Towards a Unified Firmware for Enzian

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 513b

Systems Group, Department of Computer Science, ETH Zurich

Towards a Unified Firmware for Enzian

by

Thomas Meyer-Lehnert

Supervised by

Zikai Liu
Daniel Schwyn
Prof. Timothy Roscoe

May 2024 — November 2024

D INFK

Abstract

Firmware, the code running before the operating system starts, is a vast and interesting subject. Unfortunately, most knowledge in the field is owned by silicon vendors and not openly available, hindering research projects like Enzian. A large amount of hard-to-maintain, proprietary legacy code makes it difficult to extend it for research purposes, and poses a risk in technical debt and security.

The Systems Group at ETH Zürich is developing the open Enzian research computer. The vendor-provided firmware is proprietary, thus in conflict with the openness principle, as well as difficult to adjust for research purposes. Thus, the project of creating a new firmware was initiated. Continuing the work started by the two prior theses, the original goal of this thesis was to integrate the two previously developed components and create a working firmware stack.

In the process of writing this thesis it became evident that this endeavor would be much more involved than initially anticipated. The numerous problems that arise from a project of this nature will be explored. This thesis makes progress towards a production ready firmware stack and aims to be a guide for how to efficiently proceed with this project.

Acknowledgements

I want to thank the Systems Group and Prof. Dr. Timothy Roscoe for the chance to work on such an interesting project and for all the knowledge I acquired through it.

A few individuals played an essential role while writing this thesis. My supervisors, Zikai Liu and Daniel Schwyn helped me find the path forward as well as encouraged me in times when I struggled due to my stagnating progress.

Lastly, I express my gratitude towards my girlfriend Mars, who supported me every step of the way and alleviated the stress endless debugging sessions incurred on me, as well as my friends and family for proofreading.

Contents

1 Introduction

Long ago, hardware and software were developed in harmony. Then, everything changed, when firmware was invented. At the time computers were becoming more sophisticated, hardware complexity increased dramatically. Relying on operating system programmers to implement every hardware detail for every chip in existance became unsustainable, portability was needed. The mostly proprietary nature of silicon[1] development at that time, and the lack of standardization gave birth to what we call firmware. Code that runs before any user supplied code (like an operating system), that is needed to make different machines look similar enough, so that an operating system does not have to know about every last intricacy of the system it is running on [3]. With this abstraction, hardware and software became separated. Hardware vendors would supply firmware with their chips, and operating systems would only need small adjustments to work on the platform. They would be developed separately. The hardware complexities would only be documented poorly or not at all. The firmware abstracts it, no need to worry about it.

In the modern day, firmware has become ubiquitous. Every modern 'PC' has a UEFI (Unified Extensible Firmware Interface)[15][2], which to the normal user has become the default. 'That is where I configure my hardware'. What is not well known, is how much code actually runs, before the familiar configuration interface even appears on the screen. Going even further, why is all this even needed? Should the operating system not be facilitating the hardware? This question cannot be answered easily. On x86 platforms, it has effectively become mandatory to run a UEFI implementation. Virtually every operating system expects it, and will not run without it[3].

The Enzian research computer, developed at the Systems Group at ETH Zürich, uses an Arm-based CPU. Arm chips were traditionally used in embedded environments, where the hardware and software usually are still developed together. There is a standard for how to develop firmware for Arm platforms, the ATF (Arm Trusted Firmware, or Trusted Firmware-A)[2]. In theory, this firmware could be loaded right after reset, perform hardware initialization and then load a Linux kernel to jump to it. In practice, this is not the case. The firmware for the Cavium ThunderX (the CPU used in Enzian) consists of three stages: The proprietary Cavium BDK (Bringup and Diagnostics Kit), an ATF and a UEFI implementation.

Why did this happen? The coreboot project [7] calls this the "set and forget" model of developing firmware. There is some existing code, that already provides most of what is needed for a new platform. Vendors now take this old code of theirs and just make it part of the new firmware, with minor modifications. The BDK performs many different functions (refer to Section 3), some that should

[1] Silicon is used to refer to chips.
[2] Colloquially often still referred to as BIOS
[3] At least not with non-trivial modifications.

be part of the ATF, some of the UEFI, and some that should not be done in firmware at all. Later stages then reconfigure the parts that are 'wrong'. Many stages each doing what they think is correct, with no clear separation of concerns, together with swaths of legacy code stitched together makes firmware hard to understand, maintain and extend, especially for someone outside of the vendor, even when given the proprietary source code.

As the Enzian is meant to be an open research platform, every part of it should be open. The closed source nature of the vendor-supplied firmware goes against this principle, hence it was decided to create a new, open firmware stack to allow sharing, and enable easier integration of new features.

The old firmware stack consists of three components, the Cavium BDK, an ATF and an UEFI implementation. For the new firmware stack, the plan was to eliminate the proprietary BDK, and rebuild the other two components from the ground up, leaving only code based on open source projects.

Previous work was conducted on this project (see Section 2.11), separately for the two components ATF and UEFI. The initial goal of this thesis was to unify them, fix existing issues and implement missing features to create a working firmware image that can boot into Linux.

Working on this problem however has revealed that the difficulties regarding developing this firmware were much greater than anticipated.

Developing directly on the hardware brings challenges. While the programmer has total freedom, this means they also have the freedom to commit mistakes without being made aware of them. A typo in EL3[4] code has the potential to cause devastating consequences. Address zero could become a valid memory address, hardware devices could be initialized with invalid data, or a stack overflow could cause crashes and hangs that look inexplicable at first glance. All of these happened while working on this thesis.

These difficulties make developing firmware a slow and tedious process. Still, this can be overcome with sufficient time and dedication. The real problem is the lack of knowledge. The vendor has developed the hardware. They know every little detail of it, every quirk, every edge case. Even the 2000 page hardware manual, while seeming highly detailed and useful at first, is lacking information in virtually every section. This makes it nearly impossible to create new firmware without reusing a lot of the existing code.

This thesis analyzes the existing, old firmware stack, specifically the BDK. It then explores how this code can be reused in the creation of a new firmware. This method is applied to implement stable DRAM configuration support for all possible speeds. Some non-obvious bugs in the existing code are identified and fixed. Lastly, a way this project can proceed efficiently is described.

[4]The most privileged mode of execution on Arm processors, see Section 2.10.1

2 Background

2.1 Enzian

The Enzian is a research computer developed by the Systems Group at ETH Zürich [6]. It aims to be an open, extensible and flexible platform for systems software research.

Enzian has two NUMA (Non-Uniform Memory Access) nodes. Node 0 is a Cavium ThunderX CPU, node 1 is a Xilinx FPGA (Field Programmable Gate Array). The ThunderX supports a cache coherency protocol, the CCPI™ (Cavium Coherent Processor Interconnect), for dual-node setups, which the ECI (Enzian Coherent Interconnect) is based on.

2.1.1 ThunderX

The ThunderX (CN88XX) is a full featured Arm ARMv8.1 based CPU with 48 Processing cores intended for use in datacenters. It supports up to 128GiB of DRAM (Dynamic Random Access Memory), 16MiB of level 2 cache, up to 6 PCIe (Peripheral Component Interconnect Express) interfaces, and up to $8 \times 10Gb/s$ or $2 \times 40Gb/s$ ethernet connections [5].

2.1.2 BMC — Board Management Controller

The BMC is a small Arm CPU on the Enzian motherboard. It is in charge of power management, and can provide configuration to the firmware running on the ThunderX via EFRI (Enzian Firmware Resource Interface).

By connecting to the BMC of an Enzian via SSH or serial console, manual control about the mentioned aspects is possible. Additionally, firmware can easily be flashed to the CPU.

2.1.3 EFRI — Enzian Firmware Resource Interface

EFRI is a protocol used for ThunderX - BMC communication. It allows software running on the main CPU to power down the board via an SMC (Secure Monitor Call), or allows providing configuration values such as DRAM speed to the firmware from the BMC [6].

2.1.4 FPGA — Field Programmable Gate Array

The FPGA in Enzian is connected to the main CPU by ECI, and by a PCIe link. This enables it to be used as a secondary NUMA node, or as an arbitrary peripheral device.

2.2 TWSI — Two Wire Serial Interface

TWSI is a derivation of I^2C (Inter-Integrated Circuit), which is a serial communication bus specification. The ThunderX uses it to communicate with a number of hardware devices, most notably the DRAM controller.

