Boot Firmware for Heterogeneous Systems running Linux

Bachelor Thesis

Author(s):
Montini, Axel

Publication date:
2023-08

Permanent link:
https://doi.org/10.3929/ethz-b-000634202

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor’s Thesis Nr. 465b

Systems Group, Department of Computer Science, ETH Zurich

Boot Firmware for Heterogeneous Systems running Linux

by

Axel Montini

Supervised by

Daniel Schwyn
Prof. Dr. Timothy Roscoe

February 2023 - August 2023

D INFK

Abstract

The primary role of the boot firmware is to initialize essential system components before booting an operating system. At the same time, it should provide hardware descriptions and interaction mechanism through the UEFI and ACPI standards.

The current boot firmware for Enzian is unmaintainable due to its outdated design and reliance on an obsolete EDK2 version, without any obvious way to update it. Consequently, a complete rewrite is necessary.

This thesis undertakes the development of a boot firmware from the ground up. Leveraging the latest version of EDK2, a UEFI implementation, the objective is to build a solid foundation that's both comprehensible and easily maintainable.

The firmware takes inspiration from existing platform ports of EDK2, resulting in a greatly simplified development process, which is further improved by utilizing the containerized build system.

The implementation produced, even though not yet able to boot an OS from disk, aligns with established industry practices and aims to be a well-structured base.

Acknowledgements

I would like to express my gratitude to Daniel Schwyn for his essential feedback and support throughout the past six months, and to Prof. Dr. Roscoe for the great opportunity to work on this project.

At last, I thank my family and friends for any help they provided during the semester.

Contents

1 Introduction

The boot process involves a lot of software in modern days. There's a lot of hardware to be initialized and the differences between similar components can be significant. Nonetheless, the same operating system must be able to run seamlessly on vastly different hardware configurations.

The most obvious way to achieve this is to define standard ways for the OS to interact with the firmware and hardware.

For this reason, the UEFI and ACPI specifications were developed. They introduce a standard boot sequence, an interface between the OS and the firmware and a way to describe the hardware and how to use it.

These two have been the standard on x86 machines for quite a while, whereas ARM computers have only recently started to adopt them: in the past, Linux only supported Flat Device Trees (FDTs) on ARM and lacked a complete ACPI support [2].

Since FDTs are only a way to describe how the hardware is structured, the current operation of devices is entirely a responsibility of the kernel, even down to power management. Interfaces to interact with the hardware and for platform and power management are instead provided by ACPI with the ACPI Machine Language (AML) [6]. This language can describe the devices within a system, their properties and the functions that they support, while being platform-independent. An OS supporting ACPI can then interpret this byte code to use the devices described, effectively replacing the platform-specific code in the kernel that achieved this.

Given all the advantages that ACPI can provide, ARM has decided that it should be preferred over FDTs on servers that implement Server Base Boot Requirements [10].

Enzian already had a UEFI implementation written by Cavium, though it required a full rewrite: the repository's history is not available, the structure is outdated, the documentation is lacking and EDK2 has changed quite a bit in the meantime. In my opinion, the burden of maintaining such project or porting it to a new EDK2 version is unjustifiable, as most of the stuff needs to be rewritten or heavily adapted.

Similar to the previous implementation, the UEFI firmware is booted by the Arm Trusted Firmware (ATF), which should be as small as possible and contain just enough code to accomplish its goals. This was not the case for the old ATF implementation on Enzian: together with the BDK, it was responsible for the initialization of many components that were only required in the later boot stages, such as the PCIe controllers and SATA controllers. This is undesirable, since the UEFI as a higher footprint anyway and the PEI boot phase (introduced in the next section) can be used to perform platform initialization.

In contrast, the new ATF implementation developed by A. Legnani [13] does not perform such tasks: this means that it is now the UEFI's responsibility to initialize the needed hardware. Unluckily, the project required more time than anticipated, thus the current UEFI implementation doesn't yet contain modules to set up GSER, PCI and SATA. As such, it requires the use of the old ATF in order to use said functionality, until the relevant code gets ported.

This report explains the reasoning and the process behind the development of the new firmware which, even tough still incomplete, aims to be a solid base for future projects, by attempting to closely follow current approaches in platform implementation.

2 Background

2.1 Arm Trusted Firmware (ATF)

The Trusted Firmware-A (TF-A) [15] is a reference implementation of secure software that sits before UEFI in the boot process. It was formerly known as Arm Trusted Firmware (ATF) and in the document I will refer to it by this older name.

Its tasks are to create a trusted computing environment and to enable secure system initialization before the hand-off to UEFI (or other bootloaders). It consists of several stages (BL1, etc.), the last of which is BL33, the bootloader. In our case this stage is EDK2 (introduced later), but alternatives like U-Boot are available.

2.1.1 Secure Monitor Call (SMC)

The ATF also needs to handle *SMC*s (Secure Monitor Call), conforming to their calling convention. These calls can be used by later boot stages to interact with hardware and the ATF. The `SMC` instruction generates a synchronous exception that is then handled by Secure Monitor code running in EL3. Registers are used to pass arguments and return values.

One example of the currently available SMCs is one to interact with the Enzian Firmware Resource Interface (EFRI) [19].

2.2 UEFI

The Unified Extensible Firmware Interface (UEFI) provides an interface to allow interaction between the OS and the firmware. It provides data tables containing platform information and service calls. It is only a specification, with the goal to standardize these interactions. There are implementations of course, one of which is EDK2, described later.

2.2.1 Components

Its main components are the following, each serving a distinct purpose:

Boot Manager The boot manager is responsible for managing the available boot options, enabling the user to select the desired boot configuration.

Boot Services The Boot Services provide a set of runtime processes that are available during the boot process, including but not limited to: memory allocation and file system access.

Runtime Services Similar to Boot Services, but are available *after* the OS has started, providing a way for applications and drivers to interact with the firmware.

UEFI Drivers These provide initialization and management functions, enabling the firmware to interact with various hardware components, such as storage devices and network interfaces.

UEFI Applications Standalone programs that can run directly from the firmware environment. These are commonly diagnostic tools and configuration interfaces.

2.2.2 Boot Stages

The UEFI specification defines the following boot stages, whose implementation can vary greatly: for example, some platforms might skip SEC phase completely, and the modules used in the PEI phase change depending on the architecture (e.g. the Arm-specific `ArmPlatformPkg/MemoryInitPei/MemoryInitPeim.inf`).

SEC (Security Phase) First stage in UEFI boot, but may be preceded by other binary code. Usually set up the stack and verifies PEI before hand-off. In the past it also used to be responsible for a lot of the tasks that the ATF now performs, thus it has considerably shrunk.

PEI (Pre-EFI Initialization) The PEI phase is responsible for setting up the (minimal) components required for the subsequent phases. It usually initializes main memory, a serial port, handles the case in which the system is waking up after sleep.

This phase also loads PEI modules (PEIMs) and dispatches them in a dependency-aware way: each PEIM can specify its dependencies, and it's executed after they are met, in a non-deterministic order (decided at runtime). They also inherit dependencies from the libraries they link to.

DXE (Driver Execution Environment) In this phase, a dispatcher executes DXE modules after checking their dependencies. During this phase, needed devices are initialized by DXE drivers (e.g. SATA Controllers, network interfaces, ...).

BDS (Boot Device Select) It's part of DXE and just processes boot options, initializes boot devices and starts executing them.

TSL (Transient System Load) Stage before OS hand-off. Allows one to enter an UEFI shell or run an application.

RT- Runtime UEFI hands-off to the OS. The OS is responsible to exit boot services to free all memory that is not needed anymore. Usually the OS uses its own drivers to control hardware devices. Some runtime services, such as ACPI, remain.

2.3 EDK2

Tianocore EDK2 (EFI Development Kit) is the reference implementation of UEFI by Intel. It contains various libraries and components to aid in the implementation of UEFI for a specific platform.

Most components are platform-agnostic, as they are designed to depend on pre-defined APIs. These APIs must then be implemented by custom platform-specific libraries (e.g. `PciHostBridgeLib`).

At the time of writing there are many platform implementations publicly available online[26], but none for Enzian.

By browsing through the pre-existing implementations (including the old UEFI firmware by Cavium), it's apparent that the standard practice is to modify a pre-existing platform - usually ARM VExpress - to implement new ones. It's sensible, as a big set of libraries and components is shared by all platforms; this way, the package structure is kept similar.

While the build system is quite flexible, it lacks meaningful IDE support, making editing a bit more complex than it could be if the configuration files were automatically parsed.

Now let's define some EDK2 terminology, useful to understand how the project is structured and how the build system works.

Package

Each package can contain other packages, files for a set of modules. They are described by a DSC and DEC file, and optionally contain an FDF file for platform packages. A platform package is the implementation for a specific platform, which specifies what components and libraries to use, as well as the values of each PCD and more.

`INF` file

An INF file[4] is used to define a module (see below). Its syntax is similar to an `INI` file: it's composed by sections and each section contains entries in key-value format. It describes what a module is, its dependencies, what source files it contains, and more. For example, the `Defines` section sets the name, unique ID, module type, etc., while the `Depex` section is used to specify dependencies.

Module

A module is described by an `INF` file, and it can be either a library instance or a component. It can have different types; the most relevant ones for this work are `BASE` (used for libraries), `SEC` (a SEC module), `PEIM` (a PEI Module), `DXE_DRIVER` (a driver in the DXE phase, commonly to provide protocols for devices, explained below), or `UEFI_APPLICATION` (executable EFI image, for example the UEFI Shell itself).

