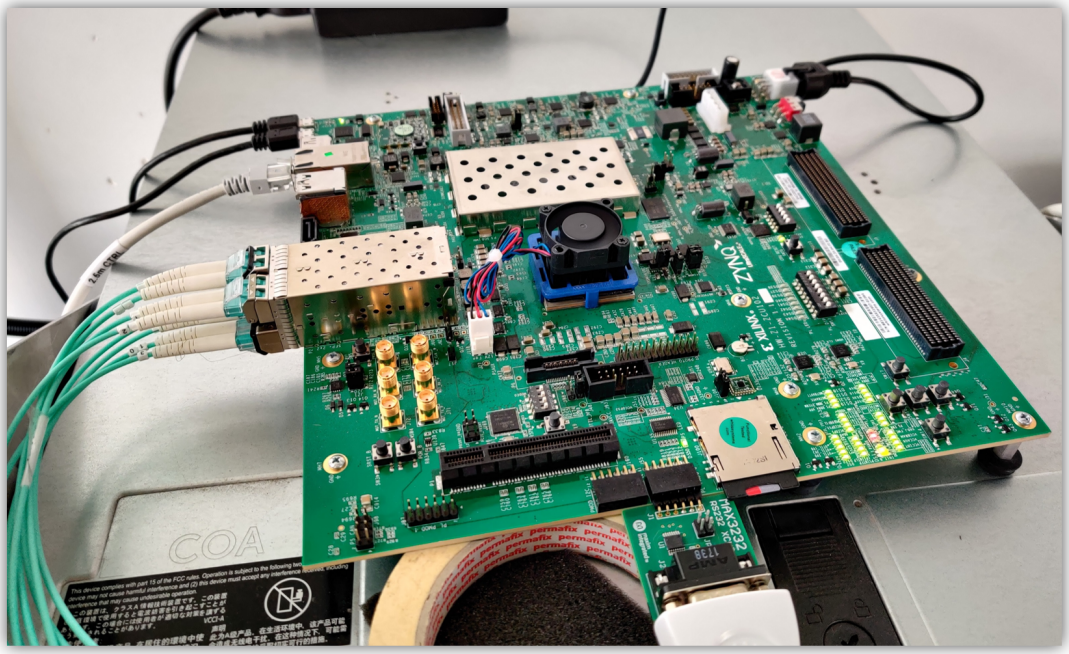
A reliable booting system for Zynq Ultrascale+ MPSoC devices

An embedded solution that provides fallbacks in different parts of the Zynq MPSoC booting process, to assure successful booting into a Linux operating system.



A thesis presented for the Bachelor of Science in Electrical Engineering
at the University of Applied Sciences Utrecht

Name	Nekija Džemaili
Student ID	1702168
University supervisor	Corné Duiser
CERN supervisors	Marc Dobson & Petr Žejdl
Field of study	Electrical Engineering (embedded systems)

February 15th, 2021
Geneva, Switzerland

CERN-THESIS-2021-031
17/03/2021

Disclaimer

Preface

This thesis was written for the BSc Electrical Engineering degree of the HU University of Applied Sciences Utrecht, the Netherlands. During the degree, I specialized in embedded systems and found myself in an environment that allowed me to excel as an engineer. I'd like to thank my professor Corné Duiser for being my mentor throughout the studies. Also for helping me through the thesis as my examiner and answering my many questions. Other professors, in particular Dr. Franc van der Bent, Hubert Schuit, and Bart Bozon are thanked for their interesting courses on embedded systems and fun company in the lab.

The thesis was carried out over the period of 14 months at the CMS Data acquisition & trigger group of CERN. The CMS DAQ group granted me the opportunity to work on a challenging and interesting project involving the Zynq Ultrascale+ MPSoC. The thesis is written for engineers that want to learn about the Zynq Ultrascale+ MPSoC and its development.

I'd like to thank Dr. Petr Žejdl for mentoring me during the project. His guidance and kindness is tremendously appreciated. Not only did he support me during working hours, but also in his free time. His encouragement and faith motivated me to excel during the project. My countless amount of questions were all answered by Petr's expertise in the field of embedded systems.

I'd like to thank Dr. Marc Dobson for being my supervisor and supporting me during my time at CERN. His positive criticism and keenness helped me many times during the thesis writing and SoC meetings. His expertise of the CMS-experiment and the data acquisition system helped me understand what I've been working for.

Lastly, I'd like to thank Dominique Gigi, Dr. Emilio Meschi, Dr. Atilla Racz, and Dr. Frans Meijers, along with the rest of the CMS DAQ team, for their help and kindness during my time at CERN. They provided me with a friendly working environment in a time of global pandemics and uncertainty.

Nekija Džemaili

Geneva, Switzerland

15$^{\text{th}}$ of February, 2021

Abstract

CERN is working on the High-Luminosity LHC upgrade which will be installed in 2025. As a result, the CMS-experiment and its data acquisition (DAQ) system will also be upgraded. The upgrade of the CMS DAQ system involves the installation of new electronics that will also host the Zynq Ultrascale+ MPSoC from Xilinx (Multiprocessor Systems on a Chip). The Zynq Ultrascale+ MPSoC will run control and monitoring software on a Linux operating system (OS).

Booting a Linux OS on the Zynq MPSoC involves a complex multi-stage booting process. The complexity of the booting process introduces possible failures that can prevent the Zynq MPSoC from booting correctly. This thesis presents the research, design, implementation, and testing of a reliable booting system that recovers the Zynq MPSoC from boot failures, upgrade failures, and running failures.

The reliable booting system consists of five fallbacks in different parts of the Zynq MPSoC booting process, to account for a wide range of failures. The fallbacks have been designed to bring the Zynq MPSoC to a well-known booted state after a failure. The booting system can also boot through the network and perform automatic firmware upgrades with a rollback on failure. Users of the hardware are automatically notified after a failure was detected and a fallback was triggered in the system. The booting system is automatically built and packaged by a continuous integration build system. It has been made portable for new hardware by integrating the system in an easy-to-use board support package.

Research on the possible failures in the Zynq MPSoC has been carried out. The test results have concluded that the fallbacks are able to successfully recover the Zynq MPSoC from all the researched failures. The results also highlighted a few areas that can be researched in a follow-up project to further improve the reliable booting system.

Table of contents

Introduction

This bachelor thesis is being conducted at the European Organization for Nuclear Research, also known as CERN (Conseil Européen pour la Recherche Nucléaire). CERN operates the largest particle physics laboratory in the world. It provides a range of particle accelerator facilities, detectors, and infrastructure, needed for high-energy physics experiments. International collaborations between nations, universities, and scientists drive CERN's research. The organization currently has around 2500 staff members that take part in the design, construction, and operation of the accelerator complex. A collection of institutes and contractors work together with the staff to build the experiments. The data collected from each experiment is being used by many scientists at CERN, universities, and research institutes [1].

Figure 1: The globe of Science and Innovation, together with the sculpture "Wandering the immeasurable" in front of CERN [2].

The project is specifically carried out in the Data acquisition (DAQ) & trigger group of the CMS (Compact Muon Solenoid) experiment. The DAQ & trigger system processes and filters the data from the CMS detector. During the next system upgrade, DAQ & trigger will integrate the Zynq Ultrascale+ MPSoC from Xilinx (Multiprocessor Systems on a Chip). This chip will provide control and monitoring for the timing and control distribution hardware and data acquisition hardware of the DAQ system. Control and monitoring will mainly be carried out in a Linux operating system that runs on the Zynq MPSoC.

The objective of this project is to develop an embedded solution that provides fallbacks in the different parts of the Zynq Ultrascale+ MPSoC booting process. These reliable booting fallbacks will assure that the system ends up in a well known state, wherever possible booting into a Linux operating system.

Chapter 1 describes CERN and its accelerator complex, as well as the CMS experiment. Details are given on the Large Hadron Collider and the other accelerators. Furthermore, the sub-detectors and data acquisition system of the CMS detector are described.

Chapter 2 is dedicated to describing the project in further detail. The background and project objectives are given. Furthermore, the requirements and preconditions are described. The reliability requirement has also been defined. Finally, the final products that are delivered at the end of the thesis are summarized.

The research, design, and implementation of the reliable booting system are described in Chapters 4 to 6. The testing of the reliable booting system, along with the results, is given in Chapter 7. Finally, a conclusion is drawn about the thesis in Chapter 8.

Future work for a follow up project is given in Chapter 9. The thesis also includes a summary of other work that was carried out during the project. This is described in Chapter 10.

1. The CERN laboratory

1.1 Introduction to CERN

After the Second World War, a small number of scientists imagined the creation of a European physics laboratory. The laboratory would act as a way to unite European scientists and share the cost of nuclear physics facilities. In the period of 1949 to 1953, various events led to the creation of CERN [3].

During its history, CERN has had some key achievements: e.g. the invention of the *World Wide Web* in 1989, the creation of the *Large Hadron Collider (LHC)*, the discovery of the *W and Z bosons*[1] in 1983, and the discovery of the *Higgs boson*[2]. Apart from CERN's primary research in fundamental particle physics, the laboratory also plays a role in developing new technologies that may be used outside its research. Examples of this are the aforementioned World Wide Web and contributions to medical technologies and aerospace applications [4].

1.2 CERN's accelerator complex

CERN operates a total of eight particle accelerators, the biggest and most powerful of which is the Large Hadron Collider (LHC). The LHC is a circular accelerator with a circumference of 27 km. It can accelerate particles in a circle until they reach the required nominal energy for the experiments. This is in contrast to a linear accelerator, where particles can only travel through the accelerator once.

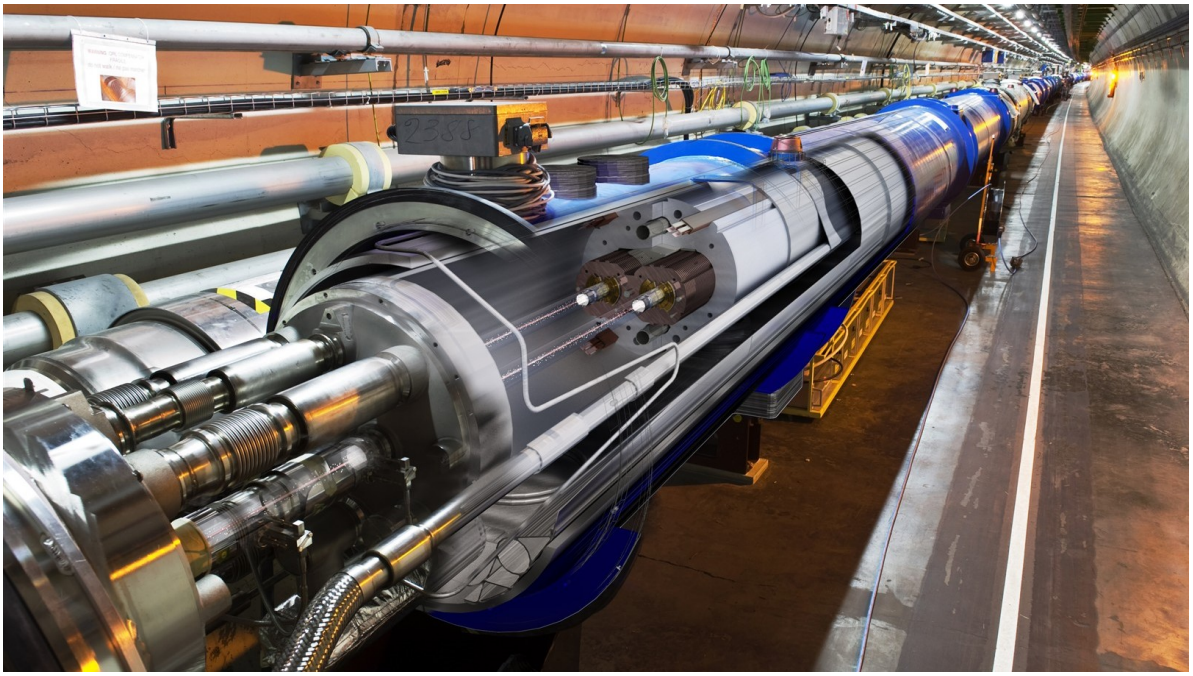


Figure 1.1: Main dipole in one of the straight sections of the LHC [5], 100 meters underground.

The LHC accelerates multiple charged particle beams in opposite directions around the ring. The beams travel through two separate beam pipes, kept at an ultra-high vacuum of around 10^{-10} mbar [6]. The vacuum assures that the particles do not collide with any gas molecules.

[1] The W and Z bosons are carriers of the weak interaction between particles. The weak interaction is responsible for the radioactive decay of atoms.

[2] The Higgs boson is the visible manifestation of the Higgs field. Particles that interact with the Higgs field acquire a mass.

The trajectory of the particle beams is bent by a magnetic field, generated by superconducting electromagnets. A total of 1232 dipole magnets are used in the LHC. Each magnet can generate a magnetic field of 8.3 T. To achieve this, the magnet is cooled to -271.3 °C and operates with a current of 11 kA. Besides, 392 quadrupole magnets are used to keep the particles in a tight beam by squeezing them together.

Acceleration of the particles is done using radio-frequency (RF) cavities. These are metallic chambers that contain a strong electromagnetic field. When charged particles move through a cavity, some energy from the electromagnetic field is transferred to the particle [7]. There are 16 RF cavities in the LHC in total. Each cavity is cryogenically cooled using liquid helium to reach a superconductive state.

Before particles enter the LHC, they are sped up in a series of other accelerators used as "injectors". These can be seen in Figure 1.2. The particles start in Linac 4 (Linear accelerator) and pass through the PS Booster (Proton Synchrotron Booster), the PS (Proton Synchrotron), and the SPS (Super Proton Synchrotron).

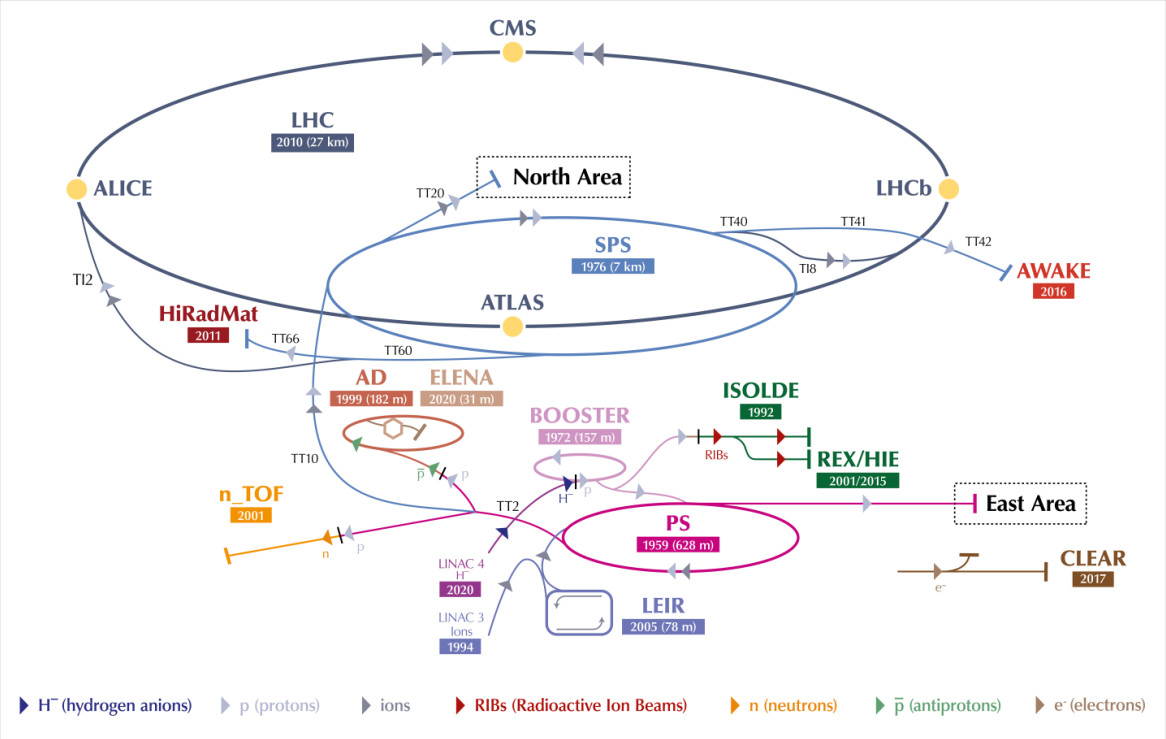


Figure 1.2: Graphical representation of CERN's accelerator complex in 2019 [8].

When the particles reach the maximum speed that one part of the accelerator chain can achieve, they are passed to the next. The particles mainly accelerated in this chain are protons. Other types of particles, such as ions, can also be accelerated and brought to collision. This is done using the Linac 3 and LEIR (Low Energy Ion Ring) accelerators (see Figure 1.2).

Figure 1.2 also shows the four main experiments around the LHC. ATLAS and CMS are general-purpose detectors, which are used for a wide range of physics experiments. They both have contributed to the discovery of the Higgs boson and have the same scientific goals. Their differences lie in the magnet designs and technical solutions that were chosen. The ALICE experiment is dedicated to heavy-ion physics. Finally, the LHCb experiment studies a type of particle called a beauty quark.

1.3 The CMS experiment

The Compact Muon Solenoid (CMS) detector is used as a general-purpose detector. It can generate a magnetic field with a strength of 4 T. To achieve this, the solenoid operates with a current of 20 kA and

is cooled to -268.5 °C [9]. The purpose of the solenoid is to bend the trajectories of the charged particles that result from the collisions. This serves two purposes:

- It helps identify the charge of the particles. Positively and negatively charged particles curve in opposite directions in the same magnetic field [9].
- It allows the measurement of the momentum of particles. A particle with high momentum will have a trajectory with a lower radius of curvature compared to the trajectory of a low momentum particle [9].

To confine the magnetic field of the detector, a steel return yoke is used in four layers. Figure 1.3 shows a 3D-model of the CMS detector and its components.

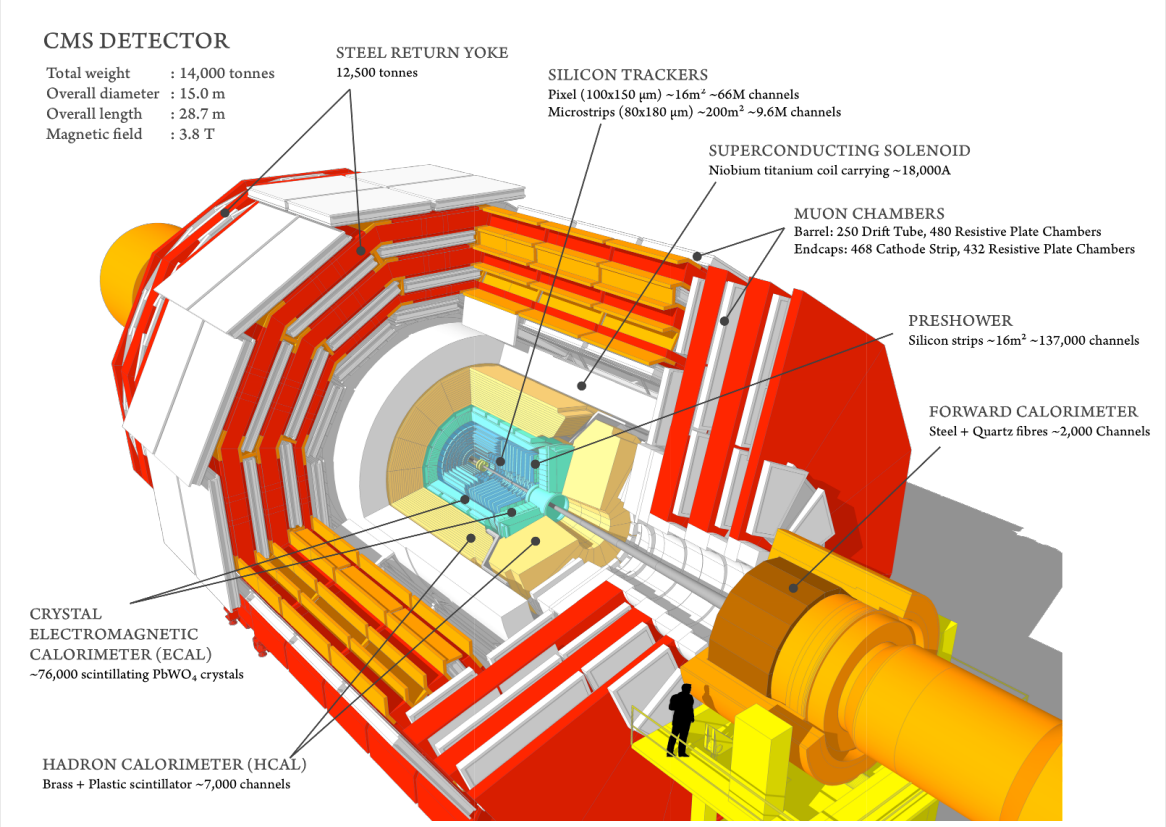


Figure 1.3: 3D-model of the CMS detector showing the solenoid and its return yoke, and the sub-detectors [10].

1.3.1 CMS sub-detectors

The CMS detector consists of multiple sub-detectors: The silicon tracker, the electromagnetic calorimeter (ECAL), the hadron calorimeter (HCAL), and the muon chambers.

The silicon tracker is the most inner part of the detector. It is able to reconstruct the tracks of charged particles coming from the collision. This reconstruction enables the measurement of the momentum of particles [11]. The tracker can reconstruct the tracks of high-energy muons, electrons, hadrons and tracks from the decay of short-lived particles.

The two calorimeters are designed to stop particles and measure the amount of energy that is released [12]. The electromagnetic calorimeter (ECAL) measures the energy of electrons and photons. The calorimeter uses dense highly transparent crystals that stop the particles. The crystals scintillate when electrons and photons pass through it [13]. The amount of produced light is proportional to the particle's energy. Photo-detectors are glued to the crystals to measure the light intensity.

The hadron calorimeter (HCAL) measures the energy, positions, and arrival times of hadrons[3]. The calorimeter consists of alternating absorber and scintillator layers. When a hadronic particle hits an absorber layer (brass or steel) it is stopped and causes an interaction that produces secondary particles [14]. These secondary particles can interact with the following absorber layers, creating more particles and causing a particle shower (Figure 1.4) [14]. As the shower develops, the particles pass through multiple scintillation layers. These layers are used for measuring the energy of the particles, just like in ECAL.

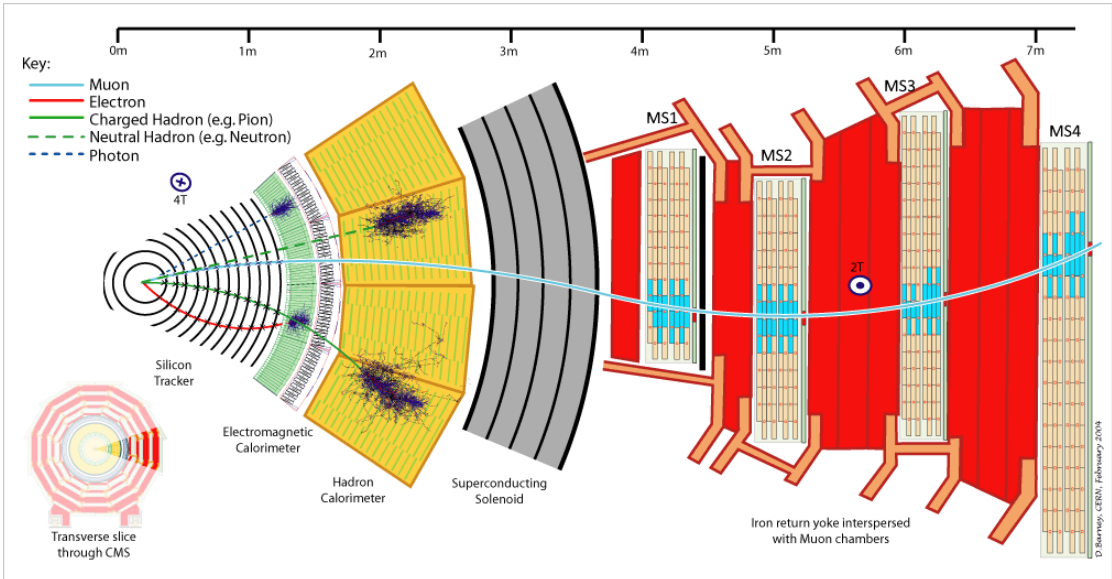


Figure 1.4: Slice of the CMS detector showing particle trajectories after a collision in the detector [16].

The only particles that are not stopped in the calorimeters are muons and neutrinos. Neutrinos[4] are very challenging to detect, because they have very little interaction with matter. The muons[5] are tracked using the muon chambers located outside the solenoid coil (Figure 1.4). Measuring the trajectory of the muons is done by fitting a curve to "hits" among the 4 muon stations (MS). In each station, several layers of gaseous ionization chambers measure the track and energy of the particles [18].

1.3.2 CMS DAQ system

The CMS detector can be seen as a big 3D camera. It will capture pictures (or events) of the particles with a frequency of 40 MHz. A large part of the events is not interesting to look at though because they don't contain any signs of interesting physics. That's why the events need to be filtered [20].

Trigger filtration system

Filtering is done using a two-level triggering system, consisting of the Level 1 trigger (L1) and the high-level trigger (HLT). The L1 trigger reduces the event rate from 40 MHz to 100 kHz. It uses FPGAs, programmed with algorithms, to decide which events are interesting [20]. The L1 trigger electronics are located close to the detector in the underground service cavern. The HLT consists of server farms above ground, which further reduce the event rate from 100 kHz to 100 Hz using software algorithms [20].

Data acquisition pipeline

The triggers are part of the data acquisition system of CMS (DAQ). The underground and above-ground parts are connected through optical fibers and links. A high-level diagram of the system can be seen in Figure 1.5.

[3]Hadrons are particles made of quarks and gluons [15].

[4]A neutrino is a particle that is similar to an electron, but has no electrical charge and almost no mass [17].

[5]Muons are charged particles that are approximately 200 times heavier than electrons or positrons [19].

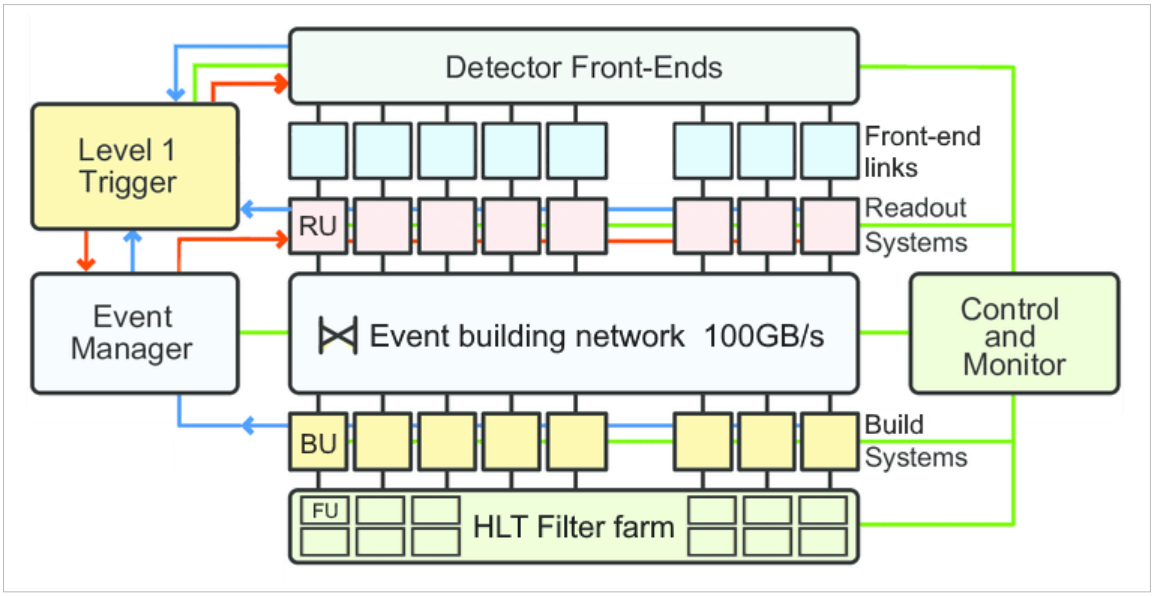


Figure 1.5: Diagram of the CMS DAQ system.

The data from the sub-detectors passes through the DAQ system, starting in the front-end readout links (FEROL). The FEROLs pack the raw data fragments into TCP/IP packets and distribute them to a set of readout units (RU) through a commercial network. The RUs aggregate the packets and pass them to the event building network.

The event building network collects the data packets from the RUs and sends them to a single builder unit (BU). The builder unit proceeds to assemble the packets into a complete event. While one of the BUs is building an event, another BU starts building the next event. This means that the events are processed in parallel. Finally, each complete event goes to the filter units (FU), which form the HLT.

Timing and control distribution system

The DAQ pipeline is controlled by the timing and control distribution system (TCDS) boards. These boards decide if the pipeline can accept more data from the sub-detectors [21]. TCDS also distributes a global 40 MHz clock from the LHC to CMS. The clock is used to synchronize data-taking between the sub-detectors, DAQ system, and the bunch crossings[6] [21].

The DTH upgrade

In 2025, the CMS DAQ system will be upgraded in preparation for the first run of the High-Luminosity LHC. The previously mentioned FEROLs, RUs, and TCDS will be replaced with the new DAQ and TCDS Hub (DTH) [21]. The DTH will be responsible for the translation of raw data to TCP/IP packets, distribution of timing signals and control, and the collection of the individual board status for monitoring [21].

The second DTH prototype is currently being created. This prototype will use a Zynq Ultrascale+ MPSoC from Xilinx for the control and monitoring tasks of the DTH.

[6] A particle bunch is a group of particles traveling in the beam pipe. When two particle bunches are made to collide, it is called a bunch crossing.

2. Project description

2.1 Background

CERN is working on upgrading the LHC to an accelerator with increased performance, the High-Luminosity LHC (HL-LHC). This upgrade is planned to be installed in 2025. The upgraded accelerator will allow for more particle collisions per bunch crossing. This means that the CMS detector will collect more data and that data processing speed needs to be increased. The CMS DAQ & trigger group has started creating hardware prototypes that will meet the requirements of the accelerator's added performance.

The current data acquisition pipeline uses a plethora of custom hardware and FPGAs (Field Programmable Gate Arrays). This hardware is controlled and monitored by rack-based server PCs (personal computers). The hardware is connected to the PCs through PCI bridges[1]. The upgrade of the DAQ system gave an opportunity to improve the control and monitoring as well. The aforementioned hardware prototypes, for the upgrade, will include an embedded system that will perform the control and monitoring tasks on the hardware boards themselves. This eliminates the use of additional racks with PCs and cabling.

The embedded system on the prototypes is the *Zynq Ultrascale+ MPSoC* from Xilinx (Multiprocessor Systems on a Chip). A simple block diagram of the chip can be seen in Figure 2.1:

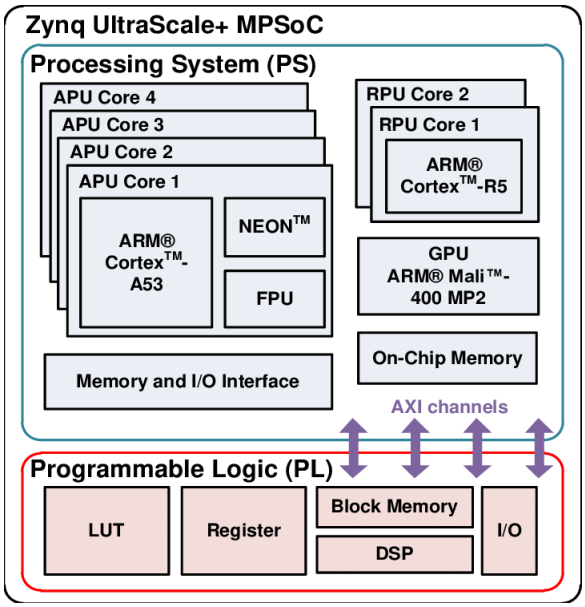


Figure 2.1: Block diagram of the Zynq Ultrascale+ MPSoC with the main components of the processing system [23].

This MPSoC consists of a processing system (PS) and programmable logic (PL/FPGA). The processing system contains a quad-core ARM processor that is capable of running an embedded Linux distribution with control and monitoring software. The Zynq MPSoC will allow control and monitoring very close to the hardware, which was not possible before.

Other experiments are also planning to use or are already using the chip from Xilinx. The HL-LHC upgrade of the CMS experiment will host several thousand embedded controllers using the Zynq MPSoC.

[1]Peripheral Component Interconnect is a local bus standard for connecting hardware devices to each other. A PCI bridge allows connections between multiple independent PCI buses [22].

2.2 Objective

The multi-core processor in the Zynq MPSoC will run a Linux operating system (OS). The chip will run CentOS 8 with a set of additional services required by the embedded system. This is currently being developed by the CMS DAQ team. CentOS is a re-branded version of Red Hat Enterprise Linux (RHEL). It aims to be compatible with RHEL while removing vendor branding and making the distribution free to use. Both RHEL and CentOS are often used in server applications [24].

Booting a Linux OS on the Zynq MPSoC involves a complicated multi-stage booting process [25]. This booting process is further studied in Section 3.2. Moreover, the complexity of the boot-up can introduce possible hardware or firmware failures that can prevent the chip from booting correctly. This results in a state where the chip hangs, making it difficult or impossible to debug. The users of the board can analyze the booting failure once the board is brought to a *well-known* booted state.

The aim of the project is to create a reliable booting system, which brings the chip into a well-known booted state after a failure. This system will consist of multiple fallback mechanisms. The fallbacks will prevent the chip from hanging, and recover from failures by booting it back into a well-known state of Linux. Besides, the system needs to inform the user about the problem if possible. The reliability requirement of the booting system is given in Section 2.4.

2.3 Requirements & Preconditions

The requirements for the reliable booting system (software and firmware) are stated in Table 2.1. Their priorities have been described using the MoSCoW method. The MoSCoW method features the following categories [26]:

Must have: These requirements are critical and must be met by the final product for the project to be successful.

Should have: These requirements are important, but not indispensable. The final product can still work without meeting these requirements.

Could have: These requirements are desirable, but not necessary (nice to have). If there is enough time and budget left, these requirements can be met.

Won't have: These requirements are not part of the project. They could be introduced in a follow-up project. This project does not have any "won't have" requirements.

Table 2.1 Project requirements.

#	Requirement	MoSCoW
1	The Zynq MPSoC can recover from a boot failure.	Must
2	The Zynq MPSoC can recover from a failed upgrade.	Must
3	The Zynq MPSoC can recover from a running failure.	Must
4	The Zynq MPSoC boots through the network by default.	Must
5	The reliable booting system is portable to new hardware.	Must
6	Each fallback reports to the user about a failure *if possible*.	Should
7	The Linux distribution and fallbacks are automatically built by a CI (Continuous Integration).	Should
8	The Linux distribution and fallbacks are automatically tested by a CI (Continuous Integration).	Could

The preconditions for the project are stated in Table 2.2. The preconditions concern all matters related to the final product (reliable booting system), but not the final product itself.

Table 2.2 Project preconditions.

#	Precondition
1	The project is developed on the Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit.
2	The project is developed using Xilinx Vivado 2019.2 and PetaLinux Tools 2019.2.
3	The Zynq MPSoC will sit on an ATCA board and run in an ATCA crate.
4	The project is tracked using GitLab.
5	The project uses official development tools provided by CERN.
6	The project must integrate with the services available in the CMS experiment network [21].
7	The board, hosting the Zynq Ultrascale+ MPSoC, is replaced when a hardware failure prevents the chip from booting.

2.4 Reliability requirement

A fully reliable booting system requires fallbacks for every failure that can occur in the Zynq MPSoC. It is difficult to know beforehand which failures the Zynq MPSoC will experience. One can speculate which failures are most probable, but it is not possible to predict all of the possible failure scenarios. This is why a truly *"reliable"* booting system is not achievable. The booting system must therefore be *sufficiently reliable*. It must recover the Zynq MPSoC from a boot failure, an upgrade failure and a running failure. Each failure that is specified in the requirements is equally important to solve. The details of each failure are researched in Chapter 4. Any other failures can be resolved by using a watchdog timer, often found in embedded devices [27]. It has been researched and found that the Zynq MPSoC contains such a hardware timer (see Section 3.3).

