2IMC05 Preparation Graduation Project Embedded Systems

Technische Universiteit Eindhoven

in association with

CERN Compact Muon Solenoid Experiment

By

Keyshav Suresh Mor (Matriculation No. 1237978)

MSc. Embedded Systems - Embedded Networking

Department of Mathematics and Computer Science

under the supervision of

Dr. Majid Nabi Najafabadi

Department of Electrical Engineering

Dr. Frans Meijers

CERN CMS DAQ

Integration of Embedded Systems in the Network, System Administration and Application Frameworks of the Data Acquisition of the Compact Muon Solenoid Experiment

June 20, 2019

Table of Content

1. Introduction **6**

2. The Large Hadron Collider(LHC) and **8**

The Compact Muon Solenoid(CMS) Experiment **8**

2.1. The Large Hadron Collider 8

2.1.1 Energy of Collision 9

2.1.2 The LHC Calendar 10

2.2 The High Luminosity LHC 10

2.3 Compact Muon Solenoid Experiment 11

2.3.1 Tracker 12

2.3.2 Electromagnetic Calorimeter 12

2.3.3 Hadronic Calorimeter 12

2.3.4 Muon Detectors 12

2.3.5 Trigger and Data Acquisition 13

3. Phase-2 Upgrade of the CMS Data Acquisition System **15**

3.1 CMS Detector Upgrades for HL-LHC 15

3.2 The CMS Level-1 Trigger 16

3.3 The CMS Data Acquisition System (DAQ) for Run 2 17

3.4 The CMS Data Acquisition System Phase-2 Upgrade 20

3.5 Timing and Control Distribution System (TCDS) 23

3.6 CMS DAQ Event Builder 24

3.7 High-Level Trigger (HLT) 25

4. Fundamentals of Embedded Computing Systems **26**

and GNU/Linux **26**

4.1 Overview of Embedded Systems 26

4.2 Overview of Operating Systems 27

4.3 The GNU/Linux 28

4.3.1 The GNU/Linux Bootloader 29

4.3.2 The GNU/Linux Kernel 31

4.3.3 The GNU/Linux Init Process 32

4.3.4 The GNU/Linux Root Filesystem 33

4.3.5 Importance of Linux based Operating Systems 35

4.3.6 Free versus Freedom 37

4.4 The Need for “Embedded” Linux 38

5. Linux for Embedded ARM Processors **39**

5.1 Linux components for ARM Processors 39
5.2 Bootloader 39
5.2.1 U-Boot 39
5.2.2 Challenges of booting 40
5.2.3 The Booting Sequence 41
5.2.3.1 Phase 1 - ROM Code 41
5.2.3.2 Phase 2 - Secondary Program Loader 42
5.2.3.3 Phase 3 - Tertiary Program Loader 43
5.2.3.4 Handing over control from Bootloader to Kernel 44
5.3 Embedded Linux Kernel 44
5.4 The Device Tree 45
5.5 Root Filesystem 46
5.5.1 INITRAMFS 47
5.5.2 NFS 47
5.5.2.1 Versions of NFS 47
5.5.2.2 NFS advantages 48
5.5.2.3 NFS disadvantages 48
5.6 How NFS works 49

6. Linux Distributions for the Xilinx Zynq Series 50

System-on-Chip FPGAs 50
6.1 What are Linux Distributions ? 50
6.2 Linux Distribution Examples 50
6.3 Linux Distributions for the CMS TDAQ Embedded Computers 51
6.4 Xilinx Linux/ PetaLinux/ Yoctolinux 53
6.5 CentOS 54
6.6 Archlinux 55

7. The Xilinx Zynq 7000 and Zynq Ultrascale+ 57

System-on-Chip FPGA platforms 57
7.1 The Avnet Zynq Zedboard 57
7.2 The Xilinx Zynq 7000 All Programmable System-on-Chip FPGA 60
7.3 The Zynq 7000 Boot Process 63
7.3.1 First Stage Bootloader (Secondary Program Loader) 64
7.3.2 U-Boot 66
7.4 The Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit 67
7.5 The Xilinx Zynq UltraScale+ MPSoC 69
7.6 The Xilinx Zynq UltraScale+ MPSoC Boot Process 71
7.6.1 Boot Process Overview 71

7.6.2 Boot Flow 72
7.6.2.1 Non-Secure Boot Flow 72
7.6.2.2 Secure Boot Flow 72
7.6.2.3 SD Boot Mode 73
7.6.2.4 Zynq Ultrascale+ Post-Configuration Boot Process 74

8. Linux Distribution building and Deployment 76

on the Xilinx Zynq Series SoCs 76
8.1 The Yocto Project 76
8.1.1 Yocto Features 76
8.1.2 Yocto Challenges 77
8.1.3 The Yocto Layers 78
8.1.4 Open-Embedded Build System and its components 78
8.1.5 The OpenEmbedded Build System Workflow 80
8.2 PetaLinux Tools 81
8.3 Distribution Development with PetaLinux Tools 83

9. Research Goals of the Project 100

10. Workplan 103

Appendix A 105
What is TFTP and how does it work? 103
What is DHCP and how does it work? 103

References 106

List of Figures

Fig.2.1 : CERN's accelerator complex schematic with all the different accelerators 9
Fig.2.2 : Long term LHC calendar with the integrated and peak luminosity previsions 10
Fig.2.3 : Sectional view of the CMS detector 11
Fig.2.4 : Overview of the CMS detector, as a transverse slice through the barrel 14
Fig.3.1 : HL-LHC projected running parameters 15
Fig.3.2 : Schematic design of the CMS Trigger System 17
Fig.3.2 : Schematic design of the CMS Trigger System 17
Fig.3.3 : The CMS DAQ for LHC Run-2 18
Fig.3.4 : Principle schematics of the baseline Phase-2 CMS DAQ[9] 18
Fig.3.5 : The DTH and its leaf boards arranged in an ATCA crate 21
Fig.3.6 : Projected readouts from sub-detectors for CMS DAQ Phase-2 Upgrade[9] 21
Fig.3.7 : Projected minimum number of DTH and DAQ D2S optical links (100 Gb/s) for each 22

sub-detector.[9] 22
Fig.3.8 : Projected throughput of the new DAQ system 23
Fig.3.9 : The event building process as explained in section 3.6 25
Fig.4.1 Architecture of an Unix-like Operating System 28
Fig.4.2 : The Linux Kernel (Source : Zynq Software Developers Guide [32]) 32
Fig.4.3 : The init process run-levels and their corresponding functions[18]. 33
Fig.4.4 : Computer operating in run-level 5 33
Fig.4.5 : A portion of the root file tree structure at a computer in the CMS department 34
Fig.5.1 : Booting in older computers 41
Fig.5.2 : Phase 1 of Booting Process. The ROM Code searches for the SPL from memory and loads it into the on-chip SRAM. 42
Fig.5.3 : Phase 2 of Booting Process. 43
Fig.5.4 : An example of the Zynq-7000 device tree describing the kernel boot arguments, the MAC address and partitioning of the QSPI. 46
Fig. 5.5 : NFS file system mount(Source : www.qnap.com) 50
Fig.6.1 : The concept of a Linux Distribution 51
Fig.6.2 : The CentOS logo 54
Fig.6.3 : The Archlinux logo 55
Fig.7.1 The Avnet Zynq Zedboard(Source : www.zedboard.org) 59
Fig.7.2 The Avnet Zynq Zedboard architecture[35] 59
Fig.7.3 The Zynq Z7020 CLG484 Bank Assignments [35] 60
Fig.7.4 : PL features of Zynq-7000 AP SoC[36] 61
Fig.7.5 : PS features of Zynq-7000 AP SoC[36] 62
Fig.7.6 : Architecture of the Zynq-7000 AP SoC(Source : www.core-vision.nl) 63
Fig.7.7 : Phase 1 BootROM process of Zynq-7000 [37] 64
Fig.7.8 : Phase 2 FSBL sequence of Zynq-7000[37] 65
Fig.7.9 : Contents of the bootable SD card. 66
Fig.7.10 : Zynq-7000 boot process. 66
Fig.7.11 : Xilinx ZCU102 Evaluation Kit(Source : www.xilinx.com) 68
Fig.7.12 : Some PL features of Zynq Ultrascale+ MPSoC[39] 70
Fig.7.13 : Zynq Ultrascale+ MPSoC architecture(Source : Xilinx) 71
Fig.7.14 : Zynq Ultrascale+ boot in SD card mode [40] 73
Fig.7.15 : Pre-boot tasks performed by PMU [40] 74
Fig.7.16 : Generic Zynq Ultrascale+ boot sequence[40] 74
Fig.7.17 : Zynq Ultrascale+ secure boot sequence[40] 75
Fig.8.1 : Various layers of the Yocto Project[41] 79
Fig.8.2 : Open-Embedded Architecture[41] 81
Fig.8.3 : PetaLinux Yocto extensible SDK[27] 82
Fig.8.4 : PetaLinux Yocto Layers[27] 82

Fig.8.5 : Xilinx Vivado Design Suite Logo 84
Fig.8.6 : Setting up PetaLinux environment 86
Fig.8.7 : PetaLinux project creation 87
Fig.8.8 : Enabling NFS and specifying the tftpboot directory 88
Fig.8.9 : Enabling TCP/IP, DHCP, BOOTP and RARP for NFS mount in kernel config 89
Fig.8.10 : Enabling NFS services in the kernel config 90
Fig.8.11 : Contents of the TFTP boot directory 91
Fig.8.12 and Fig.8.13 : Setting up the TFTP server 92
Fig.8.14 : Setting up NFS file share 93
Fig.8.15 : Setting up Zedboard jumpers for SD card boot 94
Fig.8.16 : Entering the U-Boot after board bringup 95
Fig.8.17 : Requesting DHCP for Zedboard network configuration 96
Fig.8.18 : Setting up TFTP server IP address in U-Boot environment 96
Fig.8.19 : Kernel executing and requesting DHCP server for NFS root IP and root directory 97
Fig.8.20 : Logging in to the Linux 98
Fig.8.21 : Inspecting the mounted root filesystem 99
Fig.8.22 : Directory “linuxdemo” created via Zedboard terminal appears on the NFS server root directory 99

1. Introduction

Gaining deeper understanding of the nature and the universe is the main goal of the study of physics. Complex experiments in the form of particle accelerators and detectors have achieved great success in testing the Standard Model of Physics and to study the fundamental properties and composition of matter, which forms the basis of studies in the field of particle physics. In 2012, the existence of Higgs Boson was proved through the experimental data, 48 years after it was theoretically postulated in 1964. This discovery was possible due to the largest and most powerful particle accelerator in the world, the Large Hadron Collider (LHC), which is situated at CERN (European Organisation for Nuclear Research) on the French-Swiss Border near Geneva. The LHC consists of a 27 km long accelerator ring, in which proton beams with a center-of-mass energy of 14 TeV, travel in opposite orbital directions, close to the speed of light. At dedicated points, the two beams cross each other, resulting in collisions of the protons. In order to understand these collisions and answer questions related to the fundamentals of physics, in particular the sub-atomic particles and their composition, various particle detectors exist along the LHC rings, in order to detect these collisions, track the particles emerging from these collisions and collect data of specific nature to assist in various different experiments. Compact Muon Solenoid (CMS) is one of the largest among such detectors at CERN, consisting of different sub-detectors to achieve high sensitivity to detect various particles to investigate and study a broad field of theories. The LHC and the CMS experiment are explained in more detail in Chapter 2.

The next stage of experiments and data-collection at the LHC would be focusing on gaining insight into subjects like the matter-antimatter asymmetry, dark matter or the non zero mass of neutrinos, since the Standard Model of Physics cannot convincingly explain these phenomena. More data needs to be recorded for collision at higher energies to increase the data sampling and improve the precision of the measurements. Keeping this in mind, the LHC will be upgraded to High Luminosity LHC between 2024 and 2026, whereby the instantaneous luminosity will increase to $7.5 \times 10^{34}\ cm^{-2}\ s^{-1}$, which is five times above the LHC's design value. This will result in up to 200 interactions per crossing, whose decay products would be detected by various sub-detectors. The upgraded CMS detector would be then read out at an unprecedented 50 Tb/s and an event sampling rate of 750 kHz. Interesting events would then be analysed using software algorithms in the processing nodes present at High Level Trigger and selected events would be stored permanently at 7.5 kHz for offline processing and analysis. Currently, the CMS detector is readout at 2 Tb/s data rate and 100 KHz event data sampling rate and data storage rate of 1 kHz. The change in the quality of the physics experiment under observation and the data generated because of it, stipulates the need to upgrade from current electronic systems and other legacy electronic equipment present at the detector, trigger, data-acquisition and processing levels to mature, modern technologies which can support the requirement of achieving such high data throughputs and sampling rates in the trigger and data acquisition pertaining to the upgraded CMS detector. In addition to this, the processing farms and the storage capacity of the data of selected events would also need to be upgraded. The Phase-2 upgrade of the CMS and its Trigger and Data Acquisition System(TDAQ) is explained in more detail in Chapter 3.

The CMS detector consists of various sub-detectors installed to detect different sub-atomic particles resulting from the beam collision. These sub-detectors are instrumental in generating data about specific subatomic particles which form the basis of various experiments and analysis associated with these

sub-detectors. Naturally, these experiments have their own detector read-out electronics and their own custom electronics in the trigger and data-acquisition system. These electronic systems are distinct for each specific experiment and analysis carried out under the umbrella of the CMS detector, which results in challenges for integrating these electronics in the network, network management and also poses challenges for central system administration, which is important to manage embedded systems running Linux-based operating systems in a complex network consisting of thousands of embedded computers, .

To ensure a certain uniformity and standardisation in the read-out electronics and data acquisition embedded systems of various experiments under the CMS, the System Administration team at CMS has envisioned deployment of a uniform Linux-based distribution in the embedded systems in various Xilinx Zynq 7000 and Zynq Ultrascale+ System-on-Chip installed in the CMS TDAQ. Linux is the primary choice as the embedded operating system given the familiarity of use for System Administrators at CMS, the existing community support for Linux, cost-free installation of open-source distribution and existing prevalent use of such distributions in CMS processing and data server. The fundamentals of Linux-based Operating Systems and Embedded Linux are explained in Chapter 4 and Chapter 5 respectively.

Chapter 6 presents a study of various Linux distributions which would be tested and analysed during the course of the project, keeping in mind the requirements for booting, network file sharing, network integration, IP configuration and system administration etc. Chapter 7 focuses on Xilinx Zynq 7000 SoC and Zynq Ultrascale+ SoC i.e the hardware which would be ultimately deployed with a uniform Linux based OS distribution by most experiment groups at CMS for their readout and TDAQ electronics in the CMS network. Chapter 8 gives an overview on the Linux-distribution building for ARM processors using tools like PetaLinux, Yocto Project and other tools necessary for building these custom Linux distributions. Chapter 9 discusses the research goals for the project in terms of addressing challenges in Linux distribution development, testing the embedded systems for integration in the network, the network management, system administration, root file system management and deployment of the application software stack of these SoC FPGAs. Chapter 10 discusses the current status of the project and to finalise the preparation document, the work plan schedule for the thesis is outlined.

2. The Large Hadron Collider(LHC) and The Compact Muon Solenoid(CMS) Experiment

Collider experiments are developed for accelerating particles nearly to the speed of light and colliding them with each other to analyse the decays and products of these collisions and study the elementary structure of matter to answer fundamental questions about the universe we live in. Large Hadron Collider (LHC) is a similar accelerator and particle collider, the largest and the most powerful in the world. It was commissioned in September 2008, and since then many theories of the standard model of physics have already been investigated and confirmed. In the following sections, the LHC and the CMS experiments are explained in more detail. In order to understand some key concepts, the some terminologies are explained as follows : The center of mass "$\sqrt{s}$" of the colliding particles is related to the total energy provided for the collision. The instantaneous luminosity $\mathcal{L}$ is related to the rate of potential particle collisions per surface unit . The integrated luminosity $\mathcal{L}$,is the integral of $\mathcal{L}$ over time, measures the number of collected data events and is expressed in as a certain units of inverse femtobarns(fb^{-1})[1][2][3].

2.1. The Large Hadron Collider

A cascade of pre-accelerators is used to pass the particles through it, so that they are accelerated to a minimum speed. The protons are extracted from hydrogen gas using an electric field and introduced into the LINAC 2 (LINear ACcelerator), where they get accelerated to an energy of about s = 50 MeV [2][3][4].. Then they pass through two circular accelerators,the PS (Proton Synchrotron) and the PS booster. They are then accelerated to an energy of 25 GeV, which is around 0.9993 times the speed of light. The final stage prior to the LHC is the SPS (Super Proton Synchrotron) where the protons reach an energy of 450 GeV. During these acceleration stages, the protons are separated into groups which contain about 10^{11} protons each. 2808 of these groupings are fed into each of the two LHC beam pipes, where they move in opposite directions in the circular accelerator ring. The LHC ring has a circumference of 27 km and is divided into eight sections, each consisting of curved and a straight subsection. The former part bends the particle beam on its circular trajectory through Lorentz force which is realized by 1232 dipole magnets realised by Niobium-titanium coils [2][3][4]. These dipoles are operated at a temperature of 1.9 K to achieve a superconducting state, making it the coldest place in the known Universe. A current of 11 kA is passed through these coils generating a magnetic field of 8.3 T surrounding the LHC ring [2][3][4].

The so-called cavities are installed in the straight section of the ring, generating an alternating electric field to accelerate the particles up to an energy of 7 TeV. Additionally, 392 quadrupole magnets concentrate the bunches on a cross-sectional size of up to $16 \times 16\ \mu m^2$ just before the collision point, where the two beams cross and the particles collide. On an average, only 25 protons collide inelastically out of the 10^{11} protons present per group, with instantaneous luminosity of $1{\cdot}10\ 34\ cm^{-2}\ s^{-1}$. The collision takes place at the crossing rate of 40 MHz[2]. Particle detectors are installed around the crossing points of the LHC beams, to record collisions, track the daughter particles post collision and to reconstruct the underlying physical processes. The largest of these detectors are ALICE, ATLAS, CMS, and LHCb.

Since the focus of this document is on the integration of embedded systems in the system administration and networking of the CMS Trigger and Data Acquisition system, the CMS experiment is explained in more detail in a later section.

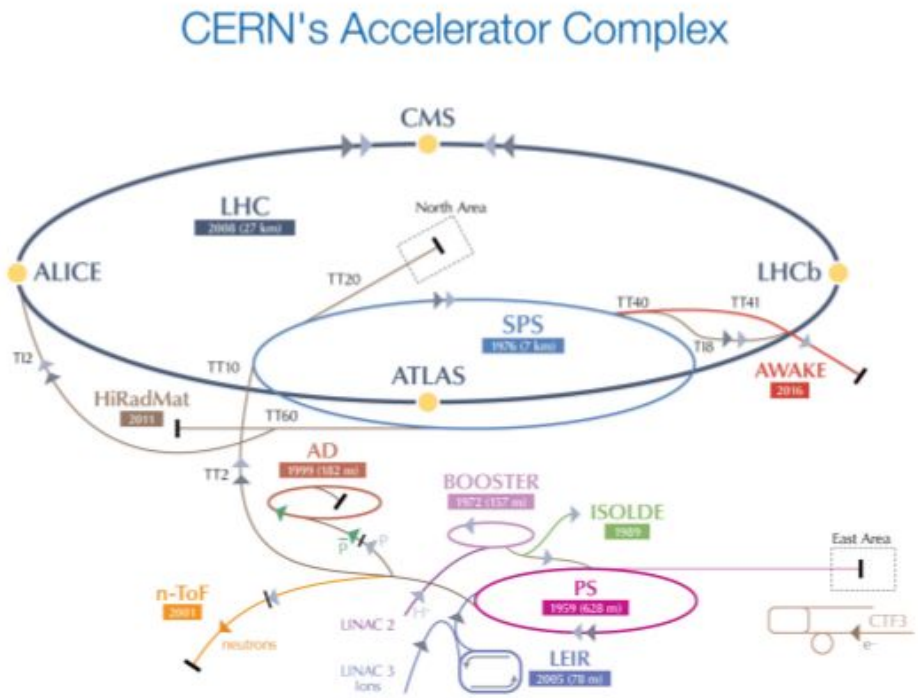

Fig.2.1 : CERN's accelerator complex schematic with all the different accelerators[1]

2.1.1 Energy of Collision

At a collider, the energy in the center of mass system matters. Laboratory system is also the center of mass system and all this energy is useful . This energy is expressed in eV which defines a single collision. The LHC is designed to work with a nominal energy of 14 TeV and today it is running at an energy of 13 TeV in case of proton-proton collisions. The energy of collisions and luminosity are increased during the life of the LHC[1][2].

2.1.2 The LHC Calendar

Between 2010 and 2035, the LHC either is in the "Run" Phase, which is the time to run the accelerator and collect data, or it is in the period of long shutdowns (LS) when maintenance and upgrade of systems happen[3][4]. Increase in luminosities over different phases is presented in the figure below.

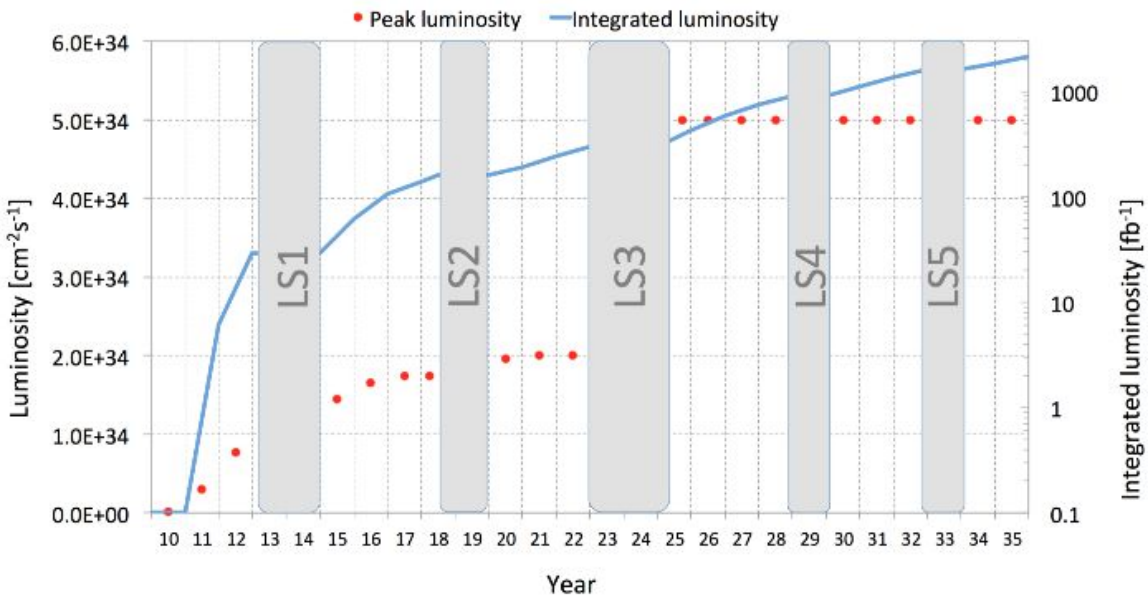

Fig.2.2 : Long term LHC calendar with the integrated and peak luminosity previsions [4]

Phase 1

Phase 1 started in 2010 and will continue up to 2023. During this phase the luminosity and collision energy would be increased progressively. It consists of 3 periods, with a shutdown between each of these operational periods.

Phase 2

Phase 2 would start in 2022 and continue up to 2035. It will start after a long shutdown (LS) and would be divided in periods similar to Phase 1. During this phase, the LHC will be upgraded to High Luminosity (HL-LHC), which is described in the following section.

2.2 The High Luminosity LHC

Higher collision energies and frequencies are necessary to increase the probability of discovering new properties related to the fundamentals of physics and matter as well as confirm various other theories from the Standard Model of Physics. This is the core philosophy behind LHC upgrade, where equipment related to the experiment as well the data-acquisition electronics, both are upgraded. In the first upgrade of 2015, the instantaneous luminosity was increased to $2 \cdot 10\ 34\ cm^{-2}\ s^{-1}$ and bunch distance was reduced to 25 ns. The centre of mass energy was increased from 8 TeV to 13 TeV. During Run 2, the LHC had an integrated luminosity of about 140 fb^{-1}[2][3][4]. The next upgrade phase will take place during the Long

Shutdown 2 from 2019 to 2020. Post this upgrade, the beam energy will rise to 14 TeV[2] and proton bunches will contain more particles to further increase the instantaneous luminosity. This is a step towards achieving integrated luminosity of over 300 fb^{-1} until the next Long Shutdown planned in 2024. This will result in High Luminosity LHC(HL-LHC), in which an instantaneous luminosity of 5-7.5 $\cdot 10^{34}$ $cm^{-2}\,s^{-1}$ [2] is planned. For this kind of LHC performance, the read-out electronics , the trigger and data acquisition would also have to be upgraded along with the changes in the superconducting magnet and other experiment equipment in the LHC ring.

2.3 Compact Muon Solenoid Experiment

The CMS(Compact Muon Solenoid) experiment[5][6][7][8] is designed to enable searches for a wide variety of new physics and perform high precision measurements and detection of electrons, protons and more specifically muons. Various sub-detectors are installed in the CMS detector which focus on detection of different subatomic particles (ex. Muons) and data collection of different aspects of physical measurements like energy through various calorimeters like HCAL(Hadron Calorimeter) to gauge the energy of hadrons and ECAL(Electromagnetic Calorimeter) to gauge the energy of photons generated in the proton-proton collisions.

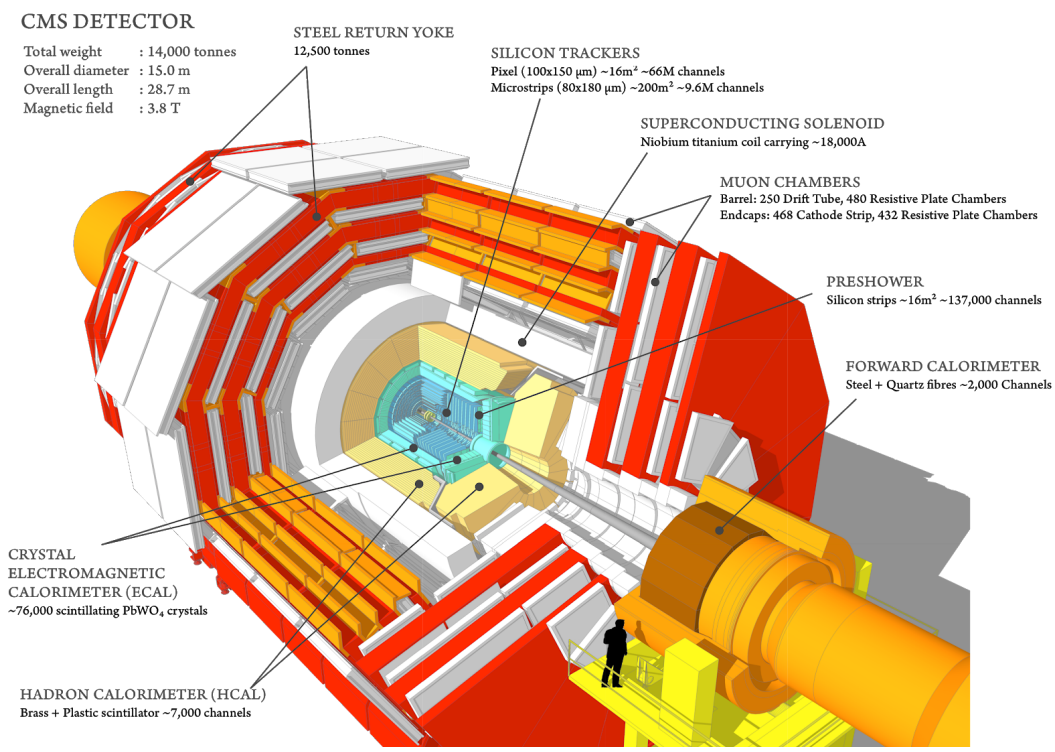

Fig.2.3 : Sectional view of the CMS detector. The LHC beams travel in opposite directions along the central axis of the CMS cylinder colliding in the middle of the CMS detector. (Source : http://cms.web.cern.ch/news/cms-detector-design)

The CMS contains a superconducting solenoid with an internal diameter of 6 m, which provides an axial and uniform 3.8 T magnetic field. A silicon pixel Inner Tracker (IT) and a silicon microstrip Outer

Tracker (OT) are situated within the volume of the solenoid[6]. The electromagnetic and hadronic calorimeters surround the trackers. The forward calorimeters ensure the angular coverage. Within the magnet's return yoke, gaseous ionisation detectors are embedded to detect muons.

2.3.1 Tracker

The CMS tracker measures the three-dimensional trajectories of charged particles. The transverse momentum (p^T) of the particles is calculated from the radius of curvature in the magnetic field. The CMS tracker is the largest all-silicon tracking detector,with an active surface area of around 200 m^2, is the largest of its kind in the world. The silicon IT extends up to a radius of 4.3 cm, where it is most likely to catch incident particles. The silicon OT is extends up to a radius of 110 cm[6]. The trackers use a split barrel-end-cap design with 13 barrel layers and 14 end-cap disks on either side.

2.3.2 Electromagnetic Calorimeter

The CMS Electromagnetic Calorimeter (ECAL) was designed to measure the energy and position of electrons and photons which are produced during the primary interaction of the proton beams colliding at the crossing points in the CMS detector. Fine grained, lead tungstate ($PbWO_4$) tapered crystals which are radiation tolerant induce electromagnetic cascades for traversing photons and electrons, resulting in scintillation light undergoing total internal reflection towards the back of the crystal. Within 25 ns, 80% [6] of the light is collected. Photodiodes amplify the relatively small signal (4.5 photoelectrons per MeV). In the forward region, the Electromagnetic Preshower (ES), a sampling calorimeter comprised of lead and silicon layer is located which generates signals that look like high energy photons from the primary interaction. The ES(2 mm) is much more finely grained than the ECAL (3 cm crystals)[6], and can resolve photons individually.

2.3.3 Hadronic Calorimeter

The ECAL is encapsulated by Hadronic Calorimeter(HCAL), which helps in the reconstruction of the energy and position of QCD jets particularly that of hadrons, commonly known as nucleons. The HCAL Barrel and Endcap Calorimeter (HE) sampling detectors consist of layers of brass plates and plastic scintillation tiles. It collects the scintillation signal and sends it to the on-detector amplifiers for amplification. Because of higher particle flux in the forward regions, the HCAL Forward Calorimeter (HF) is constructed using radiation-hardened technology. Steel attenuators containing quartz fibres are used to measure the Cherenkov radiation of a particle[6].

2.3.4 Muon Detectors

CMS as its name suggests is designed to detect and identify muons with high efficiency, given that they are very difficult to detect as they are the product of a very late decay of the daughter particles. Gas chambers are used to detect the muons and three different gas chamber technologies are used namely : Drift Tubes (DT) in the barrel region; Cathode Strip Chambers (CSC) in the endcap regions; and Resistive Plate Chambers (RPC) . DTs are able to be used in the barrel region as the neutron-induced background is small, the muon rate is relatively low, and the magnetic field is uniform. In the end-cap,

however, where the muon and background rates are much higher, and the magnetic field is non-uniform, CSC are more suitable. To provide additional coverage in the scenario that background rates significantly rise as the LHC luminosity increases, an independent, highly segmented dedicated triggering system with a fast response time, consisting of RPC is also in use. In comparison with the other muon detectors, they deliver much better time resolution, but poorer position resolution[6].

2.3.5 Trigger and Data Acquisition

The trigger system helps in selecting only certain interesting events for storage and analysis due to bandwidth and offline storage constraints and also because majority of the events occuring during the experiment are not of interest for every detector. CMS uses a two-level trigger system. The Level-1 (L1) trigger, is implemented using custom designed electronics for selecting interesting events, sampling data at 100 KHz while the collisions occur at 40 MHz. The tolerable latency is around 4 μs, including the delay involved in data-readout to the L1 trigger and the response signal to accept data of a particular event [17, 18, 19, 20, 21]. The High Level Trigger (HLT) reduces the event sampling rate from 100 KHz to about 1 KHz, which is essentially a data processing farm running reconstruction and analysis software. The HLT receives complete detector information through underlying Data Acquisition modules such as the Readout Units (RU) which collect data from individual Front-End Detectors (FED), the Event Builders which compiles the data of all interesting events received by the RUs, the Builder Units(BU) which compiles individual copies from Event Builders to feed them to Filter Units(FU), which finally feeds it to the HLT. Due to latency and bandwidth considerations for online processing of data, simplified reconstruction algorithms are used instead of complex ones such as the ones used in offline reconstruction and analysis. A series of reconstruction and filter stages allows reduction of the number of events that must undergo more complex processing stages such as track reconstruction at the HLT because of the limitations in computing power, bandwidth and storage requirements. The Worldwide LHC Computing Grid (WLCG) is a globally distributed data storage and processing infrastructure which stores and performs offline reconstruction on events that pass the HLT selection. The L1 trigger has two main subsystems, the L1 calorimeter trigger and the L1 muon trigger. These subsystems receives data from different sub detectors and the Micro Global Trigger (MicroGT) combines the outputs and makes a final triggering decision. The outputs of the trigger corresponds to detection of objects such as electrons, photons, tau lepton decays containing hadrons, jets, energy sums, and muons. The MicroGT uses a set of around 300 algorithms to make a decision based on the momentum, position, isolation and quality of these objects, which are regularly updated as per changes in the LHC running conditions[6]. The CMS Trigger and Data Acquisition System was recently upgraded has to maintain its efficiency for the selection of interesting events during LHC Run 2. It was installed and commissioned in 2015, and was successfully operated during the years 2016 and 2017. The CMS TDAQ in its current version and its upgrade are explained in detail in Chapter 3.

