Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 454b

Systems Group, Department of Computer Science, ETH Zurich

Trusted Firmware for a Research Computer

by

Alessandro Legnani

Supervised by

Daniel Schwyn
Prof. Dr. Timothy Roscoe

February 2023 – August 2023

D INFK

Abstract

Firmware plays a crucial role in the boot process of every computer. However, firmware stacks often suffer from issues related to their size and complexity, typically stemming from years of patchwork to support new platforms or fix bugs.

This thesis delves into the realm of firmware by rewriting the initial stage of the firmware stack of the Enzian research computer. The emphasis of this study is on understanding and optimizing the steps required for the successful initialization of Dynamic Random Access Memory (DRAM) in contemporary systems.

Through the process of rewriting, this work demonstrates that dedicated efforts can yield remarkable results. The effort invested in this endeavor has led to a substantial reduction in the size of the code base. This improvement has not only made the code base more manageable for research teams but has also yielded enhanced performance metrics, such as boot time. These findings underscore the practical impact of focused firmware development and the tangible benefits it can deliver.

Acknowledgements

I am grateful for the assistance and support of several individuals who played a crucial role in the completion of this thesis. I extend my heartfelt gratitude to Prof. Dr. Roscoe for his instrumental role in making this project possible.

I would also like to express my appreciation to my supervisor, Daniel Schwyn, for dedicating substantial time and effort to guide me through the research process. Your expertise and insightful feedback have been instrumental in shaping the direction of this thesis.

Furthermore, I want to extend my sincere gratitude to Dr. Cock for his invaluable contribution. The foundational code he authored formed the basis of this thesis, especially the work presented in Section 3.6, 3.8, and 4.3.1.

My family and girlfriend deserve special thanks for their constant support, proofreading assistance, and helping me overcome my try-to-undestand-DRAM-initialization-induced delirium.

Contents

1 Introduction **4**

2 Background and Motivation **7**
- 2.1 Enzian . . . 7
- 2.2 Bring-up and Diagnostics Kit (BDK) . . . 7
- 2.3 Arm Trusted Firmware (ATF) . . . 7
  - 2.3.1 ARM's Exception Levels . . . 8
  - 2.3.2 Arm's Downcalls . . . 8
  - 2.3.3 Boot Loader Stages . . . 9
  - 2.3.4 ATF Components and Libraries . . . 10
- 2.4 Board Management Computer (BMC) . . . 11
- 2.5 Enzian Firmware Resource Interface (EFRI) . . . 11
- 2.6 Dynamic Random Access Memory (DRAM) . . . 12
  - 2.6.1 Organization of DRAM . . . 12
  - 2.6.2 DRAM Registers . . . 13
  - 2.6.3 Memory Refresh . . . 13
  - 2.6.4 ZQ Calibration . . . 14
  - 2.6.5 Clock and Strobe . . . 14
  - 2.6.6 Read & Write Centering . . . 14
  - 2.6.7 Different DDR4 Topologies . . . 15
  - 2.6.8 Write & Read Leveling . . . 16
  - 2.6.9 V_{ref} Training . . . 17

3 Implementation of the ATF port **19**
- 3.1 Assessment of the old firmware stack . . . 19
  - 3.1.1 What can be improved . . . 20
- 3.2 Boot Procedure . . . 20
- 3.3 SPI Flash Memory layout . . . 21
- 3.4 The ThunderX's scratchpad . . . 21
- 3.5 Shared behavior of BL stages . . . 23
- 3.6 UART and console initialization . . . 23
- 3.7 Timer initialization (GTI) . . . 24
- 3.8 Serial Presence Detect (SPD) . . . 24
- 3.9 Memory Layout of Enzian . . . 25
  - 3.9.1 Partition into Secure and Non-Secure DRAM . . . 25
  - 3.9.2 MMU - Memory Management Unit . . . 25
  - 3.9.3 Design decision regarding the memory layout . . . 27
  - 3.9.4 Further space optimizations . . . 28
- 3.10 Loading next stages from the SPI flash . . . 28
  - 3.10.1 Firmware Image Package (FIP) . . . 28
  - 3.10.2 Storage abstraction layer setup . . . 28
- 3.11 Generic Interrupt Controller (GIC) . . . 29
- 3.12 Power State Coordination Interface (PSCI) . . . 30

4 Memory controller and DRAM initialization **31**
4.1 Overview . . . 31
4.2 Topology of the LMCs . . . 32
4.3 LMC Initialization Sequence . . . 32
4.3.1 Steps 1 through 6 . . . 33
4.4 Step 7: Early LMC Initialization . . . 33
4.5 Steps 9 through 11 . . . 33
4.6 Testing the LMCs . . . 34
4.6.1 LMC and bank selection algorithm . . . 34
4.6.2 Testing methodology . . . 34
4.7 Step 12: Write-Leveling . . . 35
4.8 Step 13: Read-Leveling . . . 36
4.9 Step 14: V_{ref} Training . . . 36
4.10 Training Algorithms . . . 37
4.10.1 1st Algorithm . . . 37
4.10.2 2nd Algorithm . . . 37
4.10.3 3rd Algorithm . . . 39
4.10.4 4th Algorithm . . . 41
4.10.5 Takeaways and future work . . . 41

5 Configuration of the Enzian **42**
5.1 Implementation of BMC-side EFRI backends . . . 42
5.2 Configuration of DRAM parameters using EFRI . . . 43
5.3 EFRI as an EL3 service . . . 44
5.4 Modifications to the ATF-side . . . 45
5.5 Future work and improvements . . . 45

6 Evaluation **47**
6.1 Firmware Size . . . 47
6.2 EFRI performance . . . 47
6.3 Time to UEFI . . . 48
6.4 DRAM latency and bandwidth . . . 49
6.5 Code base comparison . . . 50

7 Future work **52**
7.1 DRAM initialization . . . 52
7.2 Extensive testing in Linux . . . 52
7.3 Reset behaviour . . . 52
7.4 PSCI implementation . . . 53

8 Conclusion **54**

9 Appendix **55**
9.1 Updating ATF . . . 55
9.2 Building and flashing the ATF . . . 55
9.2.1 Manually building the ATF . . . 55

9.2.2 Flashing an EBB 56
9.2.3 Flashing an Enzian 56
9.3 Using EFRI on an Enzian 56
9.4 Debugging an EBB using JTAG 56
9.5 Code base measurements 57
9.6 Summary of the Capabilities and Limitations of the new ATF . . 58

1 Introduction

Modern computer systems have become increasingly complex and sophisticated. This is the case even before the operating system starts executing, when booting the system. The crucial component that performs the initial stages of system initialization is the firmware.

The booting process of a contemporary system is no longer the straightforward sequence it once was. Rather, it has morphed into an intricate dance, attempting to adhere to various historical standards while accommodating newer ones, often leading to a convoluted and cumbersome firmware stack. As a consequence, duplication of initialization code has become rampant, needlessly increasing the firmware's size and hindering the potential for optimized and streamlined boot sequences.

Furthermore, the current state of firmware implementations is the culmination of years of patchwork, necessitated by the need to support a wide array of different and/or outdated hardware platforms. Over time, as new technologies emerged and older ones persisted, firmware developers faced the challenge of accommodating divergent standards, resulting in a fragmented firmware stack that intertwines various initialization routines, often leading to redundancy and complexity.

In the case of Enzian [Cock et al., 2022], at power-on, the Baseboard Management Controller (BMC) initializes the various hardware components of the system (power controllers, clock distribution, etc.) and takes the CPU out of reset. As per the Arm convention only the first core starts executing code from the boot ROM and enters a set of initialization functions.

Before this thesis these were provided in the Bring-up and Diagnostics Kit(BDK) from the SoC's manufacturer. The BDK would, most importantly, initialize the DRAM and other subsystems like the UART. The boot flow then continues with the Arm Trusted Firmware (ATF) [Arm, 2023] that has the following major functions:

- Finish initializing the rest of the system hardware.
- Install the secure system monitor, which is code running at EL3, the most privileged execution mode available on the processor, and providing services to the OS code through traps at runtime called Secure Monitor Calls (SMC).
- Execute the next stage of the boot process, which is the Unified Extensible Firmware Interface (UEFI) firmware.

This approach at booting the system is working, but it is less than ideal as it faces the following four issues:

- Outdated ATF: The firmware relies on an older and unknown version of the ATF, which needs to be updated to leverage the latest features and improvements offered by the reference Arm Trusted Firmware.

- Proprietary Hardware Initialization: Parts of the hardware initialization, like for the DRAM and the UART consoles, is done by the BDK. This code is both proprietary and lacks any version control information. This, in turn, poses a challenge in terms of maintainability and restricts the openness of the platform.

- Duplicated Initialization Code: The initialization code found in the BDK, ATF and UEFI are either duplicated or split between these three stages. One example is PCIE: part of the initialization is done in the BDK, whilst the rest is done in UEFI.

- Bloated and Convoluted Code Base: The firmware stack for Enzian suffers from a large and complex code base, resulting from the accumulation of patchwork over time to support various hardware platforms and standards. This complexity hinders maintainability, makes it challenging to identify and fix issues.

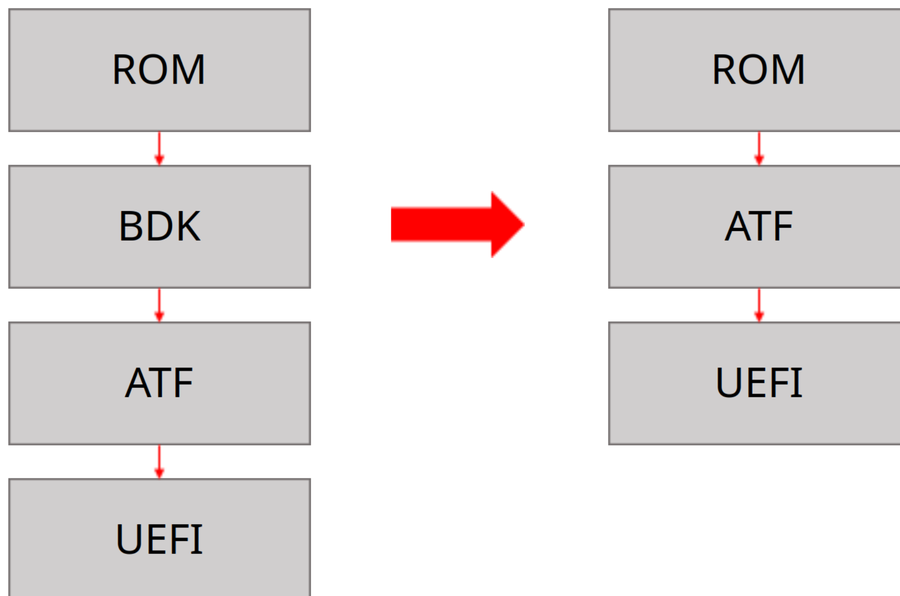


Figure 1: Diagram of both the old and new firmware stacks.

The ultimate goal of this project is to consolidate the separate initialization programs into a unified firmware solution, shown in Figure 1, and bring it up to date with the latest version of the reference ARM Trusted Firmware.

Notably, the most significant obstacle has arisen during the DRAM initialization procedure, which subsequently takes center stage in this thesis. This unification will enhance maintainability and optimize performance (see Section 6). As Enzian's strength lies in its reconfigurability the ability to make software

modifications for diverse hardware setups should be made as straightforward as possible for all involved parties.

2 Background and Motivation

2.1 Enzian

Enzian is a cache-coherent 2-node asymmetric NUMA system where one node is a 48-core Cavium ThunderX[Cavium, 2017] CPU and the other node is a Xilinx FPGA. The two nodes have access to 128 GB and up to 1 TB of DDR4 memory respectively. The Enzian Coherent Interconnect (ECI) connects the two nodes. It is based on the cache coherence protocol of the ThunderX processor and the FPGA emulates the coherence protocol, as if it were another ThunderX.

2.2 Bring-up and Diagnostics Kit (BDK)

The Bring-up and Diagnostic Kit (BDK), provided by Cavium, serves as the first stage bootloader for the Enzian project. It combines a BIOS-like configuration and a first-stage bootloader that can boot different images, such as UEFI.

Additionally, the BDK includes a unique diagnostic tool, featuring a full Lua shell, to aid in troubleshooting and diagnostics. However, the BDK has some significant drawbacks that need to be addressed for optimal performance and maintainability within the Enzian project:

- Big code size: The BDK has a substantial codebase, which can lead to unnecessary resource consumption and complexity.
- Duplicate functionality: Some functionalities within the BDK overlap with tasks that should ideally be handled by the UEFI, causing duplication and inefficiency in the boot process.
- Clunky boot process: The boot process with the BDK is not as streamlined as desired. Changing a setting, like the DRAM speed grade, requires the user to interrupt the boot process via a specific key, navigate the different menus with the command line, change the settings and save them, before continuing the boot flow.
- Support for multiple platforms: The BDK supports multiple platforms beyond the ThunderX processor, which adds complexity and complicates maintenance efforts.
- Proprietary nature: The whole BDK is proprietary, limiting the openness of the Enzian platform.
- Patchwork solutions: The BDK incorporates patched solutions to make the ThunderX work as part of the Enzian system, leading to potential complications and maintenance challenges.

2.3 Arm Trusted Firmware (ATF)

The ARM Trusted Firmware (ATF) [Arm, 2023] is an open-source firmware project developed by the ARM Foundation. It serves as a standardized firmware

implementation for the ARM platform. ATF's primary purpose is to facilitate the boot process and system initialization of ARM-based platforms. It provides a reliable and extensible firmware solution that supports the proper functioning of the hardware and software components within the system.

The ATF consists of several firmware components each responsible for specific tasks related to bootstrapping and initialization. Therefore, they need to run at different privilege levels.

2.3.1 ARM's Exception Levels

In ARM-based systems privilege levels are called Exception Levels [Arm, 2022]. There are 4 exception levels from EL0 to EL3. Additionally EL0, EL1 and EL2 are split into secure and non-secure as shown in Figure 2.
This distinction between non-secure and secure is a feature of ARM Trustzone, whereas there exist two different worlds, secure and non-secure, that are hardware separated. Code running in secure mode (or EL3) can access both worlds, whilst non-secure code can only access the non-secure world. This partitioning is used in order to increase the security of services handling authentication or cryptography. By performing a trusted boot and running a trusted OS a root of trust can be established called the Trusted Execution Environment (TEE).

As Enzian is meant as a system to be experimented upon these security measures currently are not used. The architecture does not specify at which level different software should run, but a common usage model is the following:

- EL3: Firmware and Secure Monitor
- EL2: Hypervisor
- EL1: OS kernel
- EL0: User code

2.3.2 Arm's Downcalls

Within the ARM architecture, the establishment of communication channels across different exception levels is facilitated through three distinct downcall mechanisms [Arm, 2017]:

- SuperVisor Calls (SVC): SVC downcalls enable communication from the Application code executing at EL0 to the Operating System kernel executing at EL1. This mechanism permits controlled transitions from user-mode applications to the kernel, enabling privileged operations and interactions with system resources.
- HyperVisor Calls (HVC): HVC downcalls enable communications from the OS kernel executing at EL1 to the Hypervisor operating at EL2. This channel allows the OS kernel to interact with the hypervisor, granting access to virtualization-related functionalities and facilitating management of virtualized environments.

- Secure Monitor Calls (SMC): SMC downcalls provide a means for communication from the OS kernel executing at EL1 to the Secure Monitor operating at EL3. This allows the OS kernel to interact with services provided by the Secure Monitor like power management or, in the case of the Enzian, communication with the BMC (see Section 2.4) through the EFRI link (see Section 2.5).

