1: Boot flow example of the Zynq UltraScale+ MPSoC [39].

G GitLab CI/CD pipeline keywords

G.1 Job keywords

This is a list of all the *Job keywords* used in a GitLab CI/CD pipeline as of GitLab version 15.1.

Table G.1: Job keywords [82].

#	Keyword	Description
1	<code>after_script</code>	Set of commands that is executed after the main script
2	<code>allow_failure</code>	Allow the pipeline to continue if the job fails
3	<code>artifacts</code>	List of files that are saved if the job is successful
4	<code>before_script</code>	List of commands that is executed before the main script
5	<code>cache</code>	List of files that is cached between consecutive runs
6	<code>coverage</code>	Regular exceptions to be checked in the job log
7	<code>dast_configuration</code>	DAST profile configuration on job level
8	<code>dependencies</code>	List of jobs to fetch artifacts from
9	<code>environment</code>	Environment name that the job is deployed
10	<code>except</code>	When the job is not created
11	<code>extends</code>	List of configuration entries that this job inherits from
12	<code>image</code>	Docker Image to be used
13	<code>inherit</code>	List of global defaults that the job inherits
14	<code>uninterruptible</code>	Cancel a job if it is made redundant by a newer run
15	<code>needs</code>	Jobs that need to be executed earlier
16	<code>only</code>	Specifies when the preconditions to run the job
17	<code>pages</code>	Job output is uploaded to use with GitLab pages
18	<code>parallel</code>	Number of instances of a job that should run in parallel
19	<code>release</code>	The Runner generates a release project [100]
20	<code>resource_group</code>	Limit job concurrency
21	<code>retry</code>	Number of times a job can re-run in case of failure
22	<code>rules</code>	Specify attributes of a job and if it's created or not
23	<code>script</code>	Main script of the job
24	<code>secrets</code>	List of CI/CD secrets that the job needs
25	<code>services</code>	Docker services images to be used
26	<code>stage</code>	The stage that the job belongs
27	<code>tags</code>	Tags used to match with Runner tags
28	<code>timeout</code>	Timeout for the job
29	<code>trigger</code>	Defines which pipeline is triggered in this job
30	<code>variables</code>	Variable definition on a job level
31	<code>when</code>	When the job is executed

A *job keyword* can use secondary keywords in order to make its functionality more specific [82].

G.2 Global keywords

This is a list of all the *Global keywords* used in a GitLab CI/CD pipeline as of GitLab version 15.1.

Table G.2: Global keywords [82].

#	Keyword	Description
1	<code>default</code>	Customized values for for job keywords used as defaults
2	<code>include</code>	Import extra YAML files for configuration
3	<code>stages</code>	The different stages of the project in order
4	<code>variables</code>	CI/CD variables that are used in the jobs of the project
5	<code>workflow</code>	Type of pipelines that run

A *global keyword* can use secondary keywords in order to make its functionality more specific [82].

H What is a Docker Container?

A container is a piece of software that wraps up code with all its dependencies and packages them in a way that the application can run from one computing environment to another. A Docker container image includes all the necessary tools, libraries, and settings to run an application. The initial step to building a Docker container is to create its **Dockerfile**. The **Dockerfile** is just a text script of instructions that describes the Docker Image. When the script is built, the Docker Image is generated. When a Docker Image runs, a Docker container is created. Multiple containers can be generated from a single Image [99].