Each module can have dependencies, commonly `PCDs`, `PPIs` and `Protocols`, described below.

Component

An executable image. Can be a runtime driver, a PEI module (PEIM), and more.

Library

A library consists of an implementation of a certain interface. A *Library Instance* is the actual implementation, defined by an INF file, specifying the *Library Class* (the API for a library, usually the C header) it implements. Then, in the platform package's DSC file, each library class needed is mapped to a library instance in the following way:

```
# Library Class | Implementation path
SerialPortLib|ArmPlatformPkg/Library/PL011SerialPortLib/
    PL011SerialPortLib.inf
```

There can be many implementations of the same library class and different ones can be used in the same platform at different boot stages (for example, two distinct memory allocators).

Driver

Can follow the UEFI or the non-UEFI driver model.

UEFI drivers provide bindings that expose functions to check whether a device is supported, a start and to stop function; they must not configure any hardware.

DXE drivers have fewer restrictions: they can install any protocol and perform necessary hardware and software initialization. For this reason they are the commonly chosen way to initialize devices such as PCIe controllers.

UEFI Application

Executable image that is automatically unloaded after executing. An example is the EFI Shell.

PCD

PCD stands for *Platform Configuration Database*. When referring to a PCD one usually refers to an entry in such database. PCDs can either be fixed at build time or dynamic, and they are one of the main ways to pass around values between modules. A common use case is to specify platform configuration: for example, where the stack begins (`PcdCPUCoresStackBase`), or what level of verbosity the firmware should use when printing debug information (`PcdDebugPrintErrorLevel`).

Modules can require PCDs and/or produce new ones. An example definition of a PCD value in a DSC file is

```
# PCD identifier | Value
gArmPlatformTokenSpaceGuid.PcdCPUCoresStackBase|0x00B00000
```

HOB

A *Hand-off-block* commonly contains information about resources. They are used in early stages to hand-off system information. One common use case is to pass information about system memory from the ATF to UEFI. The full list of HOB types is defined in a header file in the EDK2 repository, `edk2/MdePkg/Include/Pi/PiHob.h`. Common ones are `EFI_HOB_TYPE_CPU` and `EFI_HOB_TYPE_RESOURCE_DESCRIPTOR`.

GUID

A 128-bit globally unique identifier. They are used to uniquely identify things such as modules (using the `FILE_GUID` field) and PPIs/Protocols. In the latter case, they are defined in a C header and can then be referenced from INF files and C source files, in order to be used in a module.

These two are example GUID definitions, one in a header and one in an INF file:

```
#define EFI_DRIVER_BINDING_PROTOCOL_GUID \
{ \
    0x18a031ab, 0xb443, 0x4d1a, {0xa5, 0xc0, 0xc, 0x9, 0x26, 0x1e, 0x9f,
     0x71 } \
}
```

```
## Include/Path/To/Guid.h
gEfiDriverBindingProtocolGuid = {0x18a031ab, 0xb443, 0x4d1a, {0xa5, 0xc0,
      0xc, 0x9, 0x26, 0x1e, 0x9f, 0x71}}
```

PPI A *PEIM-to-PEIM Interface* (PPI) is an interface between different PEI modules. Each PPI is identified by a GUID and provides an API. The functions defined in the API can be called after retrieving the PPI. The GUID is used to retrieve the PPI.

```
// ... GUID usually defined elsewhere ...
EFI_GUID gEfiExampleGuid = {...};

// -- EFI_PEI_EXAMPLE_PPI's definition is omitted. --
// This initializes the API.
EFI_PEI_EXAMPLE_PPI gEfiExamplePpi = {
    Func1,
    Func2
};

// Descriptor of our PPI
EFI_PEI_PPI_DESCRIPTOR gExamplePpiDesc = {
    (EFI_PEI_PPI_DESCRIPTOR_PPI | EFI_PEI_PPI_DESCRIPTOR_TERMINATE_LIST),
    &gEfiExampleGuid,
    &gEfiExamplePpi
};

// Publish example PPI
Status = PeiServicesInstallPpi (&gExamplePpiDesc);
ASSERT_EFI_ERROR (Status);

// Retrieve and use the PPI
Status = PeiServicesLocatePpi (
   &gEfiExampleGuid,
   0,
   NULL,
   (VOID **)&ExamplePpi
);
ASSERT_EFI_ERROR (Status);

// Call a function of our API
ExamplePpi->Func1(...);
```

Protocol

A way to communicate between DXE drivers, analogous to PPIs for PEMs. There are Boot Services (Section 2.2.1) to install protocols and retrieve them, such as `InstallProtocolInterface()`, `ReInstallPorotocolInterface()`, `InstallMultipleProtocolInterfaces()`, `LocateProtocol()`, `OpenProtocol()`[24].

Examples are the Disk I/O protocol (exposing `ReadDisk` and `WriteDisk`) and PCI Root Bridge I/O protocol.

Code adapted from [24].

```
// Protocol handle. It will be assigned when installing the protocol.
EFI_HANDLE msampleHandle = NULL;

// API defined somewhere else.
EFI_SAMPLE_PROTOCOL mSampleProtocol = {
    SampleProtocolApi,
    OtherSampleProtocolApi
};

// Install the protocol (then don’t forget to handle Status)
Status = gBS->InstallMultipleProtocolInterfaces (
    &mSampleHandle,
    &gefiSampleProtocolGuid,
    NULL
);

// ...
// Somewhere else, retrieve the protocol and use it
EFI_SAMPLE_PROTOCOL *SampleProtocol;
Status = gBS->LocateProtocol(&gefiSampleProtocolGuid, NULL, (VOID**) &
     SampleProtocol);
// Handle Status before the next line...
Status = SampleProtocol->SampleProtocolApi();
```

2.4 Enzian

Enzian [9] is a research computer developed at ETH Zürich, boasting a server-class CPU (Cavium ThunderX) coupled to a large Xilinx FPGA. These two are connected through ECI, a coherent processor interconnect: this way, the FPGA can act as a second NUMA node. This means that, for example, it can then implement a memory controller to increase the system memory available to the CPU.

The Enzian machine is connected with the Board Management Controller (BMC), which is responsible for power management, providing an implementation of the EFRI protocol [19] and more.

The fact that this system's structure can change through the FPGA does not necessarily complicate boot firmware development, as long as the operating system can be booted through the components on the CPU side (e.g. on a drive attached to a SATA controller). On the other hand, if the system were to boot via PXE through one of the FPGA-connected NICs, then the firmware would need a way to interact with devices exposed by the FPGA.

2.5 Podman

Podman [17] is my tool of choice to manage OCI containers in the project. It's used to set up a build and development environment by creating a container, using a Tianocore-provided image that contains all required tools to build EDK2.

It runs on Linux, or on macOS and Windows using a podman-managed virtual machine.

I prefer it over docker because it runs without root privileges and because its CLI is Docker-compatible [12].

Alternatively Docker can be used, though it might mess with the permissions of the folders, since by default it runs as root.

2.6 Arm Development Studio

Arm Development Studio (ArmDS) [14] is a development solution specific for the Arm architecture. Its Arm Debugger is the software that I used to troubleshoot the firmware on an Enzian prototype, which was connected with an *Arm DSTREAM* debug probe.

3 Implementation

Now that the related technologies have been introduced, the implementation of the UEFI firmware for Enzian (based on EDK2) is explained, from the folder structure and the build system, to the PCI driver.

During the implementation, one of the main difficulties I've encountered was the lack of sources explaining platform porting. The documentation for EDK2 is divided into multiple documents (e.g. [5]) and is broad, but it doesn't contain many concrete examples on implementing processor-specific libraries and such. The best way to find this is to browse the source code for pre-existing drivers for other platforms, whose specifications are often proprietary and thus not publicly available: understanding why another implementation does what it does without it can be complicated.

The fact that examples on how to port a platform can be hard to comprehend is not the only obstacle: I also did not manage to uncover a complete and useful guide to do so; the process of writing a platform package and its drivers started very slowly, and it initially felt a lot like trial and error, with lots of debugging required in the early stages (see Section 3.1.3).

Another big barrier is the fact that the previous firmware uses an old folder structure (described later in Section 3.1.1) that coupled the EDK2 source to the platform package. Moreover, the full history of said firmware has not been provided. These two gave me a hard time in figuring out what was changed in EDK2 itself and how to correctly port the now obsolete components and libraries. I had to dig in the commit history of the EDK2 repository [25] to figure out what changes had been made and why, often being unable to find any meaningful results. To add insult to injury, some EDK2 libraries and drivers were modified in-place, without any obvious way to figure out which changes have been made.

In the first weeks of the thesis I started from the old firmware's platform package and tried to port it to the latest version of EDK2, without success. Many changes had occurred in the repo, with APIs changing, modules being renamed, moved or even being removed altogether. Due to the huge refactor needed, which was not even guaranteed to produce a working platform, I instead decided to start the implementation from scratch.

First, the new implementation makes some assumptions regarding what the ATF does before handing-off to UEFI. Some of these assumptions could be removed by directly implementing features on the UEFI side.

1. Memory is already initialized.
2. The serial ports are initialized.
3. The UEFI firmware is located at address `0x500000` (in memory).
4. The UEFI firmware's size is at most `0x400000`.
5. The stack is at `0xB00000`, and it can be at most `0x7000` big.