2.3.3 Boot Loader Stages

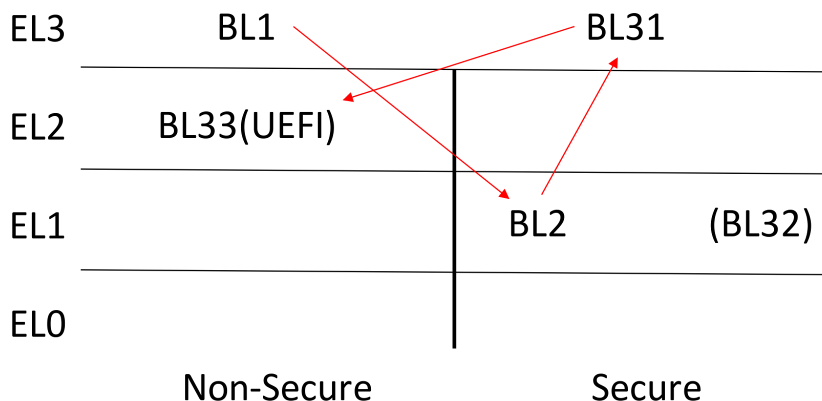


Figure 2: Different stages of the ATF and their privilege level. The red arrows indicate the flow of execution.

As mentioned before the ATF consists of multiple stages, referred to with the prefix BL (Boot Loader). The exception level at which they run is indicated in parentheses and the suffix 'S' or 'N' stand for secure and non-secure, when applicable. For a more schematic view see Figure 2.

1. BL1 (EL3) is the initial stage of the Trusted Firmware, implementing the reset vector from which execution starts after a reset or power on of the system. The primary responsibility of BL1 is to set up the hardware, perform basic initialization, and load the next stage of the boot process, BL2, into memory from non-volatile storage. It either passes control to the BL2 stage or an optional Firmware Update process (BL2U).

2. BL2 (EL1S) loads the next stages of the boot process, which typically include BL31, BL32 and BL33. If Trustzone is used BL2 also validates the authenticity and integrity of the next stages and performs some secure world specific initialization, before passing control to BL31.

3. BL2U (EL3) is an optional stage that may be included in certain implementations of the Trusted Firmware for ARM. Its primary function is to provide the capability to update the BL2 stage itself without the need for a complete system firmware update.

4. BL31 (EL3), also known as the secure monitor, is the part of the firmware that remains resident during normal execution and handles interrupts or can be invoked through SMC calls. These can be used to e.g. power off the system or get telemetry data from the 3.3V voltage rail.

5. BL32 (EL1S) represents the Trusted Operating System in the TEE. It is loaded and executed after BL31 and provides a secure execution environment for trusted applications and services. BL32 manages secure resources, such as cryptographic keys, and ensures isolation between the trusted and non-trusted parts of the system.

6. BL33 (EL2N) is the final stage of the boot loader and represents the boot loader for the Normal World (non-secure) software. It is responsible for loading and launching the non-secure operating system or other non-secure software components, such as the Unified Extensible Firmware Interface (UEFI) used by Enzian, which finishes up on platform initialization and boots the operating system.

2.3.4 ATF Components and Libraries

The ATF framework provides numerous libraries that serve as versatile and reusable solutions, eliminating the need for rewriting code for each platform and providing essential functionalities for ARM-based systems.

- XLAT (Translation Library): The XLAT library provides a layer of abstraction on top of the Memory Management Unit (MMU) in ARM systems. By handling memory address translation, XLAT ensures seamless mapping between virtual and physical addresses, facilitating memory management. Thanks to its standardization within the ARM architecture, this library can be utilized across different ARM-based platforms without modification. This reusability streamlines the development process, as developers can rely on XLAT to manage memory translation, saving time and effort while ensuring consistent and optimal memory access in various ARM systems.

- Storage Abstraction Layer: The storage abstraction layer in the ATF framework serves as a versatile and general-purpose solution for loading and parsing the Firmware Image Package (FIP) during system boot. The FIP consists of a header pointing to different binaries and the binaries itself. In the Enzian the FIP contains essential firmware components, such as the BL2, BL31 and BL33. The IO library is designed in a platform-agnostic manner, making it applicable to a wide range of ARM-based systems without code modifications. Its reusability ensures seamless and reliable firmware loading across diverse platforms, providing a consistent boot process.

- Generic Interrupt Controller (GIC): The GIC is a critical component responsible for managing interrupts in ARM processors. It handles interrupt

requests from various hardware sources and distributes them to the appropriate processing elements within the system. The GIC ensures that interrupts are delivered in a prioritized and orderly manner, minimizing latency and enabling timely response to critical events. The presence of a library to configure and use the GIC within the ATF framework enables standardized interrupt management across different ARM platforms.

- Power State Coordinator Interface (PSCI): The PSCI is a standardized interface within the ATF framework that facilitates power state coordination in ARM-based systems. It allows software components to communicate with the platform's power management firmware, enabling efficient power management and dynamic power state transitions. Through the PSCI, software can request the system to enter various power states, such as sleep, suspend, or power-off, depending on the system's operational requirements. The PSCI's standardized interface ensures that power management functionality can be seamlessly integrated into different ARM platforms, enabling developers to write power-aware software that works consistently across various devices.

In summary, the ARM Trusted Firmware libraries exemplify the benefits of standardized and reusable solutions. By adhering to the ARM architecture standards and adopting platform-agnostic designs, these libraries enable seamless memory management and firmware loading across various ARM-based platforms. This approach significantly streamlines the development process, promotes code consistency and enhances the reliability of ARM systems.

2.4 Board Management Computer (BMC)

A Baseboard Management Controller (BMC) is a critical component found in modern server computer platforms that plays a pivotal role in the overall system management and remote monitoring capabilities. In the Enzian the BMC is a full fledged computer running OpenBMC, a full Linux distribution, integrated directly into the server's motherboard, operating independently of the main CPU. The BMC acts as a dedicated management subsystem, providing essential functionalities like out-of-band remote access, system monitoring (e.g. temperature, voltage and fan speeds) and power control even when the main server is powered off.

2.5 Enzian Firmware Resource Interface (EFRI)

The Enzian Firmware Resource Interface(EFRI)[Xu, 2023] is an RPC-based protocol for flexible and extensible enumeration and implementation of platform-level firmware services. This enables us to have a communication link between the ThunderX and the BMC. EFRI relies on a schema defined in `enzian-efri.yml`[1] which is used to generate code for the ATF- and BMC-side

[1] https://gitlab.inf.ethz.ch/PROJECT-Enzian/bmc/enzian-firmware-resource-interface/-/blob/atf-port/enzian-efri.yml

of the implementation from the specification. This can be used to control the power management of the Enzian board e.g. to power down the CPU or read configuration options e.g. the speed of the DRAM. EFRI is being used in the BL1, as part of the DRAM initialization procedure, and in the BL31, where it is installed as an EL3 service accessible through SMC calls.

2.6 Dynamic Random Access Memory (DRAM)

In modern computer systems, one often takes for granted the availability of Dynamic Random-Access Memory (DRAM) as the main memory, providing the system with fast and volatile storage for data and program execution. DRAM is composed of tiny capacitors that store binary data as electrical charges, and the memory cells need to be periodically refreshed to maintain data integrity. While this design decision has revolutionized computing by offering high-speed, random access to data, it also introduces several challenges during the initialization process.

During DRAM initialization, a series of procedures are executed to ensure the memory's proper configuration and functionality. One of the primary challenges arises from the inherent nature of capacitors in DRAM cells. These capacitors tend to leak charge over time, leading to data loss if not continuously refreshed. Additionally, DRAM initialization involves configuring memory timings, voltage levels, and training algorithms to optimize memory performance and ensure stability. The process becomes even more intricate as memory technology advances, incorporating higher densities and faster data rates, requiring precise tuning to maximize the system's memory capabilities.

Despite the complexities involved, DRAM initialization remains a crucial process, as it lays the foundation for reliable and high-performance memory operations throughout the system's lifespan. The next sections will go over in more detail how DRAM works and which steps need to be taken in order to perform its initialization. As the Enzian system is designed to run with DDR4 memory the following sections will explain the DDR4 specific initialization steps, though some of it also applies to older and new revisions of the DDR standard.

2.6.1 Organization of DRAM

In modern computer systems DRAM is organized hierarchically to efficiently store and access data. Understanding this organization is crucial for optimizing memory performance and capacity. The organization of DRAM involves several levels, including DIMMs, ranks, banks, chips, and columns:

- DIMMs (Dual In-Line Memory Modules): DIMMs are physical memory modules that plug into memory slots on the motherboard. Each DIMM consists of multiple DRAM chips and come in various form factors, such as UDIMM, RDIMM, LRDIMM, and SO-DIMM, to cater to different system requirements.

- Ranks: Each DIMM can have one or more ranks. A rank is a logical group of memory chips that share the same data and control signals. Having multiple ranks on a DIMM allows the memory controller to access multiple sets of data simultaneously, increasing memory throughput.
- Banks: Inside each rank, the memory is divided into banks. A bank is a small unit of memory storage that can be accessed independently. DRAM chips have multiple banks, and each bank contains a portion of the total memory capacity. When the memory controller requests data from a specific address, the data is fetched from the corresponding bank.
- Chips (DRAM ICs): Each rank is made up of multiple DRAM Integrated Circuits (ICs) or chips. These chips are responsible for storing the actual data in the memory cells. The width of the data being stored can either be 4, 8 or 16 bits (referred to as x4, x8 or x16). Each chip has its own set of banks.
- Rows and Columns: Inside each DRAM chip, the memory cells are arranged in a two-dimensional grid. Data is accessed by specifying a row and a column address. When the memory controller issues a read or write command, it specifies the row address to activate the corresponding row and then the column address to access the desired data from that row.

2.6.2 DRAM Registers

DDR4 memory modules are equipped with a range of internal registers that are used in memory control, configuration, and performance optimization. Two registers, in particular, stand out as significant for the scope of this thesis:

- Multi Purpose Registers (MPR): These are 8-bit programmable registers that store predefined training patterns utilized during data training processes. Each memory module has four of these.
- Mode Registers (MR) are used to enable different operating modes of the DDR4 memory module. A note on notation: `MRn<y>` will refer to a bit or a range of bits `y` of Mode Register `n`.

2.6.3 Memory Refresh

DRAM cells, as already mentioned, store data as electrical charges on tiny capacitors. However, these capacitors are not perfectly isolated, leading to charge leakage over time.

To prevent data loss due to charge leakage, DRAM incorporates external circuitry, as part of the memory controller, that periodically reads each memory cell and rewrites its contents, restoring the charge on the capacitors to their original levels. This process, known as memory refresh, ensures the data's integrity over time. During each refresh cycle, a specific portion of memory cells is refreshed, and the process continues successively until all cells have been

refreshed. This sequential approach guarantees that all memory cells receive regular maintenance, mitigating the risk of data corruption and ensuring the reliability and stability of DRAM as the main memory in modern computer systems.

One crucial parameter involved in DRAM refresh is tREFI (Refresh Interval Time). tREFI determines the time interval between consecutive refresh cycles and is influenced by factors such as memory tensity and temperature. During initialization, the memory controller sets the appropriate value for tREFI based on the memory's specifications and operational requirements.

2.6.4 ZQ Calibration

In DDR4 modules, each data pin (DQ) serves as a bidirectional channel, responsible for sending data to the memory controller during reads and receiving data during writes. To ensure reliable data transmission, each DQ pin contains a set of parallel 240-ohm resistors. However, these resistors can experience fluctuations in resistance due to changes in voltage and temperature. To counteract this, the circuitry incorporates a DQ calibration control block, along with an external precision 240-ohm resistor that remains stable regardless of temperature variations.

Periodically the memory controller performs a specific command called ZQCS (ZQ Calibration Short). This calibration process enables the memory controller to fine-tune and adjust the impedance of the memory's output drivers. By optimizing the signaling characteristics through calibration, DDR4 memory achieves enhanced performance and ensures reliable data communication between the memory modules and the memory controller. A longer calibration sequence, ZQCS (ZQ Calibration Long), is performed at initialization.

2.6.5 Clock and Strobe

The clock signal (CLK) provides a consistent timing reference for data transfers and operations within the memory module. The clock signal oscillates at a specific frequency, and each clock cycle represents a fixed time interval during which data can be transferred. Strobes are related to the data lines in DDR4 memory.

Data in DDR4 is transferred on both the rising and falling edges of the clock signal, which is referred to as "Double Data Rate"(DDR). Each data signal is accompanied by a corresponding strobe signal (DQS - Data Strobe). The strobe signal is aligned with the data and indicates the valid data window, specifying when the data must be read or written.

2.6.6 Read & Write Centering

The main objective of read centering is to optimize the timing relationship between the data and strobe signals, ensuring accurate and reliable data sampling during read operations. During read centering, the memory controller fine-tunes

the timing parameters to identify the optimal position for the read data signals within the valid data window defined by the read strobe, which is called the data eye. The process involves sending test patterns to the memory module while iteratively adjusting the read data timing with respect to the strobe. The memory controller monitors the data response and makes adjustments until achieving the optimal read data position.

Similarly, the goal of write centering is to align the write data signals withing the window defined by the write strobe. The process the memory controller uses is analogous to the one used for read centering.

2.6.7 Different DDR4 Topologies

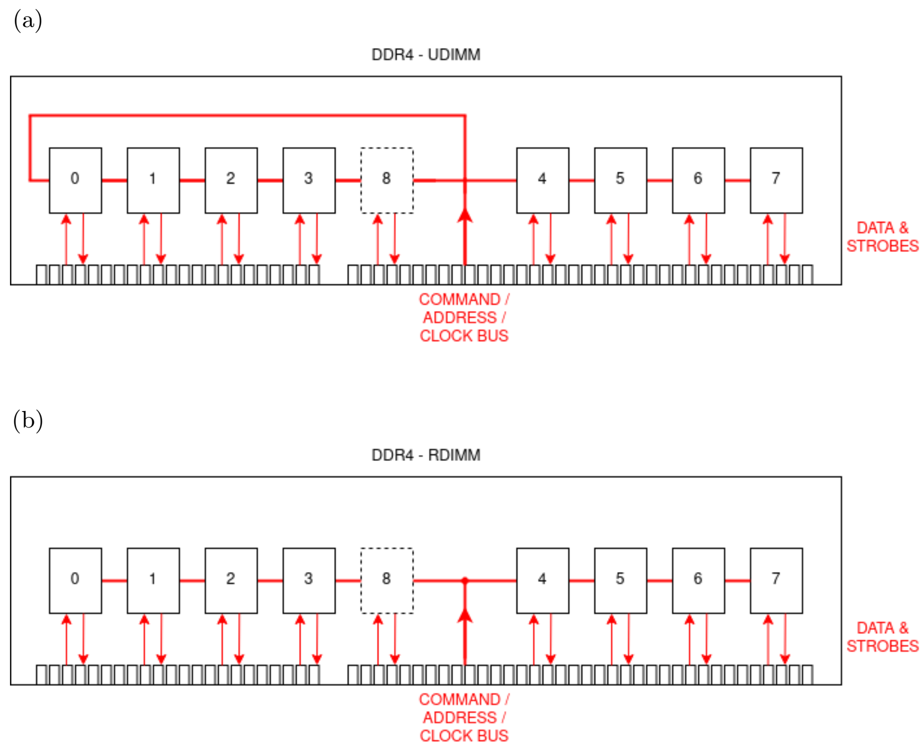


Figure 3: Fly-by topology of a (a) UDIMM and (b) RDIMM

The topology of a DDR4 memory module can exhibit variations depending on whether it is classified as a UDIMM (Unbuffered DIMM) or a RDIMM (Registered DIMM) and whether it supports ECC (Error-Correcting Code). UDIMMs are considered standard memory modules without an additional buffer between the memory chips and the memory controller (see Figure 3a). They

are commonly employed in consumer-grade systems, offering lower latency and straightforward operation. However, UDIMMs have certain limitations in terms of capacity and memory channel support, making them less suitable for high-density memory configurations.