On Enzian there are six TWSIs. They can be used to send read and write commands to connected devices. This works by writing the device address, flags and (for write operations) optionally a value into the TWSI hardware register(s), then waiting for the operation to complete and finally reading the result out of said register(s). For details, see the ThunderX manual section 34 [5].

2.3 DRAM — Dynamic Random Access Memory

DRAM is volatile[5] memory. It is an integral part of modern computers, as it allows a machine to use large quantities of reasonably quick to access and cheap memory. Usually, the programmer as well as the user expects the memory to work consistently. After writing a value to address X, reading from that address should yield back the same value, until it is overwritten again. This behavior however is not trivial to achieve. DRAM training or configuration is the process that discovers the correct parameters for the DRAM controller and configures it accordingly. This has to happen in the early stages of booting, as without it memory cannot be used.

2.3.1 Structure

Modern memory modules (DIMMs - dual in-line Memory Modules) are organized into logical groups called ranks, which in turn consist of multiple physical chips containing a number of banks per chip. To access a memory location, the memory controller issues a command to the respective module on a data line. The difficult part is to properly interpret the output from the memory chip that comes back.

2.3.2 Training

As the physical chips on a module are at different distances from the memory controller, a command for a chip that is farther away will have a larger delay before its response can be read. This delay has to be determined in a process called read/write-leveling. Additionally, the analog-digital conversion is not trivial either. The correct reference voltage (which determines if the signal is encoding a 1 or a 0) is found in a process called V_{ref} training.

This is a very short overview omits much of the complexity that the DRAM training process has. Alessandro Legnani's thesis goes into great detail about DRAM structure and DRAM training [10]. Refer to its sections 2.6 and 4.

[5] volatile = loses data when power is removed

2.4 ARMv8 Internals

2.4.1 Exception Levels

In AArch64, the processor can operate in four different exception levels called EL0 - EL3 (Exception Level). The exception level determines the level of privilege. The most important part of this is which system registers and memory regions are accessible. EL3 is the most privileged execution mode, while EL0 is the least privileged.

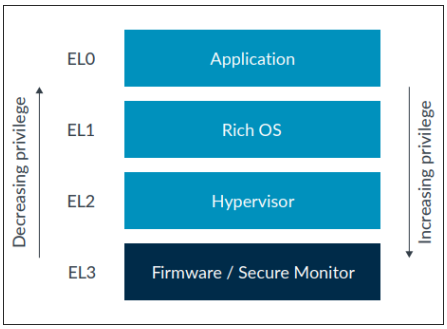

Figure 1: AArch64 Exception Levels [1]

Usually, EL3 is used for low level code like power management (accessible via SMCs), EL2 runs a Hypervisor for CPU Virtualization, The OS Kernel runs in EL1, and user code in EL0.

When taking an exception, the exception level can increase (become more privileged), or stay the same. This allows low-privilege code to access functionality by more privileged software layers [1].

2.4.2 Security States

AArch64 supports yet another method of privilege control: Security States. When running in EL2, 1 or 0, the processor can be either in Secure or Non-Secure state. Memory regions can be flagged as secure to only be accessible in the Secure State [1]. This feature is currently not in use by Enzian.

2.5 ATF — Arm Trusted Firmware (Trusted Firmware-A)

> "Trusted Firmware-A (TF-A) provides a reference implementation of secure world software for Armv7-A and Armv8-A, including a Secure Monitor executing at Exception Level 3 (EL3)"[2]

ATF, officially called "Trusted Firmware-A" (TF-A), but often still referred to under its former name, "Arm Trusted Firmware", is an open source boot firmware implementation for Arm CPUs, published and developed by Arm. It is responsible for initializing hardware, and installing a Secure Monitor to handle SMCs, then passing control over to a UEFI implementation, an OS or another payload [2].

A large part of ATF are reusable libraries that implement many common features needed in CPU firmware. Notable examples of existing features are: XLAT, a library that implements memory address translation helpers and facilities to load the later boot stages. Alessandro Legnani's thesis [10] goes into great detail about this.

When implementing firmware for new devices, the developer is encouraged to use the plethora of features ATF offers, and looking at platform specific code for different, but similar chips to derive code for the new platform.

The responsibilities of ATF are similar to those of the BDK. For a more detailed description, refer to Section 2.8.

The ATF defines a multi-stage boot process. These stages, and how they fit into the full Enzian boot process is described in Section 2.10.

2.6 UEFI — Unified Extensible Firmware Interface

UEFI is the last firmware stage, and is responsible for loading user level software, such as an operating system. The reference implementation of the UEFI specification is TianoCore EDK II (Efi Development Kit 2) [14]. Both the old and the new UEFI implementation for Enzian are based on EDK II.

In UEFI, drivers are implemented to support different hardware components, and UEFI runtime services are installed to provide functionality to the OS, such as access to hardware time measurement.

2.7 Coreboot

Coreboot is an open source boot firmware implementation for various devices [7]. It shares most of the philosophy behind this Enzian Firmware re-implementation, namely separation of concerns between boot firmware and higher-level software, and minimal responsibility of the boot firmware.

How is coreboot relevant to this thesis? Coreboot has support for the ThunderX. Because at the time of starting this thesis, the boot process using the new firmware stack was slow and unreliable (see Section 2.11.1), especially because of DRAM initialization, coreboots implementation was consulted to find a possible alternative method to configure DRAM.

It was discovered that coreboot includes part of Cavium's BDK code. This version of the code is very stripped down, only including a handful of the features of the original BDK. The part that initializes DRAM was almost identical to that in the code supplied by Cavium. How these findings were applied is described in Section 4.3.3.

2.8 Currently used Firmware Stack

Enzians current firmware stack consists of three components: The Cavium BDK, ATF and EDK II (Efi Development Kit 2) based UEFI. Both the ATF and UEFI implementations are based on unknown, old versions of the upstream ATF and EDK II codebases. This firmware stack was provided by Cavium to the Systems Group, without any version information or history. A number of modifications were made to support different requirements of Enzian.

In this firmware stack, a lot of responsibility is shared. Part of hardware initialization is done in the BDK, part in the ATF, and part in UEFI. Some devices are also initialized multiple times. It is difficult to discern how any single initialization objective is accomplished.

2.9 Desired Firmware Stack

The overarching goal is to create new firmware images from scratch (using up to date ATF and EDK II codebases) to replacing the Cavium supplied firmware stack. The BDK will not be used in the new firmware stack[6]. In this new stack, responsibilities should be clearly separated. The policy at the time of writing is, ATF should only run initialization logic that must be run before UEFI. This primarily refers to code that has to run at EL3, such as DRAM initialization.

While the old firmware stack included many features for other Cavium platforms, the new firmware stack will solely support the ThunderX CPU used in the Enzian system. Features of the ThunderX not used by Enzian, such as DDR3 support, are not considered.

2.10 Boot Process

The ThunderX reset procedure is described in the BDK documentation [4]:

- After reset, the chip starts executing instructions from an internal secure ROM at 0x87d000000000 (`RVBAR_EL3`)
- This code loads 192KiB from flash into the L2 cache at physical address 0x1000000
- For non-trusted boot, code is loaded from offset 0x20000 in flash, for trusted boot this is 0x50000
- The CPU jumps to 0x1000100

The first boot firmware stage is loaded into the so called scratchpad (a specific chunk of memory) in the L2 cache. This is mandatory, because before DRAM initialization is complete, memory cannot be used. The first stage of the firmware that is loaded here has to lock the scratchpad area such that no

[6] At least not in its original form. Parts of it, like DRAM related code, might be used in the new firmware stack, and it will remain as a reference point.

cache line can be evicted, then run DRAM initialization. After memory is working, the scratchpad can be unlocked, and execution can continue by loading later boot stages into memory.

Previously, the code that is loaded into the scratchpad was from the BDK. It would do its job as described in Section 3, then hand over to the ATF. In the new stack, the ATF is responsible for platform initialization, thus its first stage is loaded into the scratchpad.

2.10.1 ATF Stages

The ATF defines a multi-stage boot sequence. The stages are named BL1, BL2, BL31, BL32, and BL33. This sequence is universal for all platforms that use ATF, and is more of a guide how one can implement firmware [2]. In this thesis, the focus is put on what each stage is used for in Enzian's new firmware stack specifically.

ATF also supports distinction between a trusted and non trusted boot, as well as running a trusted OS in a secure execution state. As these features are currently not in use by Enzian[7], details about this have been omitted in the following explanation. See Figure 2 for a visual representation.