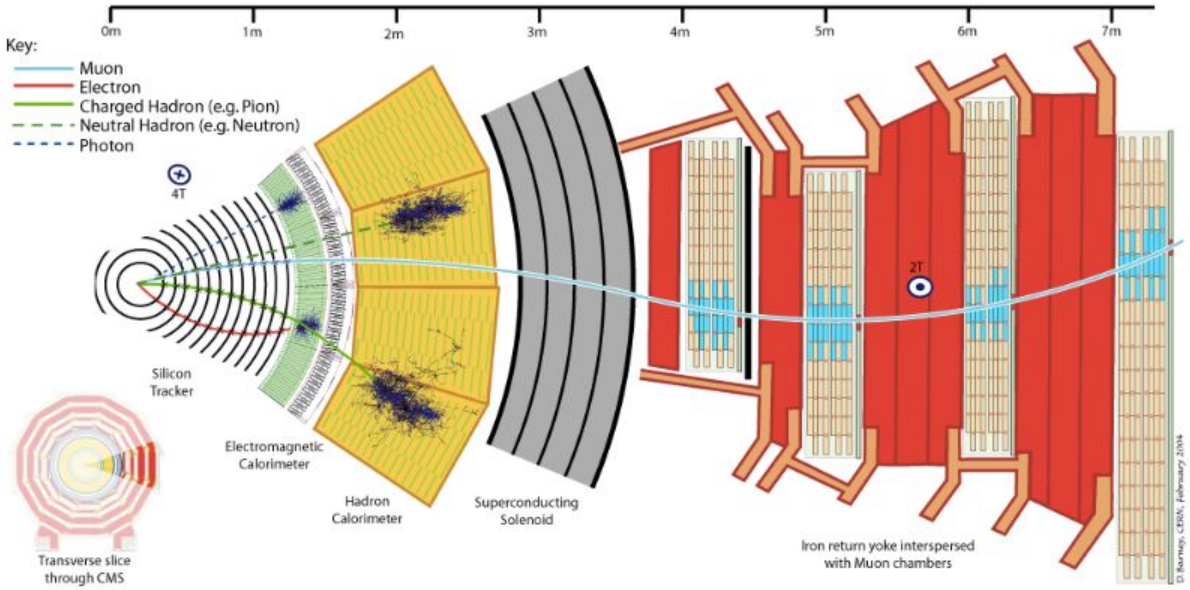

Fig.2.4 : Overview of the CMS detector, as a transverse slice through the barrel. It can be seen that CMS is constructed of sub-detector layers that are designed to measure particles produced in the LHC collisions.[6]

3. Phase-2 Upgrade of the CMS Data Acquisition System

In the previous chapter, the LHC and the CMS experiments were explained along with their purpose and experimental setup. The High Luminosity Phase of the LHC was also described in the last chapter. This High Luminosity phase of LHC would also result in upgradation of the CMS Detector for high luminosity particle collisions. All the various sub-detectors, calorimeters and trackers would be upgraded to improve the precision and the granularity of selecting interesting events. These improved detectors would also mean that the detector read-out electronics and the data acquisition system has to be upgraded for higher data throughput speeds. The High Luminosity LHC (HL-LHC), after the third Long Shutdown (LS3), will provide an ultimate instantaneous luminosity of $7.5 \times 10\ 34\ cm^{-2}\ s^{-1}$, at the price of extreme pileup of up to 200 interactions per crossing[9]. In LS3, the CMS detector will also undergo a major upgrade to prepare for the Phase-2 of the LHC physics program, starting in the second half of 2026. The upgraded detector will be read out at an unprecedented data rate of up to 50 Tb/s and an event rate of up to 750 kHz[9]. Complete events will be analysed by software algorithms running on standard processing nodes, and selected events will be stored permanently at a rate of 7.5 kHz for offline processing and analysis. For comparison, the current running parameters of CMS are 2 Tb/s data rate, 100 kHz event rate, and a total rate to permanent storage of about 1 kHz.

3.1 CMS Detector Upgrades for HL-LHC

The HL-LHC upgrade aims at increasing the integrated luminosity delivered to the experiments after the Phase-2 Upgrade which will be achieved by a combination of increased beam intensity and improved focusing, and as mentioned previously, would require upgradation of various modules of the CMS detector.

	$\mathcal{L}$	$\langle$ PU $\rangle$	Vertex Density	$\int \mathcal{L}$ / year
Baseline	$5 \cdot 10^{34}\,cm^{-2}\,s^{-1}$	140	0.8/ mm	$250\,fb^{-1}$
Ultimate	$7.5 \cdot 10^{34}\,cm^{-2}\,s^{-1}$	200	1.2/ mm	$> 300\,fb^{-1}$

Fig.3.1 : HL-LHC projected running parameters. The instantaneous luminosity $\mathcal{L}$ and average pileup figures, expressed as the average number of interactions per crossing, (PU).

The amount of overlapping interactions(pile-up),their density and the cross-section of these interactions as well as the level of radiation produced in the forward region, dictate an almost complete redesign of the tracker and of the endcap calorimeters. The CMS detector upgrade would include a new silicon tracker with trigger capabilities and extended coverage , high granularity endcap calorimetry largely based on silicon sensors , and extended muon detector coverage to enhance the CMS acceptance and selection power. A dedicated timing detector would be introduced to help distinguish particles originating from the interesting particle collision vertex and timing information will be recorded by the new endcap

calorimeter. Replacement of the front-end electronics of the detector would be required to improve radiation hardness and readout speed . A higher selectivity of the trigger system, a higher readout rate and a higher data throughput [9], will also entail new strategies for the back-end electronics. The upgraded CMS detector will be read out at a rate of up to 750 kHz by approximately 50k high-speed front-end optical links, for an average expected total event size of approximately 7.5 MB[9].

3.2 The CMS Level-1 Trigger

The Level-1 Trigger is an important part of the data acquisition process. The processes that interest the CMS detector and are relevant for new physics usually occur at a rate smaller than 10 Hz[9][10]. Post particle collision, the CMS Level-1 Trigger is responsible for selecting only the most interesting events such that the most relevant data is collected by the detector read-out electronics. The trigger system performs online selection of physics processes of interest whose cross sections are typically orders of magnitude lower than the total proton-proton cross section. Only the most interesting events are chosen for data acquisition because the experiment cannot store all the collision data, even as most of the collision data is not useful for study of physics.

The trigger is implemented as a two-level system consisting of the Level-1 Trigger and the High Level Trigger(HLT) . Level-1 is made up of custom electronics and reduces the event rate from 40 MHz to 100 kHz with a total latency of 3.8 μs[9][10].L1T receives the information coming from the electromagnetic and hadronic calorimeters as well as from the muon chambers. A global decision is taken based on the presence of energy deposits compatible with physics objects such as photons, electrons, muons, jets or hadronically decaying tau leptons[9][10].

The High-Level Trigger (HLT) is a processor farm at the end of data acquisition chain and reduces the event rate from about 100 kHz to 1 kHz[9][10]. The HLT is described a bit more in detail in later sections. The HLT is a streamlined version of the CMS offline reconstruction software running on a cluster of commercial rack-mounted computers which consists of about 26,000 processor cores.

During its Run-2 operation, the LHC delivered proton-proton collisions at a centre-of-mass energy of 13 TeV with a peak instantaneous luminosity larger than 2×10^{34} $cm^{-2}s^{-1}$, more than double the peak luminosity reached during Run 1[9][10]. The upgraded CMS Level-1 trigger was designed to improve the performance at high luminosity and large number of simultaneous inelastic collisions per crossing which is generally called “pile-up” . The trigger was upgraded with a lower trigger rate and better trigger efficiency at increased luminosity.

The key technology and system updates are the replacement of the Run 1 VME-based electronics with μTCA one; the use of the latest generation of FPGAs, Xilinx Virtex 7; serial optical links in place of parallel copper ones (link speeds increased from 1.2 Gb/s to 10 Gb/s); large optical patch panels with single fibres replaced with custom-made commercial solution (Molex Flexplane); online software rewritten to have more common code, modern libraries, more easily maintained [9][10].

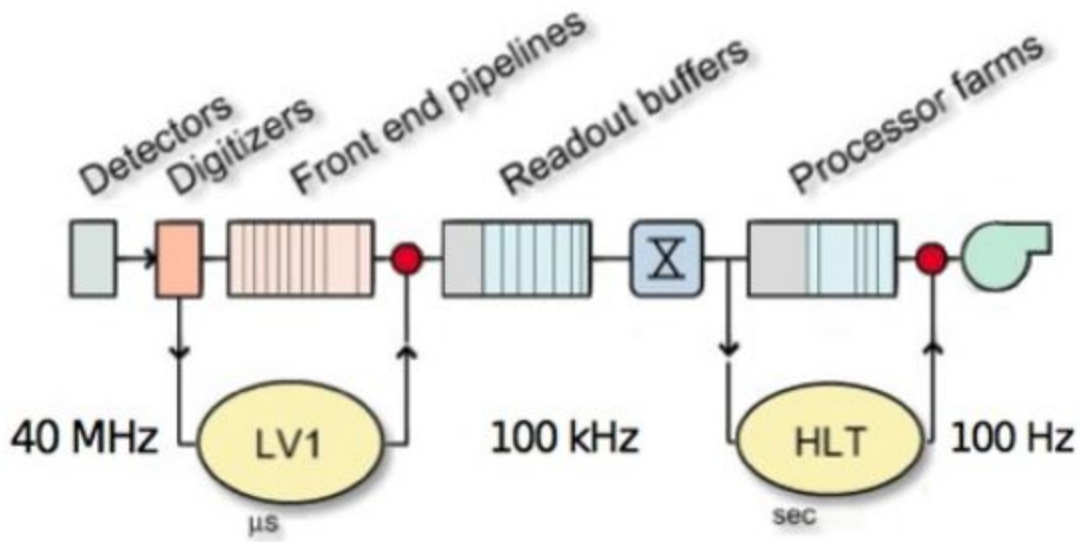

Fig.3.2 : Schematic design of the CMS Trigger System

3.3 The CMS Data Acquisition System (DAQ) for Run 2

The main purpose of the CMS Data Acquisition (DAQ) system or any other High Energy Physics (HEP) DAQ system is to provide the data pathway and time decoupling between the synchronous detector readout that interface with the interesting physics events and data reduction as well as the asynchronous selection of these interesting events and permanently storing the data from these events for offline analysis[9]. The data acquisition system (DAQ) of the CMS experiment at the CERN Large Hadron Collider (LHC) is responsible for assembling data of interesting events at a rate of 100 kHz, transporting event data at an aggregate throughput of more than 100 GB/s to the High-level Trigger (HLT) farm[12][13][14][15].

The current DAQ architecture for the Run-2 has two trigger levels, with one synchronous hardware-based first level (Level-1 trigger, L1) and the second level trigger, the High Level Trigger (HLT), performing selection on full detector event data with software algorithms running asynchronously on standard processors. The systems use the latest technology available at that time during the upgrade before Run-2. For data concentration 10/40 Gbit/s Ethernet technologies are used, 56 Gbit/s Infiniband FDR CLOS network (total throughput ≈ 4 Tbit/s) is used for the event builder explained in later sections. The DAQ - HLT interface is entirely file-based, which decouples the DAQ and HLT systems. The fully-built events are transported to the HLT over 10/40 Gbit/s Ethernet via a network file system[12][13][14][15]. The HLT accepts certain events and the corresponding metadata are buffered on a global file system before being transferred off-site for offline analysis by Physicists. Due to the introduction of new detectors, upgrade of existing detectors, increase in the read-out links from the detectors , the adoption of new computing standards such as microTCA (micro Telecommunication Architecture) , end-of-life for certain front-end electronics and upgradation of optical links was the motivation behind the upgradation of DAQ before Run-2. The second generation CMS DAQ (DAQ2) was designed to handle an event size of 2 MB at a level-1 trigger rate of 100 kHz, resulting in a throughput requirement of 200 GB/s[12][13][14][15].

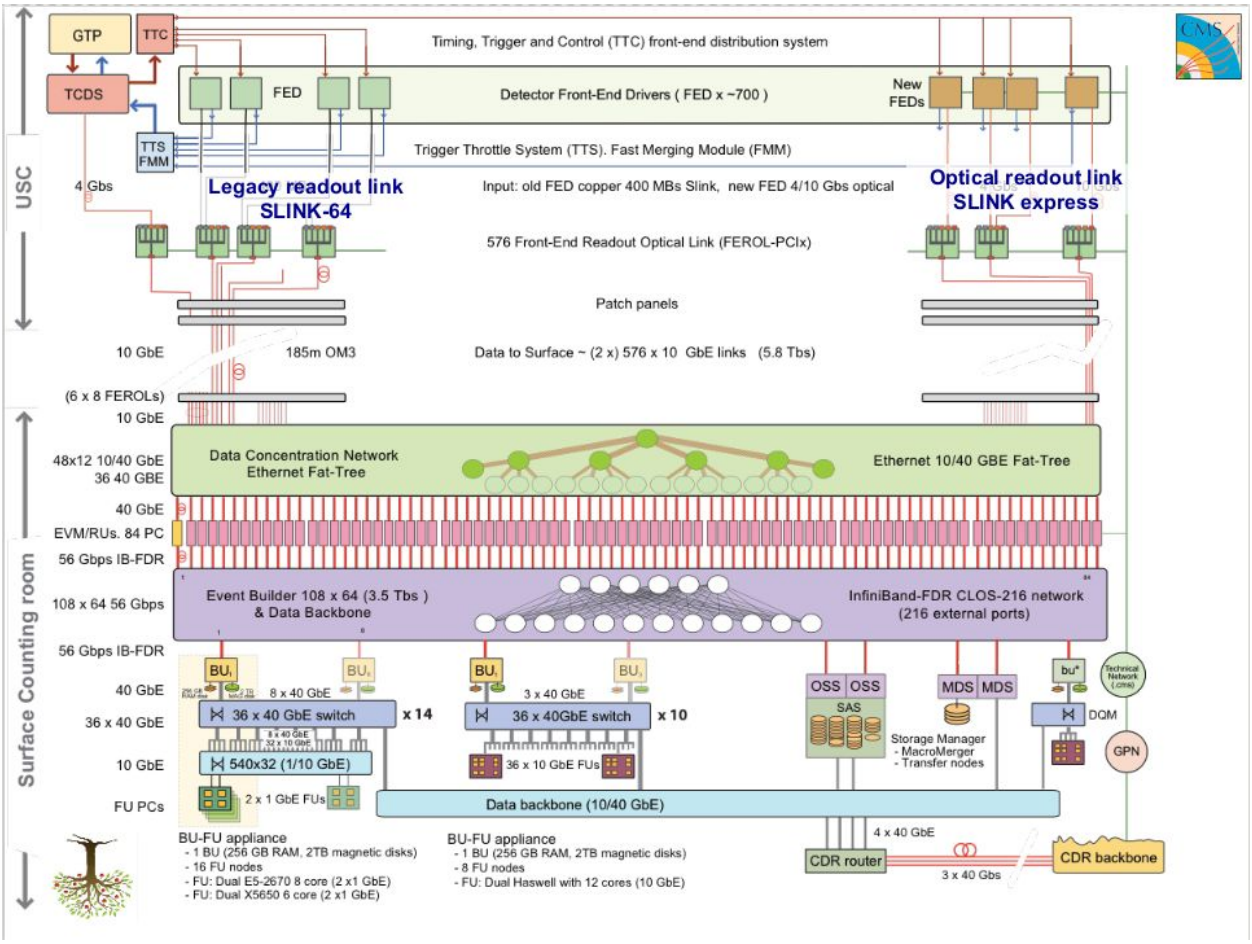

Fig.3.3 : The CMS DAQ for LHC Run-2

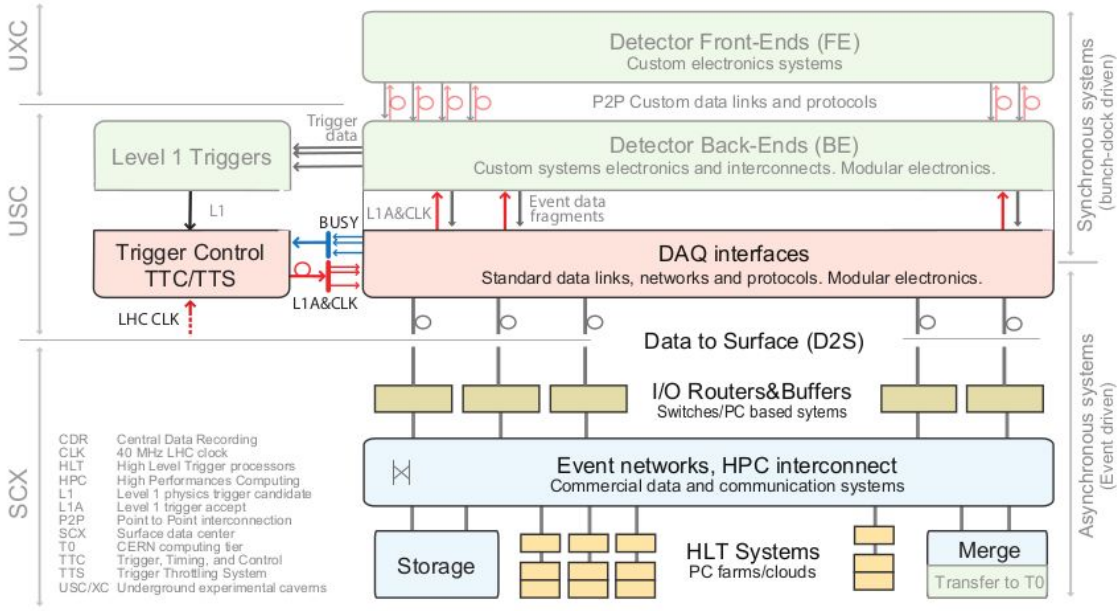

Fig.3.4 : Principle schematics of the baseline Phase-2 CMS DAQ[9]

The above figure outlines the conceptual design of DAQ during Run-2 .The conceptual DAQ2 design still reflects the 'traditional' steps that are typical to HEP(High Energy Physics) DAQ systems: dedicated readout units take care of receiving and buffering incoming front-end data, followed by builder units assembling the data from all sub-detectors into a complete event [9][13][14][15]. It shows sub-detector-dependent back end boards, work synchronously with the machine clock and receive data from the front-end, distribute TTC signals, and generate a TTS state based on the internal buffer occupancy. The Level-1 trigger input data is received by separate 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 occurred. The front-end uplinks will mostly be serial point-to-point bidirectional optical links running at 5–10 Gb/s. Some detectors stream the entirety of their readout directly to the backend at the bunch crossing rate some buffer them at the front-end waiting for a L1 Trigger acceptance. In the former case, data is buffered at the back-end. Only upon the L1 trigger accept, the relevant subset will be handed over to the DAQ.

The synchronous portion of the DAQ system distributes the master clock, trigger accept and fast control signals through the Trigger, Timing and Control (TTC) to the back-end electronics (BE), and collects the buffer status of the individual BE leaf boards in order to control the issuing of accept signals of the L1-Trigger through the Trigger Throttling System (TTS) and prevents buffer overruns. This helps in selecting the interesting physics data as well as prevents the BE buffers from overruns. The main tasks of the synchronous portion of the DAQ are to ensure the collection of all data for events selected by the hardware trigger and that the deadtime remains within a small limit.

In the Phase-1 upgrade before Run-2, MicroTCA based back-end systems were introduced, where the crate-level data aggregation, the TTC distribution, and the collection of the TTS states are combined in a single board. This board is connected to the Trigger Control and Distribution System (TCDS) using a dedicated optical link to a Partition Interface (PI) board, distributing TTC signals and collecting TTS status of multiple crates in a given partition. The aggregated data from one crate are moved, using an ad-hoc protocol, over a point-to-point optical link to a separate board called FEROL.

The DAQ2 front-end readout link (FEROL) is a custom PCI Express board receiving data over optical links from upgraded sub-detectors and interfaces to a legacy custom board, the FRL which is also based on PCI Express, that receives data over copper links from legacy sub-detector systems. The FEROL implements an in-FPGA, one-directional 10 Gbit/s TCP/IP connection at its output, that allows it to connect it to standard networking interfaces from that point onwards.

Data from the front-end readout links are routed to readout unit PCs (RUs) via a 10/40 GbE data-concentrator network. Such high speed links route data from several FEROLs to each RU. The data-concentrator network uses a 'fat tree' layout with three switching layers. These layers provide redundant routing to mitigate possible RU failures. For the event-building network, connecting the readout units to the event builder units (BUs), a 56 Gbit/s Infiniband FDR CLOS network with a total through-put of approximately 4 Tbit/s has been chosen [9][13][14][15]. After events are built in one of the I/O processors called Event Builders, they are stored locally until one of the HLT processors can pick them up and analyse them. The timescale inherent to the HLT processing is of order 1 s, but the allowed latency is larger up to 2 minutes to allow for the large service time fluctuations typical of a selection

process running primarily software algorithms. Accepted events are stored locally before being assembled into larger dataset files used for long term storage and research. This is usually carried out using some form of distributed or network storage attached to the same switched network used for event building.

Unlike the run-1 DAQ system, DAQ2 HLT processes are independent offline jobs augmented with a few modules to facilitate online monitoring and bookkeeping [9][13][14][15]. The file-based communication between DAQ and HLT is made possible by using a network file system (NFSv4) in combination with a fast (10/40 Gbit/s) network []. Each BU is equipped with a 250 GB RAM disk and a 2 TB magnetic disk. BUs and FUs(Filter Units) are grouped in subclusters, and the FUs mount both BU disks over NFS. The BU writes completely-built events onto its RAM disk, from where they are picked up by the FUs and processed [9][13][14][15]. Selected events are written to the FU's local disk, together with the corresponding event metadata. A hierarchical merging procedure combines all selected events and metadata from all filter processes on all FUs, and writes the results to a global, cluster file system, awaiting transfer off-site to the CERN computing center [9][13][14][15].Realistic system configurations containing about 65 RUs and 65 BUs deliver typical throughputs of about 3.5 GB/s per RU, for an aggregated throughput of O(240 GB/s). The presently installed system easily handles the current event size (1 MB) and trigger rate (100 kHz) requirements [9][13][14][15]. In order to reach a total throughput of 200 GB/s the system can be scaled up by adding RUs and BUs.

3.4 The CMS Data Acquisition System Phase-2 Upgrade

As the CMS detector and the sub-detectors would be upgraded for the High Luminosity LHC, the high luminosity and the subsequent high number of collisions would result in higher number of interesting events which would be of interest to the research of the physicists at CERN. As a result, the level-1 trigger event rate would be increased gradually from 100 kHz to 750 kHz [9]. The current read-out electronics and the electronics in the data acquisition network would need to be upgraded to sustain higher data generation from the CMS sub-detectors. The network and the optical links would have to be upgraded and redesigned such that it supports higher data throughput rate. As a result, the whole DAQ would have to be overhauled to meet the demands of the upgraded CMS detector.

The Phase-2 Data-to-Surface(D2S) system will aim to further integrate the functions of data aggregation, Timing,Trigger and Control(TTC) and Trigger Throttling System(TTS) onto a common platform . A DAQ and TCDS HUB(DTH) has been proposed for this integration.

The DAQ and TCDS Hub (DTH) will combine the following functionality [9] :

- Data aggregation for optimal DAQ link usage;
- Translation to a standard protocol (TCP/IP);
- Distribution of Timing signals and Control to each individual back-end board in the crate;
- Collection and (pre)processing of the individual board status for fast monitoring and statistics.

The DTH would aggregate one or multiple input data streams from each individual back-end board in an ATCA crate via dedicated links connected on the front panel, performs data buffering and protocol translation, and provide output links to be routed to the surface [9].

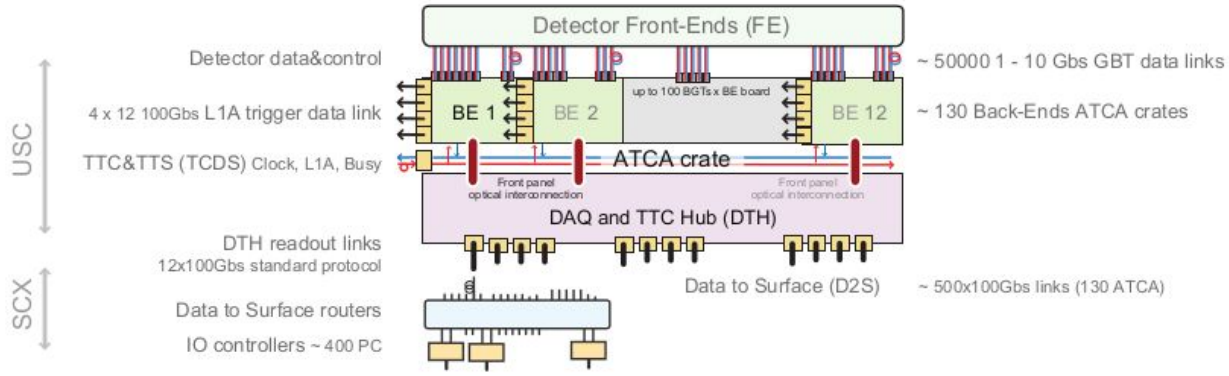

Fig.3.5 : The DTH and its leaf boards arranged in an ATCA crate

The timing and control signals (i.e. LHC clock, L1 Trigger accept, resync, reset, etc.) would be received by the DTH from the TCDS over an optical link, and distributed through the backplane to all node slots. Each node slot would send its TTS status along with monitoring data to the DTH over the backplane. The DTH would then elaborate a global status for use in higher level logic to generate trigger throttling as necessary. Monitoring data would also be aggregated or preprocessed if needed, and made available for debugging, diagnosis and statistical analysis done by system administrators.

SubDet	Av throughput (Tb/s)	Av throughput per BE crate (Tb/s)	Av throughput per BE board (Gb/s)
Outer Tracker	6.90	0.38	32
Track trigger	0.06	0.003	<10
Inner Tracker	8.64	2.16	360
MIP Timing Det. - BTL	0.14	0.14	13
MIP Timing Det. - ETL	0.22	0.22	43
ECAL Barrel	9.49	0.79	88
HCAL Barrel	1.45	0.72	80
HCAL HO	0.18	0.18	-
HCAL HF	0.36	0.36	-
Endcap CALO	12.00	1.33	111
Endcap CALO TPG	1.50	0.10	<10
muon DT	0.78	0.10	<10
muon CSC	1.20	0.60	-
muon GEM - GE1/1	0.01	0.01	-
muon GEM - GE2/1	0.01	0.01	<10
muon GEM - ME0	0.71	0.71	59
muon RPC	-	-	-
Level1	0.90	0.06	<10

Fig.3.6 : Projected readouts from sub-detectors for CMS DAQ Phase-2 Upgrade[9]

SubDet	BE crates	DTH1200 per crate	DTH400 per crate	DTH1200 p. SubDet	DTH400 p. SubDet	Min. D2S links p.crate	Min. D2S links p.SubDet	notes
Outer Tracker	18	1	-	18	-	4	72	([a])
Track Trigger	18	-	1	-	18	1	18	
Inner Tracker	4	2	-	8	-	24	96	
MIP Timing Det. - BTL	1	-	1	-	1	2	2	
MIP Timing Det. - ETL	1	-	1	-	1	3	3	
ECAL Barrel	12	1	-	12	-	9	108	
HCAL Barrel	2	1	-	2	-	9	18	
HCAL HO	1	-	1	-	1	2	2	
HCAL HF	1	-	1	-	1	4	4	
Endcap CALO	9	2	-	18	-	18	168	
Endcap CALO TPG	12	-	1	-	12	2	24	
muon DT	8	-	1	-	8	1	8	
muon CSC	2	-	1	-	2	6	12	
muon GEM - GE1/1	1	-	1	-	1	1	1	
muon GEM - GE2/1	1	-	1	-	1	1	1	
muon GEM - ME0	1	-	1	-	1	9	9	
muon RPC	-	-	-	-	-	-	-	
Level1	14	-	1	-	14	1	14	
BRIL	1	-	1	-	1	1	-	
Contingency	6	-	1	-	6	1	6	
Total	113			58	68	-	560	

[a]Conservatively assigned the DTH-1200 to the OT. In case the throughput deviations across the BE boards from the average is small, a DTH-400 (max. 25 Gb/s bandwidth per leaf) is sufficient.

Fig.3.7 : Projected minimum number of DTH and DAQ D2S optical links (100 Gb/s) for each sub-detector.[9]

The CMS trigger and data acquisition system will continue to feature two trigger levels, with one synchronous hardware-based first level (Level-1 trigger, L1) , consisting of custom electronic boards and a second level trigger, the High Level Trigger (HLT), performing selection on full detector event data with software algorithms running asynchronously on standard processors. This system would continue to be deadtime-less, such that pipelines at every stage of the synchronous process would be sized in such a way as to store data for the maximum latency required to decide whether to pass them on or drop them, under normal conditions, without losses. Normal conditions refer to the characteristic size of the data fragments and the maximum data rate sustained by the links.

Both the total throughput and the total event size result from the simple addition of the individual estimated average contributions from each sub-detector. The total throughput can be used to estimate the minimum number of D2S links required in a perfectly optimized arrangement. The total average throughput requirement for the event network is 44 Tb/s. The total required bandwidth must allow for a switch efficiency that can be of 30 to 50%, depending on the configuration[9]. Hence, the total event network bi-sectional bandwidth required will be between approximately 90 and 150 Tb/s [9].

CMS Phase-2 detector projected data links, ATCA back-end configuration and event size summary. Data are obtained from the technical design reports wherever possible. Average throughput estimated from event size assuming 750 kHz Level-1-accept rate.

Subdetector	Front-end lpGBT links[a]	Sub-event size (MB)	Back-end boards	Back-end crates	Average throughput (Tb/s)	Notes
Outer Tracker	13000	1.15	216	18	6.90	([b])
Track Trigger		0.01		18	0.06	
Inner Tracker	1260	1.44	24	4	8.64	([c])
MIP Timing Det. - BTL	1000	0.02	11	1	0.14	
MIP Timing Det. - ETL	438	0.04	5	1	0.22	
ECAL Barrel	10000	1.58	108	12	9.49	
HCAL Barrel	other	0.24	18	2	1.45	
HCAL - HO	legacy	0.03	-	1	0.18	([d])
HCAL - HF	other	0.06	-	1	0.36	([d])
Endcap CALO	8000	2.00	108	9	12.00	
Endcap CALO TPG	9000	0.20	144	12	1.50	
muon DT	3840	0.13	84	8	0.78	
muon CSC	other	0.20	-	2	1.20	([d])
muon GEM - GE1/1	other	0.002	-	1	0.01	([d])
muon GEM - GE2/1	144	0.001	8	1	0.01	
muon GEM - ME0	216	0.12	12	1	0.71	
muon RPC	other	-	-	-	-	([e])
Level1		0.15	120	14	0.90	([f])
Total		7.4	>858	>106	44	

[a] Count of links based on lpGBT-VL+ (see Appendix A), "Other" links include the use of VL(+) components with radiation-tolerant FPGA at the front-end.

[b] The Outer Tracker uses the same links to multiplex trigger and DAQ data — the proportion of trigger data is expected to be roughly 80%.

[c] Assumes some compression factor is achievable in the back-end board, see text of Section 1.3.3.

[d] Legacy MicroTCA back-end, assume ATCA adapter.

[e] Only connected to Level-1 system.

[f] The number of crates is estimated from the number of boards in the current Level-1 Trigger Interim Document draft, the throughput from the estimated output bandwidth from each stage scaled to 750 kHz.

Fig.3.8 : Projected throughput of the new DAQ system

3.5 Timing and Control Distribution System (TCDS)

The CMS Run-2 Trigger Control and Distribution System (TCDS) evolved from three Run-1 systems that were []:

- the Trigger Control System (TCS), previously part of the Global Trigger,
- the Trigger Timing and Control (TTC) system
- the Trigger Throttling System (TTS).

The trigger control part of the TCDS orchestrates the data-taking based on trigger conditions and subsystem readiness. The distribution part serves two main purposes [16], which are : controlling distribution of Level-1 accepts and timing signals (i.e., the TTC information) to the back-end and front-end electronics of all CMS sub-detectors, and collect readiness (TTS) information across all

sub-detectors and combines that into an overall throttling signal that stops triggers when readout buffers are full or out of sync.

The CMS TCDS distributes timing and control data also known synchronization data to the detector front-ends and receives back status information which informs the status of the readiness of the detector systems to handle more triggers. The clock reference that is distributed along with the fast control information is synchronous with the particles in the LHC accelerator and this clock is used to keep the data-taking in step across the various detector systems that make up CMS. The detector systems are able to provide basic status information (TTS) regarding their readiness for accepting more triggers by sending data back to the TCDS. Physics triggers are received from the GT (Global Trigger) system by the Central Partition Manager (CPM) module that is the controller responsible for orchestrating global data-taking in CMS detector having multiple sub-detectors. The system clock (synchronous with particles in LHC accelerator) is also received by the CPM for further distribution. A number of Local Partition Managers (LPMs) that sit on the same crate (ATCA or microTCA) orchestrate local data-taking for a particular sub-detector in CMS.

These LPMs contains eight identical firmware blocks that receive the generic synchronization commands emitted by the Partition Manager (Central or Local) and translate these commands into the sub-detector specific commands that are understood by the detector front-ends. The final module in the chain is the Partition Interface (PI). The PI serves as a fan-out for downstream clock and data (TTC), and an intelligent OR- based fan-in for the status information (TTS) coming back upstream[16].

3.6 CMS DAQ Event Builder

A 56 Gbps Infiniband FDR CLOS network has been chosen for the event builder which assembles fragmented event data obtained from the readout units (RU). The DAQ collects data from more than 600 custom detector Front-End Drivers (FEDs). The FEROL interfaces to the legacy FEDs through the FRL as mentioned previously. The FEDs send data fragments of 1–2 kB. Based on the fragment size, one or two FEDs might be assigned to one FEROL. The fragment size of the new FEDs is expected to be ~8 kB[17]. The FEROL sends the data from each FED as a TCP/IP stream over optical 10 Gb/s Ethernet links to one of the 72 readout units (RU) computers. A series of switches is used to concentrate the data onto 40 Gb/s Ethernet links and transport the data to the surface. Each of the RUs assembles the data from a predefined set of FEROLs into super-fragments. The super-fragments are then sent over the event-builder switch to one of the 62 builder units (BU) machines.