RDIMMs, in contrast, integrate a register or buffer component that assists in reducing the electrical load imposed on the memory controller (see Figure 3b). This feature enables RDIMMs to support higher memory capacities and allows for a greater number of memory modules per memory channel, making them well-suited for server and workstation environments. Moreover, DDR4 memory modules can further differ in their ECC support, denoting the presence of additional memory bits for error detection and correction. These additional bits are stored in an additional chip on the DIMM. This optional chip is represented as the dashed chip 8 in Figure 3 for both UDIMMs and RDIMMs.

These differences in the topology of memory modules introduce challenges related to varying latencies, data access times and skews between the synchronization signals like the clock or the data strobes.

2.6.8 Write & Read Leveling

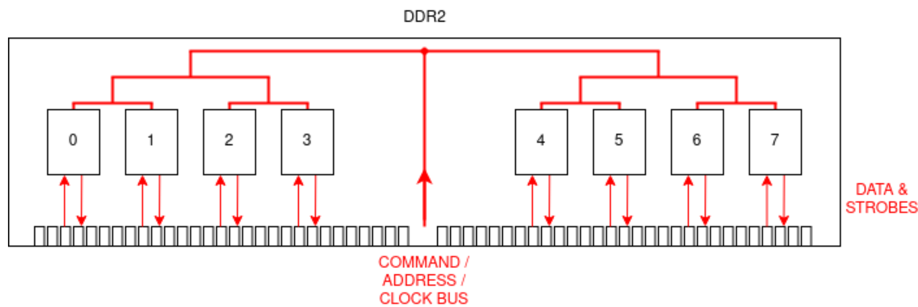


Figure 4: DDR2's symmetrical T-branch topology

From DDR3 onwards the command-, address- and clock-buses (from here onward they will be referred to as "the bus") are no longer connected to the chips on the DIMM using a T-branch as was the case with DDR2, shown in Figure 4, which ensured the same length of electrical traces to all the chips. In DDR4 the bus is routed using a fly-by approach, shown in Figure 3, to reduce the total length of the electrical traces and resulting in only one or two points were the bus needs to be terminated, depending on the type of the memory module. This design choice was made to accommodate for higher transfer speeds but it also introduces some new challenges.

The DQS, from the data lines, and the CLK, from the bus, can get miss aligned due to the different flight-times caused by the difference of length of the electrical traces. IN DDR4 memory this skew between DQS anc CLK are different from chip to chip and depend on whether a read or a write is being

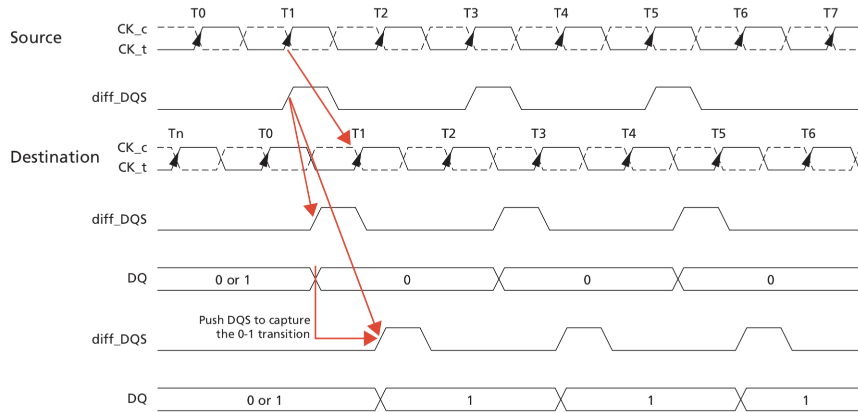


Figure 5: Waveforms of a write leveling sequence [Micron, 2014]

performed. A training step is thus needed to experimentally determine the amount of delay the memory controller has to add to align the different DQS anc CLK signals. This process is called write leveling, when performed for writes, and read leveling, when testing reads. How this training step is done varies between memory controllers but the general idea of write leveling is the following:

When the DRAM is in write-leveling mode it uses the Data Strobe (DQS) to sample the Clock (CK) and return the sampled value back to the controller through the DQ bus. In order to enable the write leveling mode `MR1<7>` is set to 1. The controller sends a series of DQS pulses, which are used to sample the CLK and the sampled value is sent back via the DQ bus. If a 0 is returned it means that DQS and CLK are not aligned. The memory controller therefore increments or decrements the DQS delay and sends a new series of DQS pulses. This is repeated until a 0-to-1 transition is observed. The delay value for the DQS is then locked in and the memory controller proceeds to calibrate the next DQS buses.

The read leveling, on the other hand, is performed by first enabling the predefined pattern for system calibration via a write to MR3. Then multiple read operations are performed at different internal delay settings, before disabling the pattern via a second MR3 write. The results from the read operations can then be used to find the best delay settings to align the DQS and CLK.

2.6.9 V_{ref} Training

The reference voltage, denoted as V_{ref} or VrefDQ, serves as the threshold voltage that distinguishes a logical 0 from a 1 on the data lines (DQ) in memory systems. Notably, in DDR4 memory architecture, a pivotal transformation in the termination style of these data lines occurred, transitioning from Center Tapped Termination (CTT), also referred to as SSTL Series-Stud Terminated

Logic, to Pseudo Open Drain (POD). This shift aimed to bolster signal integrity at higher speeds while also conserving IO power consumption.

In the CTT termination approach, a memory voltage divider circuit is employed, resulting in a reference voltage of half the DQ voltage ($\frac{V_{DDQ}}{2}$). This setting establishes a fixed point of reference for distinguishing logic levels.

Conversely, the adoption of the Pseudo Open Drain (POD) design in DDR4 introduced a departure from the voltage divider approach. Consequently, the reference voltage (V_{ref}) became adjustable, allowing it to be configured to different levels. In this architecture, the reference voltage can be tuned within two distinct ranges defined by the JEDEC specification.

Range1 spans from 60% to 92.5% of V_{DDQ}, while range2 encompasses the 45% to 77.5% interval of V_{DDQ}. Notably, both ranges consist of steps sized at 0.65%. This design decision empowers finer-grained control within the central portion of the voltage range, albeit with reduced precision towards the extremes.

It's important to underscore that there exists no universally prescribed method to establish the optimal reference voltage configuration. Typically, an iterative approach is employed, involving the repetitive writing and reading of cache lines to assess correctness and identify the appropriate value. This pragmatic technique allows for a dynamic determination of the optimal V_{ref} setting, depending on the memory module characteristics and environmental variable, like temperature, that can impact signal integrity.

3 Implementation of the ATF port

The new implementation of the ATF, at the time of writing, is based on version 2.9 of the reference design from May 2023. To update to the newest release please refer to the appendix.

Most of the code for the different subsystems of the ThunderX processor can be found in `drivers/cavium/thunderx` and its respective include directory `include/drivers/cavium/thunderx`. Here all the definitions of the Control and Status Registers (CSRs) can be found as well as convenience functions to initialize or use the subsystem e.g. the memory controller.

The implementation of the connection between the BMC and the ThunderX, via the EFRI link, is located in `lib/efri`. All the remaining code for the different BL stages, including the DRAM, MMU and GIC initialization code is located in the directory for the Enzian platform `plat/enzian`. This would, in theory, allow the code to be pushed to the upstream repository of the ATF. The only modification of code not in these directories is the addition of the definition of the EFRI service in `smccc.h`[2].

3.1 Assessment of the old firmware stack

To better understand how to port the old BDK-ATF stack to the new ATF a survey of the functions of the old stack was done. The BDK initializes (sometimes only partially) the following subsystems:

- UART console
- DRAM
- CCPI, which is the coherency link to the FPGA
- the on-chip NIC (BGX)
- USB
- PCIE (the initialization is completed in UEFI)
- TWSI (Two wire serial interface)

The ATF finishes up the initialization of the TWSI interface, initializes GPIO and SGPIO and implements a quirk for the SATA controller. (The ThunderX has an issue where SATA drives may randomly drop out if power management is enabled on two lanes of a half of a QLM. Please refer to `thunder_sata.c`[3].) Part of the old ATF implementation loads a flat-device-tree (FDT) file from the SPI flash to main memory for the UEFI to load.

[2] https://gitlab.inf.ethz.ch/PROJECT-Enzian/arm-trusted-firmware-enzian-port/-/blob/v2.9-enzian/include/lib/smccc.h

[3] https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-atf/-/blob/master/plat/thunder/thunder$_{sata.c}$

The arguably most important part of this thesis is the implementation of the DRAM initialization. The majority of it is done in `dram-init-ddr3.c`[4] in the BDK. The whole initialization process comprises around 10 thousand lines of code including behaviour specific to different processor beyond the ThunderX and specific to both DDR3 and DDR4 initialization. The DRAM initialization sequence can be split into two phases: the linear phase, which is executed once, and the iterative phase, which is repeated until suitable settings for the memory are found.

3.1.1 What can be improved

- Removal of precompiled FDT: As already hinted at before, the old ATF loads a FDT and passes it to the UEFI. Ideally UEFI should be able to identify the devices on the board automatically and generate all the necessary ACPI tables, without the need for them to be precompiled.

  The UEFI of the Enzian is, at the time of writing, in the midst of being ported to the latest release of EDK2. The design choice has been made to omit the FDT passing in favor of having it implemented in the new UEFI. This makes the new ATF not compatible with the old UEFI.

- Initialization partitioning: Initializing the various subsystems of the ThunderX processor is spread out in the different stages of the firmware. For the initialization of PCIE some steps are performed in the BDK, whilst the rest of the implementation is done in the UEFI.

  With the new design we want to clearly divide the scope of what the ATF and the UEFI have to perform in regard to initialization. The rule-of-thumb for the new implementation is to have only the necessary in the ATF, most importantly the DRAM, and move the rest to the UEFI where higher level features are available. This seems obvious but the BDK, the first stage of the firmware stack, initializes the ThunderX's NIC and has a fully working network stack. In conclusion the new ATF will only include the UART and DRAM initialization.

- Reimplementation of DRAM initialization: The old code for the DRAM initialization includes jumps and as such is, also due to its impressive size, difficult or even sheer impossible to understand. The decision has been made to completely reimplement the entirety of the DRAM initialization.

3.2 Boot Procedure

The first step in booting the ThunderX of the Enzian is the boot ROM, which starts execution in EL3 at address 0x87D_000_000_000 stored in the `RVBAR_EL3` register. At the start of the boot process DRAM is not yet initialized warranting the question how a system can run without access to memory. In the case of the

[4] https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-bdk/-/blob/master/libdram/dram-init-ddr3.c

ThunderX this problem is solved by allocating a scratchpad in the L2 cache. A more detailed explanation can be found in Section 3.4. The boot ROM loads either the Trusted or Non-trusted boot-level 1 firmware from the SPI flash into the memory region from 0x100_000 to 0x180_000, and starts executing it.

After initializing the DRAM the BL1 unlocks the cache and, if the DRAM is properly configured, the system is now running from main memory. To load the next stage of the ATF the BL1 first initializes the IO interface to the SPI flash. Then the FIP (Firmware Image Package) is loaded from address 0x80000 of the Flash to main memory at address `PLAT_FIP_BASE`. The FIP contains all the next stage bootloaders, namely BL2, BL31 and BL33 (UEFI). Using the information in the header of the FIP BL1 loads BL2 into main memory at BL2_BASE and hands of control to BL2. BL2 loads both BL31 and BL33 into main memory. Control is then handed over to BL31, which, in turn, hands over to BL33.

3.3 SPI Flash Memory layout

The Enzian board has an SPI flash that is used to boot the machine. In order for the ThunderX to execute the BL1 the flash has to have all the regions marked as "Required by ThunderX". From 0x80_000 upwards arbitrary data can be stored. We store the FIP containing BL2, BL31 and BL33 there. Depending whether Trusted or Non-trusted boot is selected the appropriate sections will be used to continue execution. In case the Non-Trusted firmware was loaded, it is placed at 0x120_000 and execution starts at 0x120_100. If a Trusted boot is performed the Trusted firmware is placed at 0x150_000 and execution starts at 0x150_100. In case of the Enzian system only the Non-trusted boot option will be used. As each stage of the ATF has to be page-aligned we need to add padding such that BL1 gets loaded to 0x121_000 and we put a jump instruction at 0x120_100 that jumps to the base of BL1 (the addresses found in the ThunderX manual in §12.2.4 are off by one additional 0). This and the populating of the "code-load information block" and "code-signature information block" is handled by the `ThunderX Boot Flash Tool` [Cock, 2022].

3.4 The ThunderX's scratchpad

As already hinted at before the ThunderX uses a scratchpad in the L2 cache to execute without main memory. This 512 KiB big memory region, ranging from 0x100_000 to 0x17F_FFF comfortably fits into the L2 cache of the ThunderX, which is 16 MiB in size. After the boot ROM fills this memory region with the boot-level 1 firmware, in this case the BL1, we lock every cache line associated with this memory region using the `L2 Cache Fetch and Lock` (see ThunderX manual §2.12.4). These cache lines will thus never get evicted and they can be used to execute the BL1 albeit with limited available memory. We do this in the BL1 to ensure that we don't accidentally evict part of our currently running binary. To later unlock the cache lines, after DRAM is initialized, the `L2 Cache Hit Writeback` instruction is used. Implementations of both functions in C can

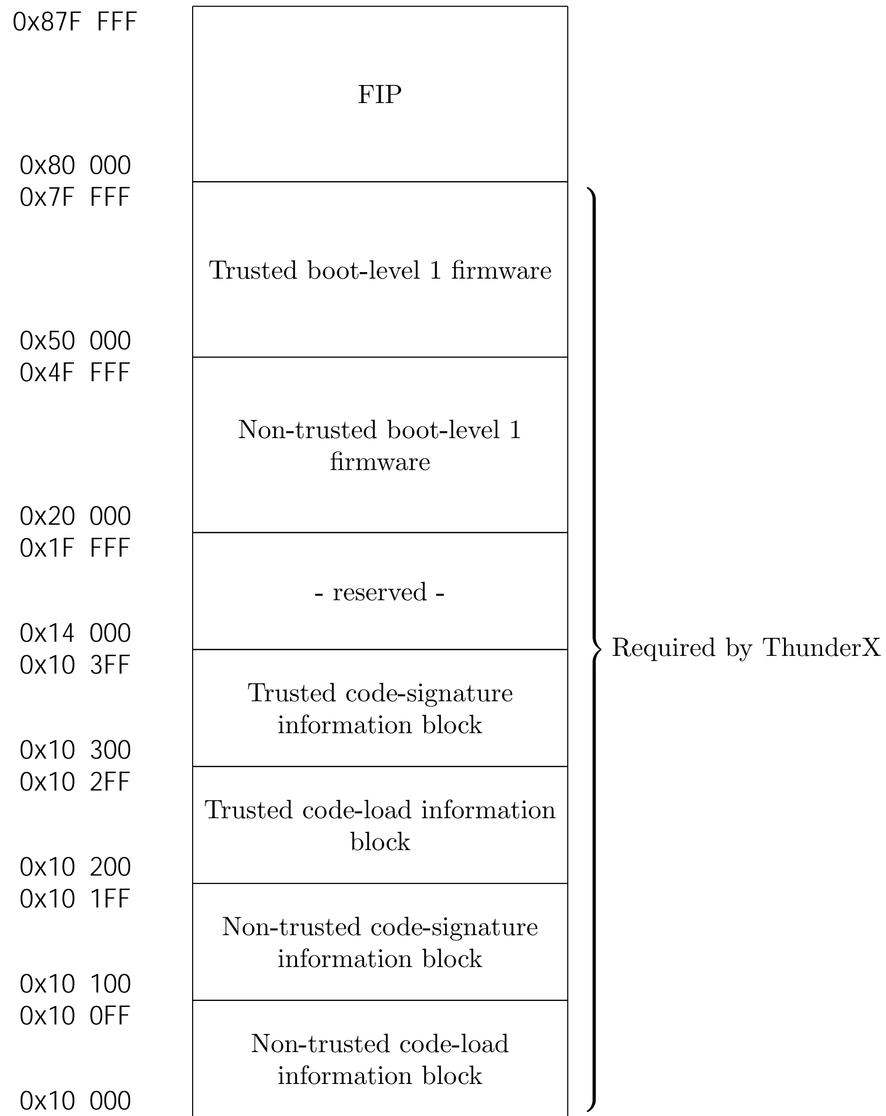


Figure 6: Memory layout of the Flash memory

be found in `thunderx_l2c.c`[5]. These can be used as a template to implement all the other CvmCACHE instructions for the instruction, L1 and L2 cache of the ThunderX as described in §2.12 of the ThunderX manual.