- BL1 runs at EL3 and is responsible for minimal platform initialization required by the later stages, as well as loading the next stage: BL2. It runs from the L2 cache until DRAM is initialized. As it is loaded by the internal ROM code, its size is limited to 192KiB.
- BL2 runs at EL1. Its task is to load all later stages into memory.
- BL31 runs at EL3. It has the same privileges as BL1, just without the binary size and memory restrictions. At this point, platform initialization requiring EL3 privilege is finished.
- BL32 is an optional stage running at EL1. It is meant to house a secure OS. At the moment of writing, this feature is not used by Enzian.
- BL33 is the last firmware stage, running at EL2. It can either be an OS bootloader, or (as in Enzian's case) an UEFI implementation, which is later used to load the OS.

It is worth to mention that the ATF implementation that Cavium provided for the Enzian is highly customized. As it runs after the BDK, it has a different execution path. It includes a 'BL0' stage[8] that takes over control from the BDK and loads the devicetree.

[7] Enzian is a research system, so unprotected access to parts usually locked away is desired.

[8] found in `enzian-atf/plat/thunder/bootstrap/`

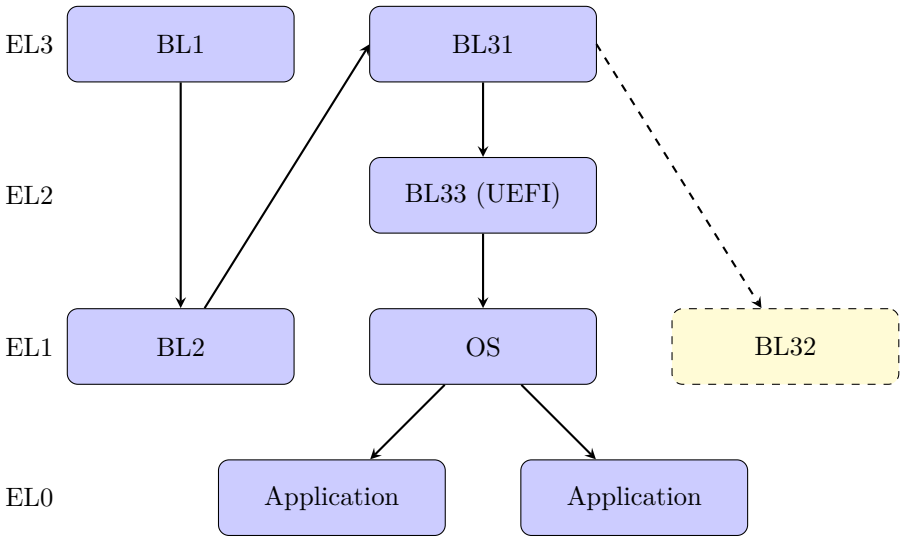

Figure 2: Boot Process under ATF on Enzian

2.11 Previous Work & Current State

Two Bachelor's theses have already been written about this topic. 'Trusted Firmware for a Research Computer' [10] by Alessandro Legnani concerns itself with the ATF, and 'Boot Firmware for Heterogeneous Systems running Linux' [13] by Axel Montini with UEFI.

2.11.1 ATF

The ATF implementation [10] has the following features:

- Serial (UART) initialization
- EFRI initialization and EL3 service installation
- Timer (GTI) initialization
- GIC initialization
- PSCI implementation (partial)
- MMU setup
- Reimplementation of DRAM training

The thesis focused on the DRAM training code. It is able to boot into the next stage, namely UEFI, albeit unreliably. Often the boot process hangs or crashes, at a few different locations. These include trying to switch to execution from DRAM after training, installing runtime services, and next stage loading.

The DRAM code works in principle, but not for the highest speed of $2133MT/s$. Also, the DRAM training takes several minutes to complete[9] each boot, drastically reducing the speed at which changes can be tested.

An open question remains about the DRAM code. The code of the new implementation is an order of magnitude smaller than that of the BDK. It implements lots of features not used by Enzian (different DIMM configurations, DDR3, etc.), but even when taking that into consideration, it seems much more complex and sophisticated than the new implementation. The issues that the new implementation has could be caused by unknown factors, such as hardware intricacies only known to Cavium, or a simple lack of accuracy.
The real reason is hard to determine. One fact discovered while working on this thesis is that the BDK uses a lot of[10] pre-defined configuration values, which look hardcoded after manual testing, to train and configure DRAM. These missing precomputed values could be what is making the difference between the working BDK code, and the only partially working new code.

Because of the lack of good testing infrastructure (no Linux boot possible), there might be a number of undiscovered critical bugs in the new ATF code. In Section 4 some that were discoverd during the work on this thesis are described.

2.11.2 UEFI

The UEFI implementation [13] has the following features:

- PCI driver (partial)
- SATA driver (partial, broken)
- Show Boot Menu

It currently assumes that the ATF

- Initialized memory
- Initialized PCIe, SATA and Serial

There was an attempt to describe Enzian's hardware in ACPI tables, though without an OS to test them, this did not produce tangible results.

Even with the old BDK/ATF, the UEFI currently cannot boot Linux. This is mainly due to the SATA driver not working. In general, the missing coordination between the UEFI and the prior stages makes it difficult to develop working UEFI functionality.

With the new ATF, PCI and SATA do not work at all. This is due to the ATF not initializing SATA, and not configuring PCI in the way that the UEFI requires.

[9] In the thesis, an average time of 4 seconds (!!) is stated for the DRAM initialization process. These measurements could not be reproduced.
[10] about 50

After the required ATF features are implemented, the UEFI can be updated to enable Linux boot. This should not be that difficult, as the EDK II already implements most of the required functionality. This prediction however does not take into account potential unforseen roadblocks.

3 The BDK

As mentioned in the introduction, the Cavium BDK is the first firmware stage used by the vendor supplied firmware stack for the ThunderX platform. Besides the 2000 page hardware manual [5], the Systems Group has access to the source code of all the firmware parts, including the BDK, albeit under NDA. This section explores what it does, how it does it, and the problems that come with it.

3.1 Purpose

The BDK documentation [4] defines the role of BDK as follows:

> The "Bringup and Diagnostic Kit" (BDK) is a set of executables and scripts designed to ease the bringup of new hardware based on Cavium's line of THUNDERX processors. To ease development and increase maintainability, diagnostics are written in a high level scripting language, Lua. [...]
>
> Features of the BDK
>
> - Simple menu driven interface for testing many I/Os.
> ...
> - GUI based interactive script debugger.
> - Lua RPC over serial, tcp/ip, PCIe, or EJTAG.
> - Remote booting over PCIe and EJTAG.

The BDK is many things. It is the first stage boot firmware that is loaded from ROM (Read Only Memory), it initializes hardware and configures DRAM. Beyond that, it provides a BIOS like interactive interface to configure various aspects of the hardware, and allows Lua scripting in every stage of the boot process.

The Lua support is quite involved. Virtually every API function of the BDK, including hardware initialization routines are exposed to the Lua environment. One included App even features a text-based graphical interface with a Lua interactive REPL and debugger, all running within firmware constraints.

The side effect of having so many features in the BDK is high complexity. The whole BDK has more than a million lines of C code[11]. That, in addition to the scarcity of the documentation, and missing code history makes it hard to add new features or understand how even just parts of it work. Note that this comment refers to the inner workings, not the public API.

[11] This number was obtained by using the scc tool by boyter on GitHub in the BDK codebase.

The BDK is meant as a development kit, on top of which Cavium customers can build their own functionality. The documentation mostly describes how the public API works, and how to develop on top of it. Creating custom firmware was clearly not intended to be done by customers.

3.2 Structure

The BDK consists of a number of parts [2].

- `libbdk-arch`: Architecture abstractions, especially CSRs (Configuration and Status Registers)
- `libbdk-dram`: Code related to initializing DRAM
- `libbdk-hal`: Hardware abstractions (mostly initialization code)
- `libbdk-lua`: Lua scripting library
- `libbdk-os`: Operating system like abstractions
- `libbdk-boot`: The application code including the interactive menu and Lua environment

Users of the BDK are supposed to write 'Apps' that utilize these libraries. An App can then be run from the interactive environment, or burned to the beginning of the flash image so that it executes after reset [4]. Writing apps is fairly straightforward, as the BDK offers a rich programming environment (see Section 3.3).

The current Enzian BDK image includes four apps:

- `boot`: Runs after reset, loads the next stage, by default this is `init`. Lets the user choose to run `setup` or `diagnostics`.
- `init`: Performs hardware initialization and tries to chainload the ATF image, falls back to `diagnostics`.
- `diagnostics`: Runs the interactive lua environment.
- `setup`: Provides an interactive configuration interface. Allows changing hardware configuration and save it to flash to persist it.