The Zynq MPSoC hardware must be setup correctly to boot. The booting system is not responsible for recovering the Zynq MPSoC from any hardware failures. The Zynq MPSoC boards are accessible and hardware failures can be resolved by replacing the board. This is in contrast to an application in space (e.g. a satellite) where the hardware is inaccessible and triple redundancy is often implemented [28].

The amount of fallbacks must be *as low as reasonably achievable*. Tradeoffs must be made between fallbacks. This is discussed further in Subsection 4.4.1.

2.5 Final products

At the end of the project, a reliable booting system will be delivered that is compliant with the requirements. All the requirements with a *must* priority have to be met. The reliable booting system will be tested to confirm if it meets the requirements.

A list of final products reads:

1. A reliable booting system, consisting of firmware and software, that can recover the Zynq MPSoC from the following failures:
   (a) Boot failure;
   (b) Failed upgrade;
   (c) Running failure;
2. GitLab CI that automatically builds the Linux distribution and fallbacks for the Zynq MPSoC.
3. GitLab CI that automatically tests the Linux distribution and fallbacks for the Zynq MPSoC (this final product has a *could* priority and will only be delivered if time is available).
4. Documentation on the reliable booting system in GitLab.

3. Background research

This chapter present the background research on the internals of the Zynq MPSoC and its booting process. It also focuses on the watchdog timer hardware in the Zynq MPSoC, and the workings of the crashkernel - a mechanism for collecting memory dumps after a system crash.

3.1 Zynq MPSoC workings and internals

3.1.1 Zynq MPSoC booting overview

The booting process of the Zynq MPSoC is split up into multiple stages in which several parts of hardware and firmware get initialized and loaded [29]. Figure 3.1 shows an example of the boot flow in the Zynq MPSoC.

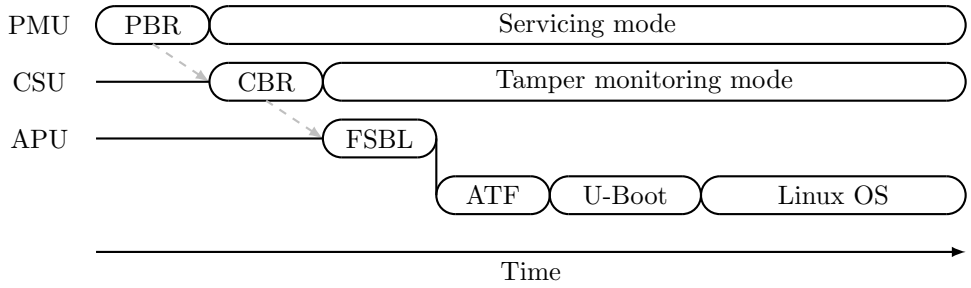


Figure 3.1: Generalized boot flow of the Zynq MPSoC [30].

The booting process in Figure 3.1 can be split up into three main stages [25]:

1. The *pre-configuration* stage is handled by the Platform Management Unit (PMU), which performs system initialization through the PMU BootROM code (PBR). The PBR resets and initializes a part of the processing system (PS). It also prepares the Configuration Security Unit (CSU) for the configuration stage. After initialization, the PMU releases the reset of the CSU and enters a servicing mode.

2. The *configuration* stage is handled by the CSU, which runs the CSU BootROM code (CBR). The CBR further initializes the processing system and determines the boot mode of the chip. It searches and loads a boot image containing the first-stage bootloader (FSBL) into on-chip memory. It also has a possibility to load the PMU firmware. After loading the FSBL, the CSU enters a tamper monitoring mode.

3. The *post-configuration* stage consists of multiple sub-stages that lead to a running Linux OS. These sub-stages are handled by the first-stage bootloader (FSBL) and second-stage bootloader (U-Boot), which run on the Application Processing Unit (APU). The first stage bootloader initializes the FPGA, double data rate memory (DDR) and APU. It also loads the ARM Trusted Firmware (ATF) and the second-stage bootloader, which is U-Boot.

The PMU, CSU, and APU are the required processing units to boot Linux on the Zynq MPSoC. Besides, some I/O peripherals are also necessary during the booting process [25, 29]. The required hardware components for booting are described in the following sub-sections.

3.1.2 Zynq MPSoC hardware overview

The Zynq MPSoC contains multiple processor units, I/O peripherals, and an FPGA. The chip is split up into two parts: the processing system and the programmable logic.

The PS houses all the processing units and I/O peripherals [31]. It also contains an on-chip memory and an interface for connecting external DDR memory. The quad-core ARM Cortex-A53 Application Processing Unit (APU) is the main component in the PS. The APU is accompanied by a Real-time Processing Unit (RPU) and a Graphics Processing Unit (GPU). The RPU and GPU will not be used in this project. The processing system also houses the Platform Management Unit (PMU) and Configuration Security Unit (CSU). Figure 3.2 shows a block diagram of the Zynq MPSoC and its components:

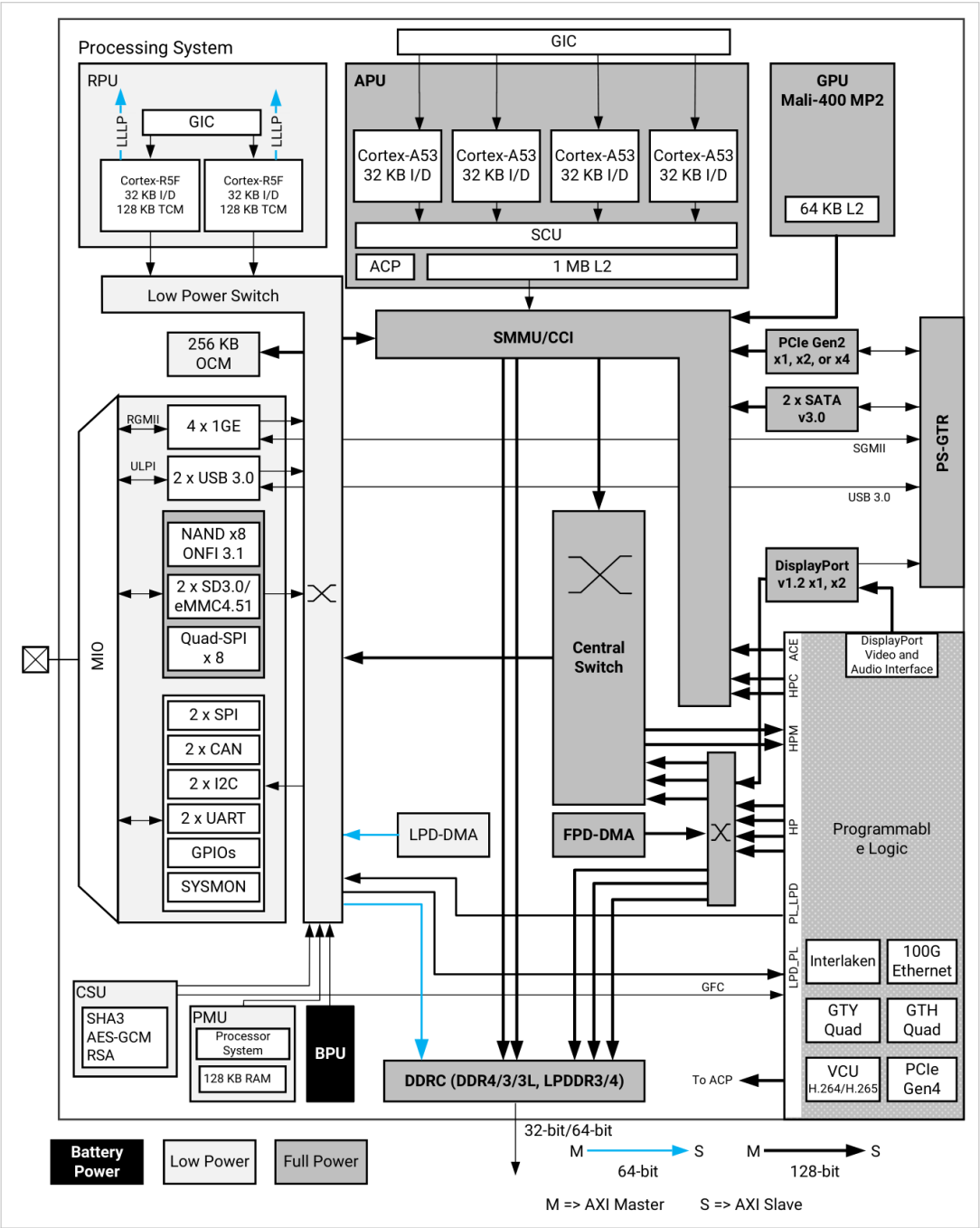


Figure 3.2: Block diagram of Zynq Ultrascale+ MPSoC hardware architecture [32].

3.1.3 The Application Processing Unit (APU)

The primary role of the Application Processing Unit (APU) is running the first- and second-stage boot-loaders, and finally Linux. The APU has the following specifications [31]:

- Quad-core ARM Cortex-A53 processor
- CPU frequency up to 1.5 GHz
- Aarch64 architecture (also known as ARM64)
- 32 kB L1-cache per processor and a shared L2-cache (1 MB)
- Floating-point unit (FPU) and cryptography extension

The APU has two levels of cache[1]. Each core has a local L1-cache. Other cores cannot access this cache. The L1-cache is split up into I-cache for instructions and D-cache for data. On top of that, there is L2-cache, which is shared between the cores (see Figure 3.3). It has more memory but is slower than the L1-cache [34]. The Snoop Control Unit (SCU) in the APU takes care of cache coherence[2] [34] and connects the two levels of cache.

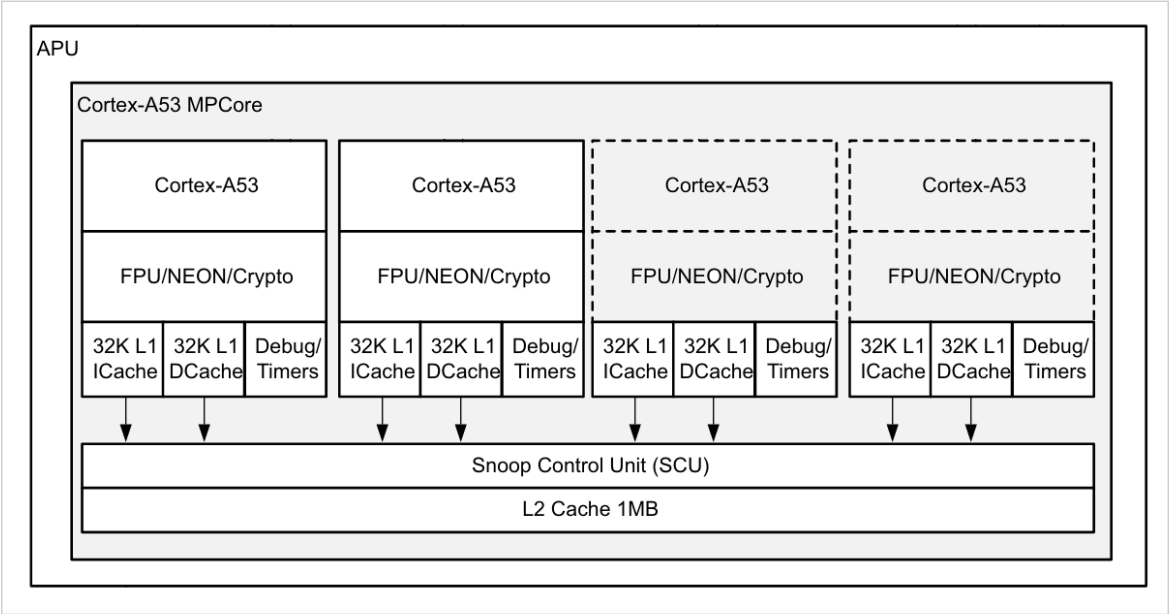


Figure 3.3: Block diagram of the Zynq Ultrascale+ MPSoC application processing unit.

The APU is connected to DDR memory through the System Memory Management Unit (SMMU)(see Figure 3.2). The SMMU performs translations from virtual memory addresses to physical memory addresses. It also makes sure that only one processor can take control of the memory bus at a time (memory arbitration). Besides, it performs memory protection so that each processor can only access the memory which was allocated to it [34].

The APU can access the other parts of the chip through the central switch. It also uses the low power switch to access I/O peripherals, the on-chip memory, the CSU, and the PMU. These switches can be seen in Figure 3.2.

3.1.4 I/O peripherals and interfaces

The Zynq MPSoC provides a range of I/O peripherals and interface options. These provide connectivity, access to external storage, and high-speed connections. The I/O peripherals are essential to the booting process. They are used by the CSU BootROM code, the first-stage bootloader, the second-stage bootloader, and Linux. The I/O peripherals are summarized in Table 3.1 [32]:

[1] A cache is a small amount of high-speed memory that is located close to the processor core. It is intended to store data and instructions from RAM that are used frequently by the processor [33].

[2] Cache coherence refers to the problem of keeping the data in multiple levels of cache consistent [35].

Table 3.1 I/O peripherals and interfaces.

Connectivity		External storage	High speed
1. SPI	5. USB	1. NAND	1. PCI Express v2.1
2. I^2C	6. GEM (ethernet)	2. Quad-SPI (QSPI)	2. SATA 3.0
3. UART	7. GPIO	3. SD-card	3. Display Port
4. CAN		4. eMMC	4. USB 3.0

The controllers for connectivity and external memory are accessible through the low-power domain (LPD)(see Figure 3.2). These peripherals are connected through the multiplexed I/O (MIO) interface. This allows the peripherals to be connected to any external pin with MIO capabilities. The MIO interface also allows the peripherals to be connected to the PL [36].

The MPSoC also supports high-speed interfaces for PCI Express, SATA, Display Port, and USB 3.0. These interfaces reside in the full-power domain (FPD). The PL also includes integrated blocks for PCI express and 100 Gb/s ethernet. The high-speed interfaces are not used in the project.

3.1.5 The Platform Management Unit (PMU)

The Platform Management Unit (PMU) uses a triple redundant MicroBlaze processor that handles initialization of the system, power management, execution of self-tests, and system error management [37]. The initialization of the system is done through the PMU BootROM code. This code is stored in a separate ROM that's part of the PMU.

The other functionalities of the PMU are handled by the PMU firmware. The firmware is split up into multiple blocks. These blocks consist of APIs and modules. Figure 3.4 shows a block diagram of the PMU firmware:

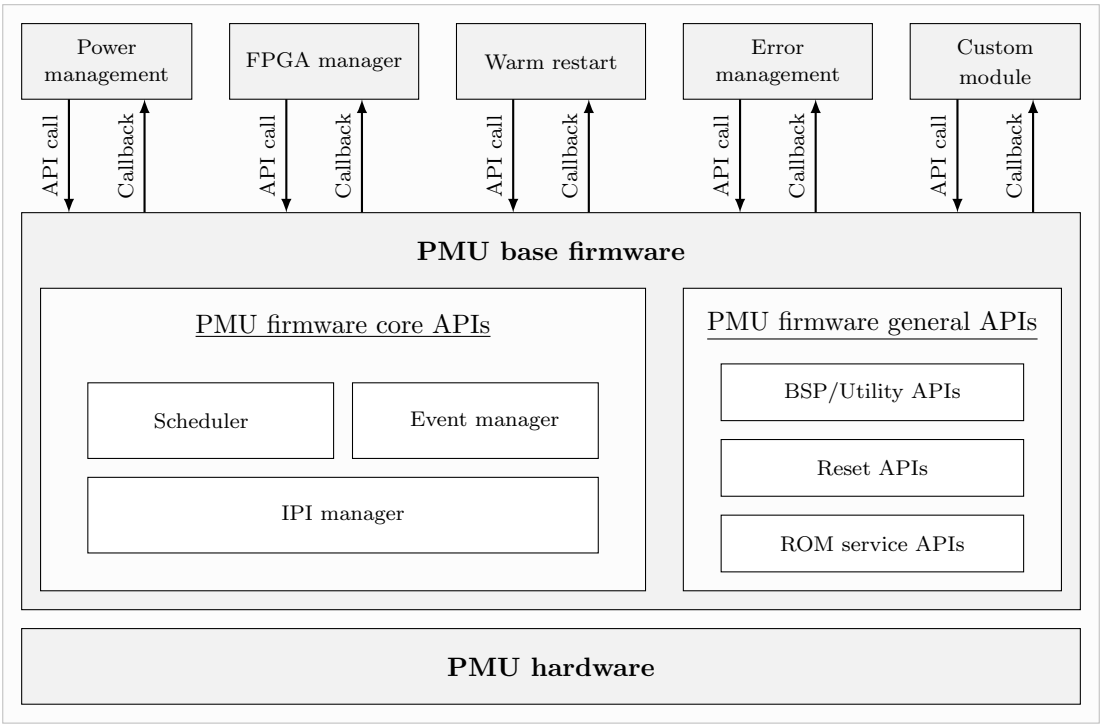


Figure 3.4: Block diagram of the PMU firmware, showing the base firmware and modules [38].

Modules use the APIs, provided by the PMU base firmware, to execute tasks and functions. The PMU firmware core APIs are essential to the modules. They provide access to the scheduler, event manager and inter-processor interrupt manager (The IPI handles interrupts that are sent between processing units).

There are several modules available for the PMU. An example is the power management module, which is enabled by default. This module is responsible, among other things, for switching the different blocks of the Zynq MPSoC on and off and managing memories and peripherals. [39]

Each module can be enabled or disabled by the user. This gives the PMU firmware modularity. It is also possible to create a custom module [39]. Adding a custom module to the PMU firmware *could* provide a fallback for the reliable booting system. This should be further investigated.

3.1.6 The Configuration Security Unit (CSU)

The Configuration Security Unit (CSU) is a processor that handles the device security of the Zynq MPSoC. It is separated into two blocks. The Secure Processor Block (SPB) and the Crypto Interface Block (CIB) (These can be seen in Figure 3.5). The primary functions of the CSU are secure booting, tamper monitoring, and key storage & management [40].

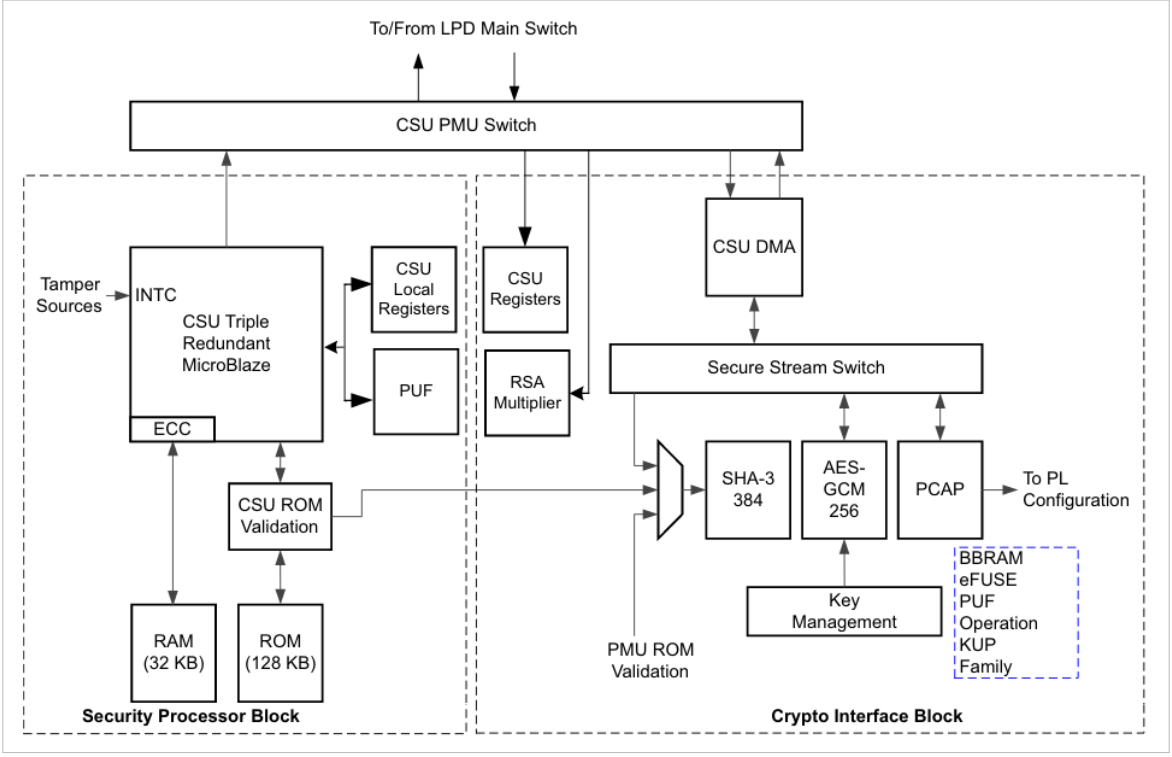


Figure 3.5: Block diagram of configuration security unit in the Zynq MPSoC [40].

The secure processor block uses a triple redundant MicroBlaze processor with an internal, uninterruptible clock source and internal RAM. It also has a ROM which holds boot code that is executed during the booting process (also known as the CSU BootROM code). The other part of the CSU is the crypto interface block. This block features three cryptographic cores that can be used during the booting process for authenticating and decrypting boot images.

After running the CSU BootROM during startup, the CSU secure processor block will enter a tamper monitoring mode. In this mode, the CSU can detect *"tamper"* events. These events can be triggered when, among other things, voltages or temperatures in the chip suddenly change, or when the JTAG interface is used.

The anti-tampering features of the CSU are not of interest to this project. The Zynq MPSoC devices will be running on a dedicated CMS network. The network is protected and has limited access.

3.2 Zynq MPSoC booting process

The overview in Subsection 3.1.1 gave a generalized explanation of the booting process. This section will give a more detailed description of each step in the booting process.

3.2.1 The PMU BootROM

The PMU hardware includes a ROM that holds boot code for the PMU. This boot code is the first task that gets executed when the Zynq MPSoC is powered up. The code executes a list of tasks that initialize the PMU and CSU, and configure several parts of the hardware in the chip (listed in Table 3.2) [41].

Table 3.2 PMU BootROM tasks.

Task	Description
1	Initialize the MicroBlaze processor of the PMU.
2	Setup the clocks used by the Memory Built-In Self Test (MBIST).
3	Initialize the PS monitoring system to monitor temperatures and voltages (PS SYSMON).
4	Scan and clear the low-power and full-power domains, and perform memory self-tests.
5	Zero the PMU RAM, CSU RAM, and memories & registers in the LPD and FPD.
6	Validate the power supply of the processing system.
7	Release the CSU reset or enter an error state.

If all tasks are run successfully, the PMU will validate the immutable CSU BootROM code (CBR). It will send the CBR through the cryptography engine of the CSU (SHA-3/384) and compare the checksum to the golden copy that is stored in the CSU ROM. If the cryptographic checksums match, the CBR code is validated and the reset to the CSU is released. The PMU enters a servicing mode after releasing the reset to the CSU [25]. In the servicing mode, the PMU handles the power-up and power-down of domains in the PS, enabling and control of the built-in self-repair (BISR), and resetting of blocks. The ROM code can also execute the PMU user firmware. This concludes the pre-configuration stage.

3.2.2 The CSU BootROM

The main objective of the CSU BootROM is to load the FSBL into the on-chip memory. Table 3.3 shows a set of tasks, performed by the CSU to achieve this [25]. After initializing the on-chip memory, the value of the *boot mode register* is read to determine which storage device should be searched to find the FSBL.

Table 3.3 CSU BootROM tasks.

Task	Description
1	Initialize the on-chip memory.
2	Determine the boot mode by reading the boot mode register.
3	Perform image search to find the boot image and boot header.
4	Read and interpret the boot header of the boot image.
5	Initialize the required PS device (either the RPU or APU).
6	Load the FSBL into the on-chip memory
7	Authenticate and decrypt the FSBL if configured in the boot image.
8	(*Optional*) Load the PMU user firmware into the PMU RAM.

The Zynq MPSoC on the ZCU102 development board supports booting from QSPI, SD-card, JTAG and USB [25, 29]. The boot mode is set through the bootstrapping pins of the chip. The state of these pins is captured in the register when the chip is powered on.

The FSBL and PMU firmware are stored in a *boot image*. The image has a pre-defined format containing a boot header, partition header, and one or more partitions (see Figure 3.6). The boot header describes boot parameters, characteristics, and details about the boot image. It has a size of 32 kB and is the first data that the CSU will look for. The partition header describes how the partitions in the image are defined. There is always one partition for the FSBL image. Other images, like for the PMU firmware, are optional [42].

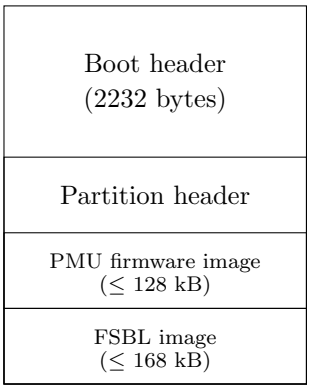


Figure 3.6: Boot image format containing the FSBL and PMU firmware [42].

The CSU searches the *boot device*, described in the boot mode register, to find the boot header of the boot image. To find a boot header, it uses the *golden image search mechanism* [43]. A boot header can be located at every 32 kB in storage. This allows for multiple boot images to be stored in the same storage device. The CSU will try to read the *identification string* "XLNX" in the boot header. If it is unsuccessful in reading this string from memory, it will offset the reading address by 32 kB and try again. If an SD-card is used as a boot device, the offset value in the `CSU_MULTI_BOOT` register is converted into a string. The offset string is then concatenated with `BOOT.BIN` (the filename of the boot image) to get a new filename.

The CSU will continue to use the golden image search until it finds a valid identification string. Figure 3.7 shows a flowchart of the golden image search mechanism:

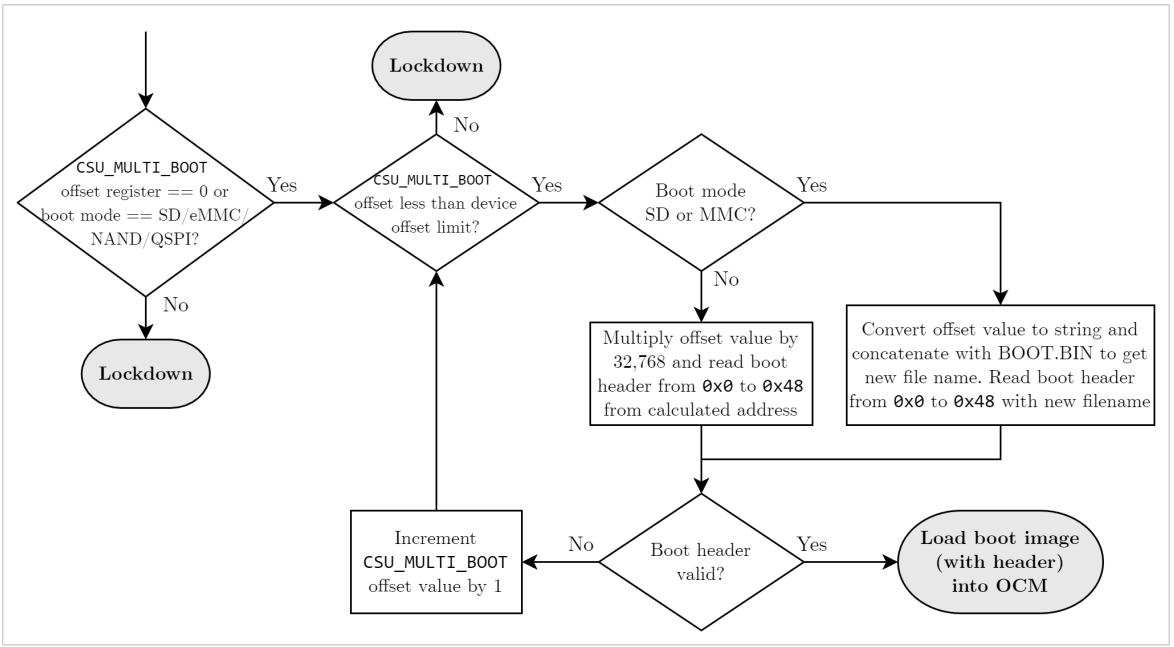


Figure 3.7: Flowchart of golden image search mechanism in the CSU BootROM [43].

If the string is found, it will validate the boot header checksum. Then it continues with the initialization of the PS device on which the FSBL will run (usually the Arm Cortex-A53 APU). Finally, it will load the FSBL image into the on-chip memory (OCM). The configuration stage concludes when the CSU releases the reset of the APU and the FSBL takes control [41].

3.2.3 The first-stage bootloader (FSBL)

The first-stage bootloader (FSBL) starts with authenticating the rest of the boot image. If it finds that the image is corrupted is some way, it will offset the boot header search address of the CSU BootROM by modifying the CSU MultiBoot register. Then it will generate a soft reset. The next time the CSU BootROM runs, it will use this offset to search for another boot header. This is called the MultiBoot mechanism [44].

The main goal of the FSBL is to load the second-stage bootloader, which is U-Boot. To reach this goal, the FSBL has to go through four stages [45]. These stages are shown in the flow diagram of the FSBL (see Figure 3.8):

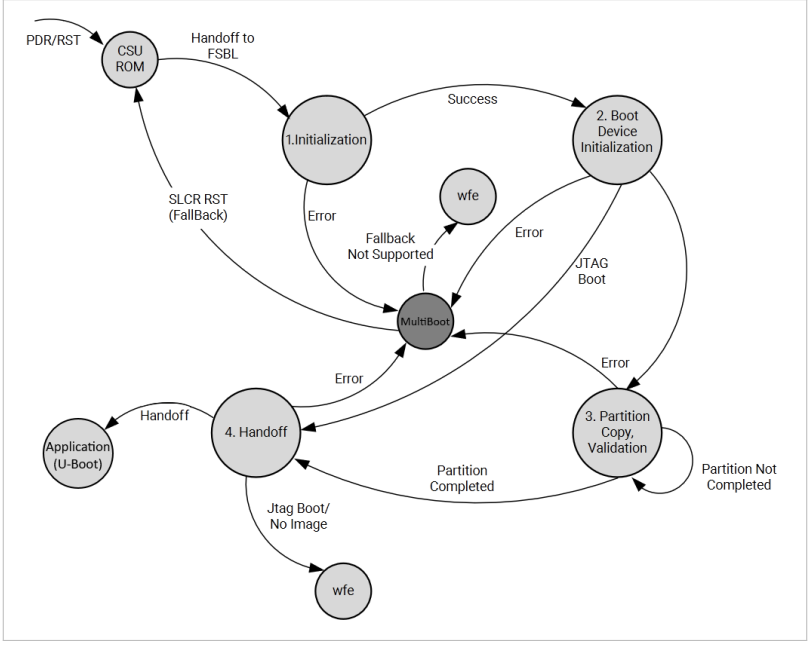


Figure 3.8: Flow diagram of the FSBL and its different stages [45].

1. **Hardware initialization:** The FSBL initializes the programmable logic (PL/FGPA), the processor, and the DDR memory.
2. **Boot device initialization:** The FSBL reads the boot mode register and identifies the primary boot device. This boot device gets initialized using the appropriate boot device driver[3]. Then, the boot header of the boot image is validated and interpreted. Lastly, the FSBL sets some initialization parameters for the ARM trusted firmware (ATF, see Subsection 3.2.4).
3. **Partition copy validation:** The FSBL will validate the partition header. It will then continue to copy every partition to memory. The PMU firmware partition gets copied directly to the PMU RAM. The ATF gets copied to the on-chip memory. The U-Boot image gets copied to DDR memory.
4. **Handoff:** The last stage of the FSBL is handing off control to U-Boot. Before doing so, the ATF is initialized. Finally, the program counter is updated for U-Boot to take control.

If there is an error during any stage of the FSBL, the bootloader will try to use the MultiBoot mechanism as a fallback to boot from another image. If the mechanism is not supported by the boot device, the FSBL will hang in the WFE[4] (Wait for Event) instruction of the processor [45].

[3]Each boot device driver provides initialization, copy, and release functions [45].

[4]WFE supports multiple wake-up events, one of which is the execution of the SEV (set event) instruction. This instruction will cause an event to be signaled to all processor cores. The SEV instruction must be executed by any of the other processor cores [46].

3.2.4 The ARM trusted firmware (ATF)

The ARM trusted firmware (ATF) starts after the FSBL and acts as a proxy to modify system-critical settings. Linux is considered as a non-secure OS and can therefore not access these settings directly [47]. The ATF grants the OS access to power management, secure monitor calls, clock management, the reset domain, etc. These settings are mostly managed by the CSU and PMU. The access restrictions are related to the *exception level model* of the APU (see Figure 3.9) [48].

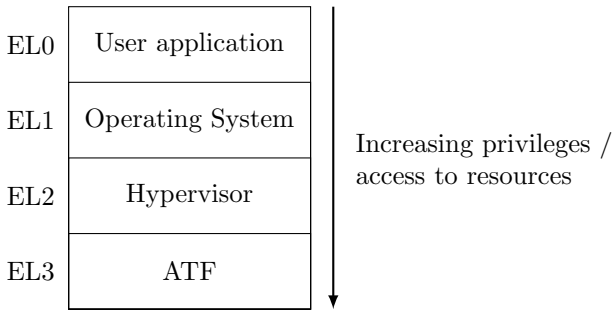


Figure 3.9: Exception level model of the ARM Cortex-A53.

User applications, running at EL0, have almost no access to system resources. In contrast, the ATF runs at EL3. This means it has the highest amount of access to the resources of the chip. In between, there is the OS and the hypervisor[5]. A hypervisor is not used, which leaves the OS (in this case Linux), running at EL1. After its initialization by the FSBL, the ATF is ready to be used by U-Boot and Linux. U-Boot starts after the FSBL and runs at EL2 [49].

3.2.5 The second-stage bootloader (U-Boot)

U-Boot is a universal bootloader and is responsible for booting the Linux OS on the Zynq MPSoC. It is a powerful second-stage bootloader with many capabilities.

U-Boot provides a command-line interface (CLI) on the serial port of the Zynq MPSoC (shown in Figure 3.10). The CLI offers commands for reading and writing flash memory, device-tree[6] manipulation, downloading files through the network, communicating with hardware, etc [51]. It even offers the use of environment variables, which can store sequences of commands [52, 53]. On top of that, it can also run Hush shell scripts [54].

```
1 U-Boot 2019.01 (Sep 21 2020 - 17:21:27 +0000)
2 Model:                CMS ZCU102 Development board
3 Board:                Xilinx ZynqMP
4 DRAM:                4 GiB
5 EL Level:        EL2
6 ...
7 U-BOOT for CERN CMS
8 Hit any key to stop autoboot:  0
9 ZynqMP> _
```

Figure 3.10: U-Boot startup messages when booting a Zynq MPSoC. Here the automatic booting process is interrupted and U-Boot drops down to its CLI.

Depending on U-Boot's configuration, the Linux kernel can be booted in several ways. By default, the Zynq MPSoC boots from a local storage device. This means that the kernel image, device-tree blob (DTB)[6], and root filesystem are stored locally. For example on an SD-card.

[5] A hypervisor enables multiple operating systems to be run simultaneously on the same processor [50].

[6] The device-tree is essential to a Linux OS and is explained in Subsection 3.2.6. The compiled version of a device-tree is called a device-tree blob (DTB).

U-Boot also supports booting through the network. It can retrieve the kernel image and DTB from a TFTP server[7] (Trivial File Transfer Protocol) [55]. The Linux root filesystem can also be stored on a server and accessed through NFS (Network File System) [56]. It is also possible to use a ramdisk during booting. A ramdisk is a small filesystem in RAM that is mounted during booting. It is used to initialize various parts of the system, before switching to the root filesystem.

Regardless of the booting method, U-Boot will load the kernel image and the DTB into memory (and possibly also a ramdisk). Then, U-Boot will pass the boot arguments to the kernel and hand over control.

3.2.6 Kernel booting

Once the kernel takes control, it will try to extract itself from its compressed version[8] [58]. The kernel will use the device-tree to identify the hardware it's running on. It also uses the device-tree to access hardware on the chip [59].

