Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semester Project

Systems Group, Department of Computer Science, ETH Zurich

Enzian Firmware Resource Interface

by

Pengcheng Xu

Supervised by

Daniel Schwyn
Prof. Dr. Timothy Roscoe

February 2023

D INFK

Abstract

We introduce the design and implementation of the **E**nzian **F**irmware **R**esource **I**nterface (EFRI), an RPC-based protocol for flexible and extensible enumeration and implementation of platform-level firmware services. We first formulate the requirements of a firmware-resource protocol on heterogeneous platforms. We then show the detailed design of EFRI against existing industrial standards. Finally, with the help of a few case studies, we evaluate the performance and ease of implementation of standard firmware services in EFRI.

Enzian Firmware Resource Interface

Pengcheng Xu

February 23, 2023

Contents

1 Introduction

Platform-level management tasks are crucial for real-world computer systems, to keep them running safely and efficiently, and to perform maintenance and introspection tasks. Some of these tasks are performed by the Baseboard Management Controller (BMC) [Frazelle(2020), Ottaviano et al.(2022)], while some others in the platform firmware [Muralidhar et al.(2012), Herdt et al.(2017)]. There exist various industrial standards that regulate how the CPU communicates with the platform firmware and the BMC to perform these management tasks [UEFI Forum Inc.(2022a), DMTF(2022), Arm Limited(2022b)].

CPU/FPGA-based heterogeneous computing systems have been proposed early on [Agron et al.(2006), Andrews et al.(2004)] and are recently gaining more research attention [Belwal et al.(2015), Iorga et al.(2021), Cock et al.(2022)]. On the other hand, the aforementioned industrial standard protocols keep the traditional view of a CPU-centric system and mostly fail to address such heterogeneity. While it is possible to retrofit these protocols to account for such modern heterogeneous systems to some degree, the system designer is often met with poor design choices, rigid protocol requirements, and difficult-to-obtain standards and reference implementations. We need a protocol designed natively for heterogeneous systems with research in mind.

In this article, we introduce the design and implementation of the Enzian Firmware Resource Interface (EFRI). We first introduce in Section 2 several traditional industrial solutions for platform-level firmware protocols and formulate the requirements of such protocols, showing that they are insufficient in one or more of the criteria. We then detail the design of EFRI according to the requirements of such a protocol in Section 3. We discuss a reference implementation of EFRI on an Enzian system in Section 4. Finally, in Section 5.1, we evaluate the ease of use of EFRI by end-users to perform platform-level tasks, as well as system integrators to implement firmware functionality. We also evaluate the overall performance of the protocol implementation in Section 5.2.

2 Background and Motivation

2.1 Enzian

Enzian [Cock et al.(2022)] is a heterogeneous system designed to support hybrid systems research. It consists of a server-grade Arm ThunderX-1 CPU and a large Xilinx XCVU9P FPGA, interconnected by the Enzian Coherent Interconnect (ECI) link. In addition, the system is equipped with ample high-speed interconnect resources, ranging from 100 Gb/s Ethernet to PCIe and NVMe links on both the CPU and FPGA side. A block diagram of Enzian is shown in Figure 1.

An especially interesting aspect of the design of Enzian is the Baseboard Management Controller (BMC) which is open for programming by the system developer. Conventional server systems offer extremely locked-down BMC in-

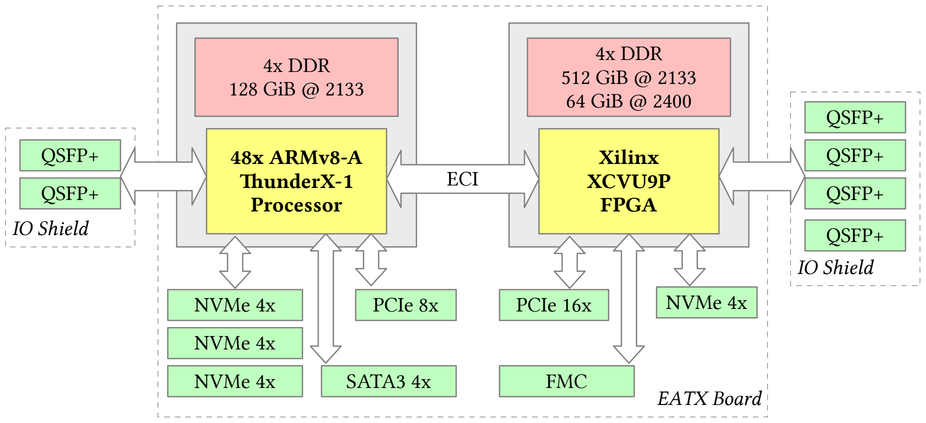


Figure 1: Block diagram of Enzian.

terfaces, usually providing only certain management functionalities. On heterogeneous platforms with an FPGA, the user may choose to implement platform-level services that ideally should be exposed via the BMC in a trusted and reliable fashion. This is made possible with the ability Enzian provided to fully customise the software that runs on the BMC.