A BDK image is a FAT formatted filesystem image. Apps are added to the image as binary files, the assembly of the Enzian image is performed by a shellscript[12].

[12] `bdk/bin/bdk-create-fatfs-image`

3.3 Programming Environment

A plethora of high level features is offered by the BDK [4].

- A C standard library implementation, powered by newlib[13]
- Multicore and Thread support
- Simple filesystem abstraction
- Network interface abstraction

The libc implementation newlib includes functions like `malloc` and `free`, which allow BDK apps to dynamically allocate and use heap memory, something that is usually refrained from in firmware for predictability and stability reasons[14].

The threading system is implemented using a 'low overhead cooperative' scheduler. Threads are scheduled using all cores in a round-robin fashion whenever a thread yields by calling `bdk_thread_yield()` [4].

Different network interfaces on the ThunderX are abstracted by a simple common interface that allows enumerating interfaces and ports. Receiving and sending packets is done by one function respectively, allowing easy implementation of transport layer protocols.

This makes programming a BDK application similarly easy as a userspace one. Heap memory can be allocated and freed at will, tasks can run in parallel using the cooperative scheduler. ROM, DRAM, UART and PCIe devices can be read using libc APIs like in POSIX, even arbitrary temporary files can be created and manipulated using the filesystem abstraction [4].

3.4 Configuration & Devicetrees

The BDK has a number of configuration parameters that are specified for each supported platform (the ThunderX or CN88XX being one of those platforms). They describe various hardware options, such as QLM (Quad Lane Module) configuration, TWSI addresses of some devices, and DRAM parameters.

These parameters are stored in the Devicetree format originally developed by OpenFirmware[15]. Devicetrees are well known to Linux developers. Linux uses them to avoid the need to hard-code platform details into the kernel. Instead, a platform can provide a Devicetree, which specifies the available devices, identifies the platform and configuration values [11].

[13] https://www.sourceware.org/newlib/

[14] Many opinions supporting this sentiment can be found in various online forums; A writeup on drawbacks that using heap memory brings can be found here: https://akhileshmoghe.github.io/_posts/embedded/firmware/memory_allocation

[15] https://www.openfirmware.org/Open_Firmware

```
/{
  compatible = "nvidia,harmony", "nvidia,tegra20";
  interrupt-parent = <&intc>;

  memory {
    device_type = "memory";
    reg = <0x00000000 0x40000000>;
  };

  soc {
    compatible = "nvidia,tegra20-soc", "simple-bus";

    intc: interrupt-controller@50041000 {
      compatible = "nvidia,tegra20-gic";
      interrupt-controller;
      #interrupt-cells = <1>;
      reg = <0x50041000 0x1000>, < 0x50040100 0x0100 >;
    };
  };
};
```

Figure 3: Part of the Devicetree used by Linux for the Nvidia Tegra Board [11].

```
/ {
  cavium,bdk {
    ENZIAN = "1";
    MULTI-NODE = "2";

    QLM-AUTO-CONFIG = "0";
    QLM-MODE.N0.QLM0 = "XLAUI_1X4";
    QLM-MODE.N0.QLM1 = "XLAUI_1X4";
    QLM-MODE.N0.QLM2 = "PCIE_1X4";
    QLM-MODE.N0.QLM3 = "SATA_4X1";
    QLM-MODE.N0.QLM4 = "PCIE_1X8";
    QLM-MODE.N0.QLM6 = "PCIE_1X4";
    QLM-MODE.N0.QLM7 = "PCIE_1X4";
    <...>
  };
};
```

Figure 4: Part of the Devicetree used by the BDK for Enzian v3. Source: BDK `/boards/enzian_v3.dts`

Although the BDK borrows the 'Devicetree Source' and 'Flattened Devicetree' file formats as specified in the Devicetree specification [8], the files found in the BDK source tree have little resemblance to the rest of the spec. Compare the excerpts from a devicetree used in Linux (Figure 3) and the one for Enzian used in the BDK (Figure 4). The BDK essentially uses the format as a key-value store for arbitrary configuration values. This means that keeping these Devicetree files around in the new firmware is not a requirement, as they carry no meaning beyond the BDK.

Configuration values are defined as string-string key-value pairs. Some keys are parameterized, for example QLM indices (see Figure 4). The information about value types (integer, string, …), default values, ranges (minimum, maximum for numeric values) and available parameters are all stored in a large lookup table[16]. Parameters are applied using C string manipulation functions, namely `sprintf` that takes the string format specified in the lookup table and inserts the parameters into it to get the final key string. This key is used to look up the value in the list of loaded key-value pairs. The value is cast to its type after its location was determined.
Instead of relying on the programming language features, this system completely circumvents any security mechanisms like type-checking. This makes it highly bug-prone, as well as non-transparent for anyone trying to analyze it.

3.5 Critique

The BDK certainly is a remarkable piece of software. Writing applications on top of it feels like writing a userland application for an operating system. This has obvious advantages: Bringup sequences are incredibly easy to customize, and extensive diagnostic capabilities are available at the lowest level of operation. On the other hand, all of this functionality is not really needed. After a boot flow has been established, it rarely needs to be changed. Even for a research computer like Enzian, it is unlikely that someone would need to do something like changing frequency parameters for PCIe slots at runtime. These adjustments should be done either by flashing a custom image, or through automated systems, like passing parameters to the firmware via EFRI.

While a high-level understanding of the BDK can be achieved without a large time investment, this is not at all the case for the low level inner workings. This thesis is about implementing a new, open firmware stack. Untangling the web of interdependent modules, deciphering the various macros in the code and reasoning about the flow of execution are just some of the challenges when trying to leverage the BDK, which is required, as the manual [5] is lacking information (see Section 6.2 for details).

The sheer size and swath of unnecessary features makes the BDK a bad fit as a firmware component. That is on top of the fact that Arm platforms have a

[16] see `bdk/libbdk-hal/bdk-config.c`

standardized way of doing firmware, the ATF, which the BDK precedes in the Cavium stack. Still, the BDK has the undeniable quality of 'it works', which cannot be said for the new stack that is in development. It will continue to be an important source of knowledge and code for the Enzian platform, at least as until the new firmware stack has been fully implemented and battle-tested for a sufficient time.

4 Implementation

The initial goal of this thesis was to take the previous work on the new ATF [10] and UEFI [13] implementations, fix remaining issues and add missing features to ultimately enable the creation of a working firmware image that is able to boot into Linux.

4.1 Unified Firmware

For the objective of creating an easy to use method of building and extending the new firmware stack, a new repository[17] was created. It includes the new ATF and UEFI repositories as submodules and offers a simple Makefile to build the firmware. The build systems and scripts of the two subprojects have also been surveyed and improved to enable new developers to get into extending them easily.
An effort was also made to improve editing experience, with working support for modern development tools such as clangd.

The documentation can be found in the respective repositories.

4.2 Portability

The original firmware stack was built to support multiple of Caviums platforms. It includes a lot of configuration and code that Enzian does not use or need. The new firmware stack is being developed from the ground up, instead of on top of the existing one. While it reuses parts from the old stack (and of course the base ATF and EDK II), it has its own structure.

Implementing each and every available feature the ThunderX has to offer would take years. Enzian version 3 is a fully specified system[18] and is unlikely to change in the near future, even though a version 4 is on the 'wishlist' frequently mentioned by members of this group. The current Enzian does not utilize the trusted boot features that the ThunderX offers, has a static device configuration (PCIe, SATA, …) and only uses a specific type of DDR4 DRAM modules.

The primary goal is to get the new firmware stack to a point at which it can be used productively, not to support as many features as possible. As such, shortcuts are taken to reduce the time needed to achieve this goal. Configuration options will be limited to those proven to be useful, like DRAM speed. A lot of others, which the BDK had specified in its Devicetree files (see Section 3.4) will be eliminated or hard-coded. Firmware support for unused features will not be provided.

[17] https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2024-bsc-thmeyer/enzian-unified-firmware

[18] see https://enzian.systems/enzian-spec/

The rationale for this decision is that when a new version of Enzian might come along, it will most probably change the CPU as well as other components. In that case, the firmware will need to be largely rewritten, regardless of how portable the firmware for version 3 is.