Mounting the root filesystem is one of the first tasks of the kernel [56]. Once the kernel has mounted the root filesystem, it will look for the init process. The init process has a process ID of 1 (PID1) and handles the startup and shutdown of the system [60]. CentOS 8, which will run on the Zynq MPSoC, uses systemd[9] as its init system. Systemd will start all the processes and services. This concludes the booting process.

3.2.7 Booting process summary

As seen in the previous sub-sections, each stage in the booting process contains multiple sub-stages and steps. Because the chip has multiple processors and other hardware components, it requires more steps to boot than e.g. a microcontroller that is running an RTOS (real-time operating system). To summarize, Table 3.4 shows the steps that the Zynq MPSoC goes through to boot up Linux.

Table 3.4 Summary of the Zynq MPSoC booting process. Also see flowchart in Appendix A.

Step	Description
1	Chip powers on, boot mode register captures the bootstrapping pins.
2	PMU BootROM code starts. Initialization of essential hardware components in PS.
3	CSU BootROM code starts. Searching for boot image and verifying the boot header.
4	CSU loads the first-stage bootloader into the OCM.
5	FSBL initializes DDR memory, the FPGA (PL), and other hardware components.
6	FSBL Loads the ATF and U-Boot into DDR memory.
7	FSBL initializes the ATF and hands control to U-Boot.
8	U-Boot loads the kernel image and device-tree blob into DDR memory.
9	Kernel boots up and performs hardware initialization using the device-tree.
10	Root filesystem is mounted by kernel.
11	Systemd service manager is started.

The analysis of the Zynq MPSoC booting process shows where possible fallbacks can be implemented. The BootROMs for the PMU and CSU cannot be changed to implement a fallback. The images for the FSBL, U-Boot and Linux *can* be changed to host a fallback though. The images can be created by using the PetaLinux tools (more on this in Section H.1). Further investigation on possible failures is needed to see how fallbacks can be implemented in the FSBL, U-Boot and Linux (see Section 4.2).

[7] TFTP is a simple protocol for exchanging files through the network. It is typically used for downloading boot images to remote devices [57].

[8] A kernel is usually saved in a compressed format, and therefore it has a self-extracting capability.

[9] Systemd is a service management system that controls services that are running in the userspace of the OS [58, 61].

3.3 Zynq MPSoC watchdog timers

A watchdog timer (WDT) is a hardware timer that automatically generates a reset if an application, running on the Zynq MPSoC, neglects to periodically service it [27]. Watchdog timers are often present in embedded devices. In Linux, there should be a daemon running that periodically restarts the timer. If Linux crashes, the daemon stops running and eventually the system would get reset by the watchdog. The watchdog timer is an essential mechanism that will increase the reliability of the booting system. The mechanism can be used to recover the Zynq MPSoC from a hang.

3.3.1 Watchdog timer workings

The Zynq MPSoC has three watchdog timers that can be used to reset the system if it hangs. Each watchdog timer guards a different part of the chip. The three watchdog timers are:

1. The low-power domain watchdog timer (LPD WDT). This watchdog timer is mainly used to reset the Real-time Processing Unit (RPU) of the Zynq MPSoC. This part of the chip is not used during the project and the LPD watchdog timer will therefore not be used.
2. The full-power domain watchdog timer (FPD WDT). This watchdog timer is mainly used to reset the APU. The APU is used to run the FSBL, U-Boot, and Linux OS. It can be used if the Zynq MPSoC hangs in any part of the booting process after the FSBL. This watchdog timer has a default expiry time of 60 seconds. The watchdog timer duration can be changed by modifying the PMU firmware configuration.
3. CSU watchdog timer (CSU WDT). This watchdog timer is used to reset the PMU if the PMU firmware hangs for some reason. This watchdog timer is handled by the CSU.

When booting up, the FSBL will initialize and start the watchdog timers [45]. The FPD watchdog timer will be configured to generate an IPI (inter-processor interrupt) to the PMU when it expires. The PMU can handle the FPD watchdog timer error through a recovery mechanism in the PMU firmware. The recovery mechanism is part of the Error Management (EM) module and can be added to the PMU firmware by compiling it with certain build flags [38, 62].

When the recovery mechanism is enabled, the PMU firmware will run a handler that resets the APU. The diagram in Figure 3.11 shows how the APU gets reset after a watchdog timer expiry.

The PMU firmware will restart the watchdog timer and generate an interrupt to the ATF to idle the APU cores. After clearing and idling each core of the APU, the ATF will generate an interrupt for the PMU to perform a reset of the APU.

If enabled, the PMU firmware will also perform *escalation*. The escalation scheme is used to reset the processing system (PS) if the ATF is not able to idle all the APU cores [62]. The scheme uses a `WDT_In_Progress` flag that is set when a watchdog timer expires. The flag gets cleared if the ATF idles all APU cores, and sends a request to the PMU to reset the APU. If the ATF is not able to clear the APU cores, the watchdog timer will expire a second time. The PMU firmware will check if the `WDT_In_Progress` flag was already set and will trigger escalation if it was.

If the Zynq MPSoC continuously fails to boot-up Linux, the watchdog timer expiries will reset the processing system indefinitely. The PMU firmware has no knowledge of the infinite reset cycle. The *"healthy bit scheme"* can help with this [62]. The PMU firmware can check if the last boot attempt was successful by checking a *healthy bit* in one of the PMU registers (the diagram for this is shown in Appendix F.1). This bit will be set by a Linux application if the system boots successfully. If the bit is not set to one, it will indicate an unsuccessful previous boot and trigger a system reset (SRST, reset of both the PS and the PL).

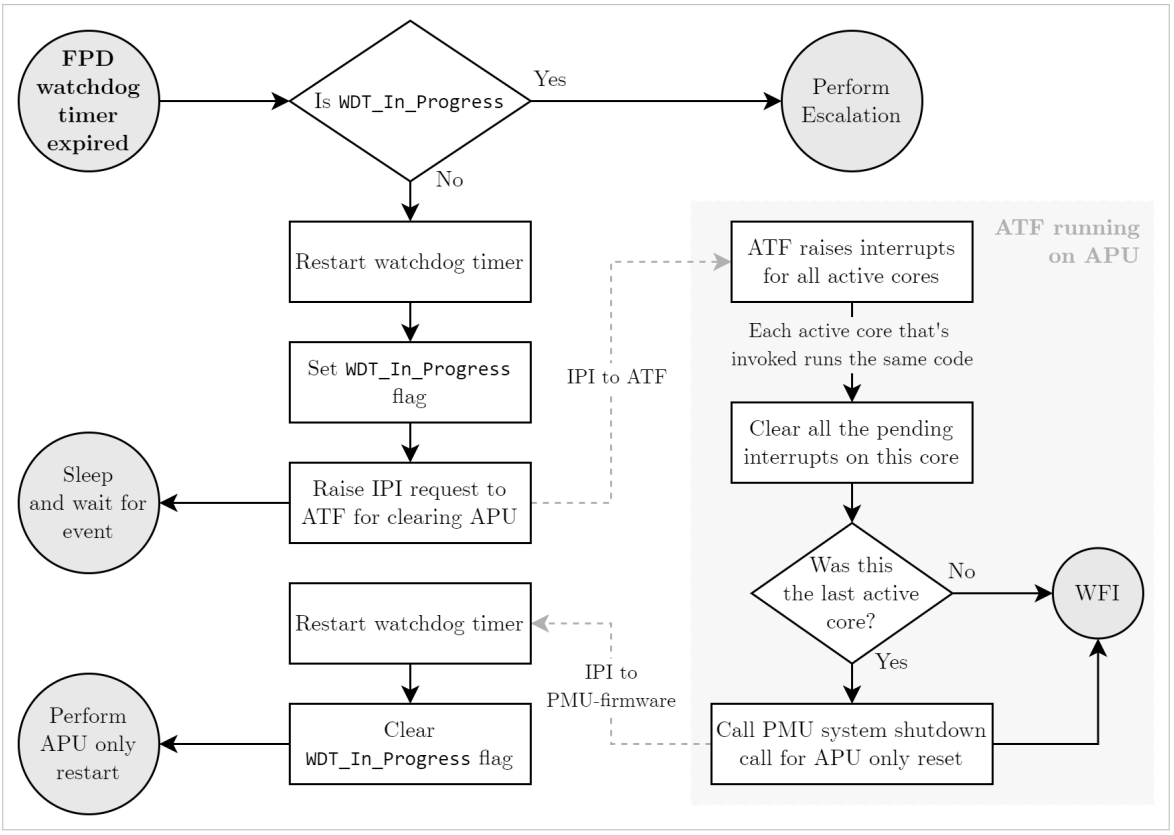


Figure 3.11: Diagram of PMU firmware watchdog timer handling and reset of the APU [62].

3.3.2 Watchdog timer heartbeat daemon in Linux

Once Linux has successfully booted, the watchdog timer should not expire and trigger a reset. The watchdog timer will have to be restarted periodically by using a *heartbeat daemon*[10]. Xilinx provides two examples of restarting the watchdog timer [62]. One uses the Busybox `devmem` utility to write a restart value to the watchdog timer register. The other example uses C and a set of libraries to access the watchdog timer register (the source code can be found in Appendix F.2).

3.4 The Linux crashkernel

The *"crashkernel"* or *"dump-capture kernel"* is a feature that can be enabled in CentOS 8. The mechanism boots a second kernel on top of a crashed kernel [64]. The second kernel will copy the contents of the system's memory into a dump file and store it locally or remotely. This dump file can later be used for post-mortem analysis of the crash. The crashkernel will also reset the system after it has finished.

3.4.1 Crashkernel workings

The crashkernel, or dump-capture kernel, uses *kdump* as a crash dumping mechanism to save the memory content of the system after a kernel panic[11]. Furthermore, it uses the *kexec system call* to load a second kernel (the crashkernel) into a reserved section of memory [65]. This second kernel is booted on top of the main system kernel without the need of a bootloader or hardware initialization [66, 67]. The effect of this is a reduced boot time for the second kernel.

[10] A daemon is a Linux process, that runs in the background. Daemons are often started at boot-up by systemd [63].

[11] Panic is a kernel function that prints an error message and then halts the system [68]. It is used as a critical error mechanism to stop the kernel.

On boot-up, the main system kernel will reserve a specific amount of memory that is required by the crashkernel (see Figure 3.12). The memory is used by the crashkernel image and its initramfs (Initial ramdisk which is used as part of the Linux startup process).

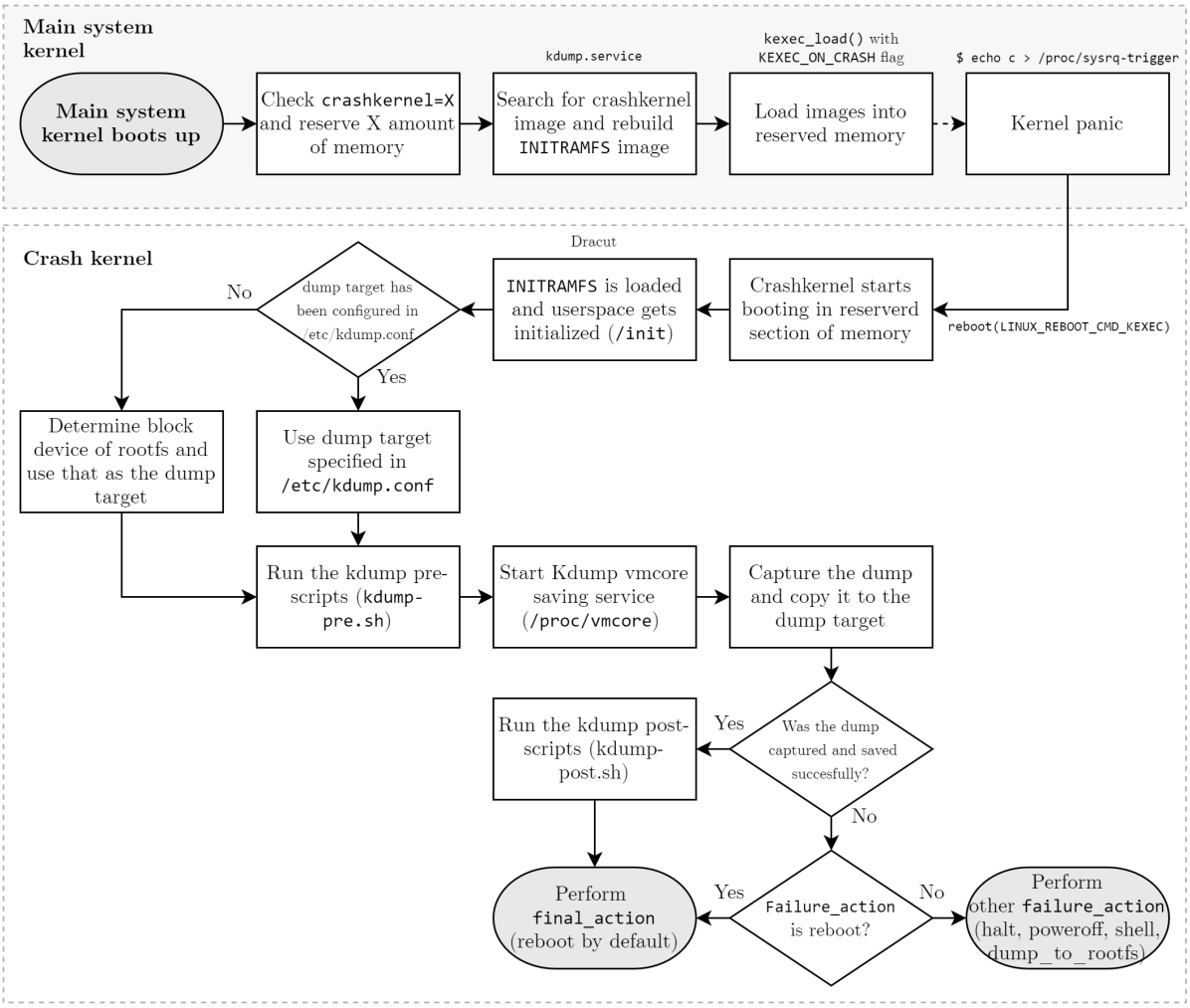


Figure 3.12: Flowchart of crashkernel workings.

Systemd will start the *kdump service* to load the crashkernel image and its initramfs into the reserved section of memory [65]. It does this by using the *kexec-tools*. The crashkernel image is loaded into memory in advance, because at the moment of a crash there will be no way to read data from the disk.

The crashkernel image is loaded using the `kexec_load()` system call. The `KEXEC_ON_CRASH` flag is passed to the system call to avoid the crashkernel from starting immediately. This flag instructs the system call to boot the crashkernel automatically on a crash [69].

When the main system kernel crashes, the panic function will use the `reboot()` system call. It will pass the `LINUX_REBOOT_CMD_KEXEC` flag to instruct the kernel to boot the crashkernel image. Once the crashkernel starts booting it will mount the initramfs and start a kdump service in the ramdisk. The service will search for the `kdump.conf` configuration file to identify the *dump target*. The dump target is the storage device that kdump will use to save the dump. Kdump will try to use the root filesystem that was mounted by the main kernel if the dump target is not specified in the configuration file.

Kdump has access to the system memory through the `/proc/vmcore` memory image [65, 70]. It will create a dump file by using the `makedumpfile` utility. This utility can compress and exclude unnecessary dump data from the dump file [64, 71]. Before and after dump capturing, it is possible to run a user-defined script. These are the kdump pre- and post-scripts. They can be enabled in the configuration [64].

Kdump will perform its final action once the dump capturing is finished. This is usually a reboot, but can be configured differently. A failed dump capture will result in kdump running the failure action. The failure action is usually a reboot, but can also be configured [64].

3.4.2 Early kdump support in CentOS 8

The release of CentOS 8 introduced a new feature to the crashkernel named: *early kdump*. A system with early kdump support enabled is able to capture dumps when the main kernel crashes during boot-up [72]. This is achieved by booting the main kernel using an early kdump ramdisk. The ramdisk contains the crashkernel, and runs an early kdump script to load the crashkernel images into the reserved memory. After finishing, the kernel switches to the main root filesystem and continues booting [72]. Early kdump can add protection against any service in the root filesystem that may panic the kernel during booting. Further investigation is needed to find if this feature can be used on the Zynq MPSoC.

3.4.3 Crashkernel user notifications

One of the project requirements is that each fallback can report to the user about a failure if possible (see Section 2.3). The crashkernel does not have a built-in option to notify users about a crash after a dump has been captured.

RedHat provides a tool that can be used to report crashes to users. It is called *Automatic Bug Reporting Tool (ABRT)* [73]. ABRT can detect crashes from applications written in C, C++, Java, Python and Ruby. It can also detect and report kernel panics. Reporting kernel panics requires the kdump service to be enabled.

ABRT reports kernel panics using the abrt-vmcore service. This service detects new dump files in the `/var/crash/` directory[12]and can send an email to the user [73]. The system administrators at CMS use ABRT on other systems as well, and have it configured to send an email when something crashes. Email is the preferred crash reporting medium.

3.4.4 Crashkernel support for Zynq MPSoC

In 2017, Xilinx stated that the crashkernel was not supported on the 64-bit ARM architecture [74] (`aarch64` / `arm64`[13]) which is used by the Zynq MPSoC. The kernel documentation on kdump, one of the crashkernel's underlying mechanisms, states that the `arm64` architecture *is* supported in 2021 [65]. This means that there is a chance that the crashkernel would run on the Zynq MPSoC.

No source has been found that shows the implementation of the crashkernel on the Zynq MPSoC. The crashkernel mechanism may not be able to work on the Zynq Ultrascale+ MPSoC architecture. This will require further investigation and testing.

[12]The `/var/crash/` directory is the default location where dump files are stored on a filesystem [64].

[13]`aarch64` and `arm64` refer to the same architecture. When ARM released the 64-bit Armv8-A architecture it was named `aarch64`. The linux kernel community later decided to name the port of the kernel to this architecture `arm64` [75].

4. Research and analysis

This chapter presents the research that was conducted on the possible failures and respective fallbacks. The failures have been categorized using each stage in the Zynq MPSoC booting process.

4.1 Failure requirements

This section gives a definition of the failure requirements of the project (see Section 2.3). It also defines which failures are not covered by the reliable booting system.

4.1.1 Booting failures

A boot failure is defined as a failure that occurs before Linux has finished booting. Boot failures can occur in U-Boot, while booting the kernel, or while starting systemd services (different stages of booting, see Section 3.2). Boot failures in the BootROMs and FSBL are defined differently. The Zynq MPSoC booting process reveals that the BootROMs and FSBL mainly initialize the required hardware components for booting. If there is a failure in the BootROMs or FSBL, it would be logical to come from a hardware failure. The preconditions state that any board with a hardware failure shall be replaced. That is why the development of the reliable booting fallbacks is mainly focused on fails in the U-Boot stage and beyond[1].

4.1.2 Upgrade failures

A upgrade failure is defined as a failure that occurs when the Zynq MPSoC is booting a new firmware version. The firmware is defined as the Linux kernel image, device-tree blob, and ramdisk image. U-Boot should be able to retrieve the newest firmware version from the TFTP server. If a boot failure occurs in the Linux booting process, it is regarded as a failed upgrade. U-Boot should be able to roll back to a previous version of the firmware on the TFTP server in case of a failed upgrade.

4.1.3 Running failures

A running failure is defined as a failure that occurs after the Zynq MPSoC has finished booting. The failure can come from any application that runs on the Linux operating system and panics the kernel. The booting system should be able to detect the kernel panic and reboot the Zynq MPSoC.

4.2 Failure categorization

Booting failures are possible in every stage of the Zynq MPSoC booting process. This section analyzes each stage to find possible failures and their relevance. It also proposes fallbacks that have been devised after the analysis of the possible failures. Further research is needed to confirm if the fallbacks are plausible for the reliable booting system.

4.2.1 Pre-boot failures

The BootROMs that are run by the PMU and CSU reside in separate ROM memories. These ROM memories are not accessible to the user and cannot be changed. The execution of the BootROMs can still fail though. A hardware failure can e.g. prevent initialization in the PMU BootROM from finishing. This leads to an error state in the PMU, preventing the booting process from continuing [41].

[1] The BootROMs and FSBL *do* have fallbacks which has been researched and can be used if setup correctly. These are the golden image search and MultiBoot mechanisms.

The CSU can provide error codes for failures that occur during the booting process. The `CSU_BR_ERR` register can store two error codes at a time [76]. These error codes can be used to debug the chip when the CSU BootROM doesn't finish successfully. Some errors that are related to the booting process are:

- Error during initialization of a boot device (QSPI, NAND, SD, or eMMC).
- Error due to an invalid boot header.
- Error due to a corrupt FSBL image partition.
- Error due to a missing boot image.

A full list of error codes, relevant to the project, can be seen in Appendix B.

An error that occurs during the initialization of a boot device is likely due to a hardware failure. The preconditions in Section 2.3 state that such a failure shall be resolved by replacing the hardware.

The other errors are related to the boot image. The boot header could be invalid, or the image can be corrupted. It is also possible that the boot image is missing completely. These errors can be resolved by using a mechanism of the CSU BootROM that can search for boot images. This is the golden image search mechanism, which is explained in Subsection 3.2.2. This search mechanism requires multiple boot images to be stored in a boot device. If an image is invalid, the CSU will search for another image in the boot device.

Pre-boot fallbacks proposal
Two fallback solutions are possible for the pre-boot failures of the PMU and CSU BootROMs. These fallbacks take the golden image search mechanism and CSU error code registers into consideration:

1. Use the golden image search mechanism of the CSU BootROM. The mechanism will search for another boot image if there is anything wrong with the default one. This fallback requires multiple boot images to be stored on the boot device (see Subsection 3.2.2).
2. Have a Linux service that reads the values of the CSU error code registers once the system has booted[2]. This service will inform the user when the previous boot attempt didn't finish successfully. This service does not exist yet and has to be created.

4.2.2 FSBL failures

The FSBL can fail during each of the stages that it goes through. These are hardware initialization, boot device initialization, partition copy validation, and the handoff (see Subsection 3.2.3). If the FSBL returns an error at any stage, it will get handled by the `XFsbl_ErrorLockDown` function [45].

The `XFsbl_ErrorLockDown()` function will try to use the MultiBoot mechanism. This is a fallback mechanism in the FSBL that works in conjunction with the golden image search mechanism of the CSU BootROM. The mechanism will update the CSU MultiBoot offset register and reset the chip to try and boot with another image (see the full explanation in Subsection 3.2.3). The MultiBoot mechanism is useful if the FSBL tries to validate a corrupt ATF or U-Boot partition in the boot image.

In addition to the MultiBoot mechanism, there is a way to implement custom fallbacks through FSBL hooks. Hooks are blank functions in the FSBL source code that are executed at strategic locations [45]. The FSBL source code can be modified by the user to define such a hook. For example, there is a function named `XFsbl_HookBeforeFallback()` which can be used to create a fallback that runs before the MultiBoot mechanism is run.

[2]The values in the CSU error code registers will remain in the register after a system reset. The system needs to be power cycled or provided with a Power On Reset (POR) [77] to reset the values of the registers.

FSBL fallbacks proposal
Two fallback solutions are possible for the FSBL failures. These fallbacks take the MultiBoot mechanism and the FSBL hooks into consideration:

1. Use the MultiBoot mechanism in the FSBL to boot from a different image when the FSBL tries to validate a corrupt ATF or U-Boot partition in the boot image. This fallback requires multiple boot images to be stored on the boot device because it works in conjunction with the golden image search mechanism of the CSU BootROM.
2. Use the FSBL hooks to create a fallback that switches the boot device. Booting of U-Boot can be unsuccessful after multiple boot attempts on the default boot device. This may be due to a hardware failure. Boot images on a second boot device will not be touched by this hardware failure. They will act as a backup. The idea is to use the `CSU_MULTI_BOOT` offset register as a counter and modify the boot mode register (mentioned in Subsection 3.2.2) to change the boot device. Once the system has booted, it can inform the user about the failed boot attempts.

4.2.3 U-Boot stage failures

The requirements in Section 2.3 state that the Zynq MPSoC should boot through the network by default. U-Boot is responsible for retrieving the kernel image and device-tree blob from a TFTP server, and booting Linux. The image retrieval and booting can fail if:

1. The networking hardware is not working.
2. The Zynq MPSoC has an incorrect IP-configuration[3].
3. The images for booting Linux cannot be retrieved from the TFTP server
   - The TFTP server is not running / not available.
   - The images are missing on the TFTP server.
   - The images on the TFTP server are corrupted.
4. A network glitch occurs during the retrieval of the images.

U-Boot stage fallbacks proposal
The list above states that there are multiple scenarios in which U-Boot can fail when booting the system. All of these failures can possibly be resolved by using one fallback mechanism:

1. Points one to three can be resolved by booting the system with a set of locally stored backup images. Backup images can be stored on an SD-card or in QSPI flash. When the image retrieval from the TFTP server fails, U-Boot will boot the system using the backup images on the local storage device. The fallback can be created using a Hush shell script in U-Boot [54]. The script will have to detect when the images can cannot be retrieved through TFTP.
2. A network glitch (point four) can randomly occur. The fallback script can have a feature to retry the image retrieval. If it fails again, U-Boot can fall back to the backup images.
3. U-Boot will not be able to boot the system with an invalid image. The fallback script can have a feature to boot the system from the backup images if the images from the TFTP server are invalid.

4.2.4 Linux kernel boot failures

There are many reasons which may prevent the kernel from booting up correctly. Some of these are:

[3] A DHCP server (Dynamic Host Configuration Protocol) is used to assign IP-addresses and other network parameters to client devices [78].

- The kernel is not configured correctly.
- The kernel cannot mount the root filesystem.
- The boot arguments are not set correctly.
- The device-tree is incorrect.

Each of these reasons will result in a kernel panic. The default behavior of the panic function is to halt the system. The behavior can be changed by using the `panic` boot argument [79]. The user can configure the kernel panic to reboot the system after a time-out.

If the kernel keeps panicking during boot-up though, the chip will never boot into Linux. A fallback can be created to prevent this. There can be a mechanism that counts how many times Linux has tried to boot. After the counter reaches a certain threshold, the system can boot using a set of backup images.

Furthermore, the Zynq MPSoC can fail when booting with a new kernel image or device-tree. This "firmware upgrade" needs to be detected by the booting system. The Zynq MPSoC needs to roll back to a stable firmware version if it fails in an attempt to upgrade to a new firmware version.

The kernel can still panic and halt the system after it has fully booted up. A running user application may contain a bug causing a fatal error that crashes the operating system. A possible solution is the use of the *"crashkernel"* (this feature is studied in Section 3.4). This mechanism boots a second kernel on top of the crashed kernel and copies the contents of the system's memory into a dump file. The dump file is stored locally or remotely, and can later be used for post-mortem analysis of the crash [64].

Linux/kernel fallbacks proposal
Three fallback solutions are proposed for the Linux/kernel failures:

1. Change the panic behaviour of the kernel by using the `panic` boot argument. It will be configured to reset the system if the kernel panics during booting or at any other time.
2. Implement a counting mechanism that will boot Linux from a set of backup images when a threshold of boot attempts has been passed. This will require further research.
3. Implement a mechanism that can automatically detect new versions of the firmware and attempt an upgrade. The mechanism should be able to roll back to a previous version of the firmware if it fails to boot the new version.
4. Research the support for the crashkernel mechanism in PetaLinux and on the Zynq MPSoC. Implement the crashkernel mechanism on the Zynq MPSoC if possible.

4.2.5 Other failures

The previous subsections have discussed possible implementations of fallbacks in each part of the booting process. There is a possibility that there are unforeseen failures that were not considered while implementing the fallbacks. The fallbacks will not be able to protect the system in case of such failures. It is also possible that the fallbacks themselves fail. These scenarios will bring the Zynq MPSoC into a halted state. A *watchdog timer* can be used to prevent this.

The Zynq MPSoC includes a watchdog timer for the APU [80]. This watchdog timer can be used as a global guardian to protect all other fallbacks. An example of a fallback that can be protected is the crashkernel (see Subsection 4.2.4). This fallback is supposed to save a memory dump and reset the board by booting a second kernel. It is possible that this second kernel does not finish successfully. This will result in a halted system. The APU watchdog timer can be used to reset the system in this case. Further research will be needed to enable and use the APU watchdog timer in the Zynq MPSoC.

Watchdog timer fallback proposal
Enable the APU watchdog timer to protect all other fallbacks in case of a very specific failure. The watchdog timer will reset the system in case of a failure that was not anticipated during the implementation of the other fallbacks. Using a watchdog timer has the risk of ending up in an *infinite boot loop*. The possibility of a boot loop and enabling the watchdog timer in the Zynq MPSoC will require further research.

4.3 Follow-up on boot and upgrade failures

The fallback proposals in the U-Boot and Linux stages (see Section 4.2) specify the implementation of a fallback that can boot the system from a local storage device when the network boot fails. This will be a *backup-boot* with images that are stored on the SD-card of the ZCU102 development board. Furthermore, it was proposed to implement a boot counting mechanism that will perform a backup-boot once the maximum amount of boot attempts is exceeded. Lastly, a proposal was made to implement a mechanism for attempting firmware upgrades.

4.3.1 Backup-boots, boot counting and firmware upgrades

The backup-boot fallback, the boot counter fallback, and the upgrade mechanism will be part of a U-Boot fallback script. The mechanisms in the script are defined as follows:

1. A network check to see if the Zynq MPSoC can boot through the network. The test will be split up into a DHCP test which attempts the retrieval of an IP-address, and a TFTP test which attempts the retrieval of a dummy file. The system will boot with a set of backup images on the SD-card if one of the tests return negative. U-Boot will only check the DHCP and TFTP server as it only interacts with them. The NFS server is not checked, because the kernel is responsible for mounting the root filesystem via NFS (see Section I.1).
2. A global boot counter that counts the boot attempts of the Zynq MPSoC. If the boot counter reaches a maximum amount of allowed boot attempts, it will boot the system using a set of backup images on the SD-card. The backup-boot will also use a root filesystem that is stored locally on the SD-card.
3. A reliable upgrade mechanism that is able to detect new firmware versions on the TFTP server and attempt an upgrade. This mechanism will also have a boot counter to count how many times the Zynq MPSoC has tried to boot with the new firmware. It will roll back to the old version of the firmware if the system fails to boot after a maximum amount of boot attempts.

Fallbacks one and two will be used in the *RELBOOT* mechanism for reliable booting (see high-level design in Section 5.1). The fallbacks will cover multiple different failure scenarios in which Linux fails to boot-up. This may be due to inability to access the DHCP or TFTP server on the network, or because the kernel panicked during booting.

The third fallback is named *RELUP* and will be used for reliable upgrades. The project requirements state that the system should be able to recover from a failed upgrade. The fallback proposals in Section 4.2 specify the implementation of such a mechanism.

The fallbacks will work in conjunction with the watchdog timer of the Zynq MPSoC. If the system hangs when trying to boot Linux, the watchdog timer will trigger and reset the Zynq MPSoC. This will be repeated multiple times until the boot counters exceed, both during regular booting and during a firmware upgrade.

There also needs to be a daemon (systemd service) that automatically starts when Linux is booted. The daemon will check whether one of the RELBOOT fallbacks was triggered. It has to detect if the system booted using the backup images on the SD-card. It should also reset the global boot counter on every boot to prevent an unwanted backup-boot after multiple reboots. Furthermore, it should be able to detect if a RELUP firmware upgrade was attempted, and if it was successful or not. Finally, it needs to inform the user if there has been a failure.

4.3.2 Existing boot counting feature in U-Boot

The U-Boot bootloader offers an *advanced feature* that can detect a repeating reboot cycle by counting the amount of boot attempts [81]. The feature uses a boot counter that is incremented every time that U-Boot starts. U-Boot can run an alternative boot command when the boot counter exceeds the maximum amount of allowed attempts. The maximum amount of boot attempts and the alternative boot command are configurable through the U-Boot environment. U-Boot states that the boot counter should be reset on every successful Linux boot using a custom application [81]. This prevents the boot counter from exceeding and triggering the alternative boot command.

The *boot count limit* feature can be implemented as a fallback in the reliable booting system. Unfortunately, U-Boot states that the feature is only available on certain chips with a *Power architecture* [81]. This means that the feature is not supported for Zynq MPSoC. It was decided to implement a custom version of the boot counting feature by using the scripting capabilities in U-Boot.

4.4 Summary and discussion of failures and fallbacks

Sections 4.2 and 4.3 researched the possible failures and fallbacks in each part of the booting process. The failures that were mainly researched and categorized are listed in Figure 4.1. The table gives a summary of the failures and corresponding fallbacks.

Table 4.1 Summary of possible failures on the Zynq MPSoC and fallbacks that can protect against the fails.

#	Failure \ Fallback	Golden image search + MultiBoot	RELBOOT & RELUP	Crash-kernel	Watchdog timer	Replace hardware
1	Boot image invalid/corrupt (BOOT.BIN)	X				
2	Missing boot image	X				
3	Invalid/corrupt ATF or U-Boot images	X			X	
4	Not able to acquire IP-address/network information		X		X	
5	Not able to retrieve boot image from TFTP-server		X		X	
6	Linux kernel is not started by U-Boot		X		X	
7	Kernel / device-tree / ramdisk image is invalid/corrupt		X		X	
8	Booting fails during firmware upgrade		X		X	
9	Linux kernel can't mount NFS root filesystem		X		X	
10	Linux kernel panics during booting		X	X	X	
11	Linux kernel panics while running			X	X	
12	Other specific failures				X	X
13	System hangs in BootROMs or FSBL					X
14	Hardware failure					X

The golden image search mechanism is used to solve the pre-boot failures related to the boot image (`BOOT.BIN`). If there is something wrong with the boot image, the golden image search mechanism should be able to detect it and switch to another boot image. The MultiBoot mechanism, which is part of the FSBL, could be used to detect invalid ATF or U-Boot image partitions in the boot image.

The RELBOOT (reliable booting) mechanism, implemented in U-Boot, will recover the Zynq MPSoC from booting failures in the U-Boot stage and kernel booting stage. RELBOOT uses a network check and boot counter to recover the system and boots the Zynq MPSoC with backup boot images stored on an SD-card. U-Boot will also have a RELUP (reliable upgrade) implementation that can detect new firmware versions on the TFTP server and attempt firmware upgrades.

An already running kernel will be protected from panics by the crashkernel. The addition of early kdump (discovered in Section 3.4) will make it possible for the crashkernel to also protect against booting failures after the root filesystem has been mounted.

4.4.1 Tradeoff between fallbacks

The reliable booting system requires a tradeoff between the effectiveness of a fallback and the cost of implementation and maintenance. The fallbacks in the booting system have been chosen to cover multiple failure scenarios. It was decided not to implement dedicated fallbacks in the PMU firmware or FSBL for specific failures. The implementation of fallbacks in these parts of the booting process introduces problems with portability. When upgrading to a new version of the PMU firmware or FSBL (which are provided by Xilinx), developers would have to port the fallbacks manually. This requires resources that could be spent elsewhere in the CMS DAQ network. Research on fallbacks in the PMU firmware and FSBL can be a topic for a follow-up project.

The watchdog timer will be able to recover the Zynq MPSoC from other failures that are not covered by the implemented fallbacks. This does not include failures that hang the system before the FSBL is able to initialize the watchdog timer. The failure requirements (see Section 4.1) states that failures in the pre-boot and FSBL stages are expected to be related to a hardware issue. The hardware will be replaced if the Zynq MPSoC board has any hardware issues, as specified in the preconditions (see Section 2.3).

The hardware that is hosting the Zynq MPSoC will be accessible in the underground CMS-cavern. The hardware can be replaced in return for time and budget that has to be spent to access and replace the board. This is in contrast to a satellite, where the hardware is inaccessible and a failure can be fatal. In that scenario each failure, even very specific failures, should have a fallback, which is not the case for this project.

4.4.2 SD-card backup boot device

The booting system will be dependent on the boot device, which stores the boot image `BOOT.BIN`, a set of backup boot images, and a local root filesystem. Two options are the use of QSPI flash or the SD-card on the ZCU102. For ease of development, it was decided to use the SD-card on the ZCU102 development board as a boot device. The SD-card is removable and can be replaced in case of a hardware failure (more information the SD-card setup in Appendix G).

SD-cards mainly use NAND flash technology to create non-volatile storage [82]. NAND flash degrades over time depending on the amount of program/erase cycles that are performed on the storage [83, 84]. After a certain amount of program/erase cycles, the flash memory will lose the ability to retain data.