6. There are some SMC calls that can be used.
7. The PCIe and SATA controllers are initialized.
8. Information about system memory, such as start address and size, can be retrieved.

These assumptions are required for convenience: for example, the ATF already prints debug information to the serial port, meaning that it already has to initialize it itself, so there is no need to do it again.

Also, memory initialization is very hard and, since the ATF has to initialize it anyway to work, it makes sense that UEFI should just worry about mapping it.

The firmware being located at said address is just for convenience during development. One could bundle ATF and UEFI together to reduce space by having no padding, but this would complicate memory mapping, and it would make the flashing procedure harder, as now the starting address of UEFI is unknown. By having a clear reserved flash region for UEFI, the firmware knows the start address at compile time, independently of the ATF. The same applies to the stack start address.

Some SMC calls can be used to interact with the hardware through the ATF, in order to avoid duplicating functionality (e.g. using EFRI).

In the old firmware by Cavium, both PCIe and SATA controllers were initialized by lower level firmware; the UEFI implementation did not contain initialization code for them neither in PEIMs nor in DXE drivers.

This is not the case anymore in the new ATF implementation, so supporting them requires either a custom PEIM or DXE driver. As of now, I did not manage to implement this due to time constraints, though the procedure could be ported from the already existing implementation in the BDK.

3.1 Project Structure

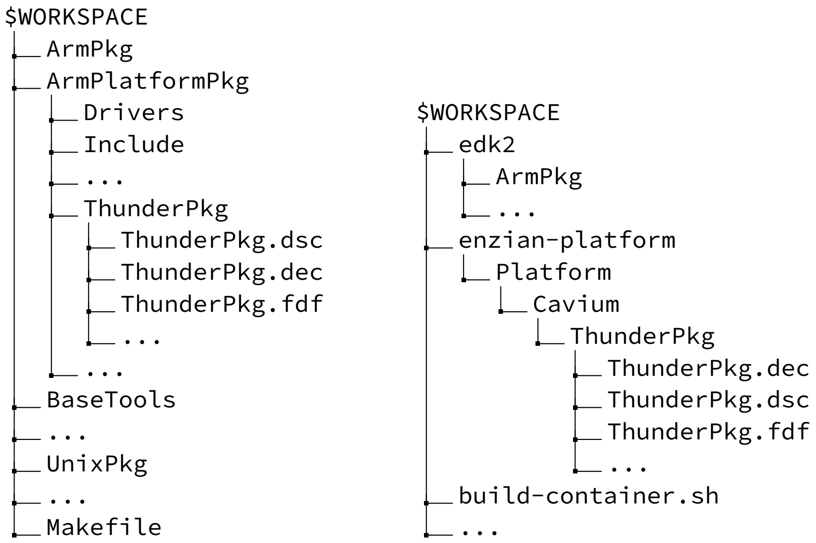
The first thing a person sees when opening the repository on GitLab is the README, accompanied by the files in the root directory. For this reason, it is important to have a clear and comprehensible structure that's easy to navigate, thus it makes sense for it to be introduced before the rest. After the structure itself, the containerized build system is illustrated.

The current Cavium firmware for the Enzian platform requires very specific tools to be built (even a custom GCC, which is not readily available) and setting up such development environment is very cumbersome. Also, the folder structure mixes together the platform-specific implementation and the platform-agnostic EDK2, making it hard to distinguish changes made to one or the other.

For this reason, I've tried to choose the most sensible setup in order to allow for quick and reproducible builds, while at the same time simplifying the time required to understand the project and make changes. Lastly, the chosen folder structure (adapted from the `edk2-platforms` repository[26]) should help in keeping the EDK2 [25] version up to date; it is explained in the next section.

3.1.1 File structure

As already said, the file structure is a very important choice that will affect how easily distinguishable different components are.



```
$WORKSPACE
├── ArmPkg
├── ArmPlatformPkg
│   ├── Drivers
│   ├── Include
│   ├── ...
│   ├── ThunderPkg
│   │   ├── ThunderPkg.dsc
│   │   ├── ThunderPkg.dec
│   │   ├── ThunderPkg.fdf
│   │   └── ...
│   └── ...
├── BaseTools
├── ...
├── UnixPkg
├── ...
└── Makefile
```

(a) Dirtree of old firmware

```
$WORKSPACE
├── edk2
│   ├── ArmPkg
│   └── ...
├── enzian-platform
│   └── Platform
│       └── Cavium
│           └── ThunderPkg
│               ├── ThunderPkg.dec
│               ├── ThunderPkg.dsc
│               ├── ThunderPkg.fdf
│               └── ...
├── build-container.sh
└── ...
```

(b) Dirtree of new firmware

Fig. 1: Different structure of old and new

The old firmware, written by Cavium, can be found in the `enzian-uefi` repository [11]. Its structure (Figure 1a) is quite awkward to work with, and it is not recommended anymore: it consists in putting the platform package inside the EDK2 ArmPlatformPkg one. This causes issues, as now the folder structures of both the firmware implementation and the platform port are tied: the repository is now a fork of edk2/master and syncing it with new changes in EDK2 is harder, as even the histories are correlated.

Also, getting started with the project structure is harder: the platform package is "hidden away" in the firmware itself and the root of the repository is composed of tens of folders, among which the README, the build scripts and the Build directory are dispersed.

Instead, I used the same structure as the edk2-platforms [26] repository: The edk2 repository is available as a separate folder in the root, while the platform package is located inside the enzian-platform folder. Also, the edk2 folder is a submodule, meaning that it can be updated and downgraded at will, independently of the rest of the project.

This kind of folder structure also allows build scripts, build directory and various READMEs to live in the root of the repository. The consequence is that they are easily identifiable from implementation-related files, as those are entirely contained in separate folders.

Inside the edk2-platforms folder, as per the official Tianocore repo's standard, there is a Platform folder. Normally one should split part of the implementation to the Silicon, which I did not yet do. Preferably, this should be the case, as it allows distinguishing between the processor-specific definitions and the board ones. In the future, this makes it possible to share a lot of the implementation between Enzian 3 and the potential Enzian 4 platform.

Currently, there is an issue with the structure: the Conf folder resides inside the edk2 directory. Some future changes might require changes to the files within it, which only requires setting an environment variable to indicate the new path of said folder; it's thus possible to keep it separate from the submodule.

The ThunderPkg is the platform package, containing all the definitions and implementations needed to build a working firmware.

As shown in figure Figure 1b, three such files are ThunderPkg.dsc, ThunderPkg.dec and ThunderPkg.fdf.

First, the ThunderPkg.dsc file states all required component/libraries and sets the Pcd values needed, as well as the build options for the platform package.

Then there's the ThunderPkg.fdf file, which describes the layout of the produced image (size, expected start address, alignment, etc.), as well as its contents (components, etc.).

The ThunderPkg.dec file is used to define new Pcds, Guids and more for this package.

Other than these files, there are directories to contain drivers, libraries, headers and others. This makes it easy to distinguish between each module type, as otherwise one would have to open the INF file to distinguish between libraries and drivers.

3.1.2 Build system

The project’s build system leverages EDK2 itself and either Podman or Docker.

The `build-container.sh` script uses one of the two to set up a build environment, preferring Podman if available.

At the present time, EDK2 recommends the use of containers to be built [22]. The container is based on a Tianocore-provided image (currently an Ubuntu-based one[1]) and mounts the entire project folder inside its work directory. Afterwards, it runs the `build.sh` script.

This script uses EDK2’s build tools to build the platform module. It functions as follows: First, the `edk2` git submodule is initialized with the recursive option (as it contains other submodules itself).

Then, the required env vars are set, either in `build.sh` or `env.sh`.

`TOOLCHAIN=GCC5`
: In our case we’re using GCC, but one could use the Microsoft compiler instead.

`ARCH=AARCH64`
: The architecture that our platform uses.

`PACKAGE=Platform/Cavium/ThunderPkg/ThunderPkg.dsc`
: The package that we want to build (our platform package).

`GCC5_AARCH64_PREFIX=aarch64-linux-gnu-`
: The prefix of our toolchain, which then is prepended to e.g. `gcc`.

`PWD=$(pwd)`
: The workdir (the current folder inside the container).

`PACKAGES_PATH="$PWD/edk2:$PWD/enzian-platform"`
: A list of root folders inside which the build system can look for packages.

`WORKSPACE="$PWD"`
: The root of the project.

`NUM_CPUS=$((‘getconf _NPROCESSORS_ONLN‘ + 2))`
: Threads to use for building.

Afterwards, different steps are performed:

First, the `edk2/edksetup.sh` script is run. This checks that `WORKSPACE` is set, checks whether `python3` is present and whether the `BaseTools` directory exists.

Then a python virtual env is set up in `.venv` (if not already present) and activated. Inside this env the required packages are then installed from the `edk2/pip-requirements.txt` file.

The last step before building is to make `BaseTools`. It looks like using the `-j` argument breaks it, so its build process is single-threaded.

[1] ghcr.io/tianocore/containers/ubuntu-22-build:latest

Afterwards, the build script found in BaseTools has been added to the PATH, thus we can run it. According to the EDK2 build instructions, it is now preferable to use stuart instead, but I've chosen build because I find it easier to get started with; it's also the one used in the old project.

This script takes some arguments, the main ones being the following:

-n $NUM_CPUS threads to use.

-a $ARCH architecture to build for.

-t $TOOLCHAIN toolchain to use.