4.3 DRAM Initialization and the BDK

When assessing the state of the new firmware stack, it was discovered that the from-scratch implemented DRAM training procedure implemented in the new ATF [10] was unreliable and slow.
It must be mentioned here, that it is an incredible feat to create a training algorithm that produces even just somewhat usable results. The DRAM controller of the ThunderX is complex, and there is not much documentation for it. The portion of code in the BDK that is responsible for DRAM initialization is about ten times as large as the new implementation, yet the small codebase was able to demonstrate tangible results. In the future, this code could be expanded upon to create a fully independent training and initialization process.

For now, it was decided that this would be too difficult of an endeavor. Another possibility was considered, namely extracting the code responsible to initialize DRAM from the BDK, and put it into the new ATF implementation as another way of handling DRAM. This would prove to be a difficult task in and of itself.

Fortunately, the DRAM initialization routine in the BDK is somewhat self-contained. There is one function `bdk_dram_config (int node, uint64_t speed)` which takes a NUMA node[19] and a clock speed[20]. This function contains the full configuration code. If all its dependencies are present, this should enable correct DRAM initialization.

4.3.1 Challenges of the BDK

Cavium uses their BDK not just for the ThunderX, but also a number of other devices[21]. Also, it seems to be a Swiss Army Knife of a tool. It is a lot more than just a boot firmware. Besides the user facing functionality, it has a lot of developer features as well.

The BDK houses a dynamically typed string based key-value configuration store mechanism, three different logging systems, and a plethora of smaller utility functions and helper libraries. Basically every part of the code uses these. Naturally, the ATF also provides similar functionality. This has to be considered

[19] Node 0 is the main CPU, and possible DRAM attached to the FPGA is not initialized here, so at this time, 0 is hardcoded.

[20] Note that the clockspeed passed to this function is not MT/s, but rather the actual hardware clock speed. There is a mapping between MT/s and clock speed which can be found in `bdk/libdram/libdram_config_load.c`, but this has not been implemented into the ATF yet.

[21] This is apparent when looking at the source code, lots of references to other platforms can be found.

when trying to integrate the code. Because of strict size restrictions[22] and code duplication concerns, as much as possible of the BDK code should be eliminated and replaced by ATF alternatives.

A challenge during the analysis of the BDK code was its build system. It uses its own custom way of handling dependencies, and instead of detailed headers opts for just one large header containing every interface used in it. This made it difficult to see how different parts of the BDK are connected.

There was yet another concern about including BDK code in the new ATF. That code is under NDA. That means it cannot be published openly without first getting permission from the vendor. Later it was discovered that the relevant parts are already publicly available as part of the coreboot project.

4.3.2 First Approach: Static Library

The first idea on how to reuse the BDK's code was to create a static library from it, that would only contain the symbols needed by the DRAM part. For this, two approaches were considered.

Approach one was to take object files from the BDK build process that seemed like they were needed, and try to strip them of unneeded symbols, then link against them in the ATF build process.

The linking itself was yet another problem. The sophisticated ATF build system is based on thousands of lines of make code, and does not have an obvious way to link a static library. The only way to achieve this that was discovered is to manually add a linker flag to the BL1, which makes it link against the desired archive file.

It was quickly discovered that this would not be feasible. There are hundreds of build artifacts from the BDK, with no clear indication of which of them contain relevant symbols. There is a build artifact with the name `libbdk.a` which (after investigation) seemed like it shoud contain all BDK symbols. Different object file analysis and manipulation tools from the GCC toolchain were used to try and filter out unused symbols, but to no avail. The resulting files would always have undefined references or the final image size was too large. Note again, this code has to be included into the BL1 stage of the ATF, which has a size limit of 192kiB, and a good chunk of it is already occupied with existing code.

Next, the 'App' functionality of the BDK was explored. The BDK allows the creation of applications that can be executed from the interactive environment, or as part of a boot procedure. The rationale was, if the code for the app only references the `bdk_dram_config` function, its resulting archive file would contain only the needed code. This approach was stopped in its tracks, as there is no proper way to use the built app as input to another build process. Not an

[22] At least in the BL1 stage

archive, but a binary image is created, which is probably meant to be jumped into directly instead of being compiled into another program.

The main problem here was binary size. Even the most stripped down version of the mentioned `libbdk.a`, that had missing references to required functions, was by itself larger than 192kiB. Link-time-optimization was considered as a solution for this, as it should strip out any last bit of unused code to reduce the size of the image. However, even though the ATF build system has a built in flag to enable it, it led to various inexplicable issues, so this was not an option.

It was clear that this approach would not lead anywhere. Additionally, the idea of including a large binary object in the source tree of the new ATF was non-ideal in the first place. The source code for DRAM initialization has to be compiled directly into the BL1 from source to allow more aggressive, manual optimization.

4.3.3 Second Approach: Leverage Coreboot

Somehow, the DRAM relevant source code had to be included into the new ATF. As already described, extracting code from the giant BDK source tree that was provided to the systems group by Cavium was not an option.

An idea came to mind. Coreboot (see Section 2.7) is an open source CPU firmware project with support for the ThunderX. Surely, it has to include code to configure DRAM properly. Indeed it does, but this code is non other than that of the BDK.

The coreboot project includes vendor supplied code for a few platforms, ThunderX being one of them. The part of the BDK code found in the coreboot source tree is very similar to the full source the system group has access to, just that it only includes the parts that are actually in use by coreboot. Besides DRAM, there is code for PCIe, USB, NIC, etc. Here, coreboot apparently decided to reuse the hardware initialization code already present instead of re-implementing it. For the mission at hand, only the DRAM part is needed.

Discovering that the coreboot project just re-uses the BDK code again strengthened the assumption that implementing this code from scratch would be a non-worthwhile effort, at least as long as the goal is to create a working system. The topic of DRAM training is one that deserves its own thesis, or even multiple theses.

Even if most of the discovered code was the same as the original, there were a number of modifications made by the coreboot team to make it usable for their project. This came in handy, as a large chunk of work sifting through files, removing unneeded code and making it more stand-alone was already done. It seemed very promising to try and integrate the modified BDK files.

4.3.4 Integration Issues

These files now live in the new ATF repository under `plat/enzian/bdk/`. A great deal of work still had to be done. After including the files into the ATF build system[23], naturally, a great number of compilation issues emerged. Dependencies on coreboot specific functionality were a problem, these had to be replaced by ATF equivalents.

Notably, TWSI functionality (which is needed to talk to the DRAM controller) is implemented by coreboot directly, and the existing ATF equivalent works in a different way than what the BDK code expects. To resolve this issue, the TWSI code was taken from the original BDK source.

Aside from many waves of compilation errors, an effort was made to fix all compiler warnings generated by the BDK code. In its current state, the ATF code with the BDK included compiles without any issues or warnings. This was a lengthy process. Unused files had to be removed on a trial-and-error basis, name clashes had to be resolved. A number of utility functions were eliminated, as equivalents are provided by the ATF. The three different logging mechanisms used by the BDK code were replaced by or aliased to ATF logging functions to allow easy logging configuration across the whole codebase.

One large blocking issue was again binary size. After fixing all the mentioned issues and removing a number of redundant code, the BL1 image would still exceed the size limit of 192kiB. Initially, in the build configuration the size limit was set to just 131kiB, so the first action was to increase it to the full 192kiB available[24]. Even then, the image would exceed the limit. Link time optimization would not work, so that was not an option to reduce the size. A compile-time switch to select which DRAM initialization method (from-scratch or BDK) should be used was added, but did not help reducing image size. The compiler already optimized out the unused code. Somehow the amount of code had to be reduced.

4.3.5 The BDK Config Mechanism

One puzzling aspect about the BDK code throughout the work up until this point was the config mechanism. It allows access to the Devicetree of the board currently in use. The values are loaded from the '.dtb' (device tree blob) file specific to the current board[25]. The Devicetree specifies options for different parts of the system, including QLM modes, TWSI addresses, and DRAM controller parameters.

[23] This fortunately was quite straightforward, as this task just consisted of listing them as additional sources for the BL1 in the relevant `Makefile`.

[24] Actually 188kiB, as the first 4kiB page is not available as it is used for a header.

[25] These files are compiled from '.dts' (device tree source) files present in the BDK source under '/boards'

There is code to load a '.dtb' file, validate it, and make it available to the `bdk_config_xxx` functions. To retrieve values, options are addressed by their string name and additional optional parameters (e.g. to select one of the four memory controllers). Lots of string formatting and comparison is needed to retrieve a simple integer value. Not to mention, there is a giant table specifying each options name, type, parameters, default value and, optionally, value range.