The SD-card will mostly be used for reading the boot image when booting. Reading operations in the SD-card do not degrade inner flash storage compared to write operations [83]. Moreover, the SD-card will not be used for writing. The root filesystem, which is stored remotely on an NFS server, will be written instead of the SD-card. The NFS server will have disk mirroring which provides redundancy. The local root filesystem on the SD-card might only be written during a backup boot, which is only used after a boot failure. It is therefore expected that the lifespan of the SD-card will be sufficient for the purpose of a backup-boot device.

5. High-level design

This chapter describes the high-level designs that were created for the fallbacks of the reliable booting system. These designs are mainly related to the RELBOOT and RELUP mechanisms.

5.1 Reliable booting system

The research results of Sections 3.1 to 4.2 were used to propose a high-level design of the reliable booting system. The high-level design is shown in Figure 5.1.

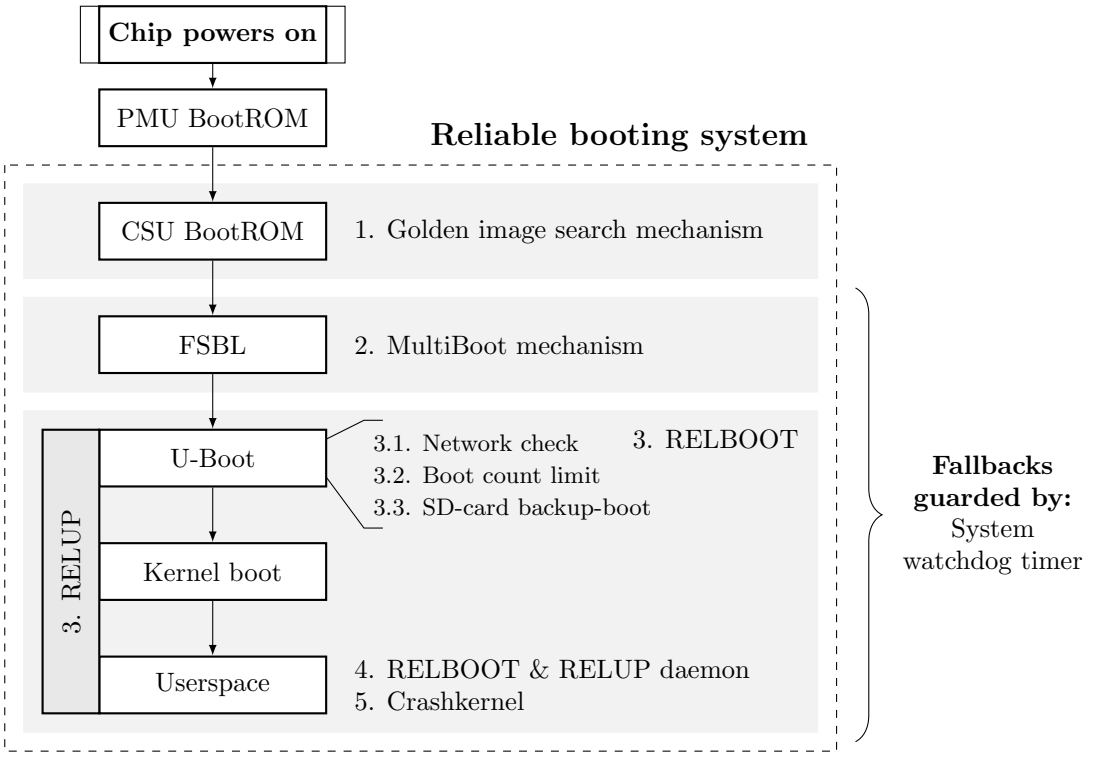


Figure 5.1: High-level design of the reliable booting system.

1, 2. The golden image search and MultiBoot mechanisms are part of the CSU BootROM and FSBL. To utilize these features, one must prepare a boot device with multiple boot images. The SD-card on the ZCU102 development board will be used as a boot device.

3. U-Boot will be equipped with the RELBOOT (reliable booting) mechanism. The mechanism will integrate a network check and a boot counter for limiting the amount of failed boot attempts. The Zynq MPSoC will boot with a set of backup-boot images on the SD-card if the network is not working, or if the system exceeds it maximum allowed boot attempts. RELBOOT will also have an extension called the RELUP (reliable upgrade) mechanism. RELUP is able to detect a new firmware version on the TFTP server and automatically attempt a firmware upgrade. It will roll back to the previous version of the firmware if the new firmware version fails to boot multiple times (RELUP will have a seperate counter).

4. RELBOOT & RELUP will require a daemon in Linux that starts when the Zynq MPSoC has finished booting. The daemon will be able to detect if the Zynq MPSoC booted using the SD-card backup-boot images. It will also be able to detect if a firmware upgrade through RELUP was successful or not.

5. The ability to use the crashkernel on the Zynq MPSoC will be researched. The mechanism will offer saving of a memory dump after a system crash for post-mortem analysis.

The fallbacks in the reliable booting system will be guarded by the system watchdog timer. The watchdog timer will perform a reset if the system hangs because of a unspecified failure, or if any other fallback fails. The implementation of the watchdog timer will require a heartbeat application that periodically resets the timer. This heartbeat application will be implemented as a daemon in Linux.

5.2 RELBOOT & RELUP mechanisms

5.2.1 RELBOOT & RELUP script

The design of RELBOOT & RELUP relies on the scripting capabilities within U-Boot. These scripts are based on the Hush shell [54]. Furthermore, RELBOOT & RELUP both rely on the watchdog timer to reset the Zynq MPSoC after a boot failure. This is essential for the boot counter functionality and detection of a failed firmware upgrade. The high-level design of the U-Boot script for RELBOOT and RELUP is shown in Figure 5.2.

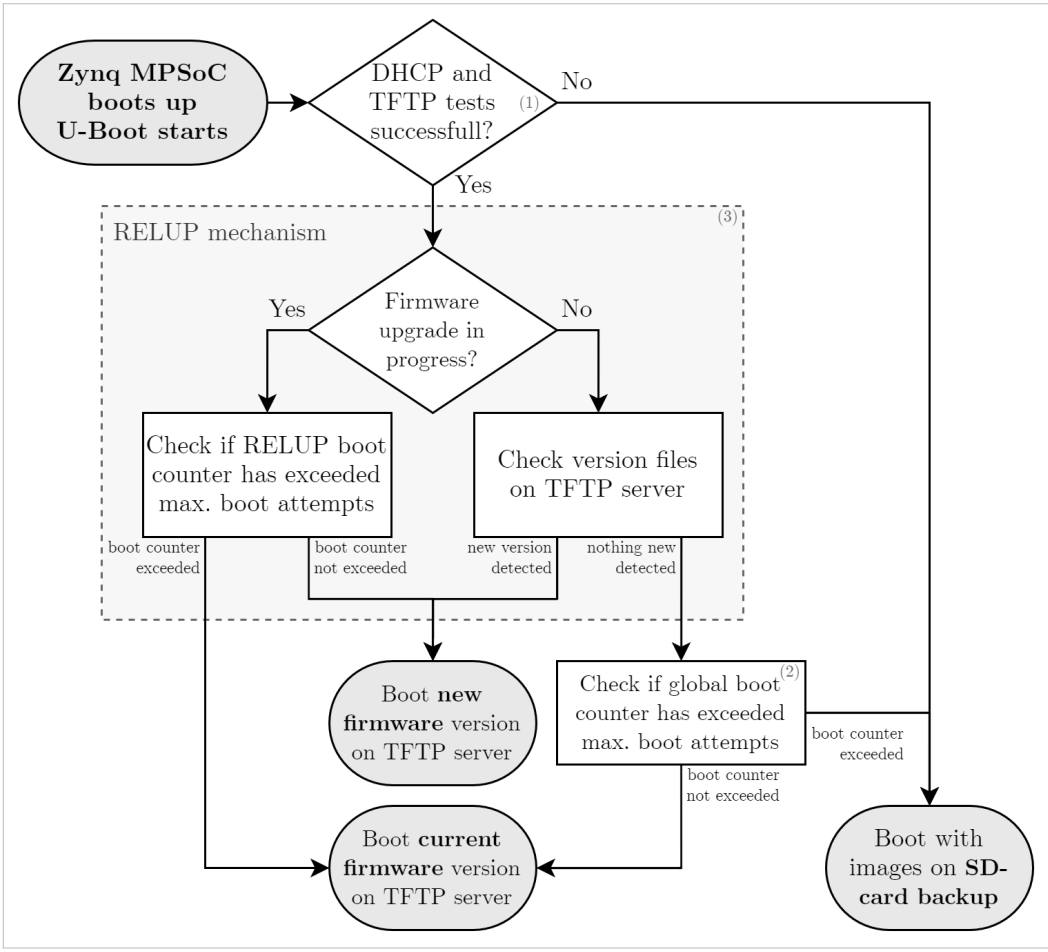


Figure 5.2: High-level design of RELBOOT & RELUP script.

The script performs a network check in the form of a DHCP request. If U-Boot successfully obtains a valid IP-address, the DHCP server is working. To verify that the TFTP file retrieval is working, the script will retrieve a *dummy file* that is located on the TFTP server. The script will continue by running the RELUP mechanism if the DHCP request and TFTP file retrieval are successful.

RELUP will check if there is a firmware upgrade in progress. If there is, it will compare the RELUP boot counter with the maximum amount of boot attempts. An exceeded boot counter results in a failed upgrade. In this case, U-Boot will roll back and boot the system using the *current firmware version*. A boot counter that is not exceeded results in another attempt to boot with the *new firmware version*.

If no upgrade is in progress, RELUP will check if it should start a firmware upgrade. It will try to retrieve two *version files* from the TFTP server. The values in the version files are compared to see whether there is a new firmware version available. If there is, RELUP will attempt a firmware upgrade. If there is no new firmware version on the TFTP server, RELBOOT will boot with the current firmware version.

The script will be able to detect if the chip failed to boot the current firmware version multiple times. U-Boot will boot the system using the backup images on the SD-card if the global boot counter exceeds. It will also use the backup images if the DHCP request or TFTP dummy file retrieval is unsuccessful.

It is possible that the TFTP server test is passed, but later U-Boot fails to retrieve the boot images from the server. U-Boot will not be able to boot Linux and drop down to its CLI. The chip will eventually be rebooted by the watchdog timer. This is repeated multiple times until the global boot counter exceeds and a backup-boot is triggered.

5.2.2 RELBOOT & RELUP Linux daemon

The RELBOOT & RELUP daemon is mainly used to check how the Zynq MPSoC was booted up (using SD-card backup, new firmware version etc). The daemon is also used to provide configuration of the RELBOOT & RELUP mechanisms. Figure 5.3 shows the high-level design of the daemon.

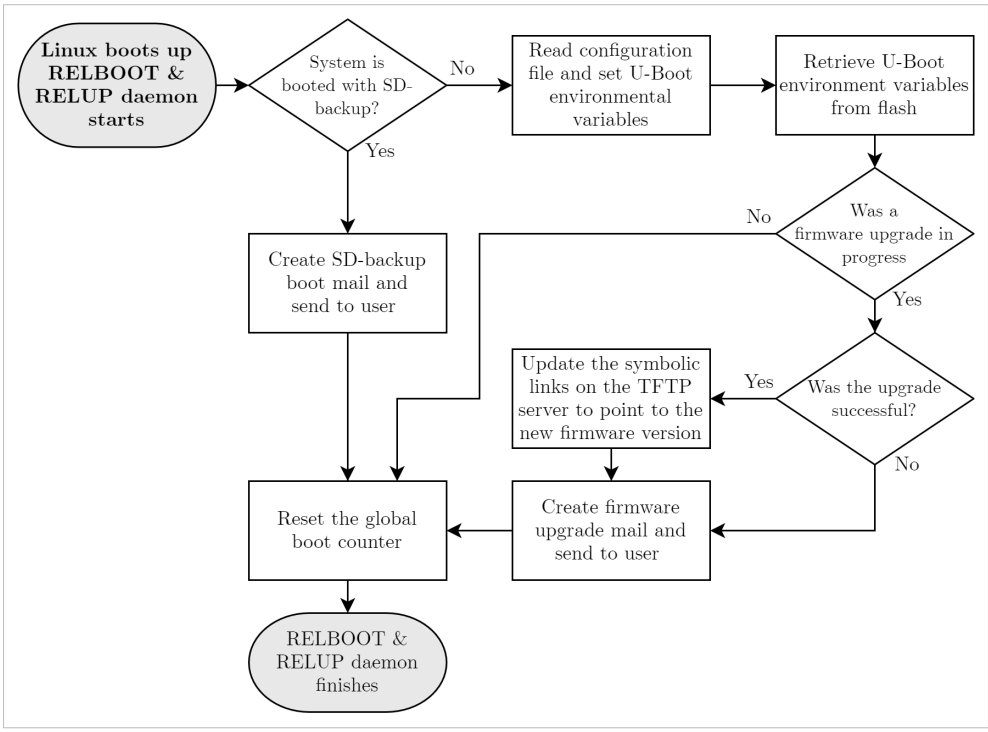


Figure 5.3: High-level design of RELBOOT & RELUP daemon that runs after Linux has booted.

The daemon starts by checking if the system was booted with the SD-card backup images. The user is notified by the daemon if the Zynq MPSoC is booted using the backup images on the SD-card. Notifications are sent through email[1]. The email might not arrive if the backup-boot was caused by a failed network.

[1]The system administrators at the CMS-experiment prefer to use email for these types of reports. This is why email is used to report to the user.

If the Zynq MPSoC was booted through the network, the daemon continues by reading a RELBOOT & RELUP configuration file. This configuration file will hold different options for RELBOOT & RELUP. An example of a configurable value is the maximum amount of boot attempts for the boot counters. The values of the configuration files will be propagated to U-Boot environment. This allows the RELBOOT& RELUP script in U-Boot to use the configuration that was set by the daemon.

The daemon will also retrieve a set of U-Boot environmental variables, including boot counters and firmware version variables. These will be used by the daemon to check if there was a firmware upgrade in progress during the boot-up. In case of an upgrade, it will be able to detect if the upgrade was successful, automatically update the firmware files on the TFTP server, and send a notification to the user through email.

The Zynq MPSoC will mount the TFTP server directory on boot-up. This directory contains all firmware versions, a directory with boot images, and a dummy file. The directory with the boot images will use a *symbolic link*[2] to point to a particular firmware version. An additional symbolic link is created when upgrading to a new firmware version (more on this in Section 6.2).

[2]A symbolic link is a file or directory that points to another file or directory [85].

6. Implementation

This chapter describes the implementation process of the fallbacks that are part of the reliable booting system. Extra effort has been spent on packaging the fallbacks into a board support package (BSP) - a template that defines how to support a particular hardware platform [86]. The BSP structure includes the sources of the reliable booting system and allows porting to different hardware. The BSP can be built automatically using a CI and PetaLinux (see Appendix H.5). The PetaLinux tools and BSP have been researched and implemented in Appendix H.

6.1 Golden image search mechanism

6.1.1 Boot image preparation

The golden image search mechanism requires the preparation of multiple boot images (`BOOT.BIN`). The images have a specific naming convention that is required by the search mechanism. The filenames of the boot images will contain an offset value which is represented by 4 figures [87]:

`BOOT0003.BIN`

An offset value of `0003` means that the search mechanism will increment the `CSU_MULTI_BOOT` register to three to boot with this image. The mechanism will require multiple images that contain different offset values. An example of a setup would be to have five boot images on the SD-card named `BOOT0001.BIN` to `BOOT0005.BIN`.

6.1.2 Enabling FSBL debug info

The CSU BootROM, which contains the golden image search mechanism, does not output any debug information on the serial console. The FSBL does print some messages by default, but they do not say anything about the booting process. The default boot messages from the FSBL can be seen in Figure 6.1:

```
1 Xilinx Zynq MP First Stage Boot Loader
2 Release 2019.2   Nov 26 2020  -  09:27:05
```

Figure 6.1: Default boot-up messages of the FSBL.

There is a way to enable detailed debug info for the FSBL. This can be done by building the FSBL with the `FSBL_DEBUG_INFO` build flag [88, 89]. The build flag can be added into the FSBL recipe of the PetaLinux project (see Appendix H.4). Figure 6.2 shows the content of the FSBL recipe file.

```
1 $ cat fsbl_%.bbappend
2
3 XSCTH_BUILD_DEBUG = "1"
4 YAML_COMPILER_FLAGS_append = " -DFSBL_DEBUG_INFO"
```

Figure 6.2: Debug info build flag for the FSBL in the FSBL recipe of PetaLinux.

After rebuilding the FSBL and `BOOT.BIN`, the debug info will be included. An example of FSBL boot-up messages with debug info enabled can be seen in Appendix C.1.

The messages will show each stage of the FSBL. The second stage debug messages are of interest for the golden image search mechanism. The boot mode, boot image filename and `CSU_MULTI_BOOT` register value can be seen here. These debug messages will be used when testing the golden image search mechanism.

6.2 RELBOOT & RELUP mechanisms

6.2.1 Firmware structure on TFTP server

The firmware on the TFTP server has been structured for the implementation of RELBOOT & RELUP and network booting. The TFTP server contains a directory for every Zynq MPSoC board in the CMS DAQ network, along with a directory for all firmware versions. It also contains the `dummy` file that is required by the TFTP file retrieval test in the RELBOOT mechanism.

Each firmware directory holds the images that are used by RELBOOT & RELUP for booting Linux. These are the `Image` kernel image, `system.dtb` device-tree blob, ramdisk, and `version` file among others. A directory tree of the TFTP server contents can be seen in Figure 6.3:

```
/tftpboot
├── dummy
├── firmware_versions
│   ├── v1.0
│   │   └── ...
│   └── v1.1
│       ├── earlykdump-ramdisk.img
│       ├── Image
│       ├── image.ub
│       ├── system.dtb
│       ├── uEnv.txt
│       └── version
├── zcu102-lab40-r01-33
│   └── boot_current -> ../firmware_versions/v1.1/
└── zcu102-lab40-r01-34
    ├── boot_current -> ../firmware_versions/v1.0/
    └── boot_new     -> ../firmware_versions/v1.1/
```

Figure 6.3: Directory tree with structure of the firmware files on the TFTP server.

Each board directory stores a symbolic link that points to the firmware version that it should boot. The RELBOOT mechanism uses the `boot_current` symbolic link to access the firmware version that it should normally boot.

In case of an upgrade, a `boot_new` symbolic link is created to point to a new firmware version. The tree in Figure 6.3 shows that the `zcu102-lab40-r01-34` board has a `boot_new` symbolic link that points to a new firmware version. Figure 6.4 gives an example of how to create a symbolic link for a firmware upgrade that will be handled by the RELUP mechanism:

```
1 $ cd /tftpboot/zcu102-lab40-r01-34
2 $ ln -sfn ../firmware_versions/v1.1/ boot_new
```

Figure 6.4: Example of creating a symbolic link to a new firmware version for a firmware upgrade using RELUP.

When the `zcu102-lab40-r01-34` board is rebooted, the RELUP mechanism will be able to detect the new firmware version by comparing the `version` files of `v1.0` and `v1.1`. The contents of the `version` file from `tftpboot/firmware_versions/v1.1/` are shown in Figure 6.5:

```
1 $ cat /tftpboot/zcu102-lab40-r01-34/boot_new/version
2 fw_version=v1.1
```

Figure 6.5: Contents of a version file on the TFTP server.

The `version` file contains the `fw_version` variable. The value of `fw_version` represents the name of the firmware directory on the TFTP server. This variable is retrieved and stored in U-Boot's environment by the RELUP mechanism.

The name of the firmware directory has to equal the value of the `fw_version` variable. The value is later used by the RELBOOT & RELUP daemon to update the `boot_current` symbolic link in case of a successful upgrade.

Note that the symbolic links use relative paths. When the RELUP mechanism tries to access the files in e.g. the `boot_current` *"directory"*, it will be redirected to the directory of the firmware version. RELUP is only able to access files on the TFTP server directory on the server. Any other files and paths are non-existent to RELUP. With an absolute path, RELUP will not be able to access the files that the symbolic link is pointing to.

6.2.2 RELBOOT & RELUP script low-level design

The RELBOOT & RELUP script has been split up into four parts. These are: the network check, the RELUP mechanism, the global boot counter, and the kernel boot-up. Figure 6.6 show the low-level design of the RELBOOT & RELUP script that runs in U-Boot.

After a successful network check, the RELUP mechanism will check if the last boot-up was performed using a new firmware version. It uses the RELUP boot counter to determine if a firmware upgrade is still in progress. By default, the boot counter would be zero. A zero value indicates that the Zynq MPSoC was not booting a new firmware version. If a new firmware version is detected by RELUP, the counter will be incremented to one. Both to start the counting, and to indicate that an upgrade is in progress.

The RELUP mechanism uses four version variables to determine if a firmware upgrade should be started. These environmental variables are used for passing and storing firmware versions. These are:

`tftp_currentver_fw`	This variable stores the firmware version of the `boot_current` directory on the TFTP server.
`tftp_newver_fw`	This variable stores the firmware version of the `boot_new` directory on the TFTP server. `tftp_newver_fw` will only hold a value when there is a `boot_new` symbolic link to a new firmware version on the TFTP server.
`qspi_bootver_fw`	This is the QSPI boot version. This variable holds the version of the firmware that the Zynq MPSoC was previously/currently booted with. During a regular boot, this variable will be equal to `tftp_currentver_fw`. During a firmware upgrade, it is equal to `tftp_newver_fw`. In case of a backup-boot, the variable will be set to "`SD-backup`".
`fw_version`	This variable is stored in the `version` files on the TFTP server. The `fw_version` variable in the U-Boot environment changes when the `version` file is retrieved from the TFTP server. The value of `fw_version` is passed to the `tftp_currentver_fw` or `tftp_newver_fw` variable.

The RELUP mechanism will first retrieve the *current firmware version* from the TFTP server. It creates a path for retrieving the `version` file by using the *hostname* of the board: e.g. if the hostname is `zcu102-lab40-r01-33`, the `version` file would be retrieved from `zcu102-lab40-r01-33/boot_current/` (see Figure 6.3). The RELUP mechanism will compare `tftp_currentver_fw` with `qspi_bootver_fw`. The values of the variables should be equal. If they are not equal, there is two possible reasons:

1. The Zynq MPSoC board is booted for the first time and the `qspi_bootver_fw` variable does not hold a value yet.
2. The `version` file in the `boot_current` directory on the TFTP server was changed. Possibly because the `boot_current` symbolic link was changed to boot with a different firmware version.

If the `boot_current` symbolic link on the TFTP server is changed, the Zynq MPSoC will boot with a different firmware version without performing an upgrade using RELUP. The RELUP mechanism will alert the user on the console that this is regarded as an unofficial firmware upgrade. A rollback is not available in this case.

RELUP continues by trying to retrieve a *new firmware version* from the TFTP server. If successful, it will compare the `tftp_newver_fw` and `tftp_currentver_fw` to determine if there is a new firmware version for an upgrade. An upgrade will be started by setting the QPSI boot version to the new firmware version, and by incrementing the RELUP boot counter.

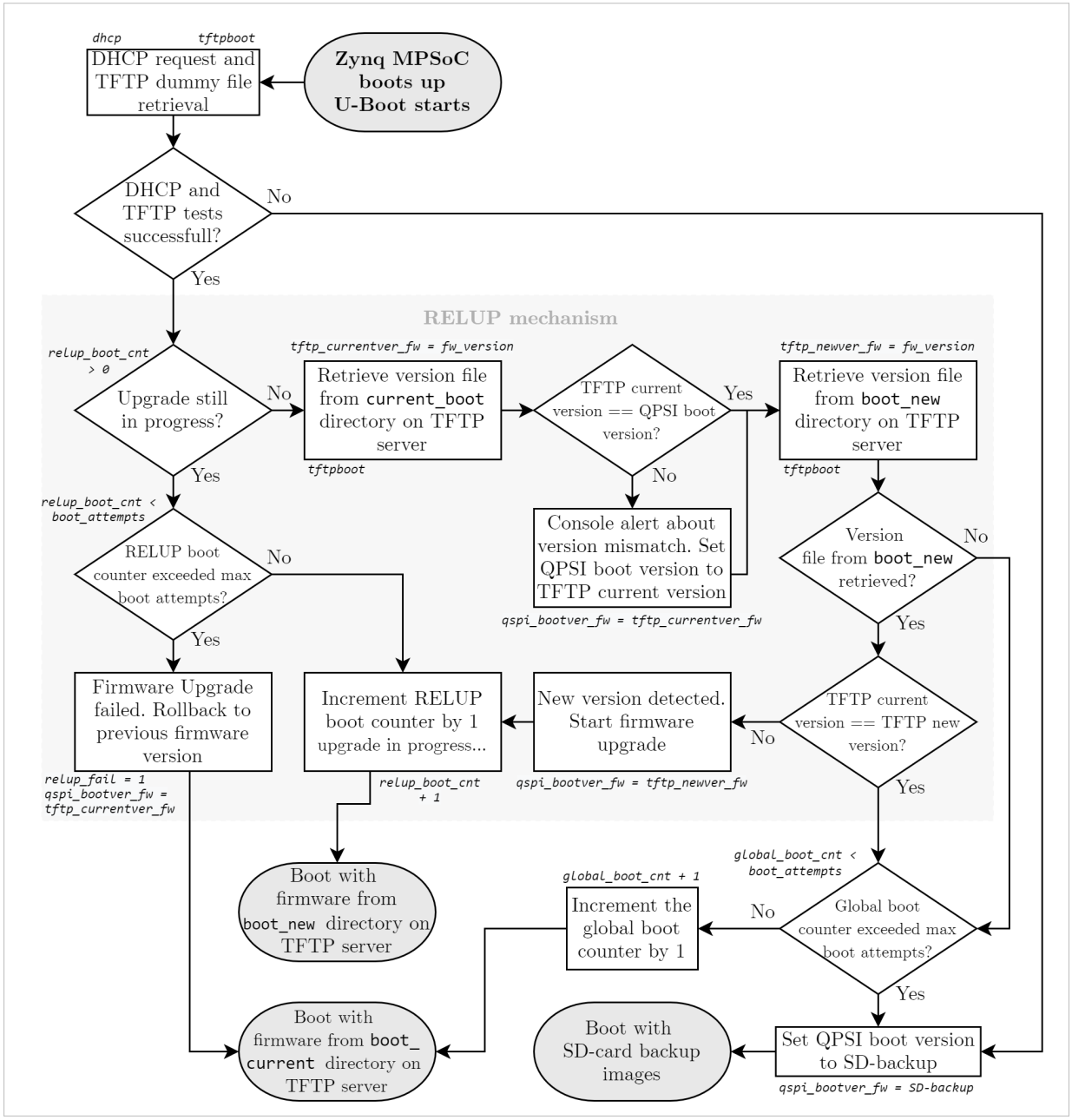


Figure 6.6: Low-level design of the RELBOOT & RELUP script in U-Boot.

The status of a firmware upgrade is set through the `relup_fail` flag variable. This flag is set to zero by default to indicate that there was no failed firmware upgrade. If the flag is set to one, it indicates that the RELUP boot counter exceeded the maximum amount of allowed boot attempts and that the firmware upgrade failed. The flag is mainly used by the RELBOOT & RELUP daemon in Linux.

The script will continue to check the global boot counter if RELUP does not find a new firmware version. RELBOOT will boot the Zynq MPSoC with the backup images on the SD-card if the global boot counter exceeds. Else, it will proceed by incrementing the global boot counter and booting using the current firmware version. The global boot counter is reset to zero once Linux boots up and the RELBOOT & RELUP daemon is started.

The RELBOOT & RELUP script also implements the ability to choose how to boot Linux on the Zynq MPSoC (see flowchart in Appendix D.1). The user has three *boot options*. The boot options allow the user to use an `image.ub` format or separate images for the kernel and device-tree (`Image` and `system.dtb`). The third option also uses separate images, but also involves a ramdisk image. The boot option is selected using the `img_type` variable. It can be configured using the configuration file of the RELBOOT & RELUP daemon in Linux. These boot options are used by the networking booting that has been implemented in the RELBOOT & RELUP mechanisms (see Appendix I).

6.2.3 Script integration in boot image

The RELBOOT & RELUP script can be embedded into the U-Boot binary to make it part of the Zynq MPSoC boot image (`BOOT.BIN`). This is done by storing the script in an environmental variable. The U-Boot binary contains a default environment that is loaded into the QSPI flash of the board when booting up for the first time.

U-Boot can be configured to add the RELBOOT & RELUP script to its default environment. This has been done through the `platform-top.h` file, in the U-Boot recipe of the PetaLinux project (see Appendix H.4). The `CONFIG_EXTRA_ENV_SETTINGS` option is used to add environmental variables to the default environment. Figure 6.7 shows how environmental variables can be added[1].

```
#define CONFIG_EXTRA_ENV_SETTINGS \
    "cms_boot_attempts=3\0" \
    "cms_tftp_currentver_fw=0\0" \
    "cms_global_boot_cnt=0\0" \
""
```

Figure 6.7: The addition of environmental variables to the default environment in the U-Boot binary.

The `CONFIG_EXTRA_ENV_SETTINGS` definition is basically an *array of strings*. The strings are separated by `NULL` characters (`\0`). The RELBOOT & RELUP script would have to be added to the `CONFIG_EXTRA_ENV_SETTINGS` as a string.

The realization of the RELBOOT & RELUP script was done in a separate file, which has more than 200 lines of code. Adding all of these lines as a string would be inefficient and time consuming. That's why it was decided to create a custom *parser* application in Bash. This *"scriptadder"* application can take a U-Boot script and add it to the `CONFIG_EXTRA_ENV_SETTINGS` definition with appropriate styling and indentation (see Appendix D.2 for more information).

6.2.4 RELBOOT & RELUP Linux daemon implementation

The RELBOOT & RELUP daemon has been created as a systemd service. The daemon is the last service that is started during the Linux booting process, because it marks a successful boot. Figure 6.8 shows the contents of the systemd service file. The service file specifies which script will be executed when systemd starts the service. It also specifies that any debug information that is printed by the daemon should be outputted on the console.

[1] Note that the variables in Figure 6.7 have a `cms_` prefix. All variables that were added for RELBOOT & RELUP, also the ones explained in Subsection 6.2.2, have this prefix.

```
$ cat /etc/systemd/system/cms-relup.service
[Unit]
Description=Linux daemon for the RELBOOT \& RELUP mechanisms

[Service]
Type=idle
ExecStart=/bin/bash /usr/bin/cms-relup.sh
StandardOutput=journal+console
SyslogIdentifier=cms-relup

[Install]
WantedBy=multi-user.target
```

Figure 6.8: Contents of the RELBOOT & RELUP systemd service file.

The `WantedBy=multi-user.target` is used to specify that the service should be started at boot time. `multi-user.target` defines a system state where all network services are started and Linux accepts logins [90]. The service is started last by adding the `Type=idle` option. This tells systemd to delay the execution of the service until all active service are dispatched [91].

The RELBOOT & RELUP daemon starts by checking how the Zynq MPSoC was booted. It checks if the `qspi_bootver_fw` variable in U-Boot's environment is equal to "`SD-backup`". The daemon will continue by retrieving a set of other variables from the U-Boot environment if it was not booted using the SD-card backup images. The U-Boot environment in QSPI flash is accessed through the `fw_printenv` and `fw_setenv` utilities. The firmware utilities are provided in the U-Boot repository [92] (more details on compiling these utilities in Appendix D.4).

These variables are used to check if an upgrade was started during the boot-up:

`tftp_currentver_fw`	Firmware version of the `boot_current` directory on the TFTP server.
`tftp_newver_fw`	Firmware version of the `boot_new` directory on the TFTP server.
`relup_fail`	Flag that indicates a failed firmware upgrade.
`relup_boot_cnt`	Boot counter used by RELUP to count boot attempts during an upgrade.
`global_boot_cnt`	Global boot counter used by RELBOOT to count boot attempts.

The daemon checks if `relup_boot_cnt` is greater than zero to determine if a firmware upgrade was started. It continues by checking the `relup_fail` flag to determine if an upgrade was in progress. The upgrade is marked as failed if the flag is set to one. A successful upgrade results in the `boot_current` symbolic link being updated to point to the new firmware version. After determining the state of the firmware upgrade, the daemon will create an notification email and send it to the user.

The RELBOOT & RELUP daemon finishes by resetting `global_boot_cnt` to zero. This marks that the boot-up of the Zynq MPSoC was successful.

Email notifications are sent using the `mailx` utility [93]. This utility requires a mail transfer agent (ATF): e.g. `postfix` [94]. Figure 6.9 shows how mail support was added to the Zynq MPSoC for the RELBOOT & RELUP daemon.

```
$ yum install mailx postfix

# How the RELBOOT & RELUP daemon sends an email:
$ mail -s "$HOSTNAME | fw-upgrade $relup_result" "$email_address" < /tmp/mailfile
```

Figure 6.9: Commands for adding mail support on the Zynq MPSoC for the RELBOOT & RELUP daemon.

The daemon creates a temporary mail file and sends this to the email address of the user. The email address of the user can be specified in the `/etc/relup.d/relup.conf` configuration file. This file provides configuration options for the RELBOOT & RELUP mechanism. The configuration options are propagated to the U-Boot environment in QSPI flash. An example of a configuration option is the maximum amount boot attempts for the boot counters. All configuration options are listed in Appendix D.3.

6.3 Crashkernel mechanism

6.3.1 Kernel configuration

Setting up the crashkernel requires multiple steps, the first of which is the kernel configuration. The main system kernel needs to be compiled with certain kernel options that are required by the crashkernel. These options are through the `petalinux-config -c kernel` command in PetaLinux (see Appendix H.2). This will open `menuconfig`. The required kernel options can be seen in Figure 6.10 [65, 70, 95]:

```
kernel hacking --->
    [*] Kernel debugging                                    CONFIG_DEBUG_KERNEL
    Compile time checks and compiler options --->
        [*] Compile kernel with debug info                  CONFIG_DEBUG_INFO

filesystems --->
    psuedo filesystems --->
        [*] /proc filesystem support                        CONFIG_PROC_FS
        [*]   /proc/vmcore support                          CONFIG_PROC_VMCORE
        [*]- sysfs file system support                      CONFIG_SYSFS

kernel features --->
    [*] kexec system call                                   CONFIG_KEXEC
    [*] build kdump crash kernel                            CONFIG_CRASH_DUMP
    [*] Build a relocatable kernel                          CONFIG_RELOCATABLE
```

Figure 6.10: Required kernel options for the crashkernel. The options in the menuconfig are shown on the left. The names of the kernel options are shown on the right.

The main system kernel needs to support kernel debugging. In addition, it needs to be compiled with debug info. The crash analysis tool requires a `vmlinux` image with debug symbols to be able to read and analyze the dump file [65].

The `PROC_FS` and `SYSFFS` options enable *pseudo file systems* that provide information about the status of the system [96, 97]. It is "pseudo", because the information is represented to the user in the form of a file that does not take up any space on the disk. The files are temporarily created by the kernel in memory when someone tries to access them.

The `vmcore` support gives the user access to the pseudo ELF (Executable Linkable Format) file of the system memory. As mentioned in Section 3.4, access to the memory is necessary to capture a dump. Finally, the crashkernel requires the kexec system call and the actual kdump dump capture mechanism.

For testing purposes, it was desired to use the same image for the main system kernel and the crashkernel. This requires the kernel to be built with the `CONFIG_RELOCATABLE` option. This option builds the kernel as a Position Independent Executable (PIE). This retains the relocation metadata of the kernel, which is used by kexec to load the crashkernel binary at a different virtual memory address [65, 98].

6.3.2 Memory reservation

The background research on the crashkernel describes that the crashkernel is loaded into a reserved part of memory that is not accessible to the main kernel (see Section 3.4). The Zynq MPSoC on the ZCU102 has access to 4 GB of memory. The amount of memory needed for the crashkernel can vary and is also dependent on the processor architecture [64, 65]. The kdump documentations of various Linux distributions suggest different values for the memory reservation [64, 70, 95]. Usually 64 MB is reserved for a `x86_64` architecture.

Automatic memory reservation for the crashkernel is also possible. RedHat states that the kernel will automatically reserve 512 MB for a system with an `arm64` architecture and 2 GB of RAM [64]. Arch Linux and RedHat suggest a memory reservation of 256 MB and *up to* 512 MB [70, 99]. An automatic memory reservation of 512 MB will use up $1/8^{\text{th}}$ of the ZCU102's memory.