[5]https://gitlab.inf.ethz.ch/PROJECT-Enzian/arm-trusted-firmware-enzian-port/-/blob/v2.9-enzian/drivers/cavium/thunderx/thunderx_l2c.c

3.5 Shared behavior of BL stages

All of the bootloader stages of the ATF, except for BL33, have some behaviour and initialization in common. Every stage starts with the MMU in bypass mode, where physical and virtual address are the same. The first thing that has to be initialized when running a new stage is the console. Once that is done the next step is setting up the MMU for the exception level at which the bootloader stage is running. In practice this is EL3 for BL1 and BL31 and EL1S for BL2.

The console initialization is done in the function `${BLstage}_early_platform_setup` and the MMU is set up in `${BLstage}_plat_arch_setup`, where `BLstage` is either `bl1`, `bl2` or `bl31`.

3.6 UART and console initialization

According to the ATF documentation, console initialization takes precedence in each stage of ATF execution. Initializing the console involves setting up the UART, the physical device, and registering it as a console using ATF's console framework.

The ThunderX platform comprises two UARTs: UART0 is used as the normal console to interact with the ThunderX and UART1 is used for the EFRI link. UART0 is initialized in all ATF stages, while UART1 is initialized only in BL1 and BL31 (see Section 5). Both UARTs operate at a baud rate of 115200. The standardized initialization of ThunderX's UART is possible due to its compatibility with the PL011 UART [ARM Limited, 2007].

Every console has a scope, which can be used to restrict its use during different phases of the ATF. As part of the scope every console can either be enabled or disabled for the following three phases:

- BOOT: The console output is active during the boot-up phase of the firmware. This includes early initialization, bootloader execution, and other early boot processes.
- RUNTIME: The console output is active during the runtime phase of the firmware. This phase involves the execution of the Trusted OS or the secure monitor.
- CRASH: The console is used by the panic handler of the ATF when an error is encountered.

In this case, enabling all scopes for UART0 console is appropriate since it's used throughout ATF execution.

The process involves initializing the UART and registering it using the `console_pl011_register` function. Further configuration includes specifying the console's scope for boot, runtime, and crash phases using `console_set_scope`. Given the UART's PL011 compatibility, initialization mainly entails proper divider and reference clock configuration.

3.7 Timer initialization (GTI)

The Global System Timers Unit (GTI)[6] includes the system counter and different watchdog timers. The watchdog timers have not been initialized in the new ATF. The system counter is used as the global sense of time for the entirety of the system. It can be read by all privilege levels, meaning that user space applications can accessed it by reading the `CNTPCT_EL0` register. The 64-bit register holding the count value of the system counter is referred to as `GTI_CC_CNTCV` in the ThunderX manual. The system counter works as follows: Every clock cycle the value of `GTI_CC_CNTRATE` is added to `GTI_CC_CNTRACC`. When `GTI_CC_CNTRACC` overflows `GTI_CC_CNTCV` is incremented by one. We are using the coprocessor clock `COPROC_CLK`, running at 100MHz, as the reference clock for the GTI. Given that we want to count in microseconds we need to overflow `GTI_CC_CNTRACC` once every $1\mu s$ or, equivalently, at a frequency of 1MHz. This frequency is referred to as the `GTI_RATE`. Given that the reference clock is running at 100MHz, we have to overflow once every 100 clock cycles. As the `GTI_CC_CNTRACC` register is 32-bit wide we need to increment it by:

$$\text{increment} = 2^{32} * \frac{\texttt{GTI_RATE}}{\texttt{COPROC_CLK}} = 2^{32} * \frac{1\text{MHz}}{100\text{MHz}} \tag{1}$$

To use the coprocessor clock we set `GTI_CC_IMP_CTL[CLK_SRC]` to 0. The increment value calculated above has to be set in `GTI_CC_CNTRATE`. Then both `GTI_CC_CNTFID0` and GTI_CTL_CNTFRQ are set to `GTI_RATE` before enabling the system counter by setting `GTI_CC_CNTCR[EN]` to 1.

3.8 Serial Presence Detect (SPD)

Serial Presence Detect (SPD) is a small EEPROM chip integrated into DRAM modules. It contains vital information about the memory module, such as its capacity, speed, timing parameters, manufacturer details, and other relevant data. This can be used by the firmware to automatically initialize the memory without the need to specify the different settings.

Within the Enzian system, SPDs are interconnected through the Two-Wire Serial Interface (TWSI), akin to the I²C bus. Notably, the bus connections differ between the Enzian (bus 0) and ThunderX development board, called EBB (bus 1). By toggling the `BUILD_FOR_EBB` flag in `platform_def.h`[7], the ARM Trusted Firmware (ATF) can be built for either platform.

Preceding the DRAM initialization sequence, BL1 undertakes SPD EEPROM probing for each DIMM slot. Upon detecting a DRAM module and successfully reading its EEPROM, the memory's contents are parsed. An all-populated 4 DIMM slot configuration is a requirement for the new ATF's operation, thus leading to a panic if this criterion isn't met. Parsed SPD data yields essential information including DIMM type (buffered/unbuffered, ECC

[6] ThunderX manual §20.3

[7] https://gitlab.inf.ethz.ch/PROJECT-Enzian/arm-trusted-firmware-enzian-port/-/blob/v2.9-enzian/plat/enzian/include/platform_def.h

support), capacity, bank/column/row sizes, timing parameters, and maximum supported speed grade.

These extracted values, elaborated further in Section 4, play a pivotal role in configuring the memory controller of the ThunderX platform.

3.9 Memory Layout of Enzian

The ThunderX uses a 49-bit virtual address space, as dictated by ARMv8, which gets mapped to a 48-bit physical address space. The physical address space is split into an IO space (<47>=1) and DRAM space (<47>=0). For the DRAM space the bits <46: 42> must be 0. The bits <41: 40> indicate the node whose DRAM will be addressed. The remaining bits <39: 0> contain the offset into the memory of said node. The IO space requires bit <46>=0 and the following two bits <45: 44> indicate the node. The remaining bits <43: 0> address a specific CSR of a device.

3.9.1 Partition into Secure and Non-Secure DRAM

The ThunderX distinguishes between secure and non-secure DRAM regions. There are a maximum of 4 so-called ASC regions[8] that can be marked as secure or non-secure. The regions have a granularity of 1 MiB. If code running in a non-secure exception level tries to access a secure region a fault is generated. The boot ROM initially only marks the region from 0x100_000 to 0x1FF_FFF as secure. Trying to access any memory region outside will result in a system exception. This necessitates splitting the entire DRAM address space into different regions and marking them as either secure or non-secure. As shown in Figure 7 the fist 2 MiB of memory are marked as secure. This is the memory region where the BL1, BL2 and BL31 live. From 0x200_000 on-wards it is set as non-secure.

For each region there exist 3 registers, where `L2C_ASC_REGION_START[ADDR]` sets the starting address of the region in MiB, `L2C_ASC_REGION_END` sets the exclusive (!) end address of the region in MiB and `L2C_ASC_REGION_ATTR` is used to specify if the region is secure or non-secure. As the end address is exclusive mapping the memory region from 0x100_000 to 0x1FF_FFF results in `L2C_ASC_REGION_START[ADDR]` = 1 and `L2C_ASC_REGION_END[START]` = 1.

By partitioning the DRAM into secure and non-secure regions we can ensure that only the firmware has access to these memory regions.

3.9.2 MMU - Memory Management Unit

The ThunderX's MMU is compatible with the reference design from ARM. In each stage of the ATF execution starts with the MMU in bypass mode, where the virtual address and physical address are equal.

Every stage of the ATF maps its RO- and RW-section and the devices in IO-space that are needed for system boot. These include the CSRs for the SPI

[8] ThunderX manual §4.4.2

interface (MPI), the coprocessor (RST), the generic interrupt controller (GIC), the L2 cache controller (L2C) and the console (UAA). Additionally, BL1 needs to map the memory region to where it copies BL2 and BL2 needs to map the region for BL31. For simplicity's sake we map the whole secure DRAM region. To copy BL33 from the FIP BL2 needs to also map the UEFI memory region.

The old ATF implementation allocated the different memory regions associated with each subsystem starting from a specific address: The first IO device was mapped at `IO_VA_BASE` and the others would be allocated sequentially. When trying the same approach with the newest release of the ATF this resulted in crashes during the configuration of the MMU and the subsequent switch to using virtual memory. The current ATF release uses the console to print information about the memory map during the switch from physical memory to virtual memory. This would crash the system as the console was still registered to the old physical address. An attempt at fixing the problem consisted of disabling the console before the switch and re-enable it afterwards. Although this worked fine it introduced unnecessary complexity. To completely solve the problem the decision was made to identity map IO device's memory. Therefore, there is no more need to disable the console or keep track of the mapping between virtual and physical IO addresses.

The mappings are performed in the `tx_configure_mmu_el3` function for BL1 and BL31 and in `tx_configure_mmu_el1` for BL2.

To perform the mapping the XLAT library from the ATF is used. This is done with the `tx_configure_mmu_el` function. It takes as arguments the start and end of the RO- and RW-sections of the current BL. The start of the RO section is the `BL_CODE_BASE` label, the end of the RO-section and start of RW-section is `BL_CODE_END` and the end of the RW-section is `BL_END`. It then adds the RO- and RW-sections and the necessary IO-regions to the translation tables. It then initializes the translation tables and enables the MMU at either EL3 for BL1 and BL31 or EL1 for BL2.

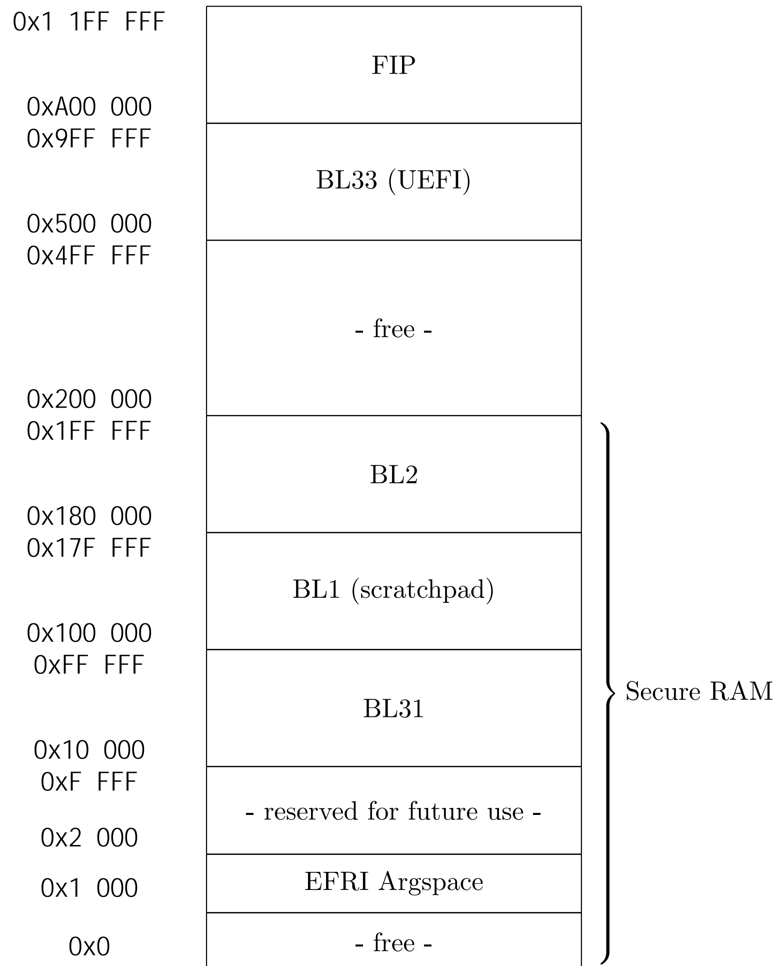


Figure 7: Memory layout of the firmware in main memory

3.9.3 Design decision regarding the memory layout

The memory layout deliberately starts with an unallocated first page, beginning at address 0x0. This approach ensures that there can't be a pointer with value 0x0 that actually is a valid pointer and not a null pointer. BL31 will map a physical page from user space to the address of the second page when using EFRI (see Section 5). In order to not have a virtual address referring both to the identity mapped physical page and the page from user space this page is not mapped. The following six pages, located before the memory region designated for BL31, are reserved for future use. One possible use for these pages is a mailbox for communication between the various cores or as a storage location for the state of the different cores. The latter needs to be implemented if the functionality of shutting down specific core for energy saving is desired (see

Section 7).

3.9.4 Further space optimizations

As the Enzian project does not use the Trustzone functionality there is no need to set up the MMU for EL1S, which is done in BL2. To reduce the size of the firmware and the code base the `RESET_TO_BL2` flag can be enabled. This flag removes BL1, runs BL2 at EL3 and sets the entrypoint of the ATF to the BL2 stage. This effectively merging the functionality of BL1 and BL2. As Enzian operates as a research machine, potential scenarios demanding the utilization of TrustZone could emerge in forthcoming research endeavors. Thus the decision has been made to not use `RESET_TO_BL2` in order to maintain flexibility.

3.10 Loading next stages from the SPI flash

Upon initializing DRAM and configuring the MMU, BL1 proceeds to hand over control to BL2. To do so the BL1 retrieves the Firmware Image Package (FIP) from the SPI flash. The driver for the SPI flash has been ported from `thunder_spi.c`[9] of the old ATF. To facilitate the subsequent steps, BL1 utilizes the storage abstraction layer provided by the ATF. This abstraction layer simplifies interactions with various storage devices while encapsulating the intricate details.

With the storage abstraction layer, BL1 loads BL2 into memory to which it hands over execution. BL2, having taken control, assumes the responsibility of loading BL31 and BL33 into memory.

3.10.1 Firmware Image Package (FIP)

The Firmware Image Package (FIP) is divided into two main sections: a header and a data segment. The header section contains entries that define specific regions within the data section. Each entry is associated with a unique identifier, which indicates the corresponding bootloader stage stored in that region. This design allows the storage abstraction layer to efficiently load the require binaries. In order to fully leverage the FIP and reduce complexity the choice has been made to bundle the BL33 (UEFI) as part of the FIP. This makes it impossible to flash the ATF or the UEFI separately in the new firmware stack. Entries can be added or removed from the FIP using the `fiptool`. The tool is built with `make fiptool`.

3.10.2 Storage abstraction layer setup

The setup of the storage abstraction layer involves the initialization of two IO devices. These devices, called `memmap_dev` and `fip_dev` serve distinct purposes:

[9] https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-atf/-/blob/master/plat/thunder/thunder_spi.c

the former accesses arbitrary memory regions, whilst the latter reads sections of the FIP. This arrangement facilitates efficient handling of firmware components.