The BU assembles the super-fragments into complete event data. The events' consistency is checked by the payload of each FED. Finally, the BU writes the event to files residing on a local RAM disk which is then exported via NFSv4 to 10–20 filter units (FUs) which run the high-level trigger code.

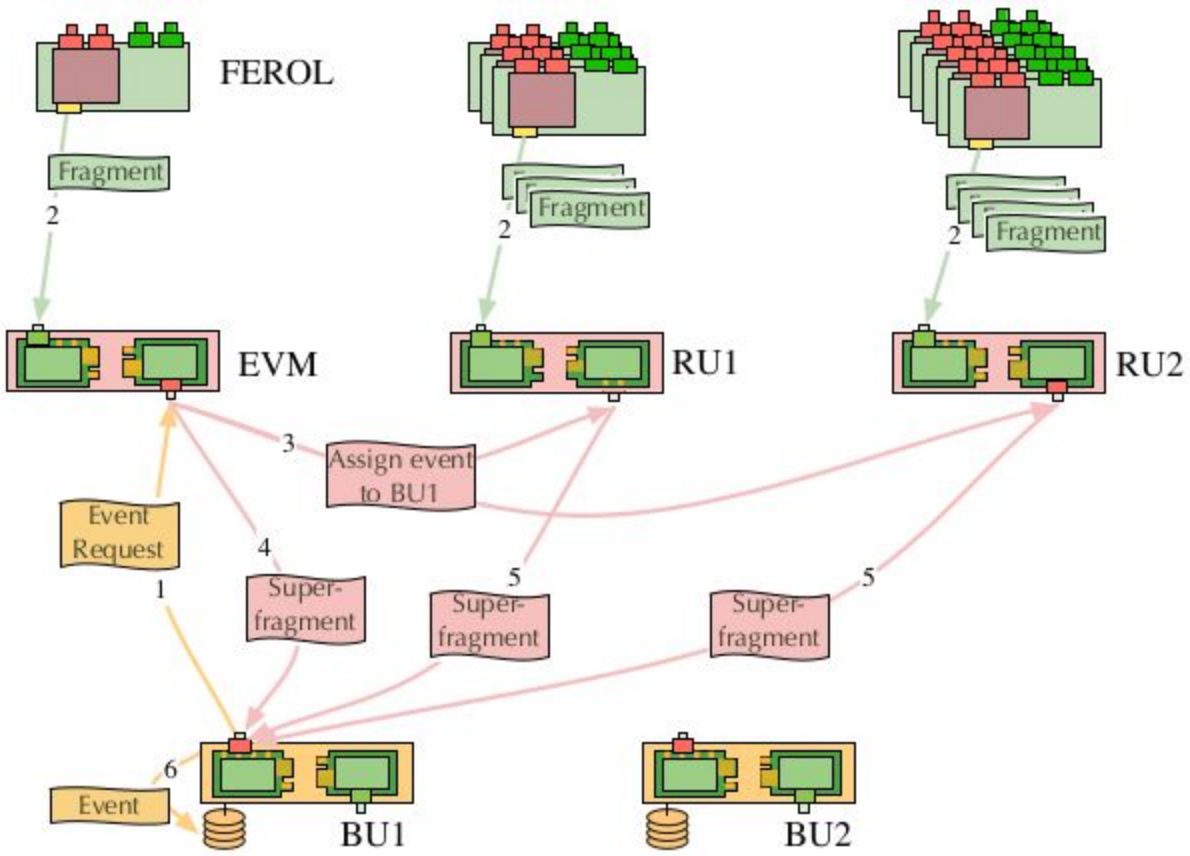

Fig.3.9 : The event building process as explained in section 3.6

3.7 High-Level Trigger (HLT)

The High Level Trigger (HLT) is implemented in software running on a processor farm of commercial computers, running over 26,000 CPU cores and reduces the L1 output rate to a sustainable level for storage and physics analysis about 1 kHz [10][11]. The HLT software consists of standardised offline reconstruction algorithms, which comply with the strict time requirements of the online selection to select data which mirrors the particle interaction that is under the observation of the CMS detector. The average processing time in 2012 was about 200 ms per event[10][11], two orders of magnitude less than the offline reconstruction. Most of the bandwidth is devoted to data for physics analysis.

The HLT's task is to ensure a large acceptance for physics signals, while keeping the output rate and CPU time under control. The software algorithms running on the HLT processor farm ensure high reconstruction efficiency and allows for stringent identification criteria and quality selections to reduce the trigger rate.

4. Fundamentals of Embedded Computing Systems and GNU/Linux

4.1 Overview of Embedded Systems

An embedded system is a computing system that is built to control a function or a range of functions []. It is a subsystem of a more sophisticated device or machine, like an engine management unit system of a car or microprocessor within various appliances. We could assume that a general processing computer is made up of a number of embedded systems such as bus controllers, audio controllers, video controllers and graphical processing units (GPU), systems which could be considered embedded systems in their own right. There are many other embedded systems around us such as WiFi routers, pacemakers, fitness bands etc. Today, many complex embedded systems are increasingly being employed in a bigger machine to collect data , do local processing and in some cases employ neural networks within these complex systems like in the case of CMS TDAQ. Embedded systems are designed to perform a specific set of tasks such as monitoring an environment, collect data or send them over a network. They are not suitable for being used as a general purpose computers because of their limited resources. Due to increase in sophistication of embedded systems in terms of increased density of integrated circuits, cheap and compact memory, increased features, increased power efficiency and increased capacity of performing multiple tasks and even running multiple processes at a given time. Smartphones and Tablets are a type of modern Embedded Systems. Typically, Embedded Systems are small in size and have a small footprint. Embedded systems usually have constraints in terms of power, storage memory and RAM due to their small size. While, most portable embedded systems are powered by batteries, most embedded systems situated in the CERN TDAQ network, do not have power constraints as they have continuous power supply from the grid. Unlike other embedded systems, these embedded systems consume a lot of power. Due to advances in electronic technology and the increasingly compact size of electronic devices, it is possible to design our own embedded systems from Commercial-Off-The-Shelf(COTS) systems. Some Embedded Systems, involved in critical applications or installed at critical locations such as Automobiles, Security facilities, Airplanes etc. are also real-time systems. They are called Real-time systems as they must respond to certain tasks within a particular deadline at all times without failure, depending on the priority of the tasks. While some embedded systems do not require operating systems and generally follow their tasks in a sequential manner, some embedded systems require operating systems, much like in case of smartphones and general purpose computers. Real-time Operating systems (RTOS) are used in real-time systems and sometimes in embedded systems where system administrators do not wish to access the embedded systems in the run-time but want an Operating system which can manage multiple tasks which the embedded system is expected to run. Due to modern embedded processors having improved processing power and memory similar to the general purpose processors, it is possible run general purpose operating systems such as Ubuntu, CentOS etc., similar to the Operating systems used for System Administration at CMS TDAQ. The Linux kernel needs to configured for the processor hardware which it is going to be used for. Linux based Operating Systems are not real-time as the kernel needs to be configured for real-time applications. Examples of RTOS are VxWorks, FreeRTOS and modified Linux based OS called Real-time Linux, which is not entirely real-time. Embedded systems can be single-board

computers with the components and memory on a PCB but modern Embedded Systems come in the form of System-on-Chip(SoC),which is an embedded system designed within a Chip thanks to the improvement in semiconductor. During this project, we work with similar SoCs such as Xilinx Zynq 7000 SoC and Zynq Ultrascale+ SoC which have ARM processors with ARMv7 and ARMv8 architectures respectively.

4.2 Overview of Operating Systems

Operating system is a set of software modules, which work in tandem for management of the computer system's hardware. Most computing systems and embedded systems operate under the control of an operating system. A kernel, software libraries, root filesystem and a number of utility programs are regular features of a Unix-like Operating System[18][19]. A piece of software called a bootloader starts the kernel when the computer system is booted on. The kernel is a program which manages the resources that various applications in a computer require and by far the most important component of the operating system. Resources are not only allocated among processes but also among users, which is a central feature of the AFS(Andrew File System) of the CERN CMS IT infrastructure, where multiple users modify, store and access data as the file system mounts over the network. The GNU/Linux kernel is discussed in detail in following sections. Most modern processors support multiple privilege level such as the Intel x86 processors which supports four privilege levels. GNU/Linux uses two, level 0 - for executing the kernel code and level 3 - for running user code. The reason for this division is to prevent user code from directly accessing certain hardware resources directly, which it is not supposed to. The kernel accesses the hardware through its subsystems and essentially controls access to the hardware. Process management is a key function of the kernel where it manages various processes or programs running on the computer, by allocating them resources and CPU time. Modern processors with multi-core processing facilities can handle multiple processes at a time. A scheduler decides which process is allocated a time slice. In RTOS, scheduler schedule as per priority of the process. Kernels also manage inter process communication. To manage the computer memory efficiently and effectively, kernels make use of virtual memory, which is the memory allocated to a process. The virtual and physical memory are divided into pages. When addressing a virtual memory, the virtual page frame is translated into a physical page frame number and the offset of the page frame is used to point to a specific memory location within the page. Kernels also look after the filesystem management, providing the functions and data structures required to manage files and directories within the memory. As mentioned previously, Kernels also control the access to various hardware attached to the computer system, through device drivers present in the kernel. The focus of this project is primarily to study the best GNU/Linux distribution for the embedded systems that would be installed in the CMS TDAQ network.

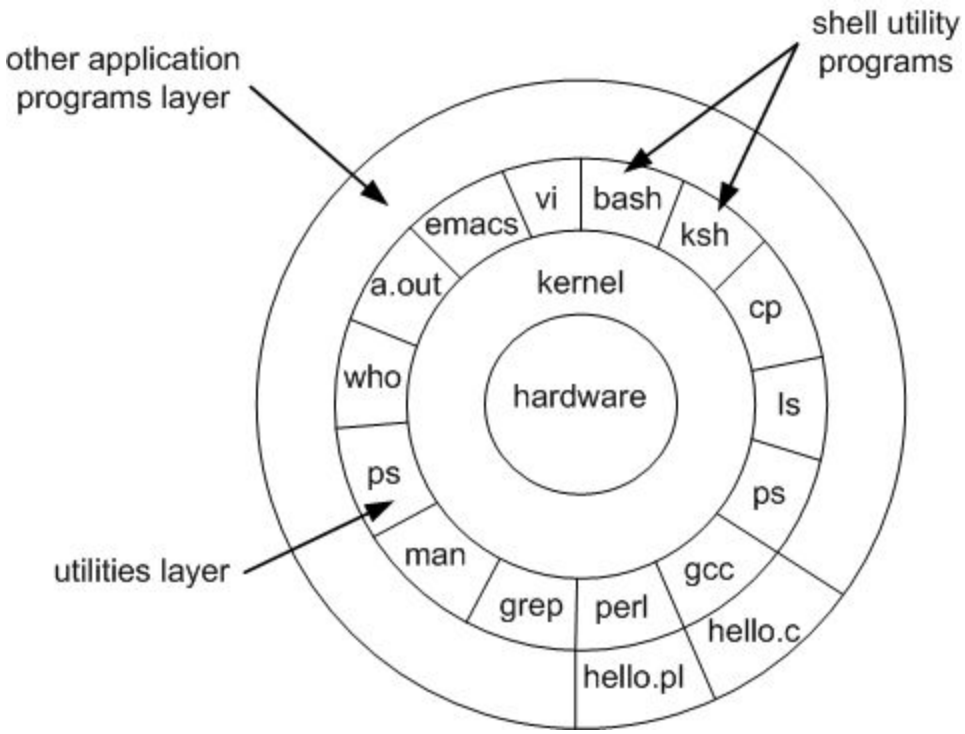

Fig.4.1 Architecture of an Unix-like Operating System

4.3 The GNU/Linux

GNU and Linux are often used interchangeably to refer to certain Unix-like Operating Systems.

GNU is an operating system and an extensive collection of computer software and is composed wholly of free software, most of which is licensed under the GNU Project's own General Public License (GPL). GNU is a recursive acronym for "GNU's Not Unix!",since GNU's design is Unix-like, but differs from Unix by being a free software and containing no Unix code.

Linux is a family of open source Unix-like operating systems based on the Linux kernel, an operating system kernel first released on September 17, 1991 by Linus Torvalds. Linux is typically offered to users in a Linux distribution, which is a collection of packages and utility software that come tightly integrated with a special flavour of Linux kernel.

Naming these operating systems based on similar concepts is a big dispute between members of the free software community and open-source software community over whether to refer to computer operating systems that use a combination of GNU software and the Linux kernel as "GNU/Linux" or "Linux".Proponents of the term “Linux” argue that it is far more commonly used by the public and media, and that it serves as a generic term for systems that combine that kernel with software from multiple other sources.Proponents of the term GNU/Linux note that GNU alone would be just as good a name for GNU

variants which combine the GNU operating system software with software from other sources, something which Linux itself does.

The GNU/Linux operating system software consists of the following components : the Bootloader, Kernel, the Shared libraries and the System commands and utilities such as the shell which is used to access the operating system services through a command line interface[18]. A full GNU/Linux distribution also includes multiple user application which it supports, but these are add-ons which are customisable by user and not integral to the operation of the underlying operating system. The scope of this thesis project would to explore linux distributions for embedded systems in the CMS TDAQ, thus studying the underlying components of the GNU/Linux is the primary goal of this chapter. The bootloader is the first piece of software to execute when the Computer powers up and ideally relinquishes control to the kernel. There are some exceptions to it. In PCs and Embedded Computers, the bootloader or the First Stage Bootloader (FSBL) can be interrupted from loading the kernel to enter boot-configuration setup or the U-Boot in case of Embedded Computers. This would be explored further in the next chapter. In case of space constraints such as in the case of Embedded computers, Boot-loader can be divided into multiple stages. The first stage bootloader is a small file and located in a specific part of system's disk or the SD card in case of embedded systems. This memory has to be a non-volatile memory. The first stage bootloader transfers control to the next stage of bootloader. These control transfer processes can sometimes also be interrupted to enter boot configuration as mentioned previously. The kernel initialises various systems and processes for which the Kernel is configured and starts the system's daemon processes such as the user login process. Daemon's are user space processes that run in the background rather than being under the control of the interactive user. Shared libraries are collections of object code that are linked into programs either at compile time or at run-time. A GNU/Linux system also requires one or more filesystem, called the root filesystem, which contains all crucial files necessary for the operating system operations and also contains the necessary files for installed user applications. The kernel and bootloader sometimes reside together on the boot filesystem.

4.3.1 The GNU/Linux Bootloader

The bootloader is a piece of software which executes when the system is booted and is primarily responsible for loading the kernel as well as some other components of the operating systems[18]. The boot process can be different for different hardware and microprocessor architecture. However the basic principle remains the same. The code for the bootloader or a part of it, like the first stage bootloader(FSBL) is stored in the boot sector of a disk , called the boot partition. This partition may or may not be necessary depending on the hardware and the Linux distribution. In case of Archlinux, we need a boot partition for ARMv7 architectures , however for the same architecture the Xilinx-Linux kernel based distributions need not always have separate partitions like in case of Zynq-7000 booting that we witness in Chapter 8. The boot-sector also contains the partition table. The bootloader is located either on the first block of the disk or in first block of a partition. In PC nomenclature, there are two types of boot sectors, namely :

- **Master boot record (MBR)** : the boot sector located at the start of a partitioned disk.

- **Volume boot record (VBR)** : the boot sector located at the start of a non-partitioned disk or at the start of a partition of the disk.

A bootloader for booting the kernel or a boot manager that chain loads bootstrap code in a VBR, could be located on the MBR. The bootloader has to be loaded from disk into memory before its execution. Many systems using x86 processors use the basic input output system (BIOS) for this purpose. However, the Xilinx ARM processors which we use for this project do not have the BIOS facility. We look closely at the Xilinx SoC boot processes at later stages in the document. On booting the computer which supports BIOS, it executes a jump instruction which passes control to the BIOS code in ROM. The BIOS initializes some hardware and then performs a power-on self-test (POST). If the POST is successful, the BIOS loads the bootloader code and the partition table from the boot partition at a specific processor memory location. The bootloader in its first instruction, loads the kernel into processor memory and relinquishes control to it. Due to its small memory size, there are limitations to the functionalities of MBR. Due to this, the MBR loads a second stage bootloader from the hard disk or a non-volatile memory like SD Card in case of embedded processors. There can be multiple bootloader stages before loading the kernel. There exist many bootloaders for booting Linux kernels[18]. A few of the popular ones are described below:

- **LILO** : The Linux loader (LILO) was the default loader for GNU/Linux distributions in the early years.LILO has been superseded by GRUB as the default bootloader.

- **GNU GRUB** : Grand Unified Bootloader(GRUB) is a sophisticated bootloader that is used extensively with GNU/Linux systems. It comes in two flavours, GRUB1 and GRUB2 though there is no longer any development for GRUB 1 and it is being phased out in favour of GRUB 2. However, GRUB is not used for Linux distributions in embedded processors such as Xilinx Zynq processors or other ARM processors.

- **SYSLINUX** : SYSLINUX is a lightweight bootloader which is ideal for GNU/Linux embedded systems. SYSLINUX can only boot from FAT and NTFS file systems.

- **U-Boot** : U-Boot is an open source bootloader that is frequently used in the Linux community, and used by Xilinx for the MicroBlaze processor and the Zynq-7000 AP SoC processor for Linux. It is available for various architectures, for example: ARM, AVR32, MIPS, PPC, x86, 68k, Nios, and MicroBlaze.

There are a number of SYSLINUX derivatives:

- **ISOLINUX** : Bootloader for ISO 9660 CD-ROM filesystems.
- **PXELINUX** : Bootloader for booting over the network from a remote server. This is based on the preboot execution environment (PXE) system.
- **EXTLINUX** : SYSLINUX derivative can boot from ext2, ext3 and ext4 filesystems.

4.3.2 The GNU/Linux Kernel

The Linux kernel operates controls the access to the computer hardware for the user-space applications.The kernel ensures the integrity of the system and enforces security, when the computer system has multiple users and multiple users authorised with modifications to the root filesystem and software packages. Linux kernel was first developed by Linus Torvalds[18][19], but today multiple versions of Linux Kernel exist, for specific hardware, which are developed and maintained by a community of developers. The Open Source community has a great contribution towards developing new features for the linux kernel. The Linux kernel is released under the GNU General Public License version 2 (GPLv2) The kernel source directory is divided into subdirectories, which have been outlined below :

- l**inux/kernel** : The core kernel code.
- **linux/include** : Contains header files for both kernel and user applications.
- **linux/arch** : Architecture dependent parts of the kernel.
- **linux/drivers** : Device driver code.
- **linux/fs** : Code the virtual file system (VFS) and the physical filesystems.
- **linux/init** : Architecture independent boot code and the initial entry point to kernel.
- **linux/ipc** : Source code for interprocess communication (IPC) methods, namely, semaphores, shared memory and messages.
- **linux/mm** : Memory management functions, namely, allocation of memory, memory mapping and paging/swapping.
- **linux/scripts** : Supporting scripts for kernel configuration, patching and documentation.

The Linux kernel resides in the /boot directory (typically) in either of three file formats: Image, the compressed zImage and bzImage (big zImage). The Xilinx Zynq processors also use either the Image or zImage file formats for loading the kernel. The kernel image consists of a compressed part preceded by an uncompressed header. After the bootloader relinquishes the control to the kernel, the kernel completes some initialisation tasks and then it switches to protected mode. The kernel is loaded into processor memory before decompressing the compressed part of the image. The kernel creates the first user process and a process ID (PID) of 1 is assigned to it. The init process is then initiated by the kernel. The Linux kernel is highly customisable, since many drivers and different configurations can be tested with the Linux kernel, as long as it is compatible with the processor hardware. These options are either built into the kernel image or as a loadable module, however during this project we have tried to configure the Xilinx Linux kernel before building the kernel image. While using the latter, an option in the kernel needs to be set to support loadable modules. Embedded systems tend to have fixed hardware configurations and, therefore, a set number of device drivers which must be loaded at all times. The kernel is dynamically configured to include the crucial drivers required by the embedded processors as they are critical to its functioning. In the case of Xilinx Zynq processors, this configuration can be automated by using the board support package (BSP) provided by Xilinx. This would be described in detail in following sections.

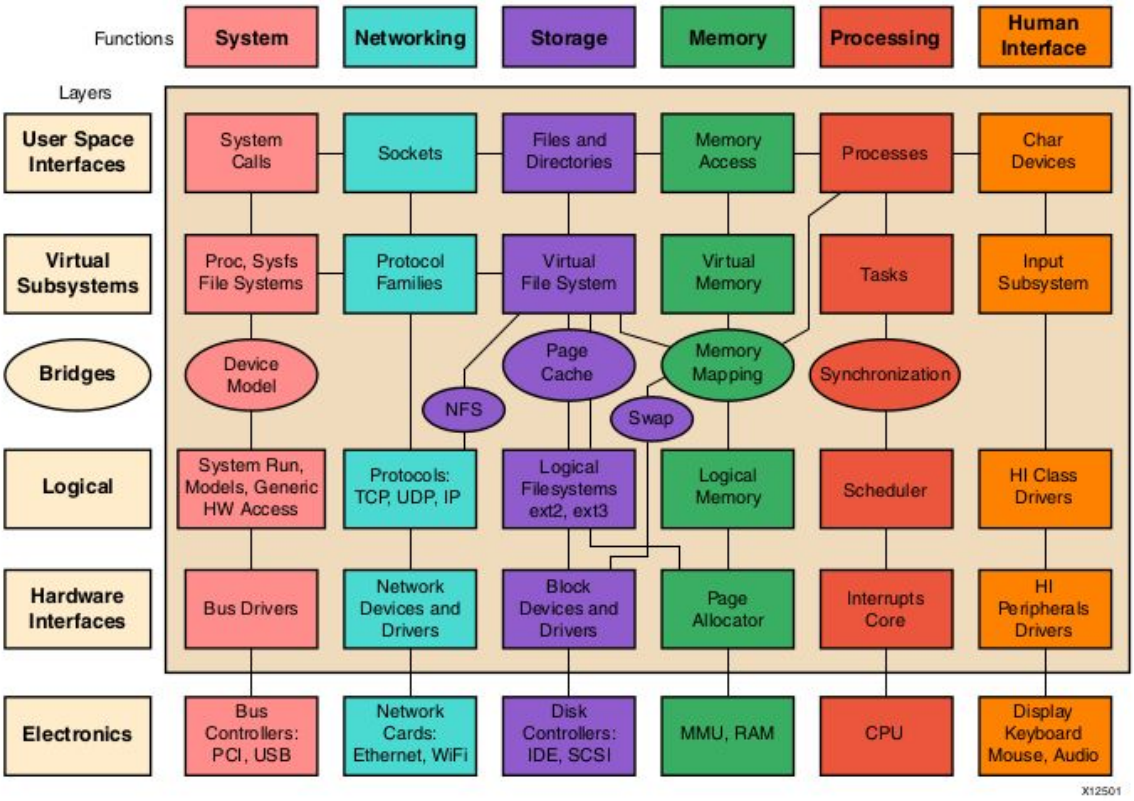

Fig.4.2 : The Linux Kernel (Source : Zynq Software Developers Guide [32])

4.3.3 The GNU/Linux Init Process

The init process is the first user space process initiated by kernel. All processes running on the system are but a result of the init process which initialises the system after the loading of the kernel into the memory is over. Multiple versions of init process exist. We examine the System V initialisation procedure which adopts the concept of runlevels. Run levels determine the state of the machine post-booting , thereby indicating the status of the processes being run in a given run level. The following different run-levels for Linux-Standard-Base(LSB)[18] :

- Single-user mode
- Multi-user mode with no network services.
- Multi-user mode with network services.
- Shutdown
- Reboot

System V runlevels have been deprecated in modern GNU/Linux systems. The common “init” process is systemd which has its own runlevels, which is a powerful form of “init”, but a bit more complex. Due to its complexity, we don’t use it for embedded processors like the ARM series processors. For Zynq and Zynq Ultrascale+, Xilinx continues to use INIT. The run-levels may vary depending on the Linux distribution.

ID	Name	Description
0	Halt	System shutdown
1	Single-user mode	Single-user mode for administrative tasks
2	Multi-user mode	Multi-user mode, no networking
3	Multi-user mode with networking	Starts the system
4	Not used/User-definable	For special purposes
5	Multi-user mode with GUI	Same as runlevel 3 with display manager
6	Reboot	Reboot the system

Fig.4.3 : The init process run-levels and their corresponding functions[18].

A simple demonstration of " who -r" command in the terminal gives us the run-level as shown in the figure below :

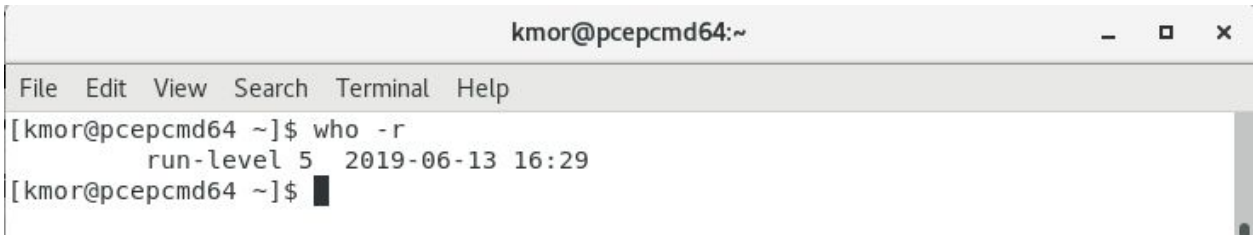

Fig.4.4 : Computer operating in run-level 5, i.e multi-user mode with a graphical user interface, a common run level for most PC users

4.3.4 The GNU/Linux Root Filesystem

The GNU/Linux file space is organised into directories, such that the directory structure imparts a logic to the file system. The files that support the operating system in its functioning are stored within this directory structure such as [18] :

- Kernel and loadable modules
- Second (and third) stage bootloaders
- Startup/shutdown scripts
- Configuration files
- System utilities

The GNU/Linux file space comprises of several filesystems "mounted" at various points called directories within the filesystem hierarchy. The root filesystem is mounted at the "/" directory (called the "root" directory), sitting at the top of hierarchy. Files and packages for software are present in various sub-directories but the kernel and bootloader could be located somewhere else, such as the SD Card in case of embedded systems located in systems that are not easily accessible. A variety of filesystem types are supported by GNU/Linux such as :

- **ext2/ext3/ext4** : The Extent filesystem is the default Linux filesystem. The original Extent filesystem, ext, has been deprecated and replaced with ext2, ext3 and ext4.
- **ReiserFS/Reiser4** : A journaling filesystem. Reiser4 is the successor to ReiserFS.
- **SquashFS** : A read-only filesystem that compresses, files, directories and inodes. SquashFS is aimed at devices with limited storage, such as embedded systems.
- **XFS** : A high performance journaling filesystem. Parallel I/O can be performed on XFS filesystems.
- **FAT/NTFS** : Microsoft filesystems.
- **NFS** : Network File System.

```
├── boot
│   ├── efi
│   ├── grub
│   └── grub2
├── cvmfs
│   ├── cms.cern.ch
│   └── cvmfs-config.cern.ch
├── dev
│   ├── block
│   ├── bsg
│   ├── bus
│   ├── cc_pcepcmd64
│   ├── char
│   ├── cpu
│   ├── disk
│   ├── dri
│   ├── fd -> /proc/self/fd
│   ├── hugepages
│   ├── input
│   ├── mapper
│   ├── mqueue
│   ├── net
│   ├── pts
│   ├── raw
│   ├── shm
│   ├── snd
│   └── vfio
```

Fig.4.5 : A portion of the root file tree structure at a computer in the CMS department. It shows the boot directory containing the grub bootloaders and the dev directory with various device drivers present in the CentOS root file-system.

The GNU/Linux directory structure is defined in the filesystem hierarchy standard (FHS) and is governed by the Free Standards Group [18]. The purpose of the FHS is to enable software and users to predict or guess the location of files and directories. Having FHS allows easy look up of essential files and packages during the execution of various processes in GNU/Linux system. A description of some of the main directories is given below:

- **/bin** : Essential command binaries. These are commands that can be used by all users, including superuser, administrator and a general user. The entries in the directory can either be hard links, where the binary file is present in the folder or could be symbolic links which have to be declared by the user. There must not be any subdirectories under /bin.
- **/boot** : Bootloader files. The configuration files and further stages of the bootloader are kept under this directory The kernel file can also be in /boot.
- **/dev** : Device and special files.
- **/etc** : Host-specific system configuration files.
- **/lib** : Shared libraries and kernel modules.
- **/media** : Removable media mount point.
- **/mnt** : Mount point for mounting a filesystem temporarily.
- **/opt** : Add-on application software packages.
- **/sbin** : Essential system binaries.
- /**srv** : Data for services provided by this system
- **/tmp** : Temporary files.
- **/usr** : Secondary hierarchy.
- **/var** : Contains variable data that changes frequently during the systems operation.

For example, log files and caching data.

- **/root** : Home directory for superuser.
- **/home** : User home directories.
- **/proc** : This is a virtual filesystem in which process and kernel information are kept. In the case of the GNU/Linux operating system, the filesystem, procfs is mounted here.

4.3.5 Importance of Linux based Operating Systems

As we know, there are many operating systems which are available in the market which can be easily deployed on computers. Some of these operating systems like the Microsoft Windows and the Apple MacOS are expensive and sometimes do not have compatibility with processors popular in embedded systems such as the ARM processors present in Xilinx Zynq SoCs. In addition to this, there are limitations to personalised configurations which can be attempted with these commercial OS. Embedded versions of Windows like the Windows 10 IoT are also a paid product, which makes these commercial OS a bad choice for systems where thousands of servers and embedded processors are going to be deployed, such as the CMS TDAQ network. Due to this, Linux based Operating systems like RHEL, CentOS, Ubuntu, Fedora etc. have become popular not only for large scale server deployment but also for embedded processors. Most of the popular operating systems come with their 32-bit and 64-bit variants for ARM processors,which are very popular processor family for embedded systems. Linux based Operating Systems are mostly free in terms of monetary cost(exceptions like RHEL do exist) and users always have the benefit of community support , as the Linux community spans worldwide.Because of these two reasons, Linux has been adopted for embedded products in the worldwide public switched telephone network, global data networks, wireless cellular networks, radio note controllers and server farms. Linux is present in automobile applications, gaming consoles, printers, routers, and billions of

smartphones that have become a part of our daily lives. Linux is increasingly being seen as a primary choice for its deployment in embedded systems and even personal computing by each passing year.

There are many factors which favour adoption of GNU/Linux in sphere of personal and enterprise computing which have been described below :

- **Community Support**

The Linux kernel is an open-source software, which many communities and users can modify as per their use and share their knowledge with others. This results in a high-quality, motivated support from anywhere across the world through various forums at any given time of the day. One can also participate in this community and help fix bugs, develop patches and driver for the kernel etc.

- **Hardware Support**

As the linux kernel can be modified by users and enterprises to suit their own needs, this has resulted in the linux kernel supporting various hardware platforms and various Linux distributions now can be deployed on various computers and embedded systems irrespective of the processor they are running. A prime example of this is the ArchLinux distribution, which support around 35 different platforms running processors from ARMv5 to ARMv8[20].

- **Open-Source Software Packages**

As Linux is open-source and its use involves no monetary cost, the same can be said for most of its software packages. Many commercial and proprietary software which cost money for their installation and usage on operating systems like Windows and macOS, often open-source, free or cheaper alternatives which can be operated on Linux based Operating systems. An example of this is 'octave' which can be used instead of MATLAB or LibreOffice instead of MS-Office. However, this is not the case with niche software like Vivado for which license must be obtained, irrespective of the OS.

- **Customizability**

With Linux, Developers and users have the option of making modifications to the kernel and the root filesystem to create an operating system experience which fulfils the needs for their computer or embedded system. This is especially true in the case of embedded systems, where users can modify, the linux kernel to a minimalist kernel, by removing excessive software packages and drivers while keeping only the necessary ones. This kind of flexibility is absent from other proprietary operating systems like Microsoft Windows[21].

- **Code Re-usage**

The code written for one hardware platform can be reused for another platform as long as both the hardware platforms are running the linux-kernel, the root-filesystem has the correct libraries needed by the code and the kernel has the driver support which is required to execute the code on the hardware.

- **Scalability**

Linux can not only be run on isolated embedded devices and smartphones, but also on large scale processor farms, server farms, switches for complex cellular, broadband networks and optical networks. This property of Linux allows it to be deployed seamlessly in networks consisting of thousands of nodes while also ensuring a synchronous operation between these nodes without affecting the network adversely.

- **Security**

Linux comes with its own advantages of open-source security features which allow users to secure their device running Linux from malicious attacks. Linux also comes up with regular updates and security patches which thwart latest security threats as they arise, thanks to the involvement of the open-source community which actively hunts for vulnerabilities in linux-kernel security and also suggest changes or help develop bug fixes for the same.

- **Serviceability**

Isolated or clustered systems running Linux can be serviced thanks to support on specific issues users can receive from the Linux community, but also because of the internal diagnostics, service logs and network monitoring which are easily accessible to the users, helping users and system administrators initiate various curative and preventive measures which can be deployed with or without minor modifications simultaneously to various nodes running Linux in the network.

4.3.6 Free versus Freedom

The Linux General Public License (GPL) exists to guarantee “free as in freedom” [18][19][22][25] of a particular body of software unlike “free as in free software”. It allows the user the freedom to use it, study it, and change it. The same freedoms apply to anyone to whom you distribute your modified code. Linux based OS and kernel can be downloaded and installed in a few minutes, free of cost. However, certain costs are associated with any software to be incorporated into a design. These include the costs of integration, modification, maintenance, and support. Additional costs also apply for obtaining and maintaining a properly configured toolchain, libraries, application programs, and specialized cross-development tools compatible with hardware of your choice. Thus, it becomes evident that developing the needed software components and development environment necessary to deploy your embedded Linux-based system is not a trivial exercise.