After testing, it was verified that the crashkernel works with a memory reservation of *256 MB*. Further testing optimized the memory reservation to 192 MB (details can be found in Appendix E.1). The memory optimization made an additional 64 MB of memory available to the main system kernel. The memory requirement can be further optimized by building a crashkernel image that is smaller and has less features. Currently, the main system kernel and crashkernel images are the same.

6.3.3 Device-tree modifications

Memory reservation for the crashkernel is done through the `crashkernel` boot argument [65]. In addition to the memory reservation, one also needs to include the `rd.earlykdump` boot argument in the device-tree. The early kdump feature uses this boot argument to see if it should load the crashkernel images into memory. The boot arguments are added in the device-tree[2] (see line 8 in Figure 6.11):

```
1  / {
2          model = "CMS DAQ ZynqMP ZCU102 board";
3          compatible = "xlnx,zynqmp";
4
5          chosen {
6              xlnx,eeprom = &eeprom;
7              bootargs = "earlycon console=ttyPS0,115200 clk_ignore_unused
8                          crashkernel=192M rd.earlykdump earlyprintk
9                          cpuidle.off=1 root=/dev/nfs ip=dhcp rw";
10         };
11 };
```

Figure 6.11: Addition of memory reservation and early kdump boot arguments in device-tree.

6.3.4 Enabling and starting kdump

Kdump and the kexec-tools can be enabled on the Zynq MPSoC once the kernel and device-tree have been modified. The `Image` file, that is used to boot the main system kernel, has been copied to the `/boot` directory on the Zynq MPSoC. It is renamed to `vmlinuz-4.19.0-xilinx-v2019.2`[3]. This image will be loaded into the reserved memory by kexec. The `vmlinux` image can also be copied to the `/boot` directory so it can later be used when analyzing a dump file.

Figure 6.12 shows how the kexec-tools can be installed and how the kdump service is enabled:

[2]The device-tree has been modified through the `system-user.dtsi` file in the device-tree recipe of the PetaLinux project (see Appendix H.4).

[3]Kdump requires this naming format. The name depends on the kernel version and PetaLinux version that is used. During the project, PetaLinux v2019.2 was used with kernel version 4.19.0.

```
$ yum install kexec-tools

$ systemctl start kdump.service
$ systemctl enable kdump.service
$ systemctl status kdump.service
```

Figure 6.12: These commands show how to install the kexec-tools and start the kdump service.

Kdump will create an initramfs image after being started. This ramdisk will be stored in the `/boot` directory. Enabling the service will allow the service to start automatically on boot-up. The status of the kdump service can always be checked using the status option of the `systemctl` command.

Enabling early kdump requires an additional ramdisk image to be built. The ramdisk is created using *Dracut*[4]. Dracut includes two modules that allow it to create a ramdisk for early kdump [72]. Figure 6.13 shows the modules and the command to create the ramdisk.

```
$ ls -l /usr/lib/dracut/modules.d/99earlykdump/
total 8
-rwxr-xr-x 1 root root 1690 Jan  4 16:33 early-kdump.sh
-rwxr-xr-x 1 root root 1879 Jan  4 16:33 module-setup.sh

$ dracut -f --add earlykdump
```

Figure 6.13: Early kdump modules in dracut and the creation of the early kdump ramdisk.

This ramdisk is used by the main system kernel when booting. When the Zynq MPSoC boots-up the kernel will mount the ramdisk and load the crashkernel. The ramdisk switches to the root filesystem on the NFS server once early kdump is finished. Note that the ramdisk should be wrapped with a U-Boot header. This can be done using the `mkimage`. that is provided by the U-Boot repository.

6.3.5 Crashkernel workarounds

Kdump can be configured through the `/etc/kdump.conf` file. The default configuration does not set a dump target. This forces kdump to try and mount the root filesystem of the main system kernel when saving a dump (see Section 3.4). Kdump has been configured to dump the SD-card (see Appendix E.2). Dumping directly to the NFS server failed when testing. Kdump failed to execute the kdump vmcore saving service in the crashkernel. The console output of the fail can be seen Figure 6.14.

```
[   12.181600] dracut-cmdline[1749]: Using kernel command line parameters:
ip=128.141.174.208::128.141.174.1:255.255.255.0::kdump-eth0:none ifname=
...
ip=eth0:static earlycon console=ttyPS0,115200 ip=dhcp rw
...
[   12.645835] dracut-cmdline[1749]: Multiple ip= arguments: assuming rd.neednet=1
[   12.677949] dracut-cmdline[1749]: Warning: Empty autoconf values default to dhcp
[   12.775885] dracut: FATAL: Sorry, 'ip=eth0:static' does not make sense for
multiple interface configurations
[   12.795600] dracut: Refusing to continue
[   12.772725] systemd[1]: Shutting down.
...
[   13.079075] reboot: System halted
```

Figure 6.14: Dracut service in the crashkernel refuses to continue after detecting multiple ip boot arguments.

[4]Dracut provides an infrastructure for the ramdisk and sets up the system.

Figure 6.14 shows how Dracut tries to use the kernel boot arguments to setup the system. It fails because of multiple incorrect `ip` options in the boot arguments (see lines 2 and 4 in Figure 6.14). Dracut refusing to continue booting and the system results in a halted state. The issue was researched, but no answers were found to why Dracut is adding multiple `ip` options to the boot arguments.

A custom kdump-post script was used as a workaround to this issue. The kdump-post script is started after the kdump vmcore saving service successfully captures a dump. Kdump will use the SD-card on the ZCU102 as a dump. The kdump-post script is able to copy the dump file from the SD-card to the NFS server. Figure 6.15 shows a design of the kdump-post script implementation:

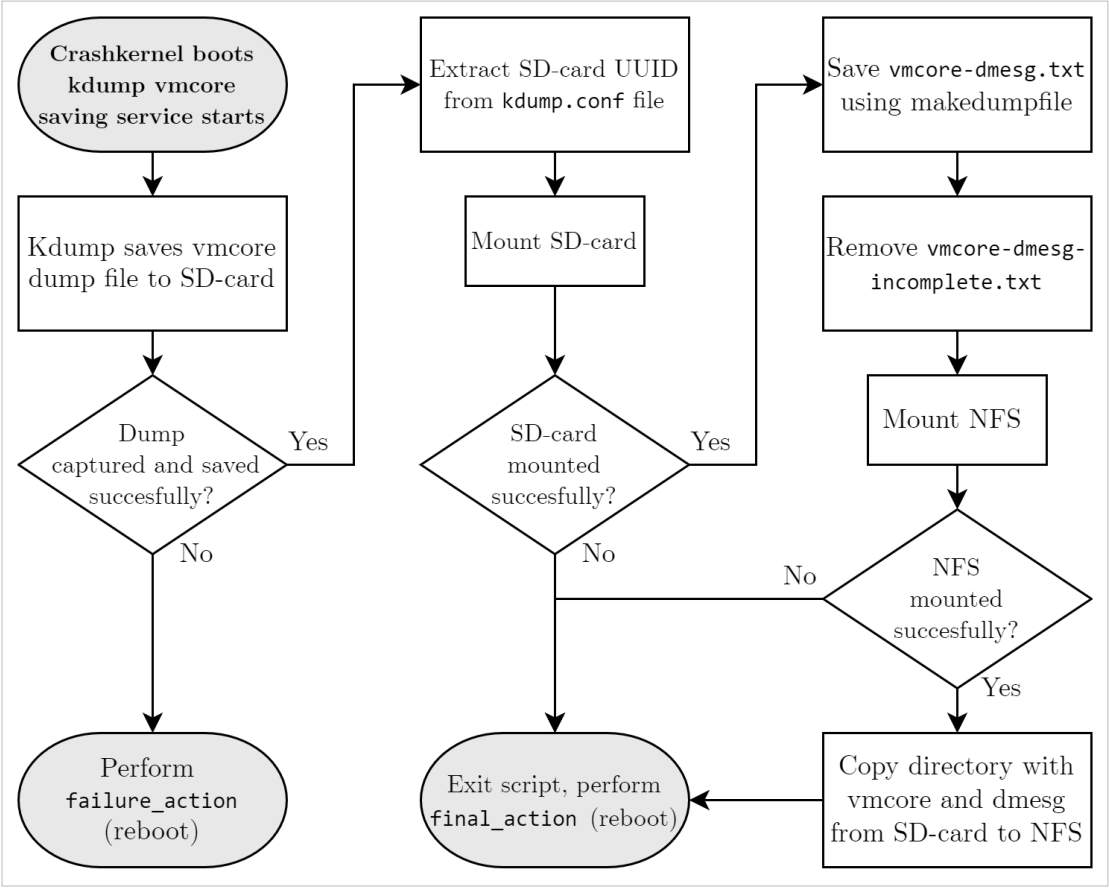


Figure 6.15: Flowchart of kdump-post script that can save dmesg and dump file to NFS.

The script starts by finding the SD-card device UUID (universally unique identifier). This information is stored in a stripped down `kdump.conf` file that stored on the initramfs of the crashkernel. The UUID is used to mount the SD-card.

During the crashkernel testing, it was also discovered that the kernel console messages (dmesg) are not saved properly (see Appendix E.4). The kdump-post script will save the dmesg by using the `makedumpfile` utility with the `--dump-dmesg` option.

The script will continue by mounting the root filesystem of the Zynq MPSoC via NFS. The directory that stores the crash dump on the SD-card will be copied to the NFS server. Finally, the kdump service will perform its final action and reboot the the Zynq MPSoC.

Once the Zynq MPSoC is booted-up, it will notify the user about the crash using an email notification from the ABRT service. Appendix E.3 shows how the ABRT service was configured to send crashkernel reports to the user.

6.4 Watchdog timer

6.4.1 PMU firmware configuration

The watchdog timers of the Zynq MPSoC are enabled through the hardware configuration in Vivado. Their timeouts are handled by the PMU firmware. The watchdog timer handling is enabled by adding additional build flags for the PMU firmware [38] (see Figure 6.16).

```
1 YAML_COMPILER_FLAGS_append = " -DENABLE_EM -DENABLE_RECOVERY -DENABLE_ESCALATION \
2                               -DCHECK_HEALTHY_BOOT -DENABLE_WDT"
```

Figure 6.16: Enabling watchdog timer handling for the PMU firmware. The build flags have been added to the PMU firmware recipe in the PetaLinux project (see Appendix H.4).

The `ENABLE_EM` build flag adds the error management module to the PMU firmware. This module is required by the `ENABLE_RECOVERY` build flag to do the actual watchdog timer handling. Furthermore, the recovery mechanism requires the PMU firmware to be compiled with the the power management module and the scheduler module (these are enabled by default) [38].

The escalation and healthy bit schemes are enabled with `ENABLE_ESCALATION` and `CHECK_HEALTHY_BOOT`. These flags allow the PMU firmware to perform a system reset when the the watchdog timer reset was not able to successfully reset the APU only.

The PMU itself can also be protected by a watchdog timer. The `ENABLE_WDT` build flag will add an interrupt service routine to the PMU firmware that periodically resets a watchdog timer that is handled by the CSU. If the PMU firmware hangs, the CSU watchdog timer will timeout and perform a system reset of the Zynq MPSoC.

6.4.2 Kernel configuration

The watchdog timers of the Zynq MPSoC can be accessed from Linux by building the kernel with certain watchdog timer drivers. The drivers can be added by running the `petalinux-config -c kernel` command in PetaLinux (see Appendix H.2). The required kernel options can be seen in Figure 6.17:

```
Device Drivers --->
    [*] Watchdog Timer Support --->              CONFIG_WATCHDOG
        [*] Disable watchdog shutdown on close   CONFIG_WATCHDOG_NOWAYOUT
        <*> Xilinx Watchdog Timer                CONFIG_XILINX_WATCHDOG
        <*> Cadence Watchdog Timer               CONFIG_CADENCE_WATCHDOG
```

Figure 6.17: Required kernel drivers to access the watchdog timers from Linux.

The Zynq MPSoC uses the Cadence watchdog timer driver. Xilinx states that both the generic Xilinx watchdog driver and Cadence watchdog driver should be enabled [100].

The watchdog timer is accessible through a `/dev/watchdog0` device file when Linux is running with these device drivers. The watchdog timer will activate as soon as the device file is opened. At this point, the watchdog timer can be reset by writing a character to the device. The watchdog timer is disabled when the device file is closed [101]. This feature is not very practical, because the Zynq MPSoC will not be rebooted if the watchdog heartbeat application is stopped.

The `CONFIG_WATCHDOG_NOWAYOUT` option makes sure that the watchdog timer is not disabled after closing the `/dev/watchdog0` device file [101]. By enabling this option, Linux has no way to disable the watchdog timer. This makes sure that the Zynq MPSoC will be rebooted if the watchdog timer is not periodically reset by a heartbeat application.

6.4.3 Device-tree modifications

It is necessary for the watchdog timer hardware to be defined in the device-tree to access it from Linux. The watchdog timer hardware is defined by default in the `zynqmp.dtsi` file. This file is added to a PetaLinux project when the project is created with the ZynqMP tempate [102] (See Appendix H). The definition has to be modified to inform Linux that the watchdog timer hardware is enabled. Figure I.3 shows the watchdog timer node in the device-tree is modified.

```
1 &watchdog0 {
2     status = "okay";
3     reset-on-timeout;
4     timeout-sec = <60>;
5 };
```

Figure 6.18: Required kernel drivers to access the watchdog timers from Linux.

The `status` property is set to "okay" to inform the kernel that the watchdog timer is enabled. The `reset-on-timeout` property informs Linux that the watchdog timer hardware is configured to reset the Zynq MPSoC on expiry. Lastly, the node informs the kernel that the watchdog timer hardware in the Zynq MPSoC is configured with a default timeout duration of 60 seconds.

6.4.4 Watchdog timer heartbeat daemon in Linux

The watchdog heartbeat daemon was created by using example code that is provided by Xilinx for servicing a watchdog timer. The example code resets the watchdog timer periodically with an interval of two seconds. The code does not use the `/dev/watchdog0` device file. Instead, it writes to the restart register of the watchdog timer directly. The source code of the heartbeat daemon can be found in Appendix F.2.

The daemon uses the `mmap()` system call to map the physical register address of the watchdog timer to a virtual address that can be used by Linux. The system call uses the `/dev/mem` device file to map the physical register address to a virtual address. The daemon restarts the watchdog timer by writing a hexadecimal value to the restart register [80]. After writing the value, the virtual memory mapping is deleted using the `munmap()` system call.

The watchdog heartbeat daemon is automatically started through a systemd service. The service is started at boot-up by specifying the `WantedBy=multi-user.target` and `Type=simple` options in the service file of watchdog heartbeat daemon. The contents of the watchdog heartbeat daemon service file are shown in Figure 6.19.

```
1  $ cat /etc/systemd/system/watchdog_heartbeat.service
2  [Unit]
3  Description=Linux service for starting the watchdog timer heartbeat application
4
5  [Service]
6  Type=simple
7  ExecStart=/bin/wdt_heartbeat
8  StandardOutput=journal+console
9  SyslogIdentifier=watchdog-heartbeat
10
11 [Install]
12 WantedBy=multi-user.target
```

Figure 6.19: Watchdog heartbeat daemon service file for systemd.

7. Testing and results

7.1 Boot system testing approach

The reliable booting system is tested by emulating multiple types of failures. The failures are categorized by the requirements. The booting system must recover the Zynq MPSoC from boot failures (b), upgrade failures (u), and running failures (r). Figure 7.1 shows a diagram of the booting process and fallbacks with the failures that were tested.

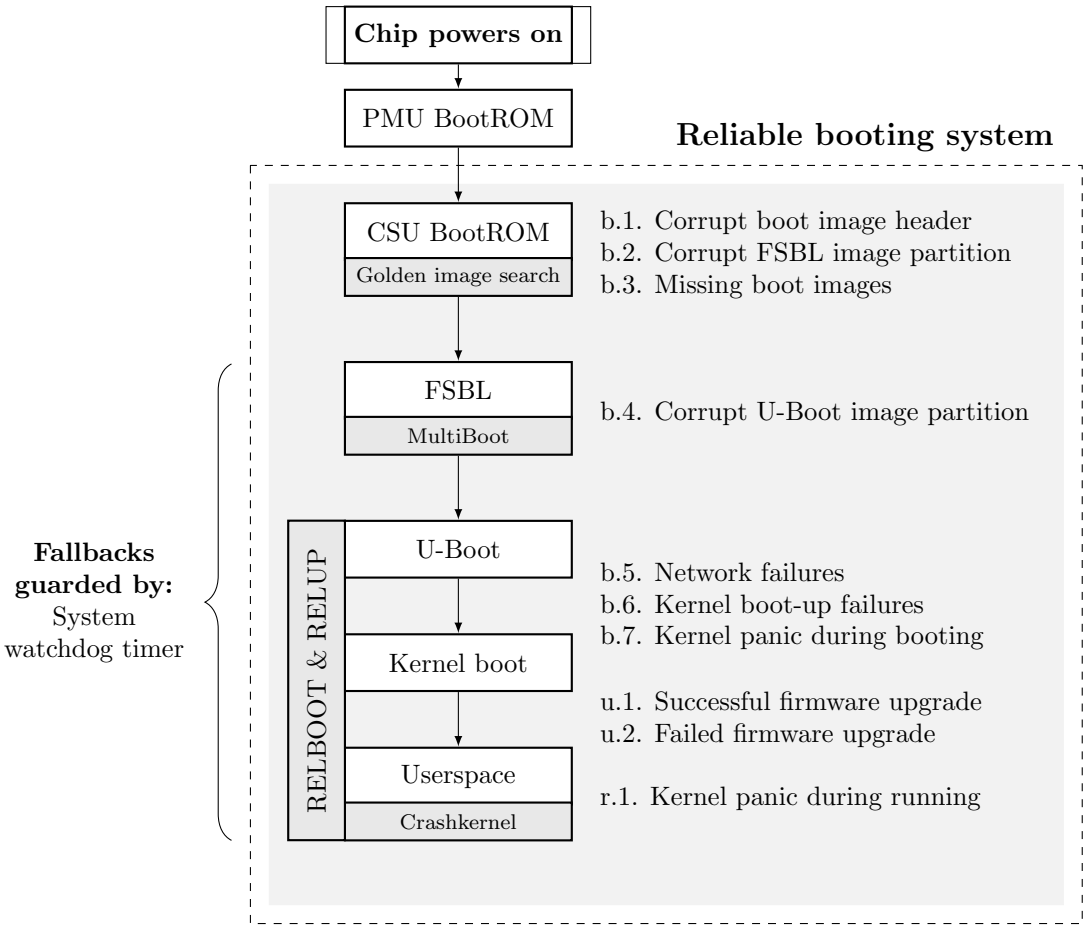


Figure 7.1: Zynq MPSoC booting diagram with implemented fallbacks and a summary of tested failure scenarios.

Sections 7.2 to 7.5 show how each fallback in the reliable booting system was tested. The failure indicators in Figure 7.1 (b.1 to r.1) are reflected in the test plans and results of each fallback.

7.2 Golden image search and MultiBoot

7.2.1 Testing plan

To test the golden image search and MultiBoot mechanisms, multiple `BOOT.BIN` images need to be prepared with different offset values. One of the images will be untouched, while the others are corrupted in different parts of the image. The untouched image will have the highest offset value. This will force the golden image search mechanism to check the corrupted images first. A summary of the prepared boot images can be seen in Table 7.1. The image offsets are not contiguous on purpose. This will test the golden image search mechanism for detection of missing boot images (b.3).

Table 7.1 Boot images for golden image search and MultiBoot testing

#	Image filename	Description
b.1	BOOT0001.BIN	Boot header is corrupted.
b.2	BOOT0004.BIN	FSBL partition is corrupted.
b.4	BOOT0005.BIN	U-Boot partition is corrupted.
b.3	BOOT0007.BIN	Untouched boot image.

Corrupting the images has been done by opening each BOOT000X.BIN file in a text editor and changing some of the characters. The boot image contains binary information, but also some plain text. This text was used to identify the different parts of the boot image. The boot header identification string, FSBL partition, and U-Boot partition were identified and corrupted by changing some characters with the text editor. It was unsure which parts of the boot image belonged to the partition header, so the golden image search mechanism has not been tested with a corrupted partition header.

7.2.2 Results

The boot images that were prepared for the golden image search mechanism were copied to the SD-card boot device. The SD-card was inserted into the SD-card slot of the ZCU102 development board. Finally, the board was powered on and the debug messages of the FSBL were checked. The following test results have been collected (see Table 7.2):

Table 7.2 Results of the golden image search mechanism and MultiBoot tests.

#	Boot images on SD-card	Boot-up	Observations and assumptions
b.1	Six copies of BOOT0001.BIN numbered one to six, BOOT0007.BIN	Success	The FSBL successfully booted-up using BOOT0007.BIN. The CSU_MULTI_BOOT offset register was set to seven. The golden image search mechanism found that there were six images with a corrupted boot header and skipped them.
b.2	BOOT0004.BIN, BOOT0007.BIN	Failed	No debug output was printed on the console. The golden image search mechanism skipped BOOT0001.BIN which contains the corrupted boot header identification string. The Zynq MPSoC tried to boot with BOOT0004.BIN, but hanged in the corrupted FSBL partition.
b.3	BOOT0001.BIN, BOOT0007.BIN	Success	The FSBL successfully booted-up using BOOT0007.BIN. The CSU_MULTI_BOOT offset register was set to seven. The Zynq MPSoC continued by booting into Linux and finished the boot-up successfully.
b.4	BOOT0005.BIN, BOOT0007.BIN	Partial	The FSBL booted-up using BOOT0005.BIN. The CSU_MULTI_BOOT offset register was set to five. The ATF got loaded correctly. The Zynq MPSoC hanged in U-Boot with some randomly printed messages.

The tests found that the golden image search mechanism is able to successfully skip a boot image with a corrupted boot header (b.1). Furthermore, it can also find a correct boot image with a valid boot header without the need for images to have filenames that are contiguous. This is apparent in test b.3, where the mechanism skips BOOT0001.BIN and goes straight to BOOT0007.BIN.

Another observation is that the golden image search mechanism did not protect against a corrupted FSBL partition. The flowchart of the Zynq MPSoC booting process (see Appendix A) indicates that the FSBL can be checked for authenticity. If the FSBL does not pass the authentication test, the golden image search mechanism should try and boot with the next boot image.

Upon further investigation, it was discovered that the FSBL authentication is not enabled during the creation of the `BOOT.BIN` image in PetaLinux. This was found in the Binary Image Format (BIF) file. A BIF file describes how a boot image for the Zynq MPSoC should look like. It defines which binaries should be stitched together during the creation of the boot image. It also allows the user to add *attributes*, which alter how the Zynq MPSoC boots-up [103].

The BIF file is used by the Bootgen utility to create a boot image [103]. PetaLinux uses the Bootgen utility when creating a boot image. Figure 7.2 shows the default BIF file of a PetaLinux project:

```
the_ROM_image:
{
    [bootloader, destination_cpu=a53-0] zynqmp_fsbl.elf
    [pmufw_image] pmufw.elf
    [destination_cpu=a53-0, exception_level=el-3, trustzone] bl31.elf
    [destination_cpu=a53-0, exception_level=el-2] u-boot.elf
}
```

Figure 7.2: Default BIF file in a PetaLinux project (v2019.2).

Each binary that is present in the tested boot images is included in the BIF file. Figure 7.2 shows that the image partitions do not have any authentication or checksums enabled. Authentication can be enabled for any partition by adding the `authentication` attribute [103]. This will require the user to also provide a set of encryption keys [104], which need to be created manually. In addition, a boot image with authentication can only be made by creating a custom BIF file and running the Bootgen utility. PetaLinux does not allow the project BIF file to be modified.

During the test b.4, it was found that the FSBL copied the U-Boot partition into memory and handed off control without validating the contents. This resulted in a corrupted version of U-Boot taking control and hanging the Zynq MPSoC. The MultiBoot mechanism did not detect the corrupt U-Boot partition.

The FSBL is able to perform checksums and authentications for every partition that it loads. This is shown in the partition validation flowchart of the FSBL (see Appendix C.2). If a partition is not valid, the FSBL will use the MultiBoot mechanism by running its *error lock down function*. This function first checks if the used boot mode supports MultiBoot. If it does, the function will increment the `CSU_MULTI_BOOT` offset register and perform a system reset. Else, it will go hang the chip by issuing the WFE instruction.

Test b.3 confirmed that the MultiBoot mechanism does not work unless authentication is enabled for the image partitions that are loaded by the FSBL. The authentication of image partition in the FSBL is enabled through the partition attributes of the BIF file [45]. Just like enabling authentication of the FSBL, enabling authentication for the other partitions will require a set of encryption keys.

The encryption of the FSBL and other image partitions does not come for free. It can create an overhead when making new images. Debugging may not be possible because of the extra security that is added. Enabling these features will need additional investigation and can be a topic for follow-up research.

7.3 RELBOOT & RELUP mechanisms

7.3.1 Testing plan

The RELBOOT and RELUP mechanisms were tested by using multiple boot failures. The tests were executed with separate boot images (Image, DTB, and initramfs). The maximum amount of boot attempts were set to three. All the other RELBOOT and RELUP configuration options were kept at their default values. Table 7.3 shows a list of tests for the RELBOOT and RELUP mechanisms.

Table 7.3 Testing plan for the RELBOOT & RELUP mechanisms.

#	Test	Description
b.5	DHCP check unsuccessful	Ethernet cable is not connected to the board.
b.5	TFTP check unsuccessful	TFTP server is not running.
b.5	Ramdisk image not retrieved from TFTP server	Ramdisk image is removed from the `boot_current` directory on the TFTP server.
b.6	Corrupt boot image	The kernel image is corrupted.
b.7	Kernel panics while booting	NFS server is not running. The crashkernel was disabled for this test.
u.1	Successful firmware upgrade	The `boot_current` and `boot_new` directories on the TFTP server have the same boot images. The `fw_version` variable in the version files are different.
u.2	Failed firmware upgrade	The device-tree of the new version will be altered. The `root=` option in the device-tree will be set to a device that does not exist.

7.3.2 Results

The test results from the RELUP mechanism are presented in Table 7.4. The supporting debug output of every test can be found in the attached ZIP-archive (see Appendix J for more information).

Table 7.4 Results from RELBOOT & RELUP mechanism tests.

#	Test	Result
b.5	DHCP check unsuccessful	The DHCP check failed. The DHCP request was retried multiple times before timing out. The Zynq MPSoC started booting with the SD-card backup images, but got reset by the watchdog timer before being able to start the watchdog timer heartbeat daemon. This caused an infinite reboot cycle.
b.5	TFTP check unsuccessful	The TFTP check failed and did not retrieve the dummy file from the TFTP server. The Zynq MPSoC booted with the SD-card backup images. The RELBOOT & RELUP daemon reported the SD-card backup boot through email.
b.5	Ramdisk image not retrieved from TFTP server	When trying to boot, The RELBOOT & RELUP script returned *"Wrong Ramdisk Image Format. Ramdisk image is corrupt or invalid"* and dropped to U-Boot's CLI. The Zynq MPSoC was reset three times by the watchdog timer, until the global boot counter exceeded its threshold. The chip was booted with the SD-card backup images. The RELBOOT & RELUP daemon reported the SD-card backup boot through email.
b.6	Corrupt kernel image	The RELBOOT & RELUP script tried to boot the kernel, but failed and dropped to U-Boot's CLI. The Zynq MPSoC was reset three times by the watchdog timer, until the global boot counter exceeded its threshold. The chip was booted with the SD-card backup images. The RELBOOT & RELUP daemon reported the SD-card backup boot through email.
b.7	Kernel panics while booting	The kernel tried to mount the NFS, but failed and panicked. The watchdog timer rebooted the Zynq MPSoC. The global boot counter exceeded its threshold after three reboots. The board booted with the SD-card backup images. The RELBOOT & RELUP daemon reported the SD-card backup boot through email.

Table 7.4 Results from RELBOOT & RELUP mechanism tests.

#	Test	Result
u.1	Successful firmware upgrade	The RELUP mechanism started an upgrade. It attempted to boot with the new firmware version and succeeded. The RELBOOT & RELUP daemon detected a successful upgrade. An email with a successful firmware upgrade report was received.
u.2	Failed firmware upgrade	The RELUP mechanism attempted to boot with the new firmware version, but failed. The kernel panicked when trying to mount a root filesystem that does not exist. The watchdog timer rebooted the Zynq MPSoC. This was repeated three times until the RELUP boot counter eventually exceeded its threshold. RELUP rolled back and booted the system with the previous firmware version. The RELBOOT & RELUP daemon reported a failed upgrade through email.

The RELBOOT and RELUP mechanisms were able to detect every failure that was part of the testing plan. The test with the unsuccessful DHCP request resulted in an infinite reboot cycle. The DHCP request was repeated multiple times before timing out. This process took around 30 seconds. The RELBOOT & RELUP script tried to use the SD-card backup-boot, but the watchdog timer was triggered before the system could finish booting. Systemd did not have enough time to start the heartbeat daemon before the watchdog timer expired.

After further investigation, it was discovered that the amount of DHCP request retries in U-Boot is configurable. It can be configured using the `CONFIG_NET_RETRY_COUNT` option. The U-Boot source code also states that DHCP request timeout duration can be calculated (see Figure 7.3).

```
1 #define TIMEOUT_MS    ((3 + (TIMEOUT_COUNT * 5)) * 1000)
```

Figure 7.3: Calculation of the timeout duration, based on the amount of retries [105]

The value of `TIMEOUT_COUNT` is equal to the value `CONFIG_NET_RETRY_COUNT`. If the latter is not defined, it will set the timeout count definition to five retries. Five retries results in a timeout duration of 28 seconds. This is consistent with the time that was measured during the testing.

The multiple DHCP request retries are useful for the RELBOOT mechanism. An accidental DHCP request fail from, let's say, a "network glitch", will not boot the Zynq MPSoC using the SD-card backup images immediately. The timeout duration of the watchdog timer was increased from 60 seconds to 180 seconds. This allows for more *"breathing room"* when booting and prevents the infinite reboot cycle. The watchdog timer timeout was increased using the `RECOVERY_TIMEOUT` flag in the PMU firmware [62].

7.4 Crashkernel

The crashkernel can be tested by crashing the system manually. This can be done by using the *magic* SysRq key. SysRq is a key combination that will give the user direct access to low-level commands on the kernel [106]. The kernel will respond to these commands regardless of what it is doing. The magic SysRq key can be enabled and accessed through the `/proc` filesystem. Figure 7.4 shows how to enable all SysRq functions and send a command to the kernel:

```
1 $ echo 1 > /proc/sys/kernel/sysrq
2 $ echo c > /proc/sysrq-trigger
```

Figure 7.4: Commands for enabling SysRq functions and triggering a kernel panic.

The `/proc/sys/kernel/sysrq` file controls the functions that are allowed through the SysRq key. Writing a "1" to the file enables all functions of SysRq [106].

When writing a character to the `/proc/sysrq-trigger` file, it sends a command to the kernel. A kernel panic can be triggered manually by writing the "c" character [106] (r.1, see Section 7.1). The panic will be triggered by dereferencing a `NULL` pointer[1]. After triggering the kernel panic, debug info about the crash is printed and the crashkernel is started. The console output is given in Appendix E.4.

The main system kernel detects the `NULL` pointer dereference and proceeds to start the crashkernel. The crashkernel mounts the CentOS 8 ramdisk that was created by kdump. After fully booting up, the kdump vmcore saving service is started. Kdump first mounts the SD-card to the system. It then proceeds to try and save the kernel console messages (dmesg) to a text file, but it fails. Kdump indicates that this may come from some kexec bug. Further research on this is needed.

The crashkernel continues by starting the kdump-post script. Figure 7.5 shows the console messages of the implemented kdump-post script in the crashkernel. The script is able to successfully copy the dump files from the SD-card to the NFS server.

```
1  [KDUMP-POST]: kdump-post.sh started
2  [KDUMP-POST]: kdump.conf contents:
3  ...
4  [KDUMP-POST]: Finding the dump target...
5  [KDUMP-POST]: dump target UUID: 83b9a606-b4ac-40a4-96a9-a3a514a1fd8d
6
7  [KDUMP-POST]: Mounting SD-card...
8  /dev/mmcblk0  /dev/mmcblk0p1  /dev/mmcblk0p2
9  [   27.701168] EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode.
10 [KDUMP-POST]: SD-card mounted successfully
11
12 [KDUMP-POST]: vmcore was saved to:
13 /kdumproot/SDCARD/var/crash/127.0.0.1-2021-01-22-17:22:10/ on the SD-card.
14
15 [KDUMP-POST]: Creating vmcore-dmesg.txt
16
17 The dmesg log is saved to
18 /kdumproot/SDCARD/var/crash/127.0.0.1-2021-01-22-17:22:10//vmcore-dmesg.txt.
19
20 makedumpfile Completed.
21 [KDUMP-POST]: Saved vmcore-dmesg.txt successfully
22 [KDUMP-POST]: Removing old vmcore-dmesg-incomplete.txt...
23
24 [KDUMP-POST]: Mounting NFS...
25 [KDUMP-POST]: NFS mounted successfully
26 [KDUMP-POST]: Copying directory with crash dump to NFS...
27 [   30.192663] systemd[1]: Shutting down.
28 ...
29 [   30.668706] reboot: Restarting system
```

Figure 7.5: Console output of custom kdump-post script for saving dmesg and copying the dump file to NFS.

[1] In C programming, a `NULL` pointer points to a memory address that does not exist. Dereferencing a `NULL` pointer means trying to access the data that is stored on that address [107]. This causes an undefined behaviour.

The UUID of the SD-card is retrieved and the SD-card is successfully mounted. The `makedumpfile` utility is also able to successfully save the dmesg of the system. The `vmcore-dmesg.txt` file contains the console messages of the main system kernel, verifying it was saved successfully.

The crash dump directory is also copied to the NFS server. The NFS server was checked to see if the crash dump was successfully copied over. The dump files on the SD-card and the NFS were identical indicating that the script works without any problems. Kdump rebooted the Zynq MPSoC after the kdump-post script was finished.

7.4.1 Early kdump testing

The main system kernel was booted using the early kdump initramfs. Early kdump loaded the crashkernel image and its ramdisk into the reserved section of memory (see Figure 7.6):

```
$ journalctl -x | grep early-kdump
Jan 22 17:22:44 zcu102 dracut-cmdline[1775]: early-kdump is enabled.
Jan 22 17:22:46 zcu102 dracut-cmdline[1775]: kexec: loaded early-kdump kernel
```

Figure 7.6: Zynq MPSoC console messages of early kdump loading the crashkernel

To test if early kdump works, a systemd service was created to trigger a kernel panic using the SysRq magic key. This service was configured to start before the kdump service on the NFS root filesystem. After enabling the panic trigger service and rebooting the Zynq MPSoC, the main system kernel was crashed during booting. The crashkernel was able to boot-up, capture a dump, and reboot the chip automatically.

7.5 Watchdog timer

7.5.1 Testing plan

The watchdog timer is able to reset the system at any time after it has been initialized by the FSBL. The watchdog timer workings were already verified during the testing of the RELBOOT & RELUP mechanisms (see Section 7.3). An additional test plan for the watchdog timer is shown in Table 7.5:

Table 7.5 Testing plan for testing the system watchdog timer.

#	Test	Description
b.4	U-Boot fails to start	The FSBL loads U-Boot into memory. U-Boot fails to start. A boot image with a corrupted U-Boot partition is used.
b.6	U-Boot fails to start kernel	The device-tree blob has been removed from the TFTP server to prevent the kernel from booting.
b.7	Booting kernel panic	The Linux kernel panics while booting. The NFS server was turned off so that the root filesystem could not be mounted. Early kdump is disabled for this test.
r.1	Kernel panic during running	A kernel panic is triggered manually. The crashkernel is disabled for this test.
r.1	Crashkernel panic	The main system kernel is panicked manually. The crashkernel is started, but it also panics and halts the system. This has been done by utilizing the kdump-pre script and triggering a panic using the magic SysRq key.

7.5.2 Results

During every test, the watchdog timer triggered successfully and reset the Zynq MPSoC. The test with the corrupted U-Boot image resulted in an infinite reboot cycle. The Zynq MPSoC was not able to get to a booted state. The results from the watchdog timer tests are presented in Table 7.6.