As the new firmware does not use these devicetree files, it was a natural step to try and remove as much of this code as possible. The only part of it necessary for the DRAM initialization are the options loaded by `libdram_config_load.c`. In that file, all DRAM relevant options are retrieved from the configuration system, and are put into a struct `dram_config_t`. Apart from this, the DRAM code is not dependent on the configuration system.

Code size had to be reduced. To achieve this, it was decided to include a populated `dram_config_t` struct in the source, instead of loading values into it dynamically. This allowed the elimination of the whole configuration system. The populated struct was extracted from the stable firmware stack, by printing out its values after it was loaded[26].

4.3.6 Achieving DRAM Initialization

After the compilation issues and size limitations were overcome, the DRAM initialization routine extracted from the BDK code included in coreboot was functional. Only minor points had to be addressed.

The first point was correctly handling memory mapping and locking/unlocking the scratchpad. Because the BDK DRAM initialization code is used outside of the BDK, none of the required setup is done. Fortunately, the new ATF already included the required steps. Before running DRAM configuration, the scratchpad in the L2 cache needs to be locked, while the rest of memory needs to be accessible as secure memory. The DRAM configuration code will read and write from various locations in memory, including address zero. After configuration has concluded, the scratchpad can be unlocked and memory can be set up properly.

Another issue emerged during testing. The boot process was overwriting seemingly random unrelated variables during execution. It turned out this was a stack overflow issue. Increasing the stack size from 0x1000 to 0x2000 fixed this problem[27].

[26] The code for this extraction can be found here: gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-bdk/-/tree/extract-dram-training-config

[27] This can be adjusted in `plat/enzian/include/platform_def.h`

4.4 Loading into UEFI

After enabling proper DRAM configuration, there was still one bug left that was preventing the new firmware stack to reliably load into UEFI. During most executions, the ATF would hang trying to initialize runtime services[28]. These services include the Power State Coordination Interface (PSCI), which relies on a so called power domain tree description to be provided by the platform code[29]. More information on PSCI can be found in section 3.12 of Alessandro Legnani's thesis [10]. The power domain tree description he wrote was not tested, so the bug was not detected. The power domain tree is described by an array of unsigned integers, each representing a power domain. Each entry in this list specifies how many children the respective node has, so there needs to be one number for each node. In the description found in the code, the numbers for some of the child nodes were missing, which led the PSCI code to try to configure power domains that did not exist, because it was reading values out of bounds of the description array. With this fixed, the new ATF is able to consistently load into UEFI.

4.5 GSER/QLM & SATA Initialization

A crucial part in achieving a successful Linux boot is to be able to load a Linux kernel. This can be done from a disk or from the network. Because there was already preliminary work done on trying to get SATA (Serial AT Attachment) disks to work [13], this route was pursued.

The SATA driver in the UEFI implementation already did not work properly (see Section 2.11.2). This is, on top of the fact that there is no prior work on SATA initialization on the ATF side. What seemed like just a bugfix away is actually much more work than anticipated.

To initialize a SATA controller on the ThunderX, one has to first initialize and configure the GSER (General Serializer/Deserializer Unit)[30]. It manages QLMs, each of which is a four data lane interface that can have multiple different purposes. The ThunderX has 16 such QLMs. Six are reserved for CCPI™, QLM0 and QLM1 can be configured for miscellaneous protocols, the remaining ones are for SATA and PCIe. For Enzian, only QLM3 is connected to a SATA controller.

The process of configuring a GSER for SATA mode is complex. The manual explains it step by step, though some of the step descriptions require knowledge from elsewhere, e.g. 'Configure the reference clock'. There are multiple options here and no indication as to which to choose. Similar problems appear in other instructions.

[28] This is part of what the ATF provides out of the box.
[29] See section 5.6.2 in the ATF documentation [2]
[30] Find detailed information in Chapter 25 of the ThunderX manual[5]

Being familiar with the BDK code after analyzing and incorporating the DRAM part of it into BL1, I chose to try the same for the SATA initialization code. In the end it is a sequence of operations that has to be performed to properly initialize the device, and the BDK code is proven to work, so reinventing the wheel does not seem reasonable. Fortunately, the different hardware initialization routines under `libbdk-hal` are somewhat independent and were thus easier to bring into the ATF and compile them.

Some investigation of the BDK boot flow showed which api functions were called, so all that had to be done was to grab the relevant files from the BDK, bake them into BL31 and call the necessary functions in the setup routine of BL31. Here we do not have to deal with the size constraints, so that is another obstacle out of the way.

Unfortunately, this alone was not enough. Naively calling the BDK functions from BL31 just resulted in invalid memory accesses. Calling the routine before and after the MMU was enabled moved the crash to a different place in the code. What is missing here is presumably correct memory mapping that allows access to the relevant CSRs. The correct mapping was not yet figured out, nor could it be confirmed that this was in fact the issue causing the crash. It might as well be that the BDK code does something that is not possible in the ATF environment. Thorough investigation and debugging are needed here to find the issue. The time constraints of this thesis prevented me from investigating further myself.

5 Evaluation

Since the DRAM initialization was implemented and the mentioned issues were fixed, the ATF implementation has not exhibited any unstable behavior. The runtime is consistent and similar to the time the BDK takes in the old stack. This is expected, as the DRAM training takes up the majority of time, and now both essentially run the same routine for that. DRAM is working, as far as a preliminary test could confirm (see next paragraph). Full testing will be possible once proper memory testing software like Memtest86 can be run.

The simple memory test was conducted as follows: In BL2, an MMU entry was added to map addresses from 0x1000000 (after any code or data sections) to 0x2000000000 (128GiB boundary) as non-cacheable direct access RAM, and then different locations within this region are tested by writing a value to them and reading it again, checking if they are equal (see the code in Figure 5). None of multiple runs of this test showed any failures. Curiously, running this test in BL2 makes a later part of the ATF code panic repeatedly, presumably because it overwrites some important data.

The new ATF implementation can load the new UEFI implementation consistently, which then crashes because of missing initialization steps described above. Average execution time of the ATF until it loads into UEFI is about 10 seconds.

The code from the BDK that was imported into the ATF consists of 15 000 lines of C source code and 140 000 lines of C header code, both numbers are without comments. This is a considerable amount, but still much less than the full BDK. The binary size of the BL1 with the BDK DRAM code now is close to the 192KiB limit imposed by the boot ROM loading routine, so if further expansions of this stage are necessary, this needs to be revisited.

```
--- Memory map entry added to plat_enzian_mmap for BL2:

MAP_REGION_FLAT(0x1000000, 0x2000000000-0x1000000,
  MT_MEMORY | MT_RW | MT_NS | MT_NON_CACHEABLE),

-- Memory test code

void test_mem(uint64_t start, uint64_t end) {
    uint64_t stride = 1024 * sizeof(uint64_t);
    uint64_t test_data = 0x1234567890abcdef;
    uint64_t successes = 0;
    uint64_t failures = 0;

    INFO("Begin Mem Test\n");
    for (uint64_t addr = start;
      addr < end - sizeof(uint64_t);
      addr += stride
    ) {
        volatile uint64_t *ptr = (uint64_t *)(addr);
        *ptr = test_data;
        uint64_t result = *ptr;
        if (result != test_data) {
            ++failures;
            WARN("Invalid value read during mem test at"
                 "memory address %lx, "
                 "expected %lx, got %lx\n",
                 addr, test_data, result);
        } else {
            ++successes;
        }
        ++test_data;
    }
    INFO("Mem Test done, Success: %ld, Failure: %ld\n",
      successes, failures);
}

--- Output from the Boot log

INFO:    Testing 0x1000000..0x2000000000
INFO:    Begin Mem Test
INFO:    Mem Test done, Success: 16775168, Failure: 0
```

Figure 5: Memory testing code and log output

6 Discussion

6.1 The Problem with Firmware

Abstraction is an indispensable tool when building computers. It allows the usage of highly complex systems through iteratively simpler interfaces. However, abstraction can also become a burden.

6.1.1 Independent Layers

When different teams develop the different layers of firmware, lack of cooperation necessitates the introduction of abstractions. The hardware vendor is expected to deliver a firmware for their chip that correctly initializes all the hardware and allows loading an operating system. The operating system requires certain information about the hardware to be able to work. In Linux for Arm, this can be done by supplying a Devicetree [11] or via ACPI (Advanced Configuration and Power Interface) tables [12].

Traditionally, Arm was primarily used in embedded systems. These systems would only ever run one OS, shipped with the device. Later, Arm chips were used in servers and personal computers, where x86 was the dominant architecture.