4.4 The Need for “Embedded” Linux

Many embedded processors, if not all of them, are capable of running a full fledged general purpose Linux based OS. The examples of such devices include the popular Raspberry Pi, the Xilinx Zynq Ultrascale+ SoC and various Snapdragon IoT devices. Smartphones also contain embedded processors running the Android, which is also a Linux variant. However, running a full fledged General Purpose OS (GPOS) on embedded processors, is not always necessary and also not efficient in terms of memory and power management, especially if the processor is intended for a specific purpose and has constraints in terms of power, memory and processing speed.

In reality, even a GPOS deployed on an embedded system can be considered an “Embedded” Linux, by virtue of serving as an operating system for the embedded device. However, it is the minimalist, task-specific and resource-efficient version of the Linux distributions which is called Embedded Linux in the scientific community. Devices like Raspberry Pi, Xilinx Zynq Ultrascale+ can serve as personal computers and used to browse the Internet or write documents, when a Screen, Mouse and Keyboard are plugged into them. However, the purpose of these devices is not to serve as personal computers, but to be deployed in complex systems for solving specific challenges which personal computers and laptops cannot solve due to their power consumption, size and monetary cost involved.

Because of this very reason, Linux based GPOS are not used “as-it-is” in embedded systems and need to be configured as per the hardware, to ensure that the resources of the embedded system are judiciously used. Many GPOS like Ubuntu, CentOS, Archlinux, Fedora etc. come with their specific versions for specific hardware to ensure that their kernel is not only compatible with the hardware but also to allow users to create and design the embedded operating system as per their needs and requirements. While most of these embedded distributions of Linux based GPOS do provide users with the possibility of activating all possible features and drivers supported by the hardware, it is generally observed that only the minimal necessary configuration is activated and any modifications are incorporated on the go or during servicing and maintenance, as deemed fit by the system administrators.

The aim of an embedded system is to perform a set of specific tasks consistently and successfully with high accuracy, reliability and availability. Because of this reason embedded processor, deployed in an automobile, a drone, a switching network or a data acquisition like that at CERN needs only a fixed set of drivers and features activated in kernel along with a fixed set of software packages and libraries to perform its tasks properly. This also ensures that the limited resources which the embedded processors have in terms of memory and power are efficiently utilised. A full fledged Linux kernel with superfluous kernel is not only too big in its size for non-volatile memory, but also consumes a lot of volatile memory along with draining the power of the embedded system, unless the system is connected to a grid. Even then a full size kernel could result in high power consumption and high cost, something which is a matter of concern for organisations like CERN. Hence, the need for a minimalist “Embedded” Linux is felt for Embedded Systems. Various features of Embedded Linux in context of ARM processors are discussed in the following chapters.

5. Linux for Embedded ARM Processors

The Xilinx Zynq 7000 SoC and the Zynq Ultrascale+ SoC FPGAs are the primary focus of this project where Linux distributions would be designed and tested for them. These FPGAs contain on chip ARM processors on which the Embedded Linux operating system is deployed. While the fundamentals of Linux do not change when it is deployed on an ARM processor, there are certain special features of Embedded Linux for ARM processors which need to be described in detail. In the last chapter, the fundamentals of Embedded computing along with GNU/Linux were detailed. In this chapter, different features of Linux for embedded ARM processors and the booting of embedded Linux of ARM processors would be explained. The details of the ARM processor architectures and the hardware features of the Zynq processor are explained in later chapters.

5.1 Linux components for ARM Processors

As described in the last chapter, the features and components of embedded Linux on ARM processors do not vary significantly from the general purpose Linux operating systems. The only significant difference arises during booting process, which is also largely governed by the processor hardware and the bootloader. The booting process for ARM devices would be investigated in this chapter in the context of the following components :

- Bootloader
- Kernel
- Device Tree
- Filesystem

5.2 Bootloader

The theory of the bootloader has been covered in the previous chapter. We also know that various bootloaders exist for various hardware. The main requirements from a bootloader are that they should be simple and customizable as per the hardware of our choice. Since the Xilinx Zynq processors use the U-Boot as second stage bootloader and also because the U-Boot is supported by a good number of processor architectures, the U-Boot boot process would be described in detail. It has been around for a long time and has a good community for support.

5.2.1 U-Boot

In the previous chapter, the functions and features of a generic GNU/Linux bootloader were explained. We know that the boot loader is like the BIOS (Basic Input/Output System) in a desktop PC or server and that the boot loader leaves when the system is booted. BIOS remains in the environment to provide low-level I/O services. Desktop Linux systems have both boot loader and BIOS. Nowadays it is GRUB2 along with the BIOS, in case of Desktop Linux systems. However in case of embedded systems which do

not use the x86_64 computers and desktop Linux distributions, GRUB2 gives way to an open-source bootloader called U-boot. It began as a PowerPC bootloader named 8xxROM , first written by Magnus Damm and the modified by Wolfgang Denk who renamed it PPCBoot. The source code was briefly forked into a product called ARMBoot by Sysgo GmbH. The name was again changed to U-Boot when the ARMBoot fork was merged back into the PPCBoot tree. U-Boot supports over 1000 different platforms and is the most popular bootloader for ARM devices[18][22][25][28]. U-Boot supports an extensive command set that allows it to boot over network, manage flash memory, define variables and scripts for booting and much more.

5.2.2 Challenges of booting

The booting of the Linux kernel on the hardware is primarily the responsibility of the bootloader. This process is associated with various challenges. Every resource required by the bootloader needs to be initialised and allocated before the use. The bootloader has little or no support from the hardware or any existing software environment to execute seamlessly. Some of its challenges associated with it are[22] :

- **DRAM Controlling**

DRAM chips cannot be directly read from or written to like other microprocessor bus resources. They require specialized hardware controllers to enable read and write cycles. DRAM data is lost if they are not constantly refreshed. The DRAM controller is responsible to configure DRAM, keep it refreshed within the manufacturer's timing specifications, and respond to the various read and write commands from the processor. The booting of the embedded system and its functionality cannot be proper if the DRAM controller and DRAM itself have not been properly initialized. This is one of the biggest challenges of the bootloader as it must initialise the memory it needs , before the memory can be used as a resource. This is very important as the bootloader must copy itself into the DRAM memory after initialising it.

- **Flash and RAM problem**

The bootloaders must be stored in non-volatile memory but often are also found to be loaded into the volatile memory or RAM. Again, the complexity arises from the level of resources available for the bootloader to rely on. In a general purpose computing system running a Linux flavour, it is relatively easy to compile a program and invoke it from nonvolatile storage. The runtime libraries, operating system, and compiler synchronously create the environment necessary to load a program from nonvolatile storage into memory and pass control to it. This infrastructure does not exist when a bootloader gains control in a resource-constrained embedded system. So in order to achieve its goals, the bootloader must create an environment for itself and move itself to the RAM.

- **Execution Context**

Another challenge faced by the bootloader is the lack of execution context. Default hardware configuration ensures that data read-write from flash memory works perfectly and it also ensures that the system clock has some default values. However, apart from this the bootloader does not know any other

thing about its environment. Most processors do not have DRAM space available at startup to store temporary boot variables or stacks needed to execute C programs. These limitations place significant challenges on the initial body of code designed to initialize the hardware. Because of this, the first task the bootloader performs is to configure enough of the hardware to enable at least some minimal amount of RAM. As a result, a stack can be allocated using a fraction of that RAM, and a proper context can be constructed to run higher-level languages such as C.

5.2.3 The Booting Sequence

We know that the bootloader has two main jobs: to initialize the system to a basic level and to load the kernel. In the following section, the booting process of a generic embedded ARM processor is described. The exact process, semantics and nomenclature of the process may vary from platform to platform, as we would see in the case of Xilinx Zynq 7000 SoC and Zynq Ultrascale+ SoC in later chapters.

5.2.3.1 Phase 1 - ROM Code

When the bootloader is located in a non-volatile memory, the boot process is much more simpler. The bootloader can be directly loaded from there. The following diagram illustrates a boot process employed in the old days[28] with the Reset vector at 0xfffffffc. The bootloader is linked so that there is a jump instruction at that location that points to the start of the bootloader code[28] :

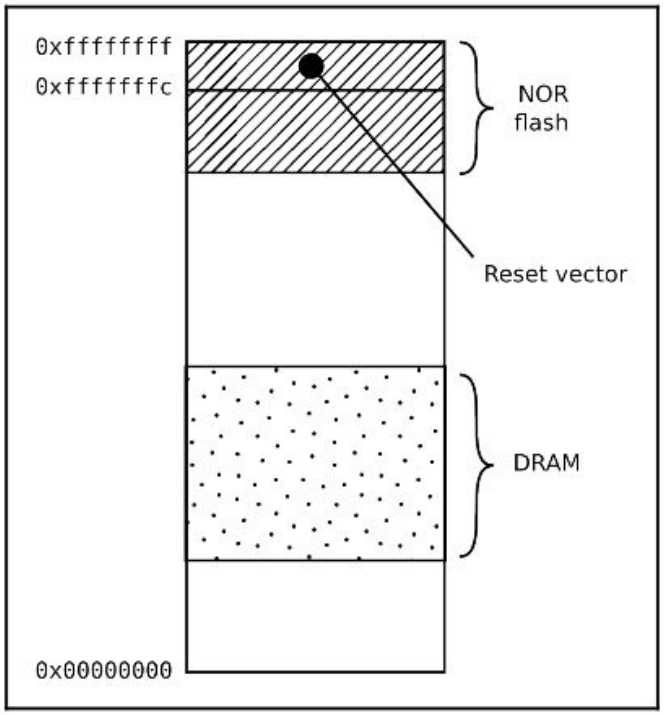

Fig.5.1 : Booting in older computers

From the point of start-up the bootloader code running in the flash memory initializes the DRAM controller, and then it copies itself into the DRAM. In absence of reliable external memory, this process is a bit more complex. This is also a challenge of the project of finding the best possible boot sequence

initialisation as loading the U-Boot from SD Card or Flash memory in the Xilinx Zynq embedded devices installed in CMS TDAQ network is not only unreliable but also a servicing hassle which needs to be solved.

In the absence of reliable external memory, the boot code can be stored on-chip in the SoC which is why it is called ROM code. It usually comes from the manufacturer but it can also be customized like we plan to do with this project. DRAM controllers are highly device-specific , whose configurations are known to the manufacturer and so it can only use Static Random Access Memory (SRAM), which does not require a memory controller.Most embedded SoC designs have a small amount of SRAM on-chip, varying in size from as little as 4 KB to several hundred KB.

The ROM code loads a small chunk of code from one of several preprogrammed locations into the SRAM. If the ROM code is not present in the flash memory, it may try to find it in the SD Card if its present or might as well try to load it over ethernet if it is configured as such. During this project, we explore booting using SD Card, JTAG and TFTP(Trivial File Transfer Protocol). Embedded SoCs can be configured for their boot modes. If the SRAM is not large enough to load a full bootloader like U-Boot, an intermediate loader called the secondary program loader or SPL is introduced as an additional stage. At the end of the ROM code phase, the SPL is present in the SRAM and the ROM code jumps to the beginning of that code.

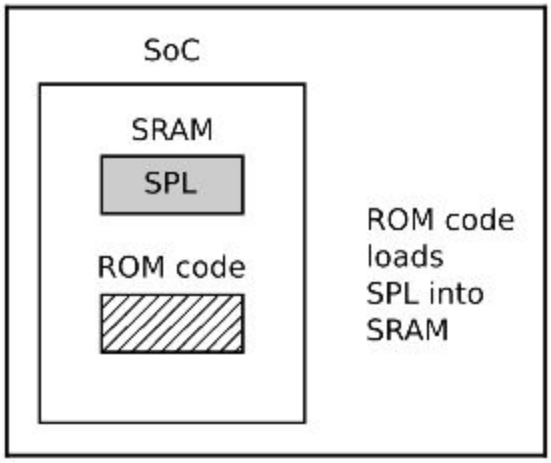

Fig.5.2 : Phase 1 of Booting Process. The ROM Code searches for the SPL from memory and loads it into the on-chip SRAM.

5.2.3.2 Phase 2 - Secondary Program Loader

The SPL initialises the DRAM controller and essential hardware of the system necessary for loading the Tertiary Program Loader (TPL) into DRAM. It can read a program from a list of storage devices similar to what the ROM Code does. It uses pre-programmed offsets in the flash device to load the TPL and also can search for essential boot files from the boot memory like the U-Boot. If the SPL has an in built file system in its compressed form, it can read bootable images and root-filesystems from its partition. The SPL also prints messages on the console.

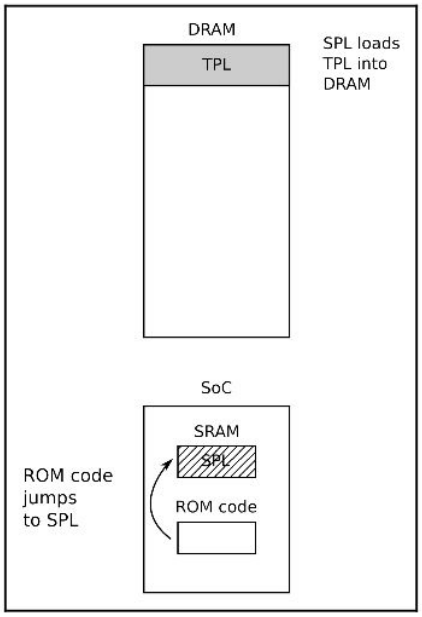

Fig.5.3 : Phase 2 of Booting Process. The SPL has been loaded into the SRAM. The SPL initialises the DRAM and essential hardware to load the TPL into the DRAM.

The SPL may or may not be open-source and could come as a proprietary code from the manufacturer in the form of a binary blob. At the end of the second phase, the TPL is present in DRAM and SPL can start operating from that location.The SPL does not allow any user interaction.

5.2.3.3 Phase 3 - Tertiary Program Loader

The TPL stage is optional. One can do the complete boot process from the SPL stage itself. However, having a TPL stage like the U-Boot helps in giving users more control over the boot process and also run boot diagnostics. After loading of the TPL we are running a full fledged U-Boot. You can enter and work in the U-Boot environment by interrupting the kernel boot process, if you have activated this feature in the U-Boot config, which we would see in chapter 8. There exists a command line interface that lets you perform different tasks such as updating the kernel image, loading the kernel into the memory, modifying the IP address, booting kernel over TFTP etc. Usually the kernel loading process happens without user intervention. The final task of the bootloader is to load the kernel into RAM and also create an environment for kernel execution. The bootloader passes a pointer to a structure containing information about the hardware configuration. The bootloader passes second pointer to the kernel command line which control how the kernel is executed. Once the kernel starts executing, the bootloader is no longer needed and the memory occupied by it can be used for kernel execution if needed. As the kernel is executing, the user has little or no control over the kernel execution and must wait to access the system until the “init” process has started the system log-in daemon.

5.2.3.4 Handing over control from Bootloader to Kernel

Upon passing the control over to the kernel, the bootloader must also share the following information with it [28] :

- The device tree binary and the machine/processor name or identification number
- The CPU clock frequency and the available RAM
- The kernel command line
- The location and size of an initial RAM disk, called the initial RAM file system (initramfs), which is by far the most popular filesystem type.

The kernel command line controls the behavior of Linux and allows it to check the status of the hardware, verifying activated drivers' functionality and the location of the filesystem. Usually the root filesystem used is called "initramfs" , which allows the kernel to userspace, which it otherwise cannot have by itself. The bootloader is responsible to load the initramfs into the memory at an early stage of the boot process. As mentioned previously, the "init" process is the first user space process initiated by kernel. All processes running on the system are but a result of the init process which initialises the system after the loading of the kernel into the memory is over.

5.3 Embedded Linux Kernel

As discussed in the last chapter, even a general purpose Linux deployed on an embedded system can be classified as an "Embedded" Linux. However, as explained previously, using general purpose Linux for embedded devices introduces a lot of overhead in terms of memory, processing speed and power consumption, which has promoted the use of Embedded Linux distributions, which are embedded processor specific and are equipped with only the most necessary features required to execute the specific task set assigned to embedded systems.

Because of these reasons, there are no special features associated with an "Embedded" Linux kernel. It is true that the kernel loading and booting process may vary from processor to processor, but the fundamentals of the Linux kernel do not change significantly. The Linux kernel has to be adapted to the embedded hardware on which it is intended to be deployed. Linux kernel releases being open-source, various hardware manufacturers and members of the Linux community modify and improve the kernel as per their requirements, to suit the hardware of the choice. They configure the kernel to add and remove the features which they need, as we would see in upcoming chapters. For example, the Xilinx Linux kernel we use for this project is based on the Linux kernel 4.14 [27] . As a result, even though special kernels exist for the purpose of embedded computing, their underlying principles remain the same as Linux kernel used for general computing purposes.

5.4 The Device Tree

It is a challenge to describe hardware when porting a Linux distribution to a given hardware. That is because the hardware description is scattered over the device drivers, kernel and bootloader but sometimes even more sub-module. This could result in software tailored for only one board and each board with the same processor architecture requiring a unique kernel image. Board Support package(BSP) try to address this problem by bringing all hardware related information in one file. However this file can be hundreds of MBs in size. Various platforms have many peripherals and it is difficult for the bootloader

and the kernel to identify them. The device tree proves to be useful in such a situation[22][25][28]. The text representation of a device tree is a file with the extension .dts. The location of the device tree may vary from device to device, especially while building an Embedded Linux for them. The device tree is a hierarchical data structure that describes the collection of devices and interconnecting busses of a computer system. It is organized as nodes that begin at a root represented by "/," just like the root file system. Every node has a name and consists of "properties" that are name-value pairs which may also contain "child" nodes. Values can be strings, lists of strings or one or more "cells" enclosed in angle brackets. The value can also be empty if the property conveys a Boolean value by its presence or absence. The device tree source code is compiled into a more compact form called a device tree blob(.dtb). The device tree compiler is actually part of the kernel source tree. The resulting "dtb" file is stored in the same flash partition as MLO and u-boot.img. The U-Boot has an environment variable loadfdt, that loads the "flattened" device tree blob into RAM. The "bootz" command that transfers control to the kernel has three address arguments: the address of the kernel, the address of the initial RAM disk, and the address of the device tree blob. When the kernel loads the device drivers, these drivers consult the device tree to determine the specific properties of their respective devices. The device tree is visible from u-boot using the fdt commands, with the help of which you can list the fdt, create new nodes and properties, or change the values of the property. Then once the kernel is booted, a read-only device tree is visible at /proc/ .

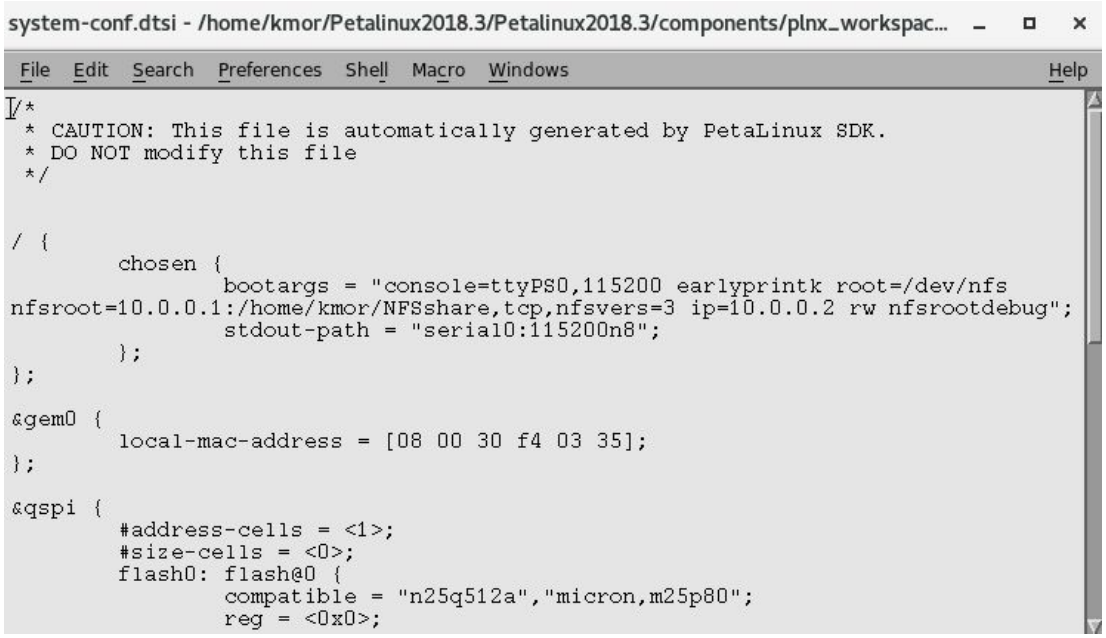

Fig.5.4 : An example of the Zynq-7000 device tree describing the kernel boot arguments, the MAC address and partitioning of the QSPI.

5.5 Root Filesystem

Similar to the Embedded Linux kernel, the embedded Linux root file-system also does not vary significantly in terms of its operations and functionalities as compared to the root file-systems found in general purpose Linux distributions. The contents of these root FS may be different for different hardware, but their hierarchy roughly remains the same. The biggest challenge for the Root Filesystem is

its storage in non-volatile memory and for that it requires file system formats which understand the underlying technology of these memory storages. A few have been listed below [] :

- **JFFS2 (Journaling Flash File System 2)**: This was the first flash filesystem for Linux, and is still in use today. It works for NOR and NAND memory, but is notoriously slow during mount.
- **YAFFS2 (Yet Another Flash File System 2)**: This is similar to JFFS2, but specifically for NAND flash memory. It was adopted by Google as the preferred raw flash file system on Android devices.
- **UBIFS (Unsorted Block Image File System)**: This works in conjunction with the UBI block driver to create a reliable flash filesystem. It works well with both NOR and NAND memory, and since it generally offers better performance than JFFS2 or YAFFS2, it should be the preferred solution for new designs.

All of these use Memory Technology Devices (MTD) as the common interface to flash memory.

However, in the project which we are doing at the CMS TDAQ, we are largely going to focus on two other popular file systems namely :

- **INITRAMFS** - A successor to init RAM Disk (initrd) which acts as an intermediate file system to load the root file system
- **NFS** - Network File System, where the filesystem is mounted over the network from a remote NFS server

While the INITRAMFS is not going to be used in the final deployment, it is the default file system supported by the Xilinx Linux kernel for Zynq 7000 and Zynq Ultrascale+ SoC to load the root file system. The NFS would be the final file system which would be used during the Linux deployment as a result it is also covered in greater detail in the following section

5.5.1 INITRAMFS

INITRAMFS is a root filesystem that the kernel has access to while booting and uses it to load the root filesystem. The only purpose of an initramfs is to mount the root filesystem[29]. The initramfs, much like any normal filesystem, is a complete set of directories that exist in the root filesystem. It is bundled into a single cpio archive and compressed using various files such as gzip or tarballs. At boot time, the boot loader loads the kernel and the initramfs image into memory and starts the kernel. The kernel checks for the presence of the initramfs in the same memory location as itself or from the location it was loaded and, if found, mounts it as / and runs /bin/init, which is the first user-space process. The init program is typically a shell script. The boot process takes longer, if an initramfs is used[29]. Initramfs is the default root-filesystem for Xilinx SoC booting process[27].

5.5.2 NFS

The Network File System (NFS) is a not exactly a root file system type but a client-server application which allows mounting file system over a network, which the user can view, update and modify remotely,

as though the file system is located on his/her own computer. It is one of the several distributed file system protocols which are used a part of network attached storage (NAS). The access to the network mounted file system by clients depends on the privileges assigned to the client user and also the restrictions associated with a file or a folder(read-only/write-restricted). One such example is that of mounting a root file system over a network and parts of which can be modified only by the root user or the superuser. NFS uses Remote Procedure Calls (RPCs) to route requests between clients and servers.

NFS was originally developed by Sun Microsystems in the 1980's and is now managed by the Internet Engineering Task Force (IETF). NFSv4.2 (RFC-7862) was ratified in November 2016 as a set of extensions to NFSv4 (RFC-3530) [31][32]. Network File System versions 2 and 3 allow the User Datagram Protocol (UDP) running over an IP network to provide stateless network connections between clients and servers, but NFSv4.2 requires use of the Transmission Control Protocol (TCP) [31][32].

5.5.2.1 Versions of NFS

NFSv2 (March 1989)

The NFS Version 2 (RFC-1094) operates over UDP. NFSv2 limits file offsets to 32-bit quantity, making maximum file size accessible by clients limited to 4.2 GB. The data transfer size limit is 8 KB, and requires that NFS servers commit data written by a client to a disk or non-volatile random-access memory (NVRAM) before responding. NFS version 2 is now obsolete [31][32].

NFSv3 (June 1995)

A stateless file system similar to NFSv2, NFS version 3 (RFC-1813) extends file offsets to 64-bits, and allows read and write transfers larger than 8KB. TCP option is added in addition to UDP. TCP transport makes it possible to use NFS over a wide area network (WAN) and enhances read and write transfer capabilities. Version 3 also added a COMMIT operation allowing for reliable asynchronous writes, and an ACCESS RPC that improves support for access control lists (ACLs) and power-users. In NFSv3, the server replies to WRITE RPCs instantly, without syncing to a disk or NVRAM, which it needed in NFSv2. To ensure that data is on stable storage, the client only need send a COMMIT RPC. NFSv3 remains the most widely used version of NFS, despite the advent of NFSv4,4.1 and 4.2 . This is the NFS version we use during this project [31][32].

5.5.2.2 NFS advantages

- NFS is a low-cost solution for network file sharing which utilises existing IT infrastructure without much overhead in terms of hardware or software.
- Central management of file system possible, particularly in the case of a complex network like the CMS TDAQ where thousands of nodes exist and managing a unique file system on individual disks for all these nodes would be a big challenge.
- NFS is user-friendly, allowing users to access files on remote hosts in the same way they access local files.

- NFS setup is fairly simple and anybody with a basic knowledge of networking and operating systems can do it.
- Reduces hardware storage requirement in the form of SD cards or external disks for remote nodes in a complex network, thereby also reducing hardware size, cost and power consumption.
- NFS is secure, so long as the file system location and the network are secure.

5.5.2.3 NFS disadvantages

- NFS is based on RPCs which are inherently insecure. Thus, they require a firewall and strong network security.
- Limiting access to unwanted users in a network containing thousands of nodes being operated by hundreds of users can be challenging.
- While using DHCP(Dynamic Host Configuration Protocol) to assign IP addresses to network nodes in a centralised manner, re-configuring NFS server and client IP addresses can be a bit challenging , especially if the server or the client shutdown and restart abruptly.

5.6 How NFS works

During initiation of NFS request, transport protocol is negotiated. The first protocol that is supported by both the client and the server is the default choice. If there is no selection in the first attempt, another transport protocol is chosen on the second attempt. The transport protocols that are supported on a system are listed in /etc/netconfig. TCP and UDP both are supported protocols for NFS.If the NFS protocol version and the transport protocol are determined by negotiation, the NFS protocol version is preferred over the choice of transport protocol. As a result, it may happen that, the NFS version 3 protocol that uses UDP is given higher precedence than the NFS version 2 protocol that uses TCP.

The file transfer size defines the size of the buffer which exchanges data between the client and the server. The NFS version 3 protocol has an unlimited transfer size, but a default buffer size of 32 KB is defined. Users can use the **-rsize** and **-wsize** options to set the transfer size manually with the mount command.

When a client needs to mount a file system from a server, it must obtain a file handle from the server that corresponds to the file system. As we would see in chapter 8, the IP address of the server can be obtained via a DHCP server that is aware of this information for a local network. This can also be a part of a pre-configured kernel boot argument or simply the user must specify the IP address of the NFS server.

The client first requests the mount port number from the portmap service running on the NFS server. Once the client receives the mount port number, that number is used to ping the service on the server. Once the client has confirmed that a service is running on that port number, the client then makes a mount request. The server responds with the file handle for the file system that is being mounted, only if the client has been verified and authorized.The default “public” file handle all the formalities mentioned above can be skipped. Now once the client is granted access by TCP or UDP wrappers, 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 access is granted, all file and directory operations are available to the user.

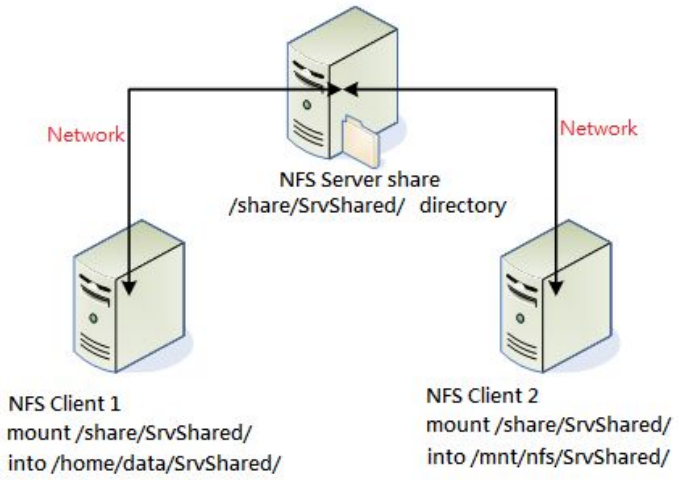

Fig. 5.5 : NFS file system mount(Source : www.qnap.com)

6. Linux Distributions for the Xilinx Zynq Series System-on-Chip FPGAs

6.1 What are Linux Distributions ?

A Linux distribution or “distro” is an operating system made from a collection of software packages,is based upon the Linux kernel and a package management system such as “apt” or “yum”. Various such distributions exist for embedded devices, personal computer, server, processor farms and network switches etc. Each Linux distribution comes with certain features that may appeal to a set of users, who are looking for an operating system that addresses one or more of their specific requirements. Linux distributions comprise of a Linux kernel, GNU tools and libraries, additional software, documentation, a graphical user interface if necessary, a window manager, and a desktop environment. Software packages in a Linux distribution may be present in form of pre-compiled binaries or also along with their source-code to allow users to make modifications. A Linux distribution usually tries to augment the Linux kernel with different software packages, libraries, tools, security and networking features etc. to service users and communities having special computing demands such as the Kali Linux which focuses on security, networking and facilitating digital forensics. Software is usually packaged for specific distributions and these packages are made available to users either by the distribution managers or sometimes even by the extended distribution using community. The software packages and distributions may be distributed around the world through repositories and mirrors from where users may download stable releases of these software. Almost six hundred Linux distributions exist, with close to five hundred out of those in active development. Many commercial and non-commercial Linux distributions like RedHat, Ubuntu, Fedora, CentOS and Archlinux have attained massive following and have their own user communities across the world. Most distributions come pre-compiled, pre-packaged for a specific

hardware architecture instruction set, while some distributions are distributed mostly in source code form and compiled locally during installation.

6.2 Linux Distribution Examples

As mentioned previously, various distributions catering to specific needs are available for installation. Some of these distributions by their usage are listed below :

- General purpose : Ubuntu , Fedora
- Routers : OpenWrt
- Internet of things : Ubuntu Core, sometimes Raspbian
- Education : examples are Edubuntu and Karoshi, server systems based on PCLinuxOS
- Scientific computer servers and workstations : Scientific Linux, BunsenLabs Linux
- Computer Security, digital forensics and penetration testing : Kali Linux and Parrot Security OS
- Privacy and anonymity : Tails
- Offline use : for example, Endless OS
- Embedded System : Archlinux, CentOS

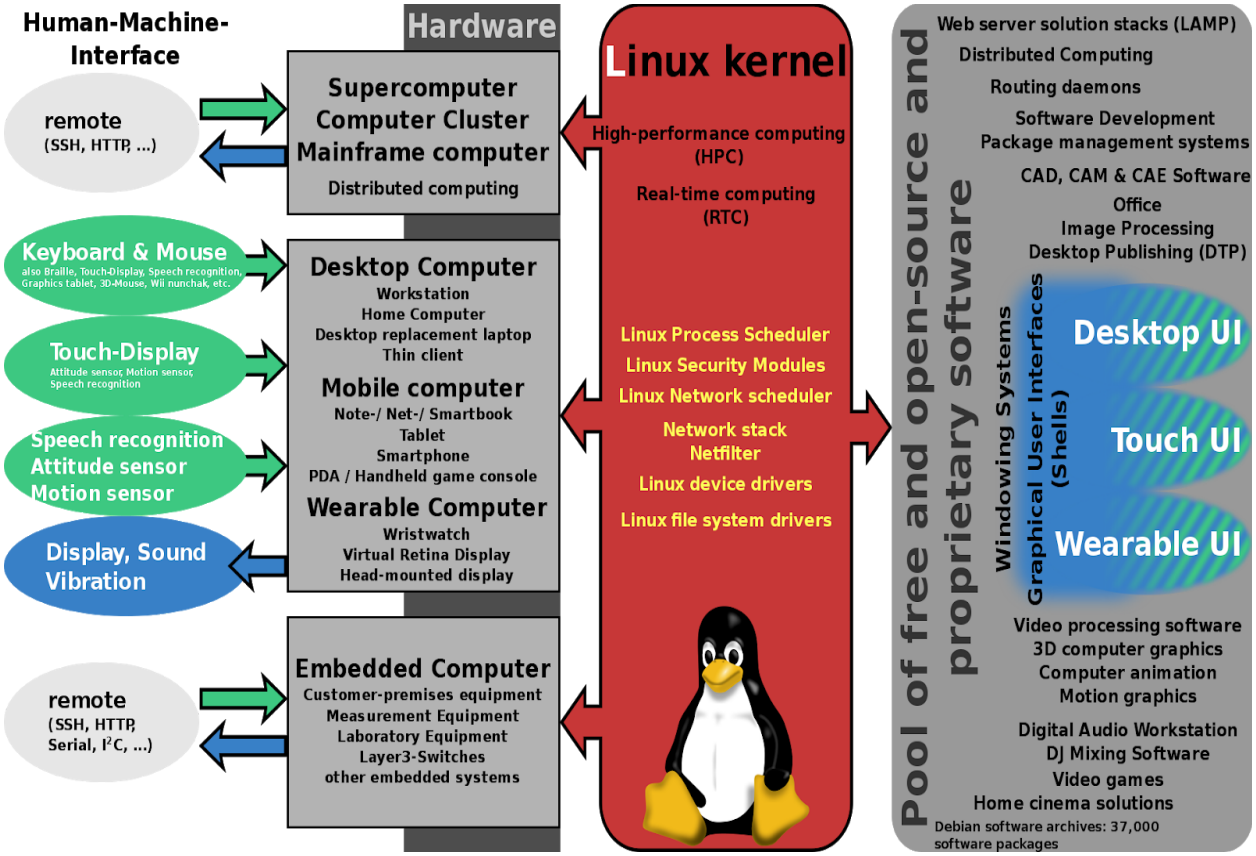

Fig.6.1 : The concept of a Linux Distribution

6.3 Linux Distributions for the CMS TDAQ Embedded Computers

The Compact Muon Solenoid (CMS) experiment group at CERN consists on many experiment sub-groups. Each of these sub-groups focus on specific research areas in the field of nuclear physics. As a result, these sub-groups utilise the CMS experiment infrastructure and the CMS detector Trigger and Data Acquisition system, to derive results for their research and prove or disprove certain postulations. Naturally, this endeavour is made possible by the specific hardware installed by these research groups in the Data Acquisition Network. These custom hardware are designed to acquire specific data which these sub-groups desire. As there is non-uniformity in the hardware and also sometimes in the software or operating system running on these hardware, it is becoming increasingly difficult to manage the network, which can have hundred thousands of network links and thousands of such boards. System administrators would find it even more challenging to manage the network and the hardware after the CMS Phase-2 upgrade as the detector and the data acquisition system would be completely upgraded. More data would be generated, at a higher throughput and many more network links would be added in the network. In such a scenario, system administrators have proposed that it would be easier to manage and configure the network if most, if not all, customised boards are running a uniform Linux distribution. This will not only help in system administration, but also give users and experiment sub-groups the possibility to access most of these boards through the centralised IT infrastructure. Another reason behind this endeavour is the fact that most experiment groups are now moving towards next-generation FPGA SoCs like the Xilinx Zynq FPGA SoC series.