-p $PACKAGE The package to build.

-b DEBUG Since the implementation is not complete I only performed DEBUG builds, with fewer optimizations and more logging.

-y report.txt Tells the build system to produce a report. This file is especially useful when looking for the dependency chains of each component, since they not only have dependencies of their own (specified in the INF file), but also inherited from all the libraries they depend on. This report saves a lot of time when figuring out why a component is not being executed by the DXE dispatcher.

After running it, a Build folder (Figure 2) will appear in the workspace together with the report.txt file.

This folder contains all built components (both as ELF files with debug symbols and as PE32+), all firmware volumes and the final flash image THUNDER_EFI.fd. This image encapsulates all firmware volumes (see Section 3.2.1) which, in turn, contain all components.

```
Build
└─ThunderPkg
   └─DEBUG_GCC5
      ├─FV ................................ FVs and final THUNDER_EFI.fd
      │  ├─THUNDER_EFI.fd ................................... file to flash
      │  ├─FVMAIN_COMPACT.Fv
      │  └─...
      └─AARCH64 ........................................ Built components
         ├─ArmPlatformPrePeiCore.debug ........... ELF file, symbols
         └─ArmPlatformPrePeiCore.efi ....... PE32+, without symbols
```

Fig. 2: Dirtree of $WORKSPACE/Build

To boot, the final image can then be flashed on a machine at the correct address (currently 0x600000).

3.1.3 Flashing and Debugging

For development, I used mostly the `hinterrugg0X` prototype boards, as the flashing procedure requires less steps than an Enzian machine. Also, to be able to debug the firmware, I needed `hinterrugg01` since, at the time, it was the only prototype boards equipped with a debugger connection.

In order to flash the image, one has to attach to the BMC console and run `thunder_update -a 0x600000 -n path/to/THUNDER_EFI.fd` to flash the image *in flash* at the given offset.

Given how hard it is to find documentation on how to port an ARM platform with EDK2, I often ran into runtime errors originating from PCDs that I did not set. This is because these PCDs, when not set, inherit their default value (specified in their definition), often `0x0`. Many PCDs are used to specify addresses of registers, offsets, sizes and so on, which resulted in runtime errors that required a debugger in order to be diagnosed, such as Data Abort Exceptions.

In later boot stages (e.g. DXE), exceptions are trapped and the corresponding error message is printed, including the location in the C source at which it occurred. This makes the debugger less of a requirement for driver development. Given how the flash image is structured, the components that necessitate a debugger the most are those in the first Firmware Volume, `FVMAIN_COMPACT`.

Adding the source files to Arm Development Studio requires uploading the `.debug` files to the machine where said software runs. Its debugger is GDB based, so to add each of them one has to use the `add-symbol-file` command. This command takes a *path* to the symbol file and the *offset* at which it's located in memory. Entering the correct path and offset combination for each file is a long and error-prone process, as the user has to manually figure out where it was placed in memory. Also, this location might change between builds, which makes a manual computation of said offsets unfeasible in the long run.

In order to automatically compute these offsets and to generate the required GDB commands to add the symbol files to the debugger, I've created the script `gen-symbol-files.py`. This script works by processing some intermediate files produced by the build system.

Inside the `Build` folder (Figure 2) there is `FV/FVMAIN_COMPACT.Fv.map`, which then contains entries in this format:

```
ModuleName (...  EntryPoint=0x1234abcd ...)
```

With a regex it becomes trivial to extract the two pieces of information in bold text, which are the name (without extension) of the symbol file and the `entry_address`. Note that this address indicates where the first instruction (entry point) resides, and not where the file actually starts in memory. This needs to be computed with the next step.

Now, the symbol file path is resolved and the `e_entry` field of its ELF header [1] is read. This field is the relative offset of `entry_address` from the start of the file, thus the *offset of the symbol file* in memory is `offset = entry_address - e_entry`.

The **ModulePath** depends on the path of the symbol file on the debugger machine, which is specified in a variable in the script. This path does not have to be fixed and can be changed if desired.

Finally, the completed commands, `add-symbol-file` **`ModulePath offset`**, can be appended to a `.ds` file that can be executed by the debugger to:

1. Add all symbol files.
2. Add a breakpoint at the firmware entry.

I did not manage to find better ways to debug the firmware other than uploading all the `.debug` files and the generated `.ds` script to the debugger machine. However, since they can be uploaded via `scp` as a folder, different iterations of the firmware can be easily debugged by using the correct bundle. Also, the procedure of uploading said auto-generated folder is quick and leaves little room for error.

The only downside is that the source files need to be uploaded separately and one has to map the paths correctly, since the symbol files contain paths relative to the build environment and not the debug one (concretely, EDK2 source files have the prefix `/source/edk2` since they are built in a container). I did not try to solve this issue, but one solution could be to bundle the sources together with the symbol files and to include a directive in the `ds` script to substitute paths.

3.2 Boot Flow

When the ATF is done executing, it hands off to UEFI. To do so, it just jumps to the start of the UEFI image (`0x500000`). Now the UEFI firmware starts executing, runs to platform initialization, and at some point it starts dispatching modules, drivers, etc. at runtime. This is where a more in-depth description the `FDF` file becomes necessary.

3.2.1 ThunderPkg.fdf

The FDF file specifies how big the image is, where it expects to sit, the alignment, which Firmware Volumes it contains and in what order (optionally where exactly they are located); then, for each FV, it tells what modules it must include and in what order. Finally, the instructions on how to combine the produced binaries.

The FDF [23] file starts with the `FD.THUNDER_EFI` section, which defines the entire flash image for the device. This contains:

`BaseAddress = 0x500000` Tells the build system that the image assumes to start the specific address when it starts executing. In the code, it's followed by `|g....Pcd...`, which assigns the value (left) to the given PCD (right).

`Size = 0x400000` This is the length of the image. By previous convention, the image has a fixed size to aid in flashing it, as one only has to erase the given memory range. This size is bigger than needed, to stay conservative in case the firmware needs to grow in the future.

`ErasePolarity = 1` To erase flash, `0xFF` bytes are written to it. Currently, the flashing procedure already erases the memory region before writing data, so this parameter is useless. Nevertheless, it is here in case this changes.

Blocks These two are related to the previous one, as `Size = BlockSize * NumBlocks`. These properties depend on the flash.

```
BlockSize = 0x1000
NumBlocks = 0x400
```

FV definition These next lines of code are strictly related to each other and define what FV is to be placed where.

```
0x00000000          |   0x00400000
...PcdFvBaseAddress |   ...PcdFvSize
FV = FVMAIN\_COMPACT
```

The Firmware Volume (FV) is what then contains modules and components. The second line of PCDs are assigned a value based on the first line: it sets `0x400000` to `PcdFvSize` and **adds** `0x0` to `PcdFvBaseAddress`, which was previously set to `BaseAddress` above! At the start of the project, a typo in the PCD name produced an image that would jump to the wrong entry point, which prevented it from functioning. The last line is the identifier of the FV to be placed in the given region.

Next, the Firmware Volume sections, which define the components to be placed within them. Note that a Firmware Volume can also contain another one, nested within.

The different `INF` statements place a component in the FV, sequentially, in order. `FILE` statements can place other FV sections, binary blobs, and more. There are two sections used, `FV.FvMain`, containing `DXE` modules, and `FV.FVMAIN_COMPACT`, containing the entry point, `PEIM`s and the entry point of `DXE`. Splitting these two parts can be beneficial, as the second one is larger and could be compressed.

First, `FV.FV_COMPACT`. The parameters are the following:

`BaseAddress = 0x500000`
Again, since this image contains the entry point, we need to specify the base address.

`FvAlignment = 16`
Aligned to 16 bytes.

Rest The other parameters are less relevant, and I've never seen them being changed for other platforms, so I omit them.

After these parameters there's the list of things to be included.

The entry point of UEFI is the first element of the list, `PrePeiCoreMPCore`, as it is placed at the start.

The other `INF` entries are just PEIMs and the DXE entry point.

The last entry is a `FILE FV_IMAGE`, which is the FV section to be nested within this one.

After this one, there's another FV section, containing all the components for the DXE phase (mostly drivers). The parameters are similar and again, the components are included with `INF` directives.

The only difference is the presence of an `APRIORI` directive: this indicates a list of modules that must be dispatched in a prescribed order, ignoring their dependencies.

3.2.2 SEC Phase

As said before, the first instruction at `BaseAddress` is a jump. This executes the entry point of `PrePeiCoreMPCore`. This module is for (almost) every Arm platform that EDK2 supports. It is written in assembly and C and its entry point does the following:

1. If executing in EL3, switch to EL2 before jumping to PEI. Otherwise, if in EL1 or EL2 Sec, it goes straight to PEI. Some setup is done depending on the level.

2. Then jump to MainEntryPoint, where only the primary core is allowed to run.

3. The temporary stack is set up (later permanent memory will be setup) and backtraces are terminated by setting the frame pointer to null.

4. Jump to `CEntryPoint`.

Now that the stack is set up, `C` code can run.

`CEntryPoint` is defined in `PrePeiCore.c`. This function invalidates cache, sets up the exception vector table, enables the VFP, and then branches depending on whether it's executing on primary core or not. The secondary cores run the `SecondaryMain` function, which I will not discuss, while the primary one, in order:

1. Runs all library constructors

2. Prints firmware version and initializes the debug agent (including the serial port).

3. Calls `ArmPlatformInitialize`

4. Calls `PrimaryMain`.

The serial port initialization is explained in its own section, see Section 3.3.