For Arm systems, Arm developed the ATF as a standardized firmware implementation. It supports every step from ROM to loading user software like a Linux kernel. In practice, instead of loading a Linux kernel directly, software like U-Boot[31] is used to enable loading user supplied software from other sources such as the network or various filesystems.

In the x86 world, UEFI and ACPI were invented to combat the lack of standardized firmware on x86 machines. Regardless of how the chip looks and what the firmware does, it loads a UEFI implementation, which exposes ACPI tables to the operating system as a well known interface.

Here lies the first problem. UEFI was designed by people primarily interested in writing the operating system part, while the firmware is developed by people designing hardware. This discrepancy leads to separated ecosystems that know almost nothing of each other. The firmware has to do a lot of hardware initialization to enable UEFI to work correctly, while the UEFI also has to do hardware configuration to make sure the operating system finds the system in a well known state. Both parts have to implement abstractions for virtual memory, hardware configuration registers, etc. Both parts have their own code structure, style and principles, as well as security standards.

From the OS' point of view, the system looks well defined and well behaved. The hardware does exactly as it is told, no undetectable errors occur. It has to trust the hardware. Here we find the second problem: Even trusting the hardware

[31] docs.u-boot.org

is not sustainable. Cloud computing hyperscalers acknowledged this and are implementing various measures to make their hardware more trustworthy, see e.g. Google [9].

It is not just the hardware we have to trust. At every step of the way towards running user software we are looking at components we cannot trust. The boot ROM, the firmware, the UEFI, the kernel; all of these are potential points of failure. The kernel is presumably the most well tested part of this chain. The lack of trust gives birth to a recurring theme: Building another layer on top of the tower of untrusted ones.

6.1.2 ThunderX Firmware

Now enter the firmware stack for the ThunderX. It is an Arm processor, so it should have firmware based on ATF. This however, is not the case. We have a custom, hardcoded boot ROM, that loads 192KiB of code into the L2 cache as the first firmware stage. This is part of the BDK, which then loads the rest of itself, runs initialization code and chainloads an ATF implementation. The ATF loads yet another stage, the UEFI. Only then we can finally venture into user software like Linux.

What do we need Linux for? To abstract away the hardware, virtualize and schedule user programs and provide a runtime environment for easy creation of such programs. Does this sound familiar? Remember the features of the BDK described in Section 3. We have a filesystem, network, multithreading and a notion of 'Apps'. Apps use the rich environment provided by the BDK to perform arbitrary tasks, in this case the task is ultimately loading the ATF. In that moment, the whole runtime, all the abstractions, all the code is thrown away, making way for the next complex system in line. The ATF leaves behind the secure monitor to be used by the operating system later, but apart from that the same thing happens again when it loads the UEFI with its own set of abstractions.

Especially the BDK seems like it was not designed to be just a firmware stage. It could be an artifact from the days that Cavium was making smart NICs (Network Interface Controllers), a complete programming environment, allowing arbitrary applications to run on top of it, without ever loading a more complex operating system. It is a complete, full stack solution for system software, not just firmware. It alone could just load a Linux kernel, without any of the later stages.

The existence of the other two stages is the result of abstraction expectations. The ATF is needed to provide the secure monitor, and the UEFI to publish ACPI tables and UEFI runtime services. These are not required to exist per se, they could as well be replaced by some other mechanism. It is simply standards stacked on top of each other that were not meant to be stacked.

6.2 Implementing a new Firmware Stack

In Section 4, the process of implementing just a small subset of features needed for a feature complete Enzian firmware is described. Learning about the three components of the old stack, about the BDK source code intricacies, and the ATF structure took the majority of time available for the creation of this thesis.

The two theses that did work on this endeavor before both did not complete their respective objectives fully due of size constraints, same as this thesis. Reading these two theses suggests that the work is almost complete, just a few tweaks and missing features, and then we are ready to switch to the new firmware stack. This is a fallacy. It turns out that, what seems at first like a minor bug or small missing feature can turn into a huge undertaking.

In Axel Montini's thesis [13], he states that he was unable to get a SATA driver to work. This is while using the old BDK/ATF stack to load his UEFI. SATA device enumeration was achieved shortly after the end of that thesis. However, there was no realization of how difficult it would be to bring SATA initialization to the ATF, which would be required for the SATA implementation in UEFI. The ThunderX manual gives a list of twenty steps that have to be taken in order to initialize a QLM to SATA mode (refer to ThunderX manual Section 25.1.1.2 [5]). These in turn require another ten steps (Section 27.4 in the manual), the first one reading 'Ensure that the SerDes reference clock is up and stable'. No further explanation. What is the reference clock? How do I configure it? What does 'stable' mean?

The deeper one goes into trying to understand the processes that the old firmware goes through, the more confusing it gets. Implementing this anew is a huge undertaking. Every component of the ThunderX is complex and will require a great deal of labour to get working.

It gets even more difficult to re-implement firmware features, when they have to work around undocumented quirks in the hardware. One example of this is the QLM initialization, where the function `__bdk_qlm_tune` is described with 'Some QLM speeds need to override the default tuning parameters'. In this function, one can find magic numbers for certain parameters that seemingly stem from internal email communications at Cavium[32].

The effort required to create a new firmware that is on-par with the old one feature wise will be a long process. It can be expected that every part of the process will bring similar difficulties as those encountered while implementing the features described in this thesis. The large features missing at the time of writing are SATA and PCIe, USB and Network. Additionally, backporting of features that were added to the old firmware stack by the systems group, and potential changes to how the OS is loaded (e.g. describing Enzian hardware via ACPI instead of a Devicetree) have to be considered before a new firmware stack could be used in production.

[32] bdk/libbdk-hal/qlm/bdk-qlm-common.c, Line 1371

Dividing this work into potential future Bachelor's theses is of questionable feasibility. The two theses that preceded this one [10][13] set up an environment and implemented the features that were already largely supported by the provided code in the respective frameworks (except for the DRAM code in the new ATF). This thesis tested how existing code could be reused instead of reinventing the wheel.

Given enough time, a working firmware stack could be created. If every future Bachelor student has to spend months on first understanding all the intricacies of the process, the hardware, and the existing code before being able to do anything useful, I project that it will require about three to five more Bachelor theses before a new firmware stack is ready for use. Additionally, future Bachelor students might find it hard to write about their learnings, because most of the details are already described in this and the two preceding theses. Yes, by reading the theses a lot of the learning process is of course greatly accelerated, but even with a lot of material at hand it is difficult to delve into this kind of firmware development, given no prior experience.

7 Future Work

The overarching goal of this project is to create a new, open firmware stack for Enzian. The following aspects are not implemented or do not work as of now:

- Full PSCI implementation and soft reset
- QLM/GSER initialization
- SATA initialization and drivers
- PCIe initialization and drivers
- USB initialization and drivers
- NIC initialization and drivers
- Extensive configurability using the BMC and EFRI

All of these items will require a significant time investment to be implemented from scratch. A way to speed up development, while also ensuring feature-parity with the old firmware, is to continue with the approach this thesis took: Understanding and adapting the relevant parts of BDK code.

To get to a testable system quickly, it may be a good strategy to focus on implementing SATA or NIC functionality, to be able to load Linux from disk or the network. This step of course contains another project: Correctly describing the hardware to the Linux kernel. After Linux loading was achieved, implementing more functionality can be done with more confidence.

The BDK contains definitions for the ThunderX hardware registers, and code to initialize all of the systems listed above. Step by step, one of them can be isolated, understood, and incorporated into the ATF codebase as part of BL31, similarly how it was attempted with the SATA code in Section 4.5.

This way, after working out incompatibilities and bugs, the ATF will be able to initialize the hardware like the BDK does. The next step is then to analyze the old ATF and port required functionality over to the new implementation. The same has to be done with the UEFI. Contrary to what the proposal for this thesis said, I would advise putting as much functionality as possible into the ATF, and keep the UEFI layer minimal. The ATF provides enough high level features to implement any desired feature comfortably, and is much more approachable than EDK II.

One should avoid trying to implement hardware initialization from scratch, solely based on the manual [5]. Either the existing BDK code is ported over to ATF, or a new implementation closely follows it. That way, many unforseeable surprises and bugs will be avoided. Section 6.2 describes why this is important.

A lot of work has to be done to ensure reliability of the firmware. Currently, virtually all features present in the firmware are barely tested, apart from the 'it does not crash' metric. It might prove useful to implement a rigorous automated testing system for each component.