Keeping this objective in focus, the Linux distribution must have some desirable aspects which favours its adoption in the CMS DAQ network. Some of these qualities are listed below :

- **Ease of use** : Most people at CERN use one or the other Linux distribution. As a result, the experienced and beginners should find it easy to use the distribution
- **Ease of development** : The development process of the customised distribution should be easy to follow for someone who has not previously developed distributions and he/she can develop the distribution by following certain guidelines.
- **Ease of deployment** : The deployment of the distribution should not be a logistical challenge
- **Security :** The distribution should have various security related features which prevent malicious penetration into the network
- **Networking Support :** Networking is key to this project and to manage the complex data acquisition network existing at CMS. The distribution should support various networking features like SSH, TFTP, NFS , DHCP, DNS etc. in order to ensure smooth booting, maintenance and administration
- **Light Distribution :** Often many distributions come with “fat” kernels that support too many features, demanding more space than necessary and also consuming power and RAM, by running background processes. At CMS DAQ, we require a light distribution where it is possible and convenient to remove superfluous features and keep only the necessary ones.

- **Run-time configurable :** The distribution should be flexible for runtime configuration and must allow enabling or disabling of services, installation or removal of packages etc.
- **Software package support :** The distribution must support diverse packages that might be required for system administration or experiment purposes
- **Reliable package manager :** The package manager of the distribution should be powerful enough to assist the user in package installation and removal. Most importantly, the distribution should have a package manager.
- **Community Support :** The distribution must have a substantial community support to troubleshoot everyday problems which may arise
- **Documentation :** The distribution or the kernel should have sufficient documentation to troubleshoot problems in-house without approaching its user community.

Keeping these requirements in mind, a few distributions have been shortlisted for testing purposes. A thorough study of these distributions would be done, resulting in feature based recommendations for final deployment in the CMS DAQ network. The shortlisted distributions are as follows :

- **Xilinx Linux** : Based on the Xilinx Linux kernel. Not technically a distribution by itself, it needs to be built using PetaLinux and Yocto tools.
- **CentOS** : CentOS is essentially a distribution based out of commercial releases of RedHat Linux and managed by the community. It is essentially RedHat without its cost but also without its support
- **Archlinux :** Archlinux's goal is to give full freedom to the end user and simplicity in using the distribution. It is based on the principle that the distribution should be minimal and its back-end must be transparent to the end user.

6.4 Xilinx Linux/ PetaLinux/ Yoctolinux

As described earlier, the Xilinx Linux is not a distribution by itself, but a kernel which is based on the Linux Kernel. Its latest release is based on the Linux kernel version 4.14. The primary purpose behind a Xilinx processor specific Linux kernel was to ensure that the users do not face challenges while developing a distribution for the Zynq series SoC and other Xilinx processor classes. The Xilinx Linux kernel ensures that the Linux kernel is customised as per the hardware of the Zynq 7000 and Ultrascale+ SoC and contains support and device drivers for various technologies and applications supported by these hardware.

The Xilinx Linux being a kernel, also allows for a Xilinx Linux or a Zynq Linux distribution by promoting distribution development through proprietary tool chains such as the PetaLinux Tools, which help in creating a Linux distribution from scratch just with the help of the Kernel and no other file system or software package from outside. Since this distribution coming out of the default settings of PetaLinux Tools and Xilinx Linux kernel has not been assigned any official name by Xilinx, we may choose to call it Zynq Linux, Xilinx Linux or PetaLinux. For the sake of simplicity, we would refer to this distribution as PetaLinux.

It is not the case that PetaLinux alone can help in creation of a distribution based on Xilinx Linux kernel, without any software package or file system being incorporated from outside. There is another tool chain called the Yocto Project which helps in creating the exact same “default” distribution that PetaLinux Tools create using the Xilinx Linux kernel. Yocto Project also does not need any filesystem or software package originating outside the Xilinx framework to create this distribution, except additional user software which would be needed by the user. For the sake of simplicity, we would refer to this distribution as Yoctolinux.

Fundamentally, there is little difference between PetaLinux Tools and Yocto Project. PetaLinux itself incorporates the Yocto Project and its build and make scripts to abstract the complex layers and processes which Yocto executes when generating a distribution. When PetaLinux is developing the distribution, it is the Yocto which is doing the underlying work and its scripts are being executed to configure the kernel, the root file system, the u-boot etc. The only advantage PetaLinux provides the user is the easy command line interface and set of commands which the user can use to develop the distribution, without worrying about scripting various scripts like the bitbake and the architecture scripts.

However, PetaLinux is less flexible in usage of external boot packages or file systems when using the u-boot and kernel provided by Xilinx. By hiding the complexities of the Yocto Project, it also deprives user of the flexibility of developing the distribution as per his/her own requirements and own resource or recipes. The Yocto project is relatively complex but allows the user a considerable degree of freedom in choosing the Linux components of his/her own choice.

The Xilinx Linux kernel is not designed just to create default distributions coming out of PetaLinux or Yocto Linux. It is a kernel which can be used to create various other distributions with the help of root filesystems and boot packages provided by distributions like CentOS or Archlinux. With the help of these resources, Xilinx Linux kernel can be used to develop other embedded Linux distributions.

6.5 CentOS

Fig.6.2 : The CentOS logo

CentOS stands for Community Enterprise Operating System.The CentOS Linux distribution is a stable, predictable, manageable and reproducible platform derived from the sources of Red Hat Enterprise Linux

(RHEL)[23][33]. CentOS is a community-supported distribution derived from sources freely provided to the public by Red Hat. As mentioned previously, CentOS is essentially an open-source and free distributable variation of Red Hat Linux, without its cost and also without its branding and support. CentOS is managed by a core team of dedicated developers who are supported by an active user community of system administrators, software developers and Linux enthusiasts across the work.In January 2014, Red Hat announced sponsorship for the CentOS project and encouraged the CentOS developers to develop software which can be interchangeably be used with both RHEL and CentOS.

CentOS was first released in 2004 and today, CentOS 7 is available for x86_64 architecture as well as AltArch versions for the IA-32 architecture, Power ISA, and for the ARMv7hl and AArch64 variants of the ARM architecture.

CentOS Linux does not receive technical support in any way and CentOS Linux is not Red Hat Linux or Fedora Linux as sometimes widely misinterpreted by new users. CentOS Linux is not a clone of Red Hat Enterprise Linux either and it is built from publicly available source code provided by Red Hat, whose RHEL has a completely different build system.

CentOS also plans to launch distributions to support other ARM architectures through the coordinated efforts with AltArch SIG. CentOS is widely used at CERN and embedded computers at different experiment groups such as ATLAS. CentOS is one of the most popular OS for servers and for networking purposes because of the various security and networking features it supports, most of which are open-source in nature. The CERN scientists and system administrators are also experienced users of CentOS and this is the reason why this distribution would be evaluated for this project.

6.6 Archlinux

Fig.6.3 : The Archlinux logo

Arch Linux is a distribution composed of non-free and open-source software that supports community involvement.The design approach for the Archlinux distribution follows the KISS principle ("keep it simple, stupid") and focuses on minimalism, simplicity , user flexibility and expects the user to to understand the underlying system thoroughly as it helps user to have full control. Regular updates of Archlinux are available through its rolling model where the downloadable Linux image or package is being constantly updated[34]. This rolling model present challenges in maintaining stable Linux version in a complex networked system like that of CMS TDAQ, where different embedded computers might be running a different set of software.

Arch Linux focuses on simplicity of design, to create an environment that is straightforward and easier for users to understand, rather than abstracting underlying system complexities from the user by providing graphical user interfaces. This is possible because of the use of configuration files that are arranged for quick access and editing, for which the user must understand certain basics of Linux. As a result, Archlinux is not for beginner level users but for advanced users. Much of Archlinux's powerful functionalities are utilised through command line.

Given that Archlinux is highly customisable, it also allows the user to choose the software packages which he or she may need, unlike other Linux distributions like Ubuntu or even CentOS, which come with a large chunk of additional packages and features, accessible through a graphical user interface, which the user might not necessarily need or want[34]. This also makes Archlinux a lighter distribution, a reason why it is popular for embedded applications.

Archlinux's simplicity also ensures that it makes only minor modifications in packages available from upstream developers, making a minimal configuration for these packages and not enabling services from these packages unless the user explicitly activates these services. This is extremely helpful in reducing the computing load on the system and frees up memory for critical embedded applications. Its stable releases. Through its rolling release model, the distribution always comes with the most up-to-date, stable features of the Linux kernel and software packages compatible with the hardware and the kernel configuration. This makes sure that so long as the Archlinux is regularly updated, there would be no compatibility issues in the system.

Archlinux is not the most-user friendly distribution but a user-centric one where users can customise the distribution as they like and develop solutions to problems along with the community through their git repositories , forums and Wiki pages. This user freedom is essential in project like ours where we need to customise the distribution to our specific needs.

Arch Linux ARM is a distribution of Linux for ARM computers and is now becoming widely used for embedded applications as embedded system designers want a customisable distribution with a freedom to choose packages for their applications. This simplicity, flexibility and stability of the distribution

combined with the community support is one of the reasons why we have opted to evaluate this distribution as well for the project.

7. The Xilinx Zynq 7000 and Zynq Ultrascale+ System-on-Chip FPGA platforms

The Xilinx Zynq 7000 and the Zynq Ultrascale+ System-on-Chip FPGA are the hardware platform which have become popular with many experiment sub-groups at CMS. Many other FPGA platforms, which do not yet have an ARM core are still a top choice for many other sub-groups. The choice and selection of the FPGA platform depends on the task which the sub-groups wish to focus upon. However, given the versatility of the Zynq series FPGAs due to its Vivado Design Suite, the ability to run an operating system on the on-chip ARM processor and ability to run complex software on the customised FPGA design using the ARM processor makes these FPGA SoCs stand out among a list of other equally powerful FPGAs, which lack a design suite like Vivado and even the on-chip ARM core. As a result, many experiment sub-groups would be using the Zynq series FPGAs for designing their hardware which will be installed in the CMS data acquisition infrastructure. Since many sub-groups would be using these FPGAs, it would be helpful from the system administration perspective to have these FPGAs run a uniform Linux distribution. That is the main objective of this project. As a result, it is also necessary to understand the underlying hardware, its architecture, its features and boot modes etc.

The Xilinx Zynq 7000 SoC and the Zynq Ultrascale+ SoC are going to be tested and evaluated through their evaluation boards, namely the Zynq Zedboard and ZCU-102 Evaluation Kit respectively. The reason behind using these evaluation boards is that they come with the FPGA as well as the peripherals needed to program and test various features supported by these FPGAs. This ensures relatively easy environment setup for starting the project.

The Zynq Zedboard and the ZCU-102 Evaluation kit are described in detail in the coming sections.

7.1 The Avnet Zynq Zedboard

The ZedBoard is an evaluation and development board developed by Avnet, which uses the Xilinx Zynq -7000 All Programmable SoC (AP SoC). Combining a dual Cortex-A9 Processing System (PS) with 85,000 Series-7 Programmable Logic (PL) cells, the Zynq-7000 AP SoC can be targeted for broad use in many applications. The ZedBoard has many peripherals and features making it ideal for testing applications for a wide range of projects The features provided by the ZedBoard consist of [35]:

- **Xilinx® XC7Z020-1CLG484C Zynq-7000 AP SoC**
- **Primary configuration = QSPI Flash**

- **Auxiliary configuration options**

1. **Cascaded JTAG**
2. **SD Card**

- **Memory**

1. **512 MB DDR3 (128M x 32) (RAM)**
2. **256 Mb QSPI Flash**

- **Interfaces**

1. **USB-JTAG Programming using Digilent SMT1-equivalent circuit**
2. **Accesses PL JTAG**
3. **PS JTAG pins connected through PS Pmod**
4. **10/100/1G Ethernet**
5. **USB OTG 2.0**
6. **SD Card**
7. **USB 2.0 FS USB-UART bridge**
8. **Five Digilent Pmod compatible headers (2x6) (1 PS, 4 PL)**
9. **One LPC FMC**
10. **One AMS Header**
11. **Two Reset Buttons (1 PS, 1 PL)**
12. **Seven Push Buttons (2 PS, 5 PL)**
13. **Eight dip/slide switches (PL)**
14. **Nine User LEDs (1 PS, 8 PL)**
15. **DONE LED (PL)**

- **On-board Oscillators**

1. **33.333 MHz (PS)**

2. **100 MHz (PL)**

- **Display/Audio**

1. **HDMI Output**
2. **VGA (12-bit Color)**
3. **128x32 OLED Display**
4. **Audio Line-in, Line-out, headphone, microphone**

- **Power**

1. **On/Off Switch**
2. **12V @ 5A AC/DC regulator**

- **Software**

1. **Xilinx Vivado 2018.3 Design Suite**
2. **Xilinx Eclipse SDK**

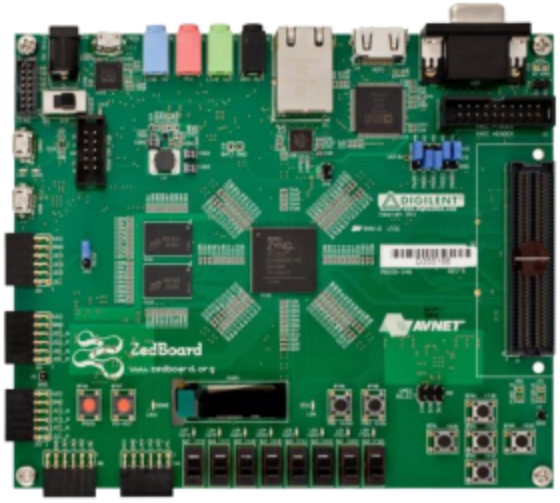

Fig.7.1 The Avnet Zynq Zedboard(Source : www.zedboard.org)

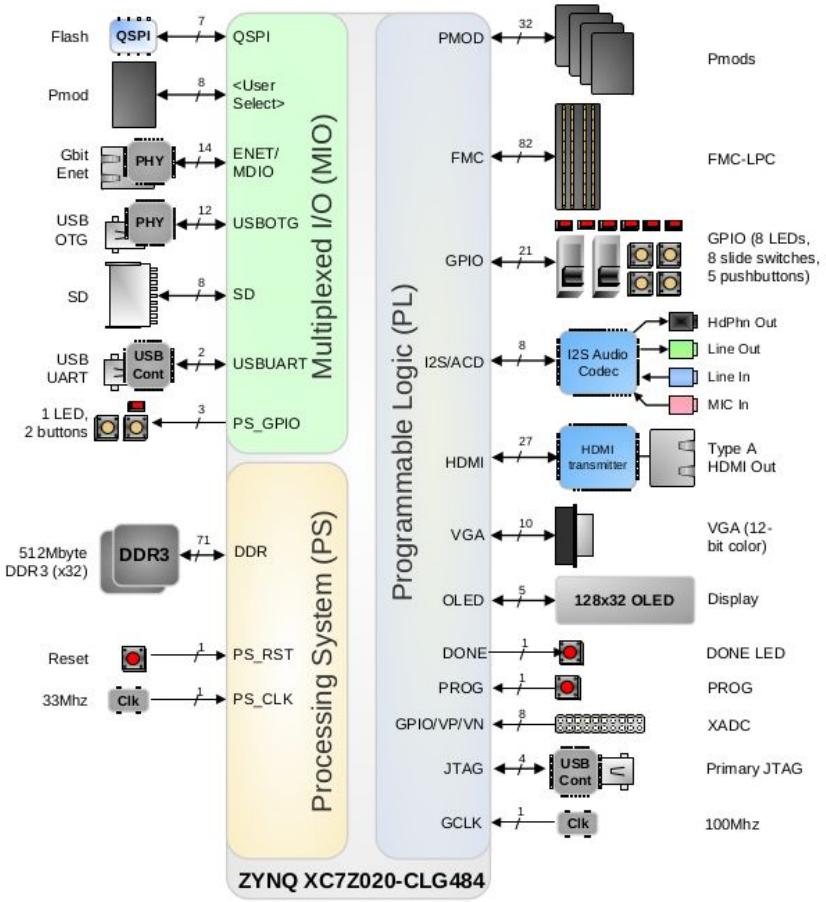

Fig.7.2 The Avnet Zynq Zedboard architecture[35]

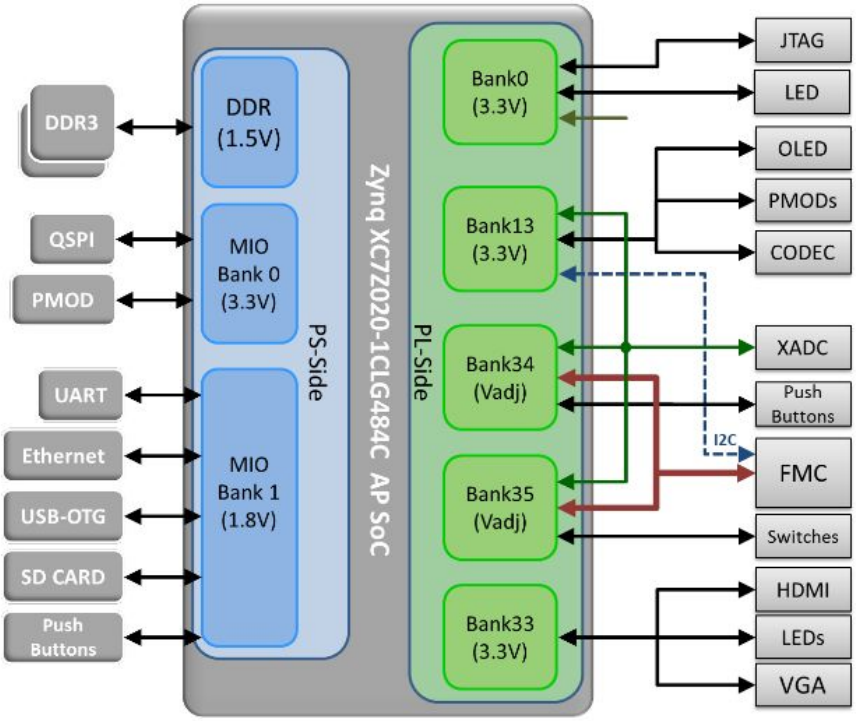

Fig.7.3 The Zynq Z7020 CLG484 Bank Assignments [35]

7.2 The Xilinx Zynq 7000 All Programmable System-on-Chip FPGA

The Zynq®-7000 family is based on the Xilinx SoC architecture. These products integrate a feature-rich dual-core or single-core ARM® CortexTM-A9 based processing system (PS) and 28 nm Xilinx programmable logic (PL) in a single device. The ARM Cortex-A9 CPUs are the heart of the PS and also include on-chip memory, external memory interfaces, and a rich set of peripheral connectivity interfaces[36].

The Zynq-7000 family offers flexibility and scalability in the FPGA design while also giving the possibility of designing a complete stand-alone embedded system due to the ARM PS and the Xilinx Vivado toolchain. It allows engineers to design cheap and effective embedded systems with the same functionality as ASICs , while giving flexibility in design. The Zynq 7000 family FPGA are extremely effective when the number of such customised FPGA are in thousands if not millions, in which case ASICs are more effective. While each device in the Zynq-7000 family contains the same PS, the PL and I/O resources vary between the devices. As a result, the Zynq-7000 and Zynq-7000S SoCs are able to serve a wide range of applications including[36] :

- Automotive driver assistance, driver information, and infotainment
- Broadcast camera
- Industrial motor control, industrial networking, and machine vision
- IP and Smart camera

- LTE radio and baseband
- Medical diagnostics and imaging
- Multifunction printers
- Video and night vision equipment

The Zynq-7000 architecture enables implementation of custom logic in the PL and custom software in the PS. Xilinx offers a large number of soft IP(Intellectual Property) core for the Zynq-7000 family. Xilinx also provides HAL(Hardware Abstraction Layer API) device drivers for bare metal applications and also allows Linux drivers for peripherals through its customised Linux kernel. With the help of the ARM PS running a Linux distribution, many third party software like OpenCV etc. can be run on the Zynq 7000 series FPGA. Other operating systems compatible with ARM-A9 processor can also be operated on the Zynq-7000 SoC. The processors in the PS always boot first, allowing a software centric approach for PL configuration as the PL configuration is managed by software running on the CPU.

Programmable Logic (PL)

Configurable Logic Blocks (CLB)

- Look-up tables (LUT)
- Flip-flops
- Cascadeable adders

36 Kb Block RAM

- True Dual-Port
- Up to 72 bits wide
- Configurable as dual 18 Kb block RAM

DSP Blocks

- 18 x 25 signed multiply
- 48-bit adder/accumulator
- 25-bit pre-adder

Programmable I/O Blocks

- Supports LVCMOS, LVDS, and SSTL
- 1.2V to 3.3V I/O
- Programmable I/O delay and SerDes

JTAG Boundary-Scan

- IEEE Std 1149.1 Compatible Test Interface

PCI Express® Block

- Supports Root complex and End Point configurations
- Supports up to Gen2 speeds
- Supports up to 8 lanes

Serial Transceivers

- Up to 16 receivers and transmitters
- Supports up to 12.5 Gb/s data rates

Two 12-Bit Analog-to-Digital Converters

- On-chip voltage and temperature sensing
- Up to 17 external differential input channels
- One million samples per second maximum conversion rate

Fig.7.4 : PL features of Zynq-7000 AP SoC[36]

Processing System (PS)

ARM Cortex-A9 Based Application Processor Unit (APU)

- 2.5 DMIPS/MHz per CPU
- CPU frequency: Up to 1 GHz
- Coherent multiprocessor support
- ARMv7-A architecture
  - TrustZone® security
  - Thumb®-2 instruction set
- Jazelle® RCT execution Environment Architecture
- NEON™ media-processing engine
- Single and double precision Vector Floating Point Unit (VFPU)
- CoreSight™ and Program Trace Macrocell (PTM)
- Timer and Interrupts
  - Three watchdog timers
  - One global timer
  - Two triple-timer counters

Caches

- 32 KB Level 1 4-way set-associative instruction and data caches (independent for each CPU)
- 512 KB 8-way set-associative Level 2 cache (shared between the CPUs)
- Byte-parity support

On-Chip Memory

- On-chip boot ROM
- 256 KB on-chip RAM (OCM)
- Byte-parity support

External Memory Interfaces

- Multiprotocol dynamic memory controller
- 16-bit or 32-bit interfaces to DDR3, DDR3L, DDR2, or LPDDR2 memories
- ECC support in 16-bit mode
- 1GB of address space using single rank of 8-, 16-, or 32-bit-wide memories
- Static memory interfaces
  - 8-bit SRAM data bus with up to 64 MB support
  - Parallel NOR flash support
  - ONFI1.0 NAND flash support (1-bit ECC)
  - 1-bit SPI, 2-bit SPI, 4-bit SPI (quad-SPI), or two quad-SPI (8-bit) serial NOR flash

8-Channel DMA Controller

- Memory-to-memory, memory-to-peripheral, peripheral-to-memory, and scatter-gather transaction support

I/O Peripherals and Interfaces

- Two 10/100/1000 tri-speed Ethernet MAC peripherals with IEEE Std 802.3 and IEEE Std 1588 revision 2.0 support
  - Scatter-gather DMA capability
  - Recognition of 1588 rev. 2 PTP frames
  - GMII, RGMII, and SGMII interfaces
- Two USB 2.0 OTG peripherals, each supporting up to 12 Endpoints
  - USB 2.0 compliant device IP core
  - Supports on-the-go, high-speed, full-speed, and low-speed modes
  - Intel EHCI compliant USB host
  - 8-bit ULPI external PHY interface
- Two full CAN 2.0B compliant CAN bus interfaces
  - CAN 2.0-A and CAN 2.0-B and ISO 118981-1 standard compliant
  - External PHY interface
- Two SD/SDIO 2.0/MMC3.31 compliant controllers
- Two full-duplex SPI ports with three peripheral chip selects
- Two high-speed UARTs (up to 1 Mb/s)
- Two master and slave I2C interfaces
- GPIO with four 32-bit banks, of which up to 54 bits can be used with the PS I/O (one bank of 32b and one bank of 22b) and up to 64 bits (up to two banks of 32b) connected to the Programmable Logic
- Up to 54 flexible multiplexed I/O (MIO) for peripheral pin assignments

Interconnect

- High-bandwidth connectivity within PS and between PS and PL
- ARM AMBA® AXI based
- QoS support on critical masters for latency and bandwidth control

Fig.7.5 : PS features of Zynq-7000 AP SoC[36]

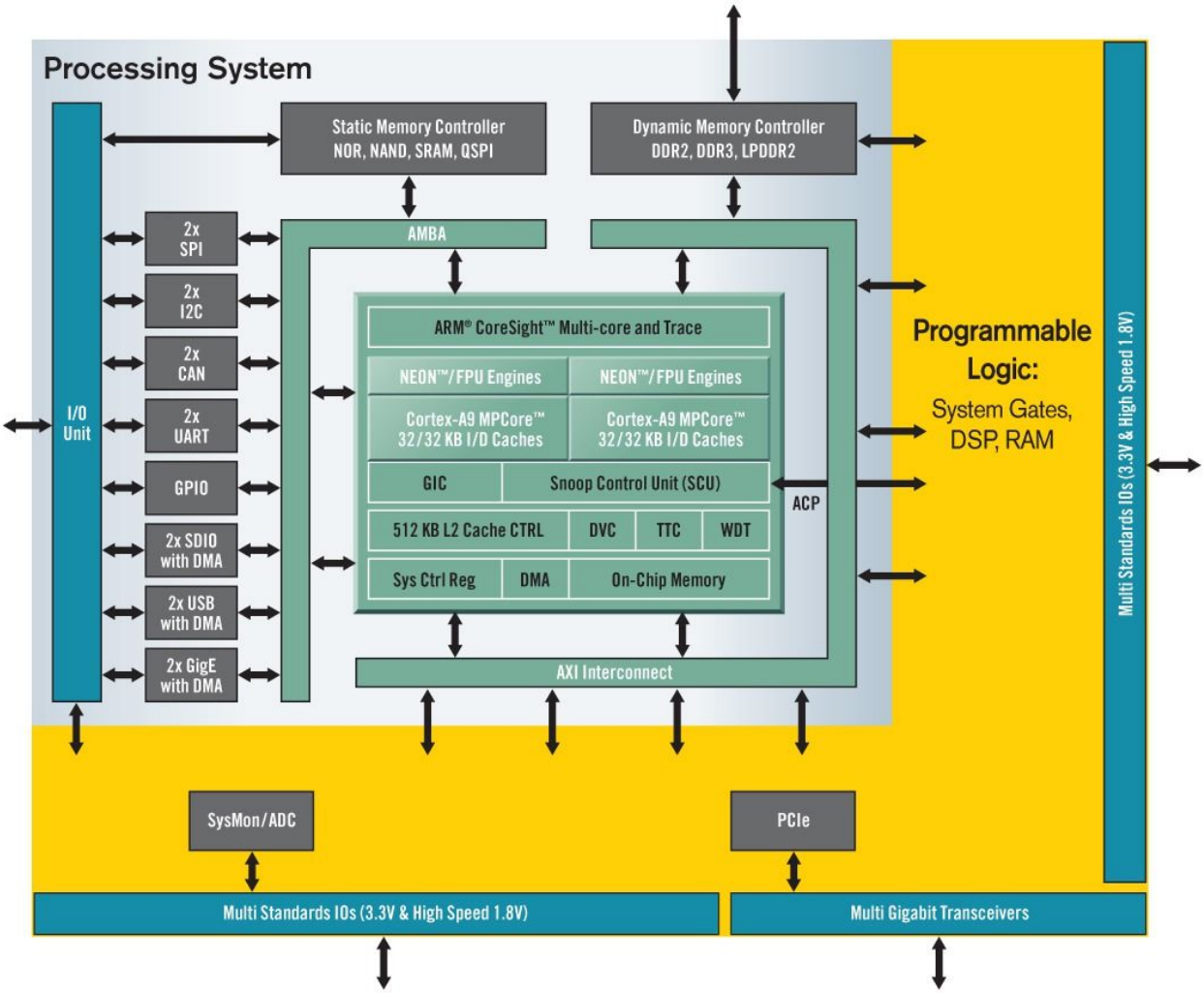

Fig.7.6 : Architecture of the Zynq-7000 AP SoC(Source : www.core-vision.nl)

7.3 The Zynq 7000 Boot Process

The Zynq-7000 All Programmable SoC devices can be booted in secure mode using static memories only with JTAG disabled or in non-secure mode using either JTAG or static memories. For the CMS DAQ network, JTAG booting is the booting method of choice.SD Cards and Quad SPI can also be used for booting the Zynq 7000 SoC.

The Zynq PS boots in two separate stages [37]:

- In stage 1 , an internal BootROM stores the stage-0 boot code, which configures one of the ARM ® processors and the necessary peripherals to start fetching the First Stage Bootloader (FSBL) boot code from one of the boot devices or through JTAG if the booting is over JTAG.
- BootROM code copies the FSBL boot code from the chosen flash memory to on-chip memory (OCM), which is the PS RAM(Random Access Memory). The size of the FSBL loaded into

OCM is compressed to 192 kilobyte. It uncomprosses to 256 kilobyte is available after the FSBL begins executing.

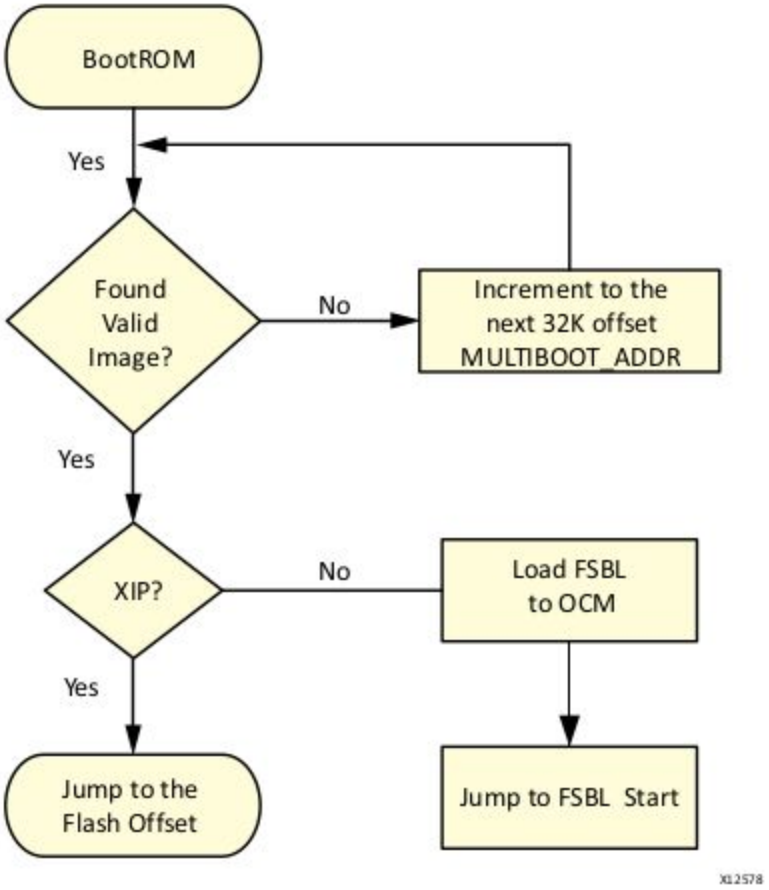

Fig.7.7 : Phase 1 BootROM process of Zynq-7000 [37]

7.3.1 First Stage Bootloader (Secondary Program Loader)

After the BootROM loads the First Stage Bootloader (FSBL) (referred to as secondary program loader in generic ARM terminology), the FSBL undertakes the following tasks :

- Initializes with the PS configuration data that Xilinx hardware configuration tools HDF(Hardware Description File) generated by Vivado
- Programs with PL with the bitstream generated by Vivado
- It loads the second stage bootloader(U-Boot) in to the DDR3 memory of the Zynq .
- Hands over control to the second stage bootloader or the bare-metal application
- Before handing over the control to the second stage bootloader, the FSBL invalidates the instruction cache and disables the cache and MMU, because U-Boot assumes it is disabled upon start.

The bitstream for the PL and the second stage bootloader, as well as other code and data used by the second stage bootloader and the Linux kernel are grouped into partitions in the flash image. The files used during the boot process are explained in detail in the next chapter. The FSBL traverses the partition header table to find the bitstream and second stage bootloader or bare-metal application partition located in the boot image.