`ArmPlatformInitialize` is defined in the `ArmPlatformLib` library. In our case it does nothing.

3.2.3 PEI Phase

After this, `PeiMain` is called. This module is the PEI entry point and its job is to describe the memory and map it (system memory, firmware locations, etc.), and then transition into DXE. It should be as small as possible. Often one might want to initialize some devices before DXE. This phase also dispatches PEIM (Pre-EFI Modules) at runtime, in a dependency-aware fashion, explained in the background section.

One of the most important libraries here is again `ArmPlatformLib`, where one must also define a function that sets up a table of memory mappings. This function, `ArmPlatformGetVirtualMemoryMap`, is responsible for creating a table that describes all memory regions and their types, such as system memory and configuration registers for devices.

The module `MemoryInitPeim` requires this library, as it calls said function in order to obtain the table of mappings. This table is then used to map memory accordingly and to know what type each region is. In our case, all physical memory ranges are mapped 1-to-1.

The main regions are:

- The available system memory (starting at `0x1400000`, after the firmware-reserved space).

- The UEFI firmware and stack themselves.

- Various registers for peripherals (UART, GIC, PCI, and more).

Each memory region also has attributes that describe what it is: for example, the UART will have attributes ARM_MEMORY_REGION_ATTRIBUTE_DEVICE, while system memory might have attribute DDR_ATTRIBUTES_UNCACHED.

Given this table, EDK2 can autonomously map each of the ranges and know what every one of them corresponds to.

Since the system memory configuration can change between boots (and some machines even have hot-pluggable memory), its range should be described in a HOB produced before this module runs, or retrieved with an SMC call. Otherwise, one must know the memory configuration at build time, which can be the case for a Single Board Computer, but does not apply to our situation. This HOB is commonly handed-off by the ATF to the UEFI, since the ATF knows exactly the configuration after initializing DRAM.

An example memory mapping is defined as follows:

```
// VirtualMemoryTable contains the list of regions to be mapped.
// DRAM Mapping for UEFI code and Stack, see header for values.
VirtualMemoryTable[Index].PhysicalBase = ARM_THUNDER_DRAM_UEFI_BASE;
VirtualMemoryTable[Index].VirtualBase = ARM_THUNDER_DRAM_UEFI_BASE;
VirtualMemoryTable[Index].Length = ARM_THUNDER_DRAM_UEFI_SZ;
VirtualMemoryTable[Index].Attributes = DDR_ATTRIBUTES_UNCACHED;
```

The last PEIM is DxeImpl, which is responsible for loading the DXE Core from an FV.

3.2.4 DXE Phase

During DXE phase, a dispatcher executes drivers based on their dependency chains. The goal of the DXE phase is to get to the BDS phase: to do so, all dependencies of the BDS driver must be met.

There are some DXE drivers that require either custom libraries or partial reimplementations depending on the platform. For example, since there is no RTC on Enzian machines, a custom driver that retrieves time through EFRI should be implemented.

One of the most commonly modified set of drivers I've encountered are those related to PCI and SATA, since different machines might have vastly different PCI configurations, with varying numbers of Host Bridges, Root Bridges and very different ways to initialize them.

Notably, the following drivers are the most important ones for the project, as they are required in order to boot from a disk: AtaAtapiPassThru, AtaBusDxe, PciHostBridgeDxe and SataControllerDxe. They are described in Section 3.4.2 and Section 3.5.

3.2.5 BDS Phase

In the BDS phase, a UI is available to the user to choose a boot option or configure the machine. From here, it's possible to also enter the UEFI Shell.

At this point, all drivers that deal with SATA drives, network interfaces need to be correctly initialized.

3.3 Serial Port

In order to display information and to read user input, the firmware needs a serial port.

First, the serial port of the ThunderX is PL011 compatible, but there are some differences, such as the clock divider used. This means that EDK2's PL011 library is not able to compute the divisor correctly given baudrate and clock. In our case, the port is already initialized, thus we can simply avoid resetting it.

The implementation in the old firmware actually modified the EDK2 `PL011` serial port library to **not write** to its configuration registers, which I believe to be bad practice.

The first issue with this approach is that this was not documented anywhere. Second of all, the structure of EDK2 is made so that one can separate the platform implementation from the universal components/libraries, making it easy to understand what is platform-specific and what is not.

In fact, I encountered issues while setting up the serial port: the PCDs defined in the old implementation's DSC file suggested that the baud was not the expected `115200`. This is because all operations done in the `PL011` library were never actually written to any register: all computed register values were indeed wrong, but they were never actually used.

My approach is also a bit hacky, even though it has the advantage of leaving the library untouched and leveraging intended behavior. Since the baud rate and the clock are fixed in our case, I've hardcoded the values in the `ThunderPkg.dsc` file.

The library is implemented so that, if the clock and the divisor are specified and equal to the **current** ones, then the library does **not** re-initialize the serial port, leaving the registers untouched. To obtain this result, I just pulled the values from the same registers and set the PCDs found in table Table 1:

`gArmPlatformTokenSpaceGuid.PL011UartInteger`	`0x48`
`gArmPlatformTokenSpaceGuid.PL011UartFractional`	`0x15`
`gArmPlatformTokenSpaceGuid.PL011UartClkInHz`	`133315200`

Tab. 1: PCDs for the PL011SerialPortLib

3.4 PCI Controller

I managed to complete a mostly-working PCI driver, though for some reason the SATA driver `AtaAtapiPassthruDxe` fails, and the drive is never read. I did not manage to diagnose the issue in time This process is explained in Section 3.4.2 and Section 3.4.3.

Again, to collect all available options, I looked at the old driver and at how the various comparable platforms did it in the platforms repository[26], especially ones from Ampere, Marvell and Hisilicon; these have characteristics analogous to our platform, except of course the FPGA connection, so they should provide insight.

Initially, I tried using the old `PciHostBridgeDxe` driver, which did not work, as the configuration registers could not be read for some reason. Then I tried implementing my own: after some unsuccessful attempts, I managed to set up a PCI stack that seems to work fine.

In order to understand the design of a PCI driver, one must know some terminology and how PCI is structured in general; different sources might use the same term interchangeably, which is confusing.

3.4.1 Structure and terminology

The terminology used is mostly taken from [18], in order to stay consistent with the one used in EDK2.

A platform can be abstracted as a set of CPUs that are connected to a set of chipset components. These components can produce a certain amount of *Host Bridges*.

The different host bridges have separate configuration, memory and IO spaces.

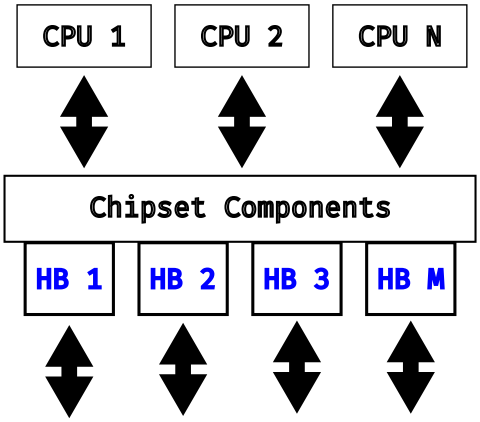


Fig. 3: Host Bridges (HB) in a system

A system might have one or more host bridges and each host bridge can then be composed of one or more *Root Bridges*.

A *Segment* is a collection of *Bus*ses that share the same *Configuration Space*. Different chipsets might need to abstracted in different ways and a `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` instance might abstract one segment or just a portion of one. It is possible to have multiple segments in a system.

The next component is the ECAM block. Software initiates ready/writes to the ECAM address space by forming an ECAM address; then this is forwarded to the correct bus/device/function.

On an Enzian machine, this ECAM address has a specific format, which includes the ECI node number, the ECAM number, the bus, the function and the register offset within the device.

In this case, I abstract the system with 1 host bridge and 4 root bridges (one per *ECAM*). Technically it has 4 host bridges, since the different ECAMs have different configuration spaces, but for EDK2 this is not a problem: the config spaces for each can be handled in the `PciSegmentLib` library, by returning a different one based on the segment.

The PEMs are used to access off-chip functions and their configuration registers are specified by the PCIe standard. Since the devices are external, these registers are implemented within the device itself. The ECAM was created for PCI Express and provides a convenient mechanism to access their PCI configuration registers.

Some devices, such as the SATA Controllers, are available through the root bridges.

Initially I represented the 4 ECAMs and the 6 PEMs as Root Bridges (they defined in the next section), though I made a mistake in the implementation: this caused the configuration registers for some of them to be read from the wrong address. This old implementation is explained in Section 3.4.5.

Finally, I only represented the 4 ECAMs as Root Bridges. I realized that the PEMs can be enumerated through two of the ECAMs and, consequently, also all PCI devices and busses in all segments.

On an Enzian machine, the different ECAMs allow enumerating and configuring many devices. These include the SATA Controllers, off-chip PCIe devices, the network interface and other internal components.

3.4.2 Drivers and Libraries

Currently, for a PCI controller to work, there are different components and libraries required. The main ones are the following:

`PciBusDxe` Probes all PCI devices and allocates Memory Mapped I/O (MMIO) space and IO space for each, optionally with hot plug support. This driver is usually left untouched.