During the process of loading a stage from the FIP, memmap_dev is initially employed by accessing the memory region containing the entire FIP. Subsequently, fip_dev uses memmap_dev to read the contents of the FIP. By leveraging the information present in the FIP's header and it can then access the precise data segment within the FIP corresponding to the bootloader stage to be loaded. This mechanism ensures accurate and targeted data retrieval.

Notably, the majority of the loading process can be succinctly specified in a declarative manner (see thunderx_io.c[10]). This is achieved by providing an array of plat_io_policy structures. These structures outline how each stage should be loaded and which IO device is to be employed.

In the context of the BL2, an additional descriptor, found in bl2_plat_mem_params_desc.c[11], is utilized. This descriptor assists BL2 in determining the appropriate order in which to load and execute images and the manner in which they should be executed, including considerations like the exception level.

3.11 Generic Interrupt Controller (GIC)

The GIC serves as a centralized and flexible interrupt management unit. It allows multiple processor cores to share and distribute interrupts effectively, reducing the complexity of interrupt handling across the system. Some of the key features of the GIC include:

- Interrupt Routing: The GIC manages the routing of interrupts to the appropriate processor core, ensuring that each core receives and processes the relevant interrupts.
- Interrupt Prioritization: Interrupts are prioritized by the GIC based on their urgency and importance. Higher-priority interrupts are serviced before lower-priority ones, helping to maintain system responsiveness.

BL31 includes GIC initialization and configuration routines to ensure proper setup and operation of the GIC before handing control over to BL33. The development of this implementation was significantly influenced by reverse engineering other platforms and the legacy ATF.

The initialization and configuration within BL31 is particularly streamlined due to the compliance of the ThunderX's GIC with Arm's V3 GIC specification. This conformance allows for a straightforward approach, wherein the intricacies of the initialization process are handled by the gicv3 library provided by the ATF. The implementation of the new ATF can be found in enzian_gic.c[12].

[10] https://gitlab.inf.ethz.ch/PROJECT-Enzian/arm-trusted-firmware-enzian-port/-/blob/v2.9-enzian/drivers/cavium/thunderx/thunderx_io.c

[11] https://gitlab.inf.ethz.ch/PROJECT-Enzian/arm-trusted-firmware-enzian-port/-/blob/v2.9-enzian/plat/enzian/bl2_plat_mem_params_desc.c

[12] https://gitlab.inf.ethz.ch/PROJECT-Enzian/arm-trusted-firmware-enzian-port/-/blob/v2.9-enzian/plat/enzian/enzian_gic.c

3.12 Power State Coordination Interface (PSCI)

PSCI is the standardized interface that enables communication and coordination of power management operations in a multi-core ARM system. The interface is a set of standardized functions and calls to control various power states, such as sleep, standby or power-off. The control revolves around power domains, which represent a processing element like a core. These are referred to as power domain level 0. A logical grouping of different cores (level 0), called a cluster, is a level 1 power domain. Following this principle a CPU being a group of clusters, is a level 2 power domain. The ThunderX has 8 clusters with 6 cores each.

The Power State Coordination Interface (PSCI) serves as a standardized mechanism for facilitating communication and synchronization of power management operations within a multi-core ARM-based system.

This interface encompasses a predefined set of functions and calls that enable the regulated control of diverse power states, such as sleep, standby, or power-off. These operations are intrinsically linked to power domains, which signify individual processing entities, primarily cores, and are designated as power domain level 0. Operating according to this structure, clusters, which group together multiple cores at level 0, constitute level 1 power domains. Further, in the hierarchy, the CPU, comprising several clusters, is classified as a level 2 power domain. To provide an example, the ThunderX architecture incorporates 8 clusters, each housing 6 cores.

Various actions can be applied to these power domains, ranging from placing cores within a cluster in standby mode to shutting down the entire ThunderX processor. Notably, the ThunderX exhibits a unique characteristic concerning power control. It lacks the capability to directly manage power regulators, a role instead assumed by the BMC. Consequently, when dealing with the shutdown command, the EFRI link is used to send the shutdown signal to the BMC.

The implementation of PSCI for the Enzian platform can be found in `enzian_topology.c`[13]. Although the function responsible for implementing the system shutdown has been successfully developed, it's important to note that the other functions pertaining to the PSCI have either not been tested or not completed. These implementations are left as future work.

[13] https://gitlab.inf.ethz.ch/PROJECT-Enzian/arm-trusted-firmware-enzian-port/-/blob/v2.9-enzian/plat/enzian/enzian_topology.c

4 Memory controller and DRAM initialization

The focus of this thesis lies in the initialization of the DRAM and its associated memory controllers on the ThunderX platform. The ThunderX manual employs the term "LMC" to denote the DRAM memory controller. This section delves into the technical aspects of the LMC's features, the sequence of initialization steps, and the challenges encountered during this process.

4.1 Overview

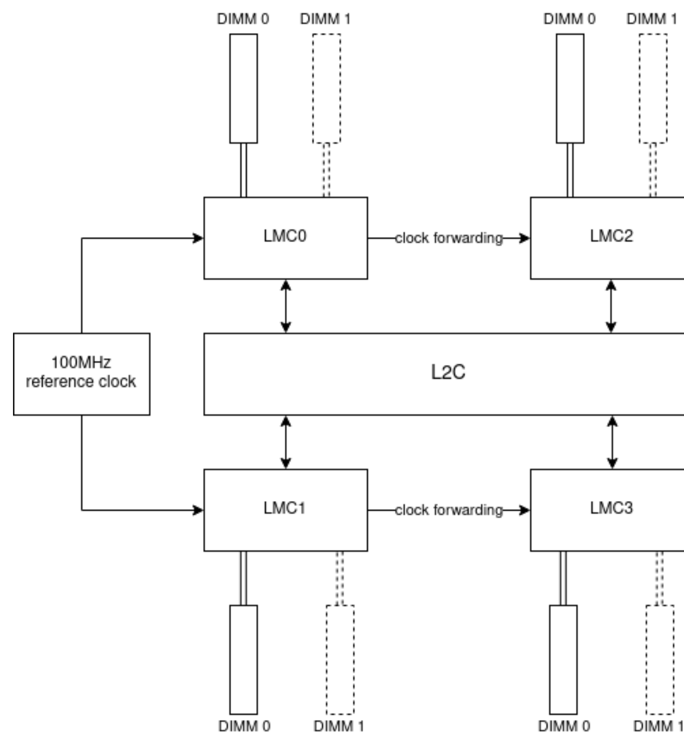


Figure 8: Topology of the LMCs in the ThunderX

The ThunderX employs four LMCs, which function as memory controllers compatible with both DDR3 and DDR4 memory. They are used to manage the complex behaviour needed to make modern DRAM work by performing the periodic DRAM refresh or the calibration steps(discussed in Section 2.6).

The ThunderX operates in two modes: the four-LMC and two-LMC modes, allowing for quad-channel and dual-channel memory transfers, respectively. The focus of the ATF port is on the four-LMC mode, which offers the highest memory throughput. The system assumes all four DIMM slots are occupied in the Enzian

setup. Additionally, due to DDR4 DIMM slots in the Enzian platform and their incompatibility with DDR3 DIMMs, support for DDR3 memory is not included.

The memory controller supports DDR4 speed grades of 1600 MT/s, 1866 MT/s, and 2133 MT/s. For the latter, configuring the LMCs' reference clock to 100MHz (instead of 50MHz) is necessary. The new ATF exclusively supports the 100MHz reference clock, enabling memory operation at all supported speeds. Both Registered DIMMs (RDIMMs) and Unbuffered DIMMs (UDIMMs) are supported.

4.2 Topology of the LMCs

When running in four-LMC mode LMC0 and LMC1 receive the 100MHz reference clock. Using a Phase-Locked Loop (PLL) this clock signal is turned into a clock frequency that can be divided down to the desired frequency at which the DRAM will run. The PLL is a control system that generates an output signal with a frequency and phase that are locked to the reference clock signal. LMC0 and LMC1 then pass on these clock signals to LMC2 and LMC3, as shown in Figure 8. Consequently, certain initialization steps are performed exclusively on LMC0 and LMC1, as discussed in Section 4.3.1.

4.3 LMC Initialization Sequence

The initialization of the LMCs involves a 15-step sequence outlined in the ThunderX manual[14]. The first 6 steps consist in enabling the 100Mhz clock, setting the desired multipliers and divisors to achieve the target speed of the memory and perform the reset procedure.

Subsequently, various timing parameters for DDR4 memory are calculated and configured in accordance to the JEDEC specification for DDR4 memory. Only for RDIMMs we have to initialize the MPRs with our desired testing pattern for the read-leveling procedure that will be performed afterwards. The subsequent three steps initiate training, including Offset, internal Vref, and Deskew training. These steps appear to address read and write centering and input voltage calibration. This is only an educated guess as the manual uses non-standard terminology. Note that this training process is not the same as the V_{ref} training for DDR4 described in Section 2.6.9. In the new ATF implementation, these training procedures occur only once as opposed to the training steps below.

The following three steps are more complex, involving write-leveling, read-leveling, and DDR4 V_{ref} training. Detailed discussions of these steps are provided in Section 4.7, 4.8 and 4.9 respectively. Since achieving successful DRAM initialization requires iterative execution of these procedures, the algorithms explored for this purpose are discussed in Section 4.10. The final step involves the clearance of ECC (Error-Correcting Code) errors, where applicable.

For a comprehensive understanding, subsequent subsections delve into the most significant steps of the sequence. The ThunderX manual contains further detailed information.

[14] ThunderX manual §7.9

4.3.1 Steps 1 through 6

The first steps of the initialization consist in selecting the four-LMC mode and setting up the 100MHz reference clock. This will then be used to initialize the PLLs of only LMC0 and LMC1 by setting their multiplier.In a later step they will be configured such that the generated clock cycle is forwarded to LMC2 and LMC3.

Step 3 consists of setting the divisor for the generated clock signal in order to get a clock signal that matches the desired DRAM speed grade. Note that all the LMCs have to run at the same frequency. The manual mandates this step only to be performed on LMC0 and LMC1. This results in LMC2 and LMC3 not working correctly, which was validated using the knowledge from Section 4.6. Furthermore by checking the implementation from the BDK we found that step 3 is performed on every LMC.

This step is then followed by resetting the LMCs in a specific order and enabling periodic calibration of the forwarded clock signal. Although supported by the ThunderX the ability to perform the initialization sequence whilst preserving the contents in memory and the ability to change the speed grade of the DRAM during execution was not implemented. Both these features were deemed either too difficult to implement correctly in the given time frame and/or non-mission-critical for the typical use case of an Enzian system: No particular use case has been found for which a sleep feature (powering down the computer except for DRAM, where the current execution state is stored) would be beneficial.

4.4 Step 7: Early LMC Initialization

DDR4 modules require the upholding of precise timing constraints to function properly. These constraints are categorized as primary and secondary timings. Primary timings are DRAM-module-specific and are derived from the SPD (see Section 3.8) information. Secondary timing parameters are only tied to memory speed grades. The process of deriving these timings correctly involved cross-referencing the JEDEC DDR4 specification, reverse engineering the BDK and validating the computed timings using memory dumps from the relevant Control and Status Registers (CSRs) in the system.

Step 7 encompasses more than just timing-related adjustments; it also involves configuring other essential settings. This includes specifying the sizes of rows, columns, and banks in the DRAM module, as depicted in Figure 9. Additionally, this step entails determining the ranks present in the DRAM modules, if the DIMMs are registered or not, and whether they support Error-Correct Code (ECC).

4.5 Steps 9 through 11

The 9th step consists of initializing the MPR registers with the training patterns that will be used in later stages of the initialization procedure. The subsequent three steps begin the training procedure. Note that they only have to be execute

once and do not require any iteration, unlike the write- and read-leveling or the V_{ref} training. The ThunderX manual refers to these training procedures with non-standard names so we can only speculate that they correspond to the write and read centering procedure described in Section 2.6.6. All three of these training procedures is performed automatically by the memory controller.

4.6 Testing the LMCs

Testing the LMCs is not a step of the initialization sequence in itself but it still needs to be performed in conjunction with the write- and read-leveling and the V_{ref} training procedure. In these steps we need to write and read a test pattern to memory and check if it gets written and read back correctly. As main memory is interleaved between the different LMCs we need to know how the LMC selection algorithm of the ThunderX works.

4.6.1 LMC and bank selection algorithm

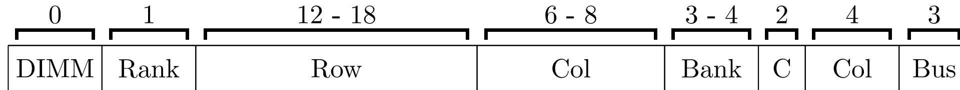


Figure 9: LMC internal addressing of DRAM

The LMC and bank selection algorithm changes based on the L2C_CTL[DISIDXALIAS] and LMC(0..3)_CONTROL[XOR_BANK] registers. These influence where a given physical address gets placed in the DRAM stick and in which one it is placed. Marvell advises to use L2C_CTL[DISIDXALIAS]=0 and LMC(0..3)_CONTROL[XOR_BANK]=1 for best performance on most workloads [15]. (Changes in these settings will break the current memory testing code.) With these settings in mind and given an address ADDR the LMC is selected using the following formula:

$$\mathtt{LMC} = \mathtt{ADDR<8:7>} \oplus \mathtt{ADDR<21:20>} \oplus \mathtt{ADDR<13:12>} \quad (2)$$

For the 3 least significant bits of the bank the following formula is used:

$$\mathtt{BANK<2:0>} = \mathtt{ADDR<11:9>} \oplus \mathtt{ADDR<16:14>} \quad (3)$$

4.6.2 Testing methodology

To perform the write-read testing needed for the training steps the pattern in Figure 10 is used. Two approaches were used to test a specific memory region:

- Cache line scrubs: The ThunderX manual recommends performing cache line scrubs with a given a pattern for the V_{ref} training. This consists in

[15]ThunderX manual §7.2

```
0xFDFD_FDFD_FDFD_FDFD
0x8787_8787_8787_8787
0xFEFE_FEFE_FEFE_FEFE
0xC3C3_C3C3_C3C3_C3C3
0x7F7F_7F7F_7F7F_7F7F
0xE1E1_E1E1_E1E1_E1E1
0xBFBF_BFBF_BFBF_BFBF
0xF0F0_F0F0_F0F0_F0F0
0xDFDF_DFDF_DFDF_DFDF
0x7878_7878_7878_7878
0xEFEF_EFEF_EFEF_EFEF
0x3C3C_3C3C_3C3C_3C3C
0xF7F7_F7F7_F7F7_F7F7
0x1E1E_1E1E_1E1E_1E1E
0xFBFB_FBFB_FBFB_FBFB
0x0F0F_0F0F_0F0F_0F0F
```

Figure 10: Testing patterns used for the various training stages

writing a cache line with the first line of the pattern, flushing the cache line to main memory and reading it back. Then the read value can be checked for bit errors and the same cache line is overwritten with the next line of the pattern. Intuitively this can be used to identify bits that are stuck either high or low because of a wrongly configured reference voltage.

- Page table writes: The second and more naive approach that was tried consists of writing several 4 KiB pages with the pattern, flushing them and reading them back to compare them with the reference pattern.

In order to test a specific rank of an LMC pointers are needed to locations in memory that are mapped to that specific LMC and rank. The pointers are constructed by setting the Rank bit from Figure 9 to the appropriate rank and adding an offset of 0x200_000. This offset ensures that when testing rank 0 of a DRAM module we don't accidentally overwrite the ATF code in the scratchpad. As a last step we check whether or not the pointers would be mapped to the desired LMC using Formula 2.