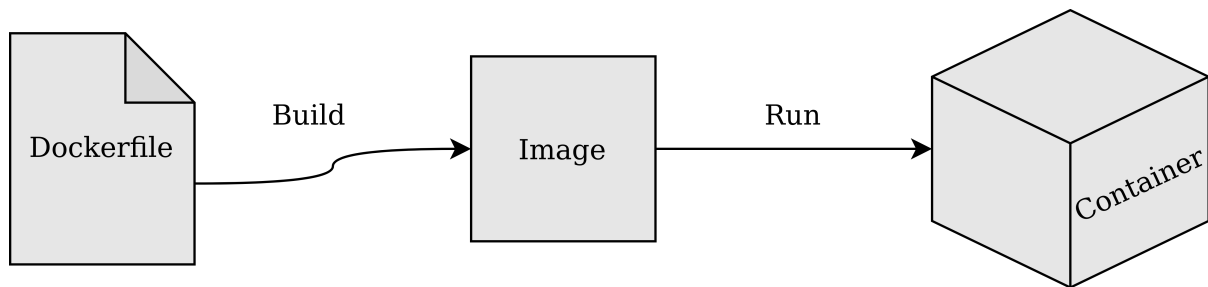


Figure H.1: Relationship between a Dockerfile, Docker Image, and Docker container [99].

Docker containers are usually discussed together with a Virtual Machine (VM). There are a lot of characteristics that containers and VMs have in common, such as resource isolation but their fundamental difference is that a container virtualizes the OS while a VM virtualizes the underlying hardware. This is what makes containers more lightweight and resource-efficient [58, 101]. Figure H.2 shows the comparison between a Docker container and a VM.

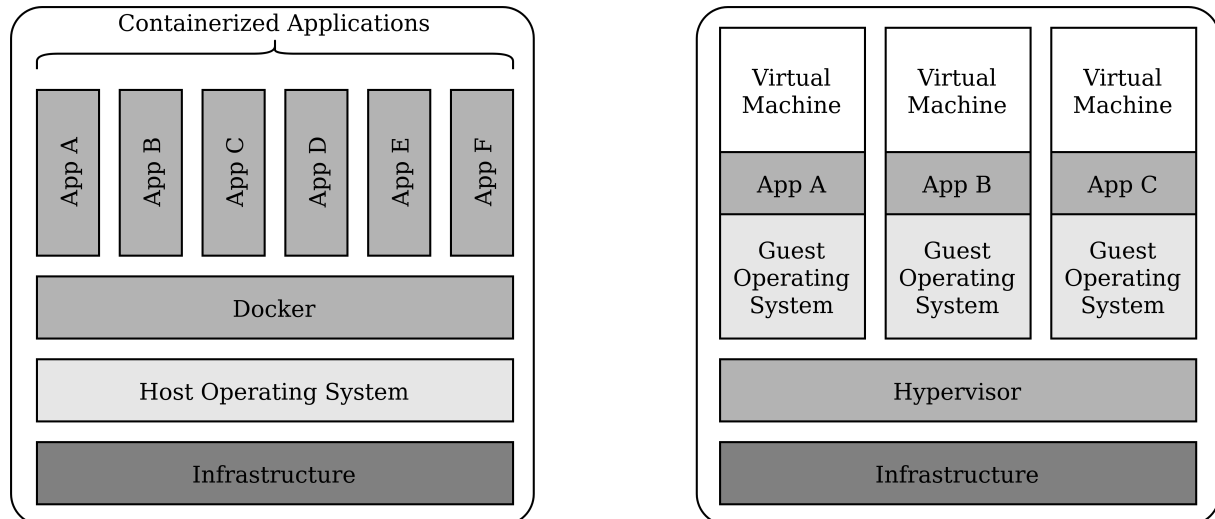


Figure H.2: Container vs Virtual Machine [58].

Docker Images are stored in Docker Registries. GitLab offers its own Docker Registry so images can be directly pushed to the GitLab Registry after their creation. Apart from its own private registry, CERN has published some of its Docker Images on Docker Hub³¹.

³¹CERN's Docker Hub profile: <https://hub.docker.com/u/cern/>.

