Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 495b

Systems Group, Department of Computer Science, ETH Zurich

Firmware Management for a Heterogeneous Platform

by

Julian Elyes

Supervised by

Daniel Schwyn
Prof. Dr. Timothy Roscoe

February 2024 – August 2024

D INFK

Abstract

Heterogeneous hardware, much like its conventional counter parts runs firmware and other low-level software like bitstreams in the case of FPGAs. Current firmware management solutions only focus on very specific hardware or do not focus on enough variety regarding their hardware support like FPGA programming. A lack of firmware management can lead to longer downtime, which in turn leads to significant profit losses in a commercial setting, something that should ideally be minimized. As a result, in this thesis, we propose a starting point for managing firmware for all kinds of different hardware by walking through the design process and presenting an implementation specific to Enzian machines. We then evaluate our approach by first qualitatively reviewing the subsequent implementation with the help of usability testing. The subsequent qualitative performance evaluation using the proposed firmware manager shows that by leveraging the distributed property of server clusters, we can lessen downtime by an upper bound of the amount of machines present in the network, which can save costs by an order of magnitude.

Acknowledgements

I would like to thank my supervisor Daniel Schwyn for his invaluable insight and enthusiasm in illuminating concepts that would have otherwise gone way over my head. Additionally I want to show my appreciation to Prof. Dr. Roscoe for enabling this project in the first place and of course everyone in the Enzian Project Group, who have always provided me with excellent and kind feedback towards my ideas as well as misconceptions. Last but not least I would like to express my gratitude towards my family and friends, which have always supported me throughout my studies and of course this thesis.

Contents

1 Introduction **7**

2 Related Work **9**

3 Firmware and Metadata **11**

3.1 Content 11

3.2 Format 13

3.3 Bundling 15

4 Design of the Firmware Engine **18**

4.1 Devices 19

4.2 API 23

4.3 Plugins 28

4.4 Sub-Engines 30

5 Implementation and Discussion **32**

5.1 Implementation Details 32

5.2 Enzian Plugin Module 32

6 Experimental Evaluation **36**

6.1 Heuristic Usability Evaluation 36

6.2 Quantitative Evaluation 39

7 Conclusion and Future Work **46**

References **48**

Appendices **51**

A Plugin Base **51**

B Measurements **51**

List of Figures

1 The initial layout of the system, arrows indicate that the source implements the destination 20
2 The current layout of the system, arrows indicate that the source implements the destination 21

List of Listings

1 A way to flash the CPU. 18
2 A way to program the FPGA 18
3 The semantically more natural structural option. 23
4 The robotic sounding option. 24
5 Usage of the add-device command. 24
6 The syntax for listing all devices of the Firmware Engine 25
7 The query command displays information about live firmware 25
8 Using the all keyword with the query command invokes it on all devices. . . 25
9 Usage of the flash command in case a file is available. 26
10 Not specifying a file gets the latest stable release. 26
11 All devices get flashed with their stable release with the `all` keyword. . . . 27
12 Rolling back a specific device 27
13 Rolling back all devices 27
14 Adding a Sub-Engine needs its ID, username, hostname and its root. 30
15 Listing all available Sub-Engines. 31
16 Usage of the Sub-Engine API 31
17 Get the Redfish authorization token via Python. 33
18 A potential flash implementation for the BMC. 34
19 The default timing methodology used. 40
20 Setting up a CPU device. 40
21 Setting up an FPGA device. 40
22 Setting up a Sub-Engine. 40
23 The query command to time. 41
24 A call with the help flag as a type of control group 41
25 Flash the CPU with an empty flash implementation. 43
26 Use the Sub-Engine API to flash all CPUs at once. 44

List of Tables

1 Results of measuring the query API, the help flag and the Wiki page. . . . 41
2 Results of measuring the stubbed flash API and the Wiki page in edit mode. 43
3 Average time heuristics of web page related events. 43
4 Results of measuring the sub-engine API with 1-3 machines as well as flashing a single machine manually. 45

1 Introduction

Heterogeneous systems have become ubiquitous in today's computing landscape. Take for example the Enzian research computer built by the Systems Lab at ETH Zurich [1], which consists of a 48-core server class CPU and a powerful Xilinx FPGA connected together with the help of the so called Enzian Coherence Interface (ECI) and managed with the help of a Baseboard Management Controller (BMC).

Just as heterogeneous hardware has evolved over the past decades, so did the associated software, which has to not only be versioned but also tracked on these computers, which if done manually like in the case of Enzian is incredibly error-prone and will not scale well at all. Since diversity of hardware and its corresponding firmware or software will not halt in the foreseeable future, a system that can support heterogeneous system configurations in querying and flashing firmware is in order.

To understand why this is necessary in the first place, we look toward the industry. Not knowing what version of firmware is active on hardware can lead to unexpected and prolonged machine downtime in case of a faulty update, a bug crashing a machine or a potential security flaw that lies dormant until found. Downtime can in turn cause massive losses to businesses and costs 91% of enterprises over $300'000 per hour and rises to an incredible $1'000'000 per hour for 44% of medium and large enterprises [2]. The majority of downtime that costs these enterprises such huge swathes of money are caused by security breaches which can in some cases also be linked back to firmware mismanagement.

Take for instance Intel's Security Advisories, specifically the INTEL-SA-01080 advisory [3], which informs of a highly severe security vulnerability within UEFI firmware for various Intel Server Products. These vulnerabilities could lead to an escalation of privilege or denial of service by a malicious party. The solution that Intel advocates is simply a firmware update, in case any of the affected hardware families were using the vulnerable firmware. This is where firmware management comes in. In this case an admin should be able to check what firmware is running and update it in the event of a security compromise. These security advisories show that being able to initiate and conclude firmware updates as quickly as possible will lower costs associated with the prolonged downtime of having to take machines offline to keep them out of harms way.

To begin with, it is important to understand what is meant by firmware. Firmware is understood as a type of software that is run from non-volatile memory on startup of a machine and provides access to low-level control for its hardware. For the sake of brevity, FPGA bitstreams are counted as firmware too. The reason is that specifying **firmware/-software** each and every time is very cumbersome and impacts readability too. Unique to this thesis, both traditional firmware and any type of software that configures hardware, like bitstreams, are included in the class of **firmware**.

Regarding the use of the term `user`. In a typical corporate setting, someone who builds firmware is usually different from someone that flashes and queries firmware. The typical user of a firmware manager for the Enzian platform can be both a researcher and someone with an administrative role. Building custom firmware is not out of the question for a research oriented platform and reverting these custom versions to normal stable releases is just as valid a use case.

To give an overview of this project, we will first be looking at projects and works that give us a rough idea of the current situation of firmware management and how they would fare with a heterogeneous system. Then we will take a look at what information to display when querying firmware, what format this information should take and how to bundle it with the firmware image. We then propose a design, that sets the foundation of the firmware manager, by walking through the entire process and providing the rationale behind each design decision. In the next step we list implementation details that were not substantial enough to be considered design decisions and provide an implementation that works specifically in the context of the Enzian ecosystem. With the implementation as a reference we then evaluate it qualitatively using usability testing as well as quantitatively through performance measures to show that it is both a usable and useful firmware manager. Finally we will be concluding the work and giving a brief overview over future work that could subsequently be done.

2 Related Work

Evidently it is in everyone's best interest to stay secure and keep downtime to an absolute minimum. This is a very broad problem that can be mitigated by a well designed management system, the question remains if any such systems have already sprung up in response to such grave vulnerabilities.

fwupd The first such project that comes to mind is `fwupd`, a system daemon with the aim of making the process of firmware updating automatic, safe and reliable. It does so by sourcing updates through the Linux Vendor Firmware Service (LVFS) and letting original equipment manufacturers write custom firmware set up programs and uploading new versions of firmware [4]. The main purpose of `fwupd` is to facilitate the firmware update process for Linux users by offering OEM's the chance to share all of their updates through a central hub. One problem is that fwupd fundamentally does not support flashing hardware on a distributed scale like in the Enzian setting. This means that flashing and querying hardware can only happen on a per machine basis. Additionally, fwupd does not natively support the programming of FPGAs

Dell OpenManage Enterprise Another tool with a similar function is Dell's OpenManage Enterprise (OME) [5], which is an infrastructure management software specifically tailored for Dell servers. It has dozens of features made for large server infrastructures, including firmware management. While there is a plugin functionality contained within the software, it was not necessarily designed to let non-Dell devices use the same functionality. Other devices would see restricted functionality if workarounds were implemented, which is undesirable considering the size (1.5 GB) with OME. In addition to the lack of official support for non-Dell devices, OME is also not open source, which makes the implementation of custom hardware even more difficult.

Intel Server Firmware Update Utility Moving on to Intel's Server Firmware Update Utility [6], which has the ability to update any server related Intel component. The tool can be put in the same class as Dell's OME since it is not open source either and will be quite a lot of work to implement support for non-Intel devices. This should not come as a surprise but as with OME, looking for workarounds is not worth the minimal subset of features that could work in the end.

SBOM In section 3 we will be looking at how we decided to format and bundle metadata information with a firmware image. In the aftermath of that section, there are related projects focusing on System Bill Of Materials (SBOM), which is a more advanced version of what is described in section 3. Examples include the widely used Software Package Data Exchange (SPDX) [7] and the more recent CycloneDX [8]. SBOM can help mitigate security vulnerabilities since every dependency and component of the associated software must be mentioned.

DMTF Redfish Redfish [9] is a standard by DMTF that uses the RESTful [10] interface to allow access to a schema based data model for management operations. The standard is used in OpenBMC [11], an open source implementation of a BMC firmware stack that works across heterogeneous systems. Managing firmare of both a BMC and a host's BIOS is possible through Redfish but the programming of FPGAs unfortunately lies outside of the scope of OpenBMCs firmware related endeavours.

Not all of the listed projects and standards are full firmware managers by themselves, e.g. SBOM is used to keep information about firmware. Redfish by itself also is not specifically designed only for firmware related functionality and as such does not support things like programming and FPGA. Intel's Server Firmware Update Utility and Dell's OpenManage Enterprise are designed with their respective server infrastructure in mind and do not natively support custom hardware, which would be especially difficult to add without access to the source code in the first place. Fwupd is the closest to a fully functioning firmware management solution for Enzian but unfortunately also does not support FPGA programming.

Evidently non of the projects are fully compatible with a heterogeneous system such as Enzian and will not work out of the box. This warrants a fresh design that can make firmware management on Enzian as usable and quick as possible.