The rules of the partition are important :

- The first partition must be the FSBL ELF file followed by the bitstream partition and then the application ELF.
- Bitstream is optional. FSBL does a handoff to the first application in the BIF order.

The order within the BIF file is important. Bitstream must be the partition after FSBL. Bitstream is not mandatory. The bitstream is required only if the PL must be programmed [37].

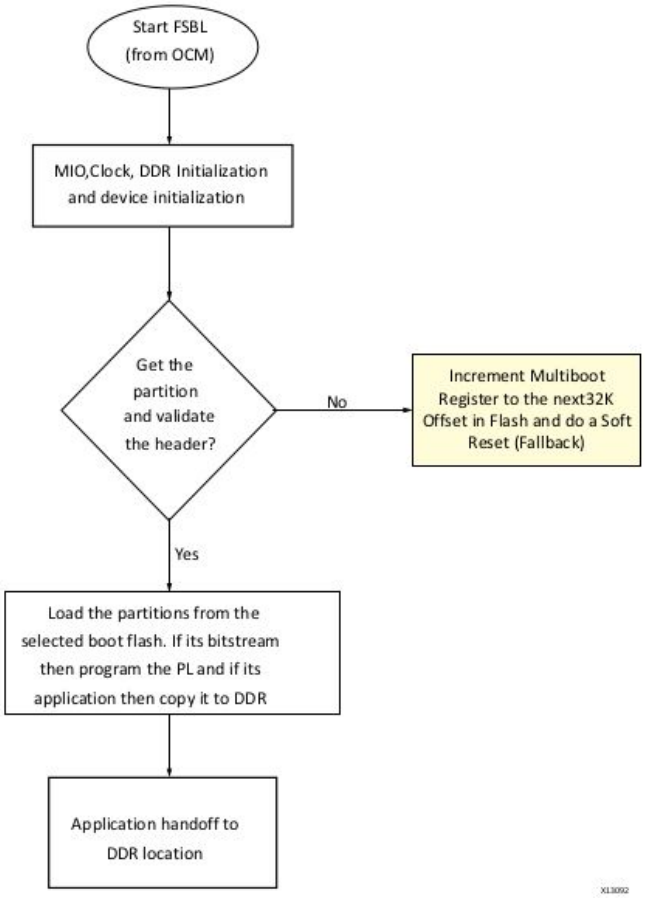

Fig.7.8 : Phase 2 FSBL sequence of Zynq-7000[37]

7.3.2 U-Boot

The U-Boot is the Second Stage Bootloader (SSBL) for the Xilinx Zynq 7000 SoC series or Tertiary Program Loader (TPL) in generic ARM boot terminology. Microprocessors can execute code that resides in processor memory, while operating systems software packages and root file system reside in large capacity permanent storage disks. As we know from previous chapters , there is no environment existing to execute process at the time of powering the processor on, as a result the U-Boot or other bootloaders are used as special software to loaded the OS kernel into memory from the media where it resides. The kernel then unpacks to create an environment for executing the first process called "init".

U-Boot is an open source bootloader that is frequently used in the Linux community, and particularly for the ARM processors. The U-Boot initialises various peripherals and memory on the PS to allow it a context to execute various commands through its command line interface, apart from loading the kernel into memory and handing over the control to the kernel. The content of a bootable SD card and the boot process are illustrated below :

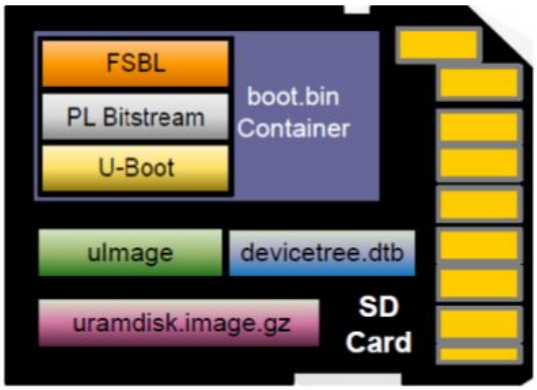

Fig.7.9 : Contents of the bootable SD card. The boot.bin file contains the FSBL, the bitstream and the U-Boot.

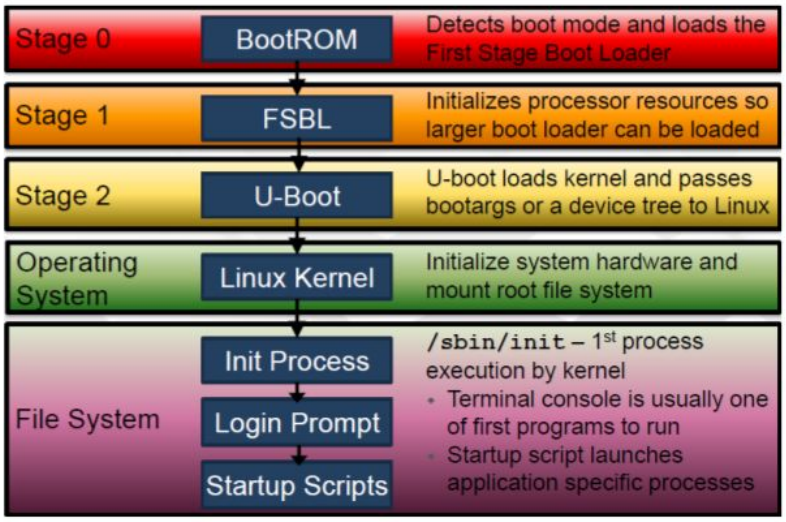

Fig.7.10 : Zynq-7000 boot process.

7.4 The Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit

The ZCU102 is a general purpose evaluation board for rapid-prototyping based on the Zynq® UltraScale+ MPSoC (multiprocessor system-on-chip). A bigger DDR3 RAM, high speed networking capacity, a graphical processing unit, a real-time processing unit, optical networking facility etc. along with other powerful features make the ZCU102 an industry standard prototyping board which has the power to be used as a general purpose computer.

The ZCU102 evaluation board features are listed below[38]:

- **XCZU9EG-2FFVB1156E MPSoC**
- **PL V CCINT for range in datasheet**
- **Form factor for PCIe Gen2x4 Host, Micro-ATX chassis footprint**
- **Configuration from QSPI**
- **Configuration from SD card**
- **Configuration over JTAG with PC4 header**
- **Configuration over JTAG with Arm 20-pin header**
- **Configuration over USB-to-JTAG Bridge**
- **Clocks (PL-system, PS_CLK, Programmable Clock, SMA, SMA_GT_REF, Ethernet, USB)**
- **PS DDR4 64-bit SODIMM**
- **PL DDR4 component (16-bit)**
- **PS GTR assignment**
- **SATA**
- **DisplayPort**
- **USB3**
- **PCIe Gen2x4 Root Port**
- **PL GTH assignment**
- **FMC #1 (8 GTH) and FMC #2 (8 GTH) PL GT assignment**
- **HDMI (3 GTH) PL GT assignment**
- **SFP+ (4 GTH) PL GT assignment**
- **SMA (1 GTH) PL GT assignment**
- **PL FMC HPC #1 Connectivity - Full LA Bus**
- **PL FMC HPC #2 Connectivity - Partial LA Bus**
- **PS MIO: QSPI**
- **PS MIO: Ethernet**
- **PS MIO: USB2 (same connector as USB3)**
- **PS MIO: SD**
- **PS MIO: CAN**
- **PS MIO: UART (using USB-to-UART bridge)**
- **PS MIO: second UART**
- **PS MIO: I2C shared across PS and PL**
- **PS/PL EMIO: Trace**
- **PL-side UART**

- **PL-side LEDs (8)**
- **PL-side DIP switch (8-position)**
- **PL-side pushbuttons (5)**
- **PS-side pushbutton (1)**
- **PS-side LED (1)**
- **System user switches (PROG, CPU Reset)**
- **PJTAG**
- **Security - PSBATT button battery backup**
- **SYSMON**
- **Operational switches (Power on/off, PROG, Boot mode)**
- **Operational status LEDs (power supply status, INIT, DONE, PG, JTAG status, DDR power good)**
- **Power Management**

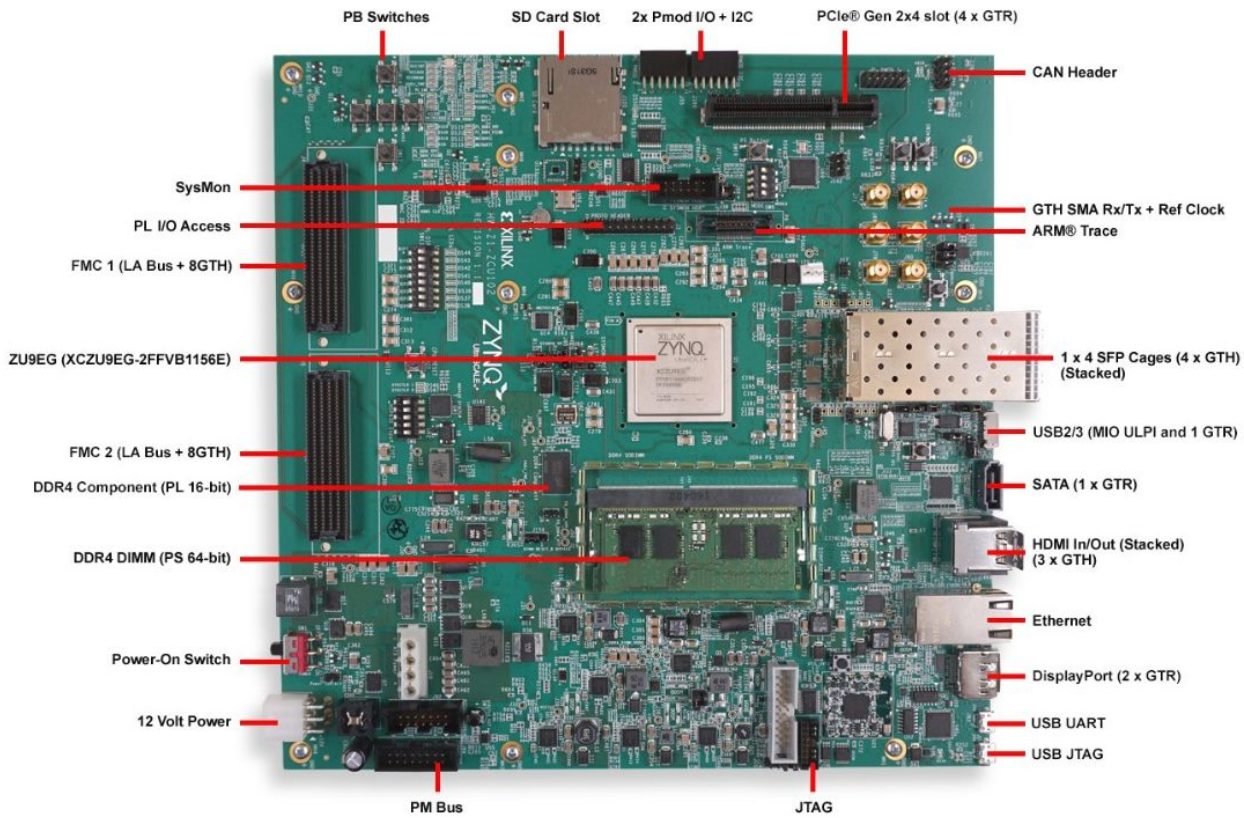

Fig.7.11 : Xilinx ZCU102 Evaluation Kit(Source : www.xilinx.com)

7.5 The Xilinx Zynq UltraScale+ MPSoC

The ZCU102 board is populated with the Zynq UltraScale+ MPSoC which is a much more powerful FPGA SoC as compared to Zynq 7000 SoC series. The processing system in the heterogeneous Zynq

UltraScale+ MPSoC has the Arm Cortex-A53 64-bit quad-core processor and Cortex-R5 dual-core real-time processor.

The Zynq® UltraScale+TM MPSoC device supports a wide range of applications that require heterogeneous multiprocessing. Heterogeneous multiprocessing system consists of multiple single and multi-core processors of differing types. It supports the following features [39] :

- Multiple levels of security
- Increased safety
- Advanced power management
- Superior processing, I/O, and memory bandwidth
- A design approach, based on heterogeneous multiprocessing presents design challenges, which includes:
- Meeting application performance requirements within a specified power envelope
- Optimizing memory access within heterogeneous multiprocessing system
- Providing low-latency, coherent communications between various processing engines
- ° Managing and optimizing system power consumption in all operational modes

The Zynq UltraScale+ MPSoC PS block has three major processing units [39]:

- Cortex-A53 application processing unit (APU)-Arm v8 architecture-based 64-bit quad-core multiprocessing CPU.
- Cortex-R5 real-time processing unit (RPU)-Arm v7 architecture-based 32-bit dual real-time processing unit with dedicated tightly coupled memory (TCM).
- Mali-400 graphics processing unit (GPU)-graphics processing unit with pixel and geometry processor and 64 KB L2 cache.
- The Zynq UltraScale+ MPSoC PS has four high-speed serial I/O (HSSIO) interfaces supporting the following protocols:
- Integrated block for PCI Express interface-PCIe base specification version 2.1 compliant.
- SATA 3.1 specification compliant interface.
- DisplayPort interface-implements a DisplayPort source-only interface with video resolution up to 4K x 2K-30 (300 MHz pixel rate).
- USB 3.0 interface-compliant to USB 3.0 specification implementing a 5 Gb/s line rate.
- Serial GMII interface-supports a 1 Gb/s SGMII interface.

The various features which the ARM Cortex A-53 APU, the ARM Cortex-R5 RPU, the GPU and the Ultrascale+ PL provide can be seen in the figures listed below :

Programmable Logic (PL)

Configurable Logic Blocks (CLB)

- Look-up tables (LUT)
- Flip-flops
- Cascadable adders

36Kb Block RAM

- True dual-port
- Up to 72 bits wide
- Configurable as dual 18Kb

UltraRAM

- 288Kb dual-port
- 72 bits wide
- Error checking and correction

DSP Blocks

- 27 x 18 signed multiply
- 48-bit adder/accumulator
- 27-bit pre-adder

Programmable I/O Blocks

- Supports LVCMOS, LVDS, and SSTL
- 1.0V to 3.3V I/O
- Programmable I/O delay and SerDes

JTAG Boundary-Scan

- IEEE Std 1149.1 Compatible Test Interface

PCI Express

- Supports Root complex and End Point configurations
- Supports up to Gen3 speeds
- Up to five integrated blocks in select devices

100G Ethernet MAC/PCS

- IEEE Std 802.3 compliant
- CAUI-10 (10x 10.3125Gb/s) or CAUI-4 (4x 25.78125Gb/s)
- RSFEC (IEEE Std 802.3bj) in CAUI-4 configuration
- Up to four integrated blocks in select devices

Interlaken

- Interlaken spec 1.2 compliant
- 64/67 encoding
- 12 x 12.5Gb/s or 6 x 25Gb/s
- Up to four integrated blocks in select devices

Video Encoder/Decoder (VCU)

- Available in EV devices
- Accessible from either PS or PL
- Simultaneous encode and decode
- H.264 and H.265 support

System Monitor in PL

- On-chip voltage and temperature sensing
- 10-bit 200KSPS ADC with up to 17 external inputs

Fig.7.12 : Some PL features of Zynq Ultrascale+ MPSoC[39]

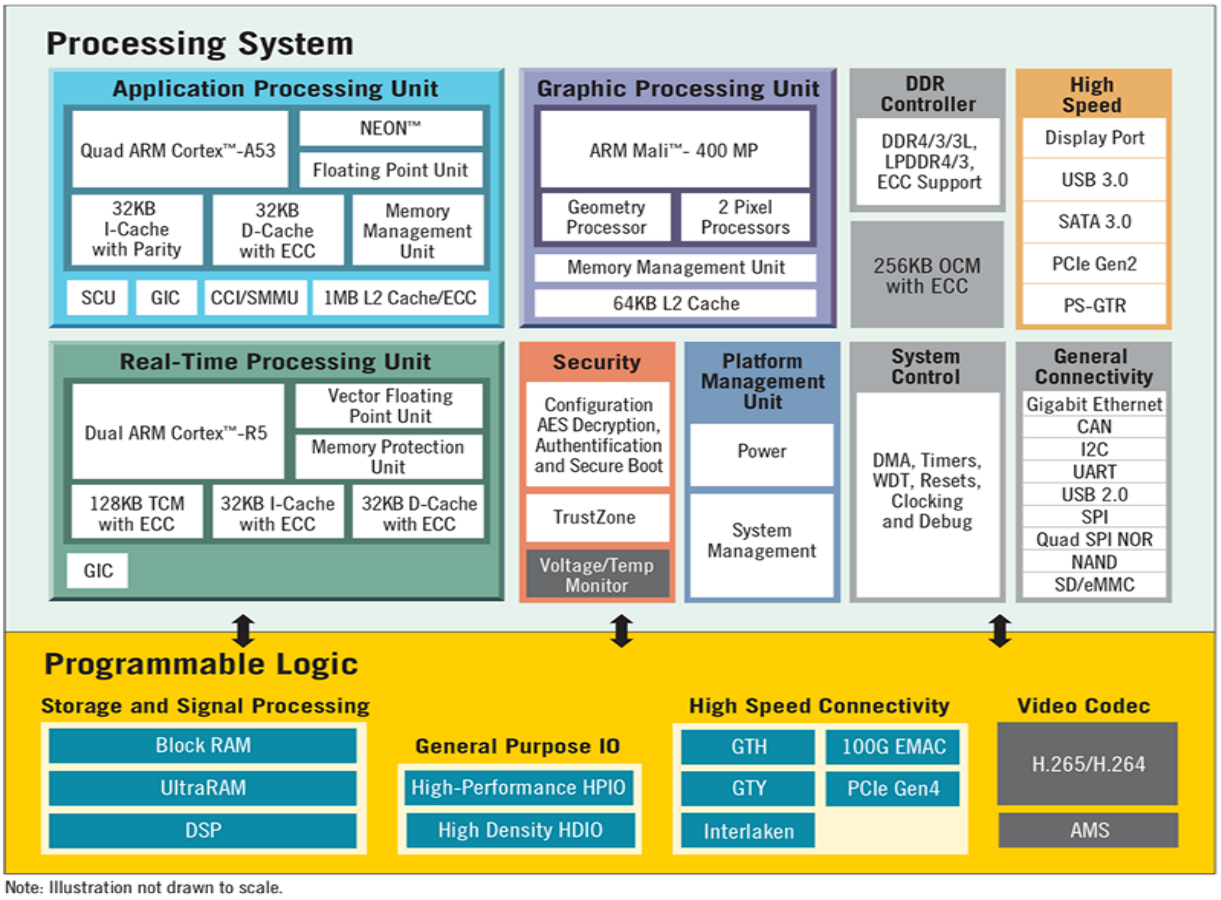

Fig.7.13 : Zynq Ultrascale+ MPSoC architecture(Source : Xilinx)

7.6 The Xilinx Zynq UltraScale+ MPSoC Boot Process

7.6.1 Boot Process Overview

The Zynq Ultrascale+ boot process is a bit different from the Zynq 7000 series SoC. While a significant part of the boot process remain identical, there are a few additional processes associated with the pre-boot process of the Zynq Ultrascale +. The platform management unit (PMU) and configuration security unit (CSU) manage and perform the multi-staged booting process. You can boot the device in either secure or nonsecure mode like in the case of the Zynq Zedboard. The boot stages are as follows [40] :

- **Pre-Configuration Stage :** The PMU primarily controls pre-configuration stage that executes PMU ROM to setup the system. The PMU handles all of the processes related to reset and wake-up.
- **Configuration Stage :** This stage is responsible for loading the first-stage boot loader (FSBL) code for the PS into the on-chip RAM (OCM). It supports both secure and non-secure boot modes. Through the boot header, you can execute FSBL on the Cortex-R5 processor or the Cortex-A53 processor. In the Cortex-R5 processor, lock-step is also supported.

- **Post-Configuration Stage:** After FSBL execution starts, the Zynq UltraScale+ MPSoC device enters the post configuration stage. After the FSBL has initialised the peripherals and setup the environment for execution of U-Boot, it loads the U-Boot into the OCM and hands over the control to the U-Boot

7.6.2 Boot Flow

As mentioned earlier, there are two boot flows associated with the Zynq Ultrascale+ : secure and non-secure boot mode. They have been explained below .

7.6.2.1 Non-Secure Boot Flow

In non-secure boot mode, the PMU releases the reset of the configuration security unit (CSU), and enters the PMU server mode where it monitors power. After releasing the CSU from its reset, the PMU loads the FSBL into OCM. FSBL can be loaded into the RPU or the Application Processing Unit (APU). PMU firmware loads into PMU RAM, which will execute from PMU RAM in parallel to FSBL in OCM. FSBL is loaded into RPU(Real-time processing Unit) and CTF. It can also be loaded in the APU .The FSBL loads the U-Boot, which in turn loads the kernel into the APU. Depending on the boot-configuration, RPU and APU may operate completely independent of each other.

- In APU, ATF(ARM Trusted firmware) will be executed after the FSBL handoff to ATF. ATF initialises the peripherals and creates an environment for a second stage boot loader like U-Boot to execute and load the kernel, such as the Linux kernel.
- In RPU, FSBL hands off to a software application, generally a bare-metal application.
- After loading the linux kernel, the root-file system is mounted and control is handed over to the user.

7.6.2.2 Secure Boot Flow

The first step of the secure boot flow is same as that of the non-secure boot flow. However after the PMU releases the CSU from reset, the CSU checks to determine if authentication is required by the FSBL or the user application. This is why this boot flow is called “secure” boot flow.
The CSU does the following:

- Performs an authentication check and proceeds only if the authentication check passes.Then checks the image for any encrypted partitions.
- If the CSU detects partitions that are encrypted, the CSU performs decryption and loads the FSBL into the OCM.Similarly, FSBL checks for authentication and encryption of each partition it tries to load. The partitions are only loaded by FSBL on successful authentication and decryption, if these partitions have been encrypted.
- In APU, FSBL hands off to ATF, which initializes the peripherals for U-Boot execution , which in turn loads the Linux kernel.

7.6.2.3 SD Boot Mode

The SD Boot Mode is the boot mode of choice for many embedded systems, especially if they are located in places that are not easily accessible or if there is no networked PC near the embedded computer to boot it over JTAG. SD Card remains a popular choice of booting embedded systems at the CMS DAQ network. The SD card boot process is described in the figure below :

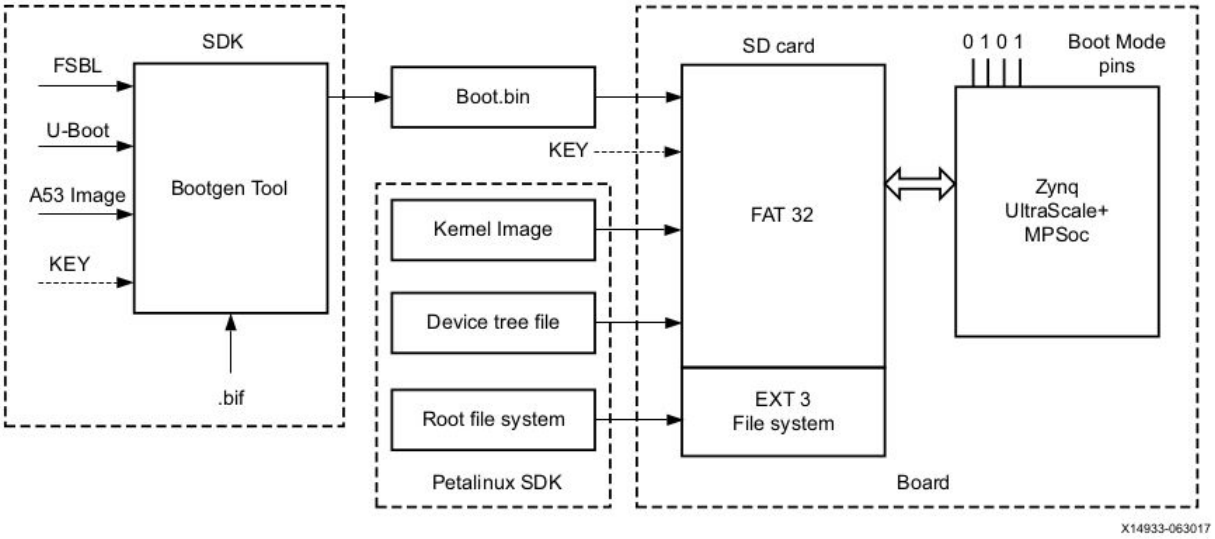

Fig.7.14 : Zynq Ultrascale+ boot in SD card mode [40]

AS mentioned previously, the PMU handles the pre-boot tasks. The PMU ROM will execute from a ROM during boot to configure a default power state for the device, initialize RAMs, and test memories and registers. It essentially acts as an extended Stage-0 BootROM process. After the PMU performs these tasks and relinquishes system control to the configuration security unit (CSU), it enters a service mode. In this mode, the PMU responds to interrupt requests made by system software through the register interface or by hardware through the dedicated I/O to perform platform management services.

As soon as the CSU reset is released, it executes the CSU bootROM and performs the following sequence :

1. Initializes the OCM.
2. Determine the boot mode by reading the boot mode register, which is determined by the pin configuration of the boot pins.
3. The CSU continues with the FSBL load and the optional PMU firmware code.
4. In APU, ATF(ARM Trusted firmware) will be executed after the FSBL handoff to ATF. ATF initialises the peripherals and creates an environment for a second stage boot loader like U-Boot to execute and load the kernel, such as the Linux kernel.
5. In RPU, FSBL hands off to a software application, generally a bare-metal application.

7.6.2.4 Zynq Ultrascale+ Post-Configuration Boot Process

The post configuration boot process of the Xilinx Zynq Ultrascale+ MPSoC is not very different from the Zynq 7000 boot process mentioned in previous sections. In addition to the PMU, the CSU, the RPU and specialised software like the ATF differentiate and sophisticate the Ultrascale+ booting process. These stages also make the boot process much more secure and advanced. However, once the pre-configuration and configuration stages are executed to load the ATF and the FSBL, the booting process from FSBL onwards in context of Linux is not different from the Zynq 7000 boot process.

Pre-Boot Task	Description
0	Initialize MicroBlaze™ processor. Capture key states.
1	Scan, and clear LPD and FPD.
2	Initialize the System Monitor.
3	Initialize the PLL used for MBIST clocks.
4	Zero out the PMU RAM.
5	Validate the PLL. Configure the MBIST clock.
6	Validate the power supply.
7	Repair FPD memory (if required).
8	Zeroize the LPD and FPD and initialize memory self-test.
9	Power-down any disabled IPs.
10	Either release CSU or enter error state.
11	Enter service mode.

Fig.7.15 : Pre-boot tasks performed by PMU [40]

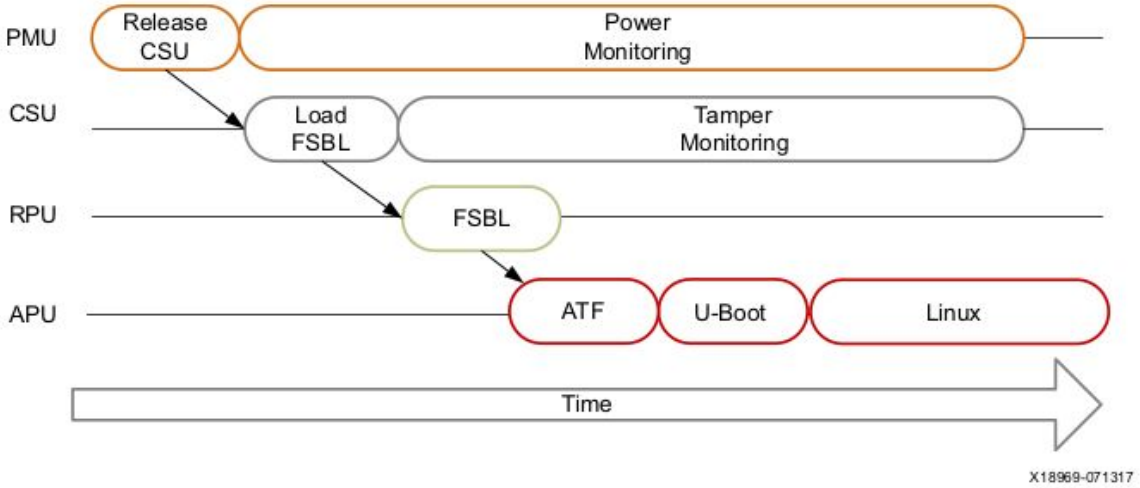

Fig.7.16 : Generic Zynq Ultrascale+ boot sequence[40]

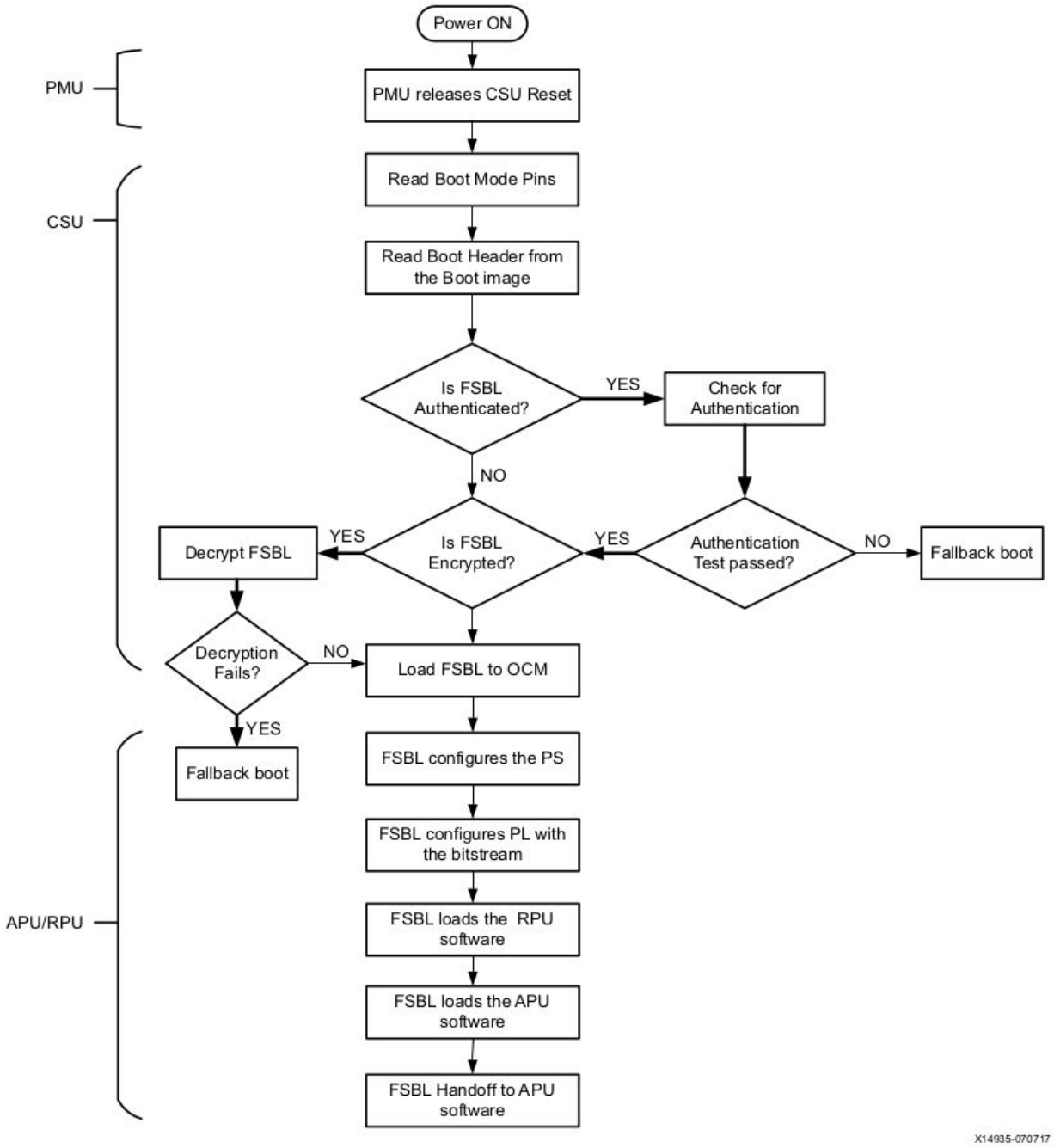

Fig.7.17 : Zynq Ultrascale+ secure boot sequence[40]

8. Linux Distribution building and Deployment on the Xilinx Zynq Series SoCs

In the previous chapters, the objectives of CERN, the LHC, the CMS experiment, the CMS Data Acquisition System and its Phase 2 upgrade have been explained. The fundamentals of the Linux kernel and the operating systems based on it have also been explained in detail. Linux for embedded systems and various distributions that would be tested during this project have also been described in previous chapters along with the hardware i.e the Xilinx Zynq 7000 SoC and Zynq Ultrascale+ SoC, that would be used as a platform to conduct various networking and system administration tests.

In this chapter, we are going to have a brief overview of the process of a Linux distribution development using PetaLinux Tools and the Xilinx Linux Kernel for the Zynq Zedboard. We would also have a look at how this distribution is deployed and booted on the hardware platform using TFTP protocol. Finally we would see how the root filesystem is mounted by the Linux distribution using the NFS protocol.

The purpose of this chapter is to illustrate the process of development and deployment of a simple Linux distribution for the Zynq Zedboard using the Xilinx Linux kernel. However, it must be remembered that given the different processes involved in building and compiling different Linux distributions, the process may vary significantly for other distributions like CentOS and Archlinux.

8.1 The Yocto Project

Even though we are using PetaLinux Tools for this demonstration, it is important to know that PetaLinux Tools is based on the Yocto Project and uses much of Yocto's basic framework, with some its own additional features to help generate distributions. Thus, it is important to focus on the Yocto Project and understands some of its basics in order to appreciate the work executed by PetaLinux Tools. It also becomes important to understand Yocto, as it would be the other tool chain with which we could develop and evaluate distributions.