4.7 Step 12: Write-Leveling

Write-leveling is the process that ensures uniformity in the timing of write operations across memory chips of a DRAM module, mitigating potential issues arising from variations in chip response times. The objective is to achieve balanced signal propagation delays and data capture times in order to maintain data integrity and optimize memory access speeds.

The ThunderX's memory controller introduces a nearly automated approach to perform write leveling, streamlining the optimization process. This controller facilitates the adjustment of delay settings for each memory chip, allowing the

delay to be configured in increments of 1/8 of a clock cycle. The configuration range spans from 0 to 4 cycles (exclusive).

The automated training sequence from the LMC only adjusts the fractional part of the per-chip delay. This has to be complemented by determining the decimal part (0, 1, 2 or 3) through software-based methods. This gives rise to a search space of considerable magnitude, approximately $2^{18} \approx 250000$ settings for each rank (or around 70000 with non-ECC memory), significantly escalating the complexity of the optimization task. Given the current system configuration of Enzian, comprising 4 dual-rank DIMMs, this translates to a staggering 2 million potential settings solely for the write-leveling aspect.

However, a key insight aids in significantly mitigating this complexity. Let us consider two memory chips, X and Y, where it is known that the delay of chip Y is greater or equal to the delay of chip X. This could arise from the physical layout of the two chips on the memory module. For instance, in both RDIMMs and UDIMMs, the delay of chip 7 should be greater than or equal to the delay of chip 6. In this case if the fractional part of chip X exceeds that of chip Y, the decimal part of chip Y must strictly surpass that of chip X. This observation drastically reduces the practical search space to an approximate worst-case size of 2000 settings. Consequently, exhaustive exploration of this refined search space becomes feasible, allowing for the selection of the optimal setting without errors, thereby striking a balance between performance and reliability.

4.8 Step 13: Read-Leveling

Similarly to write-leveling, read-leveling is the process that ensure the alignment of read operations across memory chips relative to the CLK. For this procedure the ThunderX has a completely automated approach: for every chip, the LMC performs 64 read operations of an internal pattern stored in a MPR. It then takes the longest run of successful settings and picks the middle value. This is then set automatically in the `LMC_RLEVEL_RANK` register for every chip. As the testing pattern is stored in a MPR inside the DRAM the read-leveling settings can be determined without the need for a correct write-leveling setting. From here on we will be referring to the combined procedure of write- and read-leveling only as leveling.

4.9 Step 14: V_{ref} Training

For the V_{ref} training the ThunderX manual recommends performing repeated cache line scrubs, as described in Section 4.6.2. When reading back the values we compare the amount of false-1s (bits that should be 0 but are interpreted as a 1) and false-0s. If there are more false-1s than false-0s we reduce the threshold voltage and vice versa. To set the reference voltage a write to an MPR register is used. In the case that there are still errors and the amount of false-1s and false-0s is roughly the same it probably means that something else in the training process, like write- or read-leveling, is wrong.

To calculate the initial voltage from which to start the training procedure we use the formula for the case that only one DIMM slot is populated with either a one or two rank DIMM:

$$\text{Initial } V_{ref} = \frac{1 + \frac{1 + \text{DQX_CTL}}{15 + \text{DQX_CTL} + \text{RTT_WR} \parallel \text{RTT_PARK}}}{2} \tag{4}$$

where Initial V_{ref} is the percentage of V_{dd} and $\parallel$ denotes the parallel resistance (see ThunderX manual §7.9.14).

4.10 Training Algorithms

Knowing how to perform the write-, read-leveling and V_{ref} training steps should make the implementation of the whole training procedure straightforward. But a conundrum emerges: How can one perform write-leveling without reliable reads, due to a still to be determined reference voltage? Similarly how can we determine the reference voltage if we don't have reliable writes, due to still to be determined leveling settings? Several different approaches have been explored:

4.10.1 1st Algorithm

The ThunderX manual clearly states that if the write- and read-leveling settings are known in advance it is possible to just write them to the appropriate register. This approach was tried first. By starting the boot process of the ThunderX using the old firmware stack the appropriate registers can be dumped: `LMC_WLEVEL_RANK` for the write-leveling of one rank, `LMC_RLEVEL_RANK` for the read-leveling of one rank. For the V_{ref} setting the value would be printed before it was being set in the DRAM via a `MPR` write. These values where then hard coded into the new ATF.

This approach worked fine and allowed further development of the ATF port. This solution is, of course, less than ideal due to the need of having to first extract all the leveling settings from each machine before hard coding these values into the ATF or having the BMC supply them. Changing the speed grade of the DRAM would also invalidate the settings. Ideally we would want the process of DRAM initialization to be automatic and as hassle-free as possible for the end-user of the Enzian systems.

4.10.2 2nd Algorithm

The next approach was to loop over the leveling and V_{ref} training sequence. All leveling settings would be tried and the one with the least amount of errors would be picked. Next would be the voltage training. Depending on the amount of false 0s or 1s during the testing the reference voltage would be either increased or decreased, trying to find the sweet spot resulting in no errors. A false 0 refers to a bit being read as a 0, when the testing pattern has a 1 in that bit and vice versa for the false 1. The problem of finding these suitable settings can

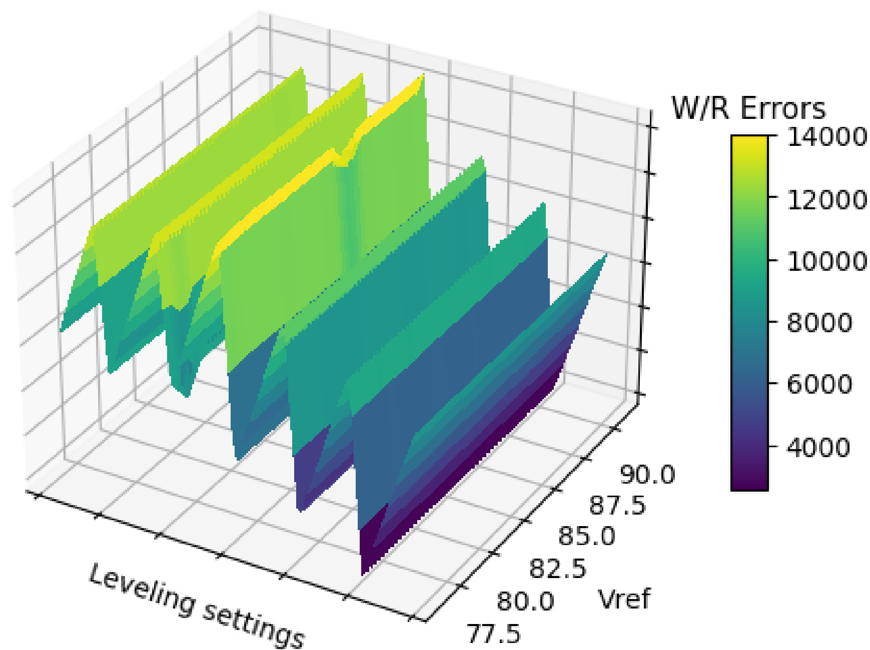


Figure 11: Partial visualization of W/R errors as function of leveling and V_{ref} settings. The dark blue region represents the global minimum.

be modeled as a hyperplane, where one axis is for the V_{ref} voltage, one for the leveling setting and one for the amount of errors encountered during testing with that particular settings combination. A section of such a hyperplane can be found in Figure 11. To find the global minimum of such hyperplane the following algorithm was devised:

1. Set the voltage to the initial value calculated with the formula from Equation 4.
2. Find leveling setting for current voltage level with least amount of errors. This is done by trying all leveling settings that are possible if taking into consideration the heuristic described in Section 4.7.
3. Depending on false 0s and 1s either increase or decrease the reference voltage by 1%.
4. Repeat from 1. until no errors are encountered.

This algorithm has a major flaw. Namely it first finds the best leveling setting, as it tries them all at each iteration and then adjusts the V_{ref} value

in 1% steps. This often resulted in a setting that would pass all the test but would then not work after the scratchpad had been unlocked and the system was running from DRAM. This phenomenon can be explained by Figure 12: The observed behaviour of the amount of bit-errors when plotted against the V_{ref} settings is, that, in the lowest range, there are a lot of errors. Then there is a region where some settings result in no errors but some other settings result in errors. Only then a sizeable range of settings can be found were no errors are encountered, referred to as the "stable range".

With this knowledge one can speculate that the algorithm finds one of these sub optimal settings in the second region. This setting can work fine during a test with a small sample size but fail later in the execution of the ATF. To fix this problem we propose the next algorithm.

4.10.3 3rd Algorithm

This algorithm works exactly the same as the one before but adds a 5th step to find the stable range of V_{ref} settings. This last step works as follows:

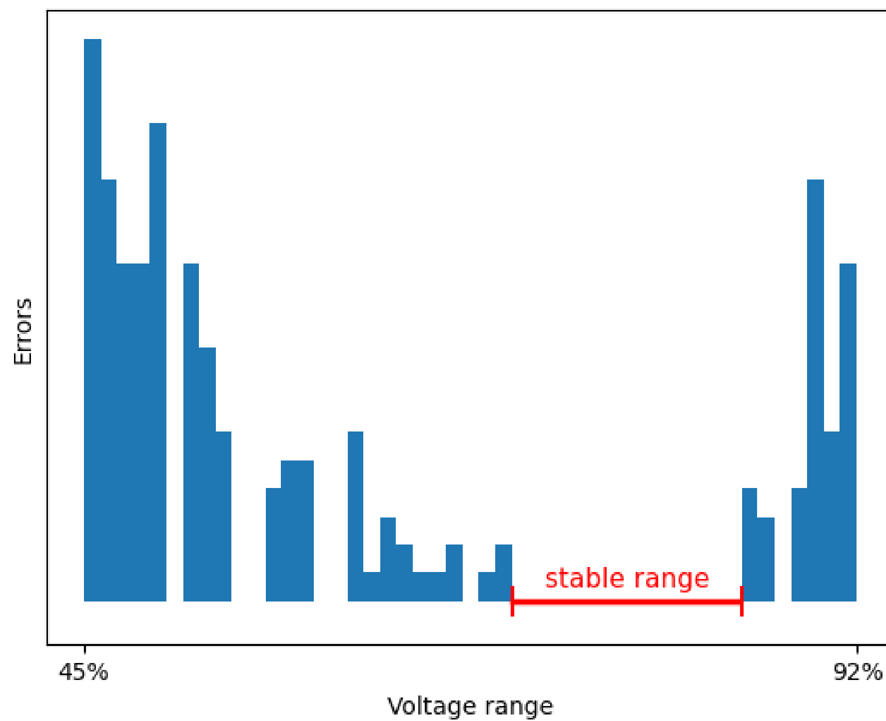
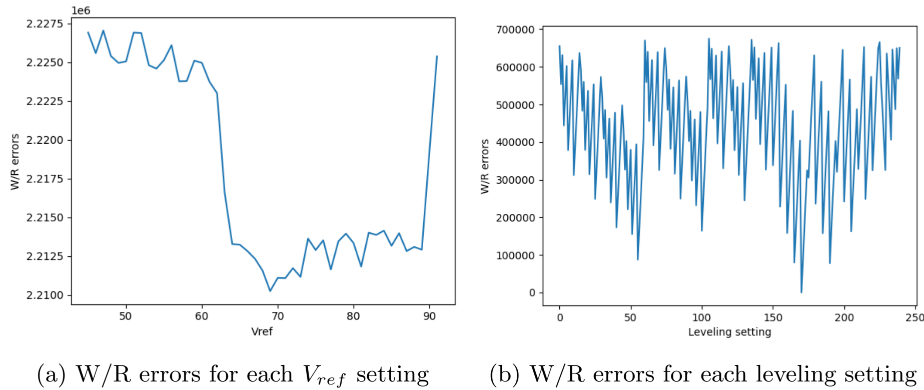


Figure 12: Write/Read errors when performing a voltage sweep with Algorithm 3.

Having a setting that resulted in the test passing with no errors we perform



(a) W/R errors for each V_{ref} setting

(b) W/R errors for each leveling setting

Figure 13: 2D plots of Figure 11

a memory test for every reference voltage value. Then the longest sequence of passing tests is selected and the median value is picked as the final V_{ref} value.

When running this improved algorithm on some ranks there is some unexpected behaviour when trying to run at 1600 MT/s or 1866 MT/s: When performing the last step of finding the reference voltage in the stable range it turns out that the amount of errors doesn't look anything like Figure 12. Instead every voltage setting from 45% to 92% results in no errors. This results in 68% being picked as the final V_{ref} setting. This still works but it does not really inspire confidence in the approach being chosen. Furthermore the algorithm has some other issues:

- It doesn't always terminate. When running at 2133 MT/s this is almost always the case where as at the two lower speed grades this happens very rarely. This could be due to the fact that leveling settings were picked that appear to be the global minimum of Figure 13b but are instead a local minimum. A fix for this problem could be to only allow a certain amount of retries for a specific leveling setting to reach the global minimum. If the amount of retries is exceeded the leveling setting is discarded and the second best is picked and the process repeated.

- The following issue only manifested on zuestoll09 when running at 2133 MT/s For every DIMM and every rank the reference voltage was very close to the one from the BDK (±1%) and the leveling where the same except for the delay of chip 8. Chip 8 is responsible for ECC. This would result in all the test running fine. Then, after finishing the initialization, when ECC is enabled the system would panic. We suspect this is due to the fact that we can't test the ECC module by simply writing and reading test patterns.

Despite encountering some challenges, particularly with the highest speed grade, the algorithm demonstrates functionality. Notably, the algorithm has facilitated successful runs of the new UEFI implementation on multiple occasions.

4.10.4 4th Algorithm

The last attempt was to try all combinations of leveling settings for every V_{ref}. The data generated by doing this was used to plot Figure 13 and 11. The best leveling setting would be picked. Then, similar to the third algorithm, the longest sequence of working reference voltages was found and the middle value picked. Conversely enough this approach doesn't even work for the two lower speed grades of 1600 MT/s and 1866 MT/s, probably due to an error in the implementation.

4.10.5 Takeaways and future work

Being short on time we couldn't further address the challenges of DRAM training. The key takeaway is that this is whole process is hard. There exists little to no academic research in this field and most of the knowledge is locked behind vendor's intellectual property. To continue this research we would suggest the following:

- Build a more robust testing framework, capable of both performing page table write and reads and cache line scrubs. Leveraging the correct testing method for different steps, like using page table writes for leveling and scrubs for the reference voltage training, could result in a more stable algorithm.

  Instead of only an overall notion of errors one could go into more detail about errors in each individual bank, row or column, which could in turn shed more light into were the current shortcomings of the algorighm are.

- Perform leveling and testing on each individual byte. This could reduce the size of the search space and would direct testing to a smaller subproblem.

- More data visualization, like the graphs above, for different DIMMs, ranks, speed grades and Enzian systems could be helpful in understanding the whole scope of the problem.

  A mistake we made was trying to solve the problem exclusively from first principle, without relying on much data or visualization.

- A potential approach to address the challenge of training DRAM efficiently and effectively is by leveraging neural networks. Neural networks have demonstrated their capability in recognizing complex patterns and relationships in various domains. Given that the problem of training the DRAM appears to be one of finding the global minimum, and the current algorithm performs gradient descent, neural networks could be a good solution.

5 Configuration of the Enzian

Using the BDK, the workflow to change the DRAM speed or other configurations of the hardware, is to interrupt the boot flow, change the settings via the BDK's BIOS-like prompt, then resume the boot operation. This is both slow and cumbersome. Ideally we would want a way to set these settings in a way that is: 1. persistent, 2. easy for a human to edit and 3. easy to edit by a machine, as the process of configuring the Enzian would ideally be performed by the `emg` command. The `emg` command is currently used to reserve Enzian systems and it would be a nice addition to add a declarative way to set up a machine before using it. To do so we can leverage the Enzian Firmware Resource Interface (EFRI) link connecting the BMC and the ThunderX. This section serves as a starting point for this end goal.