3 Firmware and Metadata

There would be no point in managing firmware if we did not at the very least know what metadata is running on a device. That was the original problem we set out to solve and this section is entirely dedicated to explaining how we realize that goal and other problems that arise from it.

Usually, files store data which can be used for all sorts of purposes. A file is not created in a vacuum though, and will have information surrounding how it was created, what it was created for, when it was created etc. This kind of information is called metadata and is, as the name suggests, data about data and is the class of data we are interested in querying from firmware. To begin querying firmware metadata, we must first define a schema, collect the types of information defined in the schema and store it in a format such that it can be associated with the firmware without being lost.

3.1 Content

Of course we would like to version firmware, but what exactly is a version? Looking at the Enzian platform, each piece of firmware and software can be traced back to a specific git hash associated with a specific commit in a repository. Ideally, version numbers should be totally ordered so that a user can easily recognize at a glance which of two versions is newer but that would only be an added bonus and not its main function. In the case of Enzian the definitive identification of firmware with its source code is the only important part the versioning must accomplish, which is why the git hash should suffice.

Version numbers are not the only type of metadata, there is lots of other information that could be useful for debugging and further identification. There is a lot of metadata to collect but we had to decide on a considerably smaller subset for practicalities sake. The following paragraphs will enumerate them and explain the thought process behind their inclusion.

Version Number The main identifier for firmware would be the version number. There are lots of different versioning schemes, one of the most widely used ones was formalized into a specification and named "Semantic Versioning" [12]. While this type of versioning is a great way to communicate what changes were made and what version is chronically superior to other versions, it is quite overkill to use in the context of the Enzian platform. The only requirement is to be able to identify firmware to its source repository commit, since all other information can be deduced. If the firmware does not originate from a git repository, then "unknown" would be displayed instead of the version number. Specifying a firmware's git branch is not needed since commit hashes are unique even over different branches [13] since branches just point to commits and do not change the underlying way they are hashed.

Target Devices While security is not in the scope of this project and a general respect to the Enzian platform is expected of anyone with access, giving a firmware manager the option of checking if the underlying firmware can be used with a device is a very useful safety. The idea is not to keep malicious actors away, but to minimize human error. If the field is empty the manager will ignore the check. Otherwise a comma separated list of device names (see subsection 4.1) is expected.

Build Type The build type of a firmware can be specified to relinquish a bit of additional information for anyone inquiring, be it a release, debug, experimental or alpha build. The point would be to give users information about why something could be functioning differently than perhaps expected. An example would be if the firmware was built with the debug flag set without the users knowledge. If at some point the user notices a performance discrepancy because there were no optimizations done, then checking the build type in the metadata should dispel any doubts about what is going on. The build type if not specified will be displayed as ”release” by default.

Git Repository Being able to identify firmware through commit hashes should usually even work over multiple repositories, but there theoretically are not any guarantees about that whatsoever. Not only that but if a user would like to easily figure out which repository a firmware was built from without having to cross-reference the hashes, then a field containing the git repository would be quite a welcome addition. In case firmware does not originate from a git repository, then the build directory is used.

Build User It could be argued that putting usernames carries a discussion to be had about privacy and data collection but in the context of the Enzian platform any user will have a known username registered by the ETH Systems Group which could be used instead of the name of a user on a private machine. Nonetheless, knowing which user built what firmware can help administrators in tracking whom a problem could have originated from. To help with such investigations a field for the user that built a firmware image should be present in the metadata.

Build Machine It is very nice to include the build user but it does not really seem useful if we can not pinpoint which machine was used in the first place. Users are not unique outside of their respective, which is ambiguous. To alleviate that metadata should also include the build machine if a build user was included.

Build Date and Time Since the version number can not communicate its chronological release, we will have to include the build date and time. This way a user can easily tell which firmware is newer, which could help with related decisions. Additionally, including the full time could also help with investigative work e.g. in case administrators would like to track how a bug entered the system.

Security While not the main goal of this project, there are definitely security concerns as it stands right now. One of these concerns is metadata spoofing. This could be alleviated through some sort of attestation scheme or by incorporating proof-carrying code [14] into the firmware and metadata this would slot into further work and discussion of the implementation.

We have now decided on the general types of information to be included in the metadata and why they should be saved. There are most definitely many other kinds of information someone would like to include, which is why it should not pose too much of a problem to add more. The content of the metadata is saved in a directory in the root folder of the Firmware Engine as a schema in a Python [15] file and can be changed any time.

3.2 Format

If we want to version firmware composed of multiple parts, then a versatile format that can be parsed through the BMC's Python interpreter is needed. We do not have to reinvent the wheel when it comes to data serialization since we already have a large amount of formats to choose from, including XML, CSV, JSON and YAML. A summary of the relevant functionality of the formats are as follows.

XML The Extensible Markup Language [16] is a plain text serialization format devised by the World Wide Web Consortium with the goal of supporting a large variety of applications all over the internet. The language was designed with ease of parsing in mind as well with minimal optional features as possible.
Python supports XML through various modules and support for dictionaries can be easily implemented if external modules are not allowed.

CSV Comma-separated values as a format can be described more accurately as a collection of very similar formats since things like delimiters or quoting characters can change from one user to another. These formats lend themselves well to tabular data or databases. An attempt at describing the format in a standard was made [17] but since the format existed long before, has not been taken too seriously.
Nonetheless, Pythons support for all sorts of CSV sub-formats through its "csv" module is impeccable with a built-in implementation for reading and writing dictionaries from CSV files.

JSON JavaScript Object Notation [18] is a very versatile format to serialize structured data in. The goals of the format is to standardize the way structured data is shared on the internet which lends itself well in our situation, where complex yet structured data must be saved.
There is a "json" module present in the Python standard library already, with built in ways to convert to and from dictionaries just as easily as with CSV files.

YAML "YAML Ain't Markup Language" [19] is a data serializaiton format with the purpose of high portability and readability in mind. It is mostly used for configuration files and would probably fare just as well as a metadata format.
Unfortunately, Pythons support for YAML in the standard library is non-existant and is alleviated by the PyYAML module, which must unfortunately be installed separately.

Custom Binary Format While designing a custom format which uses binary representation is possible, the time cost of not only designing but also documenting and implementing a rigorous parser would be too large to even consider. While storing the data in a binary format could be much more efficient, in the context of the Enzian platform it seems much less relevant to save space since storage is so readily available.

The only real hard requirement for the serialization format is that it must have a Python module for ease of integrating into the Firmware Engine and the ability to save more complex types of data like Pythons dictionaries. In our evaluation, human readability as well as a Python module integrated into the standard library were mostly nice to haves but ended up being the main reason for our choice. This is because all proposed formats ended up having a Python module capable of converting between itself and Pythons dictionaries. In the end, JSON and CSV have dictionary support from the standard library while JSON and YAML are the most readable formats. Even if YAML is more readable, JSON would still win out in total since there is no need to install any external packages.

A big consideration against JSON is that even if readable, it is still quite verbose and not very space efficient, especially if trying to embed it into a firmware file. In the end there is a choice to be made between high versatility, allowing for complex data to be saved, or a minimal footprint that takes away the least amount of file space possible.

The final format of the metadata is predominantly an implementation detail decided through the language used. If we were to implement it through a different language, like e.g. C, the decision could very well change to a more custom implementation. That again depends on how many dependencies we would limit the implementation with. The type of archive or if it is compressed is also a minor implementation detail that could be decided by all sorts of context specific factors. In this we used an uncompressed tar ball, even if there were lots of other supported types (cite supported types in shutil) in Pythons `shutil` module (cite shutil).

3.3 Bundling

We have now figured out which data serialization format to use with the complex structured data required by the multi-part firmware of the ThunderX CPU included in Enzian. We evaluated that JSON is the most suitable since it is easily incorporated into existing Python code bases and is relatively readable by a human, even if a bit verbose. What we now have to figure out is how to bundle the JSON metadata with its corresponding firmware file. There are (again) two ways of doing this.

Sidecar Files This approach sees the main firmware accompanied by a metadata file in the same vein as how a motorcycle can have its own associated sidecar, hence the name. It is an apt comparison since a motorcycle can function without a sidecar but a sidecar without a motorcycle does not see much use. The same is true for sidecar files, which are only useful as long as they can be accompanied with their main parent file. This is also one of the main points against sidecar files since there is no real way to glue files together without adding another hierarchical layer. The trust is therefore placed with the user to keep the firmware and the metadata together. One way to alleviate this weakness is to bundle the two parts in an archive. This also simplifies the sharing of firmware if it is comprised of multiple non-metadata files too so it does seem like a decent solution. When changing firmware to be archived we must also change existing tools to work with the new file type instead of just the base firmware, this could end up taking quite a bit of time. The payoff however is a way to keep any type of firmware and metadata together all while minimizing the burden carried by the user. The added versatility may be worth the extra work.

Embedding The other way of bundling firmware and metadata together is by embedding the metadata in the firmware file. This of course has as a requisite of there existing only one firmware file or at the very least a main file. Another consideration is that to embed anything in a firmware file, a lot of research and care must be put into actualizing this approach, since just prepending or appending the metadata at the beginning or end of a file may not work easily. This is especially the case for firmware and bitstreams, which have very specific places they encode metadata in. This means finding a suitable spot, usually in the header portion, and may not lend itself as easily to keeping complex structured data. On the other hand, if metadata is embedded, it can not be lost by accident and could even be read from running firmware in the associated flash storage.

These are the main options with their own advantages and drawbacks. Sidecar files are very versatile and only need some preexisting tools that use the current firmware format to change by allowing archives. Embedding information in the firmware needs a lot more specific knowledge and even then it may not even be possible if the firmware is already incredibly compact, which would have to warrant an overhaul of the files overall structure.

This would definitely take way longer than just adding archive support to a handful of tools. An example of where embedding information would pose too much of an investment for minimal return are FPGA bitstreams. While there exists a USR_ACCESS register [20] that can be configured when programming an FPGA, it can only hold a 32 bit number, which would only suffice at most for a git hash or a time stamp. It is actually possible to embed information at the start of a Xilinx bitstream since the FPGA device would only start processing configuration packets after a specific sync word `0xAA995566` [21]. The problem in this case is that it will break any tool like Vivado relying on the bitstream header to display additional information.

In the case of the ThunderX multi-part firmware incorporating the metadata for all parts at the very beginning of a unified image was successfully done and can even be read from the SPI flash which is forcibly shared between the BMC and CPU. Even if embedding successfully work, the man-hours invested in actualizing this goal was simply not worth it compared to just incorporating additional support for an archived file that holds both parts, metadata and firmware.