2.2 Existing Firmware Protocols

Generally speaking, *firmware* is a class of software that provides low-level control for a device's hardware. It is increasingly common to implement platform-level application-agnostic services, such as power control, telemetry data collection, real-time clock (RTC), etc., in the system firmware. This makes the design of the protocol between the various *clients* that make use of such services, and *actors* that provide such services, crucial for a good platform design. We introduce here several existing industrial-standard firmware protocols and analyse their strengths and weaknesses for heterogeneous platforms.

UEFI/ACPI The Unified Extensible Firmware Interface (UEFI) [UEFI Forum Inc.(2022b)] is a set of specifications written by the UEFI Forum. As a successor to the Basic Input/Output System (BIOS), it provides a rich set of *boot services* to facilitate the process of booting an operating system. In addition, it also provides a set of *runtime services* such as UEFI variables, ACPI (for power control), SMBIOS (for system topology information), and GOP (for graphics output). The runtime services remain available to be invoked by the operating system after the boot process completes. Open-source implementations of the UEFI and ACPI standards include TianoCore EDK II [The TianoCore Community(2021)] and U-Boot [The U-Boot Development Community(2021)].

The Advanced Configuration and Power Interface (ACPI) [UEFI Forum Inc.(2022a)] defines a set of interfaces for the operating systems to perform power management operations as well as device auto-configuration and monitoring, for example, Plug and Play and hot swapping. The interfaces are exposed to the operating system as various *ACPI tables* in a compact byte-code format called the ACPI Machine Language (AML), compiled from the human-readable ACPI Source Language (ASL) [OSDev Wiki(2023)]. Together with UEFI, they are widely adopted by a vast majority of computer systems, from PCs to server systems. They also form the basis of firmware requirements in the Arm Server Base System Architecture (SBSA) [Arm Limited(2022a)].

Despite their wide adoption across the industry, they are not very well suited for heterogeneous research platforms due to the rigidity of standards, not designed for extensions by system designers. The ACPI tables in AML format are also notoriously difficult to examine and experiment with, due to the compact AML format and limitation of 4-byte identifiers. In addition, the UEFI/ACPI standards have mostly matured with the x86 PC market, with no history with emerging architectures such as ARM and RISC-V [Corbet(2014)] and thus less experience overall. These all make the UEFI/ACPI standards unsuitable for a new research platform like Enzian.

IPMI The Intelligent Platform Management Interface (IPMI) [Intel et al.(2013)] specifies a computer subsystem, mainly running on the BMC, that provides platform-level *out-of-band* management (OOB) and monitoring capabilities. Common features include Field Replaceable Unit (FRU) identification, power control, sensor data acquirement, and virtual storage devices for remote OS installation. *In-band* management is often also available over SMBIOS and/or ACPI tables and used to implement platform actions such as power off from the OS [Minyard(2023)]. The standard is adopted by many server manufacturers, including HP [Hewlett Packard Enterprise Development LP(2023)], Dell [Dell Inc.(2023)], and Cisco [Cisco Systems Inc.(2020)]. Open-source BMC distributions such as OpenBMC [The OpenBMC Community(2023)] have IPMI implementations.

The IPMI standards unfortunately did not have a mechanism for discovering available actions. Messages are grouped via the so-called NetFn/LUN/Cmd tuples [The OpenBMC Community(2022a)], with no standard way at runtime to enumerate the possible actions to perform. The namespaces of messages are thus limited to a flat ID-based grouping, which does not scale well to complex systems with different levels of hierarchy. While standard actions are documented in the specification, OEM commands from different vendors have completely different syntaxes and semantics. With some vendors that do not publicly document their OEM commands, system administrators have to resort to sending *raw* commands, often protected by Non-Disclosure Agreements (NDA). This situation makes designing and implementing new platform features that are easy to explore and discover by end-users difficult and ad-hoc.

Arm SCMI The System Control and Management Interface (SCMI) by Arm [Arm Limited(2022b)] is a standard specifying interfaces for power, performance and system management for ARM System-on-Chips (SoC). The standard is designed with the Linux device tree model in mind, allowing the firmware running on the System Control Processor (SCP) to serve as providers for standard resources such as clock (for Dynamic Voltage and Frequency Scaling, DVFS), reset, sensors and performance domains. The SCP could be implemented as part of the ARM *secure world* firmware running at EL3 (the *monitor* exception level), or a standalone BMC on the platform. The client-side implementation of SCMI is open-source and integrated into the Linux kernel, while the server-side implementations are in the open-source Trusted Firmware-A [Arm Limited(2022d)] and SCP firmware [Arm Limited(2022c)] codebase.