Table 7.6 Results from the system watchdog timer tests.

#	Test	Result
b.4	U-Boot fails to start	The FSBL loaded the ATF and U-Boot successfully. The Zynq MPSoC hung in U-Boot with some randomly printed messages. The watchdog timer expired and rebooted the Zynq MPSoC. This caused a infinite reboot cycle.
b.6	U-Boot fails to start kernel	The kernel was not started because the device-tree image was not successfully retrieved from the TFTP server. The RELBOOT & RELUP script dropped to U-Boot's CLI and the watchdog timer expired causing a reset. This was repeated three times until the RELUP script decided to boot the system using the SD-card backup images.
b.7	Booting kernel panic	The kernel panicked during the booting, because it could not mount the root filesystem via NFS. The watchdog timer expired and the system was rebooted. This was repeated three times until the RELBOOT & RELUP script decided to boot the system using the SD-card backup images.
r.1	Kernel panic during running	The kernel panicked and hung the system. The watchdog heartbeat daemon stopped and eventually the watchdog timer expired. The Zynq MPSoC was reset and booted successfully with boot images from the TFTP server.
r.1	Crashkernel panic	The main system kernel panicked and the crashkernel was started. The crashkernel panicked as well after running the prekdump script. The watchdog timer eventually expired, causing a reset. The Zynq MPSoC rebooted successfully with boot images from the TFTP server..

There is a possibility to implement a fallback that will recover the Zynq MPSoC after it hangs when starting U-Boot. The watchdog timer handler in the PMU firmware can be modified to change the behavior of the recovery scheme.

The proposed fallback would check the healthy bit in the PMU global register to see if Linux was started during the previous boot. If Linux was not started, it means that the Zynq MPSoC has failed in a previous booting stage. The fallback in the PMU firmware could increment the `CSU_MULTI_BOOT` offset register that is used by the golden image search mechanism. After rebooting, the chip would use another boot image which may not be corrupted.

7.6 Summary of test results

Each fallback in the reliable booting system was tested by emulating a set of failures. The failures in each stage of the booting process were based on the research that was conducted (see Section 4.2). The failures types are based on the requirements:

1. The Zynq MPSoC can recover from a boot failure;
2. The Zynq MPSoC can recover from a failed upgrade;
3. The Zynq MPSoC can recover from a running failure.

Each failure requirement was met. The booting system was also able to recover the Zynq MPSoC from failures in the pre-boot stage. A corrupt boot image header and missing boot images are detected by the golden image search mechanism. Table 7.7 shows a summary of the tests and results.

Table 7.7 Testing plan for testing the system watchdog timer.

#	Test	Fallbacks	Passed
b.1	Corrupt boot image header	Golden image search	✓
b.2	Corrupt FSBL image partition	Golden image search	X
b.3	Missing boot images	Golden image search	✓
b.4	Corrupt U-Boot image partition	MultiBoot	X
b.5	Network failures	RELBOOT	✓
b.6	Kernel boot-up failures	RELBOOT, watchdog timer	✓
b.7	Kernel panic failures	RELBOOT, watchdog timer	✓
u.1	Successful firmware upgrade	RELUP	✓
u.2	Failed firmware upgrade	RELUP, watchdog timer	✓
r.1	Kernel panic during running	Crashkernel, watchdog timer	✓

It was discovered that the FSBL partition in the boot image does not get authenticated by the golden image search mechanism by default. The MultiBoot mechanism is also not triggered when the U-Boot partition in the boot image is corrupt. Both of these issues can be solved by enabling authentication of the boot image partitions. This encrypts each partition in the boot image. The encryption may cause debugging to be impossible. This requires further investigation and can be researched in a follow-up project.

The RELBOOT & RELUP fallbacks worked together with the system watchdog timer to recover from boot and upgrade failures. RELBOOT recovered the Zynq MPSoC from multiple network failures including a failed DHCP check and failed TFTP file retrieval. Network failures resulted in a backup-boot using the boot images and root filesystem on the SD-card.

The kernel was panicked during booting to test the ability of the RELBOOT mechanism to count the boot attempts and boot from the backup images on the SD-card. This worked well in combination with the system watchdog timer, which would reset the board after a kernel panic.

Furthermore, the crashkernel recovers the system from a running failure when the kernel panics. The dump files are saved locally on the SD-card instead of the NFS server. The crashkernel does copy the dump files to the NFS server using a custom kdump-post script. This is beneficial, because the dump files will always be saved, even if the NFS server fails.

The booting system was also tested by disabling the crashkernel and causing a crash. The Zynq MPSoC would get rebooted by the system watchdog timer. The same happened when both the main system kernel and the crashkernel were panicked.

Overall, it can be concluded that the reliable booting system is able to successfully recover the Zynq Ultrascale+ MPSoC from booting failures, upgrade failures and running failures. It is unsure if the authentication features of the golden image search and MultiBoot mechanisms should be used, as they may make debugging of the Zynq MPSoC impossible. This requires further research. Apart from that, the reliable booting system provides recovery from each failure that was researched during the project.

8. Conclusion

During the High-Luminosity upgrade of the CMS-experiment, its data acquisition system will introduce new electronics which host a Zynq UltraScale+ MPSoC. This embedded system will be used to perform the control and monitoring of these electronics. The control and monitoring tasks will be performed in a Linux operating system that is running on the Zynq MPSoC.

The research during this project found that booting Linux on the Zynq MPSoC requires a complicated multi-stage booting process. The complexity of the boot-up introduces possible failures that can prevent the system from booting correctly. A reliable booting system was successfully researched, designed, implemented, and tested to tackle this problem. The booting system includes five fallbacks that are able to recover the Zynq MPSoC from booting failures, upgrade failures and running failures. The fallbacks have been implemented in different stages of the booting process, to cover a wider range of failures:

1. The golden image search mechanism is able to protect the Zynq MPSoC from invalid boot images.
2. The reliable booting (RELBOOT) mechanism is able to recover the system from various boot failures. It can boot the Zynq MPSoC using a set of backup images when multiple failed boot attempts are detected.
3. The reliable upgrade (RELUP) mechanism is able to provide the ability to perform firmware upgrades. The mechanism can automatically detect new firmware versions and attempt an upgrade. If the upgrade is unsuccessful, the mechanism will automatically roll back to a previous firmware version.
4. The crashkernel mechanism is able to recover the Zynq MPSoC in case of a running failure that crashes Linux. The mechanism is able to save a dump of the crash and reboot the Zynq MPSoC.
5. A general fallback mechanism based on the watchdog timer in the Zynq MPSoC is able to recover the system from failures that have not been anticipated. The watchdog timer is able to reset the system during any stage of the boot-up after its initialization.

All the fallbacks in the reliable booting system are packaged together into a board support package (BSP). The BSP structure includes the sources that have to be updated by the developer to port the reliable booting system to different hardware. The reliable booting system and BSP also include a network-boot, as required by the project requirements. An evaluation matrix of the project requirements and solutions is given in Figure 8.1:

Table 8.1 Evaluation matrix of the project requirements and project solutions.

#	MoSCoW	Requirements \ Solution	Golden image search	RELBOOT	RELUP	Crashkernel	Watchdog timer	Network boot	Board Support Package	GitLab CI	
1	**Must**	The Zynq MPSoC can recover from a boot failure.	✓	✓				✓			
2	**Must**	The Zynq MPSoC can recover from a failed upgrade.			✓			✓			
3	**Must**	The Zynq MPSoC can recover from a running failure.				✓	✓	✓			
4	**Must**	The Zynq MPSoC boots through the network by default.							✓		
5	**Must**	The reliable booting system is portable to new hardware.								✓	
6	**Should**	Each fallback reports to the user about a failure if possible	✓	✓	✓	✓					
7	**Should**	The Linux distribution and fallbacks are automatically built by a CI.									✓
8	**Could**	The Linux distribution and fallbacks are automatically tested by a CI.									–

All requirements with the *must* and *should* priorities have been met. The RELBOOT, RELUP, and crashkernel fallbacks can report failures to the user through email. The golden image search fallback informs the user through debug output on the console.

Automated building of the board support package has been implemented through a continuous integration (CI) in GitLab. The CI is able to build each component that is required by the booting process separately. Research on automated testing with CI was started, however due to time constrains it is not presented in this thesis and remains as a topic for the future work.

On the suggestion of the CERN supervisors, this thesis includes detailed information such that it can be used as learning material. The thesis concentrates widespread documentation into a single document allowing anyone to start developing for the Zynq Ultrascale+ MPSoC. In addition, documentation about the reliable booting system has been written in GitLab.

The research and developments of the project have been shared with engineers and scientists at CERN and other institutes. Developments have been presented at two SoC interest group meetings during the project [108, 109]. It can be concluded that the project was finished successfully.

9. Future work

It has been concluded that the reliable booting system is able to successfully recover the Zynq Ultrascale+ MPSoC from booting failures, upgrade failures and running failures.

It was discovered during the testing of the golden image search mechanism, that the CSU BootROM does not perform authentication of the FSBL boot image partition. Authentication can be enabled in the CSU BootROM. Furthermore, the FSBL itself does not perform validation of the boot image partitions that it loads. Validation of the boot image partitions can also be enabled FSBL. Further research is needed to enable both authentication and validation.

Table 9.1 shows a list of research topics for a follow-up project of the reliable booting system.

Table 9.1 List of future work for a follow-up project.

#	Improvement / Research topic
1	Authentication of the FSBL boot image partition by the CSU BootROM.
2	Validation of boot image partitions that are loaded by the FSBL.
3	Switching to a second boot device in the FSBL/PMU firmware.
4	Implementation of a testing CI for the reliable booting system with QEMU.
5	Porting from PetaLinux 2019.2 to the newest version of PetaLinux.

There is a possibility to add a fallback in the FSBL/PMU firmware, that can switch to a second boot device if the Zynq MPSoC does not boot-up fully with the images on the SD-card. The failure may come from a hardware issue or a corrupted U-Boot partition. Boot images on a second boot device can be used as a backup.

Automated testing can be implemented by using the QEMU (Quick EMUlator) emulator [110]. This emulator can be used in combination with a testing script to create a CI to automatically test the booting system. The script can analyze the console output of the QEMU emulator to determine if the reliable booting system is functioning correctly. This will need further research.

Finally, the board support package for the reliable booting system was made using PetaLinux v2019.2. Xilinx releases new versions of the PetaLinux tools every year. The BSP should be ported to the newest version of PetaLinux.

10. Extra work during the project

The project was carried out during the global COVID-19 pandemic. This forced CERN to close the doors of the laboratory and require a majority of the staff to work remotely from home. During this time, the Zynq MPSoC hardware could not be accessed for the project. A set of extra developments was made to use the hardware remotely. Table 10.1 shows a list of extra work that was done for the reliable booting project.

Table 10.1 List of extra work that was carried out during the project.

#	Supporting work
1	Remote JTAG booting through a Xilinx hardware server and TCL script.
2	Custom IO-board created with Arduino. Provides GPIO to interact with hardware through a serial connection.
3	Zynq MPSoC boot mode switching hardware, created with the IO-board.
4	Python script that interacts with the IO-board to switch the boot mode of the Zynq MPSoC.
5	Zynq MPSoC boot mode switching by writing the boot mode register through Linux.
6	CSU error register reading through kernel module in Linux.
7	Setup of a power distribution unit (PDU) for the Zynq MPSoC.

A TCL script [111] was created to boot the Zynq MPSoC in the CMS DAQ lab remotely through JTAG. The script sends commands and binary files over the network to a Xilinx remote hardware server. The server runs in the lab and is connected to the Zynq MPSoC hardware through a JTAG cable. The TCL script is executed with the XSCT command line tool of Xilinx.

In addition to remote JTAG booting, a custom IO-board was created using an Arduino. The IO-board is used to switch the boot mode of the Zynq MPSoC on the ZCU102 development board. This is normally done using four DIP-switches (dual in-line package) on the board. The IO-board is connected to the boot mode pins of the Zynq MPSoC. Level-shifters are used for voltage translation from 5 V to 1.8 V.

The IO-board contains custom firmware that provides a CLI through the serial port of the Arduino. The CLI supports local echo for the user and line editing. The firmware offers commands for reading, writing and masking GPIO. Error handling has also been implemented into the firmware to make the IO-board robust. The IO-board was designed as a standalone tool that can be used by the CMS DAQ team in other projects as well.

The CLI of the IO-board can be accessed through a serial connection on the server in the lab. The server can be accessed remotely through an SSH connection. A python script was created to change the boot mode of the Zynq MPSoC remotely. The script send commands to the IO-board through the serial connection to change the voltage levels of the boot mode pins.

It was also researched how to access the boot mode register of the Zynq MPSoC through Linux. Soft boot mode switching was achieved by writing the boot mode register of the Zynq MPSoC using the `devmem` utility. Furthermore, the CSU error registers were are also accessed from Linux to read potential error codes of the booting process.

The ZCU102 development board can also be power cycled using a power distribution unit (PDU). The PDU was setup with the help of the CERN supervisors. It allows for remote power cycling through telnet.

List of Figures

List of Tables

Abbreviations

ABRT	Automatic Bug Reporting Tool
ALICE	A Large Ion Collider Experiment
API	Application Programming Interface
APU	Application Processing Unit
ARM	Advanced RISC Machines
ATCA	Advanced Telecommunications Computing Architecture
ATF	ARM trusted firmware
ATLAS	A Toroidal LHC Apparatus
BIF	Binary Image Format
BISR	Built-In Self Repair
BSP	Board Support Package
BU	Builder Unit
CAN	Controller Area Network
CBR	CMU BootROM
CERN	Conseil Européen pour la Recherche Nucléaire
CI	Continuous Integration
CIB	Crypto Interface Block
CLI	Command-Line Interface
CMS	Compact Muon Solenoid
CPU	Central Processing Unit
CSU	Configuration Security Unit
DAQ	Data Acquisition
DDR	Double Data Rate
DG	Director-General
DHCP	Dynamic Host Configuration Protocol
DIP	Dual In-line Package
DTB	Device-Tree Blob
DTH	Timing Contol Distribution System Hub
ECAL	Electromagnetic Calorimeter
EEPROM	Electrically Erasable Programmable Read-Only Memory
e.g.	Exempli gratia (for example)
EL	Exception Level
EM	Error Management
eMMC	embedded Multi-Media Card
EP-CMD	Experimental Physics - CMS DAQ & trigger department
FED	Front-End Driver
FEROL	Front-End Readout Link
FPD	Full-Power Domain
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit

FSBL	First-Stage Boot Loader
FU	Filter Unit
GEM	Gigabit Ethernet MAC
GPIO	General Purpose Input Output
GPU	Graphics Processing Unit
HU	Hogeschool Utrecht
HCAL	Hadron Calorimeter
HL-LHC	High-Luminosity Large Hadron Collider
HLT	High-Level Trigger
I^2C	Inter-Integrated Communication
I/O	Input-Output
IP	Internet Protocol
IPI	Inter-Processor Interrupt
JTAG	Joint Test Action Group
LEIR	Low Energy Ion Ring
LHC	Large Hadron Collider
LHCb	LHC beauty
LINAC	Linear Accelerator
LPD	Low-Power Domain
MAC	Media Access Control
MB	Megabyte
MBIST	Memory Built-In Self Test
MoSCoW	Must have, Should have, Could have, Won't have
MPSoC	Multiprocessor Systems on a Chip
MS	Muon Station
NFS	Network File System
OCM	On-Chip Memory
OS	Operating System
PBR	PMU BootROM
PCAP	Processor Configuration Access Port
PCI	Peripheral Component Interconnect
PDU	Power Distribution Unit
PL	Programmable Logic
PMU	Platform Management Unit
PS	Proton Synchrotron
PS	Processing System
PS SYSMON	PS Monitoring System
QEMU	Quick EMUlator
QSPI	Quad Serial Peripheral Interface

RAM	Random-Access Memory
RELBOOT	Reliable Booting
RELUP	Reliable Upgrade
RF	radio-frequency
RHEL	Red Hat Enterprise Linux
ROM	Read-Only Memory
RPU	Real-time Processing Unit
RTOS	Real-Time Operating System
RU	Readout Unit
SATA	Serial Advanced Technology Attachment
SCU	Snoop Control Unit
SD	Secure Digital
SHA	Secure Hash Algorithms
SMMU	System Memory Management Unit
SoC	System on Chip
SPB	Secure Processor Block
SPI	Serial Peripheral Interface
SPS	Super Proton Synchrotron
SRST	System Reset
SSH	Secure Shell
TCDS	Timing Contol Distribution System
TCL	Tools Command Language
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
UUID	Universally Unique Identifier
WFE	Wait for Event
WDT	WatchDog Timer
XSCT	Xilinx Software Command-line Tool

Bibliography

[1] CERN, "Our people." *https://home.cern/about/who-we-are/our-people*, August 2020. Accessed: 07/09/2020.

[2] J. Guillaume and CERN, "Wandering the immeasurable." *https://cds.cern.ch/record/1957174?ln=en*, October 2014. Accessed: 20/11/2020.

[3] CERN, "Our history." *https://home.cern/about/who-we-are/our-history*, August 2020. Accessed: 26/08/2020.

[4] CERN, "Advancing the frontiers of technology." *https://home.cern/about/what-we-do/our-impact*, August 2020. Accessed: 26/08/2020.

[5] M. Brice and CERN, "3d view photo of the lhc machine." *https://cds.cern.ch/record/1223589?ln=en*, November 2009. Accessed: 02/09/2020.

[6] CERN, "A vacuum as empty as interstellar space." *https://home.cern/science/engineering/vacuum-empty-interstellar-space*, July 2020. Accessed: 01/09/2020.

[7] CERN, "Accelerating: radiofrequency cavities." *https://home.cern/science/engineering/accelerating-radiofrequency-cavities*, August 2020. Accessed: 16/10/2020.

[8] E. Mobs and CERN, "The cern accelerator complex - 2019." *https://cds.cern.ch/record/2684277*, July 2019. Accessed: 25/09/2020.

[9] CERN and CMS, "About cms." *https://cms.cern/detector*, 2020. Accessed: 30/10/2020.

[10] CERN and CMS, "3d-model of the cms detector." *https://cms.cern/sites/cmsexperiment.web.cern.ch/files/cms_160312_02.png*, September 2020. Accessed: 04/09/2020.

[11] CERN and CMS, "Cms tracker." *https://cms.cern/detector/identifying-tracks*, 2020. Accessed: 30/10/2020.

[12] CERN and CMS, "Measuring energy." *https://cms.cern/detector/measuring-energy*, 2020. Accessed: 30/10/2020.

[13] CERN and CMS, "Energy of electrons and photons (ecal)." *https://cms.cern/detector/measuring-energy/energy-electrons-and-photons-ecal*, 2020. Accessed: 30/10/2020.

[14] CERN and CMS, "Energy of hadrons." *https://cms.cern/detector/measuring-energy/energy-hadrons-hcal*, 2020. Accessed: 30/10/2020.

[15] R. Curley, "Hadron." *https://www.britannica.com/science/hadron*, 2020. Accessed: 23/10/2020.

[16] S. R. Davis and CERN, "Interactive slice of the cms detector." *https://cms.cern/sites/cmsexperiment.web.cern.ch/files/cms_160312_02.png*, August 2016. Accessed: 04/09/2020.

[17] A. L. Hallin, "What is a neutrino?." *https://www.scientificamerican.com/article/what-is-a-neutrino/*, September 1999. Accessed: 23/10/2020.

[18] CERN and CMS, "Detecting muons." *https://cms.cern/detector/detecting-muons*, 2020. Accessed: 30/10/2020.

[19] R. Curley, "Muon." *https://www.britannica.com/science/muon*, 2020. Accessed: 23/10/2020.

[20] CERN and CMS, "Detecting muons." *https://cms.cern/detector/triggering-and-data-acquisition*, 2020. Accessed: 30/10/2020.

[21] "The phase-2 upgrade of the cms daq interim technical design report," Tech. Rep. CERN-LHCC-2017-014. CMS-TDR-018, CERN, Geneva, September 2017.

[22] Oracle, "Pci local bus." *https://docs.oracle.com/cd/E19683-01/806-5222/hwovr-22/*, 2010. Accessed: 23/11/2020.

[23] Z. Shen, "Block diagram of xilinx zynq ultrascale+ mpsoc device." *https://www.researchgate.net/figure/Block-diagram-of-Xilinx-Zynq-UltraScale-MPSoC-device_fig1_327171284*, 2018. Accessed: 23/11/2020.

[24] CentOS, "About centos linux." *https://www.centos.org/about/*, 2020. Accessed: 24/09/2020.

[25] Xilinx, *Zynq Ultrascale+ Device Technical Reference Manual, Chapter 11: Boot and Configuration*, August 2019. UG1085, v2.1.

[26] K. Brennan, *A Guide to the Business Analysis Body of Knowledge, 2nd edition*, ch. 6.1.5.2. MoSCoW analysis. International Institute of Business Analysis, 2009.

[27] ARM-Mbed, "Watchdog timer." *https://os.mbed.com/cookbook/WatchDog-Timer*, October 2020. Accessed: 29/10/2020.

[28] European-Cooperation-for-Space-Standardization-(ECSS), *Space product assurance: Techniques for radiation effects mitigation in ASICs and FPGAs handbook*, September 2016. ECSS-Q-HB-60-02A.

[29] Xilinx, *Zynq Ultrascale+ MPSoC Software Developer Guide, Chapter 7: System Boot and Configuration*, July 2020. UG1137, v12.0.

[30] Xilinx, *Zynq Ultrascale+ MPSoC Software Developer Guide, Boot Flow*, July 2020. UG1137, v12.0.

[31] Xilinx, *Zynq Ultrascale+ MPSoC Data Sheet: Overview*, October 2019. DS891, v1.8.

[32] Xilinx, *Zynq Ultrascale+ MPSoC Software Developer Guide, Hardware Architecture Overview*, July 2020. UG1137, v12.0.

[33] G. Torres, "How the cache memory works." *http://www.hardwaresecrets.com/how-the-cache-memory-works/*, 2007. Accessed: 29/10/2020.

[34] Xilinx, *Zynq Ultrascale+ Device Technical Reference Manual, Chapter 3: Application Processing Unit*, August 2019. UG1085, v2.1.

[35] G. Shute, "Cache coherence." *https://www.d.umn.edu/~gshute/arch/cache-coherence.xhtml*, 2007. Accessed: 26/10/2020.

[36] Xilinx, *Zynq Ultrascale+ Device Technical Reference Manual, Chapter 28: Multiplexed I/O*, August 2019. UG1085, v2.1.

[37] Xilinx, *Zynq Ultrascale+ Device Technical Reference Manual, Chapter 6: Platform Management Unit*, August 2019. UG1085, v2.1.

[38] Xilinx, *Zynq Ultrascale+ MPSoC Software Developer Guide, Chapter 10: Platform Management Unit Firmware*, July 2020. UG1137, v12.0.

[39] Xilinx, *Zynq Ultrascale+ MPSoC Software Developer Guide, Chapter 9: Platform Management*, July 2020. UG1137, v12.0.

[40] Xilinx, *Zynq Ultrascale+ Device Technical Reference Manual, Chapter 12: Security*, August 2019. UG1085, v2.1.

[41] Xilinx, *Zynq Ultrascale+ MPSoC Software Developer Guide, Detailed Boot Flow*, July 2020. UG1137, v12.0.

[42] Xilinx, *Zynq Ultrascale+ Device Technical Reference Manual, Boot Image Format*, August 2019. UG1085, v2.1.

[43] Xilinx, *Zynq Ultrascale+ Device Technical Reference Manual, Golden Image Search*, August 2019. UG1085, v2.1.

[44] Xilinx, *Zynq Ultrascale+ MPSoC Software Developer Guide, Fallback and MultiBoot Flow*, July 2020. UG1137, v12.0.

[45] Xilinx, *Zynq Ultrascale+ MPSoC Software Developer Guide, Phases of FSBL Operation*, July 2020. UG1137, v12.0.

[46] Arm, *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*, January 2021. ARM DDI 0487, issue G.a.

[47] B. Levinsky, "Xilinx wiki: Arm trusted firmware." *https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842107/Arm+Trusted+Firmware*, June 2020. Accessed: 09/10/2020.

[48] ARM, "Privilege and exception levels." *https://developer.arm.com/architectures/learn-the-architecture/exception-model/privilege-and-exception-levels*, 2020. Accessed: 09/10/2020.

[49] J. Garcia, "Soc course with reference designs: Arm trusted firmware (atf)." *https://ohwr.org/project/soc-course/wikis/ARM-Trusted-Firmware-(ATF)*, June 2020. Accessed: 09/10/2020.

[50] Xilinx, *Enabling virtualization with Xen Hypervisor on Zynq Ultrascale+ MPSoCs*, March 2016. WP474, v1.0.

[51] D. Zundel, "U-boot documentation." *https://www.denx.de/wiki/view/DULG/UBoot*, 2008. Accessed: 27/10/2020.

[52] H. Beberman, "U-boot environment variables." *https://www.denx.de/wiki/view/DULG/UBootEnvVariables*, May 2018. Accessed: 13/10/2020.

[53] U-Boot, "Environment variables commands." *https://www.denx.de/wiki/view/DULG/UBootCmdGroupEnvironment*, May 2008. Accessed: 13/10/2020.

[54] U-Boot, "U-boot command line parsing." *https://www.denx.de/wiki/DULG/CommandLineParsing*, May 2007. Accessed: 25/01/2020.

[55] D. Zundel, "U-boot download commands." *https://www.denx.de/wiki/view/DULG/UBootCmdGroupDownload*, October 2012. Accessed: 26/10/2020.

[56] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation or the sun network filesystem." *https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.473*, 1985. Accessed: 13/10/2020.

[57] ARM, "Tftp server." *https://www.keil.com/pack/doc/mw/Network/html/group__net_t_f_t_ps___func.html*, July 2020. Accessed: 15/10/2020.

[58] D. Both, "An introduction to the linux boot and startup processes." *https://opensource.com/article/17/2/linux-boot-and-startup*, February 2017. Accessed: 13/10/2020.

[59] G. Likely, "The linux usage model for device tree data." *https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt*, July 2020. Accessed: 13/10/2020.

[60] W. Soyinka, *Linux Administration: A Beginner's Guide, Eighth Edition*, ch. 7. Booting and shutting down. McGraw-Hill, 2020.

[61] A. Kili, "5 best modern linux 'init' systems." *https://www.tecmint.com/best-linux-init-systems/*, August 2016. Accessed: 13/10/2020.

[62] Xilinx, "Zynq ultrascale+ mpsoc restart solution." *https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841820/Zynq+UltraScale+MPSoC+Restart+solution*, December 2020. Accessed: 29/01/2021.

[63] M. Kerrisk, "Daemon - linux manual page." *https://man7.org/linux/man-pages/man7/daemon.7.html*, December 2020. Accessed: 10/02/2021.

[64] RedHat, "Kernel crash dump guide." *https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/kernel_administration_guide/kernel_crash_dump_guide*, October 2020. Accessed: 28/10/2020.

[65] V. Goyal and M. Soni, "Documentation for kdump - the kexec-based crash dumping solution." *https://www.kernel.org/doc/Documentation/kdump/kdump.txt*, October 2020. Accessed: 20/01/2021.

[66] Arch-Linux-wiki, "Kexec." *https://wiki.archlinux.org/index.php/kexec*, August 2020. Accessed: 20/01/2021.

[67] M. Kerrisk, "Kexec - linux manual page." *https://man7.org/linux/man-pages/man8/kexec.8.html*, December 2020. Accessed: 20/01/2021.

[68] R. Love, *Linux Kernel Development, Third Edition*, ch. 18. Debugging. Addison-Wesley Professional, 2010.

[69] P. Anand and D. Young, "Redhat kdump: usage and internals." *https://events.static.linuxfound.org/sites/events/files/slides/kdump_usage_and_internals.pdf*, June 2017. Accessed: 20/01/2021.

[70] Arch-Linux-wiki, "Kdump." *https://wiki.archlinux.org/index.php/Kdump*, August 2020. Accessed: 20/01/2021.

[71] M. Tachibana and K. Ohmichi, "makedumpfile(8) - linux man page." *https://linux.die.net/man/8/makedumpfile*, January 2021. Accessed: 20/01/2021.

[72] RedHat, "What is early kdump support and how do i configure it?." *https://access.redhat.com/solutions/3700611*, May 2019. Accessed: 20/01/2021.

[73] RedHat, "Automatic bug reporting tool (abrt)." *https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/system_administrators_guide/ch-abrt*, January 2021. Accessed: 23/01/2021.

[74] Xilinx, "Zynq ultrascale+ mpsoc petalinux/yocto/linux: Is kdump supported for aarch64 (arm 64-bit architecture)?." *https://www.xilinx.com/support/answers/68865.html*, March 2017. Accessed: 20/01/2021.

[75] P. Cordes, "Differences between arm64 and aarch64." *https://stackoverflow.com/questions/31851611/differences-between-arm64-and-aarch64*, November 2017. Accessed: 20/01/2021.

[76] Xilinx, *Zynq Ultrascale+ Device Technical Reference Manual, CSU BootROM Error Codes*, August 2019. UG1085, v2.1.

[77] Xilinx, *Zynq Ultrascale+ Device Technical Reference Manual, Reset system*, August 2019. UG1085, v2.1.

[78] M. Rouse, "Dhcp (dynamic host configuration protocol)." *https://searchnetworking.techtarget.com/definition/DHCP*, December 2019. Accessed: 15/10/2020.

[79] K. development community, "The kernel's command-line parameters." *https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html*, October 2020. Accessed: 28/10/2020.

[80] Xilinx, *Zynq Ultrascale+ Device Technical Reference Manual, System Watchdog Timers*, August 2019. UG1085, v2.1.

[81] U-Boot, "Boot count limit." *https://www.denx.de/wiki/view/DULG/UBootBootCountLimit*, August 2009. Accessed: 24/01/2021.

[82] A. Huang, "On microsd problems." *https://www.bunniestudios.com/blog/?page_id=1022*, February 2010. Accessed: 31/01/2021.

[83] S. Larrivee, "Solid state drive primer 1 - the basic nand flash cell." *https://www.cactus-tech.com/resources/blog/details/solid-state-drive-primer-1-the-basic-nand-flash-cell/*, February 2015. Accessed: 01/02/2021.

[84] R. Micheloni, L. Crippa, and A. Marelli, *Inside NAND Flash Memories*, ch. 4. Reliability issues of NAND Flash memories. Springer, 2010.

[85] Indiana-University, "Create a symbolic link in unix." *https://kb.iu.edu/d/abbe*, August 2019. Accessed: 24/01/2021.

[86] T. Zanussi and R. Purdie, "Yocto project board support package (bsp) developer's guide." *https://www.yoctoproject.org/docs/1.1.1/bsp-guide/bsp-guide.html*, March 2012. Accessed: 18/01/2021.

[87] Xilinx, "Golden image search for zynq ultrascale+." *https://forums.xilinx.com/t5/ACAP-and-SoC-Boot-and/Golden-Image-Search-For-Zynq-Ultrascale/td-p/911191*, November 2018. Accessed: 20/01/2021.

[88] Xilinx, *Zynq Ultrascale+ MPSoC Software Developer Guide, Setting FSBL Compilation Flags*, July 2020. UG1137, v12.0.

[89] Xilinx, "Zynq ultrascale+ mpsoc fsbl." *https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842019/Zynq+UltraScale+FSBL*, May 2020. Accessed: 19/01/2021.

[90] telcoM, "Why do most systemd examples contain wantedby=multi-user.target?." *https://unix.stackexchange.com/questions/506347/why-do-most-systemd-examples-contain-wantedby-multi-user-target*, March 2019. Accessed: 11/02/2021.

[91] Systemd, "Service unit configuration." *https://www.freedesktop.org/software/systemd/man/systemd.service.html*, June 2020. Accessed: 11/02/2021.

[92] Xilinx, "How to add u-boot's printenv tool in petalinux." *https://forums.xilinx.com/t5/Embedded-Linux/How-to-include-U-Boot-s-quot-fw-printenv-quot-tool-in-Petalinux/td-p/770629*, March 2018. Accessed: 24/01/2021.

[93] S. Moon, "9 mailx command examples to send emails from command line on linux." *https://www.binarytides.com/linux-mailx-command/*, June 2020. Accessed: 26/01/2021.

[94] W. Venema, "The postfix home page." *http://www.postfix.org/start.html*, January 2021. Accessed: 26/01/2021.

[95] R. Freeman, "Kernel crash dumps." *https://wiki.gentoo.org/wiki/Kernel_Crash_Dumps*, February 2018. Accessed: 21/01/2021.

[96] T. Bowden, B. Bauer, J. Nerin, S. Feng, and S. Seibold, "The /proc filesystem." *https://www.kernel.org/doc/html/latest/_sources/filesystems/proc.rst.txt*, June 2009. Accessed: 21/01/2021.

[97] P. Mochel, "The sysfs filesystem." *https://www.kernel.org/doc/ols/2005/ols2005v1-pages-321-334.pdf*, July 2005. Accessed: 21/01/2021.

[98] Linux-Kernel-Driver-DataBase, "Config relocatable: Build a relocatable kernel image." *https://cateee.net/lkddb/web-lkddb/RELOCATABLE.html*, 2021. Accessed: 03/02/2021.

[99] RedHat, "How should the crashkernel parameter be configured for using kdump on rhel6?." *https://access.redhat.com/solutions/59432*, April 2018. Accessed: 21/01/2021.

[100] Xilinx, "Cadence wdt driver." *https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842055/Cadence+WDT+Driver*, January 2021. Accessed: 29/01/2021.

[101] C. Weingel and J. Oestergaard, "The linux watchdog driver api." *https://www.kernel.org/doc/Documentation/watchdog/watchdog-api.txt*, 2002. Accessed: 11/02/2021.

[102] Xilinx, "zynqmp.dtsi." *https://github.com/Xilinx/linux-xlnx/blob/master/arch/arm64/boot/dts/xilinx/zynqmp.dtsi*, 2020. Accessed: 29/01/2021.

[103] Xilinx, *Bootgen User Guide, Chapter 3: Creating Boot Images*, December 2020. UG1283, v2020.2.

[104] Xilinx, "Zynq ultrascale+ security features." *https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841708/Zynq+Ultrascale+MPSoC+Security+Features*, May 2020. Accessed: 20/01/2021.

[105] W. Denk, "bootp.c." *https://github.com/Xilinx/u-boot-xlnx/blob/master/net/bootp.c*, January 2021. Accessed: 28/01/2021.

[106] J. Dolan, "Linux and the device tree." *https://www.kernel.org/doc/html/latest/admin-guide/sysrq.html*, January 2001. Accessed: 22/01/2021.

[107] G. Hewgill, "What exactly is meant by de-referencing a null pointer?." *https://stackoverflow.com/questions/4007268/what-exactly-is-meant-by-de-referencing-a-null-pointer*, October 2014. Accessed: 22/01/2021.

[108] N. Dzemaili, "Zynq (mpsoc) crashkernel." *https://indico.cern.ch/event/921378/contributions/3922420/attachments/2067310/3469652/2020-07-01_SoC_interest_group_-_crashkernel_presentation.pdf*, June 2020. Accessed: 15/02/2021.

[109] N. Dzemaili, "Creating a bsp for petalinux." *https://indico.cern.ch/event/952288/contributions/4033881/attachments/2116542/3561511/2020-10-06_Creating_a_BSP_for_PetaLinux.pdf*, October 2020. Accessed: 28/01/2021.

[110] Xilinx, *Zynq UltraScale+ MPSoC Quick Emulator User Guide*, June 2016. UG1169, v2016.2.

[111] Xilinx, "Xilinx software command-line tool (xsct)." *https://www.xilinx.com/html_docs/xilinx2018_1/SDK_Doc/xsct/intro/xsct_introduction.html*, April 2018. Accessed: 15/02/2021.

[112] Xilinx, "U-boot images." *https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842374/U-Boot+Images*, October 2019. Accessed: 26/01/2021.

[113] Xilinx, *PetaLinux Tools Documentation: Reference Guide, Chapter 1: Overview*, December 2018. UG1144, v2018.3.

[114] Xilinx, "Petalinux tools." *https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html*, September 2020. Accessed: 22/09/2020.