`PciHostBridgeDxe` Located in `MdeModulePkg`, this driver is what installs the main protocol `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL`, which is required by other components related to PCI. This protocol offers ways to access I/O, MMIO and PCI configuration space and more. For most platforms, this driver doesn't need rewrites, as it depends on two libraries that have been created with flexibility in mind: PciHostBridgeLib and PciSegmentLib.

PciHostBridgeLib This library's main function is to return a list of instances of type PCI_ROOT_BRIDGE, each of which is a description of a root bridge: it contains its attributes, segment and MMIO/IO ranges. This description does not include the configuration space range and any indication of what host bridge it belongs to. We need to modify this library in order to return a list describing the platform's different root bridges.

PciSegmentLib This library provides ways to access the configuration space of every segment, given an internal 64-bit *Segment Address* in the form SEGMENT:BUS:DEVICE:FUNCTION:REGISTER, encoded as a UINT64. In order to implement PCI for our platform, this library requires a lot of tweaks: it's this library's job to map the above addresses to the physical memory-mapped address in the configuration space. The design I initially preferred is the one from Marvell[16], which can be found in the edk2-platforms repository. However, as I will explain at the end of the section, I propose the use of another implementation of this library moving forward.

Now let's briefly introduce the differences between the old implementation and what I think the new one should be.

The in the old firmware there is no PciHostBridgeDxe driver in EDK2 itself, so the driver present is structured very differently from the one currently used by most platforms. Also, it did not have PciHostBridgeLib nor PciSegmentLib: I speculate that they did not exist at the time when firmware was initially written. Though I have no confirmation of this, since the history is unavailable.

In newer platforms from Ampere and Marvell, the approach used is different and, in my opinion, superior: it does not modify any driver code and platform-specific behavior is handled by implementing the two libraries above.

The requirements to write a working PCI implementation are:

- PciHostBridgeLib correctly describes all available Root Bridges.
- PciSegmentLib correctly implements configuration space access for all root bridges.
- PCI Controllers are initialized by either the ATF or UEFI, e.g. in the PciHostBridgeLib library constructor.

The implementation of each component is explained in the following sections.

3.4.3 Implementation

The first step to understand how to proceed was of course to comprehend how the old firmware's driver is implemented. The function definitions it provides are quite similar to the modern one, though their implementation varies a lot.

Not only the two libraries mentioned before don't exist, but a lot of the struct used have different fields different, e.g. the one to describe root bridges.

This old driver seems to partially work with minor modifications, but for some reason the SATA driver doesn't. As explained later, I tried both the old and the new SATA driver without success, so I figured out that the culprit is to be found in the evolution of PCI related libraries and drivers in EDK2.

For this reason and to conform to what other platforms do I decided that a reimplementation of the PCI stack is necessary. Also, adding initialization code in UEFI could be done at a later time.

The first thing I did is to extract the root bridges' configuration from the old firmware and adapt it to the new one. The configuration space memory aperture is to be used later to implement the segment library, while all other information has to be translated to the new format.

The format of the struct `PCI_ROOT_BRIDGE` that must be returned in `PciHostBridgeLib` is quite different from the old one, especially when it comes to the way that memory apertures are specified.

The old firmware did so by specifying an aperture's base *device address*, the *limit* (either top address or length), and the base *CPU address* to which the device one maps.

In the new firmware they need to be translated to a different format and each aperture is to be split in two fields, one for memory up to 4G and one for memory above 4G. The format is `device_base` address, `limit` (top address) and `translation`, where `translation = device_base - cpu_base`. For spaces starting before 4G and ending after it, one must specify both apertures. If the space starts after 4G, only the aperture above 4G is required. Unused apertures are to be specified with `device_base = UINT64_MAX` and `limit = 0`.

This way, all root bridges (ECAMs) are correctly described, and the driver can then compute base addresses, length, translate between device addresses and CPU addresses, and so on.

The next step is to implement the `PciSegmentLib` library. Here there are functions to read/write from/to configuration space. To access it, the library must, in order:

1. Get the segment index from a segment address.
2. Map said index to the base configuration space address of a root bridge.
3. Perform the access to the corrected address.

The map of addresses is derived from the old firmware root bridge definitions, while the segment index is trivial to obtain, as it is `Address[32..47]`.

3.4.4 Possible improvement for PciSegmentLib

There is a second (and possibly better) approach to be investigated in the future, the `PciSegmentLibSegmentInfo` library: it's made to access PCI configuration space *solely through ECAM*, and it allows to specify a list of `PCI_SEGMENT_INFO` to map a segment to a specific configuration space base address and a range of busses. Unluckily I discovered it only at the time of writing the report, and thus I was not able to test it in time.

With this library the Root Bridge representation might need to be adapted. It's possible that other components must be abstracted as Root Bridges and the 4 ECAMs should be left out, since the segment library explicitly uses ECAM to access the PEM configuration. Though, the current segment library is working so far, I've not investigated this alternative any further.

It could be an improvement for these three reasons:

- Reduces the amount of code to be written, since the segment library doesn't need to be modified: it performs address conversion automatically based on the list of segment information provided.
- It allows accessing segments as a Runtime Service, which our library currently doesn't.

3.4.5 Old Root Bridge representation for PciHostBridgeLib

Even though the older version I wrote doesn't work (where the 6 PEMs are also represented as root bridges), I'll explain what it did and why. Since no Linux boot has been achieved yet, I cannot rule out that the way the PCI system is represented will work. One reason is that some PEMs might not be initialized, since some of them contend a QLM with a SATA controller, and they cannot be read from. In the future this might cause the firmware to crash, depending on how PCI initialization is going to be implemented on UEFI: the disabled PEMs must not be discoverable through ECAM, so that they are never accessed.

In the previous version, where the 6 PEMs were also represented as root bridges, the library made sure that they were initialized before including them in the representation.

To do so, it populates the list of root bridges in `PciHostBridgeLib` at runtime, when the library constructor is called. The program starts with a full list of root bridges, and then filters out the unavailable PEMs by doing the following:

1. Read the configuration of the multiplexers to which this PEM is connected [3].
2. If none of them are configured as PCI, then skip this PEM and proceed to the next one.
3. Otherwise, read the register `PEM_ON[0..1]`: if it's not exactly `3`, the PEM is not usable, and thus it must be skipped.
4. Next, check the link status. To do so, I used the register `PEM_CFG_RD` (64 bit wide), which can be used to read the PCIE RC-Mode configuration registers. Write the RC register offset that needs to be read to the bottom 32 bits; the top 32 bits get set with the value contained in the given register

offset. `PEM_CFG_RD` can now be read back, and the data is found in the top half.

The `PCIERC_CFG032` register is read using this mechanism; bit `29` must be set and bit `27` has to be zero, otherwise this PEM is not ready to be used, and it's filtered out.

5. Keep the root bridge in the list as it can be used.

The ECAMs are always available, as all of them should be initialized.

Again, I believe that ECAM enumeration won't end up registering uninitialized PEMs as long as the initialization is done properly, though I am not completely sure.

3.5 SATA Controller

In order to achieve the goal of booting Linux using the new firmware, the SataControllerDxe driver is required (standard driver provided by EDK2). It depends on the protocol gEfiPciIoProtocolGuid, thus it requires a working PCI driver.

The old firmware's PCI implementation does not work for any of the SATA drivers I tested, even the old one: disks are never enumerated (and thus nothing can be read from them).

Moreover, the EDK2 standard SataControllerDxe implementation also does not yet work on Enzian, because of a fault in the current SATA controller initialization in the BDK (see the explanation in Section 3.5.2). I believe that the best way to implement this going forward is to (properly) initialize the SATA controllers in UEFI and then this standard driver should just work. I am not aware of any peculiarities that could prevent the ThunderX' SATA controllers from functioning correctly with it, assuming that PCI is fully working.

Currently, an issue prevents the AtaBusDxe driver from completing drive enumeration, though I have not yet found the culprit. Further investigation is required.

As for the driver's structure, the main three functions that require modifications are the following:

- SataControllerSupported
- SataControllerStart
- SataControllerStop

Other than this driver, the following ones are also required in order to perform disk I/O. They are provided by EDK2 and should not require any modifications, as they are implemented based on defined standards.

AtaAtapiPassThru Implements a protocol for ATA controllers. Implements the functionality required to enumerate drives and interact with them and the ports. For example, it allows to reset a port or a device.

AtaBusDxe Requires AtaAtapiPassThru and exposes a protocol for drive I/O. It implements various functions to perform reads and writes to the given disk.

DiskIoDxe Provides the Disk I/O Protocol. Used to layer on top of block devices in order to present them as byte-oriented devices; it makes accessing a disk easier, and it's the protocol used for I/O by later modules.

Fat An implementation of the FAT.

3.5.1 SataControllerSupported

This function returns EFI_SUCCESS if the given device is supported, or otherwise it returns EFI_ALREADY_STARTED if the driver is already running on the device. If it returns any other status, it means that the device is unsupported.

First, the PciIo protocol is obtained and then the class code of the device is read. This code tells us whether the device is a SATA controller.

Afterwards, we must check whether the controller is actually wired to the physical port through the multiplexer or not. This requires reading the GSER configuration (see [3]) corresponding to this port to see whether this is the case.

If both these conditions are satisfied, then the device is supported, and we return success.

3.5.2 SataControllerStart

This function is only called after obtaining a positive return value from the previous SataControllerSupported, thus it can safely assume that the device is indeed supported.