There is technically another solution in between the sidecar and embedding approach, which would count as a custom file type. The idea would be to prepend the metadata onto the firmware image and let the manager take it apart by including the size of the metadata part in an additional field. Before flashing the manager would save the metadata part and flash the normal firmware part only. This approach would need the same amount of work as the sidecar approach while enabling users and programs to read metadata without unarchiving the file first. Sadly in the case of multi-part firmware, which is not a single file, this approach would run into the same problem of not knowing where to prepend the information since there are multiple files to choose from and vice versa for reading it back again.

In the end using the sidecar approach is the most simple solution. Even if having to unarchive a file to read its metadata seems a bit arduous it is still less of a headache than thinking of all the ways we would have to embed the information into a file, be it prepending it and splitting it before flashing or researching where to hide and retrieve it in the original firmware. Not only that but a user can still unarchive the file and use the firmware file just as before, without introducing new problems.

The end result is something quite similar to the SBOM formats introduced in section 2 but quite a bit simplified. This simplification can in turn help us since we will not have to worry about the strict SBOM format and can prototype quickly, which is important when designing a system from the ground up.

In this section, we first defined that metadata should contain the version number, target devices, the build type, the source directory or git repository, the user who built the firmware, the machine which the firmware was built on and the time and date the firmware was built too. Then we went on to figure out how to serialize this information and chose to use JSON for its readability and ease of integration into the existing Python code base.

Last but not least we decided how to bundle the metadata and firmware together and ended up using the sidecar approach for its least invasive and time saving characteristics compared to the other options. This was done with the goal of being able to version each new piece of firmware that will be built in the future. The next step would be to be able to query this information somehow, that is where we get into the idea of the firmware manager.

4 Design of the Firmware Engine

This next section is dedicated to the description of the firmware manager designed to keep track of firmware for the Enzian platform's BMC, CPU and FPGA respectively.

This firmware manager is called the Firmware Engine since it functions as a guiding scaffolding. That is because its primary function is to present a simple abstraction to the user that can query firmware version information. The problem is that the framework can not query information if it never gets access to the firmware in the first place. This sets a precedent that the framework should be designed such that users would want to use it to flash and not just query firmware. So how do we compel users to use the framework instead of the usual command line tools? Create an easier to use command line tool which replaces the multitude of other tools. Users would most definitely just want to use 1. simpler commands and 2. use the same command structure for each hardware component.

To really drive home the point,the established routine to flash the CPU firmware on an Enzian is shown in Listing 1.

```
$ flashcp [firmware_file] /dev/mtd4
```

Listing 1: A way to flash the CPU.

While not overly tedious compared to what the framework would do, it still does not save information that users can query at any time about the firmware being flashed, something the framework would do automatically.

Additionally, programming the FPGA with a bitstream would require the commands shown in Listing 2.

The first line copies the bitstream file to the needed location and the second line writes the name of the bitstream file to a special file to trigger Linux's FPGA Manager to program the device with the bitstream.

```
$ cp [bitstream_file] /lib/firmware
$ [bitstream_file] > /sys/class/fpga_manager/fpga1
```

Listing 2: A way to program the FPGA

This seems like way too much work just to program a bitstream onto an FPGA so we would like to abstract most of that away such that it becomes a simple command line function in the vein of `flash [fpga] [bitstream]` and the rest of the work would be handled by the framework.

To do this, we will have to introduce the concept of devices, which will play the role of representations of the BMC, CPU and FPGA that the framework can actually work with.

As an added note the plan is to implement the manager such that it can run from a BMC and keeps its state by saving it as separate files in `/var/lib/firmware-engine` as the root of the application. In true Unix fashion almost no information has to be kept in memory at all times and can instead be read from files and folders in the file system. A list of devices is e.g. not kept in memory but can be aggregated by scanning all folders in the directory that keeps all device specific data. This lets us use the program without needing to set it up as a daemon that runs on startup. While it is not out of the question to run the manager as a daemon and may even count to the future work, it makes prototyping much easier only having to run a simple Python program. While the examples are for Linux, the same could be done for other operating systems by keeping the managers root directory in the usual application folder e.g. for Windows it could be the `Program Files` folder and adding the firmware managers script to the `PATH` variable.

There are some guiding principles that we tried to uphold during the design of this firmware manager. Remember that the problem to solve is diminishing the impact of downtime caused by the manual firmware management practices currently used on Enzian machines. Therefore, the following constraints can be established to solve this problem:

(1) **Query Information**: Querying the firmware associated with hardware is the main objective and requires the manager to source such information with a method like is described in section 3.

(2) **Support for FPGA programming**: A firmware manager must be able to accommodate all sorts of devices and since Enzian possesses a powerful FPGA they should be included.

(3) **Usability**: Documenting every component a user could come into contact with is of utmost importance in order to minimize confusion and frustration, which could lead to a refusal to use it in the first place.

(4) **Lower Downtime**: The firmware management of Enzian machines has been performed completely manually up until now. Prolonged downtime can hurt profits in businesses and can lead to needlessly frustrating research situations. This is why the firmware manager should be able to lower the downtime when performing firmware operations, including flashing and querying.

(5) **Open**: The idea of Enzian is that of full control over the whole system, as such, having black boxes that researchers can not inspect goes against that. The majority of software for Enzian is open source, and a firmware manager for the platform should be too.

4.1 Devices

We must first start with the definition of a device, the reason being that all functionality of the Firmware Engine completely relies on this very concept. This section bears the

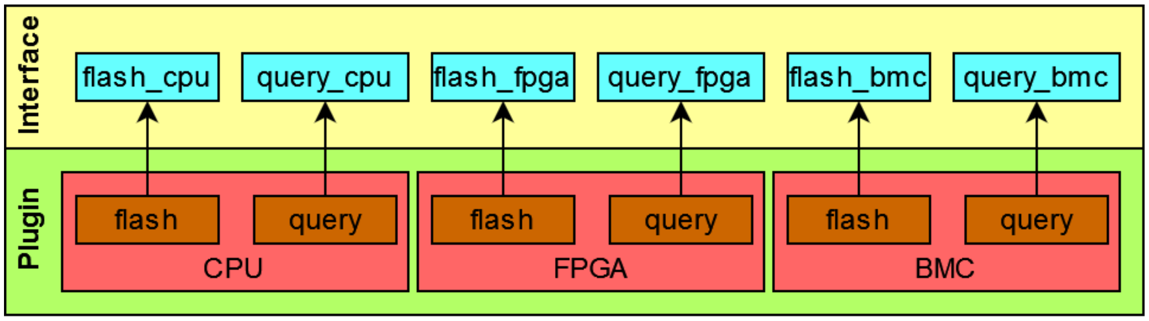

Figure 1: The initial layout of the system, arrows indicate that the source implements the destination

responsibility of explaining how we initially came up with the original idea and the what the actual concept of devices is.

When initially coming up with a way to manage firmware on the Enzian platform, the first naive instinct (see Figure 1) was to write a script for each node, and then connect them together in a master-script, an interface playing the part of the manager. This means that there needs to be a specialized script for the CPU, the FPGA and the BMC, which then are used by another script defining the API. While trying to realize that notion, it quickly became clear that there were too many functions being defined in the code base. To draw out an example, every one of the three nodes had to have hard coded function definitions for flashing aptly named: `flash_cpu`, `flash_fpga, flash_bmc`. The problem becomes even more apparent when looking at the query function, which pretty much always does the same thing, which is reading a JSON file, but the API would still need a function definition for each node: `query_cpu`, `query_fpga, query_bmc`. To remedy this bad way of exposing the scripts through an API, we devised that there should be commands that are shared with all hardware components.

To draw out an example, the CPU, FPGA and BMC all need to be flashed and they all need the ability to be queried. This is where the idea of hardware agnostic commands come in, but for that we need to abstract away the hardware itself. Using this new abstract way of viewing the hardware nodes, now called devices, we can just have a flash and query command respectively (see Figure 2) which then needs the device identifier to identify which script to use. This design minimizes the amount of functions that need to be defined on the API side while also being able to support new devices easily by adding another script.

The idea of using a script to manage firmware gave rise to the plugin system. It is easier to imagine the API design as a sort of frontend, since that is what a user of the Engine would actually care about. To every frontend, there exists an appropriate backend that houses the actual implementation. The plugin system is that backend, where all the API functions get routed to through the Firmware Engine. If a user e.g. calls the query

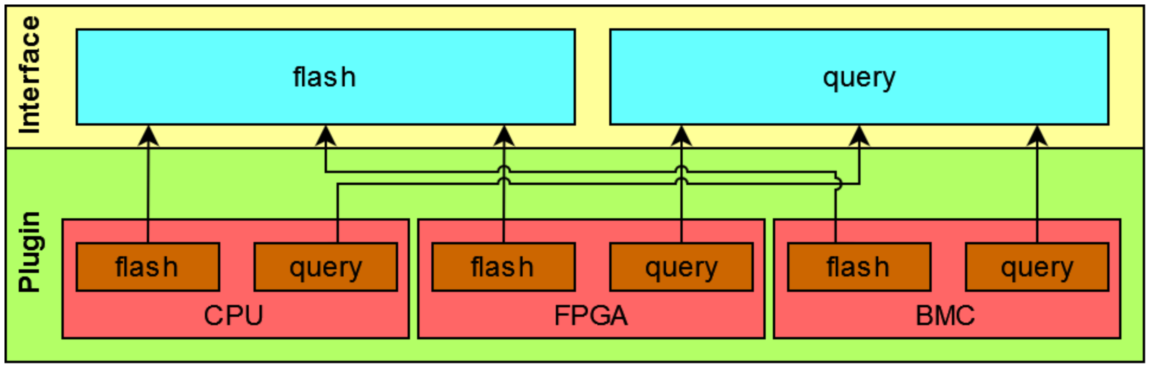

Figure 2: The current layout of the system, arrows indicate that the source implements the destination

command for a specific device, then the Engine would search for the plugin pertaining to the device in question and proceed to call the concrete query function contained therein. This functionality was a natural consequence of wanting to reuse the previously written scripts for the Enzian's CPU, FPGA and BMC while also taking advantage of the newly introduced abstract devices.

The reasoning for calling a device abstract is that it only represents and interacts with the hardware through the plugin module, which of course is not too far removed from how drivers work.

Regardless of its lack of concreteness, a device must be able to be uniquely identified and associated with a hardware component. The first question to ask is why do we care about identifying hardware and the follow up, how do we define a unique hardware identifier in the first place?