[115] Yocto, "Yocto project wiki main page." *https://wiki.yoctoproject.org/wiki/Main_Page*, August 2020. Accessed: 22/09/2020.

[116] OpenEmbedded, "Welcome to openembedded." *http://www.openembedded.org/wiki/Main_Page*, May 2017. Accessed: 23/09/2020.

[117] Yocto-Project, "Software project components: Poky." *https://www.yoctoproject.org/software-item/poky/*, 2020. Accessed: 07/10/2020.

[118] Xilinx, *PetaLinux Tools Documentation: Reference Guide, Appendix B*, December 2018. UG1144, v2018.3.

[119] Gentoo-Foundation-Inc., "Kernel/configuration." *https://wiki.gentoo.org/wiki/Kernel/Configuration*, March 2020. Accessed: 23/09/2020.

[120] R. Purdie, C. Larson, and P. Blundell, "Bitbake user manual, 1.1. introduction." *https://www.yoctoproject.org/docs/1.6/bitbake-user-manual/bitbake-user-manual.html#intro*, 2014. Accessed: 24/09/2020.

[121] R. Purdie, C. Larson, and P. Blundell, "Bitbake user manual, 1.3. concepts." *https://www.yoctoproject.org/docs/1.6/bitbake-user-manual/bitbake-user-manual.html#Concepts*, 2014. Accessed: 24/09/2020.

[122] R. Purdie, C. Larson, and P. Blundell, "Bitbake user manual, 3.5. tasks." *https://www.yoctoproject.org/docs/1.6/bitbake-user-manual/bitbake-user-manual.html#tasks*, 2014. Accessed: 24/09/2020.

[123] Xilinx, *PetaLinux Tools Documentation: Reference Guide, Chapter 3: Creating a Project*, December 2018. UG1144, v2018.3.

[124] Xilinx, "Platform hardware description file." *https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/introduction/concept_sdsocpl_hw_desc_file.html*, 2015. Accessed: 18/01/2021.

[125] Xilinx, *PetaLinux Command Line Reference Guide*, May 2019. UG1157, v2019.1.

[126] Embedded-Linux-Wiki, "Device tree reference." *https://elinux.org/Device_Tree_Reference*, February 2020. Accessed: 18/01/2021.

[127] M. Balakowicz, "U-boot new uimage source file format." *https://github.com/lentinj/u-boot/blob/master/doc/uImage.FIT/source_file_format.txt*, May 2010. Accessed: 18/01/2021.

[128] Xilinx, "Solution zynqmp pl programming." *https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841847/Solution+ZynqMP+PL+Programming*, December 2020. Accessed: 19/01/2021.

[129] Xilinx, "U-boot images." *https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842374/U-Boot+Images*, October 2019. Accessed: 19/01/2021.

[130] M. Fowler, "Continuous integration." *https://martinfowler.com/articles/continuousIntegration.html*, 2006. Accessed: 30/01/2021.

[131] GitLab, "Gitlab ci/cd." *https://docs.gitlab.com/ee/ci/*, 2021. Accessed: 30/01/2021.

[132] P. D. Smith, "Gnu make manual." *https://www.gnu.org/software/make/manual/make.html*, March 2009. Accessed: 02/02/2021.

[133] Xilinx, "Zynq ultrascale+ mpsoc: How to get mac address from eeprom on zcu102 board using petalinux?." *https://www.xilinx.com/support/answers/70176.html*, July 2018. Accessed: 28/01/2021.

[134] J. van Baren, "U-boot config-serverip." *https://lists.denx.de/pipermail/u-boot/2009-March/049568.html*, March 2009. Accessed: 28/01/2021.

[135] G. Kuhlmann, M. Mares, N. Schottelius, Horms, and C. Novakovic, "Mounting the root filesystem via nfs." *https://www.kernel.org/doc/Documentation/filesystems/nfs/nfsroot.txt*, 2018. Accessed: 29/01/2021.

[136] W. Denk, "bootp.c." *https://github.com/Xilinx/u-boot-xlnx/blob/master/README*, 2013. Accessed: 29/01/2021.

[137] Xilinx, *ZCU102 Evaluation Board User Guide*, June 2019. UG1182, v1.6.

[138] kernel org, "Pin controller bindings." *https://www.kernel.org/doc/Documentation/devicetree/bindings/pinctrl/pinctrl-bindings.txt*, 2021. Accessed: 29/01/2021.

[139] G. Likely, "Linux and the device tree." *https://www.kernel.org/doc/html/latest/_sources/devicetree/usage-model.rst.txt*, December 2020. Accessed: 21/01/2021.

[140] Power.org-Inc., "Power.org, inc. standard for embedded power architecture platform requirements." *https://elinux.org/images/c/cf/Power_ePAPR_APPROVED_v1.1.pdf*, April 2011. Accessed: 28/01/2021.

Appendices

A. Zynq MPSoC booting process flowchart

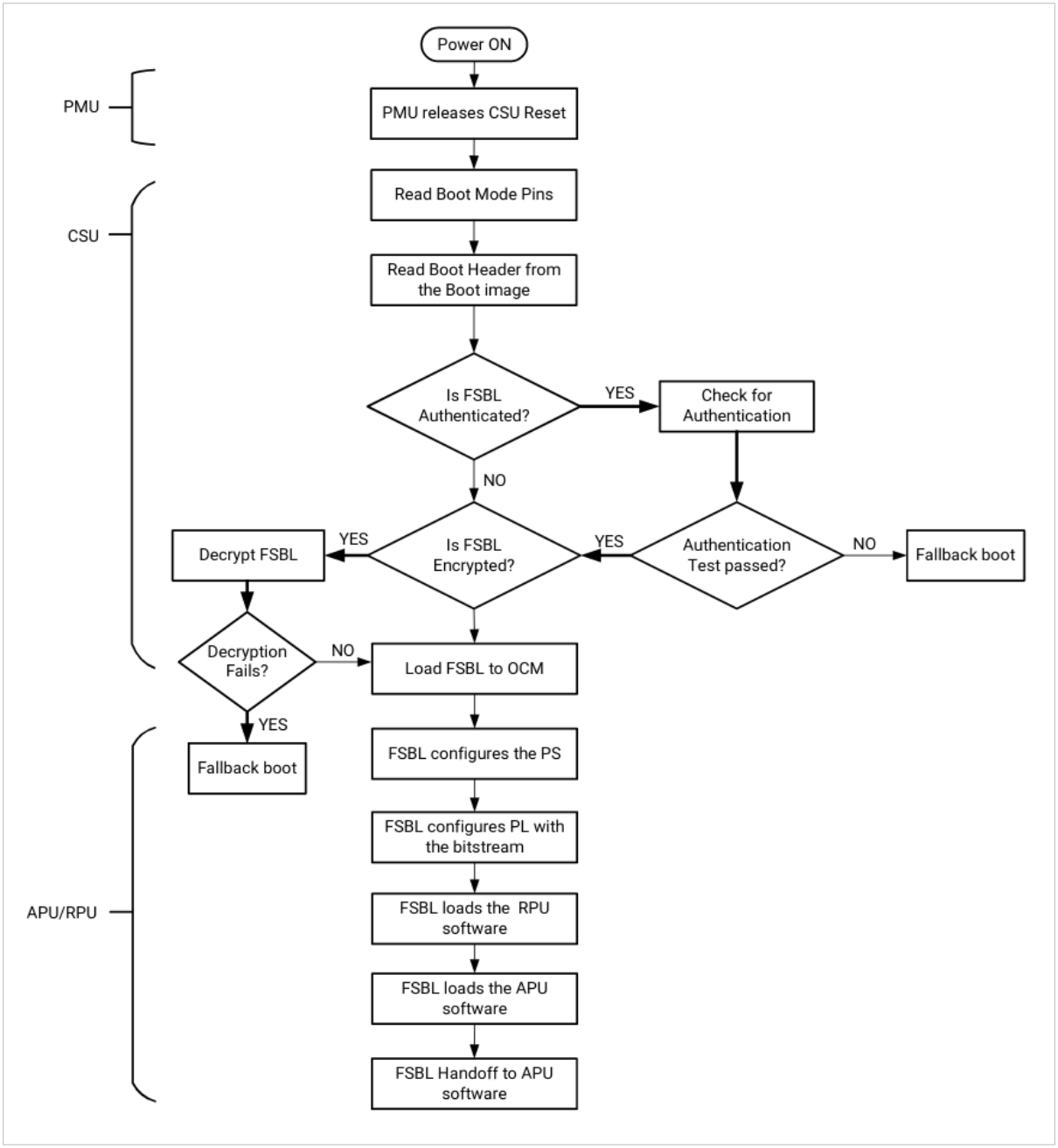


Figure A.1: Zynq Ultrascale+ MPSoC detailed boot flow example [41].

B. CSU BootROM error codes

Table B.1 CSU BootROM error codes that are related to the booting process [76].

Error code	Description
0x23	Error occurred during QSPI 24 boot mode initialization.
0x24	Error occurred during QSPI 32 boot mode initialization.
0x25	Error occurred during NAND boot mode initialization.
0x26	Error occurred during SD boot mode initialization.
0x27	Error occurred during eMMC boot mode initialization.
0x2A	Invalid boot mode is selected in the boot mode setting.
0x30	Boot header does not have an XLNX string.
0x31	Boot header checksum is wrong or boot header fields are not length aligned.
0x32	Boot header encryption status value is not valid. Key selected is not a valid key source.
0x33	Boot header attributes value is not valid. Reserved fields in image attributes are not zero.
0x34	Either of the boot header PMU firmware length and total PMU firmware length fields are not valid.
0x36	Either of the boot header FSBL and total FSBL length fields are not valid.
0x37	FSBL execution address is not in the OCM address range.
0x3B	Reading failed from the selected boot device.
0x47	Boot header signature is failed.
0x49	No image found in QSPI after searching the allowed address range.
0x4A	No image found in NAND after searching the allowed address range.
0x4B	No image found in the SD/eMMC after searching the allowed number of files.
0x60	One of the register addresses in the boot header is not allowed.
0x61	Copying from selected boot device failed after register initialization.
0x62	Boot header read after register initialization is mismatched with the original boot header.
0x70	Error occurred while copying the PMU FW.
0x71	Error occurred while copying the FSBL.
0x78	Boot image signature mismatch occurred.
0x79	Error occurred while decrypting the PMU firmware.
0x7A	Error occurred while decrypting the FSBL.
0x7B	Mismatch in the hash while checking for the boot image integrity.

C. Golden image search mechanism appendices

C.1 FSBL with debug output enabled

```
Xilinx Zynq MP First Stage Boot Loader
Release 2019.2   Jan 17 2021  -  18:42:06
Reset Mode      :       System Reset
Platform: Silicon (4.0), Cluster ID 0x80000000
Running on A53-0 (64-bit) Processor, Device Name: XCZU9EG
FMC VADJ Configuration Successful
Board Configuration successful
Processor Initialization Done
================ In Stage 2 ============
SD1 with level shifter Boot Mode
SD: rc= 0
File name is BOOT.BIN
Multiboot Reg : 0x0
Image Header Table Offset 0x8C0
*****Image Header Table Details********
Boot Gen Ver: 0x1020000
No of Partitions: 0x3
Partition Header Address: 0x440
Partition Present Device: 0x0
Initialization Success
======= In Stage 3, Partition No:1 =======
UnEncrypted data Length: 0x31DE
Data word offset: 0x31DE
Total Data word length: 0x31DE
Destination Load Address: 0xFFFEA000
Execution Address: 0xFFFEA000
Data word offset: 0x105B0
Partition Attributes: 0x117
Partition 1 Load Success
======= In Stage 3, Partition No:2 =======
UnEncrypted data Length: 0x32862
Data word offset: 0x32862
Total Data word length: 0x32862
Destination Load Address: 0x10080000
Execution Address: 0x10080000
Data word offset: 0x13790
Partition Attributes: 0x114
Partition 2 Load Success
All Partitions Loaded
================ In Stage 4 ============
Protection configuration applied
```

Figure C.1: FSBL boot-up messages with debug info enabled.

C.2 FSBL partition validation flowchart

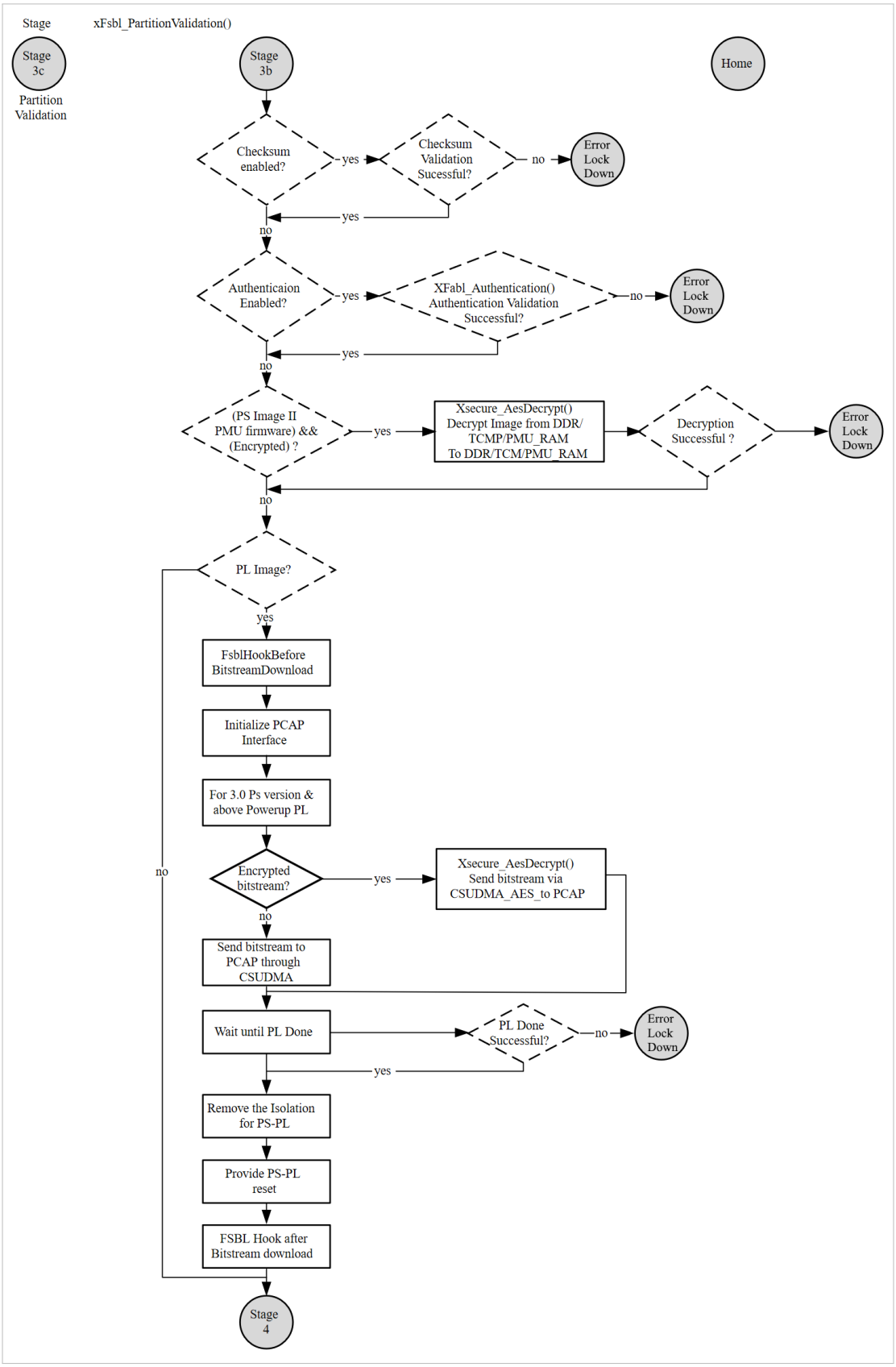


Figure C.2: Flowchart of FSBL partition validation function [45].

D. RELBOOT & RELUP mechanisms

D.1 RELBOOT & RELUP boot option flowchart

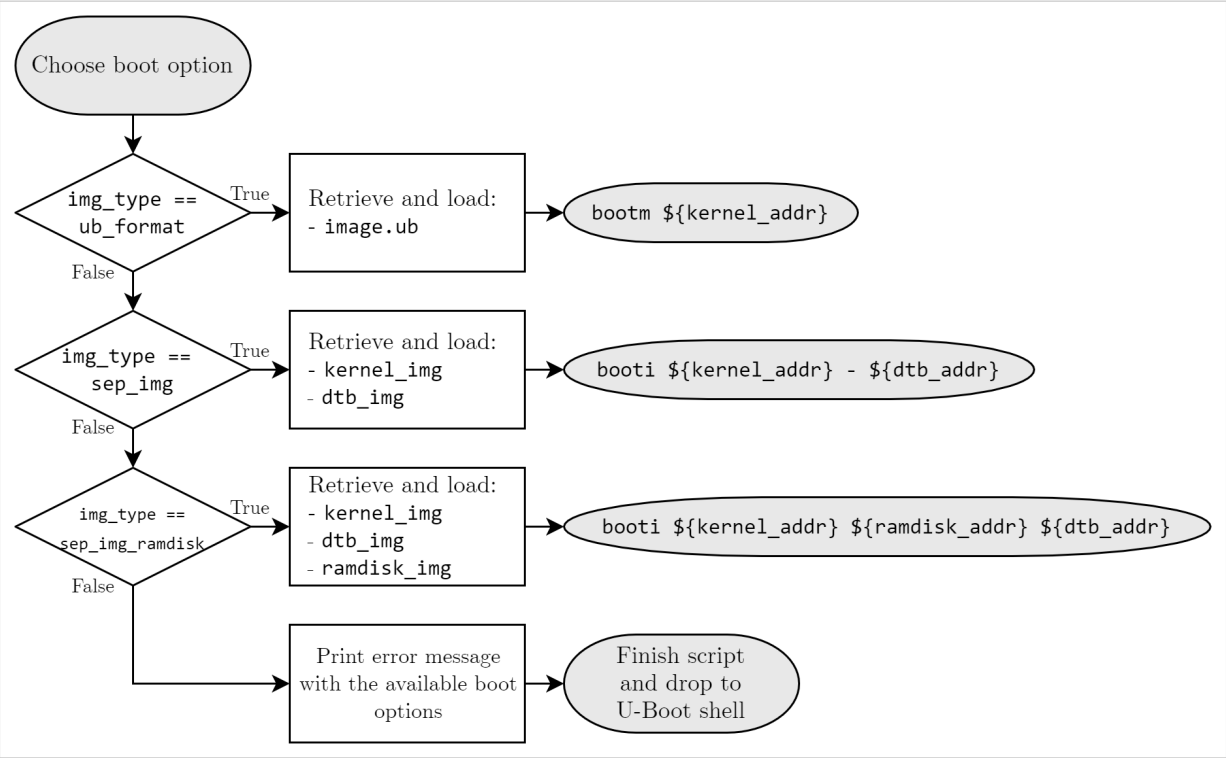


Figure D.1: Flowchart with available boot options for the RELBOOT & RELUP script in U-Boot.

D.2 Custom parser for adding scripts to to the default U-Boot environment

Figure D.2 shows the design of the custom `scriptadder` application:

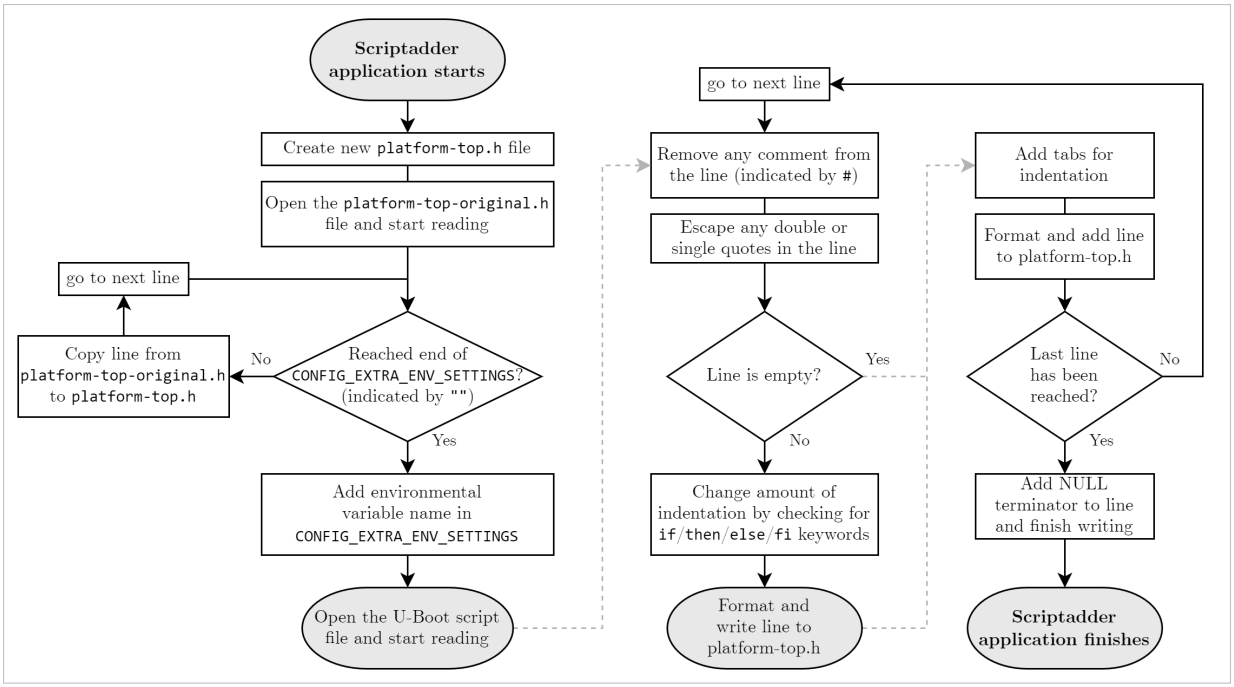


Figure D.2: Diagram of the custom scriptadder application.

For the `scriptadder` application to work, the original `platform-top.h` file needs to be renamed to `platform-top-original.h`. In this *"original"* file, it will search for the `CONFIG_EXTRA_ENV_SETTINGS` definition (which should be defined at the end of the file). Once found, the application will start copying and formatting lines from the U-Boot script file.

`scriptadder` can find control structures (e.g. if-then-else-fi) and add indentations to the lines using tabs. This indentation will be visible in the `platform-top.h` file and in the environmental variable in U-Boot. In addition, the application also adds line feeds to every line. Finally, the last line of the U-Boot script will have a `NULL` terminator added to it, before writing it to `platform-top.h`.

D.3 RELBOOT & RELUP configuration file

Table D.1 gives a description of every configuration option in the `/etc/relup.d/relup.conf` configuration file for the RELBOOT & RELUP mechanisms:

Table D.1 Summary of configuration options for the RELBOOT and RELUP mechanisms.

Option	Default value	Description
`email`	`-`	Email-address of the user.
`boot_attempts`	`3`	Maximum amount of boot attempts for the RELUP boot counter and global boot counter.
`backup_dev`	`mmc`	Backup-boot device. This device is used as the storage for the backup boot images[1].
`backup_mmc_dev`	`0:1`	Block device number and partition of the SD-card. The first partition of the SD-card is used to store the backup boot images (see Appendix G).
`tftp_currentver_fw_dir`	`boot_current`	Name of TFTP server symbolic link that points to the firmware version that the board should *currently* boot with.
`tftp_newver_fw_dir`	`boot_new`	Name of TFTP server symbolic link that points to the firmware version that the board should *upgrade* to.
`img_type`	`sep_img_ramdisk`	Boot option for U-Boot. It can be configured to `ub_format`, `sep_img` or `sep_img_ramdisk`. More information in Subsection 6.2.2 and Appendix D.1.
`version_file`	`version`	Filename of the version file.
`env_file`	`uEnv.txt`	Filename of the U-Boot environment file.
`kernel_img`	`Image`	Filename of the kernel image.
`ramdisk_img`	`ramdisk.img`	Filename of the ramdisk image.
`dtb_img`	`system.dtb`	Filename of the device-tree blob image.

The U-Boot environment file (`uEnv.txt`) can be used to modify or add environmental variables to U-Boot. The file resides on the TFTP server and can allow for some extra configuration. An example is the modification of the addresses in memory where U-Boot loads the boot images (the address variables can be seen in Appendix D.1). These are not configurable from the RELUP configuration file, but can be modified using the `uEnv.txt` file.

When booting the system with a ramdisk (`img_type=sep_img_ramdisk`), it is important to prepare the ramdisk image properly. The ramdisk image should be wrapped with a U-Boot header. This can be done by using the `mkimage` utility that is provided by U-Boot [112].

When the Zynq MPSoC is booted with the SD-card backup images, the kernel will mount the root filesystem on the SD-card. This root filesystem also contains a `relup.conf` file. This configuration file only contains the email-address option. The backup is not able to reconfigure the RELBOOT & RELUP mechanisms.

[1] The `backup_dev` option currently only supports SD-card devices (`mmc`).

D.4 U-Boot environment access from Linux

The U-Boot environment variables can be accessed from Linux through the U-Boot firmware utilities. The firmware utilities were compiled by adding the `u-boot-fw-utils` build flag in the meta-user configuration file of the PetaLinux project (see Figure D.3).

```
$ cat project-spec/meta-user/conf/petalinuxbsp.conf | grep "u-boot"
IMAGE_INSTALL_append += "u-boot-fw-utils"
```

Figure D.3: Build flag that is used to compile the U-Boot firmware utilities.

The `fw_printenv` and `fw_setenv` utilities will be located in the `/sbin` directory of the `rootfs.tar.gz` archive that is created by PetaLinux. The utilities require a configuration file that defines the address and size of the U-Boot environment in QSPI flash. The contents of the configuration file for the ZCU102 are shown in Figure D.4:

```
$ cat /etc/fw_env.config
# NOR flash device    Offset    Env. size    Flash sector size
/dev/mtdblock1        0x0000    0x40000      0x40000
```

Figure D.4: U-Boot firmware utility configuration for the ZCU102 development board.

The `fw_setenv` utility is used to write values from the RELBOOT & RELUP daemon in Linux to the U-Boot environment. The utility has a *script* option which allows the user to write a file with multiple variables to the environment at once. The RELBOOT & RELUP takes advantage of this option by copying the contents of `relup.conf` to a temporary file and writing it to the U-Boot environment.

E. Crashkernel appendices

E.1 Crashkernel memory optimization

The crashkernel was tested with different amounts of memory, starting at 64 MB and increasing in increments of 32 MB. The results of the tests can be seen in Table E.1.

Table E.1 Test results of crashkernel memory reservation optimization.

#	Reserved memory	Test results
1	64 MB	Not able to mount root filesystem
2	96 MB	Not able to mount root filesystem
3	128 MB	Not able to mount root filesystem
4	160 MB	Not able to mount root filesystem
5	176 MB	Fails to start the kdump service
6	192 MB	Saves dump file successfully
7	256 MB	Saves dump file successfully

The crashkernel was successfully able to boot and save a dump with 192 MB of memory reserved. The crashkernel would not be not be able to mount its initramfs with less memory. Testing the crashkernel with 176 MB of reserved memory resulted in the ramdisk being mounted, but the kdump service not starting.

E.2 Kdump configuration

The Zynq MPSoC on the ZCU102 has access to a root filesystem that is mounted via NFS. The system also has access to an SD-card. Dumping to the SD-card and NFS can both be tested. Figure E.1 shows the configuration options that should be added to `/etc/kdump.conf` to use the SD-card or NFS as a dump target.

```
# Dump target is SD-card
ext4 /dev/mmcblk0p2

# Dump target is NFS
nfs 128.141.174.247:/rootfs/zcu102-lab40-r01-33

# Path where dump file will be saved
path /var/crash/

# Enabling the kdump pre- and post-scripts
kdump_post /var/crash/scripts/kdump-post.sh
kdump_pre /var/crash/scripts/kdump-pre.sh
```

Figure E.1: Kdump configuration options for specifying the dump target as the SD-card and as NFS.

Kdump will use the second partition of the SD-card which has an ext4 filesystem on it (see Appendix G for more information on the SD-card). When using NFS as the dump target, the IP-address of the NFS server should be specified. By default kdump will use the `/var/crash/` path to save the dump on the dump target.

E.3 ABRT user notifications configuration

Figure E.2 shows how ABRT was installed and configured to send emails about crashes [73]:

```
$ yum install abrt-cli libreport-plugin-mailx

$ cat /etc/libreport/plugins/mailx.conf
Subject="[abrt] a crash has been detected on Zynq MPSoC"
EmailFrom="ABRT Daemon <DoNotReply>"
EmailTo="nekija.dzemaili@cern.ch"
```

Figure E.2: Installation and configuration of ABRT with email plugin.

The configuration of the crash report email is saved in the `/etc/libreport/plugins/mailx.conf` file. The recipient of the email can be set here.

After installing and configuring ABRT, the Zynq MPSoC is able to send emails of the crashkernel to the user. The email contains the reason of the crash and other information about the event. It also contains a backtrace and the kernel console messages for debugging.

E.4 Crashkernel console output

```
$ echo c > /proc/sysrq-trigger
[67446.591965] sysrq: SysRq : Trigger a crash
[67446.596079] Unable to handle kernel NULL pointer dereference
at virtual address 0000000000000000
...
... crash dump with call trace ...
...
[67446.818555] Starting crashdump kernel...
[67446.822462] Bye!

[    0.000000] Booting Linux on physical CPU 0x0000000001 [0x410fd034]
[    0.000000] Linux version 4.19.0-xilinx-v2019.2 (oe-user@oe-host)
(gcc version 8.2.0 (GCC)) #1 SMP Fri Dec 11 13:22:25 UTC 2020
...
[   11.787058] Run /init as init process
[   11.837448] systemd[1]: Detected architecture arm64.
[   11.842488] systemd[1]: Running in initial RAM disk.

Welcome to CentOS Linux 8 (Core) dracut-049-27.git20190906.el8_1.1 (Initramfs)!
...
[   20.348302] systemd[1]: Starting Kdump Vmcore Save Service...
kdump: dump target /dev/disk/by-uuid/83b9a606-b4ac-40a4-96a9-a3a514a1fd8d
is not mounted, trying to mount...
[   20.872652] EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode.
[   20.928377] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)
...
kdump: saving to /kdumproot//SDCARD//var/crash//127.0.0.1-2021-01-22-11:08:22/
kdump: saving vmcore-dmesg.txt
No program header covering vaddr 0xffffff800915afe0found kexec bug?
kdump: saving vmcore-dmesg.txt failed
kdump: saving vmcore
Copying data                                       : [100.0 %] /       eta: 0s
kdump: saving vmcore complete
...
[KDUMP-POST]: kdump-post.sh started
...
[   32.228212] systemd[1]: Shutting down.
...
[   32.867433] reboot: Restarting system
```

Figure E.3: Crash of the main system kernel and booting of the crashkernel. Kdump vmcore saving service saves a dump on the SD-card and later reboots the system.

F. Watchdog timer appendices

F.1 Watchdog timer healthy bit scheme

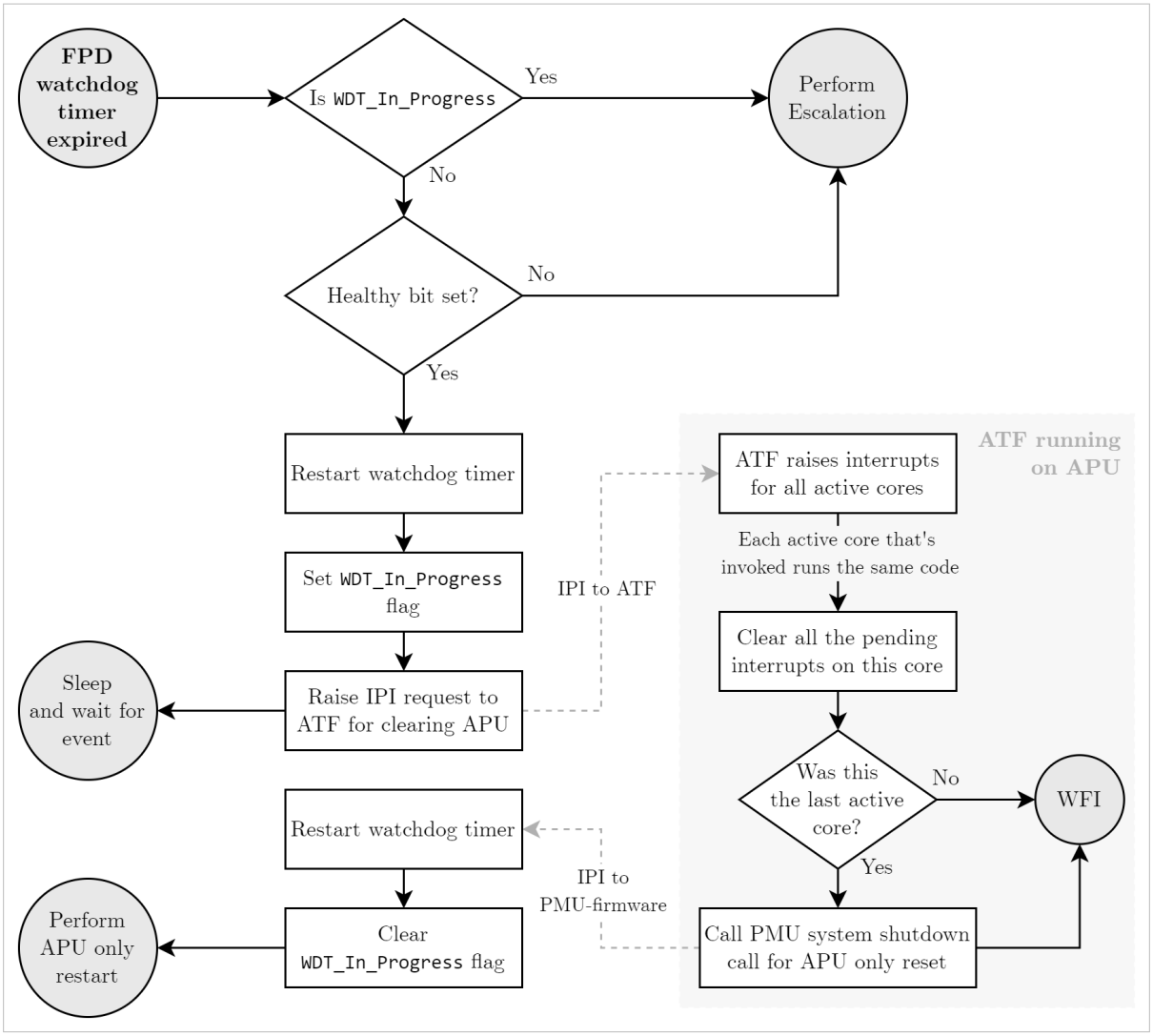


Figure F.1: Diagram of watchdog timer expiry handling with escalation and the healthy bit scheme enabled [62].

F.2 Watchdog heartbeat daemon source code

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

#define WDT_BASE                0xFD4D0000
#define WDT_RESET_OFFSET        0x8
#define WDT_RESET_KEY           0x1999
#define REG_WRITE(addr, off, val) (*(volatile unsigned int*)(addr+off)=(val))
#define REG_READ(addr,off) (*(volatile unsigned int*)(addr+off))

void wdt_heartbeat(void)
{
  char *virt_addr;
  int fd;
  int map_len = getpagesize();

  fd = open("/dev/mem", (O_RDWR | O_SYNC));

  virt_addr = mmap(NULL, map_len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, WDT_BASE);

  if (virt_addr == MAP_FAILED)
    perror("mmap failed");

  close(fd);

  REG_WRITE(virt_addr,WDT_RESET_OFFSET, WDT_RESET_KEY);

  munmap((void *)virt_addr, map_len);
}

int main()
{
  while(1)
  {
    wdt_heartbeat();
    sleep(2);
  }
  return 0;
}
```

Figure F.2: Source code of C application for resetting the watchdog timer of the Zynq MPSoC (heartbeat application) [62].

G. SD-card setup for Zynq MPSoC

One of the requirements of the project is that the Zynq MPSoC should boot through the network. Therefore it uses an NFS as its root filesystem. The Zynq MPSoC requires a disk that can store the boot image (`BOOT.BIN`) for booting the bootloaders though. After that it can retrieve the required images for booting the Linux OS from the TFTP server.

This chapter shows how the SD-card was setup for booting. The SD-card is also used for storing a local root filesystem. The SD-card will have the following two partitions (Table G.1):

Table G.1 SD-card partitions for Zynq MPSoC ZCU102.

#	Partition	Size
1	BOOT	1 GB
2	ROOTFS	The rest of the free space on the SD-card

G.1 Creating the BOOT partition

The partitions on the SD-card will be created using the *fdisk* utility. The lsblk command can be used to find the name of your SD-card disk (in this example 'sdb' is used). No other partitions can be present on the SD-card. They can be deleted using the 'd' (delete) command in fdisk.

The BOOT partition will store the boot image. It has a size of 1 Gb. The BOOT partition *needs* to be a FAT32 type partition. Else the booting will not work. The partition type can be changed by using the 't' (type) command (Figure G.1):

```
$ lsblk
$ sudo fdisk /dev/sdb

Welcome to fdisk (util-linux 2.32.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Command (m for help): n
Partition type
p   primary (0 primary, 0 extended, 4 free)
e   extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-31116287, default 2048): 2048
Last sector, +sectors or +size{K,M,G,T,P} (2048-31116287, default 31116287): 2097152

Created a new partition 2 of type 'Linux' and of size 1023 MiB.