First, the PciIo protocol is again obtained and an instance of a struct holding all private data for this controller is allocated and initialized.

The current PCI attributes are read and saved to be restored when stopping the driver.

Then, if the controller is an IDE, the ChannelCount and DeviceCount are set to their max values.

Otherwise, if it's a SATA controller, then we must obtain the port count. Normally the driver reads the PI register (AHCI HBA Memory, [20]), which is a one-time writable register holding the count of implemented ports. For some reason, this register is always zero, so it is either never written to or it's zeroed.

I then looked through the old firmware and the manual and found an alternative way to obtain the port count. Similar to the old driver, I modified the driver to read the CAP register instead, which contains the maximum number of ports supported. This behavior doesn't look right, and it should be fixed once SATA initialization is reimplemented on the UEFI side, but for now this approach should work.

Then ChannelCount is set to the discovered value plus 1 (as per the register definition) and DeviceCount either to the max value.

Then the rest of the function is the same as the EDK2-provided implementation.

The controller then allocates some arrays to hold configuration, with one entry for each channel of each device (thus DeviceCount * ChannelCount entries).

At last, the protocol for this controller is installed. Optionally some values are freed.

3.5.3 SataControllerStop

This function looks exactly the same as the EDK2 implementation. It uninstalls the protocol for this controller, frees previously allocated configuration fields and finally restores the PCI attributes that were saved in the start function.

3.6 ACPI

In order to not be limited in terms of software choice, Enzian should be able to ideally boot as many OSes as possible. Otherwise, future research could be hindered.

In order to fulfill the Server Base Boot Requirements (SBBR) [10], one of the components required is the presence of ACPI Tables fulfilling some requirements described in the cited specification, with emphasis on the Hardware-reduced ACPI model [7]. The Hardware-reduced model requires, among other things, to boot only in ACPI mode.

This is desirable for Enzian, as many OSes require SBBR compliance.

EDK2 already produces ACPI Tables on its own by including the hardware it knows about, for example the components that have been enumerated through ECAM. Moreover, the old firmware includes most of the tables required for SBBR compliance.

I have no reason to believe that the ACPI standard cannot be used to represent Enzian: by design it is able to represent NUMA nodes with or without a processor (e.g. with only a memory controller) and even convey their topology [8].

Since the firmware is not yet able to boot an operating system, I was not able to test whether these tables are up-to-date; though they seem to comply with the specification.

3.6.1 Pre- or Post-boot

Now there are two approaches I can think of to create an ACPI representation of the FPGA-synthesized hardware: in the firmware (pre-boot) or in the OS (post-boot).

For pre-boot the firmware must somehow know what components are available through the FPGA: how could it be done?

To first way to make the ACPI tables available pre-boot is to either store them somewhere when programming the FPGA, so that they can be retrieved (e.g. on the BMC, retrievable through EFRI). The second way would be to predefine the base addresses of all possible components that can be synthesized, which is obviously hard and limits the platform flexibility.

Pre-boot only makes sense if these devices need to be available before the OS boots, which is rarely the case. Network interfaces, PCIe busses, SATA Controllers, etc. are all available on the first node before the OS boot, so there should be no situations where the boot process is hindered by this.

From the SBBR specification, these tables are required to boot an OS, further consolidating the opinion that representing the FPGA is not necessary in the firmware. This makes the post-boot choice obvious.

The easiest way I can think of is modifying the ACPI tables after boot using a driver, directly from the OS. This potentially gives us even more control in terms of deciding what is represented, and when.

Also, it may be possible to bring-up and down the ECI link at runtime, maybe allowing the FPGA to be reprogrammed after boot: this requires the post-boot approach to be implemented.

Here are some examples of the hardware that could be added after boot through the FPGA:

Memory Controller The FPGA can implement a DRAM controller to increase system memory. Since Linux supports memory hot-plug it is possible to make it available at runtime.

PCIe controller PCIe can be enumerated again and be hot-plugged, so it's supported.

NICs Similar to PCIe.

In conclusion, further work should verify whether the pre-existing ACPI tables function correctly and then focus on ways to represent additional hardware post-boot, such as the post-boot one.

4 Evaluation

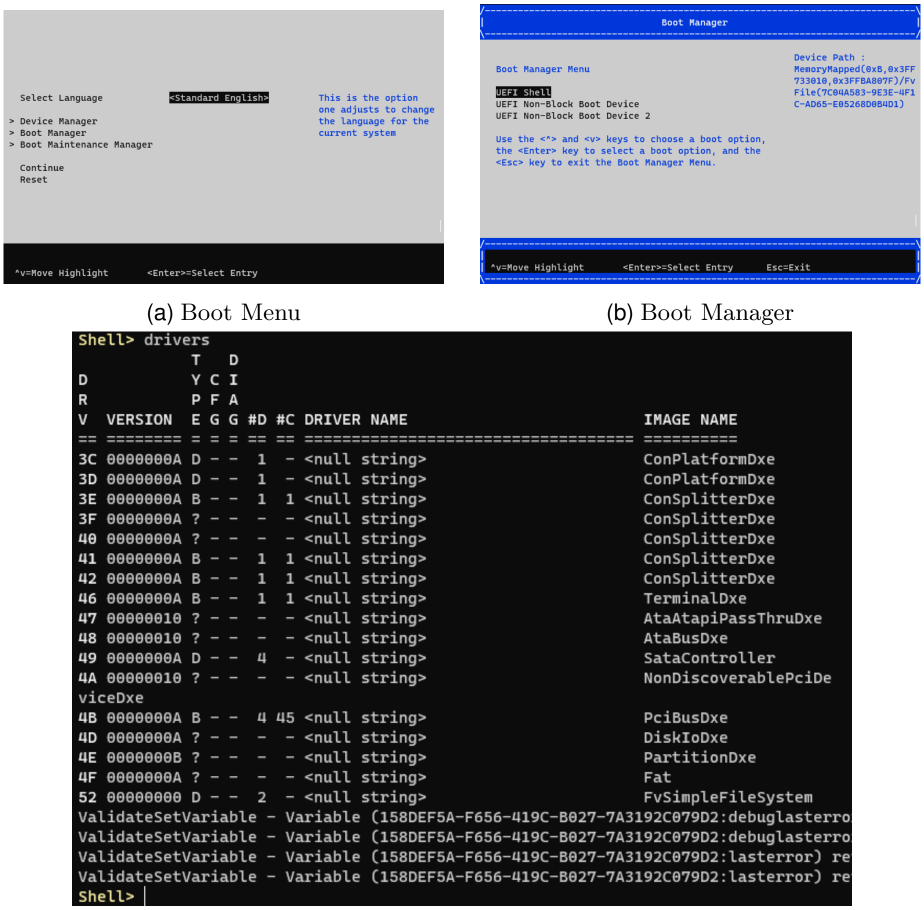
Since the firmware is not able to boot Linux, evaluation possibilities are very limited.

To test the firmware, one of the Enzian prototypes can be used. The old ATF and BDK are required.

The firmware needs to be flashed as described in Section 3.1.3.

After booting the machine, the firmware proceeds through every boot stage, until it gets to the BDS stage. At this point, the *Boot Menu* (Figure 4a) is shown. One can then enter the *Boot Manager* subsection (Figure 4b).

From there it's currently possible to enter the UEFI Shell to execute various commands. Running `drivers` results in a table illustrating the status of each driver (Figure 4c).



```
  Select Language          <Standard English>        This is the option
                                                     one adjusts to change
> Device Manager                                     the language for the
> Boot Manager                                       current system
> Boot Maintenance Manager

  Continue
  Reset

 ^v=Move Highlight      <Enter>=Select Entry
```

(a) Boot Menu

```
                              Boot Manager

  Boot Manager Menu                                   Device Path :
                                                      MemoryMapped(0xB,0x3FF
  UEFI Shell                                          733010,0x3FFBA807F)/Fv
  UEFI Non-Block Boot Device                          File(7C04A583-9E3E-4F1
  UEFI Non-Block Boot Device 2                        C-AD65-E05268D0B4D1)

  Use the <^> and <v> keys to choose a boot option,
  the <Enter> key to select a boot option, and the
  <Esc> key to exit the Boot Manager Menu.

 ^v=Move Highlight      <Enter>=Select Entry      Esc=Exit
```

(b) Boot Manager

```
Shell> drivers
            T   D
D           Y C I
R           P F A
V  VERSION  E G G #D #C DRIVER NAME                         IMAGE NAME
== ======== = = = == == =================================== ==========
3C 0000000A D - -  1  - <null string>                       ConPlatformDxe
3D 0000000A D - -  1  - <null string>                       ConPlatformDxe
3E 0000000A B - -  1  1 <null string>                       ConSplitterDxe
3F 0000000A ? - -  -  - <null string>                       ConSplitterDxe
40 0000000A ? - -  -  - <null string>                       ConSplitterDxe
41 0000000A B - -  1  1 <null string>                       ConSplitterDxe
42 0000000A B - -  1  1 <null string>                       ConSplitterDxe
46 0000000A B - -  1  1 <null string>                       TerminalDxe
47 00000010 ? - -  -  - <null string>                       AtaAtapiPassThruDxe
48 00000010 ? - -  -  - <null string>                       AtaBusDxe
49 0000000A D - -  4  - <null string>                       SataController
4A 00000010 ? - -  -  - <null string>                       NonDiscoverablePciDe
viceDxe
4B 0000000A B - -  4 45 <null string>                       PciBusDxe
4D 0000000A ? - -  -  - <null string>                       DiskIoDxe
4E 0000000B ? - -  -  - <null string>                       PartitionDxe
4F 0000000A ? - -  -  - <null string>                       Fat
52 00000000 D - -  2  - <null string>                       FvSimpleFileSystem
ValidateSetVariable - Variable (158DEF5A-F656-419C-B027-7A3192C079D2:debuglasterro
ValidateSetVariable - Variable (158DEF5A-F656-419C-B027-7A3192C079D2:debuglasterro
ValidateSetVariable - Variable (158DEF5A-F656-419C-B027-7A3192C079D2:lasterror) re
ValidateSetVariable - Variable (158DEF5A-F656-419C-B027-7A3192C079D2:lasterror) re
Shell> |
```

(c) UEFI Shell

Fig. 4: Terminal interface of the Boot Menu and UEFI Shell

The shell in Figure 4c shows that the `SataController` driver is currently managing 4 devices (column `#D`). Instead, the `Ata*` drivers are not managing any device, as they have type **`?`**, in this case resulting from an error during their initialization. The bottom four `ValidateSetVariable` prints are shown due to the verbosity level chosen.