The answer as to why we want to identify hardware uniquely is that we must be able to keep devices that are flashed differently apart. There would be no way of specifying the device to be flashed if we can not define the identifier for it first. Note that an identifier should be unique from the point of the host machine connected to these devices. This means that CPU's of the same model could be present on multiple different host machines and share their identifiers. This could enable a database of plugin implementations (see subsection 4.3).

You could argue that a whole CPU package nowadays consists of multiple physical cores, but they should not be recognized as their own devices since the whole CPU package must be flashed together. This is a guiding principle behind the question of what constitutes a device, since we started out defining a device as an abstract representation of flashable hardware.

Options for identifying a device include purely number based identifiers and unicode string names.

Number based identification When using numbers to identify a device we would most probably use UUIDs to minimize collisions and because of its ubiquity since generating a UUID is as easy as invoking a command on most computers nowadays. The advantages of number based identification is that we can easily generate them without worries of collisions but at the cost of obfuscating the real describing names of devices for users.

Unicode string names Using string names means that we would give devices their descriptive product names. An example would be the Enzian's FPGA would be called `Xilinx CVU9P FPGA`. The problem comes with components like Enzian's CPU, whose product name is `Marvell Cavium ThunderX-1 CN8890-NT CPU @ 2 GHz (48 x ARMv8.1 cores)`. This is nearly as problematic as typing out a whole 128-bit UUID.

This seems like quite a predicament but luckily humans can work better with readable text compared to seemingly random numbers. To alleviate the pain of having to type in 60 characters just to flash a processor, we propose a smarter system that does the heavy lifting of identifying devices from their string names. The firmware framework will have to find the device name containing a user input substring and refuse to continue if a substring could refer to multiple different devices.

To illustrate an example suppose the following two imaginary devices `Made Up Company YRT62` and `Different Firm RRT65`. If a user inputs the substring `RT6` then the framework would refuse to flash anything until the string is altered. Adding any non-white-space character could clear up any misunderstanding between the two names. Most people are not so terse and would probably input something closer to `RRT65` or `YRT62`, which would be closer to the intended usecase.

We can extend the functionality such that names are space separated lists and instead of user input being a simple substring, we can view it as a space separated list of filter-strings. In this case the framework will have to filter the device names – now lists of strings – with the user input list of filter words. If after filtering there still exists ambiguity on which device was meant, then the framework will just refuse to continue. As an example, consider the previously used imaginary devices and the user input `Different RT6`. The filter words would end up being `Different` and `RT6`. When applying the filter we start with [`Made Up Company YRT62`, `Different Firm RRT65`], apply the filter `RT6` and keep the same prospective names. When applying the filter `Different` we end up with the definitive name, namely `Different Firm RRT65`.

In the example of Enzian the CPU could be referred to with all of these different names: `Marvell, Cavium, ThunderX, ThunderX-1, CN8890-NT, CPU` etc., and its variations too so user inputs could look like this: `Cavium ThunderX, Cavium CN8890-NT, ThunderX CN8890-NT, Marvell CPU` and more. While some of these names are more descriptive than others it definitely beats remembering and inputting 128-bit UUIDs. As an added bonus the real identifying name can have even more descriptive information in it without forcing the user to remember it fully. In the case of the ThunderX a user can refer to it as `CPU` as long as no other device has CPU in its name. This can prove very useful if most devices and their hardware types are different from each other. In Enzian's case we can have fully descriptive device names while still referring to them as `CPU, FPGA, BMC`.

Note that this device system assumes that there is only one instance of each hardware component, therefore only one instance of a device. To extend the system to support multiple instances of the same kind of hardware we would need to include a device-type, which would use the current device identification. Then we would need to define the new device identifier to be whatever the backend needs to uniquely identify it compared to other devices with the same device-type e.g. partition name, hostname etc. In the case of the Enzian ecosystem this was not needed, which is why it was not included in the design process.

Now that we have decided how to uniquely identify devices, we will have to define how to associate them with hardware, since a device is just a name and needs some sort of connection to a flashable hardware component. That is where the plugin system comes into play. However, before diving into the description of the plugin system, which is the backend of the framework we should first take a look at the API design, in other words the frontend.

4.2 API

We have set the ground work for querying firmware of hardware components by defining what information to store, how to bundle the information with the firmware and how to uniquely identify hardware components with their abstract counterparts that the framework can work with. The next step is to design an API that enables users to interface with the devices defined in the preceding subsection 4.1. Please note that each API call is defined by a Python function in the frameworks code base and is exposed to the terminal by parsing the Python code base and converting each non-private (not prepended with an underscore) function into a terminal command.

It will become apparent that we chose the command line structure corresponding to Listing 3 as opposed to Listing 4. That is not only the case because it is grammatically more natural but also because it is simply easier to implement. The first option is more akin to an imperative in the sense of "Engine, flash this device" instead of the more robotic sounding "Engine, this device, flash it". The implementation is also much easier since we know the API in advance and can call the functions in the backend more easily but the same cannot be said for the devices.

```
$[firmware-engine] [API command] [device]
```

Listing 3: The semantically more natural structural option.

```
$ [firmware-engine] [device] [API command]
```

Listing 4: The robotic sounding option.

The following sections enumerate the minimal API needed for the framework to achieve its goal. A simple rundown of the thought process as well as a simple terminal example will be shown for each section.

4.2.1 Setting up a Device

A user can not begin to interface with the framework if there are no devices in the first place. Since the Firmware Engine keeps state through files in true Unix fashion, setting up a device can technically proceed by simply adding files. The needed files to add are a folder named after the device's unique identifier containing at least one Python file named `module.py`. The folder should be called `plugin` and be added to the `devices/` directory, which is found in the frameworks root directory e.g.

```
/var/lib/firmware-engine/devices/device0/plugin/module.py
```

Adding devices in this manner works well enough but since adding directories and files is usually done through the terminal anyways and most user interaction will happen through the terminal or by calling the native Python API functions, we will need to define an explicit API call for ease of use. The main point is that while there may be multiple ways to add a device, the framework should only officially support one way, and that is through an API function. This has the benefit of letting users discover this functionality by simply invoking the framework with the help keyword, which would list the whole API.

As a natural consequence we need to introduce a function that adds devices. The minimal information needed to set up a device is its unique identifier and the plugin associated with it. The plugin module functions as a backend for all other API calls and so needs to be defined for each device separately. This leads us to the `add-device` result shown in Listing 5. It takes as its arguments the unique device identifier and the plugin module it should be associated with, this command enables users to actually use the framework with the devices and is the stepping stone for the rest of the API to even work.

```
$ firmware-engine add-device [device_id] [module.py]
```

Listing 5: Usage of the add-device command.

4.2.2 Listing Devices

We now have a starting point for users to interface with the API by adding new devices. Another consideration would be that not all users are always adding devices, but rather will want to interface with already added ones. This calls for a way to list existing devices by their full name. The `list-devices` command can be used to bring up the full list of already set up devices. The full command would look something like Listing 6.

```
$ firmware-engine list-devices
```

Listing 6: The syntax for listing all devices of the Firmware Engine

There is no need for arguments since the point of the API call is to inform of any existing devices and not place the burden of preemptive knowledge on the user.

The expected minimal output is a vertical listing of all existing device names separated by new lines. Additional information could be added but is not the main point of the API call.

4.2.3 Query

Now that a user can inform themselves of existing devices and can create new ones, it is time to introduce the most important API call of this whole arrangement. The query command is the goal we have been leading up to this whole time, since it solves the problem of not knowing what firmware is on what hardware.

The command must be as simple as possible so that a user can easily see firmware information with the least amount of hassle possible. Because of that, only a single obligatory argument is needed to identify which device's firmware information to display. The output is the metadata content as described in subsection 3.1 displayed in the form of a table.

```
$ firmware-engine query [device_id]
```

Listing 7: The query command displays information about live firmware

The special `all` keyword displays the metadata of all set up devices.

```
$ firmware-engine query all
```

Listing 8: Using the all keyword with the query command invokes it on all devices.

While we did decide on the side-car approach in subsection 3.3, it is still possible to embed the metadata within the firmware and have the Engine read it out. For that to work, the plugin module will need a function definition for `query` that outputs a Python dictionary in a similar schema to the one mentioned in subsection 3.1.

4.2.4 Flash

While a user can now query to their heart's content, we are yet to actually use that functionality. The reason is that we need a way to save the metadata somewhere. There is an argument to be made that the query command would be enough if the metadata was embedded and if it was possible to read active firmware from a devices storage (usually flash storage).

There are two big ifs in that argument and while embedding metadata is theoretically possible, it would over complicate things drastically compared to side-car files. In addition, the reading of active firmware is not always possible without too many unnecessary work arounds, like with FPGAs. The biggest problem though, is the fact that there is no unified and general way to 1.) embed information into firmware and 2.) read active firmware for all sorts of device types.

In that case we need the Engine to also flash firmware bundled with metadata. This is so that the Engine can know where metadata is stored by default when flashing devices. That is where the flash command comes into play.

It takes two arguments, the obligatory device name and the optional file name. The device name can be a shorthand designation or the full one, as long as it can unambiguously identify a device to flash. This argument is strictly needed because there is now way for the Engine to know which device the user wants to flash otherwise. The file name is optional and if omitted the Engine calls the `get_stable_release` function defined in the device's plugin module. The stable release functions as a sort of reset button that will revert the devices firmware version to one that is trusted to work by the original plugin module authors by default. The problem it is trying to solve is that of minimizing machine downtime in case a device is flashed with faulty firmware, which could snowball into lost revenue again if the firmware is not reverted to one that is trusted to work.

The syntax for specifying a file can be seen in Listing 9. Listing 10 and Listing 11 both use the latest stable release available with the latter flashing all devices.

```
$ firmware-engine flash [device_id] [file]
```

Listing 9: Usage of the flash command in case a file is available.

```
$ firmware-engine flash [device_id]
```

Listing 10: Not specifying a file gets the latest stable release.

The file to flash onto a device must be in the form of an archive with the specific file format accepted by the plugin module as well as a metadata file in the JSON format.

The Engine automatically unarchives the file, calls the flash implementation for a specified device with the needed data which depends on the file formats accepted by a device and

```
$ firmware-engine flash all
```

Listing 11: All devices get flashed with their stable release with the `all` keyword.

if the flashing was successful saves the metadata in the devices folder as `metadata.json`.

The word "flash" was used as opposed to "update" since not every device would be updated by the Engine e.g. FPGAs are programmed and other devices may be downgraded or sidegraded for debugging or research. While this is more of a semantic distinction, clarity in what a command does at first glance is just as important as the actual function it provides.

4.2.5 Rollback

The rollback command is a late addition to the API but carries the crucial role of resetting a devices firmware state to the last successfully flashed one. This functionality was borne from a need to have a simple go back feature. In case of a failed flash the Engine will ask the user if they want to reset it to the previous version.

This way we can minimize the downtime of a device granted a working firmware is present in the history. In case of a failed flash, the Engine will ask the user if they would like to roll back to the previous firmware state. The act of rolling back a release can also be invoked through the API like shown in Listing 12 and Listing 13 when rolling back all devices.

```
$ firmware-engine rollback [device_id]
```

Listing 12: Rolling back a specific device

```
$ firmware-engine rollback all
```

Listing 13: Rolling back all devices

4.3 Plugins

The plugin system was mentioned numerous times already and is the backend of the Firmware Engine, where all devices are implemented. This is realized through the power of scripting in the same vein as most other extension systems work on other software. Examples include browsers like Google Chrome, code editors like VSCode, reverse engineering frameworks like Ghidra and games like World of Warcraft. What all of these pieces of software have in common is the ability to give its knowledgeable users the ability to add new features or extend the software in other meaningful ways like Quality of Life improvements.

The languages in many extension systems need a runtime. JavaScript, TypeScript, Java and Lua are what the previously mentioned softwares use and they would be just as capable to function as the principle scripting language of the FirmwareEngine.

The reason we opted for Python in the end was for the sake of developmental ease since the Enzian BMC already has access to the Python interpreter. Adding the Java Virtual Machine or a Lua interpreter would not necessarily be out of the question, but it would needlessly take more time and add more dependencies in the context of Enzian. Additionally, Python can easily interface shared libraries through the `ctypes` module [22], which gives access to a variety of functionalities used to create wrappers for non-Python libraries. This in essence gives us more freedom to implement plugins in other languages that can be compiled into shared libraries and only needs us to use Python as a sort of glue.

The extension systems used in the previously mentioned softwares were one of the main inspirations but the original idea for the plugin system was also partly a consequence of wanting to reuse scripts already written for the Enzian platform, and otherwise inspired by how game engines such as Unity, Godot and Unreal Engine incorporate scripts written in a variety of languages by game developers to define their games logic. Game engines are not useful if developers do not add their logic and assets through scripts, and this mirrors how the Firmware Engine is not of any use without plugins modules. This is also why it is called Firmware ”Engine”, since it acts as a vehicle to satisfy their specific firmware management needs.

The problem we are trying to solve with the plugin system is the lack of scalability. In other words, while we can abstract away the API into general commands that can work with ”devices”, we still need a way to define what e.g. the flash command should do with a specific device. The path of least resistance would be hardcoding the CPU, FPGA and BMC scripts to work with the frontend part of the Firmware Engine but that is rarely a clean solution and dooms the system to be used for only one type of machine.

In essence, we need the Firmware Engine to be able to find a module corresponding to a specified device, and then find the correct function definition associated with the frontend command invoked. When the function is found all the Engine has to do at this point is call it with the optional file argument.

Taking a step back, we can see that a plugin is a natural consequence of having uniquely identifiable devices and a frontend API that defines the commands permissible on the aforementioned devices. This comes from the fact that there must be a main Python module associated with a devices that will be called by the API. This module would be called a plugin. It is a system that is needed because of constraints and thus has constraints of its own.

The first constraint for a device's plugin module must be named `module.py` and must be found in the devices plugin folder e.g.:

```
/var/lib/firmware-engine/devices/device0/plugin/module.py
```

Why do we need a folder if it will just end up having a single Python file? Simply put, it may not only be a single file but multiple. The only compulsory file is of course the one named module.py which must contain the definitions of the API functions for the specific device but developers may opt to keep all non-API specific functions in separate files.

As indicated before the API functions must be defined, but not all of them. The only API call that demands the existence of an implementation in a plugin module is `flash`. The rest of the API calls on specific devices, namely all but `add-device` and `list-devices` have a default implementation: `rollback` is dependent on `flash` and needs no custom implementations and `query` works out of the box as long as a tarball containing a metadata file is provided when flashing since the Engine saves the metadata and displays it to the user upon querying.

In other words, the plugin module must contain at least a function definition called flash that takes as arguments a path to a firmware image specific to the device. The type of the argument is a string and the return type can be nothing. Originally, the return type used to be a boolean to indicate failure but that idea was replaced by the use of a custom `EngineError` that must be called by the plugin developer within a function. The Engine will intercept this type of error and pivot towards rolling back a version in the case of `flash`. This of course puts additional burden on the plugin developer, which will have to take all failure points of their code into account.

The reason for changing from boolean return values to exception based error handling is that raising errors does not use up the return value. In this case a function can both return a value on success and communicate a failure to the Engine. This way the query function could for example tell the Engine that something went wrong while still being able to output a dictionary of metadata information in case of success.

The query API call can optionally be defined in the plugin module but that should only be the case if a custom way of reading metadata is needed. It does open up the method of embedding metadata in a firmware file if a method of reading active firmware is available, but in most other cases it is much easier to bundle the firmware with a metadata file into a tarball and let the Engine take care of the remaining work.

Notice that the file path argument in the `flash` frontend command was deemed optional. It is simply because firmware types and versions come in all sorts of aspects. That is, firmware can be stable or unstable, built for release or for debugging or even as an experimental version that includes breaking changes. To put it simply, by default the newest stable version of firmware should be flashed to, since that is the kind of firmware guaranteed to work. The guarantee comes from the plugin developers themselves by enabling them in specifying where the most recent stable release is located through a plugin function. The function in question is aptly called `get_stable_release`, takes no arguments and returns the location of the firmware in question. It can of course raise an `EngineError`. The function body can be as simple as returning a hardcoded string but the power of having it as a plugin function instead of a configuration option when first adding a device is that more complex ways of sourcing firmware can be used. For example, firmware can be built from cloned repositories or downloaded from a TFTP server or even some other server available on the internet. The possibilities are numerous and gives developers enough choices that any type of firmware sourcing should be possible.

We have now successfully put up the constrains needed for the plugin system to work with the API frontend that users would interact with.

4.4 Sub-Engines

The Firmware Engine does not only exist on a single BMC instance in a vacuum. In fact, the Firmware Engine can be leveraged in a distributed manner by specifying Sub-Engines.

The idea behind it is that we want to be able to flash a whole cluster consisting of multiple machines each with their own devices. This is done with the help of Sub-Engines, which are instances of Firmware Engines running on other machines that can be connected to via `ssh` [23].

For this to work the main Firmware Engine must share its public RSA key with all machines containing Sub-Engines, since password prompts inhibit the use of the functionality.

As the name suggests, a Sub-Engine is a full fledged Firmware Engine with all the same API functions being callable. A Sub-Engine is identified in the same manner as a device alphanumerically and is complemented by the `add-sub-engine` and `list-sub-engines` commands that function very similarly to `add-device` and `list-devices`.

Adding a Sub-Engine To add a machine as a Sub-Engine we need its username and hostname for the `ssh` login and the location of the Firmware Engine Python source code. This amounts to the call signature presented in Listing 14.

```
$ firmware-engine add-sub-engine\
  [sub_engine_id] [username] [hostname] [root]
```

Listing 14: Adding a Sub-Engine needs its ID, username, hostname and its root.

Listing Sub-Engines As with devices, there is a corresponding command to list all Sub-Engines aptly named `list-sub-engines` that uses the format as in Listing 15.

```
$ firmware-engine list-sub-engines
```

Listing 15: Listing all available Sub-Engines.

Using a Sub-Engine In order to actually leverage the Sub-Engine API we need a command that lets a user defer API calls to Sub-Engines in the first place. That is where we introduce the `sub-engine` command (see Listing 16). It takes as its arguments the ID of a Sub-Engine and the API call to defer to it.

```
$ firmware-engine sub-engine [sub_engine_id]\
  [flash/query/rollback/add-device/list-devices]
```

Listing 16: Usage of the Sub-Engine API

In place of the Sub-Engine ID we can use the reserved `all` keyword. The keyword notifies the Engine that every single Sub-Engine should have the subsequent API call invoked. Since the set up of a main Firmware Engine with its Sub-Engines resembles a distributed network of machines, we can leverage that fact by deferring all API calls invoked with the `all` keyword to the Sub-Engines asynchronously. This means that the main Firmware Engine does not block while waiting for a Sub-Engine to complete its API call and can continue deferring to other Sub-Engines. The main Engine only starts blocking its execution when all Sub-Engines are busy and waits for the corresponding outputs.

It could be argued that there is no need for Sub-Engines since all the API calls could be invoked only from the main Firmware Engine without any Sub-Engines. While that is true, it would be bad practice since in the case of the Enzian ecosystem the gateway server can only communicate with a CPU or FPGA through a BMC. This means that regardless of the existence of Sub-Engines, the gateway will have to know how a BMC flashes a CPU, which also means that a device now has to worry about two different hardware components instead of the just the one it is trying to abstract.

5 Implementation and Discussion

The following section will introduce and discuss the Firmware Engines implementation details as well as show an example of a plugin module for the Enzian system.

5.1 Implementation Details

There were a few details that were not important enough to include as true design decisions. The goal of this section is to illustrate that these decisions are design agnostic because they

Programming Language As mentioned in section 4, the Firmware Engine was built with Python because of the languages extensive standard library, which includes JSON parsing, simple multiprocessing pools, and easy access to system calls. There is no doubt that Python is not the only language that has access to these functionalities. In fact, if time permitted, a full rewrite in a systems language like C or Rust would actually lessen the overhead associated with the Python interpreter. This could not be realized because of time issues. Python was initially just used to prototype new solutions but the source code grew large enough to keep it. A rewrite in C would take an order of magnitude more time than could be estimated. The same is true for the plugin language.

Metadata Format We decided on the JSON format mostly because of our choice of programming language. In the end one of the formats had to be chosen and since YAML is not supported in Pythons standard library, the decision was made a bit simpler.

CLI Implementation We use Pythons `argparse` module to give the functionality of a Command Line Interface (CLI). All outward facing API calls have their own docstring that are parsed and displayed in the help message (–help flag). Additional API calls can be added easily since most of the `argparse` related code automatically adds all ”public” class methods as sub-commands. Since Python does not have a private keyword, we will have to do with the usual style that private functions are prepended with and underscore e.g. `_foo()`.

The default folder to save files in right now is `./FWE`, which automatically gets created in the current working directory. Ideally, the root directory of the Firmware Engine should be in `/var/lib/firmware-engine` but that is only useful if we implement it as a system daemon that can run in the background.

5.2 Enzian Plugin Module

The following section illustrates the concrete implementation of the plugin modules associated with an Enzian machine. This includes modules for the BMC, CPU and FPGA.

BMC There is no concrete plugin module implemented for the BMC since having it update itself is quite a task in and of itself. Since all Enzian BMCs are running OpenBMC, there is an alternative approach to consider. That is to create a plugin module that leverages the Redfish protocol which is implemented in OpenBMC. This can be done from the BMC itself, as long as a Redfish Session is opened as usual.

```python
import http.client
import ssl
def get_bmc_token(bmc_hostname: str) -> str:
    conn = http.client.HTTPSConnection(
        bmc_hostname,
        context=ssl._create_unverified_context()
    )
    headers = {
        'Content-Type': 'application/x-www-form-urlencoded',
    }
    conn.request(
        'POST',
        '/redfish/v1/SessionService/Sessions',
        '{"UserName":"root", "Password":"OpenBmc"}',
        headers
    )
    return conn.getresponse().getheader('X-Auth-Token')
```

Listing 17: Get the Redfish authorization token via Python.

The code in Listing 17 gives us an authorization token `X-Auth-Token` that can be used for the rest of the API available through Redfish. After getting the token we can update the BMCs firmware with the flash implementation presented in Listing 18.

The OpenBMC documentation does mention how to update from a BMC without Redfish[1] but for some reason when trying to `scp` a firmware image to `/tmp/images` it disappears. Neither options have been tested and as such may not work from the BMC without extensive modifications. There is a chance that the POST request being made does not work with the Enzian BMC out of the box because of custom configurations, which means that we would have to reboot the BMC into RAM and write to the specific flash storages before rebooting again. Additionally For the Redfish API to work we need the OpenBMC image to be in the correct tarball format, which would need additional work that has not been done yet in the case of the automatic Enzian BMC image that gets generated.

[1]`https://github.com/openbmc/docs/blob/master/architecture/code-update/code-update.md`

```python
import http.client
import socket
import ssl
def flash(bmc_firmware: str) -> bool:
    hostname = socket.gethostname()
    conn = http.client.HTTPSConnection(
        hostname,
        context=ssl._create_unverified_context()
    )
    headers = {
        'X-Auth-Token': get_bmc_token(hostname),
        'Content-Type': 'application/octet-stream',
    }
    conn.request(
        'POST',
        '/redfish/v1/UpdateService',
        open(bmc_firmware, 'rb'),
        headers
    )
    response = conn.getresponse()
    res_read = response.read().decode('utf-8')
    print(res_read)
    return response.reason == "OK"
```

Listing 18: A potential flash implementation for the BMC.

CPU The ThunderX-1 CPUs firmware is located in an SPI flash that is multiplexed between it and the BMC. As such flashing and reading the flash storage is not possible if the CPU is powered up. This means we have to use the Enzian Bring Up Console [24] on the BMC and read the GPIO port corresponding to the SPI flash. Additionally, since the `mtd` with an associated number could change any time, we need to check `/proc/mtd` for the device associated with the name ”enzian-bdk”. After all of that information was sourced, we can just call the `flashcp` command with a file and the MTD corresponding to the SPI flash.

It is possible to update a host, which in this case is the Enzian CPU, through the Redfish API without the Firmware Engine.[2] However, this is just a wrapper that would call `flashcp` anyways. There is a point to be made that the whole firmware managing functionality could be done through Redfish API, but it would not support FPGA programming

[2] https://github.com/openbmc/docs/blob/master/architecture/code-update/host-code-update.md

easily, which would split our firmware API between what Redfish can do and what it can not do. A unified API makes more sense when taking user experience in account.

FPGA An Enzian possesses a Xilinx Virtex Ultrascale+ FPGA XCVU9P-FLGB2104-E, which can be programmed from the BMC with the help of the Linux FPGA Manager[3]. It works by first powering on the FPGA through the Enzian Bringup Console and copying the FPGA bitstream to `/lib/firmware`.
We then have to write the bitstreams file name to the correct manager

```
/sys/class/fpga_manager/fpga[0|1]/firmware
```

We can figure out the correct FPGA by checking if the name file

```
/sys/class/fpga_manager/fpga[0|1]/name}
```

contains `Xilinx Slave Serial FPGA Manager`. The CPU must be powered on too if the ECI link is to be established. Brief testing suggests that the CPU must be put into reset before and while programming and can be released afterwards.

[3]`https://www.kernel.org/doc/html/v5.0/driver-api/fpga/fpga-mgr.html`

6 Experimental Evaluation

Now that we have gone through the implementation details of the Engine, we are ready to evaluate it.

There are various ways to evaluate a piece of software but since the Firmware Engine was designed with users in mind an important aspect to consider is that of usability. Subsequently, we will be qualitatively evaluating our design with the help of heurstics and then follow up with a quantitative evaluation of the design and implementation of the Firmware Engine.

6.1 Heuristic Usability Evaluation

A relatively simple approach to usability testing for a UI are the 10 usability heuristics by Jakob Nielsen [25]. Even if a command line interface (CLI) is not often the subject of usability methods, applying and comparing the heuristics with what we have designed so far should in theory give us some good feedback on the usability of the Firmware Engine.

6.1.1 Methodology

Before beginning the qualitative evaluation we must first introduce the metrics we will be appraising our design and implementation on. The following paragraphs will be briefly explaining each of the 10 heuristics in the context of our firmware manager.

Visibility of System Status Always inform users of the current state of the system. The idea is to give the user an idea, what the outcome of their previous action was and to inform them of their next options.

Match Between the System and the Real World Use expected language and not internal jargon that a user may not know about. Also seek to follow real-world conventions such that information may appear in a natural and logical order.

User Control and Freedom Give the user a clearly marked exit path in case of an unwanted action. This is so that a user retains the feeling of control over the system in the event of a mistake, which can in turn lower frustration.

Consistency and Standards Do not make users guess if the meaning of words, situations and actions are the same as in other software or not.

Error Prevention Either prevent errors from happening or check for them and ask users for confirmation. Nielsen differentiates between two error-types: slips and mistakes. While slips are unconscious, mistakes are conscious mistakes that occur because of a mismatch between a user's mental model and the design. Avoid slips with constraints and good default values and prevent mistakes by removing the burden on a users memory and warning them.

Recognition Rather than Recall Users should not have to remember everything about the user interface by heart, but should rather be able to "rediscover" it every time by making all elements, actions and options visible through icons or marking them well. In the case of a CLI, this means that everything a user can do should somehow be discoverable.

Flexibility and Efficiency of Use Shortcuts for expert users are an important feature to possess. Not only should there exist shortcuts, but they should also be hidden from novice users such that they do not get overwhelmed by all the available methods to interact with the interface.

Aesthetic and Minimalist Design Prevent irrelevant or rarely needed information from cluttering interfaces. Focus on content and features that support the primary goals of the interface without distracting users.

Help Users Recognize, Diagnose and Recover from Errors In case of an unexpected error, users should be able to recognize the situation through error messages in plain language. The interface should also indicate what went wrong and suggest a solution.

Help and Documentation In the best case, a system would not need any additional explanation. However, keeping an easy to access documentation with concrete examples can go a long way to ensure that users can fulfill their goals.

6.1.2 Evaluation

Now that we have taken a look at all 10 usability heuristics we can now start evaluating our design and implementation on them. Each paragraph mentions what the Firmware Engine does to fulfill the heuristic or why it fails.

Visibility of System Status In the case of the CLI version of the Firmware Engine, the user is always notified through the terminal if the state of the program changed. In subsubsection 6.2.1 of the quantitative evaluation we will see that the Engine has an overhead of 1.2 seconds, which is not excruciatingly slow but could definitely be improved, since users will definitely notice the sluggishness, especially for quick commands e.g. `query`, `list-devices` etc. Every time a command is called, the CLI outputs what it is currently doing and also notifies about the success or failure of a command.

Match Between the System and the Real World The Firmware Engine does not always adhere to this heuristic. While the definition of flash, query and rollback should be understandable for a user with knowledge in computer systems, the idea of a device was never explicitly explained in the program itself but solely in this thesis. This would have to be amended to make users understand what a device even is, an abstraction of flashable hardware components.

User Control and Freedom Considering that we are operating with a CLI, unless something genuinely broke, sending an interrupt signal (Ctrl+C) should always work. In case of a failed flash or if the user wants to recover the previously flashed firmware, the `rollback` command can be used to revert to the last successfully flashed image.

Consistency and Standards The Firmware Engine consistently utilizes the same naming conventions, the problem arises with the non-standard wording of some things. Devices and Sub-Engines are never explained within the context of the software but instead in the thesis. This could be improved by writing more descriptive documentation that clarifies these concepts.

Error Prevention There are a few examples of the Firmware Engine preventing simple user errors from happening. One of them is the the target devices property within a metadata file, which checks the validity of using a specific firmware image with a specific device. Comparing the target device with the one chosen by the user can and will minimize human error, since the Engine will halt and warn the user of the mismatch.

Recognition Rather than Recall Devices are intrinsically rediscoverable because of the existence of the `list-devices` command, the same goes for Sub-Engines with `list-sub-engines`. The main `firmware-engine` command possesses a help flag, which enumerates all the API calls that can be done and briefly explains each one. The API calls themselves also possess their own help flags that not only explain the purpose of each command but also enumerate all arguments and their expected values. This way users do not have to learn anything about the Engine by heart and can instead rediscover all existing devices and API calls.

Flexibility and Efficiency of Use The Firmware Engine has two shortcuts, both to do with devices and sub-engines and their unique identifiers. The first shortcut is the option to shorten device names as long as they are still uniquely identifiable (see subsection 4.1). The second shortcut is the addition of the special `all` keyword. It functions similarly to a boolean flag while still working within both the device and sub-engine ecosystem. The `all` keyword simply tells the Engine to use every existing device or sub-engine to fulfil the specified command.

The structure of all the other commands are as brief as possible with the only exception being the Sub-Engine API, which can theoretically become infinitely long if there exists a configuration with two Firmware Engines calling each other as Sub-Engines perpetually.

Aesthetic and Minimalist Design The CLI only displays the most important information without cluttering the whole terminal window. The only marginally cluttered part of the software is the initial help display, which is a restriction of the `argparse` module.

Help Users Recognize, Diagnose and Recover from Errors All errors that the Engine can foresee are taken care of by notifying the user about what happened and what the possible solutions are. This includes: undefined plugin module commands, empty or non-existent metadata files, failed flashes and invalid device names. None of the errors have codes in them and always explain what the problem was and how to fix it.

Help and Documentation This thesis partially takes part in documenting how the Firmware Engine works and how to use it. Otherwise, the rest of the documentation is found on the git repository containing the source code[4].

6.1.3 Conclusion

In conclusion there are some improvements to be made, like writing better documentation for devices and sub-devices that can help users understand the API they are interacting Another major problem to solve is minimizing the overhead of the program by turning it into a system daemon that keeps state in memory.

6.2 Quantitative Evaluation

The following parts will contain experimental evaluations of the Firmware Engine. To begin with we must start with the methodology and follow up with the initial set up of the experiments.

Methodology The experiments have to evaluate something that can be measured quantitatively. We chose to measure the performance of the Firmware Engine by measuring the time of command line calls with the help of the `time` Linux command. Usually more than one measurement sample is taken, this was done with the help of bash's for loop and file write functionality. Any time a performance metric with n amount of samples is displayed in a table, it means that the command structure as in Listing 19 was used unless stated otherwise. The resulting `measurements.txt` file was then parsed by a Python script that automatically calculated the mean and standard distribution of all data entries.

[4]`https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2024-bsc-jelyes/enzian-firmware-metadata-tools`

```
for i in {1..n};\
do { time [command_to_measure]; } 2>> measurements.txt;\
done
```

Listing 19: The default timing methodology used.

Experimental Set Up We use the Enzian cluster as the experimental system since the goal is to closely mimic the environment that the Engine would be used in. This means that we need to set up the CPU and FPGA on an Enzian BMC as well as set up all Enzian BMCs to be Sub-Engines on the `enzian-gateway.ethz.ch` server. The three Enzian machines utilized ended up being: `zuestoll08`, `zuestoll09` and `zuestoll10`. The git repository [5] containing the source code of the Firmware Engine must be cloned onto each BMC used as well onto the gateway server, which we will refer to as the main Engine. The CPU and FPGA must be set up on each BMC as shown in Listing 20 and Listing 21 respectively.

```
$ firmware-engine add-device \
  "Marvell Cavium ThunderX-1 CN8890-NT CPU @ 2 GHz (48 x ARMv8.1 cores)" \
  CPU_module.py
```

Listing 20: Setting up a CPU device.

```
$ firmware-engine add-device "Xilinx CVU9P FPGA" FPGA_module.py
```

Listing 21: Setting up an FPGA device.

Now the Engines on the BMCs are ready for use but we will need to set up the Sub-Engines too. We add all Sub-Engines by inputting the command found in Listing 22 on the main Engine. To bootstrap the ability to query on the Engines, we flashed the stable

```
firmware-engine add-sub-engine\
zuestoll[08|09|10] root zuestoll[08|09|10]-bmc /home/root/
```

Listing 22: Setting up a Sub-Engine.

firmware image onto all CPUs so that we have at least one device with a metadata file to read. There is no need to test other firmware images for the CPU since the implementation of the flash command is a 1:1 copy of what a user would have to manually input.

[5] `https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2024-bsc-jelyes/enzian-firmware-metadata-tools`

6.2.1 Timing Query

The query command is the heart piece of the Engine and as such will be compared to manually figuring out what firmware is running on hardware. As added context, in the case of Enzian, the only way to know the firmware versions of hardware is to conduct a web search on the Systems Group NetOS Wiki and check an HTML table. In this experiment we would like to find out what the quantitative difference between our query API and conducting a search on the website is. We do this by measuring the time it takes for the query command to finish with the help of the Linux time command. The command to measure with a sample size of 10 is listed in Listing 23.

```
$ firmware-engine query CPU
```

Listing 23: The query command to time.

Then we compare the measured time with the one obtained from using Firefox's performance analysis tool on the wiki page. The tool displays how long each part of a web site takes to load and a total time as well, which is what we will be taking advantage of for our measurements. It also does two different measurements, one with a primed cache and one without, but we will not be needing the primed cache since the difference is negligible. We will collect data from the performance analysis tool on the web page with a sample size of 10.

As an added metric we also measure how large the overhead of the Firmware Engine is by timing the command found in Listing 24. When calling the program with the `help` flag, the time measured should be composed of the overhead as well as the time it takes to print the help message, which is comparably negligible.

Test (# samples)	Average Time [s]	Standard Deviation [s]
Help API (10)	1.203	0.0036
Query API (10)	1.197	0.0032
Wiki Page (10)	0.829	0.1557

Table 1: Results of measuring the query API, the help flag and the Wiki page.

Conclusion We can see that the query consistently takes 1.2 seconds to display information while a web page load on average takes almost 30% less time. Additionally we can

```
$ firmware-engine --help
```

Listing 24: A call with the help flag as a type of control group

see that even just calling the Firmware Engine with the help flag takes about the same amount as querying, which points towards the query API not really taking that long and there being an overhead somewhere else. This can be explained by how the Firmware Engine is implemented right now since the majority of those 1.2 seconds is used up by the overhead of parsing the Python source code every single time and reading files to get the device list, sub-engine configuration etc. This could be remedied by having the Engine run as a daemon that keeps its state in memory instead of having to read its files to get the state.

In the case of keeping the state in memory, the query command would presumably be in the order of microseconds, since that is what a file read takes on average.

What this means for users is that the querying from the Firmware Engine is not too far away from being as quick or quicker than conducting a search on a manually maintained Wiki page.

Something that could not be reliably evaluated is the parsing a human would have to do on an HTML table. We can heuristically try to estimate a good time a user would need to find the information for the correct machine. Consider the average visual stimuli response of a human using a mobile device [26], which lies at around 320ms ± 43ms. A human would need around two stimuli worth of time to get to the wanted version number in the table. That would end up being around 640ms. Adding those to the page load time shows that it would take around the same time if even more than just using the Firmware Engine's query API. When we add the fact that manual input of versioning information significantly is less consistent than automatically saving metadata files, we can conclude that the `query` API successfully improves the current firmware management situation of the Enzian ecosystem.

6.2.2 Timing Flash

Timing the flash implementation included in a plugin module makes little sense considering we are in essence just measuring how long the original flashcp command takes to flash the CPU's SPI flash or how long the Linux FPGA Manager takes to program the FPGA.

Something that would be of more use is measuring the pure Firmware Engine's flash API overhead and comparing it to updating the Wiki HTML table manually. We want to find out if the overhead of the flash command is small enough to warrant its use over just editing a Wiki page with the correct firmware versions.

For this to work we will can comment out the part of the flash API where the devices specific flash command is invoked and if the return value is needed replace it with `True`.

This way we can ensure that only the parts that are not actively flashing a device can be measured, which include: Finding the correct device name from a shortened version, saving metadata, saving firmware to the history and cleaning up temporary files. The command to measure will be the one shown in Listing 25. Where the file to flash can be any valid archive containing a metadata file and a different file that will not be used in this

```
firmware-engine flash CPU flash_stub.tar
```

Listing 25: Flash the CPU with an empty flash implementation.

context since nothing is flashed.

When trying to measure the time it takes to manually edit an Wiki article we can measure the page load time similarly to how we did it in subsubsection 6.2.1 but instead in editing mode. The edit page of the same wiki entry can take over 10 seconds to fully load but we are interested in the shortest time until the page is interactable. We do this by looking at the initial time the browser took to load and display the edit page. This is done manually by checking the initial load time before any additional low priority loads occur, and can very well be considered quite inaccurate. To combat this, it is imperative to minimize any bias and take the very first number displayed by Firefox.

Test (# samples)	Average Time [s]	Standard Deviation [s]
Flash Stub (100)	1.209	0.0036
Wiki Edit Page (10)	1.469	0.2131

Table 2: Results of measuring the stubbed flash API and the Wiki page in edit mode.

Conclusion It is evident that the overhead of the flash API call is very similar to the measurements taken for the query and help API too, which strengthens the hypothesis that the overhead of the Firmware Engine's memory-less implementation is quite large.

We can also see that the wiki page takes on average 20% longer to load without factoring in the editing itself. When factoring those in we have to consider the following sequence of events: Loading the wiki page in edit mode, finding the correct entry, typing in the version number and finally submitting the changes.

We can break down the heuristic times of all of the events as shown in Table 3.

Event	Average Time [s]
Page Load Time	1.469
Find Entry	0.640
Edit Entry	1.862
Submit Changes	0.829
Sum	4.800

Table 3: Average time heuristics of web page related events.

The Page Load Time is self explanatory since we just discussed it at the beginning of the paragraph. The time to find the correct entry was also discussed in section 6.2.1

```
$ firmware-engine sub-engine all 'flash CPU bootfs-stable.tar'
```

Listing 26: Use the Sub-Engine API to flash all CPUs at once.

and ended up being 640ms. Editing an entry is slighlty harder to quantify but we can estimate it with the following data: Since the average typing speed is around 51.56 words per minute with a word length of 5 characters that gives us 4.297 characters per second [27]. The version numbers seen on the Wiki table are always 8 characters long, which gives us 1.862 seconds to type in a new version number. Submitting changes takes around the same time as loading the page in read mode, which was on average 0.829 seconds (see section 6.2.1). Summing everything up, we can get a lower bound for the average user to change the firmware version, which amounts to around 4.8 seconds.

That is quite a bit longer than the 1.2 second overhead of the flash API, especially considering that the Firmware Engine saves even more information automatically and that the Wiki page has to be edited manually. When including human errors like forgetting to update information on the page, we can conclude that using the flash command is the better option of the two.

6.2.3 Reflashing an Enzian cluster

We do not just want to manage the firmware of a single Enzian machine, but a whole cluster consisting of 14 machines. This is why we must also simulate a situation where a whole cluster suffers from firmware containing a security vulnerability. In this case an administrator would have to disable machine access, log into the BMC console of each Enzian and call the flash command on each Enzian machine and finally change the information in the Wiki HTML table.

We can calculate the time it takes to sequentially go through all 14 machines in this manner and update the version information on the Wiki page.

Updating the table would take the sum of the initial edit page load time, finding and editing the entry 14 times and then submitting the changes. This amounts to 37.326 seconds just to update the information of 14 machines.

To get the time of the Firmware Engine we would need to use the sub-engine API with the all keyword. This will call the flash command in different processes that do not block each other. We measure the time the Firmware Engine needs to flash all the CPUs associated with their sub-engines with one, two and three sub-engines respectively. This is so we can see if there is a considerable overhead associated with the multiprocessing nature of the sub-engine API.

To measure the Engine we used the function presented in Listing 26.

Test (10 samples)	Average Time [s]	Standard Deviation [s]
Single CPU Manual	63.480	0.371
One Sub-Engines	62.116	0.230
Two Sub-Engines	63.280	0.506
Three Sub-Eng	62.877	0.202

Table 4: Results of measuring the sub-engine API with 1-3 machines as well as flashing a single machine manually.

Conclusion It is evident that leveraging the distributed configuration of the Enzian cluster brings us quite a bit of a speed up when flashing a large number of machines with little additional overhead incurred by the multiprocessing nature. This makes sense since the act of flashing is done by the Sub-Engines and not the main Firmware Engine, which defers its work to different subprocesses, which call the Sub-Engines.

Seeing that a single manual CPU flash takes as long as flashing three CPUs at once with the Engine, we get a speed-up of N where N is the number of machines in a cluster. In the case of the Enzian cluster that is a speedup of up to 14.

Referring back to the situation of suffering from downtime until the new firmware has been flashed on all machines. When using the Firmware Engine the estimated downtime is around x, where x is the time needed to flash a single CPU. Compare that with the manual flashing, which takes $N * x$. In concrete numbers, the downtime is around 1 minute with the Firmware Engine compared to 15 minutes when doing it manually (including updating information).

While those numbers do not sound like much when we factor in the cost of downtime at \$300'000 per hour [2] that amount to \$5'000 with the Engine and \$75'000 when done manually. That is an order of magnitude less money lost, and would be even larger for a large enterprise with thousands of machines running in clusters.

7 Conclusion and Future Work

In conclusion, we set out to minimize the downtime posed by the problem of taking vulnerable machines in a cluster offline and reflashing them.

We did this by first defining the content and format of firmware metadata that is bundled in a sidecar file with the original image. We then introduced the design of a system with a frontend (API) and backend (Plugins) that can abstract away hardware specific firmware-related implementations and leverages the distributed nature of machines in the Enzian cluster with the Sub-Engines, which can in turn fulfill their API calls independently from each other. To match the design we presented a concrete plugin implementation for the ThunderX-1 CPU and Xilinx FPGA contained within the Enzian and a theoretical way to implement a plugin for the BMC.

As the Firmware Engine is a user-centered piece of software we first evaluated it qualitatively with the help of usability heuristics and then showed that it can save quite a lot of time when having to update a large amount of devices, which in turn saves a lot of money in an enterprise setting.

There were things that unfortunately did not could not be incorporated into the thesis but nonetheless the potential future work could prove to be valuable.

Security There were no security precautions taken, as it could be an interesting project in and of itself. As it stands now, malicious entities could theoretically spoof metadata information or flash insecure firmware images if they have access to the gateway server. Thankfully the Enzian systems are secured by a whitelist, since an account and authorization by the Systems Group is needed to connect. This lessens the worries posed by the lack of security measures within the Firmware Engine.

SBOM Incorporating a true SBOM format for the metadata file could prove very useful since it would include even more information. Considering that all dependencies and third party components of the associated software are enumerated in an SBOM file, it could also be very useful for security reasons, since vulnerabilities in firmware because of third party dependencies could be identified very quickly.

User Study Conducting a meaningful user study was unfortunately outside of the scope of this project. More details are needed like the initial set up and a large sample size of users but if time permitted the way we would have roughly laid out the study was the following:

1. Set up two new devices (Enzian CPU and FPGA).

2. Figure out how many devices there are.

3. Check the version of the CPU firmware.

4. Flash the current stable firmware onto the CPU.

5. Rollback the CPU.

These tasks would have to be completed by each user and a conclusion of the performance of both the Firmware Engine and user could be statistically evaluated.

Device Variety It would be quite interesting to see how the Firmware Engine does in a non-Enzian setting with a more diverse set of hardware and how difficult the implementation of plugins would end up being. It could additionally be possible to estimate the cost of developing plugin implementations [28].

In the end the project showed that designing a firmware manager for a heterogeneous system is indeed possible and brings with it results that lower both cost and frustration compared to manual firmware management.

References

[1] 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.

[2] Information Technology Intelligence Consulting Corp. (ITIC). *ITIC 2022 Global Server Hardware, Server OS Security Report.* URL: https://www.ibm.com/downloads/cas/BGARGJRZ. (accessed: 30.07.2024).

[3] Intel Corporation. *Intel® Server Products UEFI Firmware Advisory.* URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-01080.html. (accessed: 09.07.2024).

[4] Richard Hughes. *fwupd: A system daemon to allow session software to update firmware.* URL: https://fwupd.org/. (accessed: 09.07.2024).

[5] Dell Technologies Inc. *OpenManage Enterprise.* URL: https://www.dell.com/en-us/lp/dt/open-manage-enterprise. (accessed: 09.07.2024).

[6] Intel Corporation. *Intel® Server Firmware Update Utility: User Guide.* URL: https://downloadmirror.intel.com/777361/Intel_Server_Firmware_Update_Utility_User_Guide_r1_6_3.pdf. (accessed: 09.07.2024).

[7] The Linux Foundation®. *System Package Data Exchange (SPDX®).* URL: https://spdx.dev/. (accessed: 15.08.2024).

[8] OWASP Foundation. *OWASP CycloneDX.* URL: https://cyclonedx.org/. (accessed: 15.08.2024).

[9] *Redfish Specification.* DSP0266. Version 1.19.0. DMTF. Aug. 2023. URL: https://www.dmtf.org/sites/default/files/standards/documents/DSP0266_1.19.0.pdf.

[10] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000.

[11] The Linux Foundation®. *OpenBMC: Defining a Standard Baseboard Management Controller Firmware Stack.* URL: https://www.openbmc.org/. (accessed: 16.08.2024).

[12] Tom Preston-Werner. *Semantic Versioning 2.0.0.* URL: https://semver.org/. (accessed: 19.07.2024).

[13] Scott Chacon and Ben Straub. "Pro Git". In: 2nd ed. Apress, 2014. Chap. 3.1 Git Branching - Branches in a Nutshell.

[14] George C. Necula and Peter Lee. "Safe kernel extensions without run-time checking". In: *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*. OSDI '96. Seattle, Washington, USA: Association for Computing Machinery, 1996, pp. 229–243. ISBN: 1880446820. DOI: `10.1145/238721.238781`. URL: `https://doi.org/10.1145/238721.238781`.

[15] Guido van Rossum and the Python development team. *The Python Language Reference*. Python version 3.11.2. Python Software Foundation. 2023.

[16] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. URL: `https://www.w3.org/TR/2008/REC-xml-20081126/`. (accessed: 18.07.2024).

[17] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180. Oct. 2005. DOI: `10.17487/RFC4180`. URL: `https://www.rfc-editor.org/info/rfc4180`.

[18] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Dec. 2017. DOI: `10.17487/RFC8259`. URL: `https://www.rfc-editor.org/info/rfc8259`.

[19] YAML Language Development Team. *YAML Ain't Markup Language (YAML™) version 1.2*. URL: `https://yaml.org/spec/1.2.2/`. (accessed: 18.07.2024).

[20] Kyle Wilkinson. *Bitstream Identification with USR_ACCESS using the Vivado Design Suite*. URL: `https://docs.amd.com/v/u/en-US/xapp1232-bitstream-id-with-usr_access`. (accessed: 19.07.2024).

[21] F4PGA. *Project X-Ray 0.0-3807-g72e6371b documentation: Bitstream format*. URL: `https://f4pga.readthedocs.io/projects/prjxray/en/latest/architecture/bitstream_format.html#synchronization-word`. (accessed: 19.07.2024).

[22] Guido Van Rossum. "The Python Library Reference, release 3.6.0". In: Python Software Foundation, 2017. Chap. 16.16.

[23] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. Jan. 2006. DOI: `10.17487/RFC4251`. URL: `https://www.rfc-editor.org/info/rfc4251`.

[24] Enzian Project Group. *Enzian: Documentation & Resources*. URL: `https://enzian.systems/documentation/`. (accessed: 15.08.2024).

[25] Jakob Nielsen. "Enhancing the explanatory power of usability heuristics". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '94. Boston, Massachusetts, USA: Association for Computing Machinery, 1994, pp. 152–158. ISBN: 0897916506. DOI: `10.1145/191666.191729`. URL: `https://doi.org/10.1145/191666.191729`.

[26] Kyle T. Yoshida et al. *Exploring Human Response Times to Combinations of Audio, Haptic, and Visual Stimuli from a Mobile Device*. 2023. arXiv: 2305.17180 [cs.HC]. URL: https://arxiv.org/abs/2305.17180.

[27] Clare-Marie Karat et al. "Patterns of entry and correction in large vocabulary continuous speech recognition systems". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '99. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1999, pp. 568–575. ISBN: 0201485591. DOI: 10.1145/302979.303160. URL: https://doi.org/10.1145/302979.303160.

[28] Hareton Leung and Zhang Fan. "Software Cost Estimation". In: *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Pub Co Inc, 2002, pp. 307–324. DOI: 10.1142/9789812389701_0014. eprint: https://www.worldscientific.com/doi/pdf/10.1142/9789812389701_0014. URL: https://www.worldscientific.com/doi/abs/10.1142/9789812389701_0014.

Appendices

A Plugin Base

The following is a base version of a plugin module that can be used to start implementing hardware specific functionality.

```python
from FirmwareEngine import EngineError


def flash(filename: str) -> bool:
    # Necessary code to flash the device
    raise EngineError("Flash not implemented.")

def get_stable_release() -> str:
    # Necessary code to get the stable release of a device
    # Can also just be a filename or symbolic link like so:
    #return "~/image.bin"

    # More complicated procedures are also supported,
    # like downloading an archive through RESTful API.
    # Make sure to save the file in root/tmp
    # or at least anywhere on the local disk.

def query() -> dictionary:
    # Necessary code to query device metadata
    # Will only be used if no metadata is found in root/[device_id]
    #raise EngineError

def routine_name(arg1, arg2, arg3) -> Any:
    # This function can have any name and any signature
    # In case of recoverable failure:
    #raise EngineError
```

B Measurements

All time measurements from the quantitative evaluation can be found in the following git repository: `https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2024-bsc-jelyes/enzian-firmware-metadata-tools`.

ETH

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:

Firmware Management for a Heterogeneous Platform

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

Last name(s):	**First name(s):**
Elyes	Julian

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)**
Hagendorn, 17.08.2024	[illegible]

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