Command (m for help): t
Selected partition 1
Hex code (type L to list all codes): b
Changed type of partition 'Linux' to 'W95 FAT32'.
```

Figure G.1: Example of creating the BOOT partition on the SD-card.

G.2 Creating the ROOTFS partition

The ROOTFS partition will be created as a second primary partition. The remaining space on the SD-card is used for this partition to have as much space as possible for potential files. Partitions on the SD-card can be checked by using the 'p' command. The 'w' command saves the changes and quits fdisk (Figure G.2):

```
Command (m for help): n
Partition type
p   primary (1 primary, 0 extended, 3 free)
e   extended (container for logical partitions)
Select (default p): p
Partition number (2-4, default 2): 2
First sector (2097153-31116287, default 2099200): 2099200
Last sector, +sectors or +size{K,M,G,T,P} (2099200-31116287, default 31116287): 31116287

Created a new partition 2 of type 'Linux' and of size 13.9 GiB.

Command (m for help): p
Disk /dev/sdb: 14.9 GiB, 15931539456 bytes, 31116288 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x00035bde

Device     Boot   Start      End  Sectors  Size Id Type
/dev/sdb1          2048  2097152  2095105 1023M  b W95 FAT32
/dev/sdb2       2099200 31116287 29017088 13.9G 83 Linux

Command (m for help): w
The partition table has been altered.
```

Figure G.2: Example of creating the ROOTFS partition on the SD-card.

Running lsblk again will show 2 partitions on the SD-card.

G.3 Mounting filesystems on the partitions

The filesystems can now be added to the partitions. They need to be unmounted to add a filesystem. The first partition should have a FAT32 filesystem. The second partition can have an ext4 filesystem (Figure G.3):

```
$ lsblk
$ sudo umount /dev/sdb1
$ sudo umount /dev/sdb2
$ sudo mkfs.vfat -n BOOT /dev/sdb1
$ sudo mkfs.ext4 -L ROOTFS /dev/sdb2
```

Figure G.3: Example of adding a FAT and ext4 filesystems on the partitions of the SD-card.

The SD-card is now setup and ready to be used.

H. Creating a board support package (BSP)

This chapter goes over the PetaLinux tools and the creation of a Board Support Package (BSP). The PetaLinux tools have been used during the project to configure, modify and build the bootable images for the Zynq Ultrascale+ MPSoC. They've also been used for building some of the fallback solutions for the reliable booting system.

A BSP is a template that defines how to support a particular hardware platform [86]. It allows one to define all the features for their board and package it into a reusable format. The BSP will be used to bundle all the modifications and configurations of the PetaLinux project.

H.1 What is PetaLinux?

PetaLinux is a set of development tools for embedded Linux systems, that is specifically made for FPGA-based SoCs (System on Chips) from Xilinx [113]. The tools allow for configuration and customization of the Zynq MPSoC low-level firmware, the bootloader(s), kernel, device-tree, filesystem and libraries. PetaLinux also consist of build and deployment tools [114]. The PetaLinux tools are based on the Yocto Project, which is an open-source collaboration project for creating embedded Linux distributions. It forms the base of PetaLinux.

H.1.1 Yocto layers and recipes

Yocto offers a set of tools for creating embedded Linux distributions. These tools are based on the OE-core of the OpenEmbedded project, which is a framework for embedded Linux [115,116]. Yocto combined the OE-core with a build tool and metadata[1] to create a *reference distribution*, called Poky [117]. Users can take Poky and add changes on top of it to create their own embedded Linux distributions. This is done through *layers*.

Yocto layers

A PetaLinux project consists of layers. Layers offer customization without having to edit the originally provided files. There are multiple layers available in PetaLinux. Their hierarchy is shown in Figure H.1.

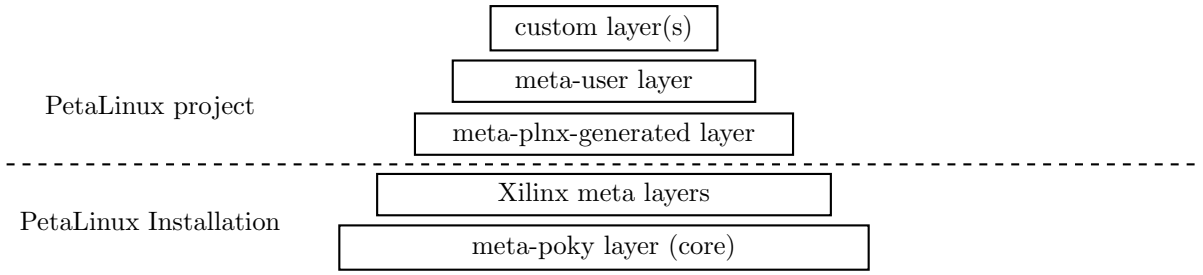


Figure H.1: Yocto metadata layer hierarchy in a PetaLinux project.

The base layer consists of metadata for Poky. There are several layers provided by Xilinx on top of that. These layers add support for Xilinx architectures, including MicroBlaze, Zynq, and Zynq MPSoC. They also add support for Xilinx tools that may be used when developing and using your board. The meta-poky and Xilinx meta layers are part of the PetaLinux installation. These layers cannot be modified.

The top three layers, shown in Figure H.1, are user-customizable. The meta-plnx-generated and meta-user layers are automatically generated when you create a PetaLinux project. On top of that, the user can manually add custom layers [118].

[1] In Yocto, metadata is a collection of files that describe the build process for a package. This is used by the build tool.

PetaLinux provides customization through configuration menus (menuconfig[2]). When changing something in these menus, the meta-plnx-generated layer gets modified. PetaLinux offers configuration menus for general configuration, the kernel, and U-Boot among others. The meta-user layer offers manual configuration by editing or adding configuration files and metadata. Any customizations made through a configuration menu can be overwritten using this layer. The advantage of the meta-user layer is that it allows you to modify configuration options that are not available in the menus.

Layers contain metadata, mainly *recipes*. This metadata is used by the build tool to create images and binaries. The build tool used by PetaLinux is BitBake.

BitBake and recipes
BitBake is a powerful and flexible build engine, based on Python. It executes builds by running tasks. A task is a set of commands that comprises a part of the building process. This can be: fetching source code, unpacking, patching, compiling, packaging, etc. The tasks are described by several files, the most common of which are the recipe, configuration and class files. These files together are called *metadata*.

The most common form of metadata is *the recipe* (denoted by the file extension `.bb`) [120]. A recipe is a file that provides a *"list of ingredients"* and *"cooking instructions"*. Recipes tell BitBake about dependencies, where to find source code, whether to apply any patches, how to compile the source code etc. Tasks are also defined in the recipe. A task can be defined as a shell or python function [121, 122].

To modify an existing recipe from another layer, one can use a BitBake append file (denoted by the file extension `.bbappend`). The meta-user and meta-plnx-generated layers, in a PetaLinux project, mainly use these files in their recipes to modify the underlying layers.

Figure H.2 shows the recipes in the meta-user layer of a PetaLinux project. There are three types of recipes in this layer. Application recipes build an application that will run on the OS. The kernel recipe is used to modify the kernel configuration. The board-specific recipes are used to modify the configurations of components that are specific to hardware, e.g. the device-tree. The kernel and board-specific recipes use the aforementioned BitBake append file. Custom recipes can also be added to the meta-user layer.

```
CMS-ZCU102-v1
├─ build
├─ components
├─ images
└─ project-spec
   ├─ configs
   ├─ hw-description
   ├─ meta-plnx-generated
   └─ meta-user
      ├─ conf
      ├─ recipes-apps
      ├─ recipes-board-specific
      │  ├─ arm-trusted-firmware
      │  ├─ device-tree
      │  ├─ fsbl
      │  ├─ pmu-firmware
      │  └─ u-boot
      └─ recipes-kernel
```

Figure H.2: Directory tree of a Peta-Linux project, showing the recipes in the default meta-user layer.

H.1.2 PetaLinux project structure

A PetaLinux project is structured with four main directories (see Figure H.2):

1. The *project-spec* directory holds the project configuration files, the hardware description from the hardware development tool (Vivado), and the user-customizable layers. Every customization in the project is stored in this directory.
2. The *images* directory stores the output images that are created after the building process.
3. The *build* directory stores all files that are generated during the building process.
4. The *components* directory is used to add external source code that may be used during the building process. E.g. one can add additional kernel source code.

[2]Menuconfig is a menu-driven user interface that is normally used for Linux kernel configuration [119]. It is also used for other applications. E.g. in PetaLinux configuration or U-Boot configuration

H.1.3 PetaLinux summary

Yocto in its core is only meant for creating the Linux distribution that will run on the embedded system. As mentioned in Section 3.2, the Zynq MPSoC requires some other components on top of that to boot Linux. PetaLinux adds these components through the Xilinx meta layers (see Figure H.1).

The Xilinx meta layers can still be used in Yocto without PetaLinux. However, PetaLinux offers some benefits over Yocto for users that do not have much experience with embedded Linux development.

Creating a new project in Yocto requires all layers and recipes to be sourced and configured manually. PetaLinux adds commands and configuration menus that simplify this. It is also the only embedded Linux development tool that officially supports the Xilinx MPSoC architecture. The downside to PetaLinux is the limited amount of configurability without moving to manual configuration through the Yocto layers.

H.2 Porting to different hardware using a BSP

One of the requirements for the project is that the reliable booting system can be ported to different hardware (see Section 2.3). The PetaLinux tools are used during the project to configure, modify and build the bootable images for the Zynq Ultrascale+ MPSoC. PetaLinux offers the ability to create *board support packages*, which can be used to port the reliable booting system to different hardware.

In PetaLinux, a Board Support Package (BSP) is a template which is used to create new PetaLinux projects for a hardware specific platform. The BSP is used as an overlay by PetaLinux and contains a pre-configured project, pre-built images, and a copy of the related Vivado hardware project. To create a BSP, one needs to make a bare-bones PetaLinux project and modify it.

Xilinx supplies BSPs for all its development boards. The ZCU102, which is preconditioned to be used during the project, also has a BSP which can be downloaded from the Xilinx website. The ZCU102 BSP is tied to the hardware that is present on the ZCU102 board.

To satisfy the requirement of having a reliable booting system that can be ported to different hardware, it was decided to create a custom BSP that is stripped down from any hardware specific components. This allows developers to use the BSP to create a PetaLinux project for any board. Once the project is created using the BSP, the developers can add their hardware specific components manually: e.g. in the device-tree. The custom BSP will contain the various fallbacks of the reliable booting system so developers do not have to add them manually to their project. Any hardware requirements that the reliable booting system *does* have, can be added manually.

Section H.3 shows how a board support package can be created in PetaLinux. Section H.4 shows how the BSP for the reliable booting system was implemented. The BSP can be built automatically by using a CI. The implementation of the CI is given in Section H.5.

H.3 PetaLinux project creation and BSP packaging

A PetaLinux project can be created by using the *zynqMP* template [123]. This will create a bare-bones project. Projects created with this template, need to be configured using a *hardware description file*. An HDF describes the hardware system of the Zynq MPSoC [124]. Hardware description files can be exported from a Vivado hardware project. The commands for creating and configuring a project can be seen in Figure H.3:

The `petalinux-config` command will show a configuration menu using menuconfig. The menu contains general configuration options for each boot component, hardware settings, and Yocto settings. For more specific configuration of a component, one can use the `-c` (or `--component`) option [125]. Examples of

```
# Creating and configuring a project
$ petalinux-create -t project -n <project-name> --template zynqMP
$ cd <project-name>
$ petalinux-config --get-hw-description=<vivado-project-path/>

# Configuring specific project components
$ petalinux-config -c <component>
```

Figure H.3: Creating and configuring a bare-bones PetaLinux project for BSP creation.

components that can be configured individually are the kernel, U-Boot, and the root filesystem. Manual modification of the meta-user layer is also possible. Once the user is satisfied with his or her modifications, they can build and package the project using the commands described in Figure H.4:

```
# Build and package bootable images into BOOT.BIN
$ petalinux-build
$ cd images/linux/
$ petalinux-package --boot --format BIN --fsbl zynqmp_fsbl.elf
 --u-boot u-boot.elf --pmufw pmufw.elf --fpga *.bit --force

# package pre-built images and create BSP
$ petalinux-package --prebuilt
$ petalinux-package --bsp -p <petalinux-project-path>
--hwsource=<vivado-project-path> --output <bsp-name>.BSP --force
```

Figure H.4: Building the PetaLinux project and packaging the BSP.

The `petalinux-package` command has multiple uses. When the `--boot` option is specified, the `BOOT.BIN` image is created [125]. This image contains the binaries for the PMU firmware, FSBL, ATF and U-Boot. In addition, it can also contain a bit file for the FPGA.

Using the `--pre-built` option it will generate a directory with images that can be used directly after creating a new project using the BSP [125]. Finally, the BSP can be packaged using the `--bsp` option. When packaging the BSP, one can add a copy of the Vivado hardware project by using the `--hwsource` option. This is optional and can be left out to make the filesize of the BSP smaller.

To test the newly created BSP, one can create a new project and build it (Figure H.5):

```
$ petalinux-create -t project -n <project-name> -s <path-to-BSP>
$ cd <project-name>
$ petalinux-config
$ petalinux-build
$ petalinux-package --boot --format BIN --fsbl zynqmp_fsbl.elf
 --u-boot u-boot.elf --pmufw pmufw.elf --fpga *.bit --force
```

Figure H.5: Creating and building a PetaLinux project using the BSP.

Once the PetaLinux project is finished with the building process, a set of bootable images will be available. Each image is explained in Table H.1 with relation to the Zynq MPSoC booting process:

Table H.1 Summary of bootable images generated by PetaLinux (The .elf images are also available as .bin files).

Image	Description
`pmufw.elf`	The PMU firmware runs on the PMU and gets loaded after executing the PMU BootROM (more info in Subsection 3.1.5).

Table H.1 Summary of bootable images generated by PetaLinux (The .elf images are also available as .bin files).

Image	Description
`zynqmp_fsbl.elf`	The FSBL is the first bootloader to run on the APU. It takes care of hardware initialization, loading of the ARM Trusted Firmware (ATF) and loading of U-Boot, the second-stage bootloader (more info in Subsection 3.2.3).
`bl31.elf`	This is the ARM Trusted Firmware. The ATF is used to handle transitions between the secure and non-secure worlds (more info in Subsection3.2.4).
`u-boot.elf`	U-Boot is the second-stage bootloader. Its purpose is to boot the Linux operating system (More info in Subsection 3.2.5).
`system.bit`	This is the FPGA bit file. It contains the information that is necessary for the FPGA to configure the programmable logic in the way it was designed.
`BOOT.BIN`	The `BOOT.BIN` image is a collection of multiple images. It contains a boot header, partition header, and image partitions. The boot header contains various characteristic, attributes and other details about the boot image [25]. The FSBL is the only mandatory image partition for a `BOOT.BIN` image.
`system.dtb`	This is the device-tree blob, the compiled version of the device-tree. The purpose of a device-tree is to provide the kernel with information about, and describe, non-discoverable hardware [126]. This image contains hardware description of the Zynq MPSoC internals and hardware components around the chip. The device-tree is also used by U-Boot.
`Image`	This is the generic Linux kernel binary image. It can be used to boot the Linux OS when used together with the DTB.
`vmlinux`	This is the uncompressed version of the Linux kernel binary image. This file has the `.elf` format though it does not include this in the filename. If enabled in the kernel options, this file will contain debug symbols. The `vmlinux` image is usually used for debugging purposes.
`image.ub`	This is a Flattened Image Tree (FIT) image that can be used by U-Boot. It combines the kernel image and DTB into one image [127]. In addition, it can also contain a ramdisk image (see Subsection 3.2.5). An `image.ub` file is generated by using the `mkimage` utility that is provided in the git repository of U-Boot.

The requirements of the Zynq MPSoC project specify that the FPGA `system.bit` file will be excluded from the `BOOT.BIN`. The FPGA will be programmed from within Linux once the Zynq MPSoC has fully booted up. This allows the programmable logic in the FPGA to be changed dynamically. Programming the FPGA from Linux is possible by using the Xilinx FPGA manager framework [128]. The `BOOT.BIN` will only include the images for the PMU firmware, FSBL, ATF, and U-Boot.

Xilinx is recommending the use of the FIT image (`image.ub`) to boot Linux on the Zynq MPSoC [129]. The FIT image has the advantage of providing security and integrity features. The project requires the use of separate images for the kernel and the device-tree (`Image` and `system.dtb`). This gives the developers the control to change one of the images without having to touch the other. This is especially useful when one of the images is getting changed multiple times per day. The security features of the FIT image are also of less interest, because the Zynq MPSoC will be running in the CMS network which is declared as secure.

H.4 PetaLinux project modifications for Zynq MPSoC reliable booting BSP

Table H.2 shows all manual modifications of the meta-user layer in PetaLinux to create a BSP. The meta-user layer can be found in under the `/project-spec/meta-user/` path in a PetaLinux project.

Table H.2 Summary of PetaLinux file modifications for the creation of the Zynq MPSoC reliable booting BSP

File	Description
`/recipes-bsp/arm-trusted-firmware/arm-trusted-firmware_%.bbappend`	Addition of the `ZYNQ_WARM_RESTART=1` flag which is required for the watchdog timer escalation scheme in the PMU firmware.
`/recipes-bsp/device-tree/files/ system-user.dtsi`	This file contains definitions for the watchdog timer, SD-card, GEM, I^2C-bus, EEPROM, and pin controllers. It also contains the boot arguments for the Linux kernel. The `system-user.dtsi` file includes a set of device-tree bindings which are stored in the same directory (`input.h`, `gpio.h`, `pinctrl-zynqmp.h`, `phy.h`).
`/recipes-bsp/device-tree/device-tree.bbappend`	Device-tree recipe file that includes `system-user.dtsi` and device-tree bindings.
`/recipes-bsp/fsbl/fsbl_%.bbappend`	Addition of the `FSBL_DEBUG_INFO` build flag for enabling debug info in the FSBL.
`/recipes-bsp/pmu-firmware/pmu-firmware_%.bbappend`	Addition of the PMU firmware build flags to add watchdog timer handling.
`/recipes-bsp/u-boot/files/eeprom.cfg`	Configuration options for enabling MAC-address retrieval from the ZCU102 EEPROM.
`/recipes-bsp/u-boot/files/platform-top-original.h`	Original `platform-top.h` file with TFTP-boot configuration options and definition of environmental variables for the RELBOOT and RELUP mechanisms.
`/recipes-bsp/u-boot/files/platform-top.h`	Same as the `platform-top-original.h` file, but with the addition of the U-Boot script for RELBOOT and RELUP.
`/recipes-bsp/u-boot/files/scriptadder.sh`	`scriptadder` application that can add U-Boot scripts to the default environment in the U-Boot binary.
`/recipes-bsp/u-boot/files/cms-relboot.ubootsh`	U-Boot script for the RELBOOT & RELUP mechanism.
`/recipes-bsp/u-boot/u-boot-xlnx_%.bbappend`	U-Boot recipe file that includes `platform-top.h` and `eeprom.cfg`.
`/conf/petalinuxbsp.conf`	Meta-user layer configuration file. The build flag for compiling the U-boot firmware utilities was added to this file.
`/recipes-bsp/versioner/versioner.bb`	Custom BitBake recipe for creating version files when building the PetaLinux project.

Changes that were made to the kernel configuration through `menuconfig`, can be found in the meta-

user layer under `/recipes-kernel/linux/linux-xlnx/devtool-fragment.cfg`. Other changes using the PetaLinux configuration menus can be found under the meta-plnx-generated layer.

H.5 Automated BSP building using Continuous Integration (CI)

The custom BSP can be built automatically by using Continuous Integration (CI). Continuous Integration is a development practice that adds a pipeline of build and/or test scripts to a Git repository [130, 131]. The scripts will automatically build and/or test any code that was pushed to the repository. This is called a job. The CI is able to report to the developer if a job was executed successfully. It can do this through email.

CERN hosts their own GitLab servers, which have a built-in solution for CI. The usage of the of the CI will require the following:

- **A GitLab Runner:** Gitlab CI uses the *GitLab Runner* to run jobs in a pipeline. The Runner can be installed on a PC and is connected to the GitLab repository of the project. The runner is able to run shell scripts. Any dependencies that the scripts might have need to be installed on the machine that hosts the GitLab Runner (the machine will need to have PetaLinux installed).
- **A pipeline:** A CI pipeline is comprised of stages. The stages define the scripts that should be run by the GitLab Runner. There could be a configuration stage, a building stage and a test stage. The stages are defined in a `.gitlab-ci.yml` file, which is placed in the Git repository.
- **A Makefile:** It is common practice to keep a `.gitlab-ci.yml` file clean and readable. The CI can make use of a *Makefile* to split and categorize the building process of the BSP PetaLinux project. The Makefile is used by the `make` utility[3].

The BSP building CI consists of four stages. The first stage takes the PetaLinux project (which is stored in the GitLab repository) and performs a *silent configuration*. This configures the PetaLinux project with the configuration options in the board support package, without opening a `menuconfig`.

The second stage builds every component of the BSP. The components are built separately using jobs. This allows the user to spot which of the components failed to build if the building stage fails. Figure H.6 shows a part of the `.gitlab-ci.yml` that builds U-Boot:

```
1 u_boot:
2   stage: build
3   dependencies:
4   - silentconfig
5   script:
6   - make u_boot
```

Figure H.6: CI job for building U-Boot.

The U-Boot job is part of the `build` stage. It is dependent on the `silenconfig` job. If the silent configuration fails, the U-Boot building job will not be started. The job runs the `make` command with the U-Boot *target* in the Makefile[4].

The third stage packages the boot image for the Zynq MPSoC and the BSP (see Appendix H.2). Finally, the fourth stage of the CI cleans the cache of the GitLab runner. This prevents the CI pipeline from

[3]The `make` utility is used to automate the compilation process of a project/program that is split up into multiple pieces. The utility uses a Makefile that describes which shell commands to use for compilation [132].

[4]The PetaLinux Tools use the command-line for building boot components and packaging a BSP. These commands have been put into a Makefile. Targets in a Makefile contain commands for building the target [132].

using cached files during a new run. Deleting the cached files allows for a clean build of the BSP. The cache cleanup stage can be disabled to save time when rebuilding the BSP with the CI.

Each stage in the CI is dependent on the previous stage. That means that none of the BSP components can be built if the silentconfig stage fails. Furthermore, the BSP is not packaged if one of the jobs in the build stage fail.

The created boot images and BSP are preserved in the GitLab repository for one week. After one week, the images and BSP are deleted. The CI needs to be rerun to recreate the images and BSP. The implementation of the continuous integration for building the board support package can be seen in Figure H.7:

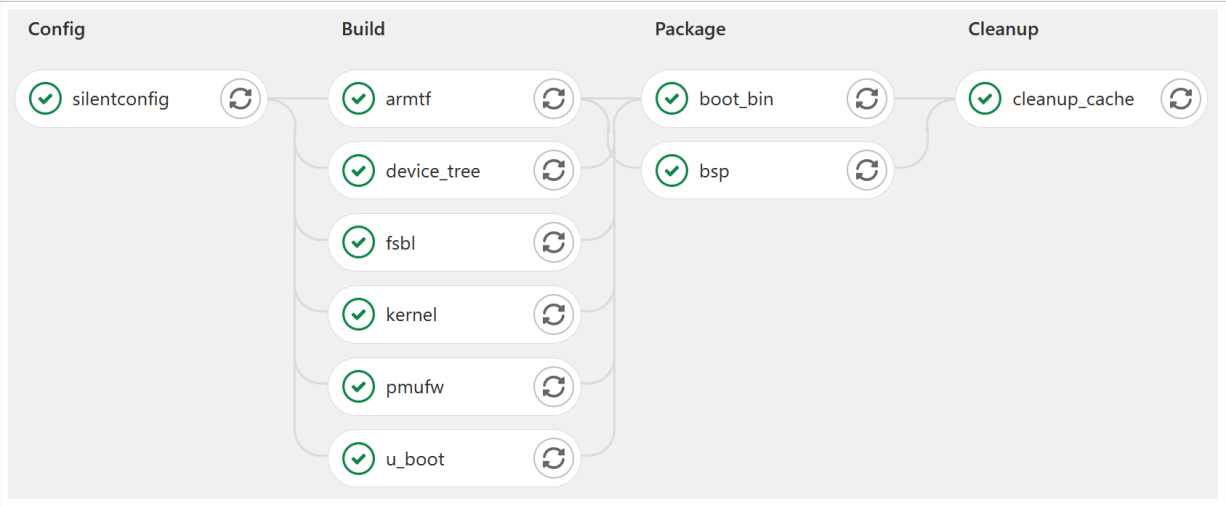


Figure H.7: Implementation of BSP building CI in GitLab.

GitLab CI provides a live terminal that shows the progress of each job. The commands that are run by GitLab runner are shown in these terminals. Figure H.8 shows an example of the debug output for the GitLab runner when building U-Boot:

```
Running with gitlab-runner 12.10.1 (ce065b93) on test CI _5DxszmT
Preparing the "shell" executor                                    00:00
Preparing environment                                             00:00
...
Running before_script and script                                  00:34
...
Building U-Boot for Zynq MPSoC Reliable Booting @ CMS DAQ
$ petalinux-build -c u-boot
[INFO] building u-boot
[INFO] sourcing bitbake
[INFO] generating user layers
...
INFO: Copying Images from deploy to images
INFO: Creating /home/gitlab-runner/builds/_5DxszmT/0/ndzemail/cms-zcu102-bsp/
images/linux directory
NOTE: copy to TFTP-boot directory is not enabled !!
[INFO] successfully built u-boot
...
Saving cache                                                      05:26
Uploading artifacts for successful job                            00:01
Job succeeded
```

Figure H.8: CI job for building U-Boot.

I. Zynq MPSoC network boot

One of the requirements for this project states that the Zynq MPSoC should boot through the network by default. The network boot consists of two main parts. Retrieval of the required boot images from a TFTP server (kernel image, DTB and potential ramdisk) and the use of a remotely stored root filesystem that can be accessed through NFS. The boot images are retrieved by U-Boot. The kernel later uses NFS during booting to mount the remote root filesystem.

I.1 Network-boot research

I.1.1 MAC-address retrieval for network communication

The Zynq MPSoC requires an IP-address (and possibly other network information) to communicate on the network. The IP-address is acquired through a DHCP request using the `dhcp` command. The request requires the Zynq MPSoC to have a valid MAC-address. This MAC-address is registered in the CMS network. It is also stored on the board that hosts the Zynq MPSoC. Xilinx states that the MAC-address on the ZCU102 development board is stored on an I^2C EEPROM [133]. The MAC-address should be retrieved from the EEPROM by U-Boot.

The default U-Boot configuration in PetaLinux configures U-Boot to use a random MAC-address while booting. The U-Boot configuration will need to be changed to use the MAC-address that is stored in the EEPROM. The board support package of the ZCU102, which is provided by Xilinx, includes the configuration for retrieving the MAC-address from the EEPROM in U-Boot.

I.1.2 U-Boot image retrieval through TFTP

U-Boot can retrieve files from a TFTP server by using the `tftpboot` command [55]. This requires the IP-address of the TFTP server to be known. The IP-address of the TFTP server can be acquired through the DHCP request. U-Boot will store the IP-address of the TFTP server in the `serverip` environmental variable. The `serverip` variable has a default definition in PetaLinux. The default definition causes U-Boot to return an error when using the `tftpboot` command:

`*** ERROR: ‘serverip’ not set` [134]

The default definition of the TFTP server IP-address does not allow the DHCP request to overwrite `serverip`. After a DHCP request, the `serverip` variable in U-Boot is not changed to the IP-address of the TFTP server that runs in the CMS network. The `serverip` variable will need to be undefined before doing a DHCP request.

I.1.3 NFS root filesystem

Mounting the root filesystem of the Zynq MPSoC via NFS allows the system to run without a local disk. The kernel can be informed to mount the root filesystem via NFS through the `root=/dev/nfs` boot argument [135]. This boot argument tells the kernel to use NFS as its root filesystem instead of using a *real* device (NFS is a pseudo-device. It is not a physical hardware device connected to the Zynq MPSoC) [135].

The kernel also needs the IP-address of the NFS server, the path to the root directory on the server, and an IP-address configuration. The DHCP server in the CMS-network has been configured to provide this information when performing a DHCP request. The kernel will perform a DHCP request when booting if the `ip=dhcp` boot argument is used [135].

I.2 Network-boot implementation

I.2.1 TFTP boot configuration in U-Boot

To allow U-Boot to save the IP-address of the TFTP server, the `serverip` variable cannot have any value assigned to it (see Subsection I.1.2). The `serverip` variable can be undefined in the U-Boot configuration. PetaLinux provides a `platform-top.h` file which is used to change the configuration. Figure I.1 shows which definitions were added into the `platform-top.h` file:

```
#ifdef  CONFIG_SERVERIP
#undef  CONFIG_SERVERIP
#define CONFIG_SERVERIP
#endif

#ifdef  CONFIG_BOOTP_SERVERIP
#undef  CONFIG_BOOTP_SERVERIP
#endif
```

Figure I.1: Modification of the U-Boot configuration to undefine the default value for the TFTP server IP-address.

The DCHP-request can save the IP-address of the TFTP server when the `CONFIG_SERVERIP` option is redefined without any value. `CONFIG_BOOTP_SERVERIP` is undefined to allow storing of the TFTP server IP-address in the `serverip` variable. This definition is used to specify that the `serverip` variable should be used to store the IP-address of the DHCP-server instead of the TFTP server [136]. This is not desired.

I.2.2 MAC-address retrieval from ZCU102 EEPROM

U-Boot retrieves the MAC-address from the EEPROM on the ZCU102 development board when configured correctly. Figure I.2 shows which configuration options need to be enabled to retrieve the MAC-address from the EEPROM [109,133]. The configuration options enable the use of an I^2C EEPROM, the I^2C address of the EEPROM, and the offset of the MAC-address in the storage device.

```
CONFIG_I2C_EEPROM                    = y
CONFIG_SYS_I2C_EEPROM_ADDR           = 0x54
CONFIG_SYS_I2C_EEPROM_ADDR_OVERFLOW  = 0x0
CONFIG_ZYNQ_GEM_I2C_MAC_OFFSET       = 0x20
```

Figure I.2: U-Boot configuration options for using the MAC-address from the I^2C EEPROM on the ZCU102.

The values of the options have been copied from the ZCU102 board support package. The configuration options have been put in a `.cfg` file. The file has been added to the U-Boot recipe of the PetaLinux project.

I.2.3 Device-tree modifications

EEPROM, I2C, and ethernet hardware

The EEPROM on the ZCU102 needs to be defined in the device-tree in addition to the U-Boot configuration. The device-tree will inform U-Boot about the EEPROM hardware[1]. The definition of the EEPROM hardware in the device-tree can be seen in Figure I.3.

The EEPROM *node* in the device-tree contains definitions of the values that are stored in the EEPROM. Figure I.3 shows that the MAC-address is stored on address `0x20` and takes up six bytes in the EEPROM.

[1] Just like the Linux kernel, U-Boot uses the device-tree to find non-discoverable hardware [126].

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

  board_sn:       board-sn@0        {  reg = <0x0 0x14>; };
  eth_mac:        eth-mac@20        {  reg = <0x20 0x6>; };
  board_name:     board-name@d0     {  reg = <0xd0 0x6>; };
  board_revision: board-revision@e0 {  reg = <0xe0 0x3>; };
};
```

Figure I.3: Definition of the I^2C EEPROM with the MAC-address in the devic-tree source code.

The device-tree also needs to define the I^2C hardware. The I^2C-bus of the Zynq MPSoC is connected to a multiplexer on the ZCU102 [137]. Hardware modules that control pin multiplexing are designed as *pin-controllers* [138]. A pin-controller node is therefore required, in addition to an I^2C-bus node, to correctly bind the device driver of the I2C multiplexer and EEPROM in U-Boot.

Furthermore, the Gigabit Ethernet Module (GEM) of the Zynq MPSoC also needs to be defined in the device-tree. This hardware module is used for networking. The definitions of the EEPROM, I^2C-bus, multiplexer, pin-controllers, and GEM have been copied from the ZCU102 board support package.

A copy of the device-tree, with all of the components that have been mentioned above, can be found in the ZIP-archive that is included with the thesis (see Appendix J)

Linking the EEPROM and adding boot arguments

The EEPROM in the device-tree also needs to be linked using a *phandle*[2]. The phandle is added in the *chosen* node. This node is part of the device-tree's *root* node and contains data that gets passed to U-Boot and the Linux kernel [139, 140]. The addition of the phandle is shown in Figure I.4.

```
/ {
  model = "CMS DAQ ZynqMP ZCU102 board";
  compatible = "xlnx,zynqmp";

  chosen {
    xlnx,eeprom = &eeprom;
    bootargs = "earlycon console=ttyPS0,115200 clk_ignore_unused
                earlyprintk cpuidle.off=1 root=/dev/nfs ip=dhcp rw";
  };
};
```

Figure I.4: Linking the EEPROM to the EEPROM node using a phandle.

In addition to the EEPROM link, there is also a possibility to add boot arguments for the kernel to the chosen node. Figure I.4 shows how the boot arguments for mounting the root filesystem via NFS have been added (The other boot arguments in Figure I.4 come from the PetaLinux configuration).

[2]Phandles are used in device-trees as "pointers" to point to the definition of a node [140].

J. Contents of attached ZIP-archive

```
Additional_thesis_content_NekijaDzemaili.zip
├── Bachelor_thesis_NekijaDzemaili.pdf
├── Reflection_paper_NekijaDzemaili.pdf
├── latex_projects
│   ├── bachelor_thesis
│   └── reflection_paper
├── git_repositories
│   ├── centos8-rootfs-aarch64
│   ├── crashkernel
│   ├── io-board
│   ├── relboot-&-relup
│   ├── reliable-booting-system-bsp
│   ├── watchdog-timer-heartbeat
│   └── zynq-mpsoc-guides-docs
└── relboot-relup_debug_output
```

Figure J.1: Directory structure of the ZIP-archive with additional content of the thesis.

Table J.1 Description of files and directories in the additional thesis content ZIP-archive.

File/directory	Description
`Bachelor_thesis_NekijaDzemaili.pdf`	Bachelor thesis PDF file.
`Reflection_paper_NekijaDzemaili.pdf`	Reflection paper PDF file.
`latex_projects`	LaTeXprojects of documents. This includes all LaTeXsource files which were used to generate the bachelor thesis and reflection paper PDF files.
`git_repositories`	All Git repositories of the reliable booting system project.
`centos8-rootfs-aarch64`	Guide for building aarch64 CentOS 8.
`crashkernel`	Source code for the crashkernel and documentation.
`io-board`	Source code for the IO-board.
`relboot-&-relup`	Source code for the RELBOOT & RELUP mechanisms.
`reliable-booting-system-bsp`	Board support package with CI.
`watchdog-timer-heartbeat`	Source code for the watchdog timer heartbeat daemon.
`zynq-mpsoc-guides-docs`	Documentation and guides written for Zynq MPSoC.
`relboot-relup_debug_output`	Debug output of RELBOOT & RELUP tests.