As a standard proposed by Arm, SCMI targets SoC platforms with clear role delegations: the Application Processor (AP) and devices are the *agents* and the platform controller is the *platform*. While this makes a lot of sense on conventional processor-centric architectures, on a heterogeneous platform like Enzian it may be desirable to explore different role assignments with the CPU and FPGA. Another less ideal design of using numerical IDs to denote different resources e.g. sensors, voltage domains, etc. This requires maintaining an external mapping of actual system resources according to topology to such IDs, before invoking the commands. This makes the discoverability claim of SCMI weaker and the protocol more difficult to use. Last but not least, SCMI is a relatively young standard (first version 2017) with relatively few adopters across the industry. The open-source code provided is also not very well documented, making it difficult to adopt the standard.

DMTF Redfish The Redfish [DMTF(2022)] standard by DMTF uses RESTful [Richardson and Ruby(2008)] semantics to access a schema-based data model for conducting management operations. The standard is proposed as a successor to the popular IPMI standard, targeting servers as well as most data centre IT equipment. The protocol employs a hierarchical namespace, the *resource tree*, for all resources present in a system, with human-readable Uniform Resource Identifiers (URI) for identifying and interacting with them. The standard is built around the OpenAPI standard [The Linux Foundation(2021)], using the Hypertext Transfer Protocol (HTTP) and the JavaScript Object Notation (JSON) as exchange formats. The standard is widely adopted by server manufacturers, including Dell [Dell Inc.(2022)], HPE [Hewlett Packard Enterprise Development LP(2022)] and Supermicro [Super Micro Computer Inc.(2023)]. The OpenBMC [The OpenBMC Community(2022b)] distribution has an implementation for Redfish.

The Redfish protocol, in terms of resource description, is the closest to what a heterogeneous system like Enzian needs, thanks to the high level of customizability as well as the resource tree denotation. Unfortunately, it is tightly coupled with the HTTP protocol and JSON data models, which can be tough to implement reliably in embedded environments like the secure world firmware or on a micro-controller on the FPGA. The integrated security model also requires encryption as part of the standard. These are non-issues for Redfish since their main goal is to support OOB management in enterprise data centres, but

Protocol	**(A)**	**(B)**	**(C)**	**(D)**	**(E)**	**(F)**
UEFI/ACPI	✗	●	✗	●	✗	●
IPMI	✓	✗	●	✓	✗	●
SCMI	✓	●	✓	✓	✗	✓
Redfish	✓	✓	✗	✗	✓	✓
EFRI	✓	✓	✓	✓	✓	✓

Table 1: Comparison of existing firmware protocols with EFRI with the proposed properties. Legend: ✓: satisfies requirement; ●: partially satisfies requirement; ✗: does not satisfy the requirement.

it would require an unnecessarily large and complicated codebase for in-band management on a heterogeneous research platform like Enzian.

2.3 Motivation

We recognise that traditional industrial solutions are not exactly what we need for a heterogeneous research platform. We summarise here the requirements of a suitable firmware protocol on such platforms.

(A) *Extensive*: the protocol should allow easy extension of firmware functionalities.

(B) *Organised and discoverable*: the various resources should be grouped in namespaces that allow client users to enumerate available resources and actions.

(C) *Modular and composable*: the protocol should not be closely coupled with a specific transport technology to allow for implementation on diverse platforms.

(D) *Simple to implement*: the protocol should be easy to implement in embedded environments with a relatively compact codebase.

(E) *Inspectable*: the protocol should have human-readable messages for easy debugging.

(F) *Usable*: the protocol should assist developers in generating boilerplate code and validity checking.

The requirements correspond to the challenges of firmware protocols on heterogeneous systems closely. A research-focused system moves forward quickly during development and various evaluations, thus necessitates the protocol to be extensive and modular. On the other hand, the protocol should be easily implementable by system designers and usable by end-users with relatively little engineering effort.

We compare the existing protocols with EFRI, the proposed firmware protocol for heterogeneous research systems, in Table 1. UEFI/ACPI is by-design strongly coupled to the x86/PC landscape, thus failing **(A)** and **(C)**; ASL offers severely limited readability and namespacing support, which fails **(E)** and impacts **(B)** **(D)** and **(F)**. The vendor fragmentation of IPMI makes it fail **(B)** and impacts **(F)**; the binary message format being specified in the standard also limits **(C)** and **(E)**. In SCMI, services are not self-describing and require an external manifest, thus limiting **(B)**; it also specifies a fixed binary message format, which fails **(E)**. Redfish is by far the most suitable protocol, but its strong dependencies on HTTP and JSON make it fail **(C)** and **(D)**. We show in Section 3 how EFRI satisfies all these requirements.

3 Protocol Design

In this section, we present the design of EFRI as a firmware protocol. We first introduce basic concepts important to defining the protocol. We then show the key design choices that make EFRI stand out among other firmware protocols in the ways we discussed in Section 2.3.

3.1 Basic Concepts

EFRI is designed to be a protocol of Remote Procedure Call (RPC)-style. The execution of an action is done via a pair of request and response messages between the action *initiator*, which we call the *client*, and the action *receiver*, which we call the *actor*. Hardware devices on the platform may hold both roles at the same time, for example, the CPU may request the BMC for telemetry data, while the BMC may also request the CPU for a graceful shutdown of the OS.

The base protocol is *synchronous* as the main design requirement of an in-band-management focused protocol and low expected latency of actions. With that said, actions that require a very long time to finish can be implemented asynchronously through the *asynchronous event mechanism*, which we introduce in Section 3.4. Between each client and actor, two *logical* channels should exist: the *synchronous* request and response channel, and the *asynchronous* events channel. The logical channels may be multiplexed over physical channels such as a Universal Asynchronous Receiver-Transmitter (UART) link, or be forwarded over other hardware components in the system.

Resources in the system are denoted with *endpoints* that host *actions* available to be invoked by clients. *Namespaces* that group actions for a clean organisation thereof are also denoted with *endpoints* and will be introduced in Section 3.2. Actions are required to be *stateless*, to avoid complicated tracking of client state information in the actor implementation in low-level firmware.

Most actions on resources are shared among different types of resources. For example, a power rail has voltage and current telemetry data, both of which support the `get` semantic to retrieve data. Sharing actions among different resources help reduce boilerplate and maintain a consistent interface. Four common semantics are defined protocol-wide in EFRI:

- `put`: write data to endpoint
- `get`: read data from endpoint
- `subscribe`: subscribe to updates of endpoint data
- `list`: list children actions of namespace endpoint

Specifications of resources on a platform are expressed in a concise format following a defined schema. This allows for the automatic generation of boilerplate code on the actor side, as well as automated checks of message validity. The generator-based approach minimises possible semantic mismatches among different resources in the system. It also makes future extensions to the protocol easy to implement, through changes to the global schema. (*requirements* **(A)** **(F)**)

3.2 Namespacing and Resource Discovery

Actions on resources in a system implementing EFRI are described by the name of the action and the *canonical name* of the resource. We name the canonical name the **F**irmware **R**esource **N**ame (FRN). To perform an action on a specific resource, the client invokes the action with appropriate arguments on the FRN. The FRN is assigned according to where the resource topologically is in the system, i.e. on the device it belongs to. Thus, an FRN can be well-known and thus hard-encoded in client logic; for example, the CPU power control resource has the well-known FRN of `cpu::power`.

On the other hand, an FRN could also be discovered *at runtime* by traversing the root namespace. *Namespaces* organise all actions on FRNs in the system hierarchically. This is a common way of structuring resources in many existing protocols, e.g. RESTful web services [Richardson and Ruby(2008)], Redfish [DMTF(2022)], and Oracle ILOM [Oracle(2014)]. A *view* describes the layout of namespaces a system adopts and is essentially an alternative index of the actions on resources in a system for easier access. Examples of views include the *by-device* view, which coincides with the FRN, and *by-subsystem* view, which groups resources by logical purposes of the resources, such as `power` and `telemetry`. An example FRN and the containing namespace is shown in Figure 2. (*requirement* **(B)**)

3.3 Security and Virtualisation

Security and virtualisation are crucial features for heterogeneous systems in production, taking into account that different components on the system may be programmed by different parties and thus inherently have different trust and authority levels. For the convenience of presentation, we show an example

$$\text{Namespace: } \underbrace{\texttt{::telemetry}}_{\text{subsystem}} : \underbrace{\texttt{platform:rails:BMC\_VCC\_3V3:voltage}}_{\text{topology}}$$

$$\text{Action: } \underbrace{\texttt{get}}_{\text{name}} \ @ \underbrace{\texttt{platform:rails:BMC\_VCC\_3V3:voltage}}_{\text{FRN}}$$

Figure 2: Example of the FRN and containing namespace of the voltage rail `BMC_VCC_3V3`. The *topology* part of the namespace coincides with the FRN of the action due to the way the namespace was generated. Colons (:) are used as separators for different segments in the FRN and namespace names.

system with virtualised tenants running VMs on the CPU and apps on the FPGA, as shown in