5.1 Implementation of BMC-side EFRI backends

```
{
    "cpu": {
        "dram": {
            "speed": 2133,
            "tFAW": 9
        }
    }
}
```

Listing 1: DRAM configuration. DRAM speed and an optional override are set

In EFRI a schema file is used to automatically generate both a BMC- and ATF-side implementation of the protocol.

To implement the functionality of configuring Enzian declaratively from the BMC a proof-of-concept variable store has been implemented. This relies on a JSON file in which configuration options are stored. A JSON file is used because it is both easy to read/write by humans and machines and because the nested nature of EFRI's paths can be represented without issues (see Listing 1).

The implementation of the variable store is added to the `dram_param type`. Every parameter of type `dram_param` can then be read from the JSON file. This has been done for a (non-exhaustive) list of DRAM parameters that can be overridden by the BMC (see Listing 2).

The Enzian system lacks a traditional RTC and instead relies on the clock of the BMC. As UEFI requires a RTC implementation an EFRI backend was implemented to satisfy this need. This backend, which is accessible under `platform::rtc`, returns the POSIX timestamp of the BMC's clock.

```yaml
common:
    # ...
    dram_param: &dram_param
        type: int32
        actions: *ro
        impl: datasource-var-store
        class: [cpu, dram]
        subsystem: config
    # ...
bmc:
    # ...
    # DRAM parameters - incomplete...
    - {name: speed, desc: Set speed of all the DIMMs, <<: *dram_param}
    - {name: tRRD_S, desc: Row-to-row delay short, <<: *dram_param}
    - {name: tRRD_L, desc: Row-to-row delay long, <<: *dram_param}
    - {name: tFAW, desc: Four Activate Window, <<: *dram_param}
    # ...
```

Listing 2: Excerpt from the Enzian EFRI schema. The 4 DRAM parameters are read from the JSON file.

5.2 Configuration of DRAM parameters using EFRI

During the execution of BL1, a setup involving EFRI takes place before DRAM initialization. This setup involves the utilization of EFRI to request various DRAM parameters from the BMC through a `get` request. If these values are defined in the accompanying JSON file, they are returned; otherwise, an error response is sent.

These returned parameter values, if present, can potentially override the default parameter values. In situations where the speed parameter is absent or unsupported by the installed DIMMs, the speed will default to 1866 MT/s. In the case that the other parameters are not set they will be computer in `tx_lmc_compute_timings`.

The implementation in the ATF for the override of DRAM parameters can be found in the functions `efri_override_dram_params`. Here additional parameters can be specified using the `OVERRIDE_DDR` macro, which, for every DIMM slot, updates the field of a given struct, containing all the parameter, by querying the specified EFRI path (see Listing 3). These structs are then used to set the LMC's registers during the DRAM initialization procedure. For future generic configuration parameters `efri_override_params` can be used along with the `OVERRIDE` macro.

```
void
efri_override_dram_params(struct jedec_ddr4_cycles *timings,
                          struct ddr4_configs *configs) {
    INFO("Handling overrides via EFRI\n");
    OVERRIDE_DDR(timings, trp, "cpu:dram::tRP")
    OVERRIDE_DDR(timings, tcke, "cpu:dram::tCKE")
    OVERRIDE_DDR(timings, tcksre, "cpu:dram::tCKSRE")
    OVERRIDE_DDR(timings, txp, "cpu:dram::tXP")
    OVERRIDE_DDR(timings, txpr, "cpu:dram::tXPR")
}
```

Listing 3: Usage of the `OVERRIDE_DDR` macro to override DRAM parameters

5.3 EFRI as an EL3 service

In addition to EFRI being used in the BL1 to load configuration parameters it is also installed as a EL3 service in BL31. It can therefore be used by the operating system or UEFI to communicate with the BMC over SMC calls. To perform a UEFI request via SMC one first needs to allocate a 4K page and then perform the `POST_ARGSPACE` SMC call with the base address of said page. This creates a shared page which is used to pass the EFRI frames between userspace and the secure monitor. To send a frame, which has been populated in the allocated page, the CLIENT_INVOKE SMC call has to be executed. After a successful exchange of messages with the BMC the response frame is placed in the shared page.

Beyond its utilization in BL1 for loading configuration parameters, EFRI also serves as an essential component installed as an EL2 service within BL31. This incorporation within BL31 extends the accessibility and functionality of EFRI, allowing the operating system or UEFI to establish communication with the BMC through SMCs.

When initiating a UEFI request via a SMC calls, a systematic process is followed. The first step involves the allocation of an aligned 4 KiB page, designated for this communication. Subsequently, the `POST_ARGSPACE` SMC call is executed, with the base address of the allocated page. This operation establishes a shared memory space, facilitating the seamless transfer of EFRI frames between userspace and the secure monitor.

To transmit a populated frame, which resides in the allocated memory page, the `CLIENT_INVOKE` SMC call is invoked. This action effectively transfers the frame's content to the BMC, initiating communication.

Following the successful exchange of messages with the BMC, the response frame generated by the BMC is deposited into the same shared memory page. This streamlined mechanism enables the passage of information between the system's various components and the BMC, facilitating communication and cooperation for system operation and management.

5.4 Modifications to the ATF-side

During the migration to the new ATF implementation, an effort was made to preserve the generated code for the ATF-side implementation, while some adjustments were necessary to ensure compatibility and functionality within the new framework. This section outlines the specific modifications made to the ATF implementation.

- Initially, an attempt was made to register a console and unbind it from the various scopes. The goal was to expose the `putc` and `getc` functions from the console. However, discrepancies between the old ATF and the new version, especially in assembly implementations, led to issues. The new ATF's assembly-based approach caused intermittent character drops.

  To address this, the solution involved reworking the `putc` and `getc` functions. The new versions closely resemble those in the old ATF. This ensured consistent and reliable console operations for the ATF-BMC communication to use.

- Changes were made to the External Firmware Runtime Interface (EFRI) code to enhance its readability. Type casts were used frequently and made the code difficult to understand. In the new implementation these have been replaced by defining new structs and unions. These simplifications are aimed at making the codebase more readable and facilitating future maintenance.

- Changes in the XLAT library API, regarding dynamic and static page allocation, forced some modifications to the generated code. In the old API there was no distinction between static and dynamic page allocation. These changes necessitated adjustments to code implementation to align with the updated API, ensuring the correct allocation mechanism was utilized. Porting the old code to the new API resulted in the following bug in the EFRI implementation: When registering two different argspages with the secure monitor the two physical pages from userspace would be alias and mapped to the same virtual address in the ATF. This was resolved by first unmapping the old page before mapping the new one.

In summary, the adaptation process sought to balance continuity with necessary adjustments. The aim was to align the existing ATF codebase with the new ATF framework's requirements, ensuring that the system's functionality, compatibility, and clarity were maintained.

5.5 Future work and improvements

The process of extending the EFRI schema for the Enzian system or introducing supplementary backends was facilitated by the inherent modularity of the protocol. This design characteristic streamlined the integration of new features and functionalities.

However, a shortcoming was identified regarding the construction of argspages for specific requests. To address this limitation, helper functions were developed for use within the ATF[16]. Although these functions served their purpose, their ad-hoc nature has to be considered suboptimal.

A potential solution for enhancing usability involves transitioning from the argspage convention to Protocol Buffers [Varda, 2008]. Protocol Buffers, due to its automated generation of getter and setter methods for specified types, presents a more streamlined approach and eliminates the need for an ad-hoc library. This transition aligns well with EFRI's modular design, offering an improved and user-friendly protocol implementation.

Another avenue for improvement involves shifting away from modifying configurations via direct writes to the JSON file, and instead utilizing the `emg` utility for this purpose. This approach holds the potential to enhance the configurability of Enzian machines, enabling them to be reserved and preconfigured according to specific requirements. By implementing this change, the process of altering configurations becomes more streamlined and versatile.

[16]https://gitlab.inf.ethz.ch/PROJECT-Enzian/arm-trusted-firmware-enzian-port/-/blob/v2.9-enzian/plat/enzian/enzian_bmc_comms.c

6 Evaluation

6.1 Firmware Size

Binary	Size (in KiB)
bl1.bin	45
bl2.bin	17
bl31.bin	37
	99

Figure 14: New firmware size

Binary	Size (in KiB)
bdk.bin	4096
bl1.bin	27
bl2.bin	27
bl31.bin	55
	4205

Figure 15: Old firmware size

By creating a new firmware stack that does not require the BDK, our expectation was to significantly reduce the overall size of the firmware image.

To validate this, we measured the sizes of the binary components in both the old and new firmware stacks. The results in Table 1 and Table 2 clearly show a substantial size reduction from about 4 MiB to around 100 KiB. This reduction is due to completely removing the BDK. The new BL1 is slightly larger in size to the old one due to the added code for the DRAM initialization. An additional advantage is that flashing the binary now takes less time, which is useful for quickly testing firmware. Including the UEFI the old stack took 40 seconds to flash, whilst the new stack took only 18 seconds, also with the UEFI.

6.2 EFRI performance

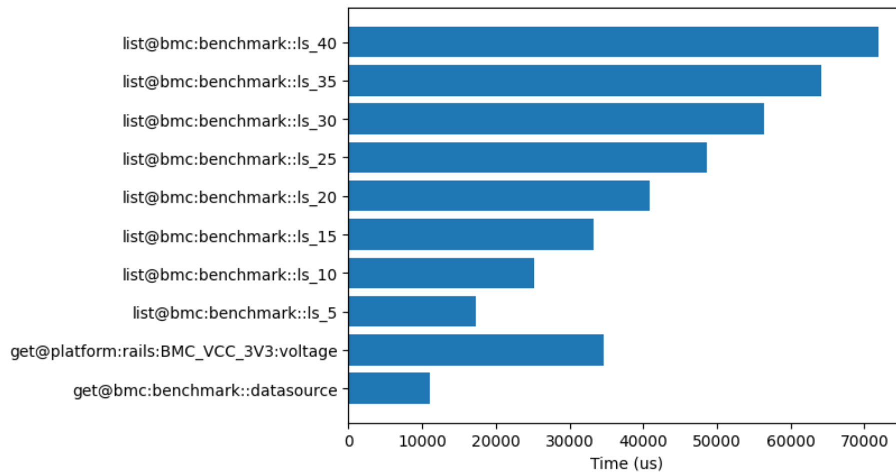


Figure 16: Latency for different endpoint invocations using the EFRI link

Table 16 provides a visual representation of the delay observed when accessing specific endpoints from the ThunderX system using the EFRI link. This latency test serves as a valuable benchmark to verify whether the new ATF performs comparably to its predecessor and whether the EFRI link operates as expected.

The test procedure involves utilizing the ATF commit `d6255b43`[17] and UEFI commit `1323f53d`.[18]. Following the transfer of control to UEFI, a 4 KiB aligned page is allocated. Subsequently, this allocated page is utilized as an argument to an SMC call to register it with the EFRI service running in the secure monitor. The assessment involves generating and transmitting multiple EFRI requests, while simultaneously measuring the time span between request initiation and receipt of a response.

Notably, the outcomes presented Table 16 exhibit a strong correlation with the findings documented in [Xu, 2023]. This is to be expected as the major bottleneck of the EFRI link lies in the baudrate of the UART.

6.3 Time to UEFI

In this test the aim is to quantify the time required for both the legacy firmware stack and the new firmware stack to reach the UEFI state after the ThunderX system has been released from reset. We also intend to measure the duration of the DRAM initialization process. This task, however, presents challenges due to the lack of an early-stage timing infrastructure during the boot process. The boot development BDK prompts for user input upon startup, and no established timing mechanism is available.

While an optimal solution would involve implementing a timer triggered by the power-on command from the BMC and stopping upon receiving an EFRI message, we opted for manual measurements due to our focus on relative rather than absolute precision. Each entry in Table 1 corresponds to the average of 15 measurements.

	new ATF	BDK + old ATF
DRAM init.	4 s*	7 s
Total	9 s*	19 s

Table 1: Time from power-on to UEFI for both the old and new firmware stacks

During testing, we observed a fluctuation in the time taken by the new ATF to initialize DRAM. Among 15 measurements, 14 consistently showed an approximate 4-second initialization time. However, one measurement deviated significantly with an 11-second time. This variability demonstrates that while

[17]https://gitlab.inf.ethz.ch/PROJECT-Enzian/arm-trusted-firmware-enzian-port/-/commit/d6255b439f0eb5df22fbf214cc738bb791401d11

[18]https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2023-bsc-alegnani/edk-2-enzian/-/commit/1323f53d2a3444ba20c3973046257546af415ddd

the algorithm typically performs well, it doesn't consistently exhibit deterministic behavior.

While a direct comparison of total boot times is infeasible due to the substantial variation in initialization steps between the old and new firmware stacks, a meaningful comparison can be drawn for the DRAM initialization phase. Notably, the new ATF exhibits a level of performance in this aspect that is either on par with or potentially surpasses that of the old version.

6.4 DRAM latency and bandwidth

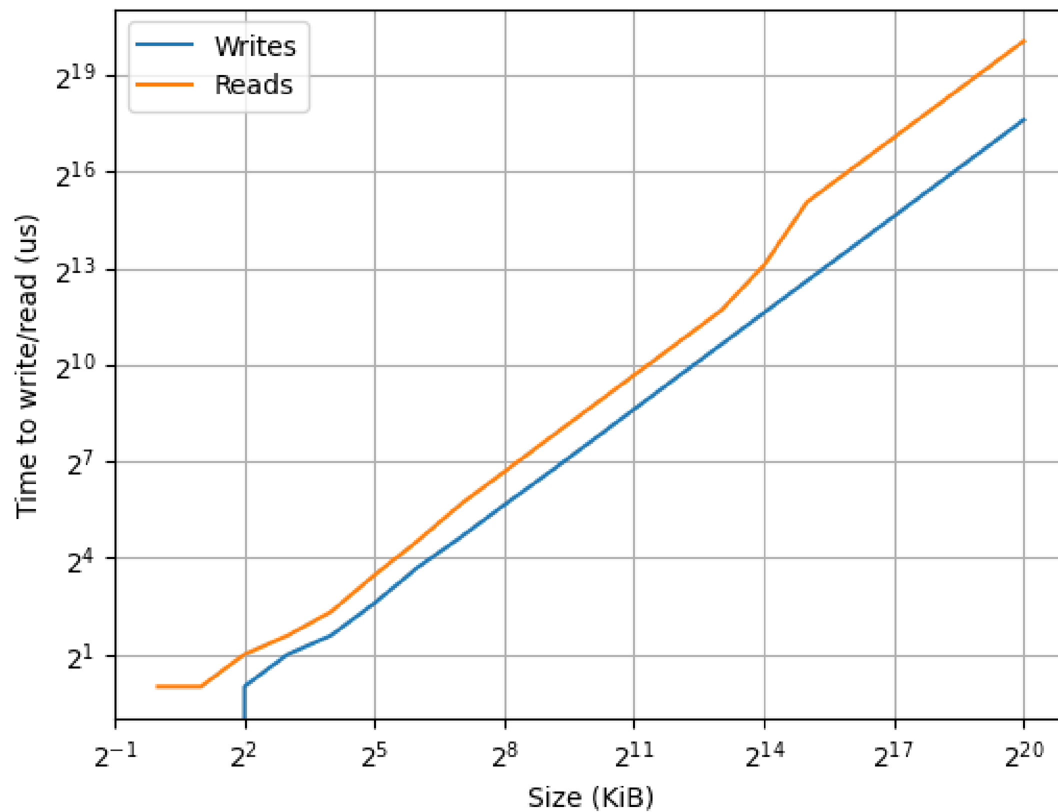


Figure 17: Main memory bandwidth of sequential reads/writes

Our objective is to assess the bandwidth of the DRAM memory, verifying its alignment with the configured speed. This evaluation was executed within the BL31 stage, just before the transition to UEFI. The benchmark entails two core actions: first, a pattern is written into a consecutive memory area; second, the written data is subsequently read back. The measured metrics encompass

both the read and write times. The test first warms up the L2-cache by zeroing the first 16 MiB of the memory region.