The Yocto Project[41] is not a Linux distribution but an open source collaboration project that helps developers create custom Linux-based systems that are designed for embedded products irrespective of the hardware architecture in use. Yocto Project provides a flexible and powerful toolchain along with a well-defined development workflow that allows embedded device developers across the world to collaborate through shared technologies, software stacks, configurations, and best practices used to create customised Linux images.

8.1.1 Yocto Features

The following features have made Yocto very popular among engineers looking to develop customised images for their computer systems[41] :

- **Industry wide Acceptance :** Various semiconductor companies, operating system , software, processor vendors have collaborated together in the Yocto project, giving wide acceptability and ensuring Yocto compatibility with various platforms.
- **Architecture Agnostic :** As an extension to the first point, Yocto supports various architectures like Intel, ARM, MIPS etc. and can also support newer architectures so long as their Board Support package (BSP) are available. It also supports QEMU emulator for emulating the environment of newly generated Linux image.
- **Images and Code Transfer Easily :** Images and applications generated by Yocto can move across platforms without the need to use another development environment. Additionally, distributions can be supported by commercial Linux OS vendors if necessary.
- **Ideal for Constrained Embedded and IoT Devices** : Yocto helps in generating lean Linux distributions for embedded systems which helps users in choosing exactly which packages they want and which they want to remove in their customised Linux distribution.
- **Comprehensive Toolchain Capabilities :** The Yocto toolchain usually supports all sorts of features available via the hardware architecture. However, in case some feature in not supported by its tool chain, tweaking of the toolchain for specific hardware or using third party tool chains for customising a particular feature is also permitted.
- **Uses a Layer Model :** The Yocto layer based development infrastructure allows users to incrementally add or reduce features in the distribution on a layer by layer basis, allowing redundancy and flexibility to the user to customise the abstraction levels of the distribution and reduce complexity.
- **Releases According to a Strict Schedule :** Planned releases occur in October and April of each year, helping development teams which depend on Yocto releases and helps them streamline their projects.
- **Community Support :** Much like any Linux oriented project, Yocto project also benefits from an active user community from across the world which help new and advanced users with complexities on the Yocto process.

8.1.2 Yocto Challenges

Various challenges exist in using the Yocto Project. However, these challenges are miniscule compared to the power toolchain and flexibility which comes with the Yocto Project. A few challenges associated with the Yocto Project are :

- **Steep Learning Curve:** A steep learning curve is associated with the Yocto project and it requires an advanced understanding of the Yocto development process as well as the Linux based operating system concepts.
- **Confusing Project Workflow :** The Yocto layer based model, combined with the lack of graphical user interface for many of its development tools can be challenging for a non-advanced user of Linux and Yocto Project.
- **Initial Build Times Can be Significant:** Initial build times of system images can be long and can also be longer if a large number of packages and features have been incorporated. With the help

of sstate, it saves the last state of the build and every new build picks up from where the last build left off.

8.1.3 The Yocto Layers

The Yocto Project's "Layer Model" is a development model for embedded and IoT Linux creation that distinguishes the Yocto Project from other simple build systems. Layers are repositories containing set of instructions which tell the OpenEmbedded Development system what to do. These layers allow customisation, collaboration and reuse.

Layers may contain changes to previous instructions or settings at any time allowing them the override capability to customize previously supplied collaborative or community layers to suit your product requirements.You may use different layers to logically separate tasks corresponding to different layers present in a general purpose computer or combine a few layers or remove some layers altogether . Isolating information into layers, helps in future customizations, maintenance and reuse.The more modular or layered the system, the easier it is to make future customisations in the project.

You can easily identify layers that ship with a Yocto Project release in the Source Directory by their names, as layers typically have names that begin with the string meta-. For example the layer associated with the Xilinx hardware supported by Yocto, is called “meta-xilinx”, which usually contains packages to support Xilinx SoCs. Various other layers like meta-user, meta-skeleton, meta-qt5 etc. exist.

8.1.4 Open-Embedded Build System and its components

The OpenEmbedded Build System is the build system specific to the Yocto Project. It is based on another project known as "Poky", which uses BitBake to build images.

"Poky"is Yocto Project's reference distribution or Reference OS Kit. Poky contains the OpenEmbedded Build System (BitBake and OpenEmbedded-Core) as well as a set of metadata to get you started building your own default distro. It contains all basic requirements of functionalities expected from an embedded system to run a Linux image binary on it hardware. Poky is a combined repository of BitBake, OpenEmbedded-Core, meta-poky, meta-yocto-bsp, and Yocto documentation. The functionality of each of these components of Poky are listed below[41] :

- **BitBake** is a task executor and scheduler that is the heart of the OpenEmbedded build system that helps build the image binaries.
- **meta-poky**, which is Poky-specific metadata layer.
- **meta-yocto-bsp**, which are Yocto Project-specific Board Support Packages (BSPs) concerned with the hardware and the architecture.

- **OpenEmbedded-Core** (OE-Core) metadata, which includes shared configurations, global variable definitions, shared classes, packaging, and recipes. Classes define the encapsulation and inheritance of build logic. Recipes are the logical units of software and images to be built.
- Documentation, which contains the Yocto source files to create user manuals or tutorials

Every aspect of the image building is controlled by metadata and you can use metadata to augment these base image generated by the reference distribution Poky by adding metadata layers that extend functionality. These layers can provide additional software layer to execute some application, help support a new hardware or add a networking layer for the image.

Metadata can be grouped into configuration files or package recipes. A recipe, like its name suggests, is a "guidelines" or a collection of "guidelines" in the form of non-executable metadata used by BitBake to set variables or define additional build-time tasks. A recipe contains fields such as the recipe description, the recipe version, the license of the package and the upstream source repository. We can define tasks, pre-requisites and sometimes the tool or services required by Bitbake to build the image.

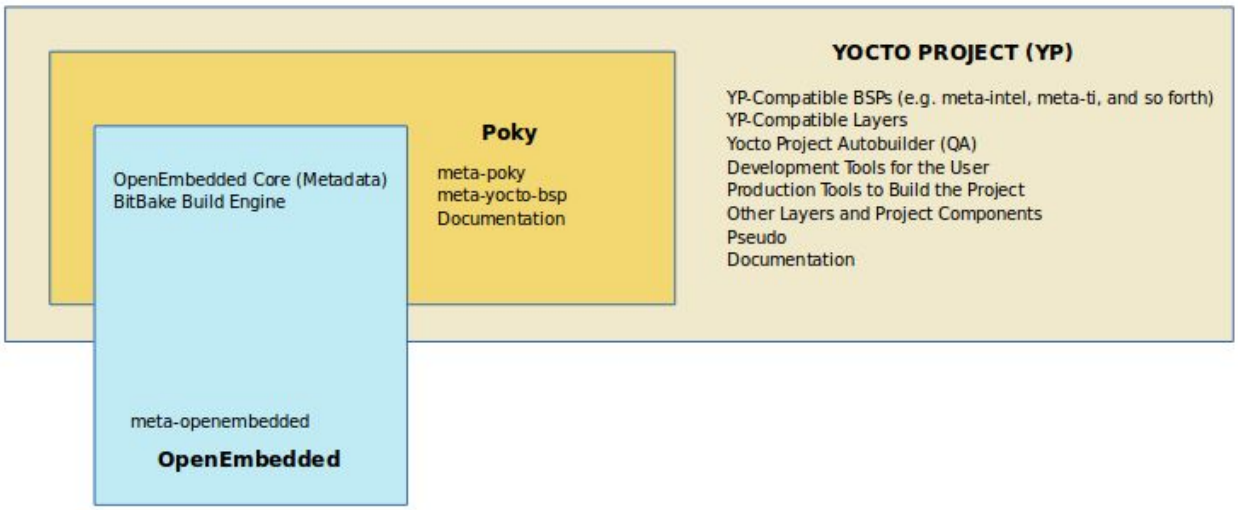

Fig.8.1 : Various layers of the Yocto Project[41]

The following list consists of components associated with the OpenEmbedded build system[41]:

- **BitBake :** BitBake is a core component of the Yocto Project and is used by the OpenEmbedded build system to build images. It is maintained separately from the Yocto Project. BitBake is a generic task execution engine that allows shell and Python tasks to be run efficiently and in parallel while it manages inter-tasks dependency constraints. BitBake is a build engine that uses the recipes provided to it, to perform specific tasks. BitBake can be imagined as a type of "kernel" managing task execution in parallel, while also ensuring that each task gets its fair share of resources needed for its execution.
- **OpenEmbedded-Core :** OpenEmbedded-Core (OE-Core) is a common layer of metadata used by OpenEmbedded-derived systems,including the Yocto Project. The Yocto Project and the

OpenEmbedded Project both maintain their OpenEmbedded-Core. Yocto Project first integrated the OpenEmbedded Core through its reference distribution Poky . After this, the Yocto Project maintenance team and the OpenEmbedded team decided to collaborate to create the common metadata layers for OpenEmbedded derived systems.

8.1.5 The OpenEmbedded Build System Workflow

The OpenEmbedded Build System workflow consists of the following functional areas [41]:

- **User Configuration** : metadata you can use to control the build process.
- **Metadata Layers** : Various layers that provide software, machine, and distro metadata.
- **Source Files** : Upstream releases, local projects, and SCMs.
- **Build System** : Processes under the control of BitBake. This block expands on how BitBake fetches source, applies patches, completes compilation, analyzes output for package generation, creates and tests packages, generates images, and generates cross-development tools.
- **Package Feeds** : Directories containing output packages (RPM, DEB or IPK), which are subsequently used in the construction of an image or Software Development Kit (SDK), produced by the build system. These feeds can also be copied and shared using a web server or other means to facilitate extending or updating existing images on devices at runtime if runtime package management is enabled.
- **Images** : Images produced by the workflow.
- **Application Development SDK** : Cross-development tools that are produced along with an image or separately with BitBake.

Following is a brief summary of the workflow [41] :

1. Developers specify architecture, policies, patches and configuration details.
2. The build system fetches and downloads the source code from the specified location. The build system supports standard methods such as tarballs or source code repositories systems such as Git.
3. Once source code is downloaded, the build system extracts the sources into a local work area where patches are applied and common steps for configuring and compiling the software are run.
4. The build system then installs the software into a temporary staging area where the binary package format you select (DEB, RPM, or IPK) is used to roll up the software.
5. Different QA and sanity checks run throughout the entire build process.
6. After the binaries are created, the build system generates a binary package feed that is used to create the final root file image.
7. The build system generates the file system image and a customized Extensible SDK (eSDK) for application development in parallel.

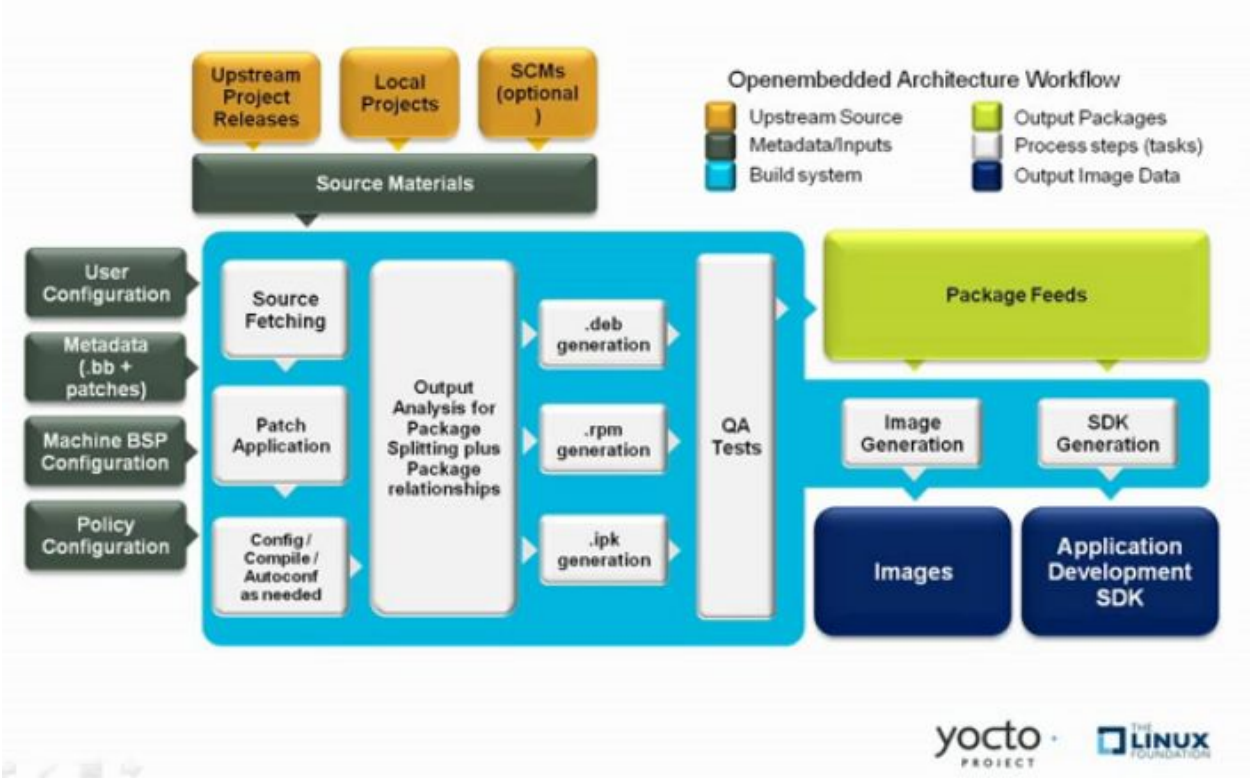

Fig.8.2 : Open-Embedded Architecture[41]

The underlying complexities of the Yocto Project can be challenging for a user looking for a quick process to develop a generic Linux distribution for his/her embedded system. Keeping this in mind, Xilinx Developed the PetaLinux Tools, which hides all the underlying complexities to generate the Linux distribution through some fixed set of commands. This process would be outlined in the following sections.

However, due to the inflexibility or complexities associated with PetaLinux for using external Root file systems and external kernels of other Linux distribution, especially with Archlinux, it is a research objective of this project to explore the Yocto Project and see how it can be used to develop customised distributions with various other kernels. Another research objective of this project is to explore how PetaLinux works with external kernel, u-boot and root filesystem and how complex is this process as compared to using the Yocto Project.

8.2 PetaLinux Tools

The PetaLinux Tools developed by Xilinx is a tool-chain based on the Yocto Project for system designers looking for a convenient distribution development mechanism, which hides the underlying complexities associated with the Yocto Project.The PetaLinux-Tools is a package which contains scripts and build environment customised for developing Linux distributions for Xilinx Zynq and Microblaze series SoC. It ensures that through some commands from the terminal, the Yocto Project is assembled an a distribution is created for the concerned hardware. Conceptually, both the Yocto Project and the PetaLinux do the

same thing as the Yocto Project is what lies underneath the PetaLinux toolchain. For the demo distribution development in following sections, we would be using the PetaLinux tool-chain.

The PetaLinux tool contains the following[27]:

- Yocto Extensible SDK

Path	Architecture
$PETALINUX/components/yocto/source/aarch64	for Zynq® UltraScale+™ MPSoC
$PETALINUX/components/yocto/source/arm	for Zynq
$PETALINUX/components/yocto/source/microblaze_full	for MicroBlaze™ full designs
$PETALINUX/components/yocto/source/microblaze_lite	for MicroBlaze lite designs

Fig.8.3 : PetaLinux Yocto extensible SDK[27]

$PetaLinux refers to the PetaLinux Tools installation guide

1. The Yocto extensible SDK (e-SDK) consists of all layers for an architecture. The Yocto e-SDk contains core, meta-oe and other popular layers.

Layer	Recipes
meta-xilinx	A collection of layers to support Xilinx products
meta-xilinx/meta-xilinx-bsp	Contains recipes of linux kernel, U-boot, Arm Trusted Firmware
meta-xilinx/meta-xilinx-contrib:	A contribution layer to support for MicroBlaze, Zynq and ZynqMP architectures. This layer depends on meta-xilinx-bsp layer
meta-xilinx-tools	Contains recipes of all embeddedsw apps: fsbl, pmu firmware, fsboot, device-tree
meta-petalinux	Contains distro recipes and package groups petalinux-image-minimal --> minimal feature set petalinux-image-full ---> Full feature set
meta-openamp	Contains openamp recipes and configurations

Fig.8.4 : PetaLinux Yocto Layers[27]

2. It consists of sstate-cache - By design, the OpenEmbedded build system builds everything from scratch unless BitBake can determine that parts do not need to be rebuilt. Fundamentally, building from scratch is attractive as it means all parts are built fresh and there is no possibility of stale data causing problems. The Yocto Project implements shared state code that supports incremental builds. It stores all task

intermediate build artifacts and reuses them if there is no change in input tasks, hence reduces the build time.

For example: The sstate-cache of Zynq UltraScale+ MPSoC is located at: $PetaLinux/components/yocto/source/aarch64/sstate-cache

3. sysroots - This contains the initial sysroot for host and the target For example: The sysroot of Zynq UltraScale+ MPSoC is at:

$PetaLinux/components/yocto/source/aarch64/sysroots

4. XSCT and tool chains -The PetaLinux tool uses XSCT underneath for all embeddedSW apps. Linux tool chain for all three architectures is from Yocto

5. 4. PetaLinux CLI tools - This contains all the PetaLinux commands that you require.

The distribution development process is the same for Zynq Zedboard and ZCU102 Evaluation Kit. For this demonstration, we would be using the Zynq Zedboard for demonstration.

8.3 Distribution Development with PetaLinux Tools

The distribution development with PetaLinux Tools requires too crucial software applications and a bunch of software packages. These are :

- Personal computer with 8GB(or more) RAM, 2 GHz (or more processing speed) and 100 GB of HDD space [27]
- 64- bit Linux Operating system : Red Hat Enterprise Linux (7.2 +). Ubuntu (16.04 LTS +) or CentOS (7.2 +) [27]
- Xilinx Vivado Design Suite (version number corresponding to the PetaLinux version number)
- PetaLinux Tools (version corresponding to Vivado Design Suite)
- TFTP server application for booting the Zedboard using TFTP protocol
- NFS server application for NFS root-filesystem mount

A few other software packages are required which would be introduced and explained in later sections.

Step 1 : Download Xilinx Vivado Design Suite

The Vivado Design SuiteHLx editions can be downloaded from the following link : https://www.xilinx.com/support/download.html . The installation process is straightforward due to a graphical user interface. One can download a 30-day trial license to evaluate the Vivado Design Suite.

Fig.8.5 : Xilinx Vivado Design Suite Logo

Step 2 : Create a minimal working and verified Hardware Design

● This is needed to have a basic design implemented in FPGA for the OS to run applications on

● Use Vivado 2018.3 or the version corresponding to the PetaLinux package version

● There are different basic designs recommended for the Zynq Ultrascale+ and Zynq 7000 series

● For Zynq Ultrascale+ following minimal design should be implemented

- External Memory with at least 64 MB of memory
- UART for serial console
- Optional non-volatile memory for Quad SPI Flash or SD/MMC
- Optional Ethernet which can be essential for network access and network booting.

● For Zynq 7000 SoC it is essential to have the following minimal design

- One Triple Timer Counter (TTC)
- External Memory with at least 32 MB of memory
- UART for serial console
- Optional non-volatile memory for Quad SPI Flash or SD/MMC
- Optional Ethernet which can be essential for network access and network booting.

● Most of these essential components are already configured when running “Run Build Automation”

● Some of these components might have to be added

● This is the most crucial stage of the PetaLinux distribution development as this design would provide the HDF to the PetaLinux package on the basis of which the PL hardware, the device tree and kernel device drivers would be defined

Step 3 : Download and install the PetaLinux Tools from Xilinx Downloads website

● This step involves downloading the PetaLinux tools installer from : https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html

● Make sure to download the PetaLinux tools Installer version corresponding with your Vivado version. There are usually 4-5 updates to both Vivado and PetaLinux in one calendar year

● You also need to install the Board Support Package (BSP) corresponding to the Zynq 7000 SoC or the Zynq Ultrascale+ MPSoC from the same weblink :

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

● The BSP is essential to building a PetaLinux project as it dictates what kind of kernel configuration would be needed for the given hardware. It essentially helps in choosing the meta-xilinx layer configuration for PetaLinux tools.

● Make sure you have 15-20 GB vacant space in your system as the installation and the subsequent project require quite a bit of space depending on the kernel configuration

● Depending on the OS, PetaLinux tools 2018.3 have certain dependencies which must be installed by looking at the Page 12 of the following weblink:

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1144-PetaLinux-tools-reference-guide.pdf

● The documentation varies and updates from version to version of Vivado and PetaLinux. In this demonstration, we have chose PetaLinux and Vivado version 2018.3 as CMS possesses the license for Vivado 2018.3. As the weblinks get updated, new versions of software and documentation may be available.

● To install PetaLinux Tools under ~/PetaLinux2018.3, execute the following commands in your home directory :

$ mkdir -p ~/PetaLinux2018.3

● Change to the installer download directory and execute the following command which will install PetaLinux tools in the PetaLinux2018.3 folder created by you :

$ ./PetaLinux-v2018.3-final-installer.run ~/PetaLinux2018.3

● The installation can be done in any directory and there are no restrictions to the directory where you can install it. However, while using the PetaLinux Tools, its environment variables must be imported from its installation directory to the directory where the PetaLinux project is intended to be created.

● Accept the EULA(End User License Agreement) and proceed with the installation

● After completion , turn of reportage of Usage statistics to Xilinx by typing PetaLinux-util --webtalk off , if you want to do it.

● Ignore the warning about the TFTP server directory, as the directory is created later during the project.

Step 4 : Set up the environment for the distribution development

● For Bash as user login shell:

$ source <path-to-installed-PetaLinux>/settings.sh

● For C shell as user login shell:

$ source <path-to-installed-PetaLinux>/settings.csh

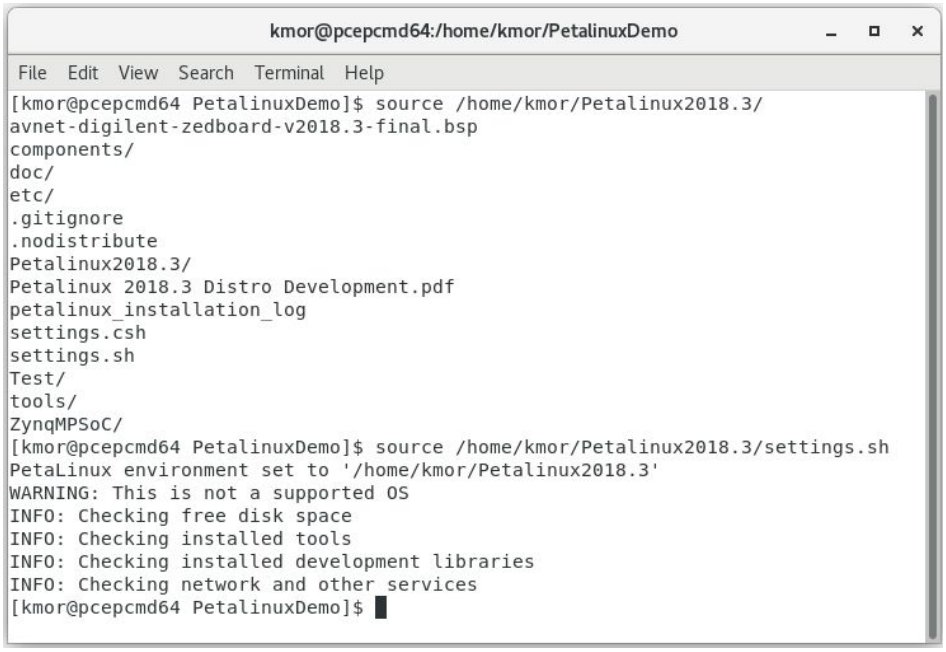

Fig.8.6 : Setting up PetaLinux environment

● Verify that the working environment has been set in your project directory :

$ echo $PetaLinux

● If you get a warning related to “/bin/sh is not bash” that means your default shell is linked to dash

● Other problems can be troubleshooted looking at Page 18 of this guide : https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1144-PetaLinux-tools-reference-guide.pdf(Documentation for PetaLinux 2018.3)

● Now you are ready to create the PetaLinux project

Step 5 : Creating a PetaLinux project for the Zynq 7000 and Zynq Ultrascale SoC

● Go to the directory where you want to store the project (preferably the installation

directory) and type in :

$ PetaLinux-create --type project -s <path-to-bsp> --name <PROJECT NAME>

OR **$ PetaLinux-create --type project --template <PLATFORM> --name <PROJECT NAME>**

● Where <path-to-bsp> is the path to the directory where the Board Support Package in installed

● <PLATFORM> is either zynq or zynqMP for Zynq 7000 or Ultrascale+ respectively

● <PROJECT NAME> as you desire

● --template option is the alternative to using BSP

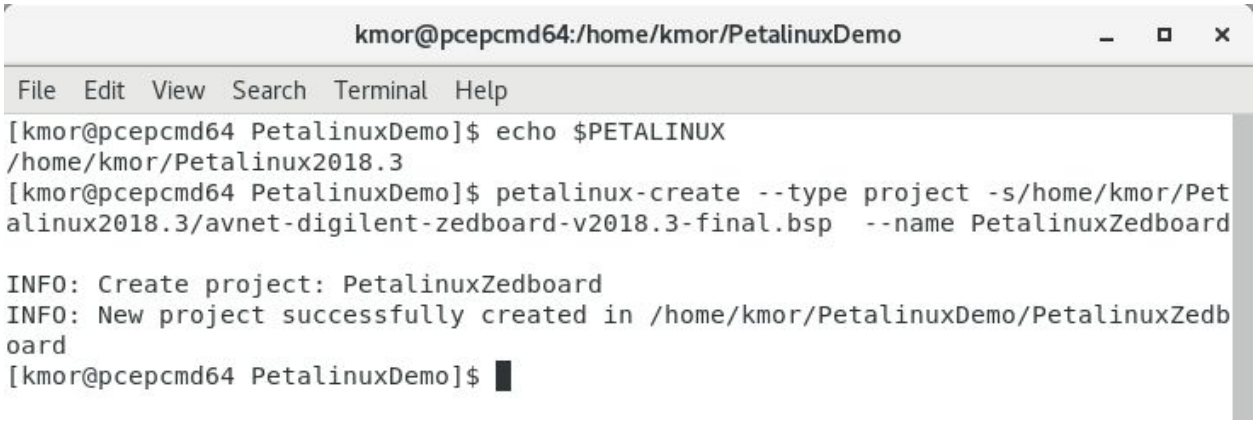

Fig.8.7 : PetaLinux project creation

● Configure the project as per the hardware designed in Step 1 by typing :

$ PetaLinux-config --get-hw-description = <PATH-TO-HDF/DSA-DIRECTORY>

● <PATH-TO-HDF/DSA-DIRECTORY> is the path to the SDK directory of your Vivado Design project

● This takes time and when the blue screen pops up, make sure that the Subsystem AUTO Hardware Settings is selected.

● In this system configuration, we can also modify the kernel and u-boot recipe sources. However, that is not the objective of this demonstration.

● In this system configuration, we enable boot over TFTP and root file system over NFS.

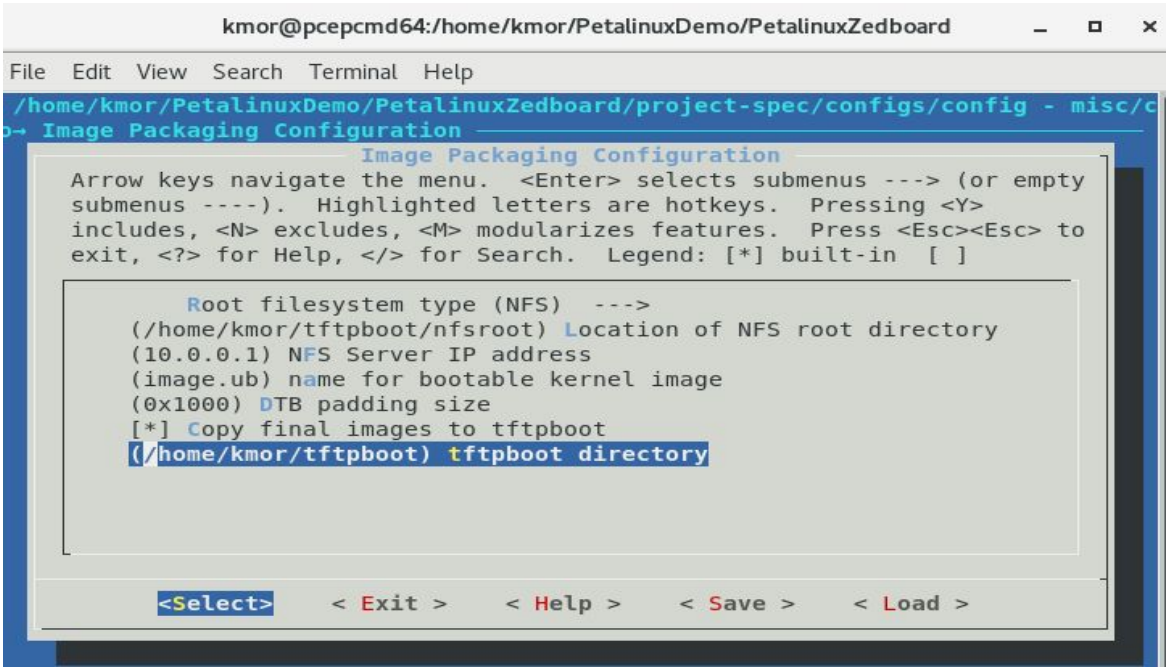

Fig.8.8 : Enabling NFS and specifying the tftpboot directory

● If you are not experienced enough, use default configuration, save and exit. This would mean that you boot over the SD Card with an INITRAMFS file system.

● With DHCP server managing IP allocation in a complex network, it is possible that the TFTP server and the NFS server might be located on separate machines.

● Save and Exit from the system configuration. It takes some time to configure. After this we configure the kernel. Even the rootfs and u-boot can be configured however we don't usually need it.

Step 5 : Configuring the Kernel

● The BSP and the Hardware Description FIle (HDF) usually configure the kernel for all the necessary functionalities expected out of the hardware. However, we may configure it additionally for certain extra features, such as networking and network file system support in this case.

● Execute the following command in the project directory and another blue configuration window pops out **: $ PetaLinux-config -c kernel**

● Select Networking support -> IP: kernel level configuration - IP:DHCP support -IP:BOOTP support -IP:RARP support

● Select File systems -> Network file systems -> Root file systems on NFS

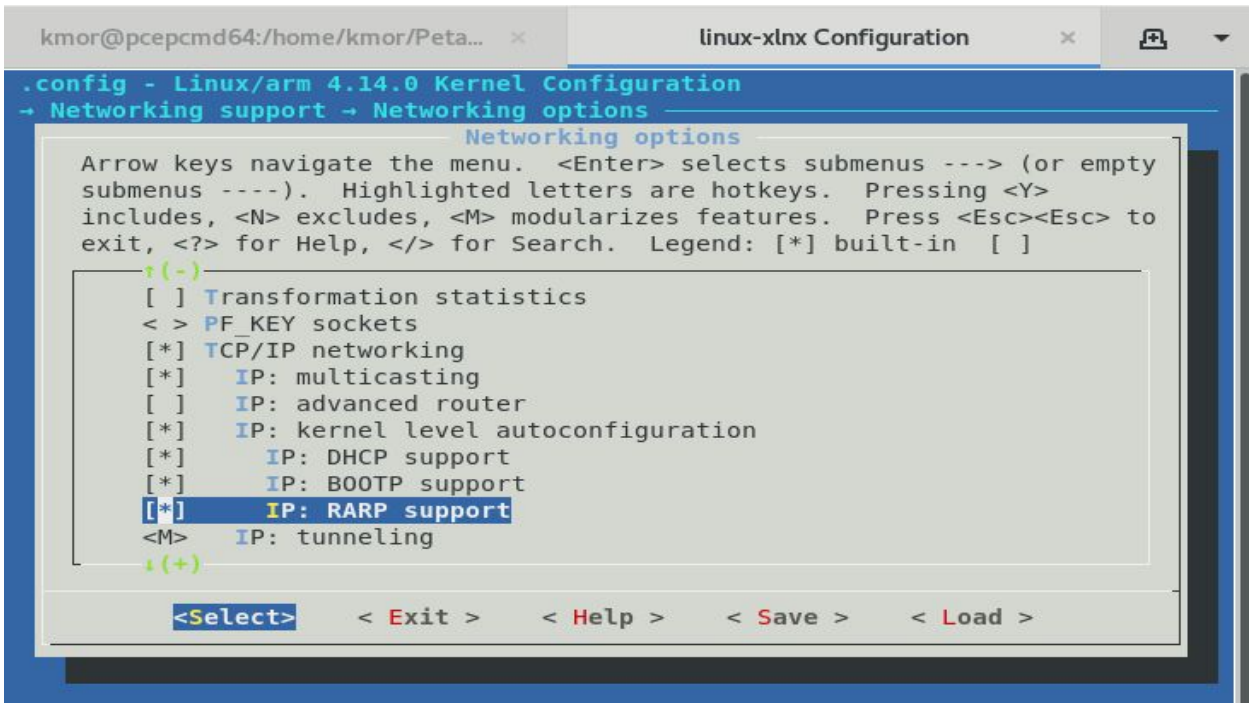

Fig.8.9 : Enabling TCP/IP, DHCP, BOOTP and RARP for NFS mount in kernel config

● NFSv3 is the version used most commonly for NFS servers and clients. Save and Exit from the kernel config.

Step 6. Building the project and generating bootable image

● Execute the following commands to build the project and generate a bootable image.