There is one more idea on how to simplify the firmware. Technically, a full UEFI implementation is not a requirement to run Linux on Arm. Instead of relying on the complicated EDK II, a simpler component like U-Boot[33] could be used. U-Boot contains code that suggests support for the ThunderX, although this is not listed in the official documentation. An email exchange with the maintainer listed in the code confirmed that the support implemented is for U-Boot as an ATF BL33 payload.

Some initial, naive experiments did not show any success. It was tried to flash different build artifacts U-Boot produced when compiled with the `thunderx_88xx_defconfig` configuration as the BL33, using the scripts that are in use for the UEFI. This probably failed to load properly or jumps to the wrong location, making the boot process fail.

If with some more experimentation U-Boot loads and successfully starts Linux (with the old BDK/ATF stack), this means we can potentially eliminate the need to write any custom code outside of the ATF entirely. All initialization logic can live inside the new ATF, and U-Boot is leveraged to handle correctly loading the Linux kernel.

[33] docs.u-boot.org/

8 Conclusion

The most evident conclusion to be drawn from this thesis is that developing firmware as a non-insider is much more difficult than it seems to be at first glance. It is very difficult to judge how long the road to success really is, even with the source code of the old firmware and all available technical documentation.

After decades of proprietary patchwork in the field of firmware, openly accessible information about the process of writing firmware is scarce. The best places to find examples are projects like coreboot and U-Boot, which both support a variety of platforms, although the reality is that support for platforms is usually contributed by the vendor, without much accompanying explanation.

A fully understood firmware rewrite for Enzian would greatly benefit its purpose as a research platform. However, this goal seems further away than previously thought. There is a lot left to analyze and implement. The further into hardware intricacies one gets, the more unforeseen hurdles appear. This makes it impossible to accurately estimate the required effort.

This thesis did achieve reliable DRAM initialization at all supported speeds in the ATF stage, as well as consistent loading into the next boot stage, namely UEFI. Effort was made to get SATA to work, but this was not completed. However, a concrete plan for how to proceed with the project was presented in Section 7. For the future of this project it should be reconsidered wether assigning more bachelor theses or semester projects is a viable way forward, as the time investment needed to get accustomed to the Enzian firmware context before being able to conduct productive work is considerable. Lastly, while implementing a completely new firmware stack, the possibility of making more radical changes that could simplify development, like replacing EDK II with U-Boot, should be considered.

Enzian is a great platform to research firmware. The struggles outlined in this thesis prove that there is much to learn from it. The more research is conducted, the easier it will be in the future to build on top of this knowledge.

Glossary

ACPI Advanced Configuration and Power Interface; A common interface to device configuration and power management between an OS and UEFI.. 32, 33

Arm A processor and software design company, primarily focused on developing Arm architecture instruction sets.. 7, 9, 20, 32, 33, 37

ARMv8.1 The 64-bit Arm instruction set used by the ThunderX.. 7

ATF Arm Trusted Firmware; Officially 'Trusted Firmware-A'.. 5, 6, 9–15, 17, 20, 22–30, 32–38

BDK Bring-up and Diagnostic Kit; First stage firmware for the ThunderX, developed by Cavium. Used in the old firmware stack and source of the DRAM training and configuration code.. 5, 6, 10–12, 14, 16–18, 20–30, 33, 34, 36, 37

BMC Baseboard Management Controller; A small computer embedded onto server Motherboards. It is used to control various aspects of the machine, such as power management and hardware configuration.. 7

CCPI™ Cavium Coherent Processor Interconnect; The cache coherency protocol of the ThunderX for dual-node NUMA setups.. 7, 28

CSR Configuration and Status Register. 17, 29

DRAM Dynamic Random Access Memory; The main memory type used in a typical computer.. 7, 8, 10–14, 16–18, 22–28, 30, 38

ECI Enzian Coherent Interconnect; The cache coherency protocol used to connect the ThunderX and the FPGA on an Enzian board.. 7

EDK II EFI Development Kit 2, the UEFI reference implementation. Enzians UEFI is based on this.. 11, 15, 22, 36–38

EFRI Enzian Firmware Resource Interface; An interface allowing the firmware on the ThunderX to communicate with the BMC. One usecase is to provide configuration to the boot process.. 7, 13, 20

FPGA Field Programmable Gate Array; A device containing programmable logic gates. Can act like a real microchip, often used for rapid prototyping and testing of new chip designs.. 7

GSER General Serializer/Deserializer Unit. 28

NIC Network Interface Controller. 33

NUMA Non-Uniform Memory Access; A computer memory design used in systems with multiple processors. The time to access a memory location depends on where it is located (local to the current processor/node, shared, or local to another node).. 7, 23

PCIe Peripheral Component Interconnect Express; A standard to connect peripheral devices to CPUs, such as graphics cards or storage devices.. 7, 20, 28

QLM Quad Lane Module; Used for CCPI, PCIe, SATA or others. One QLM provides 4 data lanes (e.g. PCIe x4).. 18, 28, 34

ROM Read Only Memory. 16

SATA Serial AT Attachment; An interface to connect storage devices.. 28, 36, 38

scratchpad A region in the ThunderX's L2 cache that is used to run code from before DRAM is initialized.. 11, 12

SMC Secure Monitor Call; A mechanism that allows calling into predefined routines running at EL3.. 7, 9

ThunderX Cavium® ThunderX™ (CN88XX), the 48 Core Arm AArch64 Processor used in Enzian. 5, 7, 8, 10, 11, 16, 18, 22, 23, 25, 28, 33, 34, 36, 37

TWSI Two-Wire Serial Interface; Also known as I2C (Inter-Integrated Circuit). 8, 18, 26

UEFI Unified Extensible Firmware Interface. 5, 6, 9–11, 13–15, 22, 28, 30, 32–34, 36–38

References

[1] Arm. *Learn the architecture - AArch64 Exception Model*. 2022. URL: https://documentation-service.arm.com/static/63a065c41d698c4dc521cb1c.

[2] Arm. *Trusted Firmware-A Documentation*. URL: https://trustedfirmware-a.readthedocs.io/en/latest/ (visited on 09/15/2024).

[3] Bryan Cantrill. "I've come to bury the BIOS, not to open it." In: Talk given at the Open Source Firmware Conference 2022. URL: https://www.osfc.io/2022/talks/i-have-come-to-bury-the-bios-not-to-open-it-the-need-for-holistic-systems/.

[4] Cavium Inc. *BDK Documentation*.

[5] Cavium Inc. *Cavium ThunderX CN88XX, Pass 2 Hardware Reference Manual*. URL: https://unlimited.ethz.ch/download/attachments/115093207/CN88XX-HM-2.7P.pdf?version=1&modificationDate=1652866562443&api=v2 (visited on 10/02/2024).

[6] David Cock et al. "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 2022. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 434–451. ISBN: 9781450392051. DOI: 10.1145/3503222.3507742. URL: https://doi.org/10.1145/3503222.3507742.

[7] *Coreboot Documentation*. URL: https://doc.coreboot.org/ (visited on 09/21/2024).

[8] *Devicetree Specification v0.4*. June 28, 2023. URL: https://www.devicetree.org/specifications/.

[9] *Google Infrastructure Security Design Overview*. URL: https://cloud.google.com/docs/security/infrastructure/design/resources/google_infrastructure_whitepaper_fa.pdf (visited on 10/24/2024).

[10] Alessandro Legnani. "Trusted Firmware for a Research Computer". Aug. 2023. DOI: 10.3929/ethz-b-000634201. URL: https://doi.org/10.3929/ethz-b-000634201.

[11] *Linux and the Devicetree*. URL: https://www.kernel.org/doc/html/latest/devicetree/usage-model.html (visited on 10/07/2024).

[12] *Linux Kernel Documentation: ACPI on Arm systems*. URL: https://www.kernel.org/doc/html/latest/arch/arm64/arm-acpi.html (visited on 10/24/2024).

[13] Axel Montini. "Boot Firmware for Heterogeneous Systems running Linux". Aug. 2023. DOI: 10.3929/ethz-b-000634202. URL: https://doi.org/10.3929/ethz-b-000634202.

[14] *Tianocore Website*. URL: https://www.tianocore.org/ (visited on 09/15/2024).

[15] *Unified Extensible Firmware Interface*. URL: https://uefi.org/ (visited on 11/09/2024).

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

- (●) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].
- ( ) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].
- ( ) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Towards a Unified Firmware for Enzian

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Meyer-Lehnert	Thomas Harald

With my signature I confirm the following:

- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zürich, 11.11.2024	

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard