

MASTER

An Embedded Linux Distribution for the Data Acquisition Hardware of the Compact Muon Solenoid Experiment at CERN

Mor, K.S.

Award date:
2020

[Link to publication](#)

CERN-THESIS-2020-397
27/01/2020



Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

An Embedded Linux Distribution for the Data Acquisition Hardware of the Compact Muon Solenoid Experiment at CERN

Keyshav Suresh Mor (Matriculation Number: 1237978)
Department of Mathematics and Computer Science

A thesis presented for the degree of
Master of Science in Embedded Systems

under the supervision of

Dr. Majid Nabi Najafabadi
Department of Electrical Engineering

and

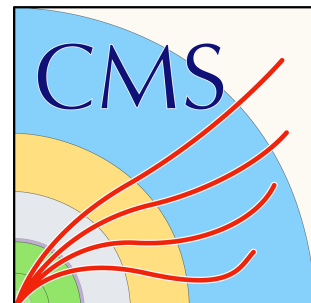
Dr. Frans Meijers
CERN CMS DAQ

at the

Technische Universiteit Eindhoven

in association with

Compact Muon Solenoid (CMS) Experiment at CERN



Place: Eindhoven, The Netherlands
January 27, 2020

Declaration of Originality

I hereby declare that this written work which I am submitting is original work which I alone have authored in my own words. I declare that I have been informed sufficiently regarding the normal academic citation rules and that I have read and understood the plagiarism guidelines provided by the Eindhoven University of Technology “Fraud and Plagiarism Handout”:

https://assets.studiegids.tue.nl/fileadmin/content/Faculteit_IEenIS/Bachelor_afbouw/TIW/fraud_and_plagiarism_v4.pdf

The citation conventions expected in this academic discipline have been respected and I declare that this written work may be tested electronically for plagiarism.

Date

Keyshav Suresh Mor

Acknowledgements

Hereby, I would like to express my sincerest gratitude and appreciation for all those who supported me during my master studies and helped me over the course of my master thesis. I would like to extend my thanks particularly to:

My parents, Rekha and Suresh Mor for encouraging me to pursue graduate studies abroad and supporting me through the course of my master studies.

The European Union and EIT Digital in particular for granting me an opportunity to study MSc Embedded Systems in two prestigious universities, namely the TU Berlin and TU Eindhoven in Germany and The Netherlands respectively.

All the professors, assistant professors, tutors and teaching assistants at the TU Berlin and TU Eindhoven for guiding me in completing the challenging coursework and fueling my curiosity about embedded systems.

My supervisor at TU Eindhoven, Dr. Majid Nabi Najafabadi for guiding me throughout my master thesis and for taking out his time to give invaluable suggestions during our meetings.

My supervisor at CERN, Dr. Frans Meijers for granting me this opportunity to complete my master thesis at the CMS DAQ group, supporting my thesis activities and for insightful questions which guided the work in the right direction.

Dr. Petr Zejdl, for having faith in me and for mentoring me throughout the thesis project and without whose support, expertise, encouragement and guidance this project would not reach a successful conclusion.

Dr. Marc Dobson, for tirelessly answering my questions, helping me understand the networking and system administration principles used in the CMS DAQ network and for his constructive criticism which helped me chart the course of my master thesis.

My former colleague, Awais bin Zahid and the Electronic Systems and Experiments (ESE) group who were kind enough to share their work with me, which formed the basis of my master thesis.

Lastly, the CMS DAQ group for the friendly working environment as well as Dominique Gigi and Dr. Hannes Sakulin for being kind enough to lend me the necessary hardware for completing this project.

Abstract

In 2025, the Large Hadron Collider (LHC) at CERN would be upgraded to be a High-Luminosity Large Hadron Collider (HL-LHC). It would result in increased frequency and intensity of particle collisions at the centre of various experiment detectors located along the circumference of the HL-LHC. This master thesis gives a brief background about the LHC, the CMS experiment, the CMS data acquisition network (CMS-DAQ) and the Phase-II upgrade on the CMS-DAQ network. The upgrade of electronics in the CMS-DAQ would see the introduction of around 1000 Xilinx Zynq UltraScale+ MPSoC (Multi-Processor System-on-Chip) based customised embedded controllers in the CMS data acquisition network.

All these newly introduced MPSoC would require a customised operating system for their hardware platforms. To this end, this master thesis elaborates upon the important Linux boot components and the process of booting Linux on the Xilinx Zynq UltraScale+ MPSoC. This thesis document demonstrates the process of building a customised Linux distribution for the Zynq UltraScale+ MPSoC using PetaLinux Tools. This thesis document presents a qualitative comparison between the PetaLinux Tools and the Yocto Project so that developers can choose a tool-chain better suited to their requirements. To ensure support for the hardware platforms and because of existing expertise at CERN to support the CentOS Linux distribution, this thesis document demonstrates the process of porting a mainstream CentOS 8 kernel 4.18 and the building of CentOS 8 root file system for the Zynq UltraScale+ based ZCU102 evaluation board. To recommend an easy-to-maintain, modifiable and manageable booting mechanism, this thesis document also presents a qualitative and quantitative analysis of different booting configurations and root file system storage mechanisms along with read-write performances of different root file system storage mechanisms (SD Card and NFS server). Based on these findings, this thesis document recommends a particular boot configuration as well as a root file system storage mechanism for the Zynq UltraScale+ based embedded controllers in the CMS data acquisition network. This thesis document demonstrates the process to implement an automated network boot process of embedded Linux on the Xilinx ZCU102 Evaluation board and also explains the ways in which challenges associated with the network boot can be dealt with.

Once these systems are installed in the CMS-DAQ network, it is important to administer software updates or install software on the CentOS 8 root file system of the different Zynq UltraScale+ based hardware platforms that are running the CentOS 8 kernel 4.18. This master thesis demonstrates two different methods of administration of software updates to the CentOS 8 root file system. This thesis document also elaborates upon the qualitative distinctions between these two software update methods and recommends use cases for these two different methods in the CMS data acquisition network.

Contents

Contents	v
List of Figures	x
List of Tables	xii
Listings	xiii
1 Introduction	1
1.1 Problem Statement	1
1.2 Thesis Contributions	2
2 Background	3
2.1 The Large Hadron Collider	3
2.2 The Compact Muon Solenoid Experiment	4
2.2.1 How the CMS Detector Detects Particles	5
2.3 The Trigger and Data Acquisition System of the CMS Experiment	5
2.3.1 The CMS Level 1 Trigger	6
2.3.2 The CMS High Level Trigger	6
2.4 The Current CMS Data Acquisition System	6
2.4.1 Timing and Control Distribution System	7
2.5 The Phase-II DAQ Upgrade	8
2.5.1 The DAQ and TCDS Hub (DTH)	8
3 Embedded Linux for Xilinx Zynq Ultrascale+ MPSoC	9

3.1	The Xilinx Zynq UltraScale+ MPSoC	9
3.2	Xilinx ZCU102 Evaluation Kit	11
3.3	The Need for an Embedded Linux at CMS DAQ	12
3.4	Linux for ARM Processors	12
3.4.1	The Bootloader	13
3.4.2	The Device-Tree	13
3.4.3	The Linux Kernel	15
3.4.4	The “init” Process	15
3.4.5	The Linux Root File System	15
3.5	The Zynq UltraScale+ Linux Boot Process	16
3.5.1	The Boot Setup Stage	17
3.5.2	The Bootloader Stage	17
3.5.3	Boot Flow	18
3.5.4	The Kernel Booting Stage	19
3.5.5	The “init” Stage	19
3.6	The Userspace	20
3.7	Summary	20
4	Building the CentOS 8 Linux Distribution	21
4.1	The Yocto Project	21
4.1.1	The Layer Model	22
4.1.2	Poky	23
4.1.3	BitBake Engine	23
4.1.4	OpenEmbedded Build System	25
4.2	Developing a Distribution with PetaLinux Tools	25
4.2.1	Minimal Vivado Design for Zynq UltraScale+	26
4.2.2	Installing the PetaLinux Tools	26
4.2.3	Creating a PetaLinux Project	27
4.2.4	Configuring the PetaLinux Project	28
4.2.5	Building a PetaLinux Project	31

4.2.6	Packaging the Boot Image	32
4.3	Conclusion of the PetaLinux Build Process	33
4.4	Porting the CentOS 8 Kernel for Zynq UltraScale+	34
4.4.1	Reasons to Port the CentOS 8 Kernel 4.18 for Zynq UltraScale+	35
4.4.2	Fetching and Patching the CentOS 8 Kernel 4.18 Source RPM	35
4.4.3	Getting the Right Kernel Configuration	35
4.4.4	Challenges in Getting the Right Kernel Configuration	36
4.4.5	Importance of Makefiles and Kconfig Files	37
4.4.6	Porting the Drivers from Xilinx Kernel 4.19 to CentOS 8 Kernel 4.18	39
4.4.7	Building the CentOS 8 Kernel 4.18	40
4.4.8	Debugging the Build Process	41
4.5	Lessons from the Porting of the CentOS 8 Kernel 4.18 for Zynq UltraScale+	42
4.5.1	Lessons from the Kernel Source Extraction and Patching the Source Code	42
4.5.2	Lessons for Setting up the Build Tool-Chain	42
4.5.3	Lessons from Kernel Configuration	43
4.5.4	Lessons from Driver Porting and the Build Process	43
4.6	Building a CentOS 8 Root File System	44
4.6.1	Reason for a CentOS 8 Root File System	44
4.6.2	Procedure of Building a CentOS 8 Root File System	44
4.6.3	The Need for QEMU Emulator	46
4.7	Differences between PetaLinux Tools and Yocto Project	46
4.8	Summary	47
5	The Embedded Linux Network Boot	48
5.1	Important Stages of the Embedded Linux Network Boot	48
5.2	Acquisition of Network Information from the DHCP Server	48
5.3	Download Boot Files from the TFTP server	50
5.3.1	Fixing the U-Boot Processing of DHCP Response	50
5.4	Importance of the Ethernet MAC Address	51

5.4.1	Reason to Study the Setting, Modification and Acquisition of Ethernet MAC Address during Linux Boot on Zynq UltraScale+	52
5.4.2	Default MAC Address Configuration for Zynq UltraScale+	52
5.5	Setting of the MAC address in the Ethernet Hardware	52
5.6	The U-Boot Environment “uEnv.txt” File	53
5.6.1	The Variables in the “uEnv.txt” File	55
5.6.2	Using the uEnv.txt File	58
5.7	MAC Address Setting in the Device-Tree by U-Boot	58
5.8	Acquisition of the MAC address by the Linux Kernel	59
5.9	Mounting the Root File System from the NFS server	60
5.9.1	Network File System Server	60
5.10	Overview of the Automated Linux Network Boot	61
5.11	Summary	63
6	Performance of Boot Methods and Root File System Locations	64
6.1	Linux Boot Time Tests	64
6.1.1	Test Infrastructure and Scripts	64
6.1.2	Boot Files Read Speeds	65
6.1.3	Average Boot Times	66
6.2	Performance of Root File System Types	70
6.2.1	Average Read Speeds from Root File System	71
6.3	Average Write Speeds to Root File System	73
6.3.1	Comparing SD Cards and Network Storage	74
6.4	Summary	76
7	Installing and Administering Software Updates	78
7.1	Administering Software Updates	78
7.1.1	DNF Package Manager	79
7.1.2	How Installation Works with DNF	79
7.1.3	NFS Server Side Software Updates	80
7.1.4	Zynq UltraScale+ Client Side Software Updates	81

7.1.5	Features of NFS Server Side Software Updates	81
7.1.6	Features of Client Side Software Updates	81
7.1.7	Comparison of the Two Software Update Methods	82
7.2	Summary	83
8	Conclusion	84
8.1	Summary	84
8.2	Future Work	85
8.2.1	Implementation of Reliable, Fault-Tolerant Linux Boot	85
8.2.2	Development of an Overlay File System	86
	Bibliography	88
	Appendices	91
	Appendix A	92
	Appendix B	99
	Appendix C	110
	Appendix D	112

List of Figures

2.1	The LHC accelerator complex [1]	4
2.2	The CMS data acquisition system for run-2 [2]	7
3.1	Block diagram of Xilinx Zynq UltraScale+ MPSoC	9
3.2	The Xilinx ZCU102 board with the TTL NUC5 desktop.	11
3.3	The Xilinx Zynq UltraScale+ MPSoC boot flow [3]	18
4.1	The Menuconfig home screen for configuring the PetaLinux project.	28
4.2	Menuconfig to activate automatic IP address assignment.	30
4.3	Menuconfig to specify root file system type.	30
4.4	Difference in Kconfig.platforms files of Xilinx kernel 4.19 and CentOS 8 kernel 4.18 source code before changes.	37
4.5	Missing Xilinx and Zynq specific drivers in CentOS 8 kernel 4.18.	39
4.6	PetaLinux Menuconfig to specify source of the U-Boot and the kernel	41
5.1	DHCP client-server interaction [4]	49
5.2	DHCP request by the U-Boot	61
5.3	Loading of kernel Image and device-tree from the TFTP server	62
5.4	Mounting the CentOS 8 root file system from the NFS server	63
5.5	User login into the CentOS 8 root file system	63
6.1	Boot File Read Speeds	66
6.2	Average Linux boot times for the three boot methods	68
6.3	Average Linux boot time breakup for the three different boot methods	69
6.4	Average read speeds from root file system (writes to RAM disk)	71

6.5	Average write speeds to root file system (reads from RAM disk)	74
B.1	Process to set MAC address in the Ethernet Hardware	106
B.2	Modification of MAC address in the device-tree by the U-Boot	107
B.3	Acquisition of MAC address by the Linux Kernel	108
D.1	Poster presented at TWEPP 2019	118
D.2	Poster presented at CHEP 2019	119

List of Tables

6.1	Statistics of 10 samples for boot file read speeds from the SD card and the boot network.	65
6.2	Statistics of 10 samples for boot timings for the full SD card boot.	66
6.3	Statistics of 10 samples for boot timings of the partial network boot.	67
6.4	Statistics of 10 samples for boot timings for the full network boot.	67
6.5	Statistics of 10 samples for read speed tests from the root file system storage types	72
6.6	Statistics of 10 samples for write speed tests to the root file system storage types .	73

Listings

3.1	A snapshot of the ZCU102 device-tree showing the contents of the I2C EEPROM chip.	13
4.1	Contents of bblayers.conf configuration file in the Yocto source directory	22
4.2	Contents of xilinx-platform-init.bbclass class file in the Yocto source directory. . .	23
4.3	Contents of u-boot-xlnx-2019.1.bb BitBake recipe file in the Yocto source directory.	24
4.4	Script for installing the PetaLinux Tools	27
4.5	Script for creating a PetaLinux Project	27
4.6	Script to configure PetaLinux project	29
4.7	Script to build and package PetaLinux images	33
4.8	Script to add Zynq UltraScale+ features to CentOS 8 kernel 4.18 default configuration	35
4.9	Makefile to build Zynq UltraScale+ firmware.	38
4.10	Kconfig file to specify Zynq UltraScale+ firmware options.	38
4.11	Command to build the CentOS 8 kernel 4.18.	41
4.12	Contents of qemu-aarch64.conf in the /etc/binfmt.d path of CentOS 7 root file system	45
5.1	The uEnv.txt file on the TFTP server	55
7.1	Configuration for DNF Package manager	79
7.2	Command for installing server side software updates	80
A.1	Script to extract, patch, configure and build a CentOS 8 kernel 4.18	92
A.2	Configuration of repositories in dnf.conf for sourcing CentOS 8 packages for the CentOS 8 root file system build script	93
A.3	Script to build the CentOS 8 root file system with the help of repositories pointed by the dnf.conf file	94
A.4	Snapshot of differences between CentOS 8 kernel 4.18 defconfig and xilinx_zynqmp_defconfig	96
B.1	DHCP server configuration in the file dhcpd.conf in the path /etc/dhcp	99

B.2	TFTP server configuration in the file tftp in the path /etc/xinetd.d	100
B.3	TFTP service configuration in the file tftp.service in the path /usr/lib/systemd/system	100
B.4	Script to setup the TFTP server	100
B.5	Snapshot of u-boot_bsp.tcl script used by PetaLinux Tools to configure the U-Boot	101
B.6	Script to setup the NFS server	102
B.7	Contents of exports file exporting the NFS file system	103
B.8	The U-Boot environment variables after loading the uEnv.txt from the TFTP server(see Listing B.1 for information offered by DHCP server)	103
C.1	Script used to do read-write tests on the root file system kept on SD card as well as NFS server	110
D.1	Output on the terminal while updating the CentOS 8 root file system from the NFS server side	112
D.2	CentOS base repository configuration for DNF package manager in CentOS 8 root file system	115
D.3	Terminal output for client side software update	116

Chapter 1

Introduction

The Large Hadron Collider (LHC) at CERN would be upgraded to be a High-Luminosity LHC (HL-LHC) in 2025 in the Phase-II upgrade. The HL-LHC would increase the frequency and intensity of the particle collisions that occur at the center of the detectors of experiments like CMS located along the LHC [5]. This upgrade would also entail the upgrade of the detector front-end sensors, the read-out electronics and the data acquisition hardware that contribute to the data-taking of the experiment. It is called the Phase-II upgrade and it would see the introduction of embedded controllers using Xilinx Zynq UltraScale + MPSoC in the CMS data acquisition network (CMS-DAQ). These embedded controllers would require their own operating system with their own root file system, software packages, kernel drivers and services which would allow them to do the computing tasks that they are designed for.

1.1 Problem Statement

Multiple experiment groups working on their own hardware platforms can build their own Linux distributions, however the experiment groups do not have the expertise. The CERN system administrators do not have enough manpower and resources to manage multiple, distinct Linux distributions whose kernels may also have security vulnerabilities. The CentOS Linux distribution is centrally supported and maintained by the system administrators at CMS and CERN IT, who provide security-tested kernel drivers and software packages for it. Previous work on building a CentOS 7 Linux distribution using the CentOS 7 root file system and the Xilinx Linux kernel had happened at the ATLAS experiment and the Electronic Systems and Experiments (ESE) group at CERN. However, there was no CentOS Linux distribution for 64-bit ARM processors like Zynq UltraScale+ MPSoC which would provide a mainstream CentOS 8 kernel (4.18) and a CentOS 8 root file system. This master thesis seeks to address this void arising out of the needs of the CMS experiment and demonstrates the porting of the CentOS kernel 4.18 for Zynq UltraScale+ and a method to build a CentOS 8 distribution for the Zynq UltraScale+ MPSoC.

There has been previous work at the CMS-DAQ group to build a Linux distribution for the 32-bit Xilinx Zynq-7000 SoC with the help of the Yocto Project. However, the work did not cover the automated Linux network boot, which is crucial for the CMS data acquisition network. This master thesis demonstrates how to build a Linux distribution using PetaLinux Tools that boots over the network without manual intervention and presents the components necessary for configuring and implementing an automated network boot of the CentOS 8 Linux distribution.

There are multiple ways to boot Linux on the Zynq UltraScale+ MPSoC and multiple methods to store the CentOS 8 root file system. Previous work at the CMS DAQ group has focused on a full SD card boot (see Chapter 6) and has not conducted studies to recommend a particular boot method and root file system storage mechanism for the data acquisition hardware in the CMS experiment network. This master thesis addresses this void by doing qualitative and quantitative analysis of different boot methods and root file system storage mechanisms and recommends one solution for each of those requirements.

The previous work at the CMS-DAQ group also did not cover the administration of software updates to the root file system of the Linux distribution. This master thesis demonstrates two methods of administering software updates to the 64-bit ARM version of CentOS 8 root file system and presents qualitative differences between the two methods to recommend uses for these two methods in the CMS data acquisition network.

1.2 Thesis Contributions

The thesis aims at making the following contributions:

1. Demonstrate the process of building a Linux distribution with the help of PetaLinux Tools. This thesis also offers a qualitative comparison of the PetaLinux Tools and Yocto Project to help readers make an informed choice of the tool chain they need to build a Linux distribution.
2. Demonstrate porting of a CentOS 8 Linux kernel 4.18 for the Xilinx Zynq UltraScale+ MPSoC as well as building of CentOS 8 root file system for 64-bit ARM processors.
3. Demonstrate the implementation of CentOS 8 automated network boot on the ZCU102 board, the setup of the network boot infrastructure and the solving of the challenges associated with the network boot. The thesis also investigates and answers questions pertaining to the setting, acquiring and updating of the Ethernet MAC address by the U-Boot and the Linux kernel during the Linux boot process on Zynq UltraScale+.
4. Present a quantitative and qualitative analysis of different boot methods and the root file system storage methods to recommend a particular solution for both aspects in the context of the CMS data acquisition network.
5. Demonstrate different methods of administering software updates to the CentOS 8 root file system and compare the methods to recommend use cases for these two software update methods in the CMS data acquisition network.

The remainder of this thesis is organized in the following chapters. Chapter 2 informs readers about the background of the LHC, the CMS experiment, the CMS-DAQ and the Phase-II upgrade. In Chapter 3, information about the Xilinx Zynq UltraScale+ MPSoC and the ZCU102 Evaluation board is provided along with the Linux boot elements and the Linux boot process on the Xilinx Zynq UltraScale+. Chapter 4 covers the Yocto Project, the process of building a Linux distribution using PetaLinux Tools, the process of porting and building the CentOS 8 kernel 4.18 as well as the process to build the CentOS 8 root file system. Chapter 5 explains the Linux network boot in detail whereas Chapter 6 consists of the qualitative and quantitative comparisons of the different boot methods and root file system storage mechanisms. Chapter 7 talks about the methods to administer software updates to the CentOS 8 root file system and Chapter 8 concludes this master thesis.

Chapter 2

Background

This master thesis project aims at developing an Embedded Linux distribution for the embedded systems that would be installed in the data acquisition network of the Compact Muon Solenoid (CMS) experiment as a part of the Phase-II upgrade of the CMS experiment. In order to understand this document better, this chapter intends to give the reader a brief background about the Large Hadron Collider (LHC), the CMS experiment, the data acquisition network and the Phase-II upgrade.

2.1 The Large Hadron Collider

Particle accelerator experiments are developed for accelerating atomic and sub-atomic particles to very high kinetic energies before colliding them with each other to analyse the byproducts of these collisions and answer the fundamental questions about physical matter. The Large Hadron Collider (LHC) [6] is one such particle accelerator. It was commissioned in 2008 and since then the discovery of Higgs-Boson has been achieved using the LHC. A series of pre-accelerators are used to accelerate the particles (protons) close to the speed of light [7] before they enter the LHC. The LHC has a circumference of 27 km [8] which is divided into eight sections, each consisting of curved and straight subsections. The curved sections bend the particle beam on its trajectory which is achieved by keeping magnetic dipoles at superconducting temperatures of 1.9 K [8]. The particles collide at the centre of the CMS detector at a rate of 40 MHz [9]. Detectors like that of the CMS are installed around the crossing points of the LHC beams to detect collisions, record data, track the collision byproducts and to reconstruct the collision to conduct physics research. ALICE, ATLAS, CMS and LHCb are some main detector experiments associated with LHC. Since the focus of this master thesis is on development of an embedded Linux for the data acquisition hardware of the CMS data acquisition system, the CMS experiment is explained in more detail in a later section.

The energy of the particles and the number of particle collisions per cross-sectional area which is known as instantaneous luminosity [9], are increased regularly during the lifetime of the LHC and would continue to increase till 2038. The Phase-II upgrade explained in later sections is a result of one such exercise which would be completed in 2025. Higher collision energies and higher luminosity are necessary to increase the probability of discovering new properties related to the fundamentals of physical matter and are necessary to confirm various other theories from the Standard Model of Physics. The Phase-II upgrade [8] would result in an upgrade of the CMS detector as well as the electronics in its data-acquisition network [5]. The LHC will transition into

its High Luminosity phase known as the High Luminosity LHC (HL-LHC) in 2025. [8].

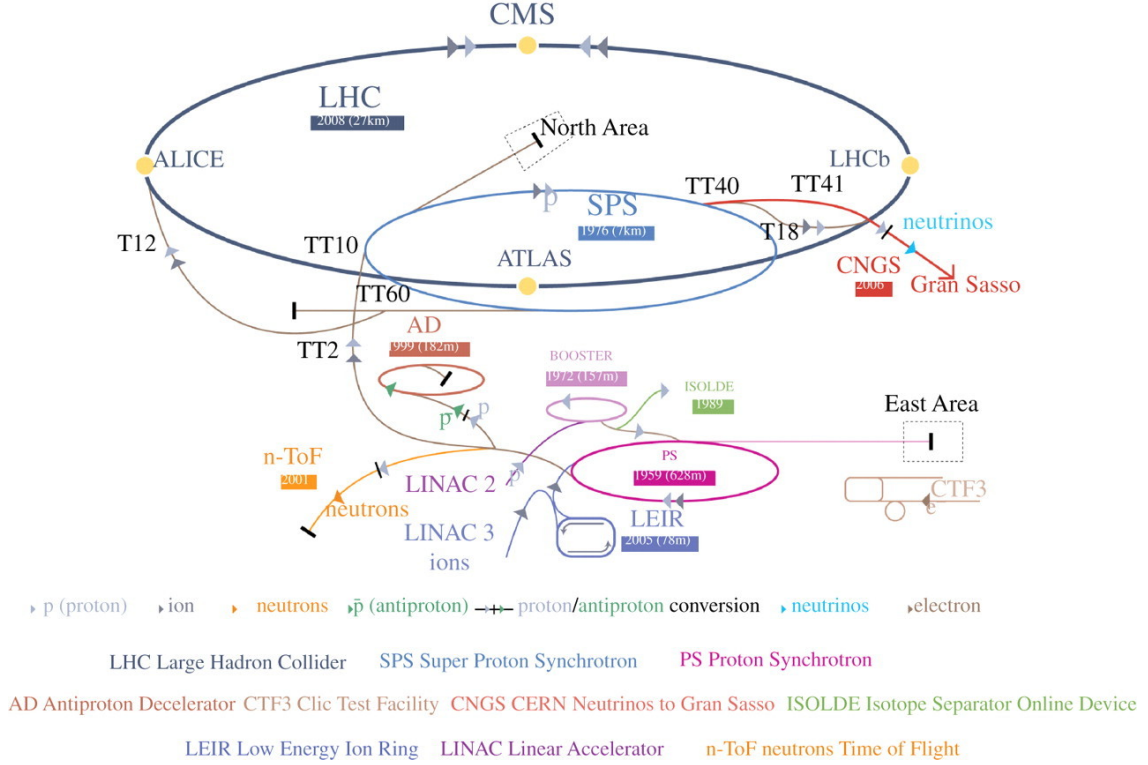


Figure 2.1: The LHC accelerator complex [1]

Figure 2.1 shows the LHC accelerator complex. The big dark-blue coloured ring at the centre is the main accelerator with different experiment detectors located along the ring, denoted by yellow dots. The CMS experiment is located at Point 5 of the LHC, seen at the top end of the blue ring. In Figure 2.1, below the big ring, a host of small pre-accelerators like Proton Synchrotron, Super Proton Synchrotron and Linear Accelerator (LINAC) are shown. They are used to accelerate protons before they enter the LHC. The particles travel in opposite directions resulting in collisions at the centre of experiment detectors. The arrows in the figure are serving two purposes: they show the motion of the particles within the accelerator as well as they are colour coded to denote different particles. The light grey is for the proton, orange for neutrons, blue for neutrinos and brown for electrons.

2.2 The Compact Muon Solenoid Experiment

The CMS (Compact Muon Solenoid) experiment [10][11][12] at CERN has the following goals:

- To detect muons, electrons and photons to study their properties through high precision measurements.
- To take high precision measurements of different daughter particles being generated post collision at the centre of the detector.
- To reconstruct the particle collisions to study the fundamental properties of particle physics and properties of various daughter particles after collision.

The CMS detector consists of a superconducting solenoid that provides an axial and uniform magnetic field of 3.8 Tesla [10]. The solenoid holds the Hadron Calorimeter (HCAL) [10] which in turn holds the Electromagnetic Calorimeter (ECAL) within itself. A silicon pixel Inner Tracker (IT) and a silicon microstrip Outer Tracker (OT) are situated within the ECAL [10]. The forward calorimeters ensure the angular coverage of the magnetic field. The solenoid's magnetic yoke made of iron is scattered with gaseous chambers to detect muons.

2.2.1 How the CMS Detector Detects Particles

After the collision of particles at the centre of the CMS detector, the particles travel outwards where they encounter the tracking system [10], which consists of the silicon pixel Inner Tracker (IT) and silicon microstrip Outer Tracker (OT) as mentioned above. These are used to measure the positions of particles passing through them and help in reconstructing their tracks. The principle that is applied here is quite simple: charged particles will always follow curved paths in the magnetic field and the curvature of their tracks will allow calculation of their individual momentum. The CMS tracker system measures trajectories of the charged particles and helps in generating three-dimensional reconstruction of their tracks.

The calorimeters are used to measure the energies of the particles that pass through the tracker. Electrons and photons interact with the electromagnetic force, as a result their energies are measured using the Electromagnetic Calorimeter (ECAL) [10]. Protons and Neutrons are types of hadrons and the Hadronic Calorimeter (HCAL) is used to measure the energies of these hadrons [10]. Due to the higher particle presence in the forward regions of the CMS detector, the HCAL Forward Calorimeter (HF) is constructed with radiation hardened components in order to prevent damage or malfunctioning due to heavy ionizing radiation in this region [10].

Since muons are heavier than the electrons, they have a higher momentum which allows them to punch through the HCAL along with the neutrinos, which do not interact much with their surroundings. The gaseous ionisation chambers are used to detect the muons and their momentum is measured from the curvature in their paths due to the CMS magnetic field. Neutrinos rarely interact with their surroundings and it is difficult to detect them.

2.3 The Trigger and Data Acquisition System of the CMS Experiment

The CMS experiment has a two-level trigger system [5][13][14] consisting of the Level 1 Trigger and the High Level Trigger (HLT). The trigger system of the CMS detector acts as a filtration and selection system which ensures that only the data of certain interesting events after the particle collision are selected. This is done for the following reasons:

- A small minority of the events after the particle collision are of interest to the physicists to advance their research.
- These processes occur at a very small rate.
- There are constraints associated with the data acquisition network's bandwidth and storage space to store the experiment data. As a result, the data of all of the events occurring at the centre of the CMS detector cannot be transported and stored for physics research.
- Due to the above three reasons, the trigger system is used select and filter the event data which the data acquisition network transfers for further analysis and storage.

2.3.1 The CMS Level 1 Trigger

The CMS Level 1 Trigger which is made of custom electronics is responsible for performing online selection of physical processes occurring at the center of the CMS detector. Level 1 Trigger in its current form reduces the event rate from 40 MHz to 100 kHz [13] and has a latency of less than 4 micro-seconds [14].

The L1 Trigger receives the information coming from the electromagnetic and hadronic calorimeters as well as from the muon chambers. A global accept or reject decision is taken on the basis of energy concentrations associated with particles such as photons, electrons, muons and jets [13][14]. The CMS Level 1 trigger will be upgraded further as a part of the Phase-II upgrade to improve the performance when the particle collision intensity increases [5] at the center of the CMS detector in HL-LHC.

2.3.2 The CMS High Level Trigger

The High Level Trigger (HLT) is essentially a processor farm of commercial computers using over 26,000 CPU cores [12] and doing the same filtration and selection process as L1 Trigger. The HLT is running software algorithms for down sampling the event rate from 100 kHz to about 1 kHz [5][14]. Unlike the L1 Trigger, the HLT is asynchronous with respect to the collisions, receiving the data after a significant delay through the data acquisition pipeline system, with collisions being processed potentially out of order. The average processing time of the HLT in 2012 was about 200 milliseconds per event [14]. For the Phase-2 upgrade, however, this time is estimated to be of the order of one second but a latency of 2 minutes is allowed [5]. The HLT receives complete detector information about L1 selected collisions, after being passed through the Data Acquisition event building system comprised of the following modules:

- The Readout Units (RU) which collect data from individual Front-End Detectors (FED) and partially build/assemble the data from a single collision.
- The Builder Units (BU) which collect data from all RUs and finish to build the complete detector data for each collision before feeding them to the HLT.
- The RUs and BUs are on high performance data networks for the transfer of the event data (FED builder and Event builder networks). The HLT is made up of Filter Units (FUs) which are its processing and selection elements (individual compute elements).

The Worldwide LHC Computing Grid (WLCG) is a distributed cloud storage and processing infrastructure that performs offline processing of events that pass the HLT selection process [10].

2.4 The Current CMS Data Acquisition System

The main purpose of the CMS Data Acquisition (CMS-DAQ) system is to provide a data pipeline and decoupling between the L1 trigger and the HLT. The DAQ system [4][15][16][17][18][19] acts as a bridge between the L1 Trigger and the High-Level Trigger, from the moment the L1 trigger selects the data of a particular interesting event to be sent to the High-Level Trigger for further selection. The DAQ-HLT interface exchanges data in the form of files, which helps in decoupling the DAQ pipeline and HLT farm. For data concentration, 10/40 Gigabit Ethernet connection is used whereas a 56 Gigabit Infiniband network is used for the event-builders [18]. The CMS DAQ

for the Run 2 between 2015-2018 was designed to assemble event data at a rate of 100 kHz with event size of 2 MB [5] and a readout bandwidth of 200 GB/s [5][16].

The DAQ for the CMS Run-2 [4][15][16][17][18][19] consists of high-performance computing nodes which are configured to accept data for different scenarios. The trigger data is received by a separate section of back-end electronics for distribution to the Level 1 trigger processors, which essentially instruct the front-end read out units to start reading data when an interesting event has been detected by front-end electronics. The DAQ takes in data from more than 600 customised Front-End Detectors (FEDs) and these FEDs send data fragments of size 12 kB to the Front-End Readout Links (FEROL) [18]. The FEROL sends the data from each FED as TCP/IP packets over optical 10 Gigabit Ethernet links to one of the 72 readout units (RU) computers. A series of network switches is used to concentrate the data from the FEROLs into super-fragments using a 10/40 Gigabit Ethernet network. The super-fragments are then sent over the event-builder switch to one of the 62 builder units (BU) machines which assemble the super-fragments into complete event data. Each builder unit (BU) has a 250 GB RAM disk and a 2 TB magnetic disk. Finally, the BU writes the event data to files residing on a local RAM disk which is then exported to filter units (FUs) which run the high-level trigger code for further selection.

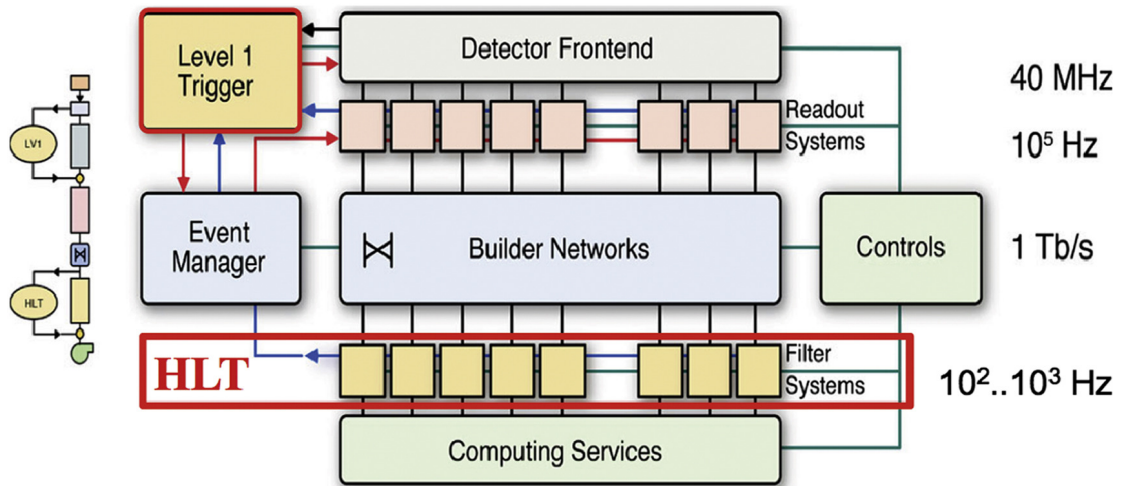


Figure 2.2: The CMS data acquisition system for run-2 [2]

Figure 2.2 displays the block diagram of the CMS data acquisition system for Run-2. It also displays the conceptual diagram of the interaction between the L1 and High-Level Triggers with the data acquisition pipeline. The detector front-ends detect the interesting events after particle collision and the data is then kept in front-end buffers till an accept signal is received from the L1 Trigger. Then the data passes through the data acquisition pipeline over to reach the HLT where further selection takes place. On the right side of the figure, we can see how the L1 and HLT triggers help in bringing down the event data rate from 40 MHz to 100 kHz.

2.4.1 Timing and Control Distribution System

The CMS Run-2 Trigger Control and Distribution System (TCDS) [5] is a culmination of three systems that were used in Run-1. They were:

- The Trigger Control System (TCS)
- The Trigger Timing and Control (TTC) system

- The Trigger Throttling System (TTS)

The TCDS is responsible for managing the data taking operations by the data acquisition pipeline based on the trigger conditions and the data acquisition system's readiness. It acts as a mediator between the trigger and the data acquisition system readiness to accept more data. The TCDS controls distribution of L1 Trigger accepts (TCS) and timing signals to the back-end and front-end electronics of all CMS sub-detectors and collects the information about the readiness of all sub-detectors (TTS) in the CMS main detector. These two tasks are useful in generating a signal that stops triggers when readout buffers are full or out of sync with the event. The clock that is distributed as reference is synchronous with the particles in the LHC and this clock helps in keeping the data-taking synchronous across all the sub-detectors of the CMS main detector.

2.5 The Phase-II DAQ Upgrade

In 2025, the upgrade of LHC (Large Hadron Collider) to HL-LHC (High Luminosity LHC) [20], would result in increased particle collisions at the centre of the detectors situated along the LHC. This increase in particle collision would present an opportunity to detect and measure more interesting events and also help in increasing the precision with which the data is captured. To ensure that this opportunity is completely utilised, the CMS detector along with its detector front-end electronics, calorimeters, sub-detectors and data acquisition system (DAQ) would be upgraded. This upgrade of the DAQ is referred to as the "DAQ Phase-II Upgrade" [5]. Once the Phase-II upgrade is complete, the detector will be designed to read out data at a rate of up to 44 Tb/s whereas the event sampling rate would increase to up to 750 kHz [5]. This sampling rate would be further reduced by HLT to 7.5 kHz [5] for offline processing and analysis. For CMS Run-2, the parameters were 2 Tb/s data rate, 100 kHz event rate, and an event storage rate of 1 kHz. The Phase-II upgrade of the DAQ would see an event size of up to 7.4 MB and an event network throughput of 44 Tb/s (up from 2 Tb/s during Run-2) [5].

2.5.1 The DAQ and TCDS Hub (DTH)

The DAQ Phase-II upgrade would see the introduction of DAQ and TCDS Hub (DTH) [5] in the CMS data acquisition network which will integrate the TCDS with the data acquisition systems read-out functionality. The DAQ and TCDS Hub (DTH) will combine the following functionality:

- Data read-out from front-end electronics
- Data aggregation for efficient usage of data acquisition network's bandwidth.
- Translation of data to packets of a standard protocol like TCP/IP.
- Distribution of Timing and Control Signals to each individual back-end board in the crate.
- Collection and pre-processing of the individual board status for faster monitoring and statistic collection.

The DTH would aggregate input data from each individual back-end board and provide output links to the surface. The timing and control signals would be received by the DTH from the TCDS over an optical link and distributed to all node slots in the crate holding the data acquisition hardware. The nodes would send their TTS status and monitoring data to the DTH through the backplane of the crate. This would help the DTH generate trigger controlling and throttling signals as and when required.

Chapter 3

Embedded Linux for Xilinx Zynq UltraScale+ MPSoC

The Phase-II upgrade of the Compact Muon Solenoid (CMS) experiment at CERN would see the introduction of the Xilinx Zynq UltraScale+ MPSoC (Multi-Processor System-on-Chip) as an embedded controller for the new data acquisition hardware. They would be used for read-out monitoring and control of the data acquisition hardware in the CMS data acquisition network. Many sub-groups associated with the CMS experiment would design their customized hardware platforms using the Zynq UltraScale+ MPSoC. These devices tightly integrate the Programmable Logic (PL) with the ARM core Processing System (PS) which allows them to run even a desktop-grade Linux distribution on the Zynq UltraScale+. This chapter aims at introducing the Xilinx Zynq UltraScale+ MPSoC and explaining the embedded Linux boot on Zynq UltraScale+ in detail.

3.1 The Xilinx Zynq UltraScale+ MPSoC

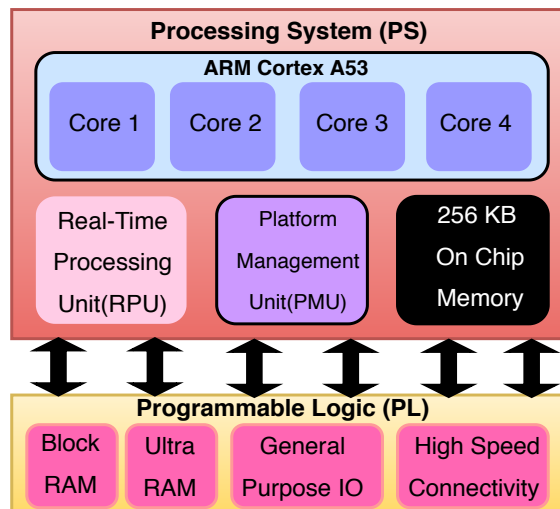


Figure 3.1: Block diagram of Xilinx Zynq UltraScale+ MPSoC

Figure 3.1 shows the block diagram of the Zynq UltraScale+ MPSoC. The Zynq UltraScale+ MPSoC has multiple processing units like the ARM Cortex A53 Application Processing Unit (APU) with 4 cores, the Real-Time Processing Unit (RPU) and the Platform Management Unit (PMU). The figure shows the interfacing between the processing system (PS) and programmable logic (PL). The PL has multiple blocks like GPIO, Block RAM and high connectivity block for implementing designs to communicate with peripherals like Ethernet and PCI Express.

The Xilinx Zynq UltraScale+ MPSoC (Multi-Processor System-on-Chip) is an advanced System-on-Chip. It includes ARM Cortex-A53 Application Processing Unit (APU), an ARM Mali-400 based Graphics Processing Unit (GPU) and an ARM Cortex-R5 dual-core Real-Time Processing Unit (RPU) [21]. In this thesis, our work deals with the Application Processing Unit (APU), which would be hosting the Embedded Linux distribution. The ARM Cortex-A53 APU has the following features [21]:

- Quad-Core processing system
- CPU Frequency: 1.5 GHz
- Support for 32/64-bit operating modes
- CPU Memory: 4 GB
- 32 KB Level-1 cache
- 1 MB Level-2 cache

In addition to the above features of the APU, the Zynq UltraScale+ Processing System (PS) also has other important features such as [21]:

- A Configuration and Security Unit (CSU)
- A Platform Management Unit (PMU)
- 256 KB On-Chip Memory (OCM)
- Gigabit Ethernet Support
- USB 3.0 support
- I2C/SPI peripheral support
- High-Speed UART support up to 1 Mb/s
- Support for Ultra-High Speed Mode of SD cards

The programmable logic (PL) of the Xilinx Zynq UltraScale+ MPSoC is the UltraScale+ FPGA series produced by Xilinx. It has the following features [21]:

- Configurable Logic Blocks
- 36 KB Block RAM
- Support for PCI Express
- Support for up to 100 Gigabit Ethernet

Due to the features mentioned above and many other such features, the Xilinx Zynq UltraScale+ would be used by many experiment sub-groups in the CMS DAQ Phase-II upgrade. More can be read about the Xilinx Zynq UltraScale+ MPSoC by reading the device data-sheet [21].

3.2 Xilinx ZCU102 Evaluation Kit



Figure 3.2: The Xilinx ZCU102 board with the TTL NUC5 desktop.

Figure 3.2 shows the development and testing setup with the Xilinx ZCU102 Evaluation Kit and the TTL NUC5 desktop. The ZCU102 hosts the Zynq UltraScale+ MPSoC at the centre (under the black cooling fan) and has all the peripherals on the board that are supported by the chipset. This helps developers test out various hardware-software designs before initiating final production. The TTL NUC5 desktop is used to communicate with the ZCU102 over UART, for development of the Linux distribution and also for testing. More has been elaborated upon the testing infrastructure in Chapter 6, section 6.1.1.

For this master thesis, the Xilinx ZCU102 Evaluation Kit has been used for development and testing of the Embedded Linux distribution on Xilinx Zynq UltraScale+ MPSoC. The evaluation kit utilizes the Zynq UltraScale+ MPSoC Processing System (PS) for processing and Programmable Logic (PL) for implementation of the programmable logic on the UltraScale+ FPGA. The Xilinx ZCU102 Evaluation Kit has the following features [22]:

- SD card interface
- USB-UART interface
- PCI Express root port slot
- HDMI Output/Input ports
- Gigabit Ethernet port 1 Gigabit connections

More can be read about the Xilinx ZCU102 Evaluation Kit by reading the device data-sheet referred to in the bibliography [22].

3.3 The Need for an Embedded Linux at CMS DAQ

An embedded system is a computing system that is installed or “embedded” in a machine or a network to control a function or a range of functions. Usually such embedded systems are installed to monitor critical processes such as the engine combustion cycle as a part of the Engine Control Unit (ECU) in a car or as a part of the radiation monitoring system at a Nuclear Power plant. These embedded systems are installed at locations where regular sized desktops and computers cannot be installed and unlike general-purpose computers, they have constraints in terms of their size, memory, power consumption and processing speed.

The customised hardware designed by the experiment sub-groups at CMS using the Xilinx Zynq UltraScale+ are also such embedded systems which would perform critical control and monitoring tasks of the data acquisition hardware in the CMS experiment. The hardware designed by the sub-groups would have to be compatible with the ATCA (Advanced Telecommunication Computing Architecture) crates and would have to comply with the dimensions, power and the networking standards being adopted by the CMS experiment group members. The hardware would come from 15-20 different sub-groups and the hardware of each group could have unique functionalities, unique hardware peripherals, unique software applications and unique driver requirements.

The Xilinx Zynq UltraScale+ MPSoC has a powerful processing system and with the help of its APU, GPU and support for various peripherals such as SATA (Serial AT Attachment), Gigabit Ethernet and HDMI (High-Definition Media Interface) Video output, it can be used as a platform to run a desktop grade Linux distribution like Ubuntu or CentOS. Even though the experiment groups at CMS would not use the hardware as a desktop, the features of Xilinx Zynq UltraScale+ MPSoC ensure that a 64-bit ARM Linux kernel with Long-Term Support (LTS) can be deployed on the hardware to run a distribution like CentOS. The Zynq UltraScale+ MPSoC presents the hardware developers an opportunity to design more compact hardware platforms whereas the Linux distribution for the Zynq UltraScale+ MPSoC allows experiment groups to directly communicate with their hardware and remotely update, configure and manage the system.

The Linux distributions compatible with the 64-bit ARM processors are not vastly different from their Intel x86 counterparts except that the kernel has to be ported for the 64-bit ARM processor and it should support the userspace by providing drivers and services for the hardware being driven by the ARM processor. Additionally, the root file system must contain software packages, libraries and files that are executable on the 64-bit ARM processor. The booting of the Linux kernel on the ARM processors also varies slightly compared to the booting of the kernel on the Intel x86 processors, which will be explained in later sections.

3.4 Linux for ARM Processors

A Linux based Operating System is offered as a Linux distribution to the users and it is a collection of software packages, libraries and utilities along with its own Linux kernel variant which is adopted or modified for a given processor architecture. Linux distributions are sometimes developed for specific use cases. The RedHat Enterprise Linux from which CentOS has been derived is a Linux distribution for enterprises and large scale server deployment.

In order to boot a Linux operating system on the ARM processor, the following components are essential:

- A Bootloader
- A Device-tree
- A Linux kernel
- An “init” process
- A root file system

The bootloader is responsible for initializing the underlying hardware, memories, registers and clocks before loading the Linux kernel in the processor memory. The Linux kernel is informed about the underlying hardware through the device-tree. The kernel activates various services and processes for which the kernel is configured and mounts the root file system. The root file system contains all crucial files necessary for the operating system operations and installed user applications.

3.4.1 The Bootloader

The bootloader [23] is a software which is executed when the system is powered on and it is responsible for loading an operating system for the hardware. The bootloader is responsible for loading the kernel, the hardware information, initialising the hardware peripherals and optionally loading the bitstream in the FPGA if the Zynq UltraScale+ PL is being used [3]. The principles behind the Linux boot process on any processor remain the same even if the underlying hardware may differ. For the Zynq UltraScale+ MPSoC, the bootloader should be present on the first partition of the booting medium, in our case SD Card and it should be a FAT16/FAT32 partition [24]. There can be a single bootloader stage or it can be divided in multiple stages [25][23][26].

The bootloader has to be loaded from disk into processor memory before its execution. U-Boot is used to load the Linux kernel on Zynq UltraScale+ [3]. For the Zynq UltraScale+, the bootloading stage is divided in two stages: the FSBL and the U-Boot stage [24][3].

The First Stage Bootloader (FSBL) can be imagined as a software piece similar to BIOS (Basic Input/Output System). Both the FSBL and BIOS are responsible for initialising and testing the system hardware before loading the second stage bootloader (in our case U-Boot for Zynq UltraScale+) from a storage device.

The U-Boot is similar to GRUB found on x86 computers. The U-Boot is responsible for loading the kernel into the processor memory which boots the operating system on the hardware platform.

3.4.2 The Device-Tree

```
&eeeprom {
    #address-cells = <1>;
    #size-cells = <1>;

    board_sn: board-sn@0 {
        reg = <0x0 0x14>;
    };
};
```

```
eth_mac: eth-mac@20 {
    reg = <0x20 0x6>;
};

board_name: board-name@d0 {
    reg = <0xd0 0x6>;
};

board_revision: board-revision@e0 {
    reg = <0xe0 0x3>;
};
};
```

Listing 3.1: A snapshot of the ZCU102 device-tree showing the contents of the I2C EEPROM chip.

Listing 3.1 shows the information of the I2C EEPROM node of the device-tree of the Xilinx ZCU102 board. It shows the addresses in the I2C EEPROM where the the board serial number, Ethernet MAC addresses, board name and board revision number are stored.

The Linux kernel needs to know the information about the processor on which it is executing, the peripherals that are associated with the processor, their interfacing with the processor and their physical addresses. In order to initialize the drivers and the services associated with these peripherals, the kernel also needs to check if the functionalities that have been activated in its configuration are actually supported by the hardware that it is controlling. This information could be about clocks and registers of the hardware or about any peripheral associated with the hardware such as the external memory, Ethernet, I2C and HDMI output.

One way of passing such information about the hardware to the kernel is by hard-coding this information with the device-driver header files in the bootloader and hard-coding the same information in the device-driver header files of the kernel itself [26]. However, this would mean that each piece of hardware would require special bootloader and kernel device-drive header files depending on the specialised hardware design. This approach does not scale well as the Linux developers would not be able to generalise their bootloader and kernel for a wide range of devices using processors with similar architectures and similar functionalities but differing hardware designs [26].

Another method implemented in modern PCs is to request a particular communication bus (PCI, USB, SPI) in the computer hardware to inform the kernel about the different devices and peripherals interfaced to the bus. This process is called enumeration. The Zynq UltraScale+ does not support passage of device-information in this manner.

The Zynq UltraScale+ uses the device-tree [3][24] to pass the device and peripheral information such as physical addresses of devices, I/O register addresses, memory address space and interrupt information to the kernel during boot time.

The device-tree is represented in textual format in a file with the extension “.dts”. This is a source text file that describes the information of devices and interconnecting buses present interfaced with a computing hardware. It is organized in the form of “nodes” that have a root location represented by “/” just like in the Linux root file system. Every node has a name which represents a device or a bus interfaced with the processor and the node consists of “properties”. Each parent node for a particular peripheral or bus may contain “child” nodes for devices that are interfaced with that peripheral or bus. Values of the properties can be strings, lists of strings or they could be empty if the absence or presence of the value conveys a Boolean logic to the kernel. The device-tree source file is compiled into a device-tree blob (.dtb) with the help of “dts” compiler.

3.4.3 The Linux Kernel

The primary functions of the Linux kernel [25][27] are as follows:

- Scheduling of processes and setting up an environment for their execution.
- Allocating memory to a process and protect the memory used by a process from other processes.
- Manage the computer resources and the access to these resources.
- Control the device I/O, networking and other hardware peripherals as well as the access to these peripherals and their services requested by the userspace.
- Manage inter-process communication to ensures efficient execution of processes.
- Ensure the integrity of the system when the computer system has multiple users authorised with modifications to the root file system and software packages.

The bootloader loads the device-tree and the kernel into the processor DRAM [23] and provides the kernel with the address of the device-tree before the kernel starts executing. The kernel then sequentially checks all the hardware peripherals, clocks and memories that have been initialized by the FSBL and then activates all the services and functionalities that are associated with the underlying hardware and have been activated in the kernel configuration. Once the kernel is finished with these tasks, it mounts the root file system and executes the “init” process which allows the user to enter the userspace.

Depending on the bootloader configuration, different file formats can be used to load the kernel. The Zynq UltraScale+ processors can load the kernel through the “Image” format or through “image.ub” file which packs the kernel Image, the device-tree blob and a compressed root file system CPIO archive together in one FIT (Flattened Image Tree) format file [24].

3.4.4 The “init” Process

The “init” [25] process is the first user space process initiated by kernel and it is responsible for initializing the system management services before allowing the users to login. Multiple versions of init process exist. System V was the system manager and the init process used in CentOS before CentOS7 switched to systemd. systemd is the init process that executes in CentOS 7 and later versions after the root file system has been mounted. The run-levels and the init process may vary across different Linux distributions. All the software services that have been installed in the root file system and enabled in the userspace are executed by the “init” process before the users login to the system.

3.4.5 The Linux Root File System

The Linux root file system [23][25][26] contains all the binary executable files, device information, process logs, software packages and libraries which are required by the user in its userspace to efficiently use the Linux operating system. The file system on the disk is organised into directories. The root file system is mounted at the “/” directory of the file system marking the top of the root file system hierarchy. Some of the main directories in the root file system are described below:

- `/bin`: This directory contains the binary utilities which can be executed by all users, including superuser, administrator and a general user. The directory may contain hard links and symbolic links to other executable binaries.
- `/boot`: The bootloader files, configuration files and the kernel Image are present in this directory.
- `/dev`: This directory contains the list of the devices which are connected to the hardware.
- `/etc`: This directory contains system configuration files such as the TFTP, NFS and DHCP server configuration files.
- `/lib`: This directory contains the shared libraries and kernel modules.
- `/media`: This directory points to removeable media attached to the computer.
- `/mnt`: This directory contains temporary file systems mounted from external devices.
- `/opt`: The user added application softwares can be placed here.
- `/var`: This directory contains temporary, variable data and logs of various processes being executed during system operation.
- `/root`: Home directory for the root or superuser.
- `/home`: Home directory for other non-root users. Contains sub-directories for different users.
- `/proc`: This directory contains temporary, variables files and data information about the computer hardware such as CPU performance and cache memory.

The Extended (EXT) file system is the file system of choice to boot Linux for Zynq UltraScale+ [24]. During this thesis project, we also use the Network File System (NFS) protocol to mount an externally located root file system over the network.

3.5 The Zynq UltraScale+ Linux Boot Process

In order to boot Linux on the Zynq UltraScale+, the boot image (BOOT.BIN) needs to be present on the booting medium as per the boot mode selected by the user [3]. The ZCU102 board supports booting via JTAG, Quad-SPI Flash, SD Card and NAND Flash Drive [22]. For this thesis project, we boot the Linux from the SD Card and thus the BOOT.BIN file must be present on the FAT32 partition of the SD Card. The BOOT.BIN file has been configured to contain the Platform Management Unit Firmware (PMUFW), the FSBL and the U-Boot executable files which is the recommended configuration for the Zynq UltraScale+ [24]. The BOOT.BIN can optionally contain the FPGA bitstream as well to program the Programmable Logic (PL). For a complete boot over SD Card, the kernel image (Image) and the device-tree blob (“project.Name.dtb”) should be present on the FAT32 partition as well. The root file system must be present on the EXT4 partition of the SD Card. The kernel image and the device-tree can be packed together in the “image.ub” file and placed on the FAT32 partition instead of being placed as separate files. The automated Network boot is explained in Chapter 4 and Chapter 5 in more detail.

The booting of Linux on the Xilinx Zynq UltraScale+ can be divided into four stages which are as follows:

- The Boot Setup Stage

- The Bootloader Stage
- The Kernel Booting Stage
- The “init” Stage

3.5.1 The Boot Setup Stage

The Platform Management Unit (PMU) and the Configuration Security Unit (CSU) are responsible for setting up the Zynq UltraScale+ MPSoC before the Linux can be booted on the PS [3].

The boot setup stage can be divided into 3 stages [3] which are as follows:

- **Pre-configuration Stage:** The PMU consists of a MicroBlaze processor which loads executable software from PMU ROM. The PMU ROM is already loaded with the PMU bootROM (PBR) code which configures the power state of the device, initialize the RAMs, test memories and registers. After the PBR code has been executed, it hands over system control to the configuration security unit (CSU). The important steps of the pre-configuration stage [3] are listed below:
 1. Initialize the System Monitor.
 2. Initialize the Phase Locked Loops (PLL) for clocks.
 3. Clear PMU RAM.
 4. Initialize the Dynamic Random Access Memory (DRAM).
 5. Release the CSU or enter error state.
- **Configuration Stage:** The CSU executes the CSU bootROM from CSU ROM and performs the following tasks [3]:
 1. Initialize the On-Chip Memory (OCM).
 2. Determine the boot-mode by capturing the boot-switch configuration of the board at Power-On-Reset (POR).
 3. Load the FSBL in the OCM as well as the PMU Firmware (PMUFW) in the PMU RAM.

The PMUFW initializes the PMU hardware and service modules for the PMU. After PMUFW begins executing, it enters the sleep mode. It wakes up only for servicing interrupts from the PMU hardware and I/O devices to perform platform management services.

- **Post-configuration Stage:** Once the FSBL begins executing, the Zynq UltraScale+ MPSoC boot enters the post-configuration stage.

3.5.2 The Bootloader Stage

First Stage Bootloader

The First Stage Bootloader (FSBL) is the Secondary Program Loader (SPL) in the terminology of generic ARM processor booting process [23]. The FSBL performs the following tasks [23][3]:

- Lookup the FAT32 Boot partition of the SD card to find the PL Bitstream and the second stage bootloader i.e the U-Boot.
- Initialize the PS hardware, I/O devices, memory and clocks as per the configuration defined by the hardware design specified in the Xilinx Vivado design suite.
- Optionally, program the PL with the FPGA bitstream.
- Loads the ARM Trusted Firmware (ATF) and the U-Boot in the APU processor memory [3].

U-Boot

The U-Boot [23][26] is the Second Stage Bootloader (SSBL) for the Xilinx Zynq UltraScale+ MPSoC or the Tertiary Program Loader (TPL) in the terminology of the generic ARM boot process [23]. When the processor is powered on, the processor memory does not contain an operating system, so a bootloader is required to load the Linux Kernel and the device-tree into the processor memory from the storage space. The U-Boot helps to fulfill this function. There is no necessity of using multiple stages of bootloaders and even one bootloader like the FSBL is sufficient to load the Linux kernel into the memory. However, the U-Boot with its command terminal helps users and system administrators in debugging the boot process, making tests to validate device initialization and modify the boot process if necessary. It has its own commands, environment variables and scripting definitions. The U-Boot can load the kernel and device-tree over the network using Ethernet, over JTAG, from the SD card, Quad-SPI flash and from NAND flash drive. The Zynq UltraScale+ follows a two stage bootloading process and uses both FSBL and U-Boot.

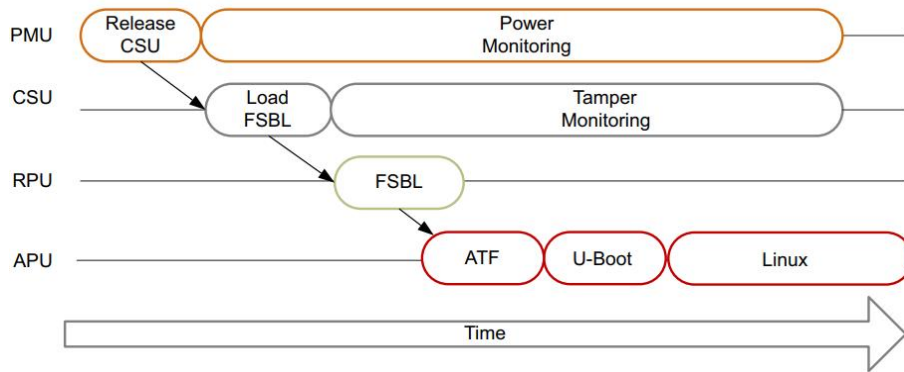


Figure 3.3: The Xilinx Zynq UltraScale+ MPSoC boot flow [3]

Figure 3.3 shows the progress of the Linux boot flow on the Zynq UltraScale+. It shows the role of the different units of Zynq UltraScale+ in booting the Linux operating system on the hardware. The PMU can be seen to release the CSU, which loads the FSBL. The FSBL then loads the ATF and the U-Boot. The U-Boot is then responsible for booting the Linux on the hardware.

3.5.3 Boot Flow

The Zynq UltraScale+ MPSoC can boot in two modes: secure and non-secure modes [3]. The boot flows have been explained below.

Non-Secure Boot Flow

In this boot mode, the PMU releases the CSU and enters the service mode where it monitors the platform. The CSU loads the FSBL in the APU's On-Chip Memory (OCM) and the PMUFW in the PMU RAM. The PMUFW executes in parallel to the FSBL execution and executes until the Linux has been booted on the Zynq UltraScale+ MPSoC. The FSBL initializes the peripherals, I/O devices, memories, clocks and hands off to the ATF which then hands-off to the U-Boot. The U-Boot then loads the Linux Kernel in the APU processor memory.

Secure Boot Flow

The secure boot flow differs from the non-secure boot flow just in a few extra authentication and decryption steps executed by the CSU [3] which are as follows:

- Check if the FSBL requires authentication.
- If the FSBL requires an authentication check, proceed only if the check has been passed and check if the FSBL has been encrypted.
- Decrypt the FSBL image and load the FSBL in the OCM.

The FSBL also checks the authentication and decryption of any files it tries to load and they are only loaded upon successful authentication and decryption. The boot flow explained above is applicable to all different ways of booting Linux whether all the necessary boot files such as the device-tree blob and kernel image are present on an SD Card, the Quad-SPI Flash or the TFTP server.

For this master thesis, we use the non-secure boot flow as the computers inside the CERN and CMS network are secured behind the CERN firewall and there is router-level security which prevent unauthorised access to the network boot infrastructure within CERN. Thus, the Linux boot files for the Zynq UltraScale+ hardware cannot be tampered with and do not require extra authentication and decryption.

3.5.4 The Kernel Booting Stage

The U-Boot is responsible for loading the kernel Image and the device-tree into the APU processor memory and handing over the control to the kernel. The kernel is also provided with the address of the device-tree blob which informs the kernel about the hardware peripherals, I/O devices, memories and clocks. On the basis of this information, the kernel begins booting and activates the drivers and services for which the kernel has been configured. Once the booting of the kernel is over, it proceeds to mount the root file system as per the boot arguments passed to the kernel. Depending on the boot argument passed to the kernel, the kernel mounts the root file system over NFS or from the SD card or any other permanent storage location.

3.5.5 The “init” Stage

Once the root file system has been mounted, the kernel looks for the executable of the init process specified for the Linux distribution. Once the kernel locates the init process, it executes the init process and hands over control to the init process. The init process then proceeds to activate

all the system management and other background services which have been installed in the root file system and enabled in the userspace. At the end of the activation of these services, the init process starts the user-login process which allows the user to enter the userspace.

3.6 The Userspace

Users enter the userspace after the login. The userspace allows the user to use different software utilities and libraries installed in the root file system and utilise the functionalities provided by them to efficiently do computing tasks supported by the computer hardware. The userspace refers to the user processes which are running in an operating system which interact with the the Linux kernel in order to efficiently use the computing hardware. The userspace provides an interface to the user to interact with the kernel and the underlying hardware.

3.7 Summary

This chapter gave the reader an overview of the Xilinx Zynq UltraScale+ MPSoC and the Xilinx ZCU102 Evaluation Board as well as informed the reader about the various components and the steps involved in booting Linux on the Zynq UltraScale+ MPSoC.

Chapter 4

Building the CentOS 8 Linux Distribution

In this chapter, the focus is on explaining the tools required to build a customised CentOS 8 Linux distribution and the method followed to build one. An overview of the Yocto Project is presented as a background to the PetaLinux Tools. The process of building a Linux distribution using PetaLinux Tools is explained. Additionally, the process of porting a CentOS 8 4.18 kernel and building a CentOS 8 root file system for the Zynq UltraScale+ MPSoC is explained in detail. Finally, a qualitative comparison between the PetaLinux Tools and the Yocto Project has been presented at the end of this chapter to help readers choose a tool-chain as per their requirements.

4.1 The Yocto Project

This section is based on the collective finding of me and my former colleague Awais bin Zahid who also worked briefly on the Yocto Project. The work with Yocto Project forms the basis of the exploration of building a Linux distribution with the help of the PetaLinux Tools. The PetaLinux Tools is a tool chain released by Xilinx which helps developers to build and customise a Linux distribution for processors developed by Xilinx. The PetaLinux Tools utilise the Yocto Project build process to build the Linux distribution while hiding the underlying complexities that are associated with direct usage of the Yocto Project. The PetaLinux Tools provide an easy-to-use command line interface to help developers configure and build the FSBL, U-Boot, the kernel and the root file-system. In order to appreciate the PetaLinux build process, it is important to understand certain key concepts associated with the Yocto Project and they have been presented in this section.

The Yocto Project is an open source project that helps developers to build customised Linux distributions for their embedded systems and it supports different processor architectures. The Yocto Project helps in building customised components like FSBL, U-Boot, device-tree and kernel Image for booting Linux on these embedded systems. The Yocto Project utilised a layered development model allowing users flexibility to make layer specific modifications that are related to the hardware board support package (BSP), the kernel, the device-tree, the root file system and the U-Boot.

4.1.1 The Layer Model

Yocto layers or meta-data layers are repositories of configuration scripts and build scripts along with build specifications (BitBake recipes) which instruct the Yocto build system (OpenEmbedded Build System) about how to build a customised Linux distribution. These layers may be hardware specific or may be flexible enough to customise them for other architectures as well.

Layers can be used to separate or combine certain build tasks depending upon how much complexity the developer wants to associate with a certain layer. Increasing the build tasks associated with a certain layer increases the complexity associated with the layer and makes it difficult for the developers and maintainers for customization, maintenance and reuse of these layers in the future. The Yocto source directory contains different layers for different layers associated with a Linux distribution (hardware, U-Boot, kernel, root file system) and these layers have the prefix “meta-” associated with the name of all different layers.

The contents of the “bblayers.conf” file in the “yocto” folder of PetaLinux Tools can be seen in Listing 4.1:

```
LCONF.VERSION = "7"

BBPATH = "${TOPDIR}"
SDKBASEMETAPATH = "${TOPDIR}"
BBLAYERS := " \
    ${SDKBASEMETAPATH}/layers/core/meta \
    ${SDKBASEMETAPATH}/layers/core/meta-poky \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-perl \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-python \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-filessystems \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-gnome \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-multimedia \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-networking \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-webserver \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-xfce \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-initramfs \
    ${SDKBASEMETAPATH}/layers/meta-openembedded/meta-oe \
    ${SDKBASEMETAPATH}/layers/meta-browser \
    ${SDKBASEMETAPATH}/layers/meta-qt5 \
    ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-xilinx-bsp \
    ${SDKBASEMETAPATH}/layers/meta-xilinx/meta-xilinx-contrib \
    ${SDKBASEMETAPATH}/layers/meta-xilinx-tools \
    ${SDKBASEMETAPATH}/layers/meta-petalinux \
    ${SDKBASEMETAPATH}/layers/meta-virtualization \
    ${SDKBASEMETAPATH}/layers/meta-openamp \
    ${SDKBASEMETAPATH}/workspace \
"
```

Listing 4.1: Contents of bblayers.conf configuration file in the Yocto source directory

Listing 4.1 shows the “bblayer.conf” file of PetaLinux Tools. It shows multiple meta-data layers such as the meta-poky, meta-openembedded, meta-xilinx and meta-petalinux layers as well as a few other layers used by PetaLinux to create a Linux distribution. The meta-openembedded layer specifies metadata for configuring the build scripts in the OpenEmbedded build system used by Yocto. The meta-xilinx layer has all the information pertaining to Xilinx processors and Linux distributions for them. The meta-petalinux layer is a PetaLinux specific layer configuring the PetaLinux build scripts. meta-poky layer is used to create a basic minimal Linux distribution which is then customised by the PetaLinux Tools as per the hardware design, U-Boot configuration and kernel configuration.

4.1.2 Poky

Poky is a reference distribution used by the Yocto Project to create customised Linux distributions. It contains the OpenEmbedded Build System along with the metadata to help the build system build a minimal Linux distribution. The “core” directory of the “yocto” folder in the PetaLinux installation directory contains the meta-poky layer along with the BitBake and OpenEmbedded-Core scripts which the OpenEmbedded Build System uses to build the Linux distribution. The basic minimal Linux distribution created with the help of Poky is customised using additional meta-layers such as the meta-xilinx and meta-petalinux layers to customise the Linux image for Xilinx processors. Metadata is essentially configuration scripts and BitBake recipe files that are explained in the following sub-section.

4.1.3 BitBake Engine

BitBake is a build engine used by the Yocto Project to build a Linux distribution and it is a part of the OpenEmbedded Build System. It can be imagined as being similar to Linux kernel in its function since it manages the execution, scheduling and parallel management of different tasks described in the OpenEmbedded build scripts and ensures that each task has the requisite resources necessary for execution. To manage these tasks, the BitBake engine uses scripts known as BitBake recipes with the extension “.bb”.

There are different types of files parsed by the BitBake Engine. They are as follows:

1. **Recipes:** The recipes provide details about which particular pieces of software should be used and where to get them from. They mention whether the software needs to be downloaded from a particular web repository, which patches to apply to the software, the tools to be used, the configuration to be used while compiling the software and the output format of the compiled code. A recipe also contains the information about the recipe version, the license of the software package and the web-link or path to the source repository. Modifications to a given recipe can be specified through files with the same recipe name but with the extensions “.bbprepend” and “.bbappend”. Prepend files are specified by developers before the configuration process and append files are specified by the BitBake engine itself after the distribution configuration process. They are used so that basic behaviour of the original BitBake recipe is not altered.
2. **Class Data:** Class files have the extension “.bbclass” and they are used to share information between BitBake recipe files. For example, the xilinx-platform-init class in Listing 4.2, which specifies common files and headers that are necessary to initialise the Zynq and Zynq UltraScale+ platforms.
3. **Configuration Data:** The configuration files with the extension “.conf” are used to define environment variables, file paths, machine settings, software versions and all other configuration variables that are necessary to govern the OpenEmbedded build process. The “bblayers.conf” file in Listing 4.1 defines the different meta-layers to be used for building a PetaLinux distribution.

The “xilinx-platform-init.bbclass” file describing the platform initialisation files for Zynq and Zynq UltraScale+ platforms:

```
# This class should be included by any recipe that wants to access or provide
# the platform init source files which are used to initialize a Zynq or ZynqMP
# SoC.
```

```
# Define the path to the xilinx platform init code/headers
PLATFORM_INIT_DIR ?= "/usr/src/xilinx-platform-init"

PLATFORM_INIT_STAGE_DIR = "${STAGING_DIR_HOST}${PLATFORM_INIT_DIR}"

# Target files use for platform init
PLATFORM_INIT_FILES ?= ""
PLATFORM_INIT_FILES_zynq = "ps7_init_gpl.c ps7_init_gpl.h"
PLATFORM_INIT_FILES_zynqmp = "psu_init_gpl.c psu_init_gpl.h"
```

Listing 4.2: Contents of xilinx-platform-init.bbclass class file in the Yocto source directory.

Listing 4.2 shows the “xilinx-platform-init.bbclass” file. We can see the paths for the Xilinx platform initialisation source code and header files specified in the “xilinx-platform-init.bbclass”. This informs the build recipe files where to find the platform initialisation files for Zynq and Zynq UltraScale+.

The “u-boot-xlnx-2019.1.bb” BitBake recipe used by PetaLinux 2019.1 to build the U-Boot for Xilinx platforms:

```
UBOOT_VERSION = "v2019.01"
XILINX_RELEASE_VERSION = "v2019.1"

UBRANCH ?= "master"

SRCREV ?= "d895ac5e94815d4b45dcf09d4752c5c2334a51db"

include u-boot-xlnx.inc
include u-boot-spl-zynq-init.inc

SRC_URI_append_kc705-microblazeel = " \
file://microblaze-kc705-Convert-microblaze-generic-to-k.patch"

LICENSE = "GPLv2+"
LIC_FILES_CHKSUM = "\
file://README; beginline=1; endline=4; md5=744e7e3bb0c94b4b9f6b3db3bf893897"

# u-boot-xlnx has support for these
HAS_PLATFORM_INIT ?= " \
    zynq-microzed-config \
    zynq-zed-config \
    zynq-zc702-config \
    zynq-zc706-config \
    zynq-zybo-config \
    xilinx-zynqmp-zcu102-rev1.0-config \
    xilinx-zynqmp-zcu106-revA-config \
    xilinx-zynqmp-zcu104-revC-config \
    xilinx-zynqmp-zcu100-revC-config \
    xilinx-zynqmp-zcu111-revA-config \
    xilinx-zynqmp-zc1275-revA-config \
    xilinx-zynqmp-zc1275-revB-config \
    xilinx-zynqmp-zc1254-revA-config \
    "
```

Listing 4.3: Contents of u-boot-xlnx-2019.1.bb BitBake recipe file in the Yocto source directory.

Listing 4.3 shows the file “u-boot-xlnx-2019.1.bb”, which is a BitBake recipe for U-Boot. It mentions the version of the U-Boot, the branch of the Git repository, the paths to a patch file that would be applied to the U-Boot source code, the license version and names of various configuration files which the BitBake recipe supports.

4.1.4 OpenEmbedded Build System

The OpenEmbedded Build System consists of two important components: the BitBake task execution engine and the OpenEmbedded core. It is a customized build system developed for the Yocto Project, and it was originally derived from the OpenEmbedded Project. The meta-openembedded layer is the layer for the OpenEmbedded Build system and it specifies the recipes, classes and configuration files for building the kernel, file systems, adding software packages to the root file systems and adding driver and software support to the U-Boot and kernel modules.

The following points describe a brief summary of the Yocto workflow:

1. Developers download the Yocto project version of their choice and different meta-layers that they wish to use to configure their customised Linux distribution.
2. Machine type, architecture, kernel and U-Boot configurations, patches, recipe modifications and amendments are specified by the developers in the prepend, append and configuration files.
3. The build system downloads or fetches the source code from the specified repositories or paths and clones it in a working directory.
4. Patches are applied to the source code and then the source code is compiled as per the configuration provided.
5. The compiled software is then packaged into an executable binary format specified by the developers and then the final Linux boot images are created.

More can be read about the Yocto Project on the following weblink:

<https://www.yoctoproject.org/docs/>

The process of configuring, fine-tuning and debugging the build process of Yocto Project to build a customised Linux distribution can be time consuming. Keeping this in mind, Xilinx developed the PetaLinux Tools, to accelerate the development process for Xilinx platforms by providing an easy-to-use command line interface and hiding the underlying complexities of the Yocto build process. A qualitative comparison of both the tool chains has been presented at the end of the chapter.

4.2 Developing a Distribution with PetaLinux Tools

The PetaLinux Tools is a tool chain designed to help hardware developers develop a Linux distribution for platforms using the Xilinx processors. In order to get the distribution built, there are certain pre-requisites that are necessary and they are as follows [24]:

1. For hardware developers developing a custom Linux distribution, Xilinx Vivado Design Suite to generate the hardware description file (HDF) and the board-support packages (BSP).
2. A computer running a 64-bit operating system with a minimum of 8 GB RAM, 2 GHz processing speed and storage space of at least 10 GB [24].

3. Software dependencies necessary to help PetaLinux Tools/Yocto execute different tasks specified in the build scripts. The packages necessary for the PetaLinux Tools 2019.1 can be found in the PetaLinux 2019.1 user guide [24].

The Vivado Design Suite and PetaLinux Tools version should be the same otherwise it could take some time to modify PetaLinux Tools' scripts for allowing HDF files from an older or newer Vivado version. It is recommended to not make modifications to the PetaLinux Tools' build scripts without understanding the whole build hierarchy as it may break other build processes.

Xilinx Vivado Design Suite can be downloaded from the following weblink:

<https://www.xilinx.com/support/download.html>

PetaLinux Tools can be downloaded from the following weblink:

<https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html>

4.2.1 Minimal Vivado Design for Zynq UltraScale+

Hardware developers designing their platforms using Zynq UltraScale+ need to have certain basic configurations activated in their FPGA design in order to build a working Linux distribution for the Zynq UltraScale+ processing system. The minimal design must contain [24]:

1. External memory with at least 64 MB of memory.
2. UART for serial console.
3. Optional non-volatile memory for Quad SPI Flash or SD card.
4. Optional Ethernet which can be essential for network access and network booting.

Alternatively, developers using Xilinx evaluation boards can use the board support packages (BSPs) offered by Xilinx for these evaluation boards. These BSPs contain the hardware description describing all the different peripherals that are connected to the Zynq UltraScale+ programmable logic array and a bitstream to program the programmable logic. These BSPs are eventually parsed by the underlying OpenEmbedded Build System recipes of PetaLinux Tools to generate a device-tree as well as the kernel and U-Boot configuration. During this project, the BSP provided by Xilinx for the ZCU102 board was used.

The Zynq UltraScale+ BSPs can be downloaded from the following weblink:

<https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html>

4.2.2 Installing the PetaLinux Tools

Once the PetaLinux Tools have been downloaded, the following bash script must be executed to install the tool chain at the desired location.

The script to install the PetaLinux Tools on your Linux system is as follows:

```
#!/bin/bash

HOME=/home/kmor

#Install all software dependencies for PetaLinux Tools
sudo yum install gawk make wget tar bzip2 gzip python unzip perl patch diffutils \
diffstat git cpp gcc gcc-c++ glibc-devel texinfo chrpath socat perl-Data-Dumper \
perl-Text-ParseWords perlThread-Queue python34-pip xz which SDL-devel xterm \
autoconf libtool zlib-devel automake glib2-devel zlib ncurses-devel \
openssl-devel dos2unix flex bison glibc.i686 screen pax glibc-devel.i686 \
compat-libstdc++-33.i686 libstdc++.i686

#Make a directory for PetaLinux Installation at your home directory

mkdir $HOME/petalinux-2019.1

#Install PetaLinux 2019.1 in the directory created above

cd ~/Downloads

./petalinux-v2019.1-final-installer.run $HOME/petalinux-2019.1
```

Listing 4.4: Script for installing the PetaLinux Tools

Listing 4.4 presents the script to install PetaLinux Tools in a Red Hat Linux environment like CentOS. The HOME variable specifies the path to the installation directory and should be modified as required. The script also installs PetaLinux Tools dependencies as mentioned in the user guide [24]. The script makes a directory for PetaLinux installation in the path mentioned by variable HOME and finally, executes the installation with the help of installation binary.

4.2.3 Creating a PetaLinux Project

Once the PetaLinux Tools have been installed, it is time to import the environment variables to have correct settings configured in the shell.

The following script helps in creation of the project.

The script to create a PetaLinux project:

```
#!/bin/bash

HOME=/home/kmor

PROJECT=PetaLinux

BSP=~/Downloads/xilinx-zcu102-v2019.1-final.bsp

#Change to the PetaLinux installation directory

cd $HOME/petalinux-2019.1/2019.1

#Import the system settings from settings.sh

source settings.sh

#Create a PetaLinux project

petalinux-create -t project -n $PROJECT -s $BSP
```

Listing 4.5: Script for creating a PetaLinux Project

Listing 4.5 presents the script to create a PetaLinux project. The `PROJECT` variable can be changed to the name that user wants. `BSP` points to the path where the hardware design's board support package is located. The script then sources the environment with `"source settings.sh"` to setup the PetaLinux build environment. The `"-t"` option specifies the project type (in this case project), `"-n"` option specifies the name of the project and `"-s"` option specifies the source path for the board support package [24].

Hardware developers can create the board support packages for their design in the Vivado design suite or leave this field blank to specify a path to the hardware design file during configuration stage. The command for that is `"petalinux-config --get-hw-description"`. This command helps PetaLinux configure the Linux distribution as per the Hardware Description File (HDF) provided. The configuration step has been explained in the next section. The passing of the path to the BSP helps in generating the configuration for building the correct device-tree, FSBL, U-Boot and kernel drivers necessary to boot the Linux on the Xilinx platform.

4.2.4 Configuring the PetaLinux Project

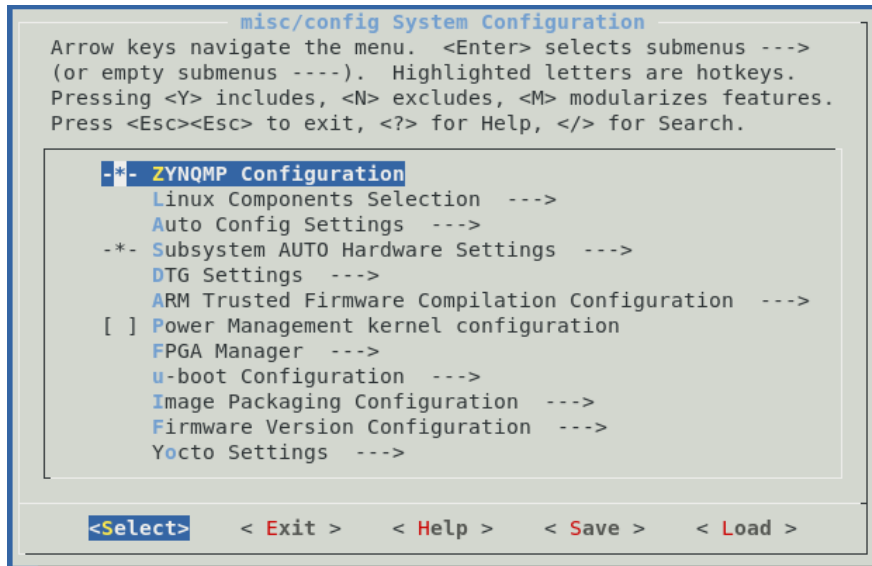


Figure 4.1: The Menuconfig home screen for configuring the PetaLinux project.

Figure 4.1 shows the Menuconfig screen which is generated after the `petalinux-command` parses the Kconfig files generated after the `petalinux-create` command. This Menuconfig is sufficient to configure the whole project. However, developers can make individual configurations to boot components as well, with similar Menuconfig interface. Individual component configurations were often required to do complex configurations in order to implement an automated network boot.

Once the project has been created, it is important to pass the correct configuration to the PetaLinux build scripts to build various Linux boot components. The PetaLinux project creation results in the generation of a basic intermediate Kconfig file with the help of the hardware design and it is located in the `"build/misc/config"` directory of the created project. This intermediate Kconfig is offered to the developer in the form of a Menuconfig window during the initial high-level configuration initiated by the developer.

This high-level configuration is used to generate the final `"config"` and `"rootfs.config"` files

which are used to configure the U-Boot, the kernel, the device-tree and the root file system. This configuration is done with the help of “make” and it consumes a lot of time as it is used to generate certain meta-layers like the “meta-plnx-generated” and ”meta-user” layers which contain some newly created BitBake recipes and BitBake appends for different Linux boot components. Additional configurations for the U-Boot, kernel, device-tree and root file system are possible with a special commands such as:

- “petalinux-config -c u-boot” - Config U-Boot
- “petalinux-config -c kernel” - Config kernel
- “petalinux-config -c device-tree” - Config device-tree
- “petalinux-config -c rootfs” - Config root file system

However, unless additional features are required (eg. features for network boot), it is not necessary to configure each of these components. Passing the BSP during the PetaLinux project creation is sufficient to configure the basic, minimal configuration necessary to boot Linux on the hardware. Each of these commands generate a Menuconfig screen by parsing their respective Kconfig file hierarchy.

```
#!/bin/bash
HOME=/home/kmor
PROJECT=PetaLinux

#Uncomment the following line to specify path to Hardware Description File
#HDF=<Path-To-HDF>

#Change to the PetaLinux installation directory
cd $HOME/petalinux-2019.1/2019.1

#Import the system settings from settings.sh
source settings.sh

#Change to the PetaLinux project directory
cd $PROJECT

#Global configuration for the PetaLinux project. Append -c with options: u-boot or
#kernel or device-tree or rootfs whichever you wish to configure individually.

petalinux-config

#Uncomment the following line to generation configuration from HDF
#petalinux-config --get-hw-description=HDF
```

Listing 4.6: Script to configure PetaLinux project

Listing 4.6 shows the script to configure a PetaLinux project. The script enters the project location, sources the PetaLinux environment settings and then enters the project directory and executes petalinux-config command to configure the project. Options for configuring the u-boot, kernel, device-tree and root file system can be appended to the command to configure boot components individually.

During the initial global configuration, it is important to specify the IP address configuration, the kernel boot arguments, the source code path for the kernel and the U-Boot and the type of

the root file system in order to ensure that the correct configuration is passed to the U-Boot and kernel. This configuration is important to implement a network boot of Linux on the Zynq UltraScale+.

The specification of the root file system type is critical as it decides which kind of file system support is activated in the kernel configuration and also the driver modules which are necessary to mount the root file system. The following set of images illustrate the different configurations specified in the menu-config window for configuring a PetaLinux distribution.

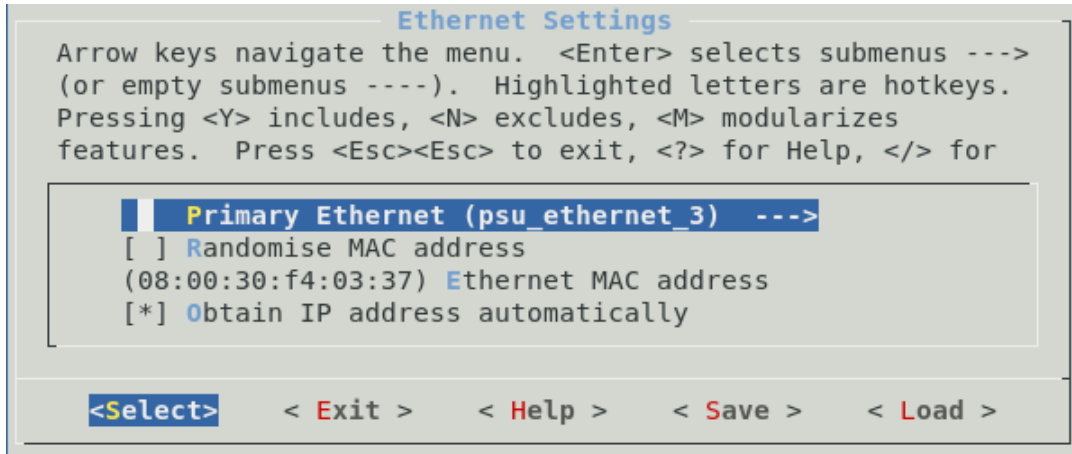


Figure 4.2: Menuconfig to activate automatic IP address assignment.

Figure 4.2 shows the options to configure the Ethernet properties in the Menuconfig. Here users can choose their primary Ethernet interface, MAC address (optional) for the U-Boot environment and whether the Ethernet acquires IP address automatically or not.

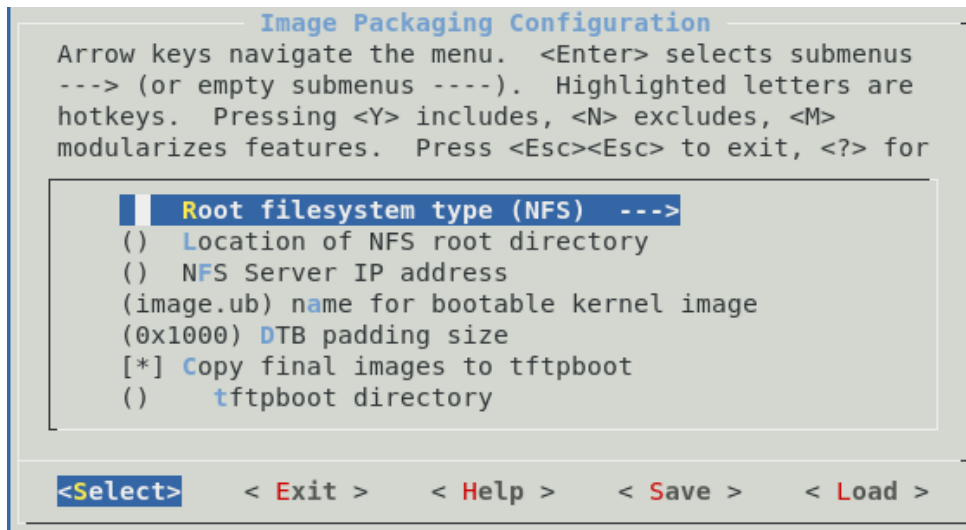


Figure 4.3: Menuconfig to specify root file system type.

Figure 4.3 shows the options to choose the root file system type, location of NFS (Network File System) root directory (optional, obtained from DHCP (Dynamic Host Configuration Protocol) server automatically), NFS server IP address (static, not recommended to activate if using automatic IP address assignment from DHCP server) and path to the TFTP (Trivial File Transfer Protocol) server directory (optional, obtained from DHCP server automatically).

All the above configurations of the root file system, the Ethernet, the support for TFTP are necessary for the network boot of Linux on the ZCU102 board but they are not sufficient to execute an automated network boot. There are many challenges while implementing the network boot which need to be tackled. One such challenge was to configure the correct NFS version on both server and client side. This was realised after numerous failed attempts of NFS mount and it was found that the server is configured for NFSv4 whereas the kernel has been configured automatically for NFSv2. The NFS server and kernel configuration had to be changed to NFSv3 for ensuring the NFS mount.

Above all, it is also important that the firewall on the NFS and TFTP servers are configured properly to allow the requests from the U-Boot and the kernel. It is even a bit more complex in a big network like CERN where multiple computers are connected to the network. The firewall had to be configured properly to allow requests for these services on specific ports and all other unnecessary services and ports were closed. Before doing this, the TFTP and the NFS requests from the U-Boot and the kernel kept getting rejected. The scripts for TFTP and NFS server configuration along with the firewall setup are presented in the Appendix B.

Other challenge faced during the implementation of the network boot was the failure of the kernel to acquire IP address from the DHCP server. After careful investigations of the DHCP server configurations and the kernel boot arguments option it was found that the PetaLinux configuration is not sufficient to get the right boot arguments and one must specify the boot arguments explicitly either in U-Boot environment or during PetaLinux configuration so that kernel makes a successful DHCP request. A significant time was spent in understanding that the PetaLinux configuration is not setting the correct boot arguments for kernel to make a DHCP request. Ultimately, the boot argument setting for the kernel had to be studied. The default boot argument generated automatically by PetaLinux was “earlycon console=ttyPS0,115200 clk_ignore_unused root=/dev/mmcblk0p2 rw rootwait”. The automatic boot argument setting had to be disabled and manually the kernel boot arguments had to be specified. The new kernel boot arguments for network boot were “earlycon console=ttyPS0,115200 clk_ignore_unused root=/dev/nfs ip=dhcp rw”. In both the boot arguments, the “root” option points to the root file system mount point. The new option “ip” specifies whether the IP address is static or provided automatically by DHCP server. The “console” option configures the kernel to print the boot debug over the UART port ttyPS0 on ZCU102 board. This port is chosen as per the hardware information provided by the BSP.

There was another challenge faced with the DHCP request made by the U-Boot where the U-Boot was unable to save the TFTP server IP address provided by the DHCP server. This basically derailed the whole automated network boot as boot files could not be downloaded from the TFTP server. This issue had to be addressed by modifying the PetaLinux build scripts which were blocking the setting of the TFTP server IP address in the U-Boot environment by the DHCP server. This has been elaborated upon in detail in Chapter 5.

4.2.5 Building a PetaLinux Project

Once the configuration is taken care of and the necessary configuration files and recipes have been generated, it is time to build the distribution. That is done through a simple command - “petalinux-build”. Although it is a small command, it is responsible for many tasks and takes the longest time to complete. The petalinux-build command is responsible for building the PMU Firmware (PMUFW) and the First Stage Bootloader (FSBL) which are the pieces of software executed before loading the U-Boot. The petalinux-build command is also responsible for downloading or copying the U-Boot and kernel source codes in the PetaLinux project “build” directory and applying patches to the source code before beginning to compile the source code with the help of make and GCC compiler. This is the same process which is carried out with Yocto Project or

standalone compilation of U-Boot or kernel with the help of make and GCC.

Once the kernel has been compiled, the `petalinux-build` script proceeds to build the device-tree as per the HDF and the driver configuration of the kernel such that the drivers for a given hardware can find the information about the hardware at the relevant node in the device-tree and can also physically locate the hardware that they are supposed to drive. The `petalinux-build` command is also responsible for building a generic root file system for the PetaLinux distribution as per the kernel configuration and the metadata present in the layers specified in the “`bbayers.conf`” file in Listing 4.1. The `petalinux-build` command can do all the tasks mentioned above. However, developers can go step-by-step by specifying options for the command such as:

- “`petalinux-build -c pmufw`” - Build PMU Firmware
- “`petalinux-build -c fsbl`” - Build FSBL
- “`petalinux-build -c u-boot`” - Build U-Boot
- “`petalinux-build -c kernel`” - Build Kernel
- “`petalinux-build -c device-tree`” - Build device-tree
- “`petalinux-build -c rootfs`” - Build root file system

4.2.6 Packaging the Boot Image

The `petalinux-build` command generates various images for the PMUFW, FSBL, U-Boot and Kernel while also generating the device-tree blob (.dtb) file and the root file system archive which is compressed in multiple formats like TAR, CPIO and GZIP. The command also generates a Flattened Image Tree (FIT) image with the name “`image.ub`”. This is an image format readable by the U-Boot and packs the kernel image, the device-tree and the INITRAMFS root file system archive in one package.

In order to obtain uniformity across the different hardware platforms designed by different experiment groups, it was decided to have a U-Boot and a kernel which tries to fulfill the minimum requirements of most of the experiment groups whereas the FSBL and device-tree can vary from experiment group to experiment group. Keeping this focus in mind, the boot process has been made modular and that is the boot process explained in Chapter 5.

Isolating the FSBL and the device-tree from the rest of the Linux boot modules also helps in delineating the roles of the hardware developers and the system administrators. While the hardware developers can be made responsible for generating the FSBL using Vivado using their HDF files, the system administrators can use the unique HDF/BSP provided by hardware developers to generate the device-tree. This allows the system administrators to centrally maintain the BOOT.BIN, U-Boot, kernel and the root file system for all the experiment groups whereas the experiment groups are just responsible for providing the correct FSBL, HDF/BSP and bitstream file. It must be noted that with each change in the HDF file - the FSBL, device-tree and the bitstream files also change and the updated HDF/BSP along with the bitstream file must be made available to the system administrators to ensure error-free booting of Linux on their hardware.

Keeping this in mind, it is important to package the PMUFW, FSBL and U-Boot (optionally the bitstream) in the BOOT.BIN as it is responsible for loading the PMUFW, the FSBL and the U-Boot in the Zynq UltraScale+ PS and programming the programmable logic with the bitstream. This is achieved with the command “`petalinux-package`”.

The script for building and packaging boot images is shown in the listing below:

```
#!/bin/bash

HOME=/home/kmor

PROJECT=PetaLinux

#Change to the PetaLinux installation directory

cd $HOME/petalinux-2019.1/2019.1

#Import the system settings from settings.sh

source settings.sh

#Change to the PetaLinux project directory

cd $PROJECT

#Building all the images for the PetaLinux project

petalinux-build

#Change to the images/linux directory

cd images/linux

#Package the PMUFW, FSBL, U-Boot and bitstream in the BOOT.BIN
petalinux-package --boot --format BIN --fsbl zynqmp_fsbl.elf \
--u-boot u-boot.elf \
--pmufw pmufw.elf \
--fpga system.bit --force
```

Listing 4.7: Script to build and package PetaLinux images

Listing 4.7 shows the script to build and package the PetaLinux images. The script enters the PetaLinux installation directory, sources the PetaLinux environment settings and enters the project directory. Then it proceeds with a full build to build all PetaLinux images and then packages the images into BOOT.BIN. The BOOT.BIN is responsible for loading the PMU Firmware (PMUFW), FSBL, U-Boot and optionally the bitstream in the Zynq UltraScale+ MPSoC.

4.3 Conclusion of the PetaLinux Build Process

The following conclusions can be drawn from the PetaLinux build process:

- PetaLinux Tools are easy to use for beginners looking to develop a Linux distribution for their Xilinx embedded systems and understanding the Linux build process.
- PetaLinux Tools provide a one tool solution to build the PMUFW, FSBL, U-Boot and Linux kernel all of which require separate tools to build each of them.
- All the Xilinx SoC platforms are supported by PetaLinux, reducing the development cycle of Linux development for platforms using these SoCs.
- The PetaLinux Tools can be easily customised to use an external kernel and U-Boot source code.
- The PetaLinux Tools abstracts the complexity associated with the Yocto build process and also reduces the time spent in understanding the build process.

- The PetaLinux Tools might be easy to use, but in case of errors associated with Linux distribution, customisation's and modifications to the build process, configurations and recipes may take significant time.
- The PetaLinux Tools are aimed at Xilinx processors and it is very time-consuming and challenging to modify the PetaLinux build scripts and recipes to build a Linux distribution for non-Xilinx processors.

4.4 Porting the CentOS 8 Kernel for Zynq UltraScale+

PetaLinux Tools download the kernel source code from the Git repository of Xilinx. Each version of PetaLinux downloads the kernel source code from a pre-configured branch in the Git repository. The same procedure applies to the U-Boot source code. While the kernel provided by Xilinx is tailored for the Zynq and Microblaze series of processors, the developers at different groups are recommended not to use the Xilinx kernel and instead use a kernel for which expertise already exists at CERN i.e the CentOS 8 kernel. This is because the CERN IT already has vast experience with the CentOS 8 kernel and they are capable of providing a security-tested kernel to different groups along with secure updates to the CentOS 8 kernel. There are other reasons why the CentOS 8 kernel is preferred over the kernel provided by Xilinx and they are as follows:

1. The kernel version downloaded by the PetaLinux Tools might change with the change in the PetaLinux Tools version [24]. As a result, if system administrators were providing support and maintenance to a few experiment groups for kernel version 4.19 that came with PetaLinux Tools 2019.1, they might have to provide support to some other groups using kernel version 4.20 that comes with PetaLinux Tools 2019.2. The system administrators have to provide support to machines commissioned in CMS for at least 10-15 years. Continuous changes in the kernel version complicates the task for system administrators.
2. CentOS mainstream kernels come with Long-Term support (LTS) and this ensures that security patches, secure software updates and other support can be provided by the CentOS community as well as the CERN IT for a long period of time during which the machines are commissioned in the CMS experiment network.
3. The kernel source obtained from the Git repository of Xilinx is an open source collaboration with many people uploading patches to the repository to improve the code. This code may or may not be tested for security vulnerabilities by the Xilinx developers. Usage of such kernels for the phase-II upgrade hardware could expose the hardware and the experiment networks to security threats from outside of CERN.
4. If such kernel are used, it may necessitate the isolation of such nodes running those kernels to protect the experiment network.
5. Software updates to such systems would be difficult due to the isolation of their systems or due to the restricted access provided to the them.

Keeping the above factors in mind, most of the experiment groups have agreed to the usage of the latest CentOS 8 kernel (in this case 4.18) during the CMS SoC Workshop conducted in June 2019.

4.4.1 Reasons to Port the CentOS 8 Kernel 4.18 for Zynq UltraScale+

The CentOS 8 kernel supports the 64-bit ARM processors like the Zynq UltraScale+. However, the CentOS 8 kernel Image without proper configuration cannot boot as-it-is on the Zynq UltraScale+. This is because the the CentOS 8 kernel needs to be informed about the platform that it is booting upon, the hardware that is interfaced with it and the drivers that are required to boot on that hardware. Each processor has certain peripherals and their drivers that are essential to boot the Linux kernel on the processor. The CentOS 8 kernel 4.18 needs to be configured properly for the Zynq UltraScale+ by having the Kconfig files properly populated with Zynq UltraScale+ specific features which the developers can activate to boot Linux on Zynq UltraScale+. Once the CentOS 8 kernel 4.18 has been properly configured, it would require the Xilinx and Zynq UltraScale+ specific drivers to be ported to the CentOS 8 source code so that they are available when the kernel build begins as per the kernel configuration. Not all the configuration files, features, drivers and Makefiles necessary to boot Linux on Zynq UltraScale+ come with the original CentOS 8 kernel 4.18's source code. Thus, these things had to be added to it and hence the porting of the CentOS 8 kernel 4.18 for Zynq UltraScale+ MPSoC is important.

4.4.2 Fetching and Patching the CentOS 8 Kernel 4.18 Source RPM

Before porting the CentOS 8 kernel 4.18, it should be fetched from the CentOS repository and should be patched with the patches provided in the source RPM package. To achieve this “rpm-utils” and “rpm-build” packages must be present in the CentOS userspace. It must be noted that the kernel 4.18 source rpm cannot be extracted and patched in a system running on a lower kernel version as the userspace might contain older, outdated software packages. As a result, a CentOS 8 distribution was installed on a desktop machine to extract and patch the CentOS 8 kernel 4.18. The kernel should be extracted from the CentOS repository and for this project, it was extracted from the following weblink:

<http://vault.centos.org/8.0.1905/BaseOS/Source/SPackages/>

The extraction and patching of the source RPM results in the kernel source code which can finally be used to port it for Zynq UltraScale+. A detailed script for extracting and patching the CentOS 8 kernel 4.18 has been provided in the Appendix A.

4.4.3 Getting the Right Kernel Configuration

There is a configuration file called the “defconfig” file present in the path “arch/arm64/configs” of the CentOS 8 kernel 4.18 source code. This file contains the default configuration which is necessary to build a CentOS 8 kernel. On the other hand, there is a “xilinx_zynqmp_defconfig” file present in the same path of the Xilinx kernel 4.19 source code. This has the default configuration to boot a kernel on the Zynq UltraScale+ MPSoC. In order to get the right kernel configuration, we need to find differences between the two defconfig files and add the Zynq UltraScale+ specific differences to the defconfig file present in the CentOS 8 kernel. This can be done by simply using the Linux utilities “diff” and “patch” as can be seen in the following script:

```
#Find differences in kernel default configuration
diff -u defconfig xilinx_zynqmp_defconfig > defconfig.patch
#Patch the CentOS 8 default configuration with Zynq UltraScale+ features
```

```
patch -p0 < defconfig.patch
```

Listing 4.8: Script to add Zynq UltraScale+ features to CentOS 8 kernel 4.18 default configuration

Listing 4.8 shows the method to find differences between the two default kernel configurations and patching the Zynq UltraScale+ specific differences to the CentOS 8 kernel 4.18 default configuration. It needs to be investigated which features really need to be added in the default kernel configuration as adding certain features could make the new CentOS 8 kernel 4.18 less secure compared to the default CentOS 8 kernel 4.18. There needs to be a complete understanding of which features are being ported and only the necessary features need to be ported. A simple idea is to add those missing features in CentOS 8 kernel 4.18 default configuration, which are obviously related to Xilinx and Zynq UltraScale+ or the hardware peripherals on the hardware platform that uses Zynq UltraScale+. They can be easily identified by the feature names. A snapshot of differences between the CentOS 8 kernel 4.18 default configuration and Xilinx kernel 4.19's Zynq UltraScale+ default configuration has been presented in the Appendix A.

Without proper kernel configuration and Zynq UltraScale+ features in the kernel configuration, the CentOS 8 kernel 4.18 would not boot on the Zynq UltraScale+. Once the appropriate additions are made to the CentOS 8 kernel 4.18 defconfig file, it has the features which are present in the default configuration of Xilinx kernel source code. It helps in generating a correct “config” file with the help of different Kconfig files in different sub directories of the CentOS 8 kernel 4.18 source code. A correct “config” file helps the Makefile in the kernel source code to build the modules and drivers necessary to boot the kernel on Zynq UltraScale+.

Alternatively, the easiest way to configure a kernel for the Zynq UltraScale+ is to directly use the “xilinx_zynqmp_defconfig” file as this configuration is also used by the PetaLinux Tools and guarantees that a kernel configured with this file would boot on the Zynq UltraScale+. However, this configuration could be vastly different from the CentOS 8 kernel 4.18 or any other mainstream LTS kernel configuration and could have features which expose the kernel to security and functional vulnerabilities which are not desirable. Hence, using the CentOS 8 kernel defconfig file with addition of differences from the “xilinx_zynqmp_defconfig” file is the best way to configure the kernel.

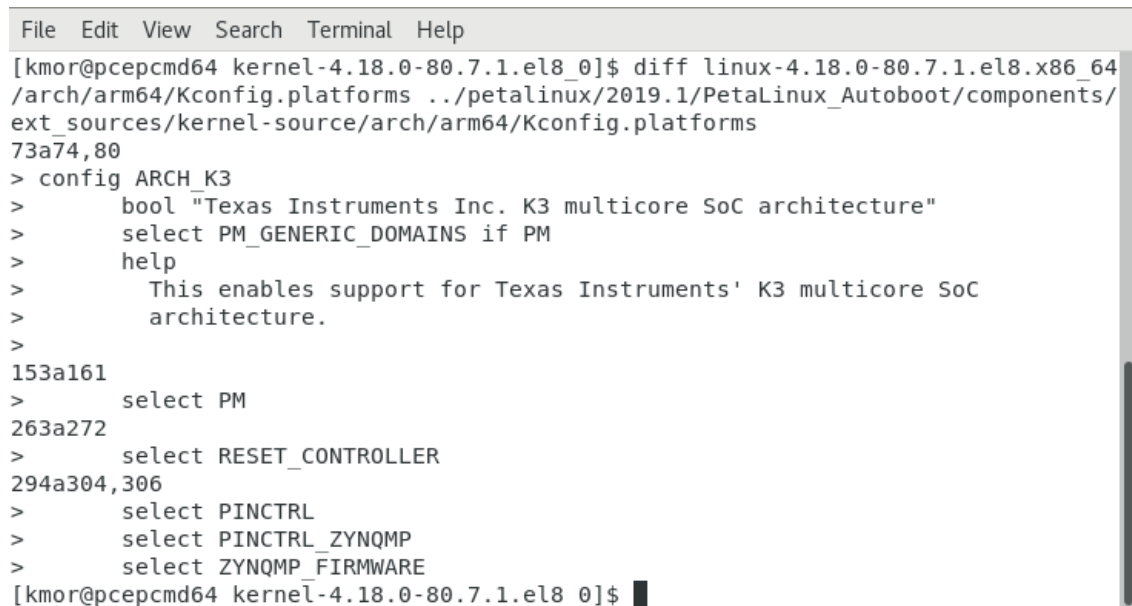
4.4.4 Challenges in Getting the Right Kernel Configuration

As seen with PetaLinux, kernel configuration process is the most crucial process in building a kernel. A wrong configuration can build the kernel with insufficient or deficient support for the peripherals and the services required by the user, stall the boot of the kernel on the Zynq UltraScale+ or even add introduce security vulnerabilities in the system. As a result, a significant time was consumed in getting the right kernel configuration for the CentOS 8 kernel 4.18. For configuring the kernel, “make” was used instead of PetaLinux Tools to speed up the configuration process. In PetaLinux Tools, the kernel configuration process takes long since additional tasks like generation of BitBake appends and configuration files are also carried out along with quality assurance checks along the configuration process. Make was used to generated a kernel config file and a Makefile in the kernel build directory as per the default Kernel configuration (Kconfig) and Makefile present in the central directory of the CentOS 8 kernel 4.18 source code. The Makefile and the Kconfig file in the kernel source code's central directory are instrumental in generating the config and the Makefile needed to build the kernel for a given architecture. They must not be modified.

Keeping this in mind, the kernel build was executed. However, when the kernel began booting on the ZCU102 board, it didn't proceed to mount the root file system and complained of missing

drivers such as the Zynq GPIO, SD card clock, Ethernet clock, Xilinx DMA drivers, Quad-SPI drivers and Cadence Ethernet driver. After going through multiple cycles of addition of driver sources and their dependencies in the kernel source code and testing the images of those kernels, it was established that there is a problem with the config file which is being generated during the kernel configuration. Apart from these missing drivers, other drivers and services had not been activated in the CentOS 8 kernel 4.18 default kernel configuration.

As a result different Kconfig files in the Xilinx and the CentOS 8 kernel 4.18 were compared. It was found out that the Kconfig.platforms file in the “arch/arm64” path of the CentOS 8 kernel has no configuration settings to activate support for the Zynq UltraScale+ processor pins, firmware and peripherals.



```
File Edit View Search Terminal Help
[kmor@pcep64 kernel-4.18.0-80.7.1.el8_0]$ diff linux-4.18.0-80.7.1.el8.x86_64
/arch/arm64/Kconfig.platforms ../petalinux/2019.1/PetaLinux_Autoboot/components/
ext_sources/kernel-source/arch/arm64/Kconfig.platforms
73a74,80
> config ARCH_K3
>     bool "Texas Instruments Inc. K3 multicore SoC architecture"
>     select PM_GENERIC_DOMAINS if PM
>     help
>         This enables support for Texas Instruments' K3 multicore SoC
>         architecture.
>
153a161
>     select PM
263a272
>     select RESET_CONTROLLER
294a304,306
>     select PINCTRL
>     select PINCTRL_ZYNQMP
>     select ZYNQMP_FIRMWARE
[kmor@pcep64 kernel-4.18.0-80.7.1.el8_0]$
```

Figure 4.4: Difference in Kconfig.platforms files of Xilinx kernel 4.19 and CentOS 8 kernel 4.18 source code before changes.

Figure 4.4 shows the difference between the Kconfig.platforms files of the Xilinx kernel 4.19 and CentOS 8 kernel 4.18 source code. It can be seen that the CentOS 8 kernel 4.18’s Kconfig.platforms does not have the options of PINCTRL, PINCTRL_ZYNQMP and ZYNQMP_FIRMWARE features (shown by angle brackets). Absence of these three features in CentOS 8 kernel’s Kconfig.platforms prevent some important Zynq UltraScale+ specific features and drivers from getting activated in the default kernel configuration, leading to a deficient kernel build. Significant time was spent in solving this problem as there was no clear documentation available from CentOS or Xilinx to help in dealing with this problem. Adding those settings to the Kconfig.platforms file in the CentOS 8 kernel source code enabled representation of Zynq UltraScale+ specific driver and feature options in the kernel configuration Menuconfig screen. It also helped in activating the Zynq UltraScale+ specific options in the CentOS 8 kernel 4.18 kernel configuration and ultimately helped in compiling the necessary kernel drivers that boot Linux on the ZCU102 hardware.

4.4.5 Importance of Makefiles and Kconfig Files

Makefiles and Kconfig files are instrumental in configuring and building the Linux kernel. These are also used in PetaLinux Tools and Yocto project explained before. The Kconfig files help in specifying configuration options and features, such that the Menuconfig screen presents those

options to the developer and help him/her configure the Linux kernel properly. Many Kconfig files are present in the kernel source code in different directories. They form a hierarchal structure which is presented to the developer in a simplified manner in the Menuconfig screen. The Makefile is a script which is used to automate certain tasks while configuring and building a kernel source code such that the developers don't have to do them manually every time they build the kernel source. The Makefile to build a kernel source code often comes with the source code and resides in its central directory. It must not be modified. Other Makefiles are present in the kernel source code sub-directories and they specify all the different targets (object files) which need to be built by the GCC compiler as per the kernel configuration provided. These targets essentially specify building of object files of drivers and features that would be included eventually in the kernel. When porting the CentOS 8 kernel 4.18 to Zynq UltraScale+ MPSoC, sometimes the contents of the Kconfig and Makefiles needed to be added from the Xilinx kernel 4.19 source code to their corresponding Kconfig and Makefiles in the CentOS 8 kernel 4.18 source code. At times the whole Kconfig and Makefile needed to be ported from the Xilinx kernel source code to the CentOS 8 kernel 4.18 source code if there was no similar file or no file specifying similar rules/features.

```
# SPDX-License-Identifier: GPL-2.0
# Makefile for Xilinx firmwares

obj-$(CONFIG_ZYNQMP_FIRMWARE) += zynqmp.o zynqmp-ggs.o
obj-$(CONFIG_ZYNQMP_FIRMWARE_DEBUG) += zynqmp-debug.o
obj-$(CONFIG_ZYNQMP_FIRMWARE_SECURE) += zynqmp-secure.o
```

Listing 4.9: Makefile to build Zynq UltraScale+ firmware.

Listing 4.9 shows a Makefile specifying rules for the GCC compiler to build Zynq UltraScale+ firmware if those options are activated in the kernel configuration.

```
# SPDX-License-Identifier: GPL-2.0
# Kconfig for Xilinx firmwares

menu "Zynq MPSoC Firmware Drivers"
    depends on ARCH_ZYNQMP

config ZYNQMP_FIRMWARE
    bool "Enable Xilinx Zynq MPSoC firmware interface"
    select MFD_CORE
    help
        Firmware interface driver is used by different
        drivers to communicate with the firmware for
        various platform management services.
        Say yes to enable ZynqMP firmware interface driver.
        If in doubt, say N.

config ZYNQMP_FIRMWARE_DEBUG
    bool "Enable Xilinx Zynq MPSoC firmware debug APIs"
    depends on ZYNQMP_FIRMWARE && DEBUG_FS
    help
        Say yes to enable ZynqMP firmware interface debug APIs.
        If in doubt, say N.

config ZYNQMP_FIRMWARE_SECURE
    bool "Enable Xilinx Zynq MPSoC secure firmware loading APIs"
    help
        Say yes to enable ZynqMP secure firmware loading APIs.
        In doubt, say N

endmenu
```

Listing 4.10: Kconfig file to specify Zynq UltraScale+ firmware options.

Listing 4.10 shows a Kconfig file that offers developers the option to activate Zynq UltraScale+ firmware in kernel configuration. Developers don't see these options until they add the three

options shown in Figure 4.4 to CentOS 8 kernel 4.18 Kconfig.platforms file and activate the Zynq UltraScale+ platform in the kernel configuration.

In simple words, the kernel Menuconfig screen parses the central and other Kconfig files in the kernel source code sub-directories help the developers in configuring the kernel whereas the central Makefile along with other Makefiles in the kernel source-code sub-directories help the developers in building the kernel as per the kernel configuration provided. The central Makefile and Kconfig files sit at the top of the build and configuration hierarchy respectively.

4.4.6 Porting the Drivers from Xilinx Kernel 4.19 to CentOS 8 Kernel 4.18

The CentOS 8 mainstream kernel 4.18 supports the 64-bit ARM architectures but has little support for the drivers required to boot the Linux kernel on a Zynq UltraScale+ ARM processing system. Thus, the drivers need to be ported from the Xilinx kernel 4.19 to CentOS 8 kernel 4.18.

```
[ 3.962612] zynq-gpio ff0a0000.gpio: input clock not found.
[ 3.968220] xilinx-zynqmp-dma fd500000.dma: main clock not found.
[ 3.974197] xilinx-zynqmp-dma fd510000.dma: main clock not found.
[ 3.980252] xilinx-zynqmp-dma fd520000.dma: main clock not found.
[ 3.986308] xilinx-zynqmp-dma fd530000.dma: main clock not found.
[ 3.992364] xilinx-zynqmp-dma fd540000.dma: main clock not found.
[ 3.998419] xilinx-zynqmp-dma fd550000.dma: main clock not found.
[ 4.004473] xilinx-zynqmp-dma fd560000.dma: main clock not found.
[ 4.010530] xilinx-zynqmp-dma fd570000.dma: main clock not found.
[ 4.016585] xilinx-zynqmp-dma ffa80000.dma: main clock not found.
[ 4.022639] xilinx-zynqmp-dma ffa90000.dma: main clock not found.
[ 4.028695] xilinx-zynqmp-dma ffaa0000.dma: main clock not found.
[ 4.034751] xilinx-zynqmp-dma ffab0000.dma: main clock not found.
[ 4.040807] xilinx-zynqmp-dma ffac0000.dma: main clock not found.
[ 4.046862] xilinx-zynqmp-dma ffad0000.dma: main clock not found.
[ 4.052920] xilinx-zynqmp-dma ffae0000.dma: main clock not found.
[ 4.058975] xilinx-zynqmp-dma ffaf0000.dma: main clock not found.
[ 4.065187] zynqmp-qspi ff0f0000.spi: pclk clock not found.
[ 4.070672] xilinx_can ff070000.can: Device clock not found.
[ 4.076296] macb ff0e0000.ethernet: failed to get macb_clk (4294966779)
[ 4.082964] cdns-i2c ff020000.i2c: input clock not found.
[ 4.088272] cdns-i2c ff030000.i2c: input clock not found.
[ 4.093628] cdns-wdt fd4d0000.watchdog: input clock not found
[ 4.099346] cdns-wdt ff150000.watchdog: input clock not found
[ 4.105112] sdhci-arasan ff170000.mmc: clk ahb clock not found.
```

Figure 4.5: Missing Xilinx and Zynq specific drivers in CentOS 8 kernel 4.18.

Figure 4.5 shows the bootlog of a failed CentOS 8 kernel boot on the ZCU102 board. It shows which drivers are missing and which drivers need to be ported. However, apart from the drivers that are being shown to be missing in the failed kernel bootlog, there could be many other drivers that are missing in the CentOS 8 kernel 4.18 and they need to be ported as well.

In order to find the missing drivers, one can search for the missing drivers in the source code directory of the Xilinx kernel 4.19 or can simply pass Zynq UltraScale+ specific features in the default configuration such that it boots Linux on Zynq UltraScale+ as shown in the previous sub-section. During this project, the second approach was adopted. Passing the correct kernel

configuration with Zynq UltraScale+ features helped the make tool to prompt an error when a certain driver or source code to be built was missing and that was added during compile time as the build of the kernel progressed. This ensured that only minimal changes are made to the kernel without affecting or breaking the directory hierarchy of the CentOS 8 kernel 4.18.

Replacement of Driver Kconfig and Makefiles

To ensure that all Xilinx and Zynq UltraScale+ related drivers are ported to the CentOS 8 kernel 4.18 from the Xilinx kernel 4.19, the following steps were taken:

- Only the necessary source files were added to the the CentOS 8 kernel 4.18 source code. These files are source code files which are present in the Xilinx kernel 4.19 but not in CentOS 8 kernel 4.18. These files were informed about whenever the make tool prompted a missing target error during the kernel code compilation.
- Any missing dependency in terms of missing function declarations/definitions, variable declarations/definitions and macro declarations/definitions were treated on case-by-case basis. Only the parts of the missing code were added to the header or source files of the CentOS 8 kernel 4.18 source code, from where the error was originating. These pieces of code were also obtained from the Xilinx kernel 4.19 source code. These errors were also pointed by the make tool during the compilation process.

To efficiently identify all the missing drivers specific to Zynq UltraScale+ and Xilinx, it was decided to identify all the relevant Makefile and Kconfig files in the Xilinx kernel 4.19 source code that are related to these particular driver files. It was done since it would be important to add them or their contents to the right Makefile and Kconfig file to the right sub-directory of the CentOS 8 kernel 4.18 source code when the kernel is built. This served two purposes, which were as follows:

- It helped present the options in kernel Menuconfig to activate drivers during the CentOS 8 kernel configuration process.
- Helped the make tool point to missing target files and missing code dependencies during the compilation process. This helped in adding the missing file and their missing code dependencies to the CentOS 8 kernel 4.18 source code during the build process.

In this way, only the missing drivers, Makefiles and Kconfig files where added to the CentOS 8 kernel 4.18 source code to port the necessary drivers and features to boot Linux on the Zynq UltraScale+.

4.4.7 Building the CentOS 8 Kernel 4.18

GCC and make were used to build the CentOS 8 kernel 4.18 Image. This is similar to the underlying tool-chain implemented by PetaLinux Tools and Yocto Project to build the U-Boot and kernel images. The compilation of the kernel source code for 64-bit ARM processors has the same tool dependencies as mentioned in the PetaLinux 2019.1 User Guide [24] and in Listing 4.4. Additionally, the GCC compiler for 64-bit ARM Linux (gcc-aarch64-linux-gnu) should be installed on x86/x86_64 machines to ensure cross-compilation for the 64-bit ARM processors. Alternatively, the compilation can also be carried out on a machine using a 64-bit ARM processor. GCC version

4.8.5 was used for compilation of CentOS 8 kernel 4.18 on an x86 PC. However, higher versions of GCC can also be used since PetaLinux uses GCC 8.2.0. GCC version 4.8.5 was used as it is the highest GCC version for 64-bit ARM Linux available for CentOS 7 on which the build was carried out. The build was also carried out on the Xilinx Zynq UltraScale+ in a 64-bit CentOS 8 environment and also with PetaLinux Tools, both with the help of GCC 8.2.0. This was done to verify if the build is dependent on any particular GCC version. All the three build methods mentioned above were successful. The build with make and GCC for ARM is carried out by executing the following command in the directory where the kernel configuration file has been written:

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image dtbs
```

Listing 4.11: Command to build the CentOS 8 kernel 4.18.

Listing 4.11 shows the CentOS 8 kernel 4.18 build command using make and GCC compiler for 64-bit ARM processors. “Image” option directs the GCC to package the kernel image to the Image format and the “dtbs” option directs the kernel to build the device-tree binary as per the device-tree source files present in the “/arch/arm64/boot/dts/xilinx” path of the CentOS 8 kernel 4.18.

The same kernel can be compiled and built in the PetaLinux Tools by using the “ext-local-src” option in the Menuconfig window that is generated by the “petalinux-config” command.

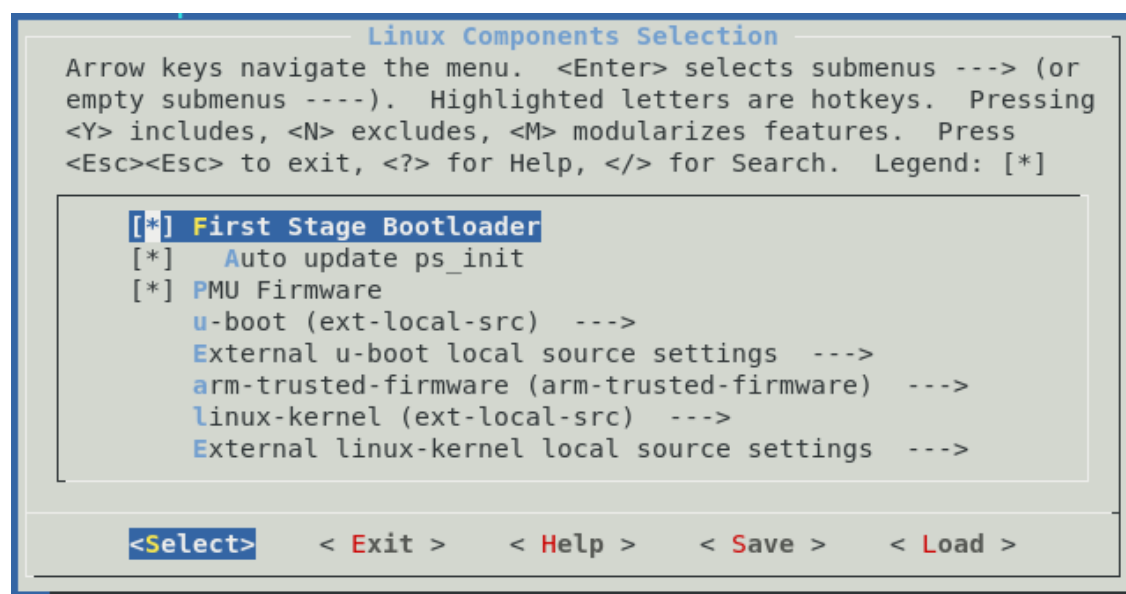


Figure 4.6: PetaLinux Menuconfig to specify source of the U-Boot and the kernel

Figure 4.6 shows the Menuconfig screen for the PetaLinux Tools to choose source of the U-Boot and the kernel. Here developers can point to a local repository, a remote repository on the web or the default Xilinx source options on GitHub.

4.4.8 Debugging the Build Process

Once the Makefile and Kconfig files have been ported and the build of the kernel has begun executing, the make tool and the GCC compiler prompt an error in the following cases:

- When the target .c or .h file is missing.
- Certain data structures or properties within data structures have not been declared or defined.
- Certain variables, macros or functions have not been declared or defined.

In order to debug these errors, careful investigation was carried out to find the origin of the error. Only the data structures, data structure elements, variable or functions that have not been defined or declared were added to the CentOS 4.18 kernel source code from the Xilinx Linux kernel 4.19. In this way, all the drivers were ported successfully to the CentOS 8 kernel. If any particular file was reported missing by the make tool, the file was added at the directory path required by the make tool.

A detailed script of how to configure and build the CentOS 8 kernel 4.18 has been presented in the Appendix A. However, these scripts do not ensure an error free build and the errors mentioned above do occur which require careful addition of the missing codes without affecting the build hierarchy. A patched and built CentOS 8 kernel 4.18 for Zynq UltraScale+ is available at the Gitlab repository weblink specified in Appendix A.

4.5 Lessons from the Porting of the CentOS 8 Kernel 4.18 for Zynq UltraScale+

4.5.1 Lessons from the Kernel Source Extraction and Patching the Source Code

The kernel source RPM for the CentOS 8 kernel 4.18 should be extracted in the userspace environment having the up-to-date “rpm-utils” and “rpmbuild” packages. Older versions from the CentOS 7 environment do not help in extracting the kernel source RPMs.

4.5.2 Lessons for Setting up the Build Tool-Chain

Make and GCC are two important tools necessary for configuring and compiling the CentOS 8 kernel. However, there are many other software dependencies which the developer might have to install in the userspace to ensure successful configuration and compilations. Developers may not be aware of the exact tool-chain setup and could be prompted with errors by make or GCC during the configuration and compilation process. A good way to know all the build dependencies can be to find the build dependencies mentioned for Yocto or the latest PetaLinux Tools. Another way is to install all kernel build dependencies in a 64-bit ARM CentOS environment with the help of the command “sudo yum-builddep kernel”. This command helps in downloading all the utilities required while building a kernel version same as the kernel version released with the CentOS version in use by the developer. The above mentioned command can be used even in x86 environments but developers may need to download the 64-bit ARM GCC compiler and 64-bit ARM binary utilities (binutils) as an extra to compensate for the lack of those build utilities in the x86 environment.

4.5.3 Lessons from Kernel Configuration

While porting the CentOS 8 kernel 4.18 for a given hardware, the following points must be always remembered while configuring the kernel:

- The defconfig file of the CentOS 8 kernel 4.18 should be used for configuring and building the kernel since it is provided by the CentOS 8 distribution maintainers and represents the most secure and functionally sound kernel configuration. Necessary additions to the defconfig file should be made by identifying the necessary differences between the defconfig file of CentOS and the defconfig file coming from the official Xilinx Linux kernel.
- The Kconfig.platforms file is very crucial in giving users the options to activate platform specific features. If the options to activate a given platform or a platform's peripherals are missing in the Kconfig.platforms file of the CentOS 8 kernel 4.18, the right kernel configuration would not be generated and the kernel won't boot properly on the given platform. Attempts should be made to identify the missing platform options and should be added to the Kconfig.platforms file of the CentOS 8 kernel 4.18 source code.
- While porting the CentOS 8 kernel 4.18 for a given platform, the Kconfig and Makefiles or their contents pointing to Zynq UltraScale+ specific drivers should be added to the CentOS 8 kernel 4.18 source code or its Kconfig and Makefiles respectively. The Kconfig files allow the Menuconfig screen to present developers with the options to activate Zynq UltraScale+ specific drivers in the kernel configuration. The Makefiles allow the make tool and GCC to identify and build the drivers selected in the kernel configuration.

4.5.4 Lessons from Driver Porting and the Build Process

The following points are important to remember while building the CentOS 8 kernel 4.18:

- PetaLinux Tools and the Yocto Project can be used to compile and build the kernel image, however due to their internal processes of cloning the source code and doing quality-assurance checks on the compiled code, the process of compiling the kernel takes too long. To speed up the compilation process, the make and the GCC compiler are two sufficient tools along with their software dependencies.
- While porting the Xilinx and Zynq UltraScale+ specific drivers and building the CentOS 8 kernel 4.18 images, only the driver source and header files should be added directly to the kernel source-code. There should not be replacement of other files in the CentOS 8 kernel 4.18 source code.
- As the build progresses, the GCC throws errors about missing pieces of code like functions, data structures, variables and macros when the drivers are being compiled. In such instances, the missing pieces of code should be found in the Xilinx kernel source code and these pieces of code should be added to the relevant source or header files of CentOS 8 kernel 4.18. There should not be mass replacement of files from one kernel source to another kernel source as it could give rise to more errors, break the build process and delay kernel development cycle.
- When the make tool complains of missing target files, the target files should be found in the Xilinx kernel source and added to the CentOS 8 kernel source code. Usually, these errors are limited to the "drivers" or the "include" folders in the CentOS 8 kernel source code. It is harmless to add these missing target files expected by the make tool as long as no replacement of files is being done. This is because replacement of files could give rise to more errors and break the building of the kernel images.

4.6 Building a CentOS 8 Root File System

4.6.1 Reason for a CentOS 8 Root File System

Even though PetaLinux distribution creates a generic root file system, it is not recommended for the experiment groups. Additionally, the usage of a unique root file system not used by other groups would mean a separate NFS file server for the experiment group which would require special maintenance. It has several drawbacks such as:

1. The PetaLinux root file system may not contain updated software packages which are necessary for the functioning of certain applications that are unique to the group.
2. Software packages in the PetaLinux root file system may introduce security vulnerabilities through the packages that are present in the root file system and have not been tested and verified by the CERN IT and system administrators.
3. Due to the above mentioned security vulnerability, CERN IT or CMS administrators may provide them with restricted or isolated networks to secure the rest of the experiment network.

Given the security concerns mentioned above and threat from external cyber-attacks on a publicly funded organisation like CERN, it is recommended to have a centrally managed single root file system which can be managed by the CERN IT and system administrators. CentOS also has a large list of security-tested software packages and applications which are available in its repositories which would present the experiment groups with a wide choice of packages to choose from.

As mentioned previously, there is a lot of knowledge and experience at CERN to maintain the CentOS distribution. The system administrators test and verify all software packages and updates for security vulnerabilities on test nodes before releasing them for CERN wide use. This also allows the experiment groups to have their hardware connected to the all other nodes in the experiment network and within CERN. As a result, a common CentOS 8 root file system has been chosen for this project.

Keeping the above factors in mind, most of the experiment groups have agreed to the usage of a common CentOS root file system during the CMS SoC Workshop conducted in June 2019.

4.6.2 Procedure of Building a CentOS 8 Root File System

This sub-section explains the tasks executed by the root file system build script presented in Appendix A. This sub-section derives its knowledge from the work conducted by my former colleague Panagiotis Papageorgiou who worked under the supervision of Dr. Ralf Spiwoks of the EP-ESE group. His work in turn is derived from the work of Matthias Wittgen of the Stanford Linear Accelerator (SLAC) Laboratory who wrote a script for building a root file system for a given processor architecture with the help of the “DNF” installer in the CentOS environment. Panagiotis Papageorgiou worked on the building of CentOS 7 root file system and the script modified by him for building a CentOS 7 root file system was referenced to create a script for building a CentOS 8 root file system. The script with the appropriate credits can be found in the Appendix A. The script builds the CentOS 8 root file system for 64-bit ARM processors by cross-installing software packages for 64-bit ARM processors on an x86 PC.

4.6.3 The Need for QEMU Emulator

QEMU (Quick EMulator) emulates hardware and creates a virtual hardware environment for developers to quickly test the software for those hardware platforms and architectures, which the developers may not currently have or when the developers want to prevent damage to the hardware while testing. For building the CentOS 8 root file system on an x86 machine, a QEMU for 64-bit ARM architectures is required to cross-install packages for the 64-bit ARM architecture on the x86 desktop. This is because RPM packages, which are required for installation of packages in Red Hat distribution variants like CentOS, might have pre-install and post-install scripts which need to run an executable for 64-bit architectures on x86 machines. The `binfmt_misc` service in CentOS 8 allows processes to call executables for different architectures using emulators like QEMU, provided the configuration is present in the “`etc/binfmt.d`” path of the host root file system as shown in Listing 4.12.

Once the CentOS 8 root file system is built, the CentOS 8 Linux distribution for the Zynq UltraScale+ MPSoC is ready. The PetaLinux Tools were used to build the FSBL, U-Boot and device-tree for the Zynq UltraScale+. The CentOS 8 kernel 4.18 was ported for Zynq UltraScale+ and built using make tools and GCC compiler. With all the important components for booting Linux on Zynq UltraScale+, an automated network boot of CentOS 8 Linux distribution on Zynq UltraScale+ was carried out. This has been explained in Chapter 5.

4.7 Differences between PetaLinux Tools and Yocto Project

The following bullet points describe some qualitative differences between PetaLinux Tools and Yocto Project:

- **Ease of Use:** The PetaLinux Tools are better in terms of the ease of use for developers due to its high-level command line interface. Developers do not have to configure the metadata layers, the tool chain and the build environment by hand when they are using PetaLinux Tools unlike Yocto where they have to setup the different metadata layers, configuration files and sometimes recipes by hand to build their desired distribution.
- **Flexibility:** The PetaLinux Tools are not as flexible as the Yocto Project. If developers want to make a distribution for a non-Xilinx processor using PetaLinux, they have to add the respective metadata layers, necessary recipes and build scripts in the PetaLinux Tools build hierarchy. Despite these changes, it might take a lot of changes to the PetaLinux build system before the correct tool chain can be setup to produce the desired Linux distribution.

On the other hand, developers can just directly use the Yocto project and configure the Yocto Tool chain to produce a Linux distribution for that processor in the same amount of time.

- **Version Conflicts:** The PetaLinux Tools expect the Hardware Description File (HDF) or the Board Support Package (BSP) to be provided from the Xilinx Vivado version mirroring the PetaLinux Tools version. This is highly inflexible as changes are required to be made in the metadata recipes and PetaLinux build scripts to ensure that a correct FSBL and device-tree is generated based on the hardware design provided from a new Vivado version.

The Yocto Project does not have any such version restrictions and it generates a working Linux distribution as per the hardware design or board support package provided to it. Similarly, commands from older PetaLinux version sometimes are not compatible in the newer PetaLinux versions [24].

- **Transparency:** The PetaLinux Tools are user friendly but not very transparent. There are many scripts and tasks being executed behind the simple user-interface commands and in order to debug a problem associated with the Linux boot components build or execution, developers have to dig deep to pinpoint where the problem is originating from. This is very time consuming. This was experienced while fixing the U-Boot processing of DHCP response which is explained in Chapter 5.

The Yocto Project on the other hand is far more transparent as it is user configured and users are far more equipped to pin-point the source of the problem than they would be while using PetaLinux Tools. It is also quicker to find source of the problem, since there is a steep learning curve associated with the Yocto Project after which the developers are well-trained to quickly point the source of the problem.

- **Learning Curve:** As mentioned above, the Yocto Project has a steeper learning curve compared to the PetaLinux Tools as far as basic usage is concerned. However, to gain mastery in understanding of both the tool chains, similar efforts and time is required.

As a conclusion of these differences, PetaLinux Tools can be recommended for beginners wanting to know more about the Yocto Project and also for developers developing Linux specifically for Xilinx platforms.

The Yocto Project on the other hand is a bit more complex and has a steep learning curve. However, it is recommended for developers looking to develop a Linux distribution for processors apart from and including Xilinx and also wanting to gain a mastery in developing, testing and debugging the Linux distributions.

4.8 Summary

In this chapter, the important components of the Yocto Project were explained. The process to build a Linux distribution with PetaLinux Tools has been explained in detail to the reader. The process of porting a CentOS 8 kernel 4.18 for the Xilinx Zynq UltraScale+ and the lessons associated with porting of the kernel have been explained to reader in detail. The process of building a CentOS 8 root file system and the challenges associated with it have been presented to the reader. At the end of the chapter, the PetaLinux Tools and Yocto Project have been compared for better understanding of the reader.

Chapter 5

The Embedded Linux Network Boot

This chapter focuses on the fully automated network boot of Embedded Linux on the Xilinx Zynq UltraScale+ MPSoC. In this chapter, the infrastructure and the procedure required to execute a fully automated network boot have been explained in detail.

5.1 Important Stages of the Embedded Linux Network Boot

The basic principles of booting Linux apply to the network boot as well. However, there are some additional stages associated with the embedded Linux network boot that were executed or examined for this thesis project which need to be presented to the reader. These are as follows:

- Acquiring IP address and network information from the DHCP server.
- Downloading the U-Boot environment, device-tree blob and kernel image from the TFTP server.
- Modification of the Ethernet MAC address in the environment, EEPROM and device-tree.
- Acquisition of the Ethernet MAC address by the U-Boot and Linux Kernel.
- Mounting the root file system from the NFS server.

5.2 Acquisition of Network Information from the DHCP Server

Enabling the Ethernet IP core in the Zynq UltraScale+ MPSoC is an essential part of the design and for successful execution of the Linux network boot. The Ethernet connection of the Zynq UltraScale+ MPSoC helps the U-Boot in downloading important files from the TFTP server, helps the kernel in acquiring network information from the DHCP server and mounting the root file system from the NFS server. However, before all these tasks are executed, it is important that the U-Boot acquires an IP address for the Ethernet interface and that is possible if there is

a DHCP server in the network which is assigning network configuration to different nodes in the network. There is no DHCP server configured by PetaLinux Tools by default. As a result, a local DHCP server was configured on a TTL NUC5 desktop (see Figure 3.2 and Chapter 6, section 6.1.1) for serving the ZCU102 board.

DHCP stands for Dynamic Host Configuration Protocol. It is defined by the RFC 1541 standard [28]. DHCP can be used for both IPv4 and IPv6 [28]. The DHCP server is used to provide network configuration to the clients in a big network. The DHCP server is deployed in large networks like the CMS data acquisition network where multiple devices need to be configured dynamically but can also be used in small sized local networks as it has been used for testing during this project. DHCP assigns static or dynamic IP address and other network configuration parameters (such as TFTP server IP, NFS root directory, NFS server IP, boot file name) to the devices. In the absence of a DHCP server, system administrators have to configure each device manually, which can be challenging and time consuming in huge networks such as the CMS data acquisition network. Additionally, an automated Linux network boot without the DHCP server can be difficult to implement.

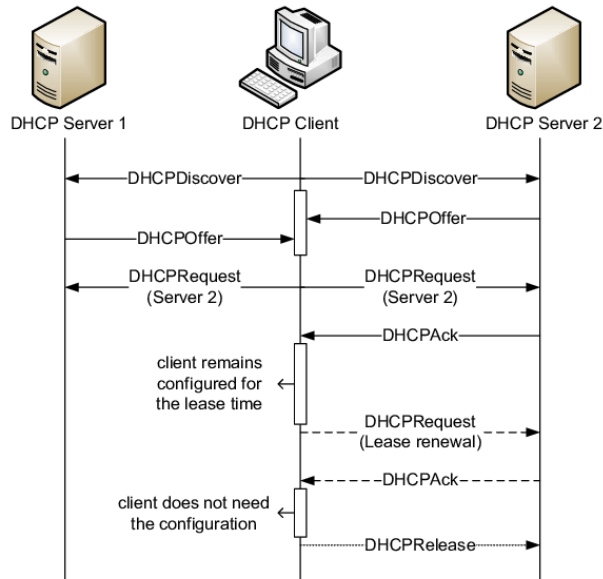


Figure 5.1: DHCP client-server interaction [4]

Figure 5.1 shows DHCP client-server interactions. The client first makes a server discovery request and the closest, available DHCP server responds. The DHCP client then makes a DHCP request for network configuration and the DHCP server acknowledges the request by providing network configuration. The client remains configured for the lease time before requesting for a renewal of IP address assignment.

The DHCP server configuration can be such that a static IP is assigned to the device based on its MAC address each time or an IP address can be offered from the range or pool of IP addresses defined by the system administrator while configuring the server [28]. Additionally, a lease time and other parameters such as boot filename, NFS server root filepath, NFS server IP, TFTP server IP maybe assigned to the client provided they are defined in the DHCP server configuration. More can be found out about the DHCP protocol by reading through the RFC 1541 standard [28].

In our test cases the DHCP server has been configured to assign the Zynq UltraScale+ MPSoC Ethernet interface with an IP address, assign the U-Boot environment variables for the boot

filename and the TFTP server IP address. The DHCP is also configured to inform the U-Boot about the kernel boot arguments, the NFS server IP address and NFS root filepath. The DHCP server configuration has been presented in Appendix B.

5.3 Download Boot Files from the TFTP server

During this project, the TFTP protocol is used to download the boot files from the TFTP server. TFTP stands for Trivial File Transfer Protocol and it is defined by the RFC 1350 standard [29]. It is a simple protocol and users cannot add, delete, modify the contents of the server as it is a simple protocol and does not have features like user authentication. TFTP uses UDP as its default transport protocol and the port 69 for communication [29] which is the configuration that has been maintained for this project. More can be read about TFTP by reading through the RFC1350 standard [29].

A TFTP transfer is initiated by the client issuing a request to read or write a particular file on the server. Once the server has granted the request, the file is sent in fixed blocks sizes of 512 bytes [29]. Each block transfer requires acknowledgment of reception before the retransmission of the next block of data. If a packet gets lost in the network, the client timeouts and retransmits the last packet that it received in order to prompt the sender to retransmit the packet that was lost. A data packet of less than 512 bytes at the end signals termination of transaction. Both the server and the client are considered senders and receivers.

The U-Boot had to be configured for supporting the TFTP protocol so that boot files could be downloaded from the TFTP server to initiate a Linux boot. For this project, the environment variables “uEnv.txt”, the device-tree blob and the kernel Image were downloaded from the TFTP server to implement a Linux network boot. The PetaLinux Tools does not configure TFTP server and the TFTP server for the ZCU102 client board was configured on a TTL NUC 5 desktop (see Figure 3.2 and Chapter 6, section 6.1.1). Scripts for TFTP server configuration are presented in Appendix B.

5.3.1 Fixing the U-Boot Processing of DHCP Response

In order to fix the issues related to the U-Boot DHCP request, the PetaLinux configuration script “u-boot.bsp.tcl” and the U-Boot source code was thoroughly studied to present readers the source of the problem and the method to solve the problem. A snapshot of the “u-boot.bsp.tcl” script has been presented in Appendix B.

After the FSBL loads the U-Boot into the processor DRAM, the U-Boot must acquire an IP address for its Ethernet interface as well as the TFTP server IP address from the DHCP server.

However, over multiple tests it was noticed that the U-Boot is not able to acquire the TFTP server IP address from the DHCP server, while it was able to obtain other information like the boot file path. This means that the DHCP request was working but the ZCU102 Ethernet interface was not able to acquire the TFTP server IP address. This meant that the Linux network boot is not completely automated and still requires manual intervention, which is not the goal of the project. The Xilinx U-Boot BOOTP source code [30] implements the DHCP communication for the U-Boot and when it was examined in detail it was noticed that the definition of the property CONFIG_BOOTP_SERVERIP in U-Boot configuration was in fact hindering the acquisition of the TFTP server IP address and setting of the “serverip” variable in the U-Boot environment, which points to the TFTP server IP address during TFTP download requests.

Careful investigation of the Xilinx U-Boot documentation [31] and BOOTP source code was done to find that the property `CONFIG_BOOTP_SERVERIP` should be activated in the U-Boot configuration only if the developer has already configured a “serverip” variable pointing to a static TFTP server IP address and wants to use that variable instead of a dynamic TFTP IP address offered by the DHCP server. This cannot work in the CMS data acquisition network as the TFTP server IP address might be changed any time and system administrators at CMS cannot set new “serverip” variable in the U-Boot environment of 1000 Zynq UltraScale+ devices installed in the CMS network. Thus, this problem needed to be solved else no automated network boot could be implemented.

The investigation of the Xilinx U-Boot documentation and BOOTP source code was done only after spending a lot of time in modifying the U-Boot configuration using PetaLinux Tools to enable automatic acquisition of TFTP server IP address. After finding out about the `CONFIG_BOOTP_SERVERIP` property, it was found that there is no option in the PetaLinux Tools’ U-Boot configuration to deactivate this property from the Menuconfig screen. As a result, the setting of the `CONFIG_BOOTP_SERVERIP` was searched for in the PetaLinux Tools’ configuration and build scripts and it was found that it is being activated by the “u-boot.bsp.tcl” script used by PetaLinux Tools to configure the U-Boot.

In order to fix the problem, it was decided that `CONFIG_BOOTP_SERVERIP` should not be set in the “u-boot.bsp.tcl” script which is responsible for helping PetaLinux Tools configure the U-Boot. This is the default TCL script provided by Xilinx in order to configure the U-Boot boot arguments, environment variables and other functional properties. The other way to fix the problem is to modify the BOOTP source code in the Xilinx U-Boot such that it allows setting of the “serverip” environment variable in the U-Boot if `CONFIG_BOOTP_SERVERIP` is defined in the U-Boot configuration. However, that alters the basic behaviour of the BOOTP driver of the U-Boot and defeats the purpose for which the property has been defined. As a solution for the future, Xilinx can provide users with the opportunity to deactivate this property in U-Boot configuration through PetaLinux Tools’ Menuconfig interface.

5.4 Importance of the Ethernet MAC Address

The knowledge of setting, acquisition and modification of the Ethernet MAC address is important for the system administrators at the CMS experiment because they need it for identification of the devices in the CMS experiment network and to facilitate their network configuration and their management in the network. The Ethernet MAC address is also something without which the device cannot obtain network configuration from the DHCP server and communicate with the rest of the network. In the context of the embedded systems and the Linux distribution envisaged for the CMS DAQ Phase-II upgrade, following points regarding Ethernet MAC address are important:

- Ensuring that all the hardware developers agree on one method of MAC address storage (eg. EEPROM).
- The interfacing of the MAC address storage with the Zynq UltraScale+ MPSoC is uniform across different hardware platforms and the representation of this interfacing in the device-tree in terms of their individuals physical addresses is also presented in a uniform manner.
- The method of setting, modifying and acquiring this MAC address should be accepted and followed by every hardware developing group.
- Ethernet MAC address of each individual board in the CMS data acquisition network is unique.

The above mentioned points are important in order to facilitate building of a common Linux distribution for the embedded systems in the CMS network and ensure ease of management for the system administrators. This is especially important since the hardware description file (HDF) and the FSBL would be sourced from the hardware developers. The HDF file is a file generated by Xilinx Vivado Design Suite to describe the hardware designed by the hardware developers and it is essential in generating the correct device-tree for the Linux kernel [24]. It is also important that the hardware developers acquire appropriate MAC address values for their devices from the CERN IT MAC address database so that the MAC address obtained for their device is unique within the CERN network.

5.4.1 Reason to Study the Setting, Modification and Acquisition of Ethernet MAC Address during Linux Boot on Zynq UltraScale+

The documentation provided by Xilinx is not clear about how the Ethernet MAC address is set in the Zynq UltraScale+ hardware, how it is modified and how it is acquired during the Linux boot process. Also the U-Boot and kernel source code documentation does not explain these concepts in detail to help developers and system-administrators understand these process. Since this is an important point of concern for the system administrators at the CMS experiment, considerable time was spent to study the U-Boot and Kernel source code to understand the procedures for setting, modifying and acquiring the Ethernet MAC address during the Linux boot on Zynq UltraScale+. There was a need expressed by developers and system administrators at CMS to better understand this process. Hence, the study of setting, updating and acquiring the Ethernet MAC address during Linux boot on Zynq UltraScale+ was done so that developers and system administrators in the future do not have to spend time in understanding these processes.

5.4.2 Default MAC Address Configuration for Zynq UltraScale+

For this project, the BSP of Xilinx ZCU102 Evaluation Kit was used to generate the U-Boot configuration, kernel configuration, the device-tree blob and the First Stage Bootloader. As per the device-tree generated with the help of this BSP, the ZCU102 board has means to store the Ethernet MAC address in the EEPROM chip as shown by the EEPROM node in the device-tree in Listing 3.1.

On the other hand, the “petalinux-config” command explained in the previous chapter allows the user to specify the Ethernet MAC address in U-Boot environment and along with the primary choice of the Ethernet interface in the Menuconfig screen as seen in Figure 4.2. When configuring and compiling the U-Boot independently using make and GCC, Ethernet MAC address environment variable can be specified in the Menuconfig screen during the U-Boot configuration process which is similar to the kernel configuration process. Setting of the MAC address in the U-Boot environment does not automatically set the Ethernet MAC address in the EEPROM of the hardware platform in use. Users can write U-Boot scripts to modify the Ethernet MAC address in the EEPROM and device-tree, if required (explained in section 5.6).

5.5 Setting of the MAC address in the Ethernet Hardware

The process of setting the MAC address was studied thoroughly in detail by going through the U-Boot Ethernet source code [32]. To help hardware developers and system administrators gain an easier understanding of the process, a flowchart was developed, which is a contribution of the

thesis and can be seen in Appendix B.

When the FSBL loads in the U-Boot in the processor DRAM of the Zynq UltraScale+ MPSoC and the U-Boot starts executing, it imports the environment variables from the SPI flash memory in to the U-Boot environment. Once the environment has been imported, the following steps are performed by the U-Boot to set the MAC address in the Ethernet hardware:

1. Check if the variable “ethaddr” is defined in the U-Boot environment and if it is defined, compare its value with the MAC address in the I2C EEPROM device. If the values are different, print a warning that the MAC address values are different and continue using the value in “ethaddr”. If the values are similar, continue using the value in “ethaddr” without a warning message.
2. If the variable “ethaddr” is not defined in the environment, check if the MAC address is defined in the I2C EEPROM device. If it has been defined in the EEPROM device, set the value of “ethaddr” to the value present in the EEPROM device and continue using the value in “ethaddr” variable.
3. If the MAC address is not present in either the environment or the I2C EEPROM device, check if CONFIG_NET_RANDOM_ETHADDR has been defined in U-Boot configuration and if it has been defined, generate a random MAC address and assign it to the “ethaddr” variable. If the property has not been defined, print a warning that MAC address is not set.
4. Once the value in the variable “ethaddr” is set, check if the variable “ethmacskip” has been set in the U-Boot environment. If it has been set, the MAC address value is not written to the Ethernet hardware so long as the variable is set.
5. If “ethmacskip” is not set, check if the MAC address value is valid (not all zeros, not a multicast address and not FF:FF:FF:FF:FF:FF). If the value is invalid, print a warning that the MAC address value is illegal and the MAC address is not written to the hardware.
6. If the MAC address value is valid, the value is written to the Ethernet hardware.

This procedure has been outlined in a flowchart that can be seen in Appendix B.

5.6 The U-Boot Environment “uEnv.txt” File

The U-Boot HUSH environment is similar to the BASH environment present in Linux systems and it has its own environment variables which assist the boot process and also helps in debugging the Linux boot process. The U-Boot supports loading and modification of U-Boot environment during run-time with the help of a uEnv.txt file. This file is generated by PetaLinux Tools while building the U-Boot. Usually this file is kept on SD card to important some environment variables required by the developers. This file is not a static file and it can contain additional environment variables specified by the developers. This file also acts as a model file for developers to write HUSH shell scripts and environment variables for future use.

The U-Boot environment variables are loaded by the U-Boot in the SPI Flash memory when the FSBL loads the U-Boot in the processor DRAM. These environment variables are essentially HUSH shell scripts which utilise the U-Boot binary utilities and drivers to fulfill certain tasks. Some of these environment variables are configured by default during the U-Boot configuration and are a part of the U-Boot image. Developers can also specify additional environment variables during the configuration process before compiling the U-Boot. Otherwise, developers using

PetaLinux can specify variables in the “u-boot.bsp.tcl” script to introduce new variables in the U-Boot environment. The default storage for environment variables for the ZCU102 kit is the SPI Flash memory, however, users can specify other storage locations in U-Boot configuration provided they have that type of memory in their hardware.

However, both the above methods are compile time mechanisms. In a network like the CMS data acquisition network, there could be a need to modify or add new variables in the U-Boot environment during run-time for debugging some U-Boot or hardware (ex. wrong MAC address in EEPROM) configuration errors. Otherwise, there could be a need to specify new kernel boot arguments, boot commands and TFTP server file path if the U-Boot configuration is not satisfactory or if a new TFTP server needs to be tested. The uEnv.txt file can then be used to load the variables from a local memory or from the TFTP server in to the U-Boot environment.

System-administrators can do the following configuration tasks with the uEnv.txt file sitting on the TFTP server:

- Remotely configure U-Boot environment variables without re-compiling the U-Boot.
- Modify path to kernel Image, device-tree, BOOT.BIN present on the TFTP server if there is modification in their paths, without changing the SD card contents or U-Boot configuration. (testing of new images or new TFTP server paths on the machine)
- Specify scripts to modify Ethernet MAC address in the environment, EEPROM and in the device-tree, without re-configuring the U-Boot or the device-tree.
- Specify new load addresses in processor memory for kernel Image and device-tree if required.
- Specify “preboot” environment variable to execute some debug sequences before the user can interrupt the U-Boot.
- Specify variables to load/download kernel Image and device-tree from different media, in case of a failed network boot (see future work in Chapter 8).
- Specify different “default_bootcmd” types to execute full SD card boot, partial network boot and full network boot (explained in Chapter 8).
- Specify any other environment variables or scripts that the system-administrators desire.

During this project, this file was kept on the TFTP server. The presence of the “uEnv.txt” file on the TFTP server is crucial for future modifications and customisations of the network boot, which is not so easy when storing the uEnv.txt file on the local board storage. The uEnv.txt file on local storage like SD card is static in the sense that it cannot be modified immediately with ease when the U-Boot environment or the hardware like EEPROM are not configured properly. The placing of uEnv.txt file on the TFTP server made it easy for dynamically loading new environment variables in the U-Boot environment, without re-configuring or rebuilding the U-Boot. This setting helped immensely in testing different boot methods and modifying the Ethernet MAC address in EEPROM and device-tree on-the-go which helped in investigating the setting, acquiring and modifying of the Ethernet MAC address during the Linux boot on Zynq UltraScale+. The uEnv.txt file on the TFTP server was also helpful in pointing to test kernel Images, device-tree blobs and TFTP servers without the need to reconfigure the DHCP server and the information that it offered. The uEnv.txt file sitting on the TFTP server is the most flexible way to configure U-Boot behaviour and some of the underlying hardware properties (eg. MAC address) without rebuilding the U-Boot. The uEnv.txt file on the TFTP server is extremely useful for the CMS system administrators due to its versatile nature and this configuration is an important contribution of this master thesis.

5.6.1 The Variables in the “uEnv.txt” File

In this section, the variables present in the “uEnv.txt” file during the course of this project are elaborated upon. This file is not a static file and system-administrators can make modifications to the file on TFTP server even when the embedded systems are deployed in the CMS DAQ network.

The following variables are essential to execute a completely automated network Linux network boot on Zynq UltraScale+:

```
#Address to load the kernel Image
netstart=0x1000b000

#Address to load the device-tree blob
clobstart=0x10000000

#Address to load the uEnv.txt file
loadbootenv_addr=0x00100000

#MAC Address
mac="08:00:30:f4:03:37"

#Path to the bootfile on TFTP server (in this case uEnv.txt, can be kernel Image or
the device-tree as well)

bootfile=zcu102/uEnv.txt

#Variable to specify whether DHCP request downloads the bootfile or not. In this
case, it does not.
autoload=no

#Path on TFTP server to load the device-tree blob from
dtb.img=zcu102/system.dtb

#Path on TFTP server to load the kernel Image from
kernel.img=zcu102/Image

#Path on TFTP server to load the uEnv.txt file from
bootenv=zcu102/uEnv.txt

#Variable to specify the tasks executed before the U-Boot begins booting
preboot=echo U-BOOT for xilinx-zcu102-2019.1;setenv autoload no;echo U-BOOT for
CERN CMS;setenv ethmacskip;setenv ethlmacskip

#Variable to select the I2C bus connecting to the EEPROM storing the Ethernet MAC
address and modifying the 6-bytes of MAC address
update_ethaddr=i2c dev 5;i2c mw 54 20.1 08;i2c mw 54 21.1 00;i2c mw 54 22.1 30;i2c
mw 54 23.1 f4;i2c mw 54 24.1 03;i2c mw 54 25.1 37

#Variable to download the device-tree blob from the TFTP server
download_fdt=tftpboot ${clobstart} ${dtb.img};

#Variable to download the kernel Image from the TFTP server
download_kernel=tftpboot ${netstart} ${kernel.img};

#Variable to update the Ethernet MAC address in the device-tree
update_fdt=fdt addr ${clobstart};fdt set ethernet0 local-mac-address ${mac}

#Variable to load the kernel Image from the SD card to processor DRAM
cp_kernel2ram=mmcinfo && fatload mmc ${sdbootdev} ${netstart} ${kernel.img}

#Variable to load the device-tree blob from the SD card to processor DRAM
cp_dtb2ram=mmcinfo && fatload mmc ${sdbootdev} ${clobstart} ${dtb.img}

#Variable to execute a Linux Boot
netbooti=booti ${netstart} - ${clobstart}
```

```
#Variable to download uEnv.txt from the TFTP server and save it in the SPI flash
memory
tftpimportbootenv=echo Importing environment from TFTP Server; tftpboot ${
loadbootenv_addr} ${bootenv}; env import -t ${loadbootenv_addr} $filesize

#Variable to execute a fully automated network boot (depending on the kernel
configuration and boot arguments.)

default_bootcmd=dhcp;run tftpimportbootenv;run download_fdt;run download_kernel;run
netbooti

#Variable to execute a full SD card or partial network boot. Uncomment the
following line to execute either of the boots (depending on kernel
configuration and boot arguments.)

#default_bootcmd=run tftpimportbootenv;run cp_dtb2ram;run cp_kernel2ram; run
netbooti
```

Listing 5.1: The uEnv.txt file on the TFTP server

Listing 5.1 presents different variables/scripts used during the course of the project. Their role has been explained below:

- **netstart, clobstart and loadbootenv_addr:** These variables specify the addresses in the processor memory where the device-tree blob, kernel Image and the uEnv.txt file are downloaded by the U-Boot.
- **mac:** This variable specifies the MAC address that would be used to replace the MAC address in the device-tree if required. This variable was defined and used during the investigation of how U-Boot interacts with the device-tree and the Ethernet MAC address stored in it. However, after conclusion of the investigation, the need to use this variable was not felt. The explanation of interaction between U-Boot, MAC address and device-tree is presented in section 5.7.
- **autoload:** This is a default U-Boot environment variable which decides whether or not the DHCP request automatically downloads the boot file specified by the DHCP server configuration from the TFTP server. The value of this variable is required by the U-Boot source code while processing DHCP requests.
- **dtb_img, kernel_img and bootenv:** These variables specify the path to the device-tree blob, the kernel Image and the uEnv.txt file's location respectively on the TFTP server.
- **preboot:** This variable command is the default command executed by U-Boot before the user is allowed to interrupt the boot process. The command can be modified to suit the user needs. In our uEnv.txt file the preboot command is used to print a few messages on the terminal and remove certain environment variables like ethmacskip that might have been set during any previous U-Boot session. Since U-Boot saves the updated environment in the SPI Flash on user demand and loads this same environment during the next U-Boot session, it is important to remove certain unwanted variables before the booting begins. This is done so that the ethmacskip variable is not set to allow the writing of the MAC address to the Ethernet hardware (used to investigate U-Boot interaction with Ethernet hardware mentioned in section 5.5).

By default, the PetaLinux configures U-Boot such that the preboot command is deleted after its execution. The default preboot configuration given by PetaLinux is “echo U-BOOT for \${hostname};setenv preboot;dhcp”. The “setenv preboot” command deletes the preboot variable. This was modified for this project as the preboot variable was useful in debugging the U-Boot execution before users can interrupt the boot process.

- **update_ethaddr:** This variable is essentially a string of HUSH shell commands and was defined to execute the U-Boot I2C driver “i2c” to modify the MAC address in the I2C EEPROM. As per the device-tree information displayed in Listing 3.1, the variable then updates the Ethernet MAC address at the address pointed by the property “eth-mac@20” in the “eeprom@54” device-tree node. This variable is not a default boot command as it has to be executed by the user if the need arises. This variable is not generated by the U-Boot and is user-defined. It can be used in cases where wrong MAC address has been entered in the EEPROM chip of the hardware. It was used to modify the default boot address that ships with the ZCU102 board and replace it with the MAC address assigned to it by CERN IT. This prevented any redesign of hardware to modify the chip content.
- **download_fdt, download_kernel, cp_kernel2ram, cp_dtb2ram:** download_fdt and download_kernel were defined to download the device-tree blob and the kernel Image from the TFTP server at the specified netstart and clobstart addresses respectively, with the help of the U-Boot binary command “tftpboot”. cp_kernel2ram and cp_dtb2ram are similar commands which perform the operation of loading the kernel Image and the device-tree files respectively from the SD card. Except cp_kernel2ram, all other variables were specially defined for this project to implement an automated network boot.
- **update_fdt:** This variable uses the same principles as the update_mac_address and was defined to modify the MAC address in “ethernet0” device-tree node. This is executed using the U-Boot binary utility “fdt” which is used to view, modify and update the device-tree. This variable was used during the investigation of how U-Boot interacts with the device-tree and the Ethernet MAC address. However, after conclusion of the investigation, the need to use this variable was not felt. The explanation of interaction between U-Boot, MAC address and device-tree is presented in section 5.7.
- **netbooti:** This variable uses the U-Boot binary command “booti” to boot the kernel image format “Image” assisted by the device-tree blob. The processor memory addresses of the device-tree blob and the kernel Image file are passed to the “booti” command. This variable is also a part of our default_bootcmd and was defined to implement an automated network boot.
- **tftpimportbootenv:** This variable was defined to download the uEnv.txt file using the tftpboot command, import the updated variables in the U-Boot environment and save the environment in the SPI Flash memory. This is the first variable executed when the default_bootcmd is executed.
- **default_bootcmd:** This variable is executed by the U-Boot if the user does not interrupt the boot process by initiating a keyboard interrupt. It is pre-defined by U-Boot but can be modified by users. It first makes a DHCP request and then the updated environment is downloaded from the TFTP server which then points to the path of the device-tree blob and the kernel Image and then these boot files are downloaded from the TFTP server and the booting of the kernel image is carried out. Modification of the default_bootcmd was important in implementing the automated network boot.

The default_bootcmd provided by PetaLinux is “run uenvboot; run cp_kernel2ram && bootm \${netstart}”. The “uenvboot” command loads a uEnv.txt file from the SD card if any and the “bootm” command executes the boot of “image.ub” FIT image from the memory address pointed by the “netstart”. Even if the PetaLinux is configured properly and the CentOS 8 kernel is built with the Zynq UltraScale+ drivers, the automated Linux boot cannot happen if the default_bootcmd is not configured properly to automate the network boot. The default boot command can be defined in the “u-boot.bsp.tcl” script of the PetaLinux Tools if the user wants a predefined default_bootcmd. Otherwise the default_bootcmd can be modified even during the U-Boot run-time and imported from the uEnv.txt file on the TFTP server. This helped in testing different boot methods such as full SD card boot, partial network

boot and full network boot without rebuilding the U-Boot. The command for full SD card boot and partial network boot is presented at the bottom of Listing 5.1 (both method load boot files from the SD card).

5.6.2 Using the uEnv.txt File

Once the system administrators and the developers have decided to use a uEnv.txt file on TFTP server to help them in configuring the U-Boot behaviour and the Linux boot, they need to do the following things:

- Usually, the DHCP server offers the kernel Image path on TFTP server to the U-Boot. This information is stored in the variable “filename” in DHCP server configuration (see Appendix B). This information is stored by the U-Boot in the variable “bootfile”. The system-administrators need to modify the path in “filename” to the path of uEnv.txt file on TFTP server. The U-Boot does not care about the file type as it only downloads the uEnv.txt file to a predefined address (loadbootenv_addr, this can also be modified). Thus, if the “filename” points to the uEnv.txt file on TFTP server, “bootfile” in U-Boot environment will also point to the same path on TFTP server. This helps the U-Boot in downloading the environment variables from the TFTP server without any manual intervention during the execution of the default_bootcmd presented in Listing 5.1.

This could raise an error only if the U-Boot is told to treat the uEnv.txt file as a kernel Image file and made to boot it with commands like “bootm” , “booti” (see in Listing 5.1), which expect only kernel Image and device-tree blob files and no other file. Apart from this U-Boot does not raise an error and care should be taken that “autostart” variable is not set in the U-Boot, as it can automatically start the boot process with the uEnv.txt file which will raise an error (usually not set by default in U-Boot environment).

- The default boot command (default_bootcmd) should be modified in the U-Boot configuration such that it always downloads and saves new environment variables from the TFTP server before loading the boot files. Care should be taken that the uEnv.txt file contains the TFTP server path for the kernel Image and device-tree blob which the system-administrators want the U-Boot to download.

Once all of this is set, the U-Boot receives the uEnv.txt file path from the DHCP server, downloads the new environment variables and then proceeds to download the kernel Image and device-tree blob from the paths specified by the uEnv.txt file to boot Linux on the Zynq UltraScale+ MPSoC.

5.7 MAC Address Setting in the Device-Tree by U-Boot

The process of the setting the MAC address by the U-Boot in the device-tree was thoroughly understood by studying the Xilinx U-Boot source codes [33][34][35][36]. This investigation can be understood by the readers by going through the flowchart presented in Appendix B, which is a contribution of this thesis to give better understanding of this process to hardware developers and system-administrators.

During the testing of the Linux network boot process, it was noticed that the kernel does not always acquire the MAC address from the Ethernet hardware registers every time it is booting. In some cases, the MAC address was acquired from the device-tree. On further investigation, it was found that U-Boot is actually modifying the MAC address in the device-tree. Since it is

important for the system administrators to know where the MAC address is being set, modified and acquired from in order to ensure uniformity of hardware behaviour, a considerable time was spent in understanding thoroughly how the U-Boot is modifying the MAC address in the device-tree. The Xilinx U-Boot documentation was not explicit enough in this regard. The following points explain how the U-Boot modifies the MAC address in the device-tree of Zynq UltraScale+ MPSoC:

- The U-Boot checks if the property `CONFIG_OF_LIBFDT` is defined in the U-Boot configuration in order to pass the hardware information to the kernel through the flattened device-tree blob. If it has not been defined, the boot hangs.
- If the property has been defined, the U-Boot checks if there is a node named “aliases” in the device-tree. The “aliases” node holds the information of nodes describing the properties of the hardware such as Ethernet, I2C, SPI and other peripherals activated in the design. If there is no node named “aliases”, the U-Boot does not update the device-tree.
- If there is a node named “aliases”, the U-Boot proceeds to add or replace property called “mac-address” in the Ethernet device-node and assigns the value of the MAC address stored in the “ethaddr” environment variable to the property.
- The U-Boot then adds or replaces a property called “local-mac-address” in the Ethernet device-node and assigns the value of the MAC address stored in the “ethaddr” environment variable to the property.

This procedure can be seen in a flowchart in the Appendix B.

5.8 Acquisition of the MAC address by the Linux Kernel

The process of acquisition of the MAC address by the Linux kernel in the device-tree was thoroughly understood by studying the source codes of the Cadence Ethernet driver [37][38]. The code was studied since Cadence Ethernet hardware interfacing is used for the Zynq UltraScale+ on ZCU102 board and there was not sufficient documentation available from CentOS or Xilinx to understand this process. This investigation can be understood by the readers by going through the flowchart presented in Appendix B, which is a contribution of this thesis to give better understanding of this process to hardware developers and system-administrators.

The following points explain how the kernel acquires the MAC address:

- The “macb” Ethernet driver by Cadence is initialised by the kernel and it checks if the property “mac-address” in the Ethernet device-tree node has a valid MAC address value assigned to it. If it has a valid MAC address value, the kernel uses this MAC address to obtain network information from the DHCP server.
- If there is no property “mac-address” in the Ethernet device-tree node or if it does not have a valid MAC address value assigned to it, the kernel checks if the property “local-mac-address” has a valid MAC address value assigned to it. If it has a valid MAC address value assigned to it, the kernel uses this MAC address to obtain network information from the DHCP server.
- If there is no property “local-mac-address” in the Ethernet device-tree node or if it does not have a valid MAC address value assigned to it, the kernel checks if the property “address” in the Ethernet device-tree node has a valid MAC address value assigned to it. If it has a valid MAC address value assigned to it, the kernel uses this MAC address to obtain network information from the DHCP server.

- If there is no property “address” in the Ethernet device-tree node or if it does not have a valid MAC address value assigned to it, the kernel checks if there is valid MAC address value stored in the Ethernet hardware registers. If there is valid MAC address value stored in the Ethernet hardware registers, the kernel uses this value to obtain network information from the DHCP server.
- If there is no valid MAC address value stored in the Ethernet hardware registers, the kernel generates a random MAC address and tries to obtain network configuration from the DHCP server.

This procedure can be seen in a flowchart in the Appendix B.

5.9 Mounting the Root File System from the NFS server

The kernel before mounting the root file system sends a request to the DHCP server for obtaining the NFS server IP address and the NFS root file system path. Once the information has been obtained from the DHCP server, the root file system is mounted and the “init” process is launched to enter the Linux userspace.

5.9.1 Network File System Server

The Network File System (NFS) is a file system protocol which allows the Linux kernel to mount file systems over a network as if the file system is on a local disk. NFS has multiple versions such as NFSv2, NFSv3 [39] and the latest NFSv4 [40][41]. NFSv3 is most commonly used NFS protocol even today and we have also utilised NFSv3 during the testing of this project. NFSv 2 uses UDP/IP for communication between the client and the server but from NFSv3 the support for TCP/IP was also added. NFS uses Remote Procedure Calls (RPCs) to route requests between clients and servers. The access to the network mounted file system depends on the permissions assigned to the client user (which is defined in the NFS server configuration).

During initiation of NFS request, transport protocol is negotiated between the client and the server. The first protocol that is supported by both the client and the server is the default choice otherwise another transport protocol is chosen on the second attempt. The NFS protocol version is given precedence over the choice of the transport protocol. As a result it may happen that NFSv4 with TCP/IP would be given precedence over NFSv3 configured for UDP/IP (both should support the common transport protocol version). The client must know the NFS root file system path and NFS server’s IP address to request a mount of the root file system. The server must also have a configuration which clearly defines that devices with certain IP addresses, devices having an IP address within a given range or devices with any IP address are allowed to mount the root file system.

The client requests the mount port number from the portmap service running on the NFS server and if that request is granted, the port number is used to see if the mount service is running on the server. Once that has been confirmed, the client then makes a mount request. The NFS server refers to its configuration file, “/etc/exports”, to determine whether the client is allowed to access any of the exported file systems. Once that has been determined, the client is allowed or denied permission to mount the root file system.

The NFS server configuration script has been presented in Appendix B.

5.10 Overview of the Automated Linux Network Boot

In the previous sections, different topics related to the Linux network boot have been explained in detail. The following steps give an overview of the Linux network boot flow on a Zynq UltraScale+ (For the steps taking place before the FSBL is loaded in the On-Chip Memory, please refer to Chapter 3, sub-section 3.5.1):

1. The FSBL initializes the PS hardware, I/O devices, memories and clocks as per the configuration defined in the HDF file provided by the hardware developers. The FSBL then loads the U-Boot in the processor DRAM of the APU.
2. The U-Boot loads the environment from the SPI flash memory, writes the MAC address to the Ethernet hardware and requests IP address assignment and other information such as boot file name, boot file path, kernel boot arguments and TFTP server IP address from the DHCP server.
3. Once the information is obtained from the DHCP server, the U-Boot waits for a pre-configured time to allow user interrupts and in the absence of a user interrupt, executes the `default_bootcmd` environment variable to download a new environment from the TFTP server, save the new environment, download the device-tree blob and the kernel Image file from the TFTP server as per the file paths provided by the new environment.
4. Once the device-tree blob and kernel Image have been downloaded and loaded in the processor memory, the U-Boot hands over the control to the kernel and the “booti” command starts booting the kernel.
5. The kernel with the assistance of the device-tree blob initialises the services and the drivers that have been configured in the kernel configuration and at the end of this process, requests the DHCP server for IP address assignment, NFS server IP address and NFS root file system path.
6. Once this information has been obtained, the kernel makes a mount request to the NFS server and the root file system is mounted. The init process is executed as the first process after mounting of the root file system and various services within the root file system are initialized to start the operating system.
7. At the end of the init process, the user can log-in.

```
Model: ZynqMP ZCU102 Rev1.0
Board: Xilinx ZynqMP
Net: ZYNQ GEM: ff0e0000, phyaddr c, interface rgmii-id
eth0: ethernet@ff0e0000
U-BOOT for xilinx-zcu102-2019_1
U-BOOT for CERN CMS
ethernet@ff0e0000 Waiting for PHY auto negotiation to complete..... done
BOOTP broadcast 1
BOOTP broadcast 2
DHCP client bound to address 128.141.174.218 (251 ms)
Hit any key to stop autoboot: 0
```

Figure 5.2: DHCP request by the U-Boot

Figure 5.2 shows the U-Boot making the DHCP request and acquiring an IP address for the Ethernet interface.


```
[ 5.683573] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[ 5.704487] mmc0: new high speed SDHC card at address 0007
[ 5.710472] mmcblk0: mmc0:0007 SD32G 29.0 GiB
[ 5.717164] mmcblk0: p1 p2
[ 5.949565] [drm] Cannot find any crtc or sizes
[ 8.725902] macb ff0e0000.ethernet eth0: link up (1000/Full)
[ 8.731578] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[ 8.749625] Sending DHCP requests ., OK
[ 8.769356] IP-Config: Got DHCP answer from 192.168.100.1, my address is 128.141.174.218
[ 8.777442] IP-Config: Complete:
[ 8.780666]     device=eth0, hwaddr=08:00:30:f4:03:37, ipaddr=128.141.174.218, mask=255.255.255.0, gw=128.141.174.1
[ 8.791176]     host=128.141.174.218, domain=, nis-domain=(none)
[ 8.797261]     bootserver=128.141.174.229, rootserver=128.141.174.229, rootpath=/home/kmor/nfs/zcu102_vanilla_centos8/,tcp,v3
[ 8.797264]     nameserver0=192.168.100.1
[ 8.812967] cfg80211: Loading compiled-in X.509 certificates for regulatory database
[ 8.952062] cfg80211: Loaded X.509 cert 'sforshee: 00b28ddf47aef9cea7'
[ 8.958595] clk: Not disabling unused clocks
[ 8.962863] ALSA device list:
[ 8.965826]   #0: DisplayPort monitor
[ 8.969824] platform regulatory.0: Direct firmware load for regulatory.db failed with error -2
[ 8.978437] cfg80211: failed to load regulatory.db
[ 8.994577] VFS: Mounted root (nfs filesystem) on device 0:16.
[ 9.000822] devtmpfs: mounted
[ 9.003968] Freeing unused kernel memory: 832K
[ 9.021535] Run /sbin/init as init process
```

Figure 5.4: Mounting the CentOS 8 root file system from the NFS server

Figure 5.4 shows the DHCP request by the kernel and the mounting of the CentOS 8 root file system from the NFS server by the kernel. At the bottom of the figure, it can be seen that the kernel executes the “init” process after mounting the root file system.

```
CentOS Linux 8 (Core)
Kernel 4.18.0 on an aarch64

128 login: root
Password:
Last login: Tue Jan  7 11:42:30 on ttyPS0
[root@128 ~]# ls /
bin    dev    home  lib64  mnt    proc   run    srv    tmp    var
boot  etc    lib   media  opt    root   sbin   sys    usr
```

Figure 5.5: User login into the CentOS 8 root file system

Figure 5.5 shows the login of the user in the CentOS 8 root file system at the end of the init process. The root file system contents are also visible in the figure.

5.11 Summary

In this chapter the readers were informed about the work undertaken in this thesis to fix the issues with the U-Boot’s processing of the DHCP response and how the setting, modification and acquisition of Ethernet MAC address takes place during the Linux boot process on Zynq UltraScale+. This chapter informs the reader about importance of the “uEnv.txt” file for remotely configuring the Linux network boot on the Zynq UltraScale+. This chapter also explains the reader about the DHCP, TFTP and NFS servers and their functioning along with the steps followed during automated Linux network boot.

Chapter 6

Performance of Boot Methods and Root File System Locations

In this chapter, performance of different boot mechanisms and performance of SD card and NFS root file system storage types is presented. A qualitative comparison between the SD card and network storage method for boot files and root file system has also been presented to arrive at a choice of boot mechanism and root file system storage type for the CMS data acquisition network.

The CentOS 8 Linux distribution can be booted on the Xilinx Zynq UltraScale+ MPSoC in multiple ways. During this project, we have tried to boot Linux on the Zynq UltraScale+ by implementing a full SD card boot, a partial network boot and a full network boot. All the methods have their advantages and disadvantages and can be used in the CMS data acquisition network. However, in order to recommend a boot method for the hardware platforms that would be installed as a part of the CMS DAQ Phase-II upgrade, tests were conducted to determine the performance of the different boot mechanisms. These have been presented in the following section.

6.1 Linux Boot Time Tests

For testing the boot performance reported in the following sub-sections, 3 different testing configuration were used. They are as follows:

- **Full SD card boot:** BOOT.BIN, device-tree blob, kernel Image and CentOS 8 root file system are kept on the SD card.
- **Partial network boot:** BOOT.BIN, device-tree blob, kernel Image are kept on the SD card and while the CentOS 8 root file-system is on the NFS server.
- **Full network boot:** BOOT.BIN is kept on the SD card, whereas the device-tree blob, kernel Image are on the TFTP server and the CentOS 8 root file-system is on the NFS server.

6.1.1 Test Infrastructure and Scripts

The following equipment was used for conducting the tests:

- Kingston Canvas React 32 GB (Class 10 UHS-I U3 A1 V30) SD card [42] in UHS (Ultra-High Speed) mode to determine boot timings and read-write speeds of the boot files and root file system.
- TTL TECKNOPACK NUC5 i5MYHE desktop computer [43] as NFS and TFTP server having features such as a 64-bit Intel i5 5th Generation Dual Core Processor with a 250 GB SSD storage, 16 GB RAM and support for 1 Gigabit Ethernet. The desktop has no RAID cards for redundant disks. The uncached disk read speed measured for the TTL NUC5 Solid State Storage Disk (SSD) was around 400 MB/s whereas the sequential write speed of small files (1000 1 MB files) to the disk was about 283 MB/s and that of big files (1 GB) was around 400 MB/s. All these speeds are of course limited by the 1 Gigabit Ethernet interface over which the ZCU102 client communicated with the TFTP and NFS servers running on the device.
- 1 Gigabit Ethernet network to compare with the SD card UHS class read speed of 104 MB/s [44] and also since it is supported by both the ZCU102 board and the NUC5 desktop.

A total of 10 test samples each were taken for the different boot methods and also for different read-write performance tests associated with different root file system types. The tests were done using only the ZCU102 evaluation board with 1 TFTP server, 1 NFS server and 1 DHCP server. The aforementioned servers were located on the TTL NUC5 desktop.

6.1.2 Boot Files Read Speeds

Boot File Read Speeds (Megabytes/second)						
	Kernel Image			Device-Tree		
Statistics	Kingston SD card	1 Gigabit Ethernet	Kingston SD card	1 Gigabit Ethernet	Kingston SD card	1 Gigabit Ethernet
Mean	15.06	10.07	2.16	4.13		
Minimum	14.84	9.78	2.16	4.13		
First Quartile	15.05	10.00	2.16	4.13		
Median	15.08	10.11	2.16	4.13		
Third Quartile	15.08	10.17	2.16	4.13		
Maximum	15.21	10.28	2.16	4.13		
Std. Deviation	0.11	0.14	0.00	0.00		

Table 6.1: Statistics of 10 samples for boot file read speeds from the SD card and the boot network.

Table 6.1 shows the statistics of 10 samples of the boot file read speeds from the SD card and over the 1 Gigabit Ethernet network from the TFTP server. The standard deviation of loading the kernel Image from the Kingston SD card was 0.11 Megabytes/second whereas that of downloading over the 1 Gigabit Ethernet was 0.14 Megabytes/second. Standard deviation to load device-tree from either of the mediums was zero as the read speeds for loading the device-tree were consistent and varied only in their 3rd and 4th decimal places. All the values in Table 6.1 are corrected to fit 2 decimal places.

The size of the device-tree blob (system.dtb) used during the tests is 41.3 KB where as the size of the kernel Image was 18 MB. For this test, the Kingston SD card is used in the UHS mode.

As can be seen from Table 6.1, Ethernet network downloaded the device-tree blob at an average speed of 4.13 MB/s whereas the SD card loaded it at 2.16 MB/s. But the SD card loaded the kernel Image file at an average speed of 15.06 MB/s whereas the Ethernet network loaded

the kernel Image at an average speed of 10.07 MB/s. Both the mediums perform significantly lower compared to their maximum read speed capacity and thus, this test did not give us accurate understanding of the read-write speeds to and from the SD card and the TFTP server.

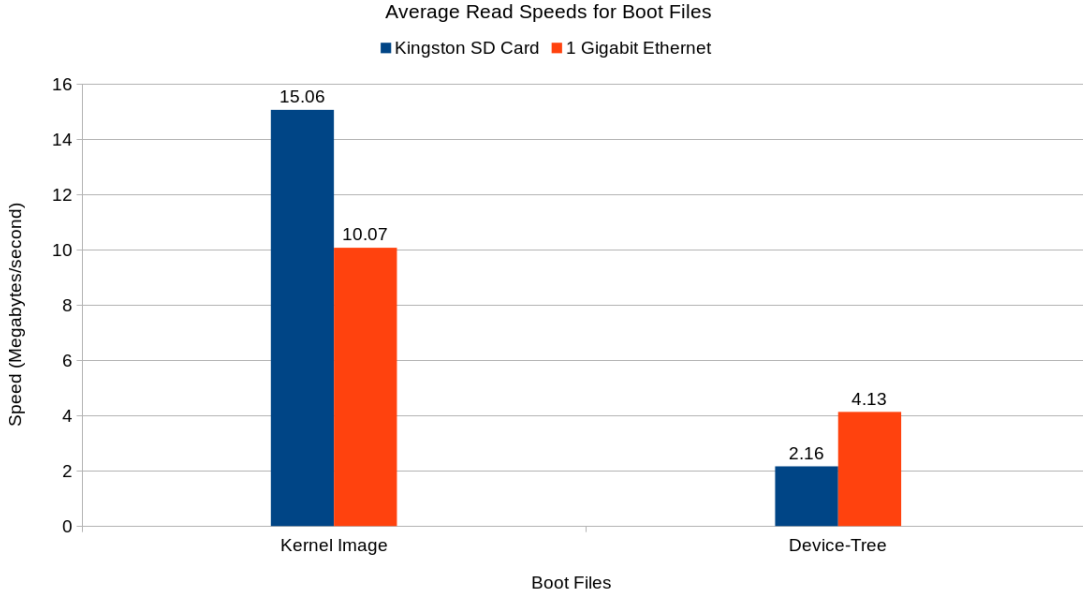


Figure 6.1: Boot File Read Speeds

Figure 6.1 shows average read speeds for reading boot files from the Kingston SD card and over 1 Gigabit Ethernet network. It can be seen that the Kingston SD card is faster than 1 Gigabit Ethernet network in terms of reading the kernel Image but slower than the Ethernet network while loading the device-tree blob file.

6.1.3 Average Boot Times

In order to establish which medium is better when it comes to booting the Linux, the average boot timings were tested with the Kingston SD card and 1 Gigabit Ethernet network in 3 different boot methods: full SD card, partial network boot and full network boot. 10 samples for each boot method were taken.

Statistics for Full SD Card Boot Times (seconds)					
Statistics	Pre-Boot Phase	Kernel Load Phase	Device-Tree Load Phase	Kernel Boot	Userspace
Mean	10.15	1.20	0.02	6.73	16.39
Minimum	10.04	1.19	0.02	6.46	15.95
First Quartile	10.08	1.20	0.02	6.56	16.04
Median	10.16	1.20	0.02	6.61	16.16
Third Quartile	10.23	1.21	0.02	6.85	16.55
Maximum	10.27	1.22	0.02	7.50	17.43
Std. Deviation	0.09	0.01	0.00	0.31	0.52

Table 6.2: Statistics of 10 samples for boot timings for the full SD card boot.

Table 6.2 shows the statistics for the 10 samples of the timing of full SD card boot. In it, statistics of different phases of the Linux boot process are also presented. In Table 6.2, it can be seen that the standard deviation of the time spent in the different boot phases before the kernel boot phase is very low. This is because the values of the time spent in different boot phases before the kernel boot was fairly consistent across the 10 different samples and the variations usually lay in the 3rd or 4th decimal place and beyond. In full SD card boot, files like kernel Image and device-tree are loaded from the SD card before mounting the root file system from the SD card. The standard deviation values of the time spent in the kernel boot and userspace phase are slightly higher than other standard deviation values as the time spent in these phases varies depending on how fast the processor executes the kernel boot process, how fast the kernel mounts the root file system and how fast the “init” process initialises the system management services as the Linux boot enters the CentOS 8 userspace. No manual intervention was involved during the Linux boot process.

Statistics for Partial Network Boot Times (seconds)					
Statistics	Pre-Boot Phase	Kernel Image Loading	Device-Tree Loading	Kernel Boot	Userspace
Mean	10.11	1.20	0.02	9.33	17.74
Minimum	9.90	1.19	0.02	9.19	16.82
First Quartile	10.03	1.20	0.02	9.19	17.39
Median	10.15	1.20	0.02	9.22	17.80
Third Quartile	10.21	1.20	0.02	9.26	18.05
Maximum	10.25	1.22	0.02	10.26	18.74
Std. Deviation	0.12	0.01	0.00	0.33	0.53

Table 6.3: Statistics of 10 samples for boot timings of the partial network boot.

In Table 6.3, the statistics for the 10 samples of the timing of partial network boot are presented. In Table 6.3, statistics of different phases of the Linux boot process are also presented. In Table 6.3, it can be seen that the standard deviation of the time spent in the different boot phases before the kernel boot is very low. This is because the values of the time spent in different boot phases before the kernel boot was fairly consistent across the 10 different samples and they are quite similar to the values seen In Table 6.2. This is because in the partial network boot, boot files are also loaded from the SD card before the root file system is mounted from the NFS server. The standard deviation values of the time spent in the kernel boot and userspace phase are slightly higher than other standard deviation values as the time spent in this phase varies depending on how fast the processor executes the kernel boot process, how fast the kernel mounts the root file system from the NFS server and how fast the “init” process initialises the system management services as the Linux boot enters the CentOS 8 userspace. No manual intervention was involved during the Linux boot process.

Statistics for Full Network Boot Times (seconds)					
Statistics	Pre-Boot Phase	Kernel Image Loading	Device-Tree Loading	Kernel Boot	Userspace
Mean	10.10	1.79	0.01	9.24	17.53
Minimum	9.91	1.76	0.01	9.18	17.12
First Quartile	10.07	1.78	0.01	9.19	17.27
Median	10.16	1.79	0.01	9.20	17.57
Third Quartile	10.18	1.81	0.01	9.20	17.80
Maximum	10.21	1.85	0.01	9.61	17.85
Std. Deviation	0.10	0.03	0.00	0.13	0.28

Table 6.4: Statistics of 10 samples for boot timings for the full network boot.

In Table 6.4, the statistics for the 10 samples of timing of the full network boot are presented. In Table 6.4, statistics of different phases of the Linux boot process are also presented. In Table 6.4, it can be seen that the standard deviation of the time spent in the different boot phases before the userspace phase is very low. This is because the values of the time spent in different boot phases including the kernel boot phase was fairly consistent and varied only in their 3rd and 4th decimal places and beyond. The standard deviation of the time spent in the CentOS 8 userspace is slightly lower than the standard deviation for the time spent in the same phase for the other two boot methods. As mentioned previously, the time spent in this phase depends on how fast the “init” process initialises the system management services. No manual intervention was involved during the Linux boot process. The contents and services initialised in the CentOS 8 root file system were consistent for all the three boot methods.

In all the three tables, Table 6.2, Table 6.3 and Table 6.4, it can be seen that the time spent in the pre-boot phase and the standard deviation for the time spent in the pre-boot phase for different boot methods was fairly consistent and varied only slightly. This is because this phase is entirely dependent on the internal boot setup process followed by the Zynq UltraScale+ (explained in Chapter 3) and no manual intervention is possible in this phase till the user interrupts the U-Boot. Additionally, the time spent in loading the kernel Image and the device-tree for different boot methods is fairly consistent for that particular boot method. This is because the speed of loading these boot files from the storage medium does not vary a lot as seen in Table 6.1.

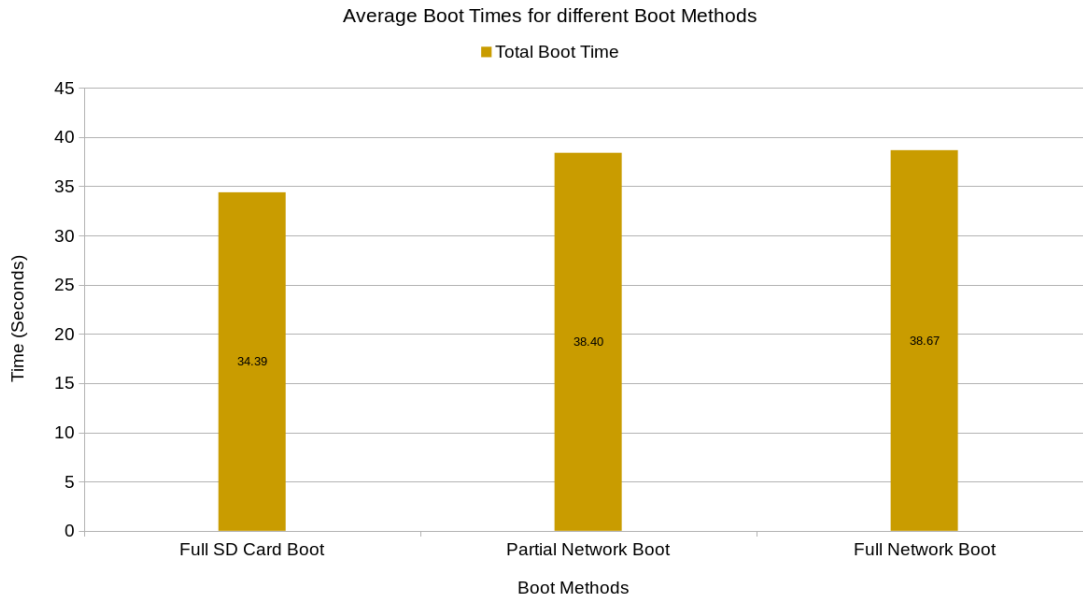


Figure 6.2: Average Linux boot times for the three boot methods

Figure 6.2 depicts average boot times for different boot methods like full SD card boot, partial network boot and full network boot. It was found out that the full SD card boot was faster than partial network boot and full network boot by an average of about 4 seconds. The explanation for this phenomenon has been explained along with the explanation of the graph presented in Figure 6.3. As seen in Figure 6.2, the full SD card boot had an average boot timing of 34.39 seconds, the partial network boot had an average boot timing of 38.4 seconds and the full network boot was slightly slower than partial network boot and required 38.67 seconds for complete boot. This data alone is not conclusive enough for saying that the full SD card boot is better than the network boot. In order to understand where the differences in the boot timing is coming from, tests were carried out to find the breakup of the boot timing in terms of time spent in loading the U-Boot,

time spent to download the boot files, time spent in booting the kernel and the time spent in CentOS 8 userspace initialisation. The statistics of these values have been presented in Table 6.2, Table 6.3 and Table 6.4.

The time spent in the pre-boot phase was obtained with the help of a digital stop watch, for the lack of any other method which could be used to measure that timing. The time spent in loading the kernel Image and device-tree was calculated from the boot file read speeds reported in the U-Boot terminal (an example of which seen is Figure 5.3) and as per the data presented in Table 6.1. The time spent in the kernel boot phase and the CentOS 8 userspace was measured with the help of an internal CentOS 8 tool called “systemd-analyze” which accurately reports the time spent in the kernel boot phase and the CentOS 8 userspace phase.

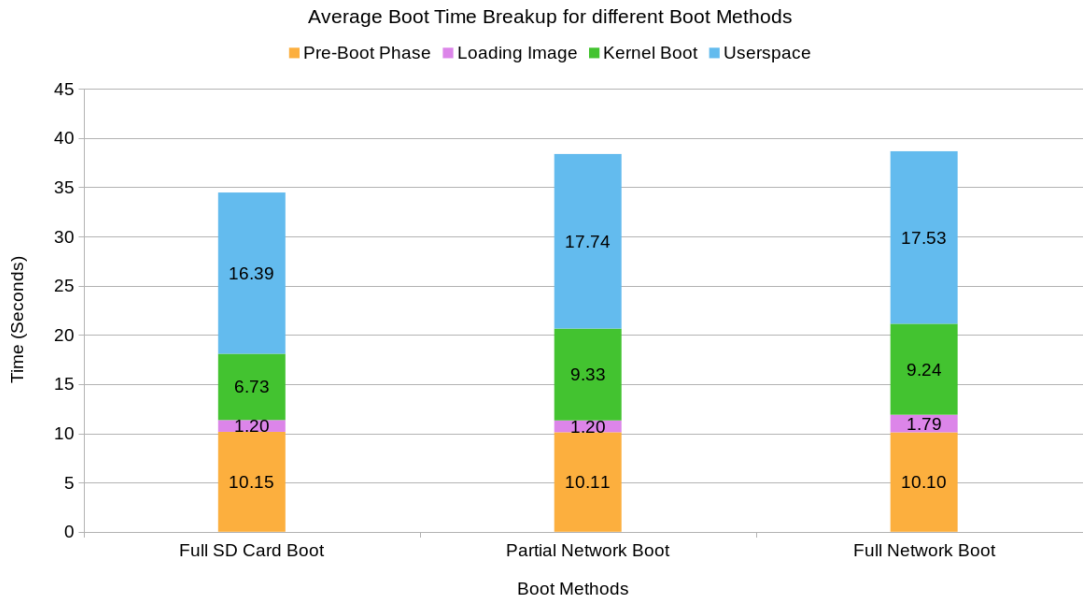


Figure 6.3: Average Linux boot time breakup for the three different boot methods

Figure 6.3 depicts the average boot time breakup for different boot methods like full SD card boot, partial network boot and full network boot. The breakup consists of time spent in pre-boot phase, loading the boot files, the time spent in booting the kernel and the time spent in user-space after execution of the init process.

In Figure 6.3, it can be seen that the average time spent in the pre-boot phase (loading the FSBL and U-Boot) is 10.15 seconds for full SD card boot, 10.11 seconds for partial network boot and 10.10 seconds for full network boot. It can be seen that this timing is quite similar for all boot methods. This is because this part of the boot process is entirely dependent on the performance of the PMU bootROM, PMU firmware, CSU bootROM and the speed at which the FSBL and U-Boot are loaded in the Zynq UltraScale+(explained in Chapter 3). There is no human intervention possible here.

In Table 6.2, Table 6.3 and Table 6.4 it can be seen that the average time spent in loading the device-tree is too small (between 0.01 and 0.02 seconds) and on the scale of the graph shown in Figure 6.3, their share in the boot process would not be visible as the time required is insignificant in context of the larger boot process. Hence, those values are not included in the graph. The average time spent in loading the kernel Image is 1.20 seconds for both the full SD card boot and the partial network boot since both methods load the device-tree and kernel Image from the SD card and as seen in Table 6.1, there is very little variation in the load speeds of kernel Image and

device-tree from the SD card and the TFTP server. The average time spent in loading the kernel Image for the full network boot is slightly higher at 1.79 seconds. This slightly slow timing can be attributed to the time spent in the request, grant and acknowledgement transactions associated with the TFTP server. The time spent in loading the boot files also depends on the size of the boot files. If the hardware description in the device-tree is longer or if more features are activated in the kernel, the size of the device-tree and the kernel Image would increase and as such the time spent in loading these files would also increase.

In Figure 6.3, it can be seen that for the full SD card boot, the average time spent in booting the kernel Image is 6.73 seconds, for a partial network boot the value is 9.33 seconds and for a full network boot the value is 9.24 seconds. When the kernel is booting, it is looking for the hardware addresses of different peripherals for which drivers and services have been activated in the kernel to support the hardware. Upon finding the location of these hardware peripherals, the kernel enables and tests these drivers and services in parallel. This booting of the kernel is not subject to manual intervention and this timing may increase or decrease depending on how fast the processor executes the kernel boot process. The difference in kernel boot timing between the full SD card and the other two boot methods is around 2.50 seconds. This difference is due to the time spent by partial network boot and full network boot in activating the Ethernet interface, making a DHCP request for the NFS server IP address and root file system path as well as the time spent in mounting the root file system from the NFS server. The full SD card boot method does not have to make a DHCP request and thus is able to mount the root file system directly from the SD card's EXT4 partition, saving some time in kernel booting.

As can be seen from Figure 6.3, there is no significant difference between the different boot methods as far as the average time spent in the CentOS 8 userspace is concerned. Once the root file system is mounted, the full SD card boot spent 16.39 seconds in userspace on an average, the partial network boot spent 17.74 seconds and the full network boot spent 17.53 seconds in the userspace on an average. The full SD card boot is slightly faster in this context possibly because of faster read speeds of small executable system initialisation files from the SD card compared to the read speeds of these files from NFS over 1 Gigabit Ethernet (tests presented in later sections). The time spent in the userspace depends on how fast the processor executes the "systemd" initialisation and other system management services which are activated in the kernel as well as enabled in the CentOS 8 root file system. If more services are activated in the kernel and the userspace, the time spent in the userspace during the Linux boot also increases.

From the test results presented above, it was clear that the full SD card boot is faster for boot files and root file systems of similar sizes. However, it needed to be investigated if the SD card performs as well as the 1 Gigabit Ethernet network with SSD storage on NFS servers when it comes to the read-write speeds to the root file system.

6.2 Performance of Root File System Types

The most frequent operations executed by the the Zynq UltraScale+ based embedded controllers in the CMS data acquisition network would be to read files from and write files to the CentOS 8 root file system to perform their computation tasks. These operation could involve copying a big application file and its libraries measuring in Gigabytes (GB) in to the processor RAM and executing the application. These operations could also involve reading small binary utility files required in execution of a script by the Zynq UltraScale+ processor or the reading and execution of small files during the CentOS 8 userspace initialisation phase of the Linux boot process.

On the other hand, Zynq UltraScale+ based embedded controllers might want to write performance or error logs to the root file system. Privileged users of these controller platforms might

want to install a large software packages of a few hundred Megabytes or a Gigabyte on the root file systems (explained in Chapter 7), which could also involve reading and writing many small files and small install scripts to and from the root file system. Thus, considering all these different use cases, it is important to study the read-write performance of two different root file system types namely: the SD card storage and the storage on the NFS servers.

The tests were conducted using the same equipment mentioned in section 6.1.1. The read-write tests were automated with the help of a script presented in Appendix C. The script helps in creating files and folders to store the files, clearing the cache, copying the files to and from the processor RAM disk and deleting the files at the end of the tests.

6.2.1 Average Read Speeds from Root File System

For measuring the read speed from the root file system, 2 different types of files were used: 1 big file of 1 GB and 1000 small files of 1 MB. These files mirror the kind of files that would be read by the Zynq UltraScale+ processor from the root file system for its operations. The files were copied from the root file system on the SD card and the NFS server and written to the Zynq UltraScale+ PS RAM disk, mirroring the operations that would be carried out when the embedded systems are deployed in the CMS DAQ network. The Kingston SD card was tested in the HS (High-Speed) and the UHS (Ultra-High Speed) mode. The Kingston SD card supports 25 MB/s read speed in HS mode and 104 MB/s read speed in UHS mode [42]. The NFS server was running on the TTL NUC5 server and was communicating over 1 Gigabit Ethernet network. The SSD of the TTL NUC5 server gave uncached disk read speeds of up to 400 MB/s when it was tested using the command “`hdparm -tT /dev/sda`”, where `hdparm` tests cached and uncached disk read speeds and `/dev/sda` points to the disk mount point. These are read-speeds which are largely in access of the theoretical speed limit of 125 MB/s associated with 1 Gigabit Ethernet network, over which the ZCU102 client communicated with the NFS server.

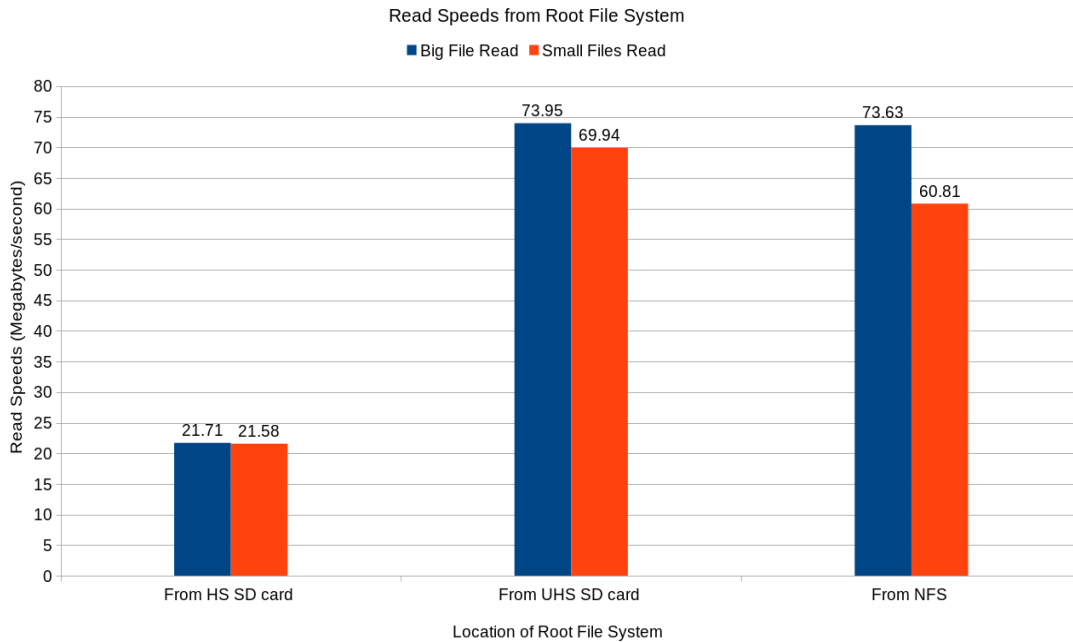


Figure 6.4: Average read speeds from root file system (writes to RAM disk)

Figure 6.4 shows the average read speeds from the root file system for the HS and UHS mode of SD card as well as the read speeds over 1 Gigabit Ethernet from the NFS server. As can be seen in Figure 6.4, it was observed that the UHS mode of the SD card and the 1 Gigabit Ethernet have similar average read speeds for big files. The read speed of 1000 1 MB files from UHS SD card was slightly faster (69.94 MB/s) than reading 1000 1 MB files over 1 Gigabit Ethernet from the NFS server (60.81 MB/s). The HS mode of the SD card reads big files and small files at an average speed which is 3-4 times less than the average read speeds from the UHS SD card and the NFS server respectively for either of the file size types. The cache memory was cleared before every test so that the data would not be reused from the cache during the tests, so as to ensure accurate test results.

Statistics for Read Speed Tests from the Root File System (Megabytes/second)						
Statistics	Big File (One 1 GB file)			Small Files (1000 1MB files)		
	From HS SD card	From UHS SD card	From NFS server	From HS SD card	From UHS SD card	From NFS server
Mean	21.71	73.95	73.63	21.58	69.94	60.81
Minimum	20.53	73.35	73.20	21.51	69.78	57.05
First Quartile	21.81	73.68	73.35	21.57	69.89	59.22
Median	21.82	73.83	73.54	21.59	69.95	61.65
Third Quartile	21.84	73.88	73.88	21.60	70.03	62.38
Maximum	21.99	75.91	74.26	21.62	70.03	62.93
Standard Deviation	0.42	0.71	0.37	0.03	0.09	2.11

Table 6.5: Statistics of 10 samples for read speed tests from the root file system storage types

In Table 6.5, the statistics of the 10 samples of read speed tests for reading big and small files from the root file system are presented. Here we can see that the standard deviation in the read speed values is under 1 MB/s except for the read speeds of small files from the NFS server, where the standard deviation is 2.11 MB/s. This is because the read speeds of small files from the NFS server vary between 57 MB/s and 63 MB/s whereas other read speeds do not show this kind of large variation. It is evident from the differences present between the first quartile, median and third quartile values of read speeds of small files from the NFS server that the read speed values are spread across the range between minimum and maximum read speeds of 57.05 MB/s and 62.93 MB/s respectively. This fluctuation in small file read speeds from the NFS server can be attributed to the network latency and possible congestion in the CERN network during the time of the testing.

Thus from Figure 6.1 and Table 6.5, it can be seen that the UHS mode of the SD card and the 1 Gigabit Ethernet perform equally well while reading big files from the root file system and the UHS SD card is slightly faster than 1 Gigabit Ethernet while reading small files from the root file system. For the storage types, reading of 1000 1 MB files is slower than a 1 GB file because of the overhead associated with searching the path of each small file, finding it and writing it individually to the processor RAM disk. While the performance does not get affected for one 1 MB file, the performance gets affected when searching for 1000 1 MB files individually before writing them to the processor RAM disk. On the other hand, the 1 GB file is searched for only once and written to the processor RAM disk. Thus, reads of big files are faster than reading small files.

6.3 Average Write Speeds to Root File System

Just like the reading tests, to measure the write speeds to the root file system, 2 different types of files were used: 1 big file of 1 GB and 1000 small files of 1 MB. The files were read from processor RAM disk and written to the root file system storage locations. The SD card was tested in the HS and the UHS mode. The cache memory was cleared before every test so the data should not be reused from the cache during the tests, so as to ensure accurate test results. The Kingston SD card supports write-speeds up to 70 MB/s [42].

Statistics for Write Speed Tests to the Root File System (Megabytes/second)						
	Big File (One 1 GB file)			Small Files (1000 1MB files)		
Statistics	To HS SD card	To UHS SD card	To NFS server	To HS SD card	To UHS SD card	To NFS server
Mean	13.04	34.22	104.41	13.20	37.59	36.72
Minimum	12.20	31.60	99.81	12.50	36.79	35.26
First Quartile	13.02	32.13	103.57	13.13	36.90	35.55
Median	13.17	34.45	105.35	13.31	37.81	35.68
Third Quartile	13.22	36.01	106.14	13.39	38.21	35.81
Maximum	13.29	37.65	107.11	13.43	38.33	41.81
Standard Deviation	0.33	2.20	2.58	0.29	0.66	2.39

Table 6.6: Statistics of 10 samples for write speed tests to the root file system storage types

In Table 6.6, statistics for the write speed tests are presented for SD card and NFS root file system. Here it can be seen that the standard deviation of write speeds of writing big files to UHS SD card and NFS server is high along with the standard deviation of write speeds of writing small files to the NFS server. The standard deviation for these cases is high due to a bigger difference between maximum and minimum values compared to other cases as well as values being spread across the range of these values. It is evident from the differences present between the first quartile, median and third quartile values of write speeds of big files to the UHS SD card and NFS server and write speeds of small files to the NFS server. This fluctuation in write speed to the NFS server can be attributed to the network latency and possible congestion in the CERN network along with the varying performance of write speeds to the SSD on the NFS server.

Figure 6.5 (seen on Page 74) shows the average write speeds to the root file system stored on the SD card in the HS and UHS mode as well as the write speeds to the NFS over 1 Gigabit Ethernet network. As can be seen in Figure 6.5, it was observed that the average write speeds of writing big files to the NFS are 10 times higher than the SD card in HS mode and three times higher than the SD card in UHS mode. The average write speeds of writing small files to the NFS server and UHS mode of the SD card are almost equivalent at 36.72 MB/s and 37.59 MB/s respectively which is three times higher than the write speeds of small files to the SD card in HS mode (13.2 MB/s). Thus, it can be seen that write speeds to NFS are faster than the SD card for big files and equivalent to the UHS mode of the SD card when small files are being written.

The writes of small files to the SD card or the SSD storage on the NFS server is very slow. For writing a big file to the NFS server, the write-speeds obtained were close to the theoretical speed of 1 Gigabit Ethernet. It must be understood that write speeds to the disk (either SD card or SSD) are slower compared to read speeds since they are limited not only by their interfacing with the processor, but also by the scheduling of the writes to the disk by the Linux kernel and the time spent in finding the data to be read, reading the data, finding a free space on disk to write the data and finally writing the data. As a result, the small file write speeds to the NFS server are considerably slower than writing a big file of 1 GB on the NFS since small file writes have to be done 1000 times for 1 MB files and the process is repeated for each file. The SD card

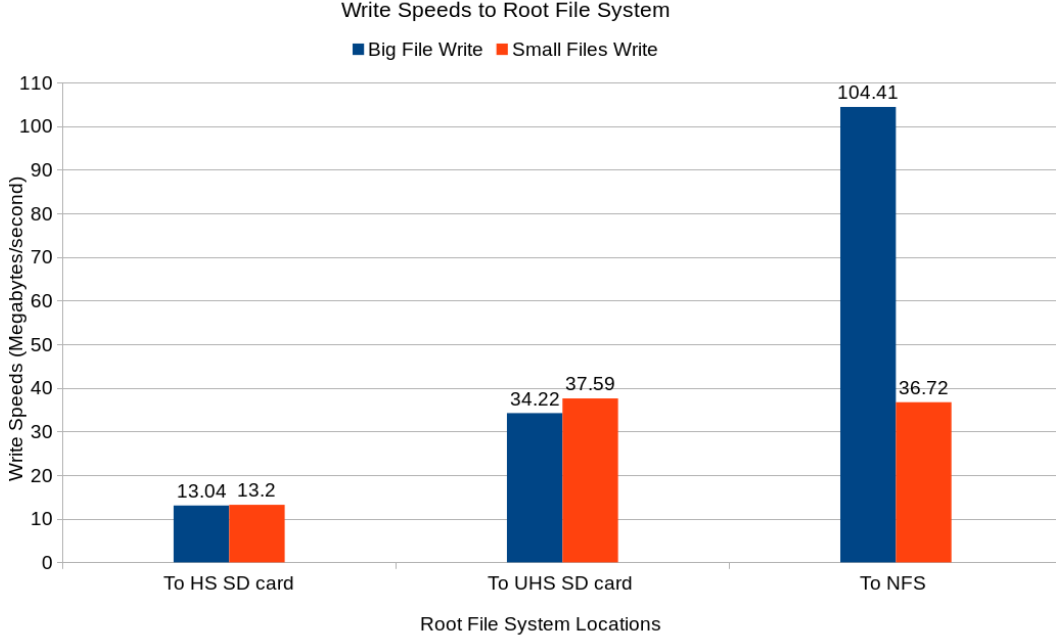


Figure 6.5: Average write speeds to root file system (reads from RAM disk)

interface in UHS mode can theoretically do a maximum of 104 MB/s [44], however that is not achieved even during the SD card reads as seen in section 6.2. The Kingston SD card used for testing promises only 70 MB/s write speeds [42] and that is further limited by scheduling of the writes by the kernel and the write procedure followed (as explained above). As a result, writes of big and small files both are slow when it comes to the HS and UHS mode of the Kingston SD card.

6.3.1 Comparing SD Cards and Network Storage

In this section, we compare a Kingston 32 GB Canvas React (Class 10 UHS-I U3 A1 V30) SD card [42], a Kingston 32 GB High Endurance (Class 10 UHS-I) SD card [45] and a Samsung 32 GB Pro Endurance (Class 10 UHS-I) SD card [46] with an Intel D3-S4510 960 GB Solid State Storage Disks (SSDs) [47] that comes with the Dell R440 [48] server machines that would host the TFTP and NFS server in the CMS data acquisition network. A 32 GB SD card is considered since most of the experiment groups at CMS do not require root file systems which occupy more than 32 GB of space on the disk.

- **Costing:** A 1000 hardware platforms with Zynq UltraScale+ MPSoC would be installed in the CMS data acquisition network for a period of 10 to 15 years. Some of these boards would be designed using high-end FPGA (Field Programmable Gate Array) and the cost of each of these boards would be around \$15-20,000. A fraction of the new hardware platforms would use very high-end FPGAs and the cost of such boards would be around \$40,000.

Both the Kingston 32 GB Canvas React and Kingston 32 GB High Endurance SD card cost around \$15 (costs for 2020 as seen on Kingston online retail store). Thus, a thousand 32 GB SD cards of either type would cost around \$15000. The Kingston High Endurance 32 GB SD cards promise 5000 hours of continuous reading without interruptions [45]. It means that continuous read operations can be carried out from these SD cards for up to 6 months

without damages to the SD cards. The Samsung 32 GB Pro Endurance SD cards promise continuous reads for up to 17,520 hours i.e 2 years for a marginally higher cost of \$25 for one SD card. For similar number of continuous read hours, we would need up to 3 Kingston High Endurance SD cards. Assuming just continuous read-operations from SD cards for 2 years, in the best case scenario the Samsung Pro Endurance cards would need to be replaced every 2 years and at least 5 to 7 such replacements would be needed over the life time of Zynq UltraScale+ embedded controllers. Even then, the cost of these SD cards (considering even 5 to 7 replacements) would be less than 1 % (\$125 to \$175) of the cost of the actual hardware boards with the Zynq UltraScale+ based controllers.

However, it can be safely assumed that there would be no cases of exclusive reads from the SD cards for 2 years at a stretch and the usage of the SD would be mostly for reading boot files and certain important files from the root file system interspersed with occasional writes to the SD cards. Thus, it is assumed that SD card would not require frequent replacements and would only be discarded when their performance goes down due to aging. Thus, it is assumed that maybe 4 rounds of SD card replacements would be required.

As compared to this, the R440 server with 4, 960 GB Intel D3-S4510 SSDs can host root file systems of 40-50 hardware platforms. The R440 server machines bought by CMS cost \$2000 each, the SSDs cost around \$560 each and 4 such SSDs will cost around \$2240. Since there would be 1000 such hardware platforms, around 20-25 R440 servers with 4 SSDs each would be required. Thus, the total cost of such a network configuration can go up to \$106,000. These servers come with RAID (Redundant Array of Independent Disks) cards which ensure that if one of the SSDs of the NFS server fails, a redundant SSD with replicated data is always ready to serve the client. If the RAID cards and the R440 servers malfunction, the manufacturer guarantees free-of-cost replacement of the server. Thus, the R440 servers and the SSDs that come with it are redundant and reliable, guaranteed by the manufacturer. These servers would be replaced every 5 years over a period of 10 to 15 years (three times). Thus, this server configuration would cost around \$300 per Zynq UltraScale+ device for a period of 10 to 15 years which is around 1.5% of the cost of the actual hardware boards with the Zynq UltraScale+ based controllers.

Thus, the SD card storage and TFTP/NFS server storage for a period of 10 to 15 years would cost less than 2% of the Zynq UltraScale+ hardware platform. But the TFTP and NFS server storage promises redundancy, reliability and ease-of-maintenance (explained later) that is not associated with the SD card storage.

- **Speeds:** The Kingston 32 GB Canvas React SD card promises read speeds up to 100 MB/s and write-speeds up to 70 MB/s [42]. The Kingston 32 GB High Endurance SD card promises read speeds of up to 95 MB/s and write speeds of 30 MB/s [45]. The Samsung 32 GB Pro Endurance SD cards promise read speeds of up to 100 MB/s and write speeds of up to 30 MB/s [46]. Even for more expensive cards like the Lexar 1 TB SD card [49], the write speeds don't improve even though they can cost up to \$400 (costs for 2020 as seen on Amazon online store). All these cards are bound by the UHS-I class read-write speeds and the Zynq UltraScale+ hardware platforms would only be able to support the UHS-I class speeds due to hardware limitations of the Zynq UltraScale+ SD card interface [21].

The Intel D3-S4510 SSDs that come with the the Dell R440 support bus-speeds of up to 6Gbps or 750 MB/s [47]. It supports sequential read speeds up to 560 MB/s and sequential write speeds 510 MB/s [47]. All these values are for peak performance and could be lower for reading and writing smaller files. Thus, some tests were conducted by the CERN system-administrators on these disk with RAID cards to check the real read-write speeds of these disks. The read-speeds for a RAID array of 4 SSDs was found to be 2.09 GB/s for a 1.6 GB file i.e 536 MB/s for each SSD. The write-speeds with the RAID arrangement were found to be 972 MB/s for 10,000 1 MB files i.e 243 MB/s to each SSD. Cache data was emptied before each test. Thus, it can be seen that read-write speeds of the SSD are significantly higher than that of the UHS SD cards.

Since one R440 server would be serving 40-50 boards and all of these boards would be sharing the common root file system, most of their reads would be cached reads, as they would be reading common files. Thus, the read-speeds experienced by the Zynq UltraScale+ platforms would not be affected by the simultaneous reads being carried out by 50 other similar boards. The write speeds to the NFS server for simultaneous writes by 50 boards would drop to about 20 MB/s (dividing the write speeds of small files explained in the paragraph above by 50) which is close to the write speeds observed in Figure 6.5. However, it is not expected that 50 devices would simultaneously write a huge number of files to the NFS servers and better write speeds to the NFS server can be expected.

- **Ease of maintenance:** Making modifications to the files on the SD card or replacing the SD card requires one to go to the experiment site (Point 5 of CERN) and replace it. It also involves taking the hardware platform out of service at least for sometime. This implies high operational costs in terms of time and money.

The storage of files on a server is far more flexible and easy to maintain as boot files on TFTP server and root file systems on the NFS server can be easily modified, replaced or backed-up whenever necessary.

- **Redundancy:** If the SD card storing the root file system malfunctions, the SD card has to be replaced with another SD card. However, there is no way of reliable, regular backing up of the data on the SD card if the SD card malfunctions. Additionally, the system experiences downtime till the SD card is replaced. If the data is not backed up, it is lost.

In that context as well, network storage of root file system and boot files is far more beneficial than storing everything on the SD card. The root file system on the NFS servers and the boot files on the TFTP servers are backed up regularly at multiple server locations to provide much needed redundancy for similar costs as the SD cards (over a period of 10 to 15 years). Additionally the RAID cards on the R440 servers ensure that if one disk fails, the other disks with replicated data are always ready to serve the Zynq UltraScale+ devices.

- **Reliability and Availability:** The R440 servers are used with RAID (Redundant Array of Independent Disks) cards making them more reliable in case of disk failure as data is duplicated across identical, independent disks for no additional cost.

The hardware platforms do not have such sophisticated RAID configurations for SD cards and thus the reliability of such SD cards is low as compared to the network storage even though the SD card storage configuration costs are similar as the network storage of boot files and root file system (over a period of 10 to 15 years) when compared with the cost of the Zynq UltraScale+ hardware platforms.

From all the points mentioned above, it is quite evident that the method to store boot files and root file system on the network storage does not cost very different from the SD card storage method and costs only a fraction of the cost of Zynq UltraScale+ based embedded controllers. Despite this, the network storage involving the Dell R440 server and Intel D3-S4510 SSDs offers much higher read-write speeds, ease-of-maintenance, reliability, availability and flexibility as compared to the SD card storage method, irrespective of the UHS-I SD card type in use.

6.4 Summary

From the performance of different booting mechanisms, root file system storage methods and comparison between the SD card and network storage method, the following inferences can be drawn:

- The full SD card boot is faster than the partial network boot and full network boot, but that is only due to the fact that the full SD card boot does not spend time in making a DHCP request and root file system mount from the NFS server. Otherwise, all the boot methods perform equally well, where the full network boot offers more flexibility, reliability and ease-of-maintenance when looked at from the perspective of the CMS data acquisition network. In addition, the difference in the boot timing of full SD card boot and full network boot is less than 10%.
- The reads speeds of reading big and small files are similar for the root file system on the Kingston 32 GB SD card in UHS mode and for the root file system on the NFS server hosted by the TTL NUC5 desktop. The write speeds of big files to NFS server over 1 Gigabit Ethernet are higher than the write speeds of big files to the SD card in UHS mode but the write speeds of small files to the NFS server is equivalent to the write speeds of small files to the SD card in UHS mode. However, during the final deployment Dell R440 servers with Intel D3-S4510 SSDs and RAID cards would be used to operate the NFS server in the CMS data acquisition network. This would significantly improve the read-write performance of the network storage, but it will still be limited by the 1 Gigabit per second performance of the client's network interface.
- Buying expensive SD card with bigger storage capacity like the Lexar 1 TB SD card does not improve the read-write performance of the SD card even in the UHS mode. The maximum read speeds of the SD card in the UHS mode is 104 MB/s [44] and this is still a fraction of the read-write speeds offered by the Intel D3-S4510 SSDs as seen in the previous section. However, High Endurance SD cards are still required for storing the BOOT.BIN file for loading the FSBL and U-Boot in the Zynq UltraScale+. Usage of High Endurance SD cards ensures minimal replacements of SD cards.
- The cost of the SD card storage and network storage of bootfiles and root file system for 1000 Zynq UltraScale+ devices in the CMS data acquisition network is less than 2% when seen in comparison to the cost of the Zynq UltraScale+ hardware. Considering this low-cost and additional benefits like read-write performance, ease-of-maintenance, reliability, availability and redundancy, the network storage of boot files (TFTP) and root file system (NFS) seems the better option for the CMS data acquisition network.

After considering the points mentioned above in the summary, a full network boot with a NFS storage of the root file system is recommended for the Zynq UltraScale+ based embedded controllers in the CMS data acquisition network.

Chapter 7

Installing and Administering Software Updates

In this chapter, two different methods of installing and administering software updates on the CentOS 8 root filesystem are demonstrated. The features of the two methods are presented to the reader before discussing use-cases for the software update methods for better understanding of the reader.

7.1 Administering Software Updates

Two methods are described in this chapter which use the “DNF” package manager (see section 7.1.1) for installing software updates on the CentOS 8 root file system. The methods adopted are as follows:

1. **Client side updates:** Initiating software updates by executing package manager commands on the running client. The package manager will read from and write to the root file system of the client over NFS.
2. **NFS server side updates:** Initiating software updates by executing package manager commands on the NFS server. The package manager will read from and write directly to root file systems of individual clients, which are stored on a local disk of the machine hosting the NFS server.

The NFS server side updates are essentially installing software updates on the root file systems of different Zynq UltraScale+ devices that are stored on the local disk of the NFS server. The updates are initiated from the userspace of the same machine that hosts the NFS server. Here the software packages for the CentOS 8 root file system are downloaded by the machine and written directly to the individual CentOS 8 root file systems belonging to different Zynq UltraScale+ devices. This is similar to the building of the CentOS 8 root file system demonstrated in Chapter 4.

On the other hand, client side updates are initiated from the userspace of the Zynq UltraScale+ hardware platforms, where the software update is written only to their own CentOS 8 root file system located on the NFS server.

7.1.1 DNF Package Manager

Dandified YUM (DNF) is a package manager for RPM (Red Hat Package Manager)-based Linux distributions like CentOS. It helps in installing, updating and removing RPM packages on the CentOS Linux distribution. In the CentOS 8 Linux distribution, both yum and dnf are the same binaries used interchangeably and use the same software package repositories. The repository configuration files for DNF are nothing but a collection of text files which point to different software package directories on the CentOS 8 repository mirror running locally in CERN or on a dedicated server on the web from where computers can download RPM packages for installation (shown in Appendix D). The “dnf.conf” file in the “/etc/dnf” path of the root file system configures the general behaviour of the package manager and is configured by the package manager itself during building of the CentOS 8 root file system. Developers can edit this configuration if they want some additional features in their DNF configuration.

```
[main]
cachedir=/var/cache/dnf/$basearch/$releasever
keepcache=0
debuglevel=2
logfile=/var/log/dnf.log
exactarch=1
obsoletes=1
gpgcheck=1
plugins=1
installonly_limit=5
bugtracker_url=http://bugs.centos.org/set_project.php?project_id=23&ref=http://bugs
.centos.org/bug_report_page.php?category=yum
```

Listing 7.1: Configuration for DNF Package manager

In Listing 7.1, a sample configuration of the DNF package manager is presented. In the listing, all options equal to 1 are enabled and the options equal to 0 are disabled. The “cachedir” option points to the directory where the RPM packages are cached. “\$basearch” refers to the architecture of the processor and “\$releasever” refers to the version of CentOS. The “keepcache” option decides whether to store or delete the RPM packages downloaded during the previous installation. “debuglevel” can have value from 0 to 10 and the higher the value, the higher is the debug level or the details printed during installation. “logfile” refers to the temporary log file where the installation logs are stored. “exactarch” specifies to consider only the architecture for which the install is being done. “obsoletes” allows replacement of obsolete packages when an updated package is available, which is important during updates. “gpgcheck” specifies whether the CentOS key should be authenticated or not during the installation process. “plugins” activates or disables the global repositories to be used. Alternatively, individual repositories can be disabled in repository configuration files by using the option “enabled=0” (sample repository configuration in Appendix D). “installonly” points to how many version of the same packages can be installed at the same time. “bugtracker_url” points to the web-link where bugs encountered during installation are reported.

7.1.2 How Installation Works with DNF

The following steps are followed by DNF during package installation and update:

- The user specifies the packages to be installed or the user asks for a global update of packages on the system (“dnf install \$package” or “dnf upgrade”).
- If an installation command has been received, the package manager checks if the package

is already installed. If it is installed, the user is informed. Otherwise, the repositories are searched for the package.

- If an update command has been received, the package manager checks all packages that have been installed and compares them with the packages available in the repository. The version specifications of these packages are compared and only if there are updated packages available, the user is informed.
- In both the cases mentioned above, the user is asked whether he/she wishes to continue with installation (unless the “-y” option is added as an argument to the command which implies yes to all questions from the package manager).
- Once the user approves the installation, the packages are downloaded and the obsolete packages are updated.

7.1.3 NFS Server Side Software Updates

The CentOS 8 root file system is exported to the Zynq UltraScale+ hardware via NFS. If the installation or update to this root file system (meant for 64-bit ARM processor) is done from an x86 machine hosting the NFS server, the NFS server side update can also be called a server side cross-installation of software updates.

Software updates can be done on the CentOS 8 root file system located on NFS server disk with the help of DNF package manager. The DNF configuration file used in building a CentOS 8 root file system for Zynq UltraScale+ is used for configuring the repositories for the DNF package manager. It can be seen in Appendix A. This file points the DNF package manager of the machine hosting the NFS server to the CentOS 8 repositories for 64-bit ARM processors. The NFS server side installation of software updates was done on the TTL NUC5 desktop (Intel Core i5 processor with 16 GB RAM) which hosted NFS server. The specifications for this machine is given in Chapter 6.

The process of NFS server side installation of software updates on the root file system is similar to the process explained in section 7.1.2. There is a need for a QEMU emulator in the “usr/local/bin” path of the CentOS 8 root file system for 64-bit ARM processors, if the software update is being done to that root file system on an x86 machine hosting the NFS server. The reason to use the QEMU emulator for this kind of installation of software updates on the CentOS 8 root file system is provided in the section 4.6.3 in Chapter 4.

The following command is used to administer software updates to the CentOS 8 root file system for 64-bit ARM architectures located on an NFS server:

```
sudo dnf -y -c /home/kmor/CentOS8/etc/dnf/dnf.conf \
--skip-broken --releasever=8 \
--forcearch=aarch64 \
--repo=arm64-epel,centos-base,centos-updates,\
centos-extras,centos-appstream,\
centos-powertools,centos-centosplus \
--installroot=/home/kmor/CentOS8 \
update
```

Listing 7.2: Command for installing server side software updates

Listing 7.2 shows the command to install NFS server side software updates on the CentOS 8 root file systems of individual Zynq UltraScale+ hardware platforms located on the NFS server. In Listing 7.2, the option “releasver” specifies the release version of CentOS, “forcearch” specifies the 64-bit ARM architectures, “repo” points to the CentOS repositories to be used, “-c” points to the “dnf.conf” file belonging to the target CentOS 8 root file system and “install-root” points to the directory where the root file system is present. Spaces have been added between the lines in the listing to improve readability. Spaces between the dnf options should be removed when using the command.

The terminal output from this command is presented in Appendix D.

7.1.4 Zynq UltraScale+ Client Side Software Updates

The Zynq UltraScale+ devices in the CMS data acquisition network can also connect to the CentOS 8 software package repositories after obtaining an IP address from the DHCP server. Thus, software updates to their individual CentOS 8 root file systems on the NFS server disks can be initiated from the userspace of Zynq UltraScale+ clients in the CMS data acquisition network. DNF package manager can be used for this update method. Before initiating this kind of software update, users must verify if they have the right DNF configuration file in their root file system (path: /etc/dnf/dnf.conf) and if the repositories for this DNF package manager have been configured properly during the build of the CentOS 8 root file system. An example of the root file system’s DNF configuration is Listing 7.1 and an example repository configuration file has been given in Appendix D.

“dnf upgrade” command is used to initiate client side software updates on the hardware platform. Its terminal output is presented in Appendix D.

7.1.5 Features of NFS Server Side Software Updates

- This installation of software updates can be commissioned directly by system-administrators without having an additional process running on the hardware platforms or direct involvement of the hardware platform.
- It is fairly easy to configure scripts that can be executed to update multiple CentOS 8 root file systems on the NFS server (each dedicated to one hardware platform) simultaneously or in phases to efficiently use the network bandwidth.
- A common read-only root file system (see future work in Chapter 8) can be updated simultaneously for multiple hardware platforms, allowing the updates to be in sync for all hardware platforms and also avoid separate update of every individual root file system.
- There is a need for QEMU, if the software update to a CentOS 8 root file system for 64-bit ARM processors is being carried out on an x86 machine hosting the NFS server.

7.1.6 Features of Client Side Software Updates

- Once the repositories and package managers are configured, it is comparatively simpler. An emulator like QEMU is not required.
- It can be initiated by the privileged user of the hardware at any time such that only his/her root file system is updated.

- The root file system location is not important. It could be on NFS or on the SD card and it would still install the updates.
- If the root file system is on the NFS, then the hardware platform is simultaneously downloading the packages in its processor memory and writing them to the root file system over the same Ethernet interface. This makes the updates process slow. Compared to this, server side software update downloads and installs the packages on the same machine, which is comparatively faster.
- When multiple hardware platforms have initiated the software update on the same NFS server, the software update latency increases. This latency is potentially higher in a large network like that of the CMS experiment.

The reader should consider the points mentioned above when deciding upon the methods to administer software updates to the root file system.

7.1.7 Comparison of the Two Software Update Methods

A disadvantage of having two different software installation methods for the same root file system is that there can be problems if both the system administrator and the privileged user are trying to install packages on the root file system from the server side and the client side respectively and then there is a clash of these two methods. In such a situation, either of them can be prevented from installing the updates for some time while the other has a transaction lock. System administrators at CMS can avoid the problem by informing users in advance about an upcoming software update/installation session of the root file systems and that their hardware platforms would be temporarily disconnected for some time.

However, each of the above two methods have their own use cases. Only system administrators at CMS are allowed access to the NFS servers hosting the root file system and only they are authorised to make software updates to the CentOS 8 root file systems. Thus, they can initiate centralised updates to multiple root file systems on the NFS server such that they are all synchronised in terms of their content. Privileged users on the other hand are allowed to install a single or a bunch of packages they need in their userspace.

A use case for the client side installation is that the users do not always have to approach system administrators to do a server side installation or update of a single software package. If it is a just a single package, they can initiate its install or update from the client side. However, update or install of a large chunk of software packages has to be done on the server side as the client side software update/install method can be quite slow compared to server side software update. If multiple users initiate client side updates at the same time, it can result in large latency in the software installation. Here it would be of help if there is a common read-only root file system which can be updated centrally for many client platforms and only single software package installation or update is done on the read-write, board specific overlay file system (see future work in Chapter 8).

Client side software updates can be used if the network is fast enough and the number of nodes in the network is small. Centralised server side software updates are much more efficient and easy to configure for administering software updates to root file system of devices in a large network like the CMS data acquisition network.

7.2 Summary

In this chapter, two different methods were demonstrated for administering software updates to the CentOS 8 root file system situated on the NFS server. Different features of these methods were presented to the reader along with the use cases for the two software updated methods to help readers choose a software update method for their networks.

Chapter 8

Conclusion

8.1 Summary

This master thesis demonstrates the process of building a CentOS 8 Linux distribution for the Zynq UltraScale+ MPSoC and demonstrates how an automated network boot of CentOS 8 Linux distribution can be implemented for the Zynq UltraScale+ embedded controllers installed in the CMS data acquisition network. To this end, the process of building a Zynq UltraScale+ specific Linux distribution with PetaLinux Tools has been demonstrated to the reader along with the process to port the CentOS 8 mainstream kernel 4.18 for the Zynq UltraScale+ MPSoC on Xilinx ZCU102 board. This thesis also explains the process of building a CentOS 8 root file system for 64-bit ARM processors and demonstrates different methods of administering software updates to the CentOS 8 root file system when the Zynq UltraScale+ based embedded controllers would be installed in the CMS data acquisition network. This thesis also presents a comparative study of different Linux boot methods and root file system storage types to recommend a full network boot with an NFS root file system for the Zynq UltraScale+ based embedded controllers in the CMS data acquisition network.

This thesis explains the different components involved in booting Linux on the Zynq UltraScale+ and the Linux boot sequence followed by the Zynq UltraScale+. This thesis explains the Yocto Project with its underlying components and presents a qualitative comparison between the Yocto Project and the PetaLinux Tools to help the readers choose a tool-chain to build a Linux distribution as per their requirements.

This thesis demonstrates the process of configuring and building a Linux distribution with the help of PetaLinux Tools. It also elaborates upon the various modifications that were made to the PetaLinux configuration and build process (eg. setting the correct NFS version, assigning the correct kernel boot arguments and fixing the processing of DHCP response by U-Boot) to help the Linux distribution to boot completely over the network. This thesis demonstrates how the CentOS 8 kernel 4.18 was configured for Zynq UltraScale+ and how Xilinx and Zynq UltraScale+ specific drivers were added to the CentOS 8 kernel 4.18 while porting the kernel for Zynq UltraScale+ MPSoC. To have a CentOS 8 Linux distribution, it is necessary to have CentOS 8 root file system and this thesis explains the process of building a CentOS 8 root file system for 64-bit ARM processors.

For the automated network boot of CentOS 8 Linux distribution on Zynq UltraScale+, this thesis demonstrates how the DHCP, NFS and TFTP servers were configured and explains how the error associated with U-Boot's processing of the DHCP server response was fixed. To answer

the questions of system administrators and hardware developers at CERN relating to the setting, acquiring and modifying of the Ethernet MAC address during the Linux boot on Zynq UltraScale+, this thesis presents flowcharts of these processes (in the Appendix B) obtained after a thorough study of Xilinx U-Boot and Ethernet driver source codes. The “uEnv.txt” file is used during this project for remotely configuring the Linux boot process and for investigating the setting, acquiring and modifying of the Ethernet MAC address during the Linux boot process. The thesis explains the importance of the uEnv.txt file and demonstrates its use cases to help the reader exploit the versatility of this file while implementing an automated network boot.

A qualitative and quantitative analysis of different Linux boot methods for Zynq UltraScale+ has been presented in this thesis to recommend a full network boot for the Zynq UltraScale+ based embedded controllers in the CMS data acquisition network. A qualitative and quantitative analysis of SD card storage and network storage of boot files and root file system for the CentOS 8 Linux distribution has been presented in this thesis to recommend a TFTP server for boot files and NFS server for the root file system of Zynq UltraScale+ based embedded controllers in the CMS data acquisition network.

In this thesis, two different software update methods for the CentOS 8 root file system (NFS server side updates and Zynq UltraScale+ client side updates), using the DNF package manager have been demonstrated to the reader. Their features and use cases have also been presented to help the reader choose an appropriate software update for their CentOS 8 root file systems depending upon the size of their network.

Over the course of the work involved in the preparation of this master thesis, posters related to different stages of the work progress were presented at the Topical Workshop on Electronics for Particle Physics (TWEEEP) 2019 Conference and International Conference on Computing in High Energy and Nuclear Physics (CHEP) 2019 Conference. These two posters are presented in Appendix D.

8.2 Future Work

As a continuation of the work presented in this thesis, a few topics can be further researched as an extension of this project. These tasks are related to the implementation of a unique overlay root file system which is unique to a board and the development of a reliable, fault-tolerant Linux boot process.

8.2.1 Implementation of Reliable, Fault-Tolerant Linux Boot

During the boot process in the CMS network, if any board fails to boot properly, the system administrators at CMS would need to debug each case separately which can be both constraining and time-consuming. This would delay getting these boards operational. As a result, it is important to develop a reliable and fault-tolerant boot process wherein these boards can be booted to a known state such that the board is always accessible to the system-administrators for remotely debugging, analysing and resolving the problems.

There are multiple ways in which the reliable, fault-tolerant boot process can be implemented. It can be implemented in the pre-boot stage, during the loading of the boot files and also when the kernel is booting. Here are a few ways in which the fault-tolerant, reliable boot can be implemented:

1. During the pre-boot stage, if the the U-Boot is unable to get network configuration from the DHCP server for any reason, it would obscure the TFTP server IP address from the U-Boot. If the problem is pertaining to the Ethernet hardware, developers can implement a U-Boot HUSH script which would look for backup device-tree blob and kernel Image in the local storage like QSPI flash memory or SD card. This can be stored in a uEnv.txt file (see Chapter 5, section 5.6) on the local storage from where the U-Boot can import these scripts.
2. If there is a boot failure due to corrupted device-tree blob and kernel Image files or due to U-Boot's inability to load these files in the processor DRAM for hardware or software reasons, developers can develop U-Boot HUSH scripts which would be executed upon such boot failures to look for backup boot files in the TFTP server or on the local storage like QSPI flash or SD card. This can be stored in a uEnv.txt file (see Chapter 5, section 5.6) on the local storage or the TFTP server from where the U-Boot can import these scripts.
3. The kernel boot could fail for two reasons: errors in the kernel boot due to improper configuration or due to bugs in the kernel source code. For this scenario, developers can configure the Watchdog Timer kernel driver such that when the kernel panics after the boot failure, the Watchdog Timer can reset the hardware and begin the boot process from the scratch. This kernel driver should also add a U-Boot environment variable in the SPI flash memory for counting the number of successive boot failures. If this number becomes equal to a preset value (eg. five), for the next boot cycle U-Boot can fall back to another boot mechanism (eg. load a tested kernel Image and device-tree from TFTP server or local storage). Upon a successful boot, this counter can be reset to the value zero.
4. If there are failures during mounting of the root file system due to missing root file system in the specified storage or NFS path, the Watchdog Timer kernel driver mentioned in the point above can be utilised in this case as well.

8.2.2 Development of an Overlay File System

Around 1000 different boards using the Xilinx Zynq UltraScale+ MPSoC would be installed in the data acquisition network of the CMS experiment. Some of these boards would be individual installations by different experiment groups whereas some would be a part of a group of hardware platforms designed by other experiment groups. Each of these boards would require a Linux distribution with their own root file system that hosts the libraries, software packages and files that their hardware platforms would require for computing. However, an increased number of hardware platforms in the data acquisition network would demand a linear increase in the number of root file systems which will increase the amount of disk space occupied by the root file systems on the NFS servers.

One way to reduce the occupied disk space by these root file systems would be to implement an Overlay File System (OFS) [50]. An OFS allows users to have one or multiple layers of read-write file systems sitting on a common read-only file system. If the userspace of individual boards in the CMS network makes changes to a file in the common root file system, this file is stored in the OFS of that particular board. These overlay file systems are like backups for the files modified by individual boards and works in conjunction with the original common root file system. All these overlay file systems contain directories, software packages and files to which a particular board has previously made changes and is permitted to make changes in the future.

This is extremely useful in the context of the CMS Phase-II upgrade since multiple boards would have common requirements in terms of the read-only libraries, software packages, binary utilities etc. that should be present in their root file system. At the same time, the owners of individual boards or software in the userspace of these boards might need to make changes to certain files in the root file system for their individual needs and these changes can be stored in

the OFS. It would also help in having a common server side software update for different hardware platforms whereas clients can install software packages or updates to software packages to their individual Overlay file systems.

Bibliography

- [1] Esma Mobs. The CERN accelerator complex. Complexe des accélérateurs du CERN. Jul 2016. General Photo. URL: <https://cds.cern.ch/record/2197559>. xx, 4
- [2] M Jeitler. The Upgrade of the CMS Trigger System. *Journal of Instrumentation*, 9(08), Jun 2014. doi:10.1088/1748-0221/9/08/c08002. xx, 7
- [3] Xilinx Zynq UltraScale+ Software Development Guide. xx, 13, 14, 16, 17, 18, 19
- [4] Ruben Rios, Jose Onieva, and Javier Lopez. Covert Communications through Network Configuration Messages. *Computers Security*, 39:3446, 11 2013. doi:10.1016/j.cose.2013.03.004. xx, 6, 7, 49
- [5] CMS Collaboration. The Phase-2 Upgrade of the CMS DAQ Interim Technical Design Report. Technical Report CERN-LHCC-2017-014. CMS-TDR-018, CERN, Geneva, Sep 2017. This is the CMS Interim TDR devoted to the upgrade of the CMS DAQ in view of the HL-LHC running, as approved by the LHCC. URL: <https://cds.cern.ch/record/2283193>. 1, 3, 5, 6, 7, 8
- [6] Oliver Sim Brning, Paul Collier, P Lebrun, Stephen Myers, Ranko Ostojic, John Poole, and Paul Proudlock. *LHC Design Report*. CERN Yellow Reports: Monographs. CERN, Geneva, 2004. URL: <https://cds.cern.ch/record/782076>, doi:10.5170/CERN-2004-003-V-1. 3
- [7] Jérôme Lemaire. Exploring FPGA hardening solutions at the detector level for the future high luminosity phase of the CMS experiment at the LHC, 2016. URL: <https://cds.cern.ch/record/2290846>. 3
- [8] Erica Brondolin. Track reconstruction in the CMS experiment for the High Luminosity LHC, Dec 2017. Presented 17 Jan 2018. URL: <https://cds.cern.ch/record/2308020>. 3, 4
- [9] Nicolas Maximilian Rowert. Assembly and Characterization of a First Functional 2S Module for the CMS Phase-2 Upgrade at LHC, Nov 2018. Presented 25 Mar 2019. URL: <https://cds.cern.ch/record/2671895>. 3
- [10] Thomas Owen James. A hardware track-trigger for CMS at the High Luminosity LHC, 2018. URL: <https://cds.cern.ch/record/2647214>. 4, 5, 6
- [11] Davide Cieri. Development of a Level-1 Track and Vertex Finder for the Phase II CMS experiment upgrade, Jan 2018. Presented 26 Feb 2018. URL: <https://cds.cern.ch/record/2317060>. 4
- [12] Mia Tosi. The CMS trigger in Run 2. Technical Report CMS-CR-2017-340, CERN, Geneva, Oct 2017. URL: <https://cds.cern.ch/record/2290106>, doi:10.22323/1.314.0523. 4, 6
- [13] Alex Tapper. The CMS Level-1 Trigger for LHC Run II. *PoS*, ICHEP2016:242, 2016. doi:10.22323/1.282.0242. 5, 6

- [14] Daniele Trocino. The CMS High Level Trigger. *J. Phys. Conf. Ser.*, 513:012036, 2014. doi:10.1088/1742-6596/513/1/012036. 5, 6
- [15] Jean-Marc Andre, Anastasios Andronidis, Ulf Behrens, James Branson, Philipp Brummer, Olivier Chaze, Cristian Contescu, Benjamin Craigs, Sergio Cittolin, Georgiana-Lavinia Darlea, Christian Deldicque, Zeynep Demiragli, Marc Dobson, Samim Erhan, Jonathan Fulcher, Dominique Gigi, Frank Glege, Guillelmo Gomez-Ceballos, Jeroen Hegeman, and Petr Zejdl. Performance of the new DAQ system of the CMS experiment for Run-2. pages 1–4, 06 2016. doi:10.1109/RTC.2016.7543164. 6, 7
- [16] Gerry Bauer et al. The new CMS DAQ system for LHC operation after 2014 (DAQ2). *J. Phys. Conf. Ser.*, 513:012014, 2014. doi:10.1088/1742-6596/513/1/012014. 6, 7
- [17] Jeroen Hegeman et al. The CMS Timing and Control Distribution System. In *Proceedings, 2015 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC 2015): San Diego, California, United States*, page 7581984, 2016. doi:10.1109/NSSMIC.2015.7581984. 6, 7
- [18] K. Albertsson et al. A New Event Builder for CMS Run II. *J. Phys. Conf. Ser.*, 664(8):082035, 2015. doi:10.1088/1742-6596/664/8/082035. 6, 7
- [19] Jean-Marc Andr, Ulf Behrens, Andrea Bocci, James Branson, Sergio Cittolin, Diego Gomes, Georgiana-Lavinia Darlea, Christian Deldicque, Zeynep Demiragli, Marc Dobson, Nicolas Doualot, Samim Erhan, Jonathan Fulcher, Dominique Gigi, Maciej Gladki, Frank Glege, Guillelmo Gomez-Ceballos, Magnus Hansen, Jeroen Hegeman, and Petr Zejdl. The CMS Data Acquisition System for the Phase-2 Upgrade. 06 2018. 6, 7
- [20] Apollinari G., Bjar Alonso I., Brning O., Fessia P., Lamont M., Rossi L., and Tavian L. *High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1*. CERN Yellow Reports: Monographs. CERN, Geneva, 2017. URL: <https://cds.cern.ch/record/2284929>, doi:10.23731/CYRM-2017-004. 8
- [21] Xilinx Zynq UltraScale+ MPSoC Datasheet. 10, 75
- [22] Xilinx ZCU102 Evaluation Kit User Guide. 11, 12, 16
- [23] Chris Simmonds. *Mastering Embedded Linux Programming: Unleash the full potential of Embedded Linux*. Packt Publishing., 2017. 13, 15, 17, 18
- [24] PetaLinux Tools Documentation Reference Guide UG1144 (v2019.1). 13, 14, 15, 16, 25, 26, 27, 28, 34, 40, 46, 52
- [25] Alan Holt and Chi-Yu Huang. *Embedded Operating Systems. A Practical Approach*. Springer, 2018. 13, 15
- [26] Christopher Hallinan. *Embedded Linux Primer*. Prentice Hall, 2019. 13, 14, 15, 18
- [27] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. OReilly, 2006. 15
- [28] RFC 1541 - Dynamic Host Configuration Protocol. URL: <http://www.faqs.org/rfcs/rfc1541.html>. 49
- [29] The TFTP Protocol (Revision 2). URL: <https://tools.ietf.org/html/rfc1350>. 50
- [30] Xilinx. U-Boot BOOTP source code. URL: <https://github.com/Xilinx/u-boot-xlnx/blob/7412151d78bd688cf99ee7e46949362d1fe24873/net/bootp.c>. 50
- [31] Xilinx. U-Boot README. URL: <https://github.com/Xilinx/u-boot-xlnx/blob/7412151d78bd688cf99ee7e46949362d1fe24873/README>. 51

- [32] Xilinx. U-Boot Ethernet source code. URL: <https://github.com/Xilinx/u-boot-xlnx/blob/7412151d78bd688cf99ee7e46949362d1fe24873/net/eth-uclass.c>. 52
- [33] Xilinx. U-Boot bootm source code. URL: <https://github.com/Xilinx/u-boot-xlnx/blob/7412151d78bd688cf99ee7e46949362d1fe24873/arch/arm/lib/bootm.c>. 58
- [34] Xilinx. U-Boot Kernel Image supporting function source code. URL: <https://github.com/Xilinx/u-boot-xlnx/blob/3ccf78ee29093c00e3144b279759a5532c4e553a/common/image.c#L1487>. 58
- [35] Xilinx. U-Boot Image and Device-Tree interaction source code. URL: <https://github.com/Xilinx/u-boot-xlnx/blob/7412151d78bd688cf99ee7e46949362d1fe24873/common/image-fdt.c#L464>. 58
- [36] Xilinx. U-Boot Device-Tree supporting function source code. URL: https://github.com/Xilinx/u-boot-xlnx/blob/7412151d78bd688cf99ee7e46949362d1fe24873/common/fdt_support.c#L482. 58
- [37] Xilinx. Cadence Ethernet Driver. URL: <https://github.com/Xilinx/linux-xlnx/tree/8807ecf79df2330d273b65e4a655d7f19a04aef8/drivers/net/ethernet/cadence>. 59
- [38] Xilinx. Networking OF Driver, Oct 2018. URL: <https://github.com/Xilinx/linux-xlnx/tree/8807ecf79df2330d273b65e4a655d7f19a04aef8/drivers/of>. 59
- [39] NFS Version 3 Protocol Specification. URL: <https://tools.ietf.org/html/rfc1813>. 60
- [40] Network File System (NFS) Version 4 Protocol. 60
- [41] Network File System (NFS) Version 4 Minor Version 2 Protocol. URL: <https://tools.ietf.org/html/rfc7862>. 60
- [42] Kingston Canvas React Class 10 UHS-I SD Card. URL: <https://www.kingston.com/us/memory-cards/canvas-react-microsd-card>. 65, 71, 73, 74, 75
- [43] TTL TEKNOPACK NUC5i5MYHE Desktop Product Specification. 65
- [44] Bus Speed (Default Speed/High Speed/UHS/SD Express) - SD Association. URL: https://www.sdcard.org/developers/overview/bus_speed/index.html. 65, 74, 77
- [45] Kingston High Endurance Class 10 UHS-1 SD Card. URL: <https://www.kingston.com/us/memory-cards/high-endurance-microsd-card>. 74, 75
- [46] Samsung UHS-1 Pro Endurance SD Card. URL: https://www.samsung.com/semiconductor/global.semi.static/Samsung_Data_sheet_2018_PRO_Endurance.pdf. 74, 75
- [47] Intel SSD D3-S4510 Series (960GB, 2.5in SATA 6Gb/s, 3D2, TLC) Product Specifications. URL: <https://ark.intel.com/content/www/us/en/ark/products/134912/intel-ssd-d3-s4510-series-960gb-2-5in-sata-6gb-s-3d2-tlc.html>. 74, 75
- [48] The PowerEdge R740 Rack Server. URL: <https://www.dell.com/en-us/work/shop/povw/poweredge-r740>. 74
- [49] Lexar Professional 633x SDHC/SDXC UHS-I SD cards. URL: https://www.lexar.com/portfolio_page/professional-633x-sdhcsdxc-uhs-i-cards/#specifications. 75
- [50] Overlay File System. URL: <https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt>. 86

Appendices

Appendix A

Script for Extracting, Patching, Configuring and Building the CentOS 8 Kernel 4.18

```
#!/bin/bash

#Ensure you are in a CentOS 8 environment to install the correct RPM utilities and
  extract the kernel source.

#Make sure you have enough space in the directory where this script is executed.
  User home directory is recommended.

cd /home/kmor

mkdir CentOS8

#Following variable specifies path to store the CentOS kernel build.
KERNEL=/home/kmor/CentOS8

sudo yum install yum-utils rpm-build

#Download the CentOS kernel 4.18 source RPM. Make sure you are downloading the
  desired version by modifying the following variable value.
VERSION=kernel-4.18.0-80.1.2.el8_0.src.rpm

sudo wget http://vault.centos.org/8.0.1905/BaseOS/Source/SPackages/$VERSION

#Download the kernel's build dependencies before it is extracted and patched.
  Alternatively, refer to build dependencies of PetaLinux Tools and install them
  manually.

sudo yum-builddep kernel

#Once the kernel source has been downloaded, it needs to be extracted and patched.

rpm -ivh ./VERSION

#The RPM is extracted by default to the rpmbuild/SOURCES directory in the user home
  folder. Change to the rpmbuild/SPECS directory.

cd rpmbuild/SPECS

#Patch the source code as per the kernel.spec file specifications

sudo rpmbuild -bp kernel.spec

#The source code is patched and can be seen in the rpmbuild/BUILD directory.
```

```

cd ../BUILD

# Now its time to configure and build the kernel. Install the binutils and GCC
  compiler for 64-bit ARM architectures.

sudo yum install binutils-aarch64-linux-gnu gcc-aarch64-linux-gnu

cd kernel-4.18.0-80.1.2.el8_0

# Configure the kernel and specify the output directory. A muneconfig screen shall
  appear. Choose the platform and features that you desire.

make ARCH=arm64 O=$KERNEL defconfig menuconfig

#Once the configuration is over, time to build the kernel Image and device-tree
  blob as per chosen platform.

cd $KERNEL

make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image dtbs

echo Build Over

```

Listing A.1: Script to extract, patch, configure and build a CentOS 8 kernel 4.18

Git Repository Links

The following web-link points to the GitLab repository of CERN holding the CentOS 8 4.18 kernel source and build images:

<https://gitlab.cern.ch/hardware/zynq/centos8/kernel>

The following web-link points to the GitLab repository of CERN holding the important Linux network boot files as well as instructions to replicate a network-boot:

<https://gitlab.cern.ch/hardware/zynq/petalinux/network-boot>

Scripts to Configure DNF Repositories (dnf.conf) and Build CentOS 8 Root File System

```

[arm64-epel]
name=Epel rebuild for arm64
baseurl=https://dl.fedoraproject.org/pub/epel/$releasever/Everything/aarch64/
enabled=1
gpgcheck=0

[arm-epel]
name=Epel rebuild for armhfp
baseurl=https://armv7.dev.centos.org/repo/epel-pass-1/
enabled=1
gpgcheck=0

[centos-base]
name=CentOS-$releasever - Base

```

```

baseurl=http://mirror.centos.org/centos/$releasever/BaseOS/$basearch/os/
gpgcheck=0
enabled=1

[centos-updates]
name=CentOS-$releasever - Updates
baseurl=http://mirror.centos.org/centos/$releasever/BaseOS/$basearch/os/
gpgcheck=0
enabled=1

[centos-extras]
name=CentOS-$releasever - Extras
baseurl=http://mirror.centos.org/centos/$releasever/extras/$basearch/os/
gpgcheck=0
enabled=1

[centos-appstream]
name=CentOS-$releasever - AppStream
baseurl=http://mirror.centos.org/centos/$releasever/AppStream/$basearch/os/
gpgcheck=0
enabled=1

[centos-powertools]
name=CentOS-$releasever - PowerTools
baseurl=http://mirror.centos.org/centos/$releasever/PowerTools/$basearch/os/
gpgcheck=0
enabled=1

[centos-centosplus]
name=CentOS-$releasever - centosplus
baseurl=http://mirror.centos.org/centos/$releasever/centosplus/$basearch/os/
gpgcheck=0
enabled=1

[l1ct-yocto]
name=l1ct-yocto
baseurl=file:///home/ppapageo/yocto/rocko/poky/build/tmp/deploy/rpm/aarch64/
gpgcheck=0
enabled=1

```

Listing A.2: Configuration of repositories in `dnf.conf` for sourcing CentOS 8 packages for the CentOS 8 root file system build script

```

#####
## Script based on original script written by Matthias Wittgen ##
#####

import os
import sys
import subprocess
import logging
import shutil
import argparse
import crypt
import augeas

dirname, filename = os.path.split(os.path.abspath(__file__))

dnf_conf=dirname+'/dnf.conf'
print(dnf_conf)
qemu_bins=dirname
etc=dirname

def run_dnf(rootfs, inst, what):
    cmd=[ "dnf", "-y", "--skip-broken", "-c", dnf_conf, "--releasever=8", "--",
          forcearch="+arch", "--repo=centos-base,centos-updates,centos-extras","+epel,"

```

```

        "--verbose", "--installroot="+rootfs, inst ] + what
    print(cmd)
    try:
        process=subprocess.Popen(cmd,stdout=subprocess.PIPE,shell=False)
        while process.poll() is None:
            output = process.stdout.readline()
            if output:
                print(output.strip().decode('utf-8'))
    except:
        return

parser = argparse.ArgumentParser(description='Tool to build a root filesystem for
Centos Linux ARM')

FORMAT = '%(levelname)s : %(message)s'

parser.add_argument('-v','--verbose',action='store_true',
                    help='verbose output')
parser.add_argument('-r','--root',nargs=1,
                    help='directory of new rootfs')
parser.add_argument('-a','--arch',nargs=1,
                    help='architecture of target')
parser.add_argument('-e','--extra',nargs=1,
                    help='file with a list of extra packages to be installed')
args = vars(parser.parse_args())
if args['verbose']:
    logging.basicConfig(format=FORMAT,stream=sys.stdout, level=logging.DEBUG)

if args['root'] is not None:
    rootdir=args['root'][0]
    print (rootdir)
else:
    print("Use --root=<dir> to set new rootfs directory")
    exit(-1)
if args['arch'] is not None:
    arch=args['arch'][0]
    print("Building for ",arch)
else:
    print("Use --arch=<arch> to specify build architecture")
    exit(-1)
if arch not in ["armv7hl","aarch64"]:
    print("Invalid CPU architecture")
    exit(-1)
if args['extra'] is not None:
    text_file = open(args['extra'][0],"r")
    lines=[]
    for x in text_file:
        x=x.replace("\n", " ")
        lines.append(x)

    text_file.close()
if os.getuid()!=0:
    print ("Program must to run as superuser")
    print ("Relaunching as: sudo ", " ".join(sys.argv))
    os.execvp("sudo",[
        "sudo",
        "PATH="+os.getenv("PATH"),
        "LD_LIBRARY_PATH="+os.getenv("LD_LIBRARY_PATH"),
        "PYTHONPATH="+os.getenv("PYTHONPATH"),
        ]+sys.argv)
    exit(0)

if arch == "armv7hl":
    print ("Using qemu-arm-static")
elif arch == "aarch64":

```

```

print ("Using qemu-aarch64-static")

if arch == "aarch64":
    epel="arm64-epel"
elif arch == "armv7hl":
    epel="arm-epel"

run_dnf(rootdir,"clean",["all"])
run_dnf(rootdir,"update",[" "])
print ("Running dnf: group install")
run_dnf(rootdir,"groupinstall",['Minimal Install'])
if args['extra'] is not None:
    print("Installing user defined packages...")
    run_dnf(rootdir,"install",lines)

rootpwd=crypt.crypt("centos8forcms", crypt.mksalt(crypt.METHOD_SHA512))
aug=augeas.Augeas(root=rootdir)
aug.set("/files/etc/shadow/root/password",rootpwd)
aug.set("/files/etc/sysconfig/selinux/SELINUX","disabled")
aug.save()
aug.close()
if arch=="armv7hl":
    os.remove(rootdir+"/etc/yum.repos.d/CentOS-armhfp-kernel.repo")

```

Listing A.3: Script to build the CentOS 8 root file system with the help of repositories pointed by the dnf.conf file

Differences in CentOS 8 kernel 4.18 default kernel configuration and Xilinx kernel 4.19's Zynq UltraScale+ default kernel configuration

```

--- defconfig 2019-11-15 11:35:27.808712000 +0100
+++ xilinx_zynqmp_defconfig 2019-10-09 18:51:21.125276000 +0200
@@ -1,686 +1,399 @@
CONFIG_SYSVIPC=y
CONFIG_POSIX_MQUEUE=y
CONFIG_AUDIT=y
-CONFIG_NO_HZ_IDLE=y
+CONFIG_NO_HZ=y
CONFIG_HIGH_RES_TIMERS=y
-CONFIG_IRQ_TIME_ACCOUNTING=y
CONFIG_BSD_PROCESS_ACCT=y
-CONFIG_BSD_PROCESS_ACCT_V3=y
CONFIG_TASKSTATS=y
CONFIG_TASK_DELAY_ACCT=y
CONFIG_TASK_XACCT=y
CONFIG_TASK_IO_ACCOUNTING=y
CONFIG_IKCONFIG=y
CONFIG_IKCONFIG_PROC=y
-CONFIG_NUMA_BALANCING=y
-CONFIG_MEMCG=y
-CONFIG_MEMCG_SWAP=y
-CONFIG_BLK_CGROUP=y
-CONFIG_CGROUP_PIDS=y
-CONFIG_CGROUP_HUGETLB=y
-CONFIG_CPUSETS=y
-CONFIG_CGROUP_DEVICE=y
-CONFIG_CGROUP_CPUACCT=y
-CONFIG_CGROUP_PERF=y
-CONFIG_USER_NS=y
-CONFIG_SCHED_AUTOGROUP=y

```

```

+CONFIG.LOG_BUF_SHIFT=16
+CONFIG.CGROUPTS=y
  CONFIG.BLK_DEV_INITRD=y
-CONFIG.KALLSYMS_ALL=y
+CONFIG.EMBEDDED=y
  # CONFIG.COMPAT_BRK is not set
+CONFIG.SLAB=y
  CONFIG.PROFILING=y
-CONFIG.JUMP_LABEL=y
-CONFIG.MODULES=y
-CONFIG.MODULE_UNLOAD=y
-CONFIG.ARCH.SUNXI=y
-CONFIG.ARCH.ALPIRE=y
-CONFIG.ARCH.BCM2835=y
-CONFIG.ARCH.BCMIPROC=y
-CONFIG.ARCH.BERLIN=y
-CONFIG.ARCH.BRCMSTB=y
-CONFIG.ARCH.EXYNOS=y
-CONFIG.ARCH.LAYERSCAPE=y
-CONFIG.ARCH.LG1K=y
-CONFIG.ARCH.HISI=y
-CONFIG.ARCH.MEDIATEK=y
-CONFIG.ARCH.MESON=y
-CONFIG.ARCH.MVEBU=y
-CONFIG.ARCH.QCOM=y
-CONFIG.ARCH.ROCKCHIP=y
-CONFIG.ARCH.SEATTLE=y
-CONFIG.ARCH.SYNQUACER=y
-CONFIG.ARCH.RENESAS=y
-CONFIG.ARCH.R8A7795=y
-CONFIG.ARCH.R8A7796=y
-CONFIG.ARCH.R8A77965=y
-CONFIG.ARCH.R8A77970=y
-CONFIG.ARCH.R8A77980=y
-CONFIG.ARCH.R8A77990=y
-CONFIG.ARCH.R8A77995=y
-CONFIG.ARCH.STRATIX10=y
-CONFIG.ARCH.TEGRA=y
-CONFIG.ARCH.SPRD=y
-CONFIG.ARCH.THUNDER=y
-CONFIG.ARCH.THUNDER2=y
-CONFIG.ARCH.UNIPHIER=y
-CONFIG.ARCH.VEXPRESS=y
-CONFIG.ARCH.XGENE=y
-CONFIG.ARCH.ZX=y
  CONFIG.ARCH.ZYNQMP=y
  CONFIG.PCI=y
-CONFIG.PCI_IOV=y
-CONFIG.HOTPLUG_PCI=y
-CONFIG.HOTPLUG_PCI_ACPI=y
-CONFIG.PCI_AARDVARK=y
-CONFIG.PCI_TEGRA=y
-CONFIG.PCIE_RCAR=y
-CONFIG.PCI_HOST_GENERIC=y
-CONFIG.PCI_XGENE=y
-CONFIG.PCI_HOST_THUNDER_PEM=y
-CONFIG.PCI_HOST_THUNDER_ECAM=y
-CONFIG.PCIE_ROCKCHIP_HOST=m
-CONFIG.PCIE_LAYERSCAPE=y
-CONFIG.PCI_HISI=y
-CONFIG.PCIE_QCOM=y
-CONFIG.PCIE_ARMADA_8K=y
-CONFIG.PCIE_KIRIN=y
-CONFIG.PCIE_HISI_STB=y
-CONFIG.ARM64_VA_BITS_48=y
-CONFIG.SCHED_MC=y
-CONFIG.NUMA=y

```

```

-CONFIG_PREEMPT=y
-CONFIG_KSM=y
-CONFIG_MEMORY_FAILURE=y
-CONFIG_TRANSPARENT_HUGEPAGE=y
-CONFIG_CMA=y
-CONFIG_SECCOMP=y
-CONFIG_KEXEC=y
-CONFIG_CRASH_DUMP=y
-CONFIG_XEN=y
-# CONFIG_CORE_DUMP_DEFAULT_ELF_HEADERS is not set
+CONFIG_PCIE_XILINX_NWL=y
+CONFIG_NR_CPUS=8
-# CONFIG_DMI is not set
CONFIG_COMPAT=y
-CONFIG_HIBERNATION=y
-CONFIG_WQ_POWER_EFFICIENT_DEFAULT=y
+CONFIG_CPU_IDLE=y
CONFIG_ARM_CPUIDLE=y
CONFIG_CPU_FREQ=y
-CONFIG_CPU_FREQ_STAT=y
-CONFIG_CPU_FREQ_GOV_POWERSAVE=m
-CONFIG_CPU_FREQ_GOV_USERSPACE=y
-CONFIG_CPU_FREQ_GOV_ONDEMAND=y
-CONFIG_CPU_FREQ_GOV_CONSERVATIVE=m
-CONFIG_CPU_FREQ_GOV_SCHEDUTIL=y
+CONFIG_CPU_FREQ_DEFAULT_GOV_USERSPACE=y
CONFIG_CPUFREQ_DT=y
-CONFIG_ACPI_CPPC_CPUFREQ=m
-CONFIG_ARM_ARMADA37XX_CPUFREQ=y
-CONFIG_ARM_BIG_LITTLE_CPUFREQ=y
-CONFIG_ARM_SCPI_CPUFREQ=y
-CONFIG_ARM_TEGRA186_CPUFREQ=y

```

Listing A.4: Snapshot of differences between CentOS 8 kernel 4.18 defconfig and xilinx_zynqmp_defconfig

Here “-” refers to the properties present in the original CentOS 8 kernel 4.18 default configuration. Similarly, “+” refers to the properties present in the original Zynq UltraScale+ default kernel configuration obtained from Xilinx.

Appendix B

DHCP Server Configuration

```
#
# DHCP Server Configuration file.
#   see /usr/share/doc/dhcp*/dhcpd.conf.example
#   see dhcpd.conf(5) man page
#
# dhcpd.conf
#
# Sample configuration file for ISC dhcpd
#

ignore unknown-clients;
default-lease-time 600;
max-lease-time 7200;
log-facility local7;

# A slightly different configuration for an internal subnet.
subnet 128.141.174.0 netmask 255.255.255.0 {
    default-lease-time 86400;
    max-lease-time 86400;
    option routers 128.141.174.1;
    option broadcast-address 128.141.174.255;
    option subnet-mask 255.255.255.0;
    interface "enp0s25";
}

host daqlab40-zcu102-01 {
    hardware ethernet 08:00:30:F4:03:37;
    fixed-address 128.141.174.218;
    next-server 128.141.174.229;
    server-name "128.141.174.229";
    filename "zcu102/uEnv.txt";
    option root-path "128.141.174.229:/home/kmor/nfs/zcu102_vanilla_centos/,tcp,v3";
}
```

Listing B.1: DHCP server configuration in the file dhcpd.conf in the path /etc/dhcp

Here, the "filename" points to the "uEnv.txt" file explained in section 5.6 of Chapter 5. "default-lease-time" is the default lease time (in seconds) of network configuration given to the client by the DHCP server. "interface" points to the Ethernet interface to be used on the machine hosting the DHCP server. "fixed-address" is the fixed IP address provided to the client with a particular "ethernet" MAC address. "server-name" points to the TFTP server IP address. "root-path" points to the NFS root file system server path on the NFS server with IP address 128.141.174.229.

TFTP Server Configuration

```
# default: off
# description: The tftp server serves files using the trivial file transfer \
# protocol. The tftp protocol is often used to boot diskless \
# workstations, download configuration files to network-aware printers, \
# and to start the installation process for some operating systems.
service tftp
{
    socket_type          = dgram
    protocol             = udp
    wait                 = yes
    user                 = root
    server               = /usr/sbin/in.tftpd
    server_args          = -c -s /home/kmor/tftp
    disable              = no
    per_source            = 11
    cps                  = 100 2
    flags                = IPv4
}
```

Listing B.2: TFTP server configuration in the file tftp in the path /etc/xinetd.d

```
# default: off
# description: The tftp server serves files using the trivial file transfer \
# protocol. The tftp protocol is often used to boot diskless \
# workstations, download configuration files to network-aware printers, \
# and to start the installation process for some operating systems.
service tftp
{
    [Unit]
    Description=Tftp Server
    Requires=tftp.socket
    Documentation=man:in.tftpd

    [Service]
    ExecStart=/usr/sbin/in.tftpd -c -s /home/kmor/tftp
    StandardInput=socket

    [Install]
    Also=tftp.socket
    WantedBy=multi-user.target
}
```

Listing B.3: TFTP service configuration in the file tftp.service in the path /usr/lib/systemd/system

```
#!/bin/bash

#Make a directory to store the files on TFTP server

TFTP=/home/kmor/tftp

mkdir $TFTP

#Install the necessary packages

sudo yum install tftp tftp-server xinetd

#Configure the TFTP server and service as per the Listing 5 and Listing 6. Once the
configuration is done, run the following commands

#Make the directory read-writable for everyone
```

```

sudo mkdir chmod -R 777 $TFTP

#Make exceptions in firewall

sudo firewall-cmd --zone=public --permanent --add-port=69/udp
sudo firewall-cmd --zone=public --add-service=tftp --permanent
sudo firewall-cmd --reload

#Enable the TFTP server , service and Firewall

sudo systemctl enable tftp xinetd firewalld

#Restart the TFTP server , service and Firewall

sudo systemctl restart tftp xinetd firewalld

echo All done

```

Listing B.4: Script to setup the TFTP server

Snapshot of “u-boot_bsp.tcl” Script of PetaLinux Tools Showing the BOOTP/DHCP Options

```

proc uboot_common {fid kconfig_fid} {
    global kconfig_dict target_cpu
    set cpu_arch [get_sw_proc_arch $target_cpu]
    # TODO: check if nor flash present in the system
    # spi only - define CONFIG_SYS_NO_FLASH

    set data ""

    /* BOOTP options */
    /* #define CONFIG_BOOTP_SERVERIP */
    #define CONFIG_BOOTP_BOOTFILESIZE
    #define CONFIG_BOOTP_BOOTPATH
    #define CONFIG_BOOTP_GATEWAY
    #define CONFIG_BOOTP_HOSTNAME
    #define CONFIG_BOOTP_MAY_FAIL
    #define CONFIG_BOOTP_DNS
    #define CONFIG_BOOTP_SUBNETMASK
    #define CONFIG_BOOTP_PXE

    /*Command line configuration */
    #define CONFIG_CMDLINE_EDITING
    #define CONFIG_AUTO_COMPLETE

    #define CONFIG_IMAGE_FORMAT_LEGACY
    #define CONFIG_SUPPORT_RAW_INTRD

    /* Miscellaneous configurable options */
    #define CONFIG_SYS_CBSIZE 2048/* Console I/O Buffer Size */
    #define CONFIG_SYS_PBSIZE (CONFIG_SYS_CBSIZE + \
        sizeof(CONFIG_SYS_PROMPT) + 16)
    #define CONFIG_SYS_BARGSIZE CONFIG_SYS_CBSIZE

    /* Use the HUSH parser */
    #define CONFIG_SYS_PROMPT_HUSH_PS2 "> \"

    #define CONFIG_ENV_VARS_UBOOT_CONFIG

```

```

#define CONFIG_ENV_OVERWRITE /* Allow to overwrite the u-boot environment
    variables */

#define CONFIG_LMB

/* FDT support */
#define CONFIG_DISPLAY_BOARDINFO_LATE

"
    uboot_set_kconfig_value $kconfig_fid "BOOTARGS" "n"
    uboot_set_kconfig_value $kconfig_fid "USE_BOOTARGS" "n"
    if {[string equal -nocase $cpu_arch "microblaze"] == 1} {
        append data "
/* architecture dependent code */
#define CONFIG_SYS_USR_EXCEP /* user exception */
#define CONFIG_SYS_HZ 1000

/* Boot Argument Buffer Size */
#define CONFIG_SYS_MAXARGS 32 /* max number of command args */
#define CONFIG_SYS_LONGHELP

"
    } elseif {[string equal -nocase $cpu_arch "armv7"] == 1} {
        append data "
/* architecture dependent code */
#define CONFIG_SYS_HZ 1000

/* Boot Argument Buffer Size */
#define CONFIG_SYS_MAXARGS 32 /* max number of command args */
#define CONFIG_SYS_LONGHELP

```

Listing B.5: Snapshot of u-boot_bsp.tcl script used by PetaLinux Tools to configure the U-Boot

NFS Server Configuration

```

#!/bin/bash

#Make a directory to store the files on NFS server
NFS=/home/kmor/nfs

mkdir $NFS

#Install the necessary packages
sudo yum install nfs-utils

#Make the directory read-writable for everyone
sudo mkdir chmod -R 777 $NFS

#Make exceptions in firewall

sudo firewall-cmd --permanent --zone=public --add-port=111/udp --add-port=111/tcp
--add-port=2049/udp --add-port=2049/tcp

sudo firewall-cmd --zone=public --add-service=nfs --add-service=mountd --add-
service=rpc-bind --permanent

sudo firewall-cmd --reload

#Enable the NFS server, services and Firewall

sudo systemctl enable nfs-server nfs-lock nfs-idmap mountd rpcbind

```

```
#Restart the NFS server , service and Firewall

sudo systemctl restart nfs-server nfs-lock nfs-idmap mountd firewalld rpcbind

echo All done
```

Listing B.6: Script to setup the NFS server

Once the server has been configured, populate the NFS export in the file “exports” in the path /etc.

```
/home/kmor/nfs *(rw,sync,no-root-squash,no-all-squash)
/home/kmor/CentOS8 *(rw,sync,no-root-squash,no-all-squash)
```

Listing B.7: Contents of exports file exporting the NFS file system

Default U-Boot Environment

```
ZynqMP> run tftpimportbootenv
Importing environment from TFTP Server
Using ethernet@ff0e0000 device
TFTP from server 128.141.174.229; our IP address is 128.141.174.218
Filename 'zcu102/uEnv.txt'.
Load address: 0x100000
Loading: #
          75.2 KiB/s
done
Bytes transferred = 1779 (6f3 hex)
Saving Environment to SPI Flash... SF: Detected n25q512a with page size 512 Bytes,
      erase size 128 KiB, total 128 MiB
Erasing SPI flash... Writing to SPI flash... done
OK
ZynqMP> printenv
arch=arm
autoload=no
baudrate=115200
board=zynqmp
board_name=zynqmp
boot_img=BOOT.BIN
boot_targets=mmc0
bootcmd=run default_bootcmd
bootdelay=4
bootenv=zcu102/uEnv.txt
bootenvsize=0x40000
bootenvstart=0x1e00000
bootfile=zcu102/uEnv.txt
clobstart=0x10000000
console=console=ttyPS0,115200
cp_dtb2ram=mmcinfo && fatload mmc ${sdbootdev} ${clobstart} ${dtb_img}
cp_kernel2ram=mmcinfo && fatload mmc ${sdbootdev} ${netstart} ${kernel_img}
cpu=armv8
default_bootcmd=dhcp;run tftpimportbootenv; run download_fdt;run download_kernel;
      run netbooti
dfu_mmc_info=set dfu_alt_info ${kernel_image} fat 0 1\\;dfu_mmc=run dfu_mmc_info &&
      dfu 0 mmc 0
dfu_ram=run dfu_ram_info && dfu 0 ram 0
dfu_ram_info=setenv dfu_alt_info image.ub ram $netstart 0x1e00000
download_fdt=tftpboot ${clobstart} ${dtb_img};
download_kernel=tftpboot ${netstart} ${kernel_img};
download_kernel_package=tftpboot ${netstart} ${kernel_package};
dtb_img=system.dtb
dtb_img_2=zcu102/PetaLinux_Basic/system.dtb
```

```

dtbnetstart=0x23fff000
eraseenv=sf probe 0 && sf erase ${bootenvstart} ${bootenvsize}
ethact=ethernet@ff0e0000
ethaddr=08:00:30:f4:03:37
fault=echo ${img} image size is greater than allocated place - partition ${img} is
NOT UPDATED
fdtcontroladdr=7dd9ec88
fdtfile=xilinx/zynqmp-zcu102-rev1.0.dtb
fileaddr=100000
filesize=609
gatewayip=128.141.174.1
importbootenv=echo "Importing environment from SD ..."; env import -t ${
loadbootenv_addr} $filesize
initrd_high=79000000
install_boot=mmcinfo && fatwrite mmc ${sdbootdev} ${clobstart} ${boot_img} ${
filesize}
install_jffs2=sf probe 0 && sf erase ${jffs2start} ${jffs2size} && sf write ${
clobstart} ${jffs2start} ${filesize}
install_kernel=mmcinfo && fatwrite mmc ${sdbootdev} ${clobstart} ${kernel_img} ${
filesize}
ipaddr=128.141.174.218
jffs2_img=rootfs.jffs2
kernel_img=Image
kernel_package=zcu102/PetaLinux_Basic/image.ub
load_boot=tftpboot ${clobstart} ${boot_img}
load_dtb=tftpboot ${clobstart} ${dtb_img}
load_jffs2=tftpboot ${clobstart} ${jffs2_img}
load_kernel=tftpboot ${clobstart} ${kernel_img}
loadaddr=0x10000000
loadbootenv=load mmc $sdbootdev:$partid ${loadbootenv_addr} ${bootenv}
loadbootenv_addr=0x00100000
mac_addr_testing=run tftpimportbootenv;run download_kernel;run download_fdt;run
update_fdt;fdt print ethernet0;print ethaddr;run netbooti
modeboot=sdboot
nc=setenv stdout nc;setenv stdin nc;
netboot=tftpboot ${netstart} ${kernel_img} && bootm
netbooti=booti ${netstart} - ${clobstart}
netbootm=bootm ${netstart}
netmask=255.255.255.0
netstart=0x1000b000
preboot=echo U-BOOT for xilinx-zcu102-2019_1;setenv autoload no;echo U-BOOT for
CERN CMS;setenv ethmacskip;setenv ethlmacskip
psserial0=setenv stdout ttyPS0;setenv stdin ttyPS0
reset_reason=SOFT
rootpath=128.141.174.229:/home/kmor/nfs/zcu102_vanilla-centos8/,tcp,v3
sd_uEnvtxt_existence_test=test -e mmc $sdbootdev:$partid /uEnv.txt
sd_update_dtb=echo Updating dtb from SD; mmcinfo && fatload mmc ${sdbootdev}:1 ${
clobstart} ${dtb_img} && run install_dtb
sd_update_jffs2=echo Updating jffs2 from SD; mmcinfo && fatload mmc ${sdbootdev}:1
${clobstart} ${jffs2_img} && run install_jffs2
sdbootdev=0
serial=setenv stdout serial;setenv stdin serial
serverip=128.141.174.229
soc=zynqmp
stderr=serial@ff000000
stdin=serial@ff000000
stdout=serial@ff000000
test_crc=if imi ${clobstart}; then run test_img; else echo ${img} Bad CRC - ${img}
is NOT UPDATED; fi
test_img=setenv var "if test ${filesize} -gt ${psize}; then run fault; else run ${
installcmd}; fi"; run var; setenv var
tftpimportbootenv=echo Importing environment from TFTP Server; tftpboot ${
loadbootenv_addr} ${bootenv}; env import -t ${loadbootenv_addr} $filesize;
saveenv
thor_mmc=run dfu_mmc.info && thordown 0 mmc 0
thor_ram=run dfu_ram.info && thordown 0 ram 0

```

```

uenvboot=if run sd_uEnvtxt_existence_test; then run loadbootenv; echo Loaded
environment from ${bootenv}; run importbootenv; fi; if test -n $uenvcmd; then
echo Running uenvcmd ...; run uenvcmd; fi
update_boot=setenv img boot; setenv psize ${bootsize}; setenv installcmd "
install_boot"; run load_boot ${installcmd}; setenv img; setenv psize; setenv
installcmd
update_dtb=setenv img dtb; setenv psize ${dtbsize}; setenv installcmd "install_dtb"
; run load_dtb test_img; setenv img; setenv psize; setenv installcmd
update_ethaddr=i2c dev 5;i2c mw 54 20.1 08;i2c mw 54 21.1 00;i2c mw 54 22.1 30;i2c
mw 54 23.1 f4;i2c mw 54 24.1 03;i2c mw 54 25.1 37
update_fdt=fdt addr ${clobstart};fdt set ethernet0 local-mac-address "08:00:30:f4
:03:37";fdt rm /amba/mmc@ff170000 no-1-8-v;
update_jffs2=setenv img jffs2; setenv psize ${jffs2size}; setenv installcmd "
install_jffs2"; run load_jffs2 test_img; setenv img; setenv psize; setenv
installcmd
update_kernel=setenv img kernel; setenv psize ${kernelsize}; setenv installcmd "
install_kernel"; run load_kernel ${installcmd}; setenv img; setenv psize;
setenv installcmd
vendor=xilinx

```

Listing B.8: The U-Boot environment variables after loading the uEnv.txt from the TFTP server(see Listing B.1 for information offered by DHCP server)

Setting of MAC Address in the Ethernet Hardware

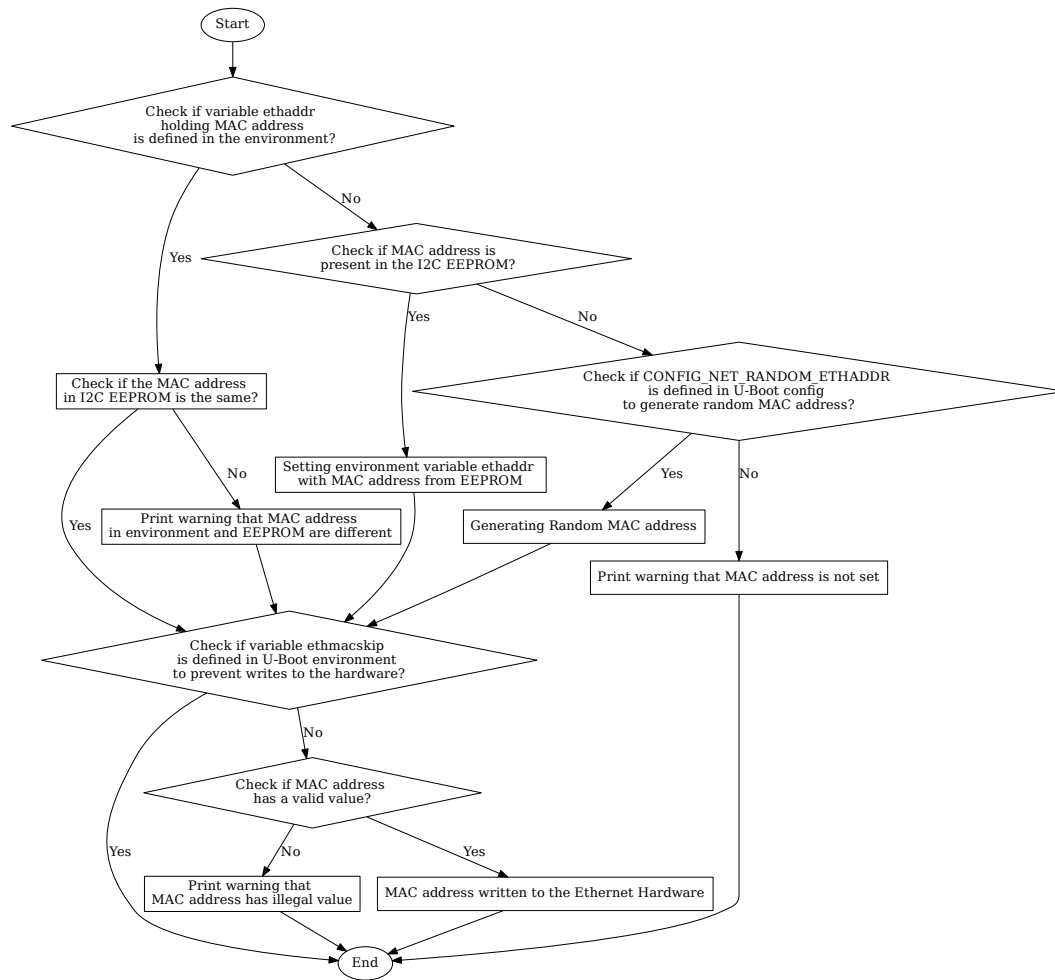


Figure B.1: Process to set MAC address in the Ethernet Hardware

In Figure B.1, readers can follow the procedure of setting of the MAC address in the Ethernet hardware which has been mentioned in the bullet points of the section 5.5 of Chapter 5 to understand how the Ethernet MAC address is set in the Ethernet hardware.

Setting of the MAC Address in the Device-Tree

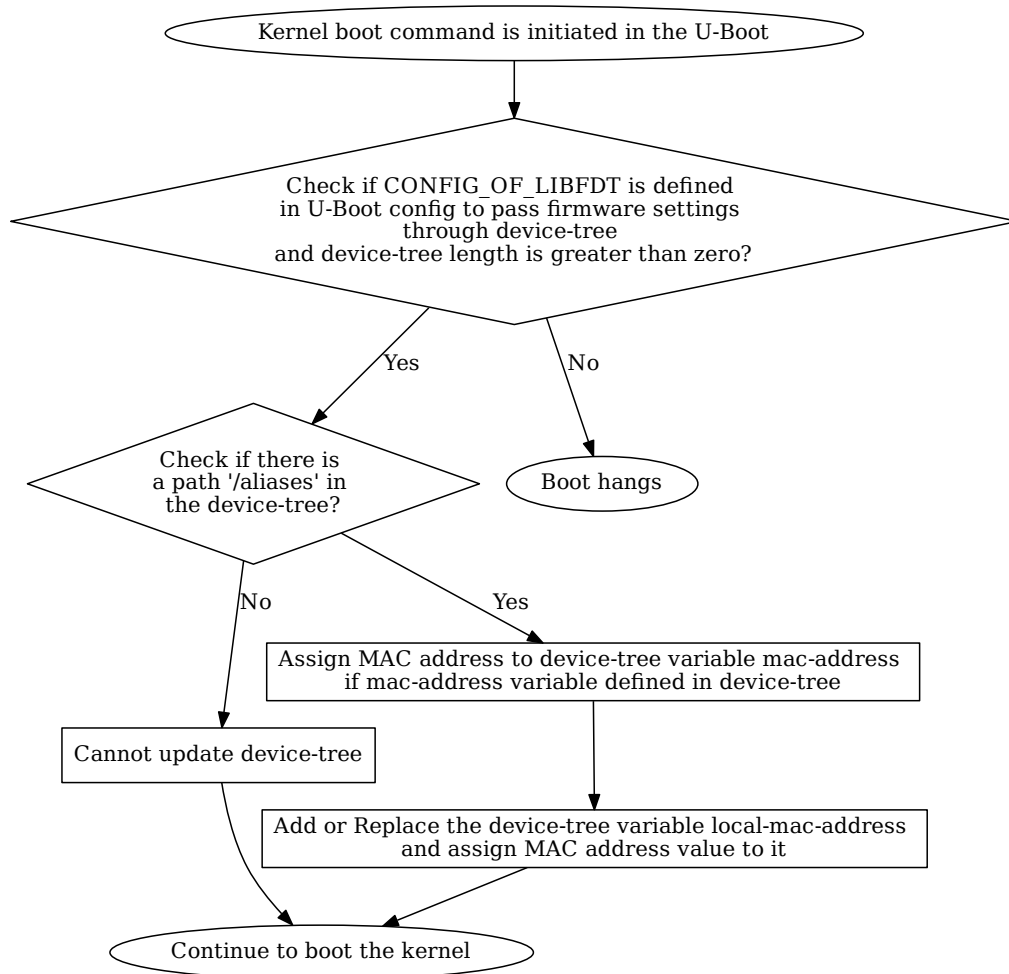


Figure B.2: Modification of MAC address in the device-tree by the U-Boot

In Figure B.2, readers can follow the procedure of setting of the MAC address in the device-tree mentioned in the bullet points of the section 5.7 of Chapter 5 to understand how the Ethernet MAC address is set in the device-tree.

Acquisition of the MAC Address by the Linux Kernel

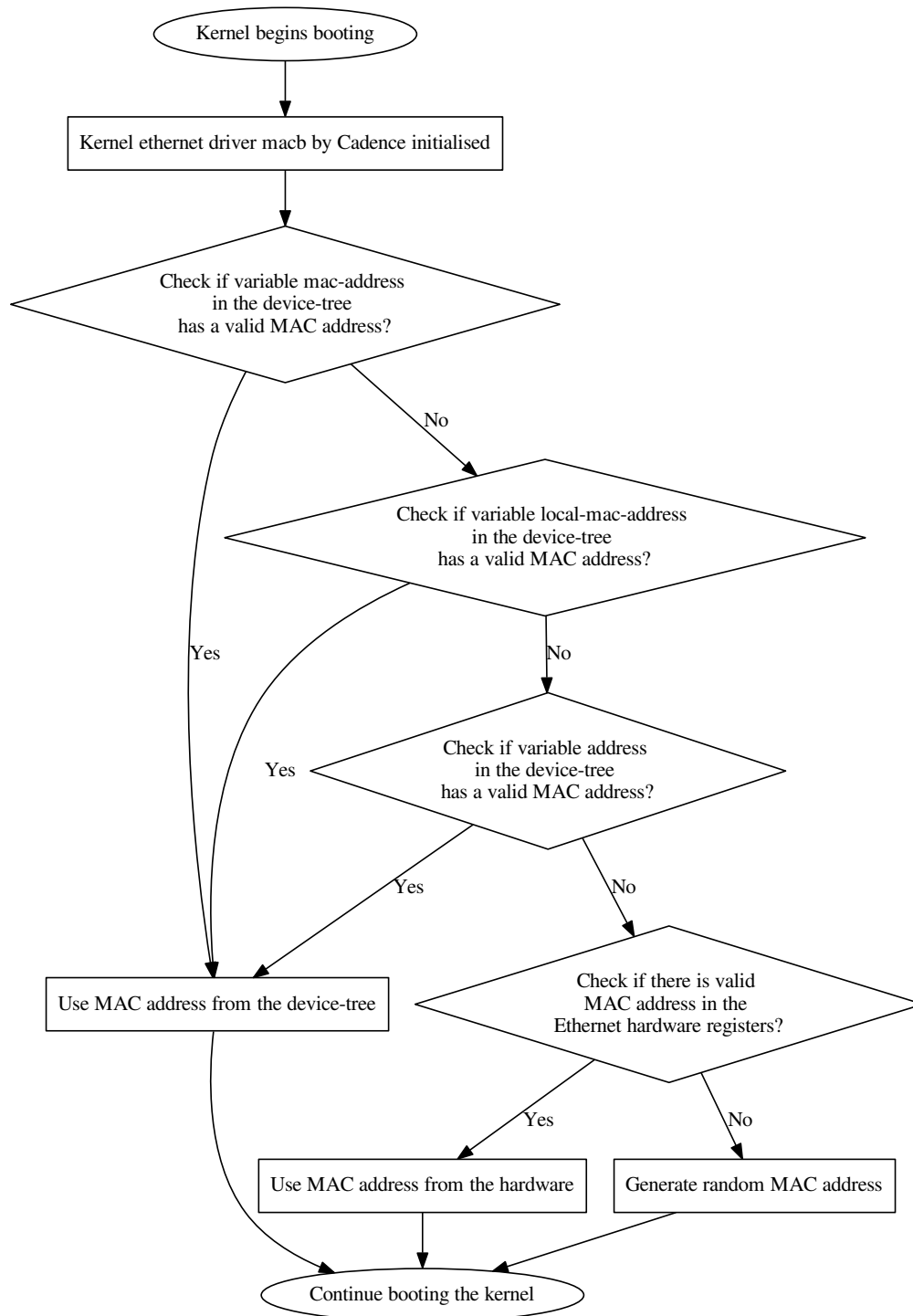


Figure B.3: Acquisition of MAC address by the Linux Kernel

In Figure B.3, readers can follow the procedure of acquisition of the MAC address by the Linux kernel mentioned in the bullet points of the section 5.8 of Chapter 5 to understand how the Ethernet MAC address is acquired by the kernel.

Appendix C

Script to Test Read-Write Speeds of the Root File System Types

```
#!/bin/bash

#Delete any old test files and folders created during previous tests

echo Deleting old test folders
PATH=/home/kmor

sudo rm -r $PATH/bigFiles $PATH/smallFiles $PATH/smallRamFiles $PATH/bigRamFiles

#Create new directories for storing test files

echo Creating new directories
mkdir -p $PATH/smallFiles $PATH/smallRamFiles $PATH/bigFiles $PATH/bigRamFiles

#Create one 1 GB file for writing to the RAM disk
echo Creating One Big file of 1GB

#Use the /dev/zero functionality in the kernel to generate a file full of zeros
dd if=/dev/zero of=$PATH/bigFiles/file.txt bs=1048576 count=1024 conv=fsync

#Take 10 samples of the timing to read a 1 GB file from the root filesystem and
#write to the RAM disk
#Clear the cached data before sampling

for value in {1..10}
do
    sudo rm -rf /dev/shm/*
    echo Timing the copy of 1GB file from Root FS to RAM disk
    sync; echo 3 > /proc/sys/vm/drop_caches
    time sh -c 'cp $PATH/bigFiles/file.txt /dev/shm; sync'
done

#Take 10 samples of the timing to read a 1 GB file from the RAM disk and write to
#the root filesystem
#Clear the cached data before sampling

for value in {1..10}
do
    sudo rm -rf $PATH/bigRamFiles/*
    echo Timing the copy of 1GB file from RAM disk to Root FS
    sync; echo 3 > /proc/sys/vm/drop_caches
    time sh -c 'cp /dev/shm/file.txt $PATH/bigRamFiles; sync'
done

#Clear the RAM disk for further tests due to limited space in RAM disk
echo Deleting the big file from RAM disk
```

```

sudo rm -rf /dev/shm/*

#Create 1000 1 MB files for testing read-write speeds for small files

echo Creating 1000 files of 1 MB

for value in {1..1000}
do
    dd if=/dev/zero of=$PATH/smallFiles/file_-$value.txt bs=1024 count=1024 conv=
    fsync
done

#Take 10 samples of the timing to read 1000 1 MB files from the root file system
    and write to the RAM disk
#Clear the cached data before sampling

mkdir -p /dev/shm/smallFiles

for value in {1..10}
do
    sudo rm -rf /dev/shm/smallFiles/*
    sync; echo 3 > /proc/sys/vm/drop_caches
    echo Timing the copy of 1000 1MB files from Root FS to RAM disk
    time sh -c 'cp -a $PATH/smallFiles/. /dev/shm/smallFiles/; sync '
done

#Take 10 samples of the timing to read 1000 1 MB files from the root file system
    and write to the RAM disk
#Clear the cached data before sampling

for value in {1..10}
do
    sudo rm -rf $PATH/smallRamFiles/*
    sync; echo 3 > /proc/sys/vm/drop_caches
    echo Timing the copy of 1000 1MB files from RAM disk to Root FS
    time sh -c 'cp -a /dev/shm/smallFiles/. $PATH/smallRamFiles/; sync '
done

sudo rm -rf /dev/shm/*

exit

```

Listing C.1: Script used to do read-write tests on the root file system kept on SD card as well as NFS server

Appendix D

Terminal Output of Update via NFS Server Side Installation

Last metadata expiration check: 1:09:05 ago on Wed 11 Dec 2019 03:36:40 PM CET. Dependencies resolved.				
Package	Arch	Version	Repository	Size
Upgrading:				
dracut	aarch64	049-10.git20190115.el8_0.1	centos-updates	358 k
dracut-network	aarch64	049-10.git20190115.el8_0.1	centos-updates	96 k
dracut-squash	aarch64	049-10.git20190115.el8_0.1	centos-updates	52 k
glibc	aarch64	2.28-42.el8.1	centos-updates	3.5 M
glibc-common	aarch64	2.28-42.el8.1	centos-updates	750 k
glibc-devel	aarch64	2.28-42.el8.1	centos-updates	1.0 M
glibc-headers	aarch64	2.28-42.el8.1	centos-updates	456 k
Transaction Summary				
Upgrade 9 Packages				
Total download size: 31 M				
Downloading Packages:				
(1/9): dracut-network-049-10.git20190115.el8_0.1.aarch64.rpm				
350 kB/s 96 kB 00:00				
(2/9): dracut-config-rescue-049-10.git20190115.el8_0.1.aarch64.rpm				
154 kB/s 51 kB 00:00				
(3/9): dracut-squash-049-10.git20190115.el8_0.1.aarch64.rpm				
455 kB/s 52 kB 00:00				
(4/9): dracut-049-10.git20190115.el8_0.1.aarch64.rpm				
769 kB/s 358 kB 00:00				
(5/9): glibc-common-2.28-42.el8.1.aarch64.rpm				
4.6 MB/s 750 kB 00:00				
(6/9): glibc-devel-2.28-42.el8.1.aarch64.rpm				
798 kB/s 1.0 MB 00:01				
(7/9): glibc-headers-2.28-42.el8.1.aarch64.rpm				
4.6 MB/s 456 kB 00:00				
(8/9): glibc-2.28-42.el8.1.aarch64.rpm				
910 kB/s 3.5 MB 00:03				
(9/9): glibc-all-langpacks-2.28-42.el8.1.aarch64.rpm				
3.2 MB/s 25 MB 00:07				

Total

3.8 MB/s | 31 MB 00:08

Running transaction check

Transaction check succeeded.

Running transaction **test**

Transaction **test** succeeded.

Running transaction

Preparing :

1/1

Upgrading : glibc-all-langpacks-2.28-42.el8.1.aarch64

1/18

Upgrading : glibc-common-2.28-42.el8.1.aarch64

2/18

Running scriptlet: glibc-2.28-42.el8.1.aarch64

3/18

Upgrading : glibc-2.28-42.el8.1.aarch64

3/18

Running scriptlet: glibc-2.28-42.el8.1.aarch64

3/18

Upgrading : dracut-049-10.git20190115.el8_0.1.aarch64

4/18

Running scriptlet: glibc-headers-2.28-42.el8.1.aarch64

5/18

Upgrading : glibc-headers-2.28-42.el8.1.aarch64

5/18

Upgrading : glibc-devel-2.28-42.el8.1.aarch64

6/18

Running scriptlet: glibc-devel-2.28-42.el8.1.aarch64

6/18

Upgrading : dracut-config-rescue-049-10.git20190115.el8_0.1.aarch64

7/18

Upgrading : dracut-network-049-10.git20190115.el8_0.1.aarch64

8/18

Upgrading : dracut-squash-049-10.git20190115.el8_0.1.aarch64

9/18

Running scriptlet: glibc-devel-2.28-42.el8_0.1.aarch64

10/18

Cleanup : glibc-devel-2.28-42.el8_0.1.aarch64

10/18

Cleanup : glibc-headers-2.28-42.el8_0.1.aarch64

11/18

Cleanup : dracut-squash-049-10.git20190115.el8.aarch64

12/18

Cleanup : dracut-network-049-10.git20190115.el8.aarch64

13/18

Cleanup	: dracut-config-rescue-049-10.git20190115.el8.aarch64
14/18	
Cleanup	: dracut-049-10.git20190115.el8.aarch64
15/18	
Cleanup	: glibc-common-2.28-42.el8_0.1.aarch64
16/18	
Cleanup	: glibc-all-langpacks-2.28-42.el8_0.1.aarch64
17/18	
Running scriptlet:	: glibc-all-langpacks-2.28-42.el8_0.1.aarch64
17/18	
Cleanup	: glibc-2.28-42.el8_0.1.aarch64
18/18	
Running scriptlet:	: glibc-all-langpacks-2.28-42.el8.1.aarch64
18/18	
Verifying	: dracut-049-10.git20190115.el8_0.1.aarch64
1/18	
Verifying	: dracut-049-10.git20190115.el8.aarch64
2/18	
Verifying	: dracut-config-rescue-049-10.git20190115.el8_0.1.aarch64
3/18	
Verifying	: dracut-config-rescue-049-10.git20190115.el8.aarch64
4/18	
Verifying	: dracut-network-049-10.git20190115.el8_0.1.aarch64
5/18	
Verifying	: dracut-network-049-10.git20190115.el8.aarch64
6/18	
Verifying	: dracut-squash-049-10.git20190115.el8_0.1.aarch64
7/18	
Verifying	: dracut-squash-049-10.git20190115.el8.aarch64
8/18	
Verifying	: glibc-2.28-42.el8.1.aarch64
9/18	
Verifying	: glibc-2.28-42.el8_0.1.aarch64
10/18	
Verifying	: glibc-all-langpacks-2.28-42.el8.1.aarch64
11/18	
Verifying	: glibc-all-langpacks-2.28-42.el8_0.1.aarch64
12/18	
Verifying	: glibc-common-2.28-42.el8.1.aarch64
13/18	
Verifying	: glibc-common-2.28-42.el8_0.1.aarch64
14/18	
Verifying	: glibc-devel-2.28-42.el8.1.aarch64
15/18	

```

Verifying      : glibc-devel-2.28-42.el8_0.1.aarch64
    16/18
Verifying      : glibc-headers-2.28-42.el8_1.1.aarch64
    17/18
Verifying      : glibc-headers-2.28-42.el8_0.1.aarch64
    18/18

Upgraded:
dracut-049-10.git20190115.el8_0.1.aarch64
dracut-config-rescue-049-10.git20190115.el8_0.1.aarch64
dracut-network-049-10.git20190115.el8_0.1.aarch64
dracut-squash-049-10.git20190115.el8_0.1.aarch64
glibc-2.28-42.el8_1.1.aarch64
glibc-all-langpacks-2.28-42.el8_1.1.aarch64
glibc-common-2.28-42.el8_1.1.aarch64
glibc-devel-2.28-42.el8_1.1.aarch64
glibc-headers-2.28-42.el8_1.1.aarch64

Complete!
[kmor@pcep64 centos8-rootfs]$

```

Listing D.1: Output on the terminal while updating the CentOS 8 root file system from the NFS server side

CentOS 8 DNF Repository Configuration

```

# CentOS-Base.repo
#
# The mirror system uses the connecting IP address of the client and the
# update status of each mirror to pick mirrors that are updated to and
# geographically close to the client. You should use this for CentOS updates
# unless you are manually picking other mirrors.
#
# If the mirrorlist= does not work for you, as a fall back you can try the
# remarked out baseurl= line instead.
#
#
# CentOS 8 uses local repositories at http://mirror.centos.org
#

[BaseOS]
name=CentOS-$releasever - BaseOS
baseurl=http://mirror.centos.org/centos/$releasever/BaseOS/$basearch/os
gpgcheck=0
enabled=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial

#Extra Packages for Enterprise Linux for CentOS8

[epel]
name=Extra Packages for Enterprise Linux 8 - $basearch
baseurl=http://linuxsoft.cern.ch/epel/8/Everything/$basearch/
enabled=1
gpgcheck=0

#released updates (currently no update repository available on the CentOS 8 mirror)
[updates]
name=CentOS-$releasever - Updates
baseurl=http://mirror.centos.org/centos/$releasever/BaseOS/$basearch/os

```

```

gpgcheck=0
enabled=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial

#additional packages that may be useful
[extras]
name=CentOS-$releasever - Extras
baseurl=http://mirror.centos.org/centos/$releasever/extras/$basearch/os
gpgcheck=0
enabled=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial

#additional packages that extend functionality of existing packages

[centosplus]
name=CentOS-$releasever - Plus
baseurl=http://mirror.centos.org/centos/$releasever/centosplus/$basearch/os
gpgcheck=0
enabled=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial

#AppStream packages

[AppStream]
name=CentOS-$releasever - AppStream
baseurl=http://mirror.centos.org/centos/$releasever/AppStream/$basearch/os
gpgcheck=0
enabled=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial

#PowerTools packages

[PowerTools]
name=CentOS-$releasever - PowerTools
baseurl=http://mirror.centos.org/centos/$releasever/PowerTools/$basearch/os
gpgcheck=0
enabled=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial

#CR packages

[cr]
name=CentOS-$releasever - cr
baseurl=http://mirror.centos.org/centos/$releasever/cr/$basearch/os
gpgcheck=0
enabled=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial

[fasttrack]
name=CentOS-$releasever - fasttrack
baseurl=http://mirror.centos.org/centos/$releasever/fasttrack/$basearch/os
gpgcheck=0
enabled=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial

```

Listing D.2: CentOS base repository configuration for DNF package manager in CentOS 8 root file system

Terminal Output for Client Side Software Update

```

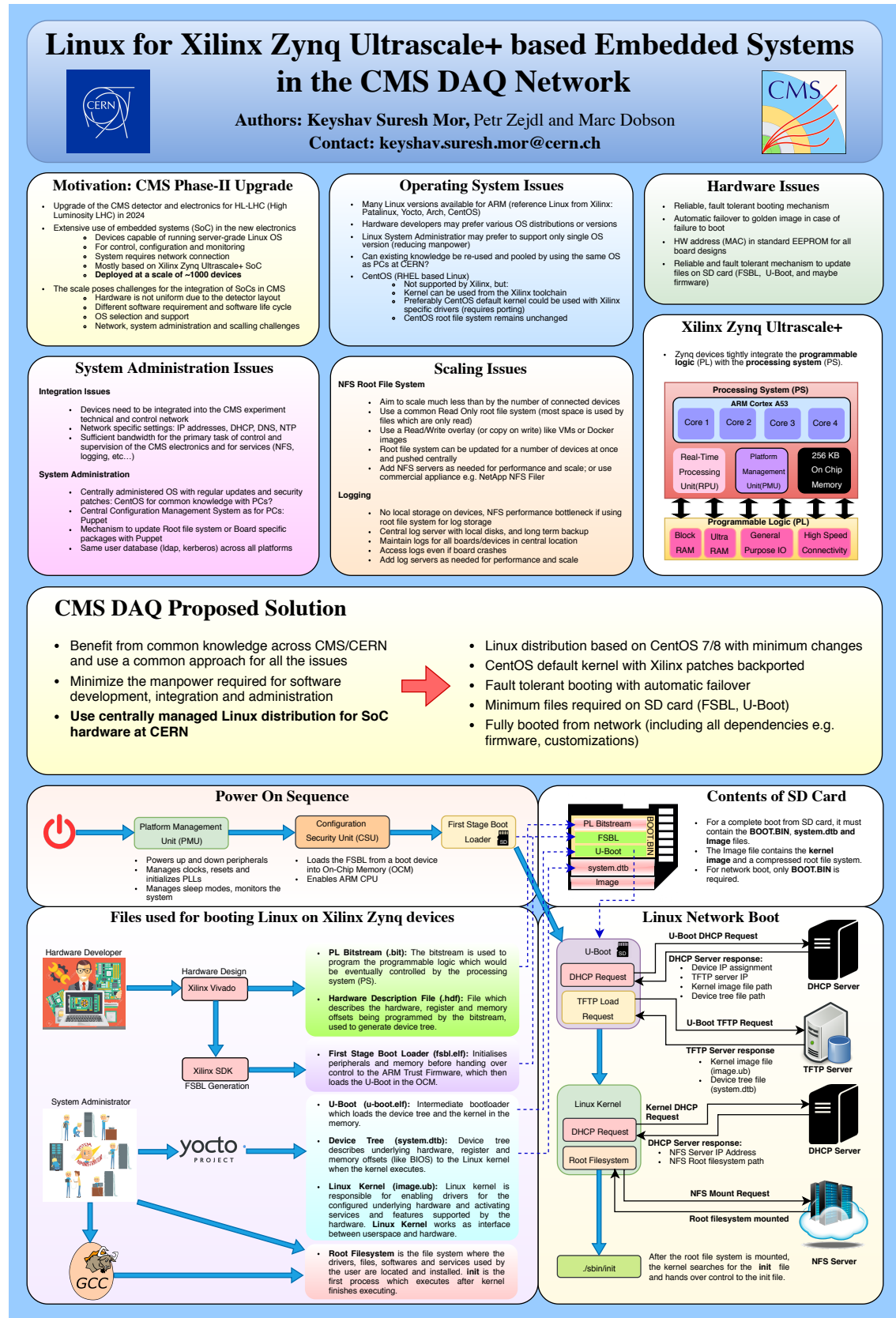
[root@128 ~]# dnf upgrade --best --allowdowngrading
Last metadata expiration check: 1 day, 22:33:14 ago on Sat Jan 11 21:17:38 2020.
Dependencies resolved.

```

Package	Architecture	Version	Repository	Size
Upgrading:				
openjpeg2	aarch64	2.3.1-1.el8	cr	145 k
openjpeg2-tools	aarch64	2.3.1-1.el8	cr	79 k
turbojpeg	aarch64	1.5.3-10.el8	cr	135 k
Removing dependent packages:				
openjpeg2-devel	aarch64	2.3.0-8.el8	@PowerTools	68 k
turbojpeg-devel	aarch64	1.5.3-7.el8	@PowerTools	64 k
Transaction Summary				
Upgrade 3 Packages				
Remove 2 Packages				
Total download size: 1.6 M				
Is this ok [y/N]: y				
Downloading Packages:				
(1/3): openjpeg2-tools-2.3.1-1.el8.aarch64.rpm		346 kB/s	79 kB	00:00
(2/3): openjpeg2-2.3.1-1.el8.aarch64.rpm		603 kB/s	145 kB	00:00
(3/3): turbojpeg-1.5.3-10.el8.aarch64.rpm		2.6 MB/s	135 kB	00:00
Total		4.8 MB/s	0.4 MB	00:00
Running transaction check				
Transaction check succeeded.				
Running transaction test				
Transaction test succeeded.				
Running transaction				
Preparing	:			1/1
Upgrading	:	openjpeg2-2.3.1-1.el8.aarch64		1/7
Upgrading	:	openjpeg2-tools-2.3.1-1.el8.aarch64		2/7
Upgrading	:	turbojpeg-1.5.3-10.el8.aarch64		3/7
Erasing	:	turbojpeg-devel-1.5.3-7.el8.aarch64		4/7
Cleanup	:	openjpeg2-tools-2.3.0-8.el8.aarch64		5/7
Cleanup	:	openjpeg2-2.3.0-8.el8.aarch64		6/7
Cleanup	:	turbojpeg-1.5.3-7.el8.aarch64		7/7
Verifying	:	openjpeg2-2.3.1-1.el8.aarch64		1/8
Verifying	:	openjpeg2-2.3.0-8.el8.aarch64		2/8
Verifying	:	openjpeg2-tools-2.3.1-1.el8.aarch64		3/8
Verifying	:	openjpeg2-tools-2.3.0-8.el8.aarch64		4/8
Verifying	:	turbojpeg-1.5.3-10.el8.aarch64		5/8
Verifying	:	turbojpeg-1.5.3-7.el8.aarch64		6/8
Verifying	:	openjpeg2-devel-2.3.0-8.el8.aarch64		7/8
Verifying	:	turbojpeg-devel-1.5.3-7.el8.aarch64		8/8
Upgraded:				
openjpeg2-2.3.1-1.el8.aarch64		openjpeg2-tools-2.3.1-1.el8.aarch64		
turbojpeg-1.5.3-10.el8.aarch64				
Removed:				
openjpeg2-devel-2.3.0-8.el8.aarch64		turbojpeg-devel-1.5.3-7.el8.aarch64		

Listing D.3: Terminal output for client side software update

Poster at TWEPP 2019



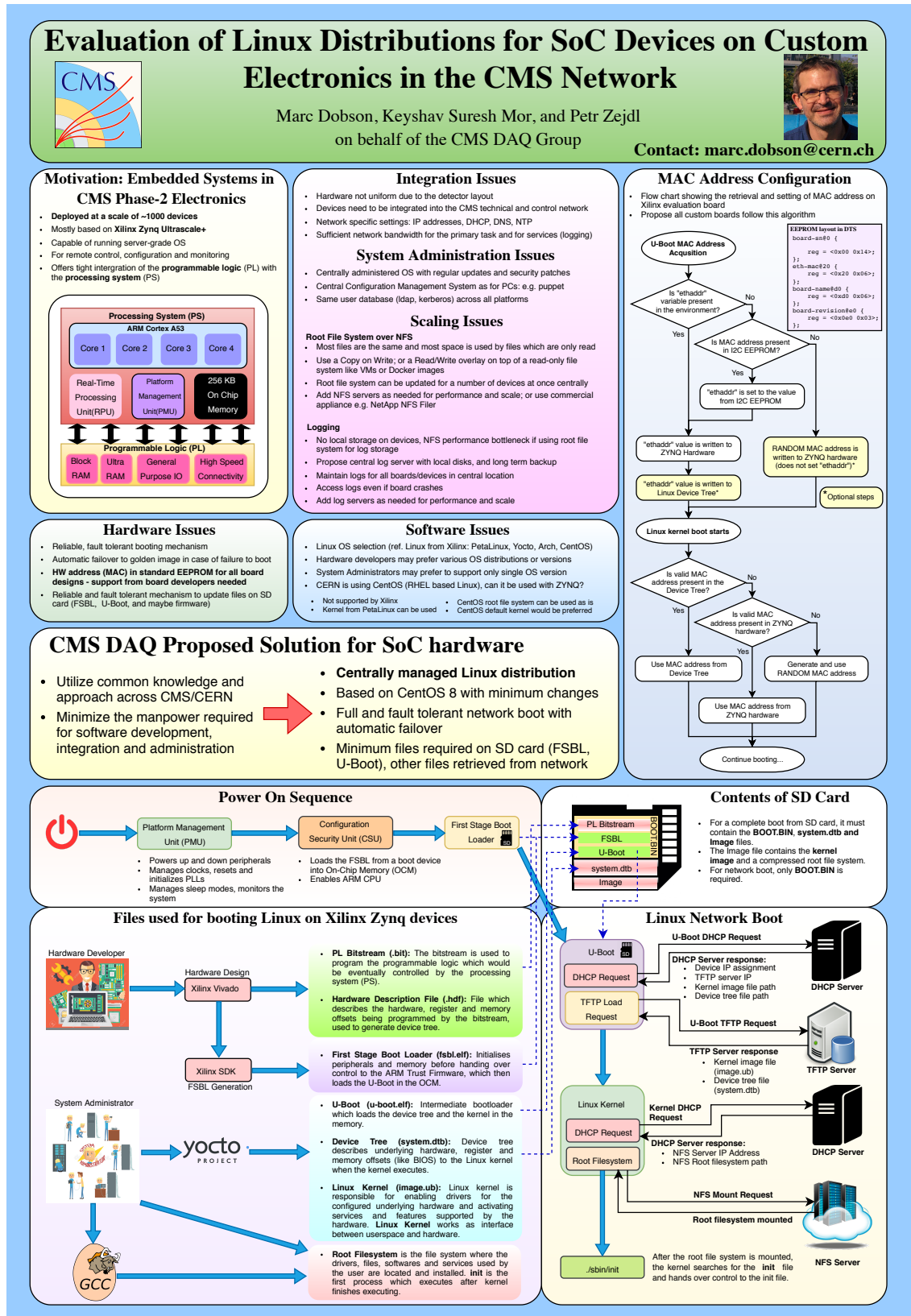


Figure D.2: Poster presented at CHEP 2019