H.1 Vivado Docker Container

The Vivado Docker Container contains Vivado and all its dependencies. These dependencies are needed both for installation and operation. The following `Dockerfile` is able to create a Docker Image that contains all the necessary tools to run a Vivado project in it. In order to achieve this, it needs the Vivado tar file along with a configuration file (`install_config.txt`) for the installation and the file which sets up the licenses (`settings-xilinx-2019.2.sh`).

```

1 FROM gitlab-registry.cern.ch/linuxsupport/c8-base
2
3 ARG VIVADO_FILE
4 ARG VIVADO_TAR_FILE=${VIVADO_FILE}.tar.gz
5 COPY ${VIVADO_TAR_FILE} /
6
7 # Install dependencies
8 RUN dnf install -y gcc gcc-c++ make java-1.8.0-openjdk glibc-
    langpack-en sudo ncurses-compat-libs vim passwd && dnf clean all
9
10 # Copy necessary files into the project
11 COPY install_config.txt settings-xilinx-2019.2.sh /
12
13 # Install vivado
14 RUN mkdir -p /opt && \
15     ls -la && \
16     echo "Extracting Vivado..." && \
17     tar -xzf ${VIVADO_TAR_FILE} --directory /opt/ && \
18     echo "Removing Tarball..." && \
19     rm -rf ${VIVADO_TAR_FILE} && \
20     echo "Installation..." && \
21     /opt/${VIVADO_FILE}/xsetup --agree 3rdPartyEULA,WebTalkTerms,
    XilinxEULA --batch Install --config /install_config.txt && \
22     rm -f /install_config.txt
23
24 # Vivado User creation and setup
25 RUN echo "Creating Vivado User..." && \
26     useradd -s /bin/bash vivado && \
27     passwd -f -u vivado && \
28     usermod -aG wheel vivado && \
29     echo "vivado ALL=(ALL) NOPASSWD: ALL" >> /etc/sudoers && \
30     echo "source /opt/Xilinx/Vivado/2019.2/settings64.sh" >> /home/
    vivado/.bashrc && \
31     mv /settings-xilinx-2019.2.sh /opt/Xilinx/ && \
32     echo "source /settings-xilinx-2019.2.sh" >> /home/vivado/.bashrc
33
34 # Setup the environment
35 USER vivado
36 ENV HOME /home/vivado
37 ENV LANG en_US.UTF-8
38 ENV LANGUAGE en_US:en
39 ENV LC_ALL en_US.UTF-8
40 WORKDIR /home/vivado

```

Figure H.3: Vivado Dockerfile.

H.2 PetaLinux Docker Container

The PetaLinux Docker Container contains the PetaLinux tools with all their dependencies. The following Dockerfile is able to create a Docker Image that contains all the necessary tools to run a PetaLinux project in it. In order to achieve this, the PetaLinux installer is needed.

```

1 FROM gitlab-registry.cern.ch/linuxsupport/c8-base
2
3 ARG PETA_RUN_FILE
4 COPY accept-eula.sh ${PETA_RUN_FILE} /
5
6 # Download required dependencies
7 RUN dnf install --skip-broken -y autoconf automake bison bzip2
   chrpath \diffstat diffutils dos2unix expect flex freetype gcc gcc
   -c++ git glibc-devel glibc-langpack-en glib2-devel iproute
   libtool make ncurses ncurses-compat-libs ncurses-devel net-tools
   openssl-devel passwd patch perl python2 rsync SDL-devel socat
   sudo texinfo tftp-server unzip vim-common wget which xterm zlib-
   devel && dnf clean all
8
9 # Create symbolic link for python
10 RUN ln -s /usr/bin/python2 /usr/bin/python
11
12 # Create the PetaLinux User
13 RUN useradd plnx && \
14     passwd -f -u plnx && \
15     usermod -aG wheel plnx && \
16     echo "plnx ALL=(ALL) NOPASSWD: ALL" >> /etc/sudoers
17
18 COPY accept-eula.sh ${PETA_RUN_FILE} /
19
20 # Install PetaLinux
21 RUN chmod a+rx /${PETA_RUN_FILE} && \
22     chmod a+rx /accept-eula.sh && \
23     mkdir -p /opt/Xilinx && \
24     chmod 777 /tmp /opt/Xilinx && \
25     cd /tmp && \
26     sudo -u plnx -i /accept-eula.sh /${PETA_RUN_FILE} /opt/Xilinx/
   petalinux-v2019.2 && \
27     rm -f /${PETA_RUN_FILE} /accept-eula.sh && \
28     echo "Print Installation Log" && \
29     cat /home/plnx/petalinux_installation_log
30
31 # Setup environment
32 USER plnx
33 ENV HOME /home/plnx
34 ENV LANG en_US.UTF-8
35 ENV LANGUAGE en_US:en
36 ENV LC_ALL en_US.UTF-8
37 WORKDIR /home/plnx
38
39 # Add PetaLinux tools to path
40 RUN echo "source /opt/Xilinx/petalinux-v2019.2/settings.sh" >> /home
   /plnx/.bashrc

```

Figure H.4: PetaLinux Dockerfile.

H.2.1 Accept EULA script

During the installation of PetaLinux, the interactive shell expects from the user to agree with the EULA. A script (`accept-eula.sh`) was implemented in order to interact with the shell and accept the EULA automatically.

```
1  #!/usr/bin/expect
2
3  set timeout -1
4  set install_dir [lindex $argv 1]
5  set installer [lindex $argv 0]
6
7  spawn $installer $install_dir
8  set timeout 2
9  expect {
10     "ERROR: Invalid options:" {spawn $installer -d $install_dir }
11     timeout { }
12 }
13
14 set timeout 600
15 expect "Press Enter to display the license agreements"
16 send "\r"
17 set timeout 2
18
19 expect {
20     "* >*" {send "y\r"}
21     timeout {send "q"; sleep 1; exp_continue}
22 }
23 expect {
24     "* >*" {send "y\r"}
25     timeout {send "q"; sleep 1; exp_continue}
26 }
27 expect {
28     "* >*" {send "y\r"}
29     timeout {send "q"; sleep 1; exp_continue}
30 }
31
32 set timeout -1
33 expect "INFO: Checking PetaLinux installer integrity..."
34 expect "INFO: PetaLinux SDK has been installed!!!"
35 #interact
```

Figure H.5: Accept EULA script.

H.3 CentOS 8 ROOTFS Docker Container

The CentOS 8 ROOTFS Docker Container includes all the required tools to generate the CentOS 8 ROOTFS for `aarch64` architecture. The Docker Container is created with the following steps:

1. Required packages and tools are installed using `dnf`.
2. QEMU is downloaded and becomes executable.
3. It is copied into the necessary directories.

4. The config files for the ROOTFS that were described in Section 6.4 are copied into the container.
5. The `mkrootfs.py` file which is responsible for generating the CentOS 8 ROOTFS is copied into the container in the `/usr/local/bin` directory.

```
1 FROM gitlab-registry.cern.ch/linuxsupport/c8-base
2
3 RUN dnf install -y python3 rsync python3-augeas
4
5 RUN curl -S -L https://github.com/multiarch/qemu-user-static/
   releases/download/v4.0.0/qemu-aarch64-static -o qemu-aarch64-
   static \
6     && chmod +x qemu-aarch64-static
7 RUN mkdir -p /usr/local/bin \
8     && cp -a qemu-aarch64-static /usr/local/bin
9 RUN mkdir -p /arm/sysroot/usr/local/bin \
10    && mv qemu-aarch64-static /arm/sysroot/usr/local/bin
11
12 COPY files/ /arm/files/
13 COPY mkrootfs.py /usr/local/bin/
14
15 RUN ln -s /usr/bin/python3 /usr/bin/python
```

Figure H.6: CentOS 8 ROOTFS Dockerfile.

I CentOS 8 Filesystem Generation

The CentOS 8 Filesystem is generated by the `mkrootfs.py` file. The `mkrootfs.py` is a python file that essentially runs `dnf` commands to generate the filesystem.