From here, not much else can be done. Various commands can be executed to get information about the system:

- To verify that no SATA drive is currently mapped, the command `map` can be run.
- The command `pci` shows all enumerated PCI devices and their hierarchy.
- By running `drivers`, information about each driver is printed.

Performing benchmarks and other measurements would require the ability to boot an operating system, which is not currently possible.

5 Future Work

To recap, a lot of things still aren't implemented or don't work. Here's a list ordered by decreasing priority of the things I feel like should be done.

1. Re-implement `SegmentLib` with the second proposed approach, described in Section 3.4.4.
2. Fix all the issues in the SATA driver that prevent it from working.
3. Get to successfully enumerating SATA drives and accessing them.
4. Try booting different UEFI applications from disk (e.g. GRUB).
5. Make sure that the PCI driver is fully working.
6. Obtain the memory configuration to correctly map memory, either through a HOB from the ATF or through an SMC call.
7. Implement a EFRI-based UEFI variable store.
8. Port the ACPI tables from the old firmware, this is required in order to successfully and properly boot Linux [21] [10].
9. Implement a way to represent FPGA-synthesized hardware at runtime, directly in the OS.

The following tasks, instead, are *optional*. They are probably good to improve the project further, but are not necessary in order for it to work. Still, they can increase its longevity.

10. Separate the current implementation in `enzian-platform` into `Platform` and `Silicon`. This splits the code in two modules, one that's processor-specific and one that's platform-specific. After, the platform package can include the definitions from the silicon package, making the project more modular.

    The concrete advantage is that, in the future, firmware for a next version of Enzian could share the processor-specific modules with the previous one, with improvements to that being reflected to every other platform.

    Most platforms within `edk2-platforms` [26] are implemented this way.

11. Implement a proper RTC library that retrieves the current time through EFRI, since the proper endpoint has recently been implemented. This is not necessary by any means, but it's a nice-to-have.

6 Conclusion

Contrary to initial expectations, the progress made on the implementation has not allowed to successfully boot Linux. Nevertheless, the still-standing barriers are few and probably simple to overcome.

The differences between the old firmware and the ideal model for a new one turned out to be great: both the structure and the implementation needed to be heavily changed.

With the complete rewrite, the project structure is clear and, as a result, maintenance is easier. Future work on the firmware will not require reverse-engineering a previous implementation that takes very specific tools to set up and build. Instead, it can be quickly set up and modified. Finally, since the structure is similar to the one used for other platforms, one can analyze previous work done on those in order to determine how to develop new drivers.

Overall, getting started with firmware development is a much quicker experience, given the containerized build environment (assuming one has Podman or Docker installed), and the base knowledge required to do so can be found in the background section of this document. Most of the reverse-engineering work has been done, and any further work can benefit from the information obtained.

The current state of the firmware is as follows: It is able to boot from the old ATF, to map some RAM (enough for initial testing), and to proceeds through the boot stages, finally stopping at the SATA enumeration phase; a yet-unresolved issue causes a failure, making it impossible to boot from disk. Then the Boot Menu is shown (with no disks listed) and the user can choose to launch the UEFI Shell. The shell correctly shows all the available PCI devices with The `pci` command. Also, with the `drivers` command, one can verify that the SATA driver is started, while `AtaAtapiPassThru` has encountered an error.

The firmware's performance is currently limited by the baud of the serial port, since the debug verbosity is kept high for development reasons. It can be reduced as desired through the `PcdDebugPrintErrorLevel` PCD. Also, the `DEBUG` build includes very little optimizations, to simplify debugging. The time taken to reach the UEFI Shell is in line with the old firmware with debug info enabled.

Overall this is a solid base that will simplify any further development, from the completion of the firmware itself to implementation of other UEFI drivers.

References

[1] TIC Committee. *Tool Interface Standard (TIS) Portable Formats Specification*. Version 1.1. Oct. 1993. Chap. Executable And Linkable Format (ELF), pp. 1–3. URL: https://refspecs.linuxfoundation.org/elf/TIS1.1.pdf.

[2] Grant Likely. *When Will UEFI and ACPI Be Ready On Arm?* Feb. 3, 2014. URL: https://www.linaro.org/blog/when-will-uefi-and-acpi-be-ready-on-arm/.

[3] Inc. Cavium. *Cavium ThunderX CN88XX Hardware Reference Manual*. Version Pass 2. Sept. 2017.

[4] Tianocore. *EDK II Module Information (INF) File Specification*. Version 1.29. July 2019. URL: https://tianocore-docs.github.io/edk2-InfSpecification/draft/.

[5] Tianocore. *EDK II Driver Writer's Guide for UEFI*. Version 2.3.1. Jan. 2020.

[6] Inc. UEFI Forum. *Advanced Configuration and Power Interface (ACPI) Specification*. Version 6.4. Jan. 2021. URL: https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/20_AML_Specification/AML_Specification.html.

[7] Inc. UEFI Forum. *Advanced Configuration and Power Interface (ACPI) Specification*. Version 6.4. Jan. 2021. URL: https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/03_ACPI_Concepts/ACPI_Concepts.html.

[8] Inc. UEFI Forum. *Advanced Configuration and Power Interface (ACPI) Specification*. Version 6.4. Jan. 2021. URL: https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/17_NUMA_Architecture_Platforms/NUMA_Architecture_Platforms.html.

[9] 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 '22. 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.

[10] Arm Limited. *Arm® Base Boot Requirements*. Version 2.0. Apr. 15, 2022.

[11] Tianocore, Cavium, and ETHZ. Apr. 19, 2022. URL: https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-uefi.

[12] *Docker: Accelerated Container Application Development*. Aug. 19, 2023. URL: https://www.docker.com.

[13] Alessandro Legnani. *Trusted Firmware for a Research Computer*. Bachelor's Thesis. 2023.

[14] Arm Limited. *Arm Development Studio*. Aug. 15, 2023. URL: https://developer.arm.com/Tools%20and%20Software/Arm%20Development%20Studio.

[15] Arm Limited and Contributors. *Trusted Firmware-A*. Version 13.1. May 16, 2023. URL: https://trustedfirmware-a.readthedocs.io/en/latest/.

[16] Marvell. *PciSegmentLib (Armada7k8k)*. Aug. 15, 2023. URL: https://github.com/tianocore/edk2-platforms/blob/84ccada59257a8151a592a416017fbb03b8ed3cf/Silicon/Marvell/Armada7k8k/Library/Armada7k8kPciSegmentLib/PciSegmentLib.c.

[17] *Podman: A tool for managing OCI containers and pods*. Aug. 19, 2023. URL: https://podman.io.

[18] Inc. UEFI Forum. *Unified Extensible Firmware Interface (UEFI) Specification*. Version 2.10. Aug. 15, 2023. URL: https://uefi.org/specs/UEFI/2.10/14_Protocols_PCI_Bus_Support.html.

[19] Pengcheng Xu. *Enzian Firmware Resource Interface*. en. Student Paper. Zurich, Feb. 2023. DOI: 10.3929/ethz-b-000603460.

[20] Intel. *Serial ATA Advanced Host Controller Interface (AHCI)*. Version 1.3.1. Chap. 3.1 Generic Host Control.

[21] Al Stone et al. *ACPI on ARMv8 Servers*. URL: https://www.kernel.org/doc/Documentation/arm64/arm-acpi.txt.

[22] Tianocore. *Build Instructions*. URL: https://github.com/tianocore/tianocore.github.io/wiki/Build-Instructions.

[23] Tianocore. *EDK II Flash Description (FDF) File Specification*. URL: https://tianocore-docs.github.io/edk2-FdfSpecification/release-1.28.01/.

[24] Tianocore. *EDK II Module Writer's Guide*. Chap. 8.4 Communication between DXE Drivers. URL: https://tianocore-docs.github.io/edk2-ModuleWriteGuide/draft/8_dxe_drivers_non-uefi_drivers/84_communication_between_dxe_drivers.html.

[25] Tianocore. *tianocore/edk2*. URL: https://github.com/tianocore/edk2.

[26] Tianocore. *tianocore/edk2-platforms*. URL: https://github.com/tianocore/edk2-platforms.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Boot Firmware for Heterogeneous Systems running Linux

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Montini	Axel

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date

Taverne, 21/08/2023

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.