$ PetaLinux-build -c external-hdf

$ PetaLinux-build -c device-tree -x do_prepare_recipe_sysroot

$ PetaLinux-build -c device-tree

● To know more about this, please read the Page 35 & 36 of the following document : https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1144-PetaLinux-tools-reference-guide.pdf

● Prepare the U-Boot with following commands

$ PetaLinux-build -c u-boot-xlnx -x do_prepare_recipe_sysroot

● " -c" command ensures that all dependencies are automatically taken care of

● After this stage, build the project finally by running the following command :

$ PetaLinux-build

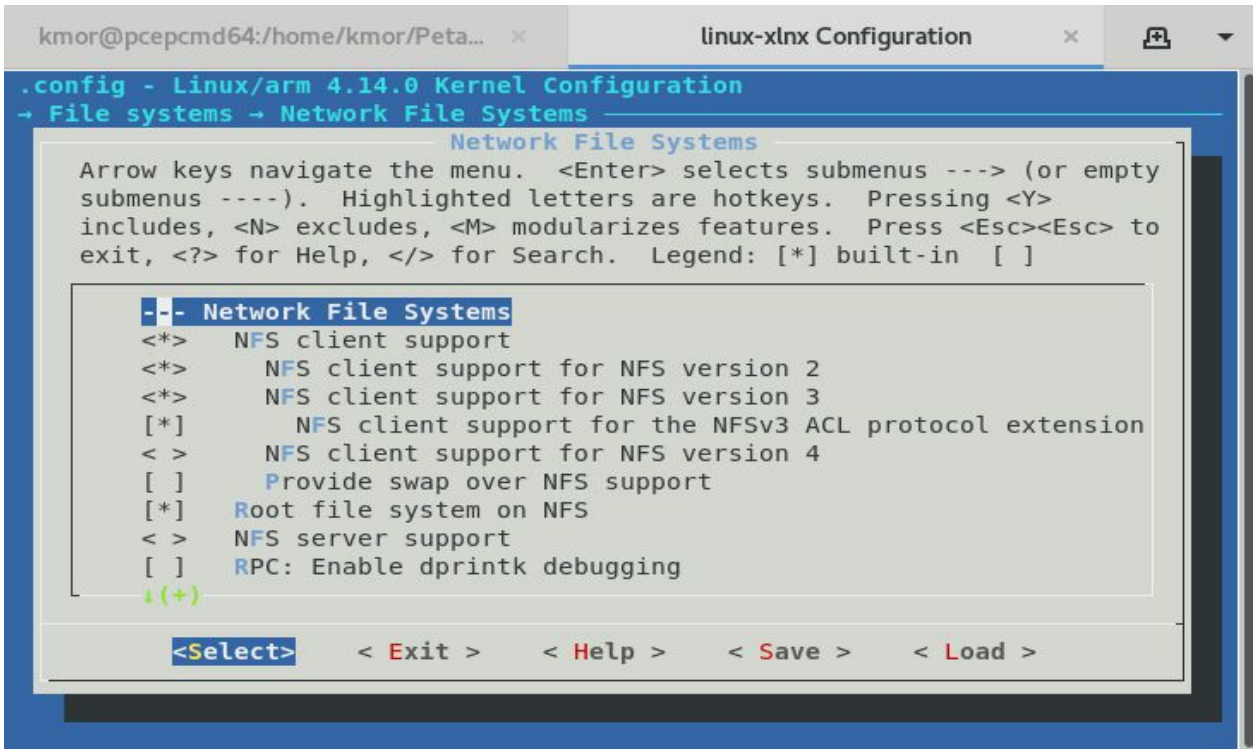

Fig.8.10 : Enabling NFS services in the kernel config

● Generate uImage by typing in the following command :

$ PetaLinux-package --image -c kernel --format uImage

● Generate a Boot Image for Ultrascale+ by typing in the following command :

$ PetaLinux-package --boot --format BIN --fsbl images/linux/zynqmp_fsbl.elf --u-boot

images/linux/u-boot.elf --pmufw images/linux/pmufw.elf --fpga images/linux/*.bit --force

● Generate a Boot Image for Zynq 7000 SoC by typing in the following command

$ PetaLinux-package --boot --fsbl <FSBL image> --fpga <FPGA bitstream> --u-boot

● The FSBL file is located in the images/linux directory whereas the bitstream can be found at the SDK directory of the Vivado project

● Now your PetaLinux project is ready to be tested. We would be booting it over TFTP which would also mount the Root File system over NFS. The directory specified for TFTP and NFS server(common directory in this case) contains all the relevant files needed for boot and rootfs mount.

```
kmor@pcepcmd64:/home/kmor/tftpboot
File  Edit  View  Search  Terminal  Help
[kmor@pcepcmd64 tftpboot]$ ls -a
.          media                  rootfs.ext4.gz        System.map.linux
..         mnt                    rootfs.jffs2          tmp
bin        petrzejdl              rootfs.manifest       u-boot.bin
boot       proc                   rootfs.tar.bz2        u-boot.elf
BOOT.BIN   rootfs.cpio            rootfs.tar.gz         usr
dev        rootfs.cpio.bz2        rootfs.testdata.json  var
etc        rootfs.cpio.gz         run                   vmlinux
home       rootfs.cpio.gz.u-boot  sbin                  zImage
image.ub   rootfs.ext3            sys                   zynq_fsbl.elf
keyshav    rootfs.ext3.bz2        system.bit
lib        rootfs.ext4            system.dtb
[kmor@pcepcmd64 tftpboot]$
```

Fig.8.11 : Contents of the TFTP boot directory

Step 6. Setting up TFTP and NFS server

● For booting over the network, TFTP protocol is very popular. More has been explained about the TFTP protocol in the Appendix A.

● To setup the TFTP server, we need to install a few utilities for TFTP (yum for CentOS/Fedora ; apt for Ubuntu) :

$ sudo yum install tftp tftp-server* xinetd*

● Then we need to configure the /etc/xinetd.d/tftp and the /usr/lib/systemd/system/tftp.service files :

```
# 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/tftpboot
        disable                 = no
        per_source              = 11
        cps                     = 100 2
        flags                   = IPv4
}
```

```
kmor@pcepcmd64:~
File  Edit  View  Search  Terminal  Help
[Unit]
Description=Tftp Server
Requires=tftp.socket
Documentation=man:in.tftpd

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

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

Fig.8.12 and Fig.8.13 : Setting up the TFTP server

● Then we need to modify the tftp server directory using chmod to allow users permission to read,write and modify the file-system with the following commands:

$ sudo chmod -R 777 <tftp-boot-directory>

● We also need to configure the firewall to allow TFTP service and then start the server with following commands :

$ sudo firewall-cmd --zone=public --add-service=tftp --permanent
$ systemctl start xinetd

$ systemctl start tftp
$ systemctl enable xinetd
$ systemctl enable tftp

● Now the NFS server needs to be configured for the same directory. For this first we need to download NFS utilities using following commands (yum for CentOS/Fedora ; apt for Ubuntu) :

$ sudo yum install -y nfs-utils

● Enable and start NFS server using the following commands :

$ systemctl start nfs-server rpcbind
$ systemctl enable nfs-server rpcbind

● Go to /etc/exports and add the file system that would like to share over NFS.

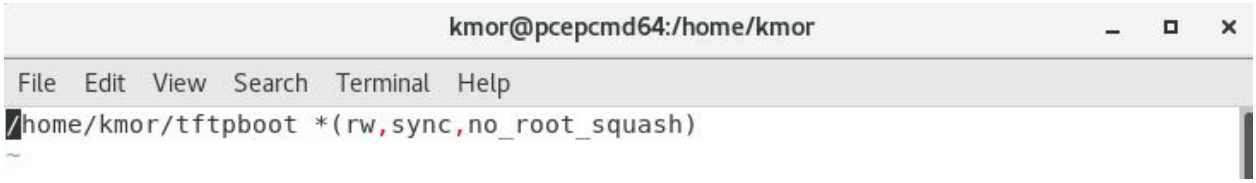

Fig.8.14 : Setting up NFS file share

*** :** All IP Addresses permitted

rw: Writable permission to shared folder

sync: All changes to the according filesystem are immediately flushed to disk; the respective write operations are being waited for.

no_root_squash: If no_root_squash is selected, then root on the client machine will have the same level of access to the files on the system as root on the server.

● Export shared directories using following command :

$ exportfs -r

-r : re-exports all directories after /etc/exports is modified

● Allow firewall to accept NFS requests with following commands :

$ firewall-cmd --permanent --add-service mountd
$ firewall-cmd --permanent --add-service rpc-bind
$ firewall-cmd --permanent --add-service nfs
$ firewall-cmd --reload

● Now the Zedboard is ready to boot over the network.

Step 7. Booting the Zedboard over the Network

● In this method, we use the SD Card for loading the bitstream, FSBL and the U-Boot on the ZedBoard's On-Chip Memory (RAM). We can also use JTAG, which is a slightly slower and manual process for booting.

● Two files are important for booting Linux on the Zedboard :

1. **BOOT.BIN** : A binary file which is responsible for loading the FPGA bitstream, the FSBL and the U-Boot. We keep it on SD card for this demonstration
2. **Image.ub** : This is the kernel and device tree in FIT image format which is loaded by the U-Boot during the boot process into the PS RAM. In this demonstration, we place it on the TFTP server. Device-tree is necessary to define the hardware in a manner in which the kernel understands.

● A serial UART communication between the PC and Zedboard is necessary to access the Zedboard via terminal and see its boot process. This can be achieved by following command :

$ sudo minicom -D /dev/ttyACM0 -b 115200 -8 -o

This sets up a serial communication of baud rate 115200 with the ZedBoard

● Configure the ZedBoard for SD Card boot by modifying the Jumper settings :

Fig.8.15 : Setting up Zedboard jumpers for SD card boot

● Reset the ZedBoard, within moments, the serial terminal and the U-Boot springs to life :

```
U-Boot 2018.01 (Jun 10 2019 - 17:12:39 +0000)

Model: Zynq Zed Development Board
Board: Xilinx Zynq
Silicon: v3.1
DRAM:  ECC disabled 512 MiB
MMC:   mmc@e0100000: 0 (SD)
SF: Detected s25fl256s_64k with page size 256 Bytes, erase size 64 KiB, total 32 MiB
In:    serial@e0001000
Out:   serial@e0001000
Err:   serial@e0001000
Model: Zynq Zed Development Board
Board: Xilinx Zynq
Silicon: v3.1
Net:   ZYNQ GEM: e000b000, phyaddr 0, interface rgmii-id
eth0: ethernet@e000b000
Hit any key to stop autoboot:  0
Zynq>
```

Fig.8.16 : Entering the U-Boot after board bringup

● As mentioned in previous chapters, U-Boot has its own rich environment , with a command terminal and many pre-compiled binaries which help it to execute various services. These can be seen by typing “help” in the U-Boot terminal as follows :

```
Zynq> help
?       - alias for 'help'
base    - print or set address offset
bdinfo  - print Board Info structure
boot    - boot default, i.e., run 'bootcmd'
bootd   - boot default, i.e., run 'bootcmd'
bootefi - Boots an EFI payload from memory
bootelf - Boot from an ELF image in memory
bootm   - boot application image from memory
bootp   - boot image via network using BOOTP/TFTP protocol
bootvx  - Boot vxWorks from an ELF image
bootz   - boot Linux zImage image from memory
clk     - CLK sub-system
cmp     - memory compare
coninfo - print console devices and information
cp      - memory copy
crc32   - checksum calculation
dcache  - enable or disable data cache
dfu     - Device Firmware Upgrade
dhcp    - boot image via network using DHCP/TFTP protocol
```

● One such binary, the “dhcp” command can be used to obtain an IP Address from DHCP server along with address of NFS and TFTP server address, the root file system directory and many other things depending on how the DHCP server is configured. We would use it extensively during the project. We use the CERN IT infrastructures DHCP server for booting over the network.

```
Zynq> dhcp
BOOTP broadcast 1
DHCP client bound to address 192.168.100.10 (3 ms)
```

Fig.8.17 : Requesting DHCP for Zedboard network configuration

● IP address 192.168.100.10 has been assigned to the board by the DHCP server, which recognises the MAC address of the Zedboard. Since the DHCP server does not specify the IP address of the server due to its current configuration, we need to set the server IP address using the "setenv" command which sets environment variables for U-Boot. For this we also need to know the IP address of the server :

```
Zynq> setenv serverip 192.168.100.1
Zynq> print serverip
serverip=192.168.100.1
Zynq>
```

Fig.8.18 : Setting up TFTP server IP address in U-Boot environment

● Environment variables in the U-Boot are like environment variables in the Linux. They help the pre-compiled binaries in U-Boot in executing their by allowing users to pass arguments to the binaries by defining environment variables.

● After the server IP address is set, we run the "run netboot" command. This command starts the TFTP client on Zedboard, which requests the TFTP server to share the "image.ub" file from the TFTP server directory. The server shares the file with the client i.e the U-Boot, which then loads the Kernel at a predefined offsets as shown below :

```
Zynq> run netboot
Using ethernet@e000b000 device
TFTP from server 192.168.100.1; our IP address is 192.168.100.10
Filename 'image.ub'.
Load address: 0x10000000
Loading: ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
         ##########
         7.3 MiB/s
done
Bytes transferred = 3956632 (3c5f98 hex)
## Loading kernel from FIT Image at 10000000 ...
   Using 'conf@system-top.dtb' configuration
```

●After the kernel is loaded, the device is loaded into the memory as follows :

```
## Loading fdt from FIT Image at 10000000 ...
   Using 'conf@system-top.dtb' configuration
   Trying 'fdt@system-top.dtb' fdt subimage
     Description:  Flattened Device Tree blob
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0x103c20d8
     Data Size:    14671 Bytes = 14.3 KiB
     Architecture: ARM
     Hash algo:    sha1
     Hash value:   321ad5c0d94ff72d9775218f351bf0fb42529656
   Verifying Hash Integrity ... sha1+ OK
   Booting using the fdt blob at 0x103c20d8
   Uncompressing Kernel Image ... OK
   Loading Device Tree to 07ff9000, end 07fff94e ... OK
```

●Once the device is loaded, the kernel starts unpacking and starts configuring the system as per the kernel configuration. Once such configuration which we activated the kernel for was DHCP,BOOTP,TCP/IP support along with NFS support. The kernel executes the DHCP request over Ethernet during its unpacking and obtains the NFS root server IP address and the NFS root directory.

●Once the kernel is done unpacking, it requests the NFS root server for mounting the root file system from the server. Once the request is approved, the root file system is mounted over NFS to the Zedboard. It does not mean that the root file system is "loaded" to the SD card or the Zynq PS memory. It means that the Zedboard's root file system is on a server, to which the Zedboard now has an access to read,write and modify.

```
macb e000b000.ethernet eth0: link up (1000/Full)
IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
Sending DHCP requests ., OK
IP-Config: Got DHCP answer from 192.168.100.1, my address is 192.168.100.10
IP-Config: Complete:
     device=eth0, hwaddr=08:00:30:f4:03:35, ipaddr=192.168.100.10, mask=255.255.255.0, gw=192.168.100.1
     host=192.168.100.10, domain=, nis-domain=(none)
     bootserver=192.168.100.1, rootserver=192.168.100.1, rootpath=/home/kmor/tftpboot,tcp,v3     names1
ALSA device list:
  No soundcards found.
VFS: Mounted root (nfs filesystem) on device 0:12.
devtmpfs: mounted
Freeing unused kernel memory: 1024K
INIT: version 2.88 booting
Starting udev
udevd[722]: starting version 3.2.2
udevd[723]: starting eudev-3.2.2
FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
Mon Jun 10 16:09:16 UTC 2019
Starting internet superserver: inetd.
```

Fig.8.19 : Kernel executing and requesting DHCP server for NFS root IP and root directory

●Once the root file system has been mounted, the kernel searches for the “INIT” process in the root file system and once it is found, executes the “INIT” binary, which is the first process to run after booting. The INIT process enters run-level 5, which allows multi-user mode with GUI and then runs the login-daemon process through which we can now log into the Linux running on Zedboard.

```
INIT: Entering runlevel: 5
Configuring network interfaces... RTNETLINK answers: File exists
ifup skipped for nfsroot interface eth0
run-parts: /etc/network/if-pre-up.d/nfsroot: exit status 1
Starting Dropbear SSH server: dropbear.
Starting syslogd/klogd: done
Starting tcf-agent: OK

PetaLinux 2018.3 avnet-digilent-zedboard-2018_3 /dev/ttyPS0

avnet-digilent-zedboard-2018_3 login: random: crng init done
root
Password:
root@avnet-digilent-zedboard-2018_3:~#
```

Fig.8.20 : Logging in to the Linux

●Now the boot process is over and we can operate the Linux as we do it on any other computer.

```
root@avnet-digilent-zedboard-2018_3:~# cd /
root@avnet-digilent-zedboard-2018_3:/# ls
BOOT.BIN              media                 rootfs.ext4           system.bit
System.map.linux      mnt                   rootfs.ext4.gz        system.dtb
bin                   petrzejdl             rootfs.jffs2          tmp
boot                  proc                  rootfs.manifest       u-boot.bin
dev                   rootfs.cpio           rootfs.tar.bz2        u-boot.elf
etc                   rootfs.cpio.bz2       rootfs.tar.gz         usr
home                  rootfs.cpio.gz        rootfs.testdata.json  var
image.ub              rootfs.cpio.gz.u-boot run                   vmlinux
keyshav               rootfs.ext3           sbin                  zImage
lib                   rootfs.ext3.bz2       sys                   zynq_fsbl.elf
root@avnet-digilent-zedboard-2018_3:/# mkdir -p linuxdemo
root@avnet-digilent-zedboard-2018_3:/#
```

Fig.8.21 : Inspecting the mounted root filesystem

●The directory created in the command “mkdir -p linuxdemo” also appears on the server , due to the chmod permissions.

```
[kmor@pcepcmd64 tftpboot]$ ls
bin          mnt                    rootfs.jffs2          tmp
boot         petrzejdl              rootfs.manifest       u-boot.bin
BOOT.BIN     proc                   rootfs.tar.bz2        u-boot.elf
dev          rootfs.cpio            rootfs.tar.gz         usr
etc          rootfs.cpio.bz2        rootfs.testdata.json  var
home         rootfs.cpio.gz         run                   vmlinux
image.ub     rootfs.cpio.gz.u-boot  sbin                  zImage
keyshav      rootfs.ext3            sys                   zynq_fsbl.elf
lib          rootfs.ext3.bz2        system.bit
linuxdemo    rootfs.ext4            system.dtb
media        rootfs.ext4.gz         System.map.linux
```

Fig.8.22 : Directory “linuxdemo” created via Zedboard terminal appears on the NFS server root directory

9. Research Goals of the Project

As we saw in the previous chapters, the High Luminosity LHC and the CMS detector upgrade is also going to result in the upgradation of the read-out and the back-end electronics in the CMS Trigger and

Data acquisition network. The increased data generation from the detector front-end electronics, the increase in data throughput rate and the complex software which would be required by different experiment sub--groups for their research are the primary reasons behind various experiment sub-groups designing the electronic hardware using the Xilinx Zynq and Zynq Ultrascale+ SoC FPGA platforms. Similar such upgrade is happening at ATLAS which is another major experiment like CMS at CERN. The total number of such new hardware with SoCs in the DAQ network could run into thousands at both these experiments. This is an unprecedented situation which presents a few challenges for the CERN IT, CERN Computer Security and experiment system administrators because of the scale of upgrade happening across various experiments.

The challenges arising out of the upgrade can be summarised as follows :

- **The problem of plenty :** Various sub-groups working on different aspects of physics research, design their electronics, which might be using the same or different SoC and might be having completely different PCB design. In addition to this, all of these groups might want to develop their own Embedded Linux, resulting in tens of Linux distributions to be configured, managed and updated by the system administration team. Such customized hardware designed by different groups may require different bitstream configuration, different kernel configuration and different software packages and libraries. This kind of diversity is a challenge for system administrators as they have to provide long term support and management for a variety of hardware and software.
- **Network integration :** When thousands of embedded systems are integrated in the experiment network, their identification, allocation of IP address and most importantly their service configuration is a big challenge. For example. two devices sharing the same MAC address can result in denial of networking to each other and also cause misidentification of the hardware. The embedded systems should also not be running conflicting services that bring the network down. Network bandwidth has to be considered while booting of thousands of embedded systems in the network and also the run-time bandwidth sharing between these systems needs to be planned. We also need to consider if need a powerful global server or a server cluster to boot these many devices over the network. System administrators need to find solutions to these challenges on a regular basis.
- **Security :** Diversity in hardware is good as long as the designers are aware of the security features of the hardware and take responsibility for it. Similarly , diverse Linux distributions are also welcome so long as the sub-group takes responsibility for the security and OS update aspects of its hardware. System administrators at CERN expect users to secure their devices by using the latest kernel updates during system commissioning and by coordinating with other groups to ensure some uniformity in user software and OS configuration. This may not always happen and thus, system administrator teams face the challenge of securing all the systems in various internal networks from external attacks. System administrators try to mitigate these risks by regularly applying security patches to systems in the network, through updates and ensure an enterprise grading operating systems support by integrating software patches from the distribution upstream. Hence, having a lot of different hardware, software and OS to manage complicates the work of system administrators.

Given these challenges, there are a number of research goals associated with the project. These research goals deal with networking, software stack deployment, the booting process, the kernel , the management of the file system and system updates.

The research goals can be summarised as follows :

- **Building a common Linux distribution :** An important goal of the project is to build and propose common Linux distribution such that it at least meets minimal requirements of maximum sub-groups, system administrators and computer security teams. The minimal requirements can be hardware support in the form of kernel drivers, networking support, secure updates and application support.
- **Defining a common boot process :** With many sub-groups designing their own hardware and potentially building a Linux distribution for their hardware using a standardised procedure in the future, it is important to propose a common minimal Linux booting process, which is shared by all the SoC embedded systems deployed in the network such that all these embedded systems ideally boot similarly and load board specific bitstream, kernel and root file system over the network. This will help in board specific updates being incorporated after reboot. It is also of interest to investigate if a common FSBL and U-Boot is possible for most or all of the embedded systems in the network.
- **Defining a flexible and scalable NFS root filesystem :** As uniformity in the Linux distribution and boot process is proposed, it is also crucial to maintain uniformity in the root file system. However, root file system contains software packages, rpms and other utilities which are specific to each board. At the same time, it is true that many features required from the root filesystem , such as device drivers and /boot directory etc. would be common. Thus, it is proposed that a common root file-system be maintained for all the embedded systems and research on ways to mount an additional layer of board-specific root filesystem directories. It needs to be investigated if we can incorporate a board specific readable and writable filesystem on top of a common root filesystem. If it is possible, it needs to be investigated in what way should it be implemented.
- **Defining a method of deploying software and updates :** Embedded systems in the network might use software that are operational with a specific kernel configurations and FPGA designs. In such a scenario, even if the boot process and the distribution have commonalities, the system updates pertaining to installed software packages and deployment of new software packages or applications cannot be common for each board, as it might require a lot of memory space or in the worst case, cause operational conflicts. Thus, it is important to define a method of deploying updates and new software to the embedded systems during run-time. We need to investigate different ways of achieving this. It can be done using configuration management software on the server or on the boards or on both sides. The system update and software deployment question is tightly linked to the root file system over NFS and its role in this regard also needs to be investigated further.
- **Testing the CentOS kernel with the Xilinx Zynq Ultrascale+ SoC :** The CentOS AArch64 kernel for 64-bit ARM processors needs to be tested with the Zynq Ultrascale+ . While the Xilinx Linux kernel comes fully equipped with all necessary functionalities necessary for deploying a Linux distribution on the Ultrascale+ SoC, it also needs to be investigated what kind of features

does the AArch64 kernel provide for Zynq Ultrascale+ and how much additional effort would be required to make this kernel useful for CMS groups instead of Xilinx Linux kernel. The motivation behind this endeavour is the familiarity of CERN system administrators, IT and security teams with CentOS, its update cycle and experience in debugging CentOS specific problems.

10. Workplan

I will work with the CMS DAQ group at CERN Meyrin site with 100% attendance (5 days/week) from July 1,2019 until January 17,2020 (approximately 24 weeks).

Week	Task
Week 1-2	Investigate the building of Linux distribution using Xilinx Kernel and Yocto Project. Try to boot the distribution over network. Test PetaLinux for Zynq Ultrascale+ and boot over network.

Week 3-4	Investigate the building of Linux distribution using CentOS root filesystem and Xilinx Linux kernel with Yocto Project and Petalinux respectively. Try to boot the distribution over network. Investigate differences in the build process.
Week 5-6	Investigate additional configurations,scripts and sources required to make CentOS kernel have Xilinx specific features
Week 7-8	Investigate the building of Linux distribution using CentOS kernel with Yocto Project and Petalinux respectively. Try to boot the distribution over network. Investigate differences in the build process.
Week 9-10	Fix problems with the U-Boot and its DHCP configuration.Investigate loading of FPGA firmware over the network. Investigate loading of device tree over the network
Week 11-12	Build a repository of packages and applications for CentOS 7 distribution and verify its seamless installation through package manager system of CentOS 7.
Week 13-14	Investigate board-specific scalable read-write root filesystem. Compare software update through root filesystem with update using package manager.
Week 15-16	Try installing custom software and applications on the CentOS on Zynq Ultrascale+ over the network.
Week 17-18	Test network configuration manager installation on boards. Test how board OS can be configured using these managers.
Week 19-20	Investigate strategy of regular system updates over the network for Embedded Linux on systems. Investigate board-specific update.
Week 21-22	Coordinate with system administrators to integrate the Zynq Ultrascale+ into CMS network and run tests.
Week 23-24	Documentation.

Appendix A

What is TFTP and how does it work?

TFTP stands for Trivial File Transfer Protocol and it is a popular file transfer protocol used by clients to get files or put files onto a remote server. It cannot add,delete, modify the contents of the remote server as it is a simple protocol and does not have features like user authentication. It is usually used on local networks. It was standardised in 1981 and its latest specification are defined by the RFC1350 standard. TFTP uses UDP as its transport protocol and the port 69 for communication. TFTP is popularly used for booting systems over the network.

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 length blocks which are 512 bytes in size by default. Each block transfer must be acknowledged.A data packet of less than 512 bytes at the end signals termination. Both the server and the client are considered senders and receivers. One sends data and receives acknowledgments, the other sends acknowledgments and receives data.

What is DHCP and how does it work?

DHCP stands for Dynamic Host Configuration Protocol. DHCP is a network management protocol that uses UDP/IP , which is capable of assigning static or dynamic IP address and other network configuration parameters(such as TFTP server IP, NFS root directory, NFS server IP) to each device that requests such an assignment from the DHCP. In the absence of a DHCP server, a computer or other device on the network is assigned IP addresses manually. Various DHCP versions defined by different RFC standards.

The DHCP also operates based on the client–server model. When a computer or other device connects to a network, for which it does not have an IP address, its DHCP client software sends a DHCP broadcast query requesting the necessary information to configure itself for the network. Any DHCP server on the network may service the request. If one server fails to respond, another DHCP server may respond. The DHCP server configuration specifies a range of IP addresses with the server can assign. It is called a pool of IP addresses. It defines the range within which unique IP addresses may be assigned. It also specifies the information about client configuration parameters such as default gateway, domain name, time server, TFTP server address etc.

The DHCP server may respond to the request with specific information as per defined in the DHCP server configuration. A DHCP client usually requests while booting as seen in chapter 8, and might be assigned a different set of information of client reboot or server reconfiguration.

References

1. Brondolin, E. (December 2017). Track reconstruction in the CMS experiment for the High Luminosity LHC - Semantic Scholar.

2. Röwert, Nicolas M.(October 2018). Assembly and Characterization of a First Functional 2S Module for the CMS Phase -2 Upgrade at LHC. RWTH Aachen.

3. Hugo,Dewitte(2016). Implementation of an FPGA-based calibration procedure at the detector level for the future high-luminosity phase of the CMS.Vrije Universiteit Brussel.

4. Lemaire, Jérôme(2016). Exploring FPGA hardening solutions at the detector level for the future high luminosity phase of the CMS experiment at the LHC. Vrije Universiteit Brussel.

5. Cieri,David(January 2018). Development of a Level-1 Track and Vertex Finder for the Phase II CMS experiment upgrade. University of Bristol.

6. James,Tom(February 2018). A hardware track-trigger for CMS at the High Luminosity LHC. Imperial College London.

7. Tosi, Mia. (2017). The CMS trigger in Run 2. 523. 10.22323/1.314.0523.

8. ZABI, Alexandre. (2015). The CMS calorimeter trigger upgrade for the LHC Run II. 414. 10.22323/1.213.0414.

9. CMS Collaboration. (September 2017). The Phase 2 Upgrade of the CMS DAQ - Interim Technical Design Report.CERN.

10. Tapper, Alex. (2017). The CMS Level-1 Trigger for LHC Run II. 242. 10.22323/1.282.0242.

11. Trocino, Daniele. (2014). The CMS high level trigger. Journal of Physics: Conference Series. 513. 012036. 10.1088/1742-6596/513/1/012036.

12. Andre, Jean-Marc & Andronidis, Anastasios & Behrens, Ulf & Branson, James & Brummer, Philipp & Chaze, Olivier & Contescu, Cristian & G. Craigs, Benjamin & Cittolin, Sergio & Darlea, Georgiana-Lavinia & Deldicque, Christian & Demiragli, Zeynep & Dobson, Marc & Erhan, Samim & Richard Fulcher, Jonathan & Gigi, Dominique & Glege, Frank & Gomez-Ceballos, Guillelmo & Hegeman, Jeroen & Zejdl, Petr. (2016). Performance of the new DAQ system of the CMS experiment for run-2. 1-4. 10.1109/RTC.2016.7543164.

13. André, Jean-Marc & Behrens, Ulf & Bocci, Andrea & Branson, James & Cittolin, Sergio & Da Silva Gomes, Diego & Darlea, Georgiana-Lavinia & Deldicque, Christian & Demiragli, Zeynep & Dobson, Marc & Doualot, Nicolas & Erhan, Samim & Richard Fulcher, Jonathan & Gigi, Dominique & Gladki, Maciej & Glege, Frank & Gomez-Ceballos, Guillelmo & Hansen, Magnus & Hegeman, Jeroen & Zejdl, Petr. (2018). The CMS Data Acquisition System for the Phase-2 Upgrade.

14. Bauer, Gerry & Bawej, Tomasz & Behrens, Ulf & Branson, James & Chaze, Olivier & Cittolin, Sergio & Antonio Coarasa, Jose & Darlea, Georgiana-Lavinia & Deldicque, Christian & Dobson, Marc & Dupont, Aymeric & Erhan, Samim & Gigi, Dominique & Glege, Frank & Gomez-Ceballos, Guillelmo & Gomez-Reino, Robert & Hartl, Christian & Hegeman, Jeroen & Holzner, Andre & Zejdl, Petr. (2014). The new CMS DAQ system for LHC operation after 2014 (DAQ2). Journal of Physics: Conference Series. 513. 012014. 10.1088/1742-6596/513/1/012014.

15. T. Bawej et al., "The new CMS DAQ system for run-2 of the LHC," 2014 19th IEEE-NPSS Real Time Conference, Nara, 2014, pp. 1-1.doi: 10.1109/RTC.2014.7097437

16. Hegeman, Jeroen & Andre, Jean-Marc & Behrens, Ulf & Branson, James & Chaze, Olivier & Cittolin, Sergio & Darlea, Georgiana-Lavinia & Deldicque, Christian & Demiragli, Zeynep & Dobson, Marc & Erhan, Samim & Fulcher, Jonathan & Gigi, Dominique & Glege, Frank & Gomez-Ceballos, Guillelmo & Hansen, Magnus & Holzner, Andre & Jimenez-Estupinan, Raul & Masetti, Lorenzo & Zejdl, Petr. (2015). The CMS Timing and Control Distribution System. 1-3. 10.1109/NSSMIC.2015.7581984.

17. K. Albertsson et al., "A New Event Builder for CMS Run II", Journal of Physics: Conference Series, vol. 664, no. 8, pp. 082035, 2015.

18. Holt, A., & Huang, C. (2018). Embedded operating systems: A practical approach. Cham, Switzerland: Springer.

19. Tanenbaum, A. S. (2015). Modern operating systems Global edition. Pearson Education.

20. Arch Linux ARM. (n.d.). Retrieved from https://archlinuxarm.org/

21. Kumar, S. K., Mackenzie, Shiva Kumar, & Microsoft. (2018, October 09). Operating Systems | Windows 10 vs Mac OS vs Linux vs Chrome OS! Retrieved from https://vsbytes.com/windows-10-vs-mac-vs-chrome-os-vs-linux/

22. Hallinan, C. (2019). Embedded Linux Primer. Publisher : Prentice Hall.

23. Abbott, D. (2018). Linux for embedded and real-time applications.

24. Kroah-Hartman, G. (2007). Linux kernel in a nutshell. Sebastopol: OReilly.

25. Streif, R. J. (2016). Embedded Linux systems with the Yocto Project. Boston: Prentice Hall.

26. Bovet, D. P., & Cesati, M. (2006). Understanding the Linux kernel. Beijing etc.: OReilly.

27. Xilinx(December 2018). PetaLinux Tools Documentation : Reference Guide(UG1144).

28. Simmonds, C. (2017). Mastering embedded Linux programming: Unleash the full potential of embedded Linux. Birmingham, Mumbai: Packt.

29. Chapter 5. File and Disk Management. About initramfs : http://www.linuxfromscratch.org/blfs/view/svn/postlfs/initramfs.html

30. Initramfs : https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt

31. What is Network File System (NFS)? Retrieved from : https://searchenterprisedesktop.techtarget.com/definition/Network-File-System

32. Chapter 9. Network File System (NFS). Retrieved from : https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/storage_administration_guide/ch-nfs

33. CentOS Wiki. Retrieved from https://wiki.centos.org/

34. Archlinux Wiki. Retrieved from https://wiki.archlinux.org/

35. Avnet(2014). Zedboard Hardware User's Guide.

36. Xilinx(2018). Zynq-7000 SoC Datasheet : Overview(DS190 v1.11.1).

37. Xilinx(September 2015). Zynq-7000 All Programmable SoC Software Developers Guide.

38. Xilinx(2019). ZCU102 Evaluation Board User Guide(UG1182).

39. Xilinx(2018). Zynq Ultrascale+ MPSoC Data Sheet : Overview(DS891 v1.7).

40. Xilinx(June 2018). Zynq Ultrascale+ MPSoC Software Developer Guide.

41. Yocto Project Overview and Concepts Manual. Retrieved from : https://www.yoctoproject.org/docs/2.7/overview-manual/overview-manual.html