I.1 Input Configuration and Options

The `mkrootfs.py` file accepts 4 input arguments. When the script is executed, the arguments are parsed and the configuration is applied accordingly. The input arguments of the `mkrootfs.py` file are:

1. **DNF Configuration** `--dnf_conf`: The `dnf` command is configured in order to fetch packages from the CERN repositories. The DNF configuration is implemented in the `dnf.conf` file. The `mkrootfs.py` file accepts as input the DNF configuration file and applies it to the `dnf` command.
2. **Root Directory** `--root`: The filesystem is generated in a directory in the Docker container. The path of this directory can be configured.
3. **Architecture** `--arch`: The architecture of the generated root filesystem.
4. **Extra packages** `--extra`: A text file which contains the extra packages that will be installed after the generation of the root filesystem.

```
1 mkrootfs.py --root='/arm/sysroot' --dnf_conf='arm/files/dnf.conf' --
  arch=aarch64 --extra='/arm/files/extra_rpms.txt'
```

Figure I.1: Input arguments of the `mkrootfs.py` file.

I.2 Additional Binary Formats

In order to allow additional binary formats to run on the host PC the `binfmt_misc` service needs to be configured. This service allows arbitrary binary file formats to be passed to certain userspace applications such as VMs and Containers [102].

The `binfmt_misc` service needs to be informed to use QEMU for executables with `aarch64` architecture. A register file is used in order to pass the necessary information to the `binfmt_misc`. The name of the file is `qemu-aarch64.conf` and is located in the `/etc/binfmt.d` directory. The register file follows the format shown in Figure I.2.

```
1 :<name>:<type>:<offset>:<magic>:<mask>:<interpreter>:<flags>:
```

Figure I.2: Form of the register file of the `binfmt_misc` service.

These specific keywords used in the register file are described below:

- `<name>`: Name of the binary format.

- **<type>**: Type of recognition for the binary format. It is either E or M.
 - If it is E, the file format is determined by the extension which is **<magic>**, while **<offset>** and **<mask>** are ignored.
 - If it is M, the file format is identified by the **<magic>** number with an absolute **<offset>** which is default to 0 and the **<mask>** is a bitmask which is default to 0xFF.
- **<interpreter>**: Program that runs with the binary file as an argument.
- **<flag>**: Optional. Extra configuration.

The configuration of the register file of the `binfmt_misc` service that executes `aarch64` executables with QEMU are shown in Figure I.3.

```
1 :qemu-aarch64:M::\x7fELF\x02\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00
  \x00\x02\x00\xb7\x00:\xff\xff\xff\xff\xff\xff\xff\xff\x00\xff\xff\xff
  \xff\xff\xff\xff\xff\xfe\xff\xff\xff:/usr/local/bin/qemu-aarch64-
  static:
```

Figure I.3: Register file of the `binfmt_misc` service in order to execute `aarch64` binaries with QEMU.

I.3 DNF input arguments

The DNF command has a lot of options for input arguments that provide a lot of flexibility to its use. Table I.1 shows the options that are used in order to generate the CentOS 8 ROOTFS [86].

Table I.1: DNF command input arguments.

Option	Description
<code>-y</code>	Answers yes to all the questions.
<code>--skip-broken</code>	Removes packages that are causing problems.
<code>-c dnf.conf</code>	Location of the configuration file.
<code>--releasever=8</code>	Packages are downloaded for the specific architecture. QEMU is used for different architectures from the base one.
<code>--forcearch=aarch64</code>	
<code>--repo=centos-base, centos-updates,centos-extras, centos-cern,arm64-epel</code>	Enable specific repositories.
<code>--verbose</code>	Show debug messages.
<code>--installroot='/arm/sysroot'</code>	Alternative installroot is specified.

Since the configuration and the input arguments of the DNF command are set, the DNF command runs with the `groupinstall` which allows configuring a group of packages to be downloaded, and `Minimal Install` which downloads the minimum amount of files that are needed to have a basic CentOS 8 Filesystem [86].

J Network Boot

The Network Booting of the board is precondition No.6 in Table 2.2. The network boot is divided into two main parts. The boot images (Kernel Image and Device-Tree) need to be fetched from a TFTP server while the root filesystem of the Kernel is stored in a remote location and is accessed over NFS. U-Boot is responsible to fetch the boot images from the TFTP server while the Kernel mounts the root filesystem over NFS.