For this assessment, measurements were collected across a contiguous memory region spanning from 1 KiB to 1 GiB.

This comprehensive range was chosen to provide a thorough understanding of the memory's performance characteristics at varying data sizes.

As demonstrated in Figure 17, when viewed on a logarithmic scale, the time taken for memory region writing exhibits a linear growth trend. Conversely, the read time displays a distinct bump at the 2^{14} KiB mark. This behavior aligns with expectations due to the ThunderX's L2 cache size of 16 MiB.

An interesting observation is the absence of the same elevation in the write time measurements. This phenomenon is attributed to the fact that a single ThunderX processor cannot fully saturate the DRAM bandwidth. Therefore, the write time measurement reflects the performance of the L2 cache rather than the DRAM bandwidth.

However, it's worth noting that read time measurements beyond the 16 MiB size capture the read bandwidth. Despite this, the measured read speed of 3.7 GB/s is notably lower than 56 GB/s, the maximum bandwidth of a quad-channel configuration of DDR4-1866. This discrepancy is likely due to the limiting factor of the ThunderX processor's speed, suggesting that the read bandwidth is constrained by the processor's performance rather than the DRAM's potential burst bandwidth.

Due to the inconclusive findings from the test, it is left as future work to perform a memory benchmark using multiple cores of the ThunderX in order to fully saturate the available DRAM bandwidth.

6.5 Code base comparison

SLOC	new ATF	BDK	old ATF
DRAM init.	2,575	12,397	–
Total	4,198	1,107,777	7,473

Table 2: Code base comparison between the new ATF, BDK and old ATF

One of the goals of the rewrite was to reduce the code base and make it more manageable. To get the measurements in Table 2 the SLOCcount [Wheeler, 2004] utility was used. To measure the DRAM initialization code the directories and files in Table 3, where used to quantify the lines of code.

The code base of the new ATF is significantly smaller then the old ATF although it contains the DRAM initialization. Although the ATF doesn't implement every feature needed to run the Enzian, like the PCIE initialization, we can compare the code base for the DRAM initialization. The new ATF's implementation is 5x smaller then the one in the BDK. From the measurements we can observe that the code base of the new ATF is significantly smaller than its predecessor, even when considering its inclusion of the DRAM initialization.

The DRAM initialization implementation in the new ATF is approximately five times smaller than the equivalent one found in the BDK. It's crucial to emphasize that transitioning to the new ATF eliminates the substantial BDK code base from the maintenance equation, significantly streamlining the code base and reducing the maintenance burden. This does not paint the whole picture as one must keep in mind that some of the BDK's functionality will have to implemented in the UEFI.

7 Future work

Due to the constraints of the limited timeframe and the comprehensive scope of this thesis, certain aspects have not been fully addressed. Notably, the key features listed below have been left as future work.

7.1 DRAM initialization

The new implementation of the DRAM calibration is still not completely reliable especially (and almost exclusively) for the highest speed grade of 2133 MT/s. As there are plans to use the second revision of the ThunderX on future versions of Enzian, which additionally supports the 2400 MT/s speed grade, the algorithm has to be refined further to ensure stability of the system. Suggestions on how it can be improved can be found in Section 4.10.5.

7.2 Extensive testing in Linux

The ATF port has been created without considering backward compatibility with the old UEFI. At present, the rewrite of the UEFI [Montini, 2023], is in the process of development. However, this new version does not yet possess the capability to boot the Linux operating system. This absence of a functional operating system significantly limits the available range of testing opportunities.

As the development of the new UEFI progresses, and it reaches a point where it can successfully initiate the boot process for Linux, a comprehensive suite of testing options becomes available. The foremost initial test entails the successful booting of the Linux operating system. This test is pivotal as it confirms the adequate implementation of all necessary features within the ATF.

Furthermore, a critical aspect involves the execution of thorough memory benchmarks and tests, including assessments like the MemTest86 suite [PassMark Software, 2023]. These memory-focused evaluations provide valuable insights into the performance and reliability of the DRAM initialization in the new ATF.

7.3 Reset behaviour

The ThunderX features both hard and soft reset capabilities. In a hard reset a complete restart takes place where all components are initialized anew. This action clears any data in the DRAM.

A soft reset on the other hand restarts the processor whilst retaining the data stored in the DRAM. Currently, only the hard reset is implemented as, to implement the soft reset, changes to the DRAM initialization code are necessary. At the time of writing the ATF implementation treats a soft reset the same as a hard reset.

7.4 PSCI implementation

As previously indicated in Section 3.12, it is important to reiterate that the current implementation of the Power State Coordination Interface (PSCI) remains incomplete. Specifically, the aspect that requires further development pertains to the initiation of multiple core power-ups. This topic was not addressed in this thesis due to the inherent challenges of testing multi-core functionality without the ability to boot Linux. Similarly, the capacity to power down cores during periods of inactivity has not been explored.

To address this limitation, a potential solution lies in utilizing the reserved memory space situated below BL31, as illustrated in Figure 7. This unutilized memory segment can serve as storage for managing core shutdown states. While the full implementation and testing of these features were beyond the scope of this thesis, the allocated memory area offers a valuable resource for future endeavors aimed at achieving dynamic core power management.

8 Conclusion

From the work conducted in this thesis, several significant points arise for careful consideration, particularly concerning DRAM initialization.

DRAM initialization inherently encompasses intricacies such as voltage adjustments, timing configurations, and signal integrity management. The complexities inherent to this process create a promising avenue for comprehensive research. Addressing these intricacies could potentially yield advancements in memory performance and overall system reliability.

An important observation stems from the limited extent of research within the domain of DRAM initialization. This scarcity can be attributed to the prevalence of proprietary intellectual property held by various chip manufacturers. Another obstacle to research is the diverse landscape of memory controllers. Varied memory controllers offer distinct characteristics and approaches to DRAM initialization. This diversity restricts the applicability of research to a specific subset of memory controllers.

The emergence of the RISC-V architecture as an open and standardized platform bears significance for academic research endeavors. RISC-V's openness fosters collaboration and facilitates the establishment of standardized methodologies and approaches [Fisher, 2020]. This architectural standardization offers potential as a foundational element for future research and optimization initiatives, enabled by a universally standardized memory controller architecture. This could be used to perform generalized research in the field of memory initialization.

Seen more broadly, the entirety of firmware has the same underlying issues. Throughout the history of computers, firmware has evolved in a piecemeal manner, reflecting incremental changes over time. Notably, prevailing standards like UEFI and BIOS highlight the persistence of outdated models, underscoring the necessity to comprehensively reassess the foundational principles that underlie system design[Cantrill, 2022]. This is also highlighted by Section 6, that shows that there is a significant margin for improvement in metrics like code base size, binary size and boot time. The need for open research into these aspects of computing becomes evident, given that it is the foundation of computing.

9 Appendix

9.1 Updating ATF

Assume one is working on the branch `v2.9-enzian`, which has been branched off `master` at the `v2.9` tag. First get the upstream changes from the official repository:

```
git checkout master
# add the upstream repository
git remote add upstream git@github.com:ARM-software/arm-trusted-firmware.git
# fetch latest changes
git fetch upstream
# Merge the change using the fast-forward strategy
git merge upstream/master --ff-only
```

With the update master branch we want to rebase our branch onto the latest release, which here we assume is v2.10:

```
git checkout v2.9-enzian
# create new branch to reflect the version change
git checkout -b v2.10-enzian
# rebase v2.10-enzian onto master at the v2.10 tag
git rebase --onto v2.10 master
# fix possible merge conflicts and continue
git rebase --continue
# if something goes wrong abort using
git rebase --abort
```

9.2 Building and flashing the ATF

The ATF is being built using a Docker container and the thunderx-boot-flash-tool repository, which is included as a submodule in the trusted firmware repository. To initialize the submodule for the first time run `git submodule update --init`. Then create the Docker image by running the `build-docker-image.sh` script. Run `build-atf.sh` to create the `atf.bin` binary in the root of the repository. The above script requires the `THUNDER_EFI.fd` UEFI file to be in the root of the repository.

9.2.1 Manually building the ATF

```
# make the BL1
make PLAT=enzian LOG_LEVEL=LOG_LEVEL_INFO bl1
# make the BL2 and BL31 and package them into the FIP
make PLAT=enzian LOG_LEVEL=LOG_LEVEL_INFO fip
# package BL1 into the format the ThunderX expects in flash
python3 create.py -e --untrusted bl1.bin bl1-flash.bin
```

```
# make the tool to modify the FIP
make fiptool
# Add the UEFI(BL33) to the FIP
./fiptool update --nt-fw THUNDER_EFI.fd fip.bin
# Concatenate the (packaged) BL1 and FIP
cat bl1-flash.bin fip.bin > atf.bin
```

9.2.2 Flashing an EBB

To flash the ATF onto the EBB first copy the file over to `enzian-server` with `scp atf.bin enzian-server:/var/lib/tftpboot/enzian/<your-id>`. After getting the BMC console for the EBB on `enzian-server` with `console hinterrugg0x-bmc` press `P` to power on the machine and run `thunder_update -n enzian/<your-id>/atf.bin` to flash the new firmware. Remember to turn off the machine by pressing `P` when you are done or people in the office will rightfully get mad.

9.2.3 Flashing an Enzian

To flash the ATF onto an Enzian first copy the file over to the bmc of an Enzian with `scp atf.bin zuestoll0x-bmc:<your-id>`. After getting the BMC console for the Enzian as described in the quickstart guide, execute `common_power_up()` to power up the board. Without powering on the CPU ssh into `zuestoll0x-bmc` and run the following command `flascp <your-id>/atf.bin -v /dev/mtd0`. Afterwards power on the CPU from the shell by using `cpu_power_up()`. Remember to power of the CPU and flash a stable version of the boot image after use.

9.3 Using EFRI on an Enzian

The code for the EFRI interface can be found in this repo[19]. Generate the BMC-side code by running `python generate.py enzian-efri.yml bmc efri/target/generated.py > efri/target/generated.py`. Then change the path at which the json file will be located on the BMC in `var_store.py`. SCP the whole `target` directory to the BMC and start executing the module with `python -m target`.

9.4 Debugging an EBB using JTAG

At the time of writing only the EBB `hinterrugg01` can be debugged via JTAG. Every Enzian board has a JTAG connection but it's not yet functional. The debugging uses arm Studio which is available on `enzian-build` via X11 forwarding. To use it you need a configuration file in your home directory. To launch arm Studio run:
`ssh enzian-build -X /opt/arm/developmentstudio-2021.2/bin/armds_ide`

[19] `https://gitlab.inf.ethz.ch/PROJECT-Enzian/bmc/enzian-firmware-resource-interface/-/tree/atf-port`

-configuration /storage/user/<your-id>/.armds. Create a new hardware debug connection and select Cavium - ThunderX AP1 as the target. Then select the core to debug (Debug ThunderX_00 and set the address of the debug probe to dstream1.ethz.ch. Happy debugging! NOTE: Before flashing a new image to the EBB disconnect the debugger! Otherwise the BMC of the EBB will complain about a power supply issue. If this happens ssh to enzian-build and kill the armds processes.

9.5 Code base measurements

Measurement	Included directories & files
new ATF - Total	plat/enzian drivers/cavium/ lib/efri include/drivers/cavium include/lib/efri
new ATF - DRAM init.	drivers/cavium/thunderx/thunderx_lmc.c plat/enzian/enzian_dram_init.c include/drivers/cavium /thunderx/thunderx_csr_lmc.h include/drivers/cavium/thunderx/thunderx_lmc.h
old ATF - Total	plat/thunder lib/efri include/lib/efri
BDK - Total	.
BDK - DRAM init	libbdk-dram libdram

Table 3: Directories and files used for the code base measurements

9.6 Summary of the Capabilities and Limitations of the new ATF

- all 4 DIMM slots populated with DDR4 memory running in quad-channel
- speed grades 1600, 1866, 2133 (the latter not working)
- UDIMMs or RDIMMs supported
- no support for hybrid or non-monolithic DIMMs
- support for x4 and x8 single or double rank DRAM
- no support for asymmetric ranks or different DRAM capacities
- no backwards-compatibility with the old UEFI
- ATF and UEFI can't be flashed separately
- ATF initializes the consoles, DRAM, PSCI, timer and interrupt controller; other interfaces like USB, PCIE etc. will be handled by the UEFI
- watchdog timer has not been initialized
- EFRI link, enabling communication between the CPU and the BMC, fully working
- only hard reset implemented, no soft reset
- PSCI not fully working, no power gating of cores, state of multicore support untested

References

[Arm, 2017] Arm (2017). Aarch64 exception and interrupt handling. https://documentation-service.arm.com/static/5f872814405d955c5176de2e.

[Arm, 2022] Arm (2022). Aarch64 exception model. https://documentation-service.arm.com/static/63a065c41d698c4dc521cb1c.

[Arm, 2023] Arm (2023). Trusted Firmware-A documentation. https://trustedfirmware-a.readthedocs.io/en/latest/index.html.

[ARM Limited, 2007] ARM Limited (2007). Primecell uart (pl011) technical reference manual. https://documentation-service.arm.com/static/5e8e36c2fd977155116a90b5.

[Cantrill, 2022] Cantrill, B. (2022). I have come to bury the bios, not to open it: the need for holistic systems. https://www.osfc.io/2022/talks/i-have-come-to-bury-the-bios-not-to-open-it-the-need-for-holistic-systems/.

[Cavium, 2017] Cavium (2017). *Cavium ThunderX CN88XX, Pass 2 Hardware Reference Manual, Revision 2.7.*

[Cock, 2022] Cock, D. (2022). Thunderx boot flash tool. https://gitlab.inf.ethz.ch/PROJECT-Enzian/thunderx-boot-flash-tool.

[Cock et al., 2022] Cock, D., Ramdas, A., Schwyn, D., Giardino, M., Turowski, A., He, Z., Hossle, N., Korolija, D., Licciardello, M., Martsenko, K., Achermann, R., Alonso, G., and Roscoe, T. (2022). Enzian: An open, general, cpu/fpga platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 434–451, New York, NY, USA. Association for Computing Machinery.

[Fisher, 2020] Fisher, Y. (2020). Fostering open innovation in hardware.

[Micron, 2014] Micron (2014). 4gb: x16 ddr4 sdram features. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/4gb_ddr4_dram_2e0d.pdf.

[Montini, 2023] Montini, A. (2023). Boot firmware for heterogeneous systems running linux.

[PassMark Software, 2023] PassMark Software (2023). MemTest86.

[Varda, 2008] Varda, K. (2008). Protocol buffers: Google's data interchange format. Technical report, Google.

[Wheeler, 2004] Wheeler, D. (2004). Sloccount. https://dwheeler.com/sloccount/.

[Xu, 2023] Xu, P. (2023). Enzian firmware resource interface.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Trusted Firmware for a Research Computer

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Legnani	Alessandro

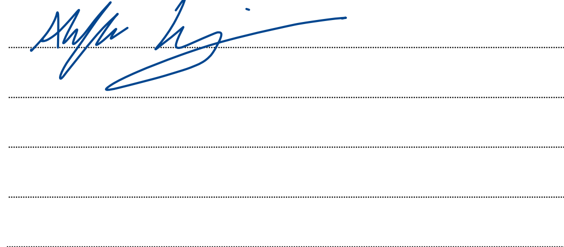
Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum

Milano, 21. 8.2023

Unterschrift(en)



Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.