J.1 MAC Address

The Zynq UltraScale+ MPSoC requires many network information not only during its boot process but also for its communication over the network. The Zynq UltraScale+ MPSoC will be connected to the private and protected CMS network. In order to communicate with other interfaces, an IP address is required. The IP address is acquired from a central server using the `dhcp` command. In order to acquire a valid IP address, a MAC address that is registered in the CMS network is required. This MAC address is stored in the EEPROM. This MAC address can be used by the Zynq UltraScale+ MPSoC to acquire a valid IP address [103].

The U-Boot and the Linux Kernel will execute their own `dhcp` commands to acquire the IP address. U-Boot is configured by default to use a random MAC address. This configuration can be changed by undefining the `ethaddr` variable in U-Boot and applying the following configuration file (`eeeprom.cfg`) in the build process of U-Boot:

```
1 CONFIG_I2C_EEPROM           = y
2 CONFIG_SYS_I2C_EEPROM_ADDR   = 0x54
3 CONFIG_SYS_I2C_EEPROM_ADDR_OVERFLOW = 0x0
4 CONFIG_ZYNQ_GEM_I2C_MAC_OFFSET = 0x20
```

Figure J.1: Configuration options in U-Boot to retrieve MAC address from EEPROM.

In order to inform U-Boot about the MAC address in the EEPROM, the Device-Tree is configured in order to include this information. The Board Support Package (BSP) that is provided by Xilinx for the Zynq UltraScale+ MPSoC provides this information. The EEPROM can hold other information as well, such as board name or serial number. The node that is added in the device-tree is shown in Figure J.2. The MAC address is stored in the address 0x20 and has a length of 6 bytes.

```
1 &eeeprom {
2     #address-cells = <1>;
3     #size-cells = <1>;
4
5     eth_mac: eth-mac@20 { reg = <0x20 0x6>; };
6 };
```

Figure J.2: EEPROM node in the Device-Tree.

The EEPROM is accessed through an I2C switch. The I2C switch uses a multiplexer to access various components on the I2C bus. In order to drive the pins that are connected to the

I2C bus a pin-controller³² is used. The configuration of the I2C switch and the pin-controller is also added to the device-tree. A *phandle*³³ is also needed in order to integrate the EEPROM node to the root node of the Device-Tree. The link of the EEPROM to the device-tree is shown in Figure J.3 [106].

This information are available in the BSP of the Zynq UltraScale+ MPSoC provided by Xilinx.

J.2 Boot Images retrieval from TFTP server

U-Boot is responsible to retrieve the boot images from the TFTP server. This is done with the `tftpboot` U-Boot command. The `tftpboot` command requires the address of the TFTP server. This address is acquired with a DHCP request. In order to save the IP of the TFTP server, the `CONFIG_SERVERIP` needs to be defined with an empty value and the `CONFIG_BOOTP_SERVERIP` needs to be undefined [107,108].

J.3 Filesystem mounting over NFS

Using the NFS protocol to mount a remote filesystem as the root filesystem of the Zynq UltraScale+ MPSoC allows the system to run without a local disk. It also allows easier access to the boards' filesystems for maintenance and upgrades. The kernel is instructed to mount its root filesystem over NFS with the `root=/dev/nfs` boot argument.

```
1 / {
2   model = "CMS DAQ ZynqMP";
3   compatible = "xlnx,zynqmp";
4
5   chosen {
6     xlnx,eeeprom = &eeeprom;
7     bootargs = "earlycon console=ttyPS0,115200 clk_ignore_unused
8               earlyprintk cpuidle.off=1 root=/dev/nfs ip=dhcp rw";
9   };
10 };
```

Figure J.3: Boot arguments of the Linux Kernel. EEPROM link to the root note of the Device-Tree using a *phandle*.

In order to access the NFS server and mount the root filesystem, its IP address is required. The DHCP server is configured such that it can provide the address of the NFS server as well. The kernel is instructed to make a DHCP request for its IP address by the `ip=dhcp` boot argument.

³²Pin-Controller is a hardware interface that can control the pin multiplexing [104].

³³Numerical identifier for nodes. Used by other nodes in order to refer to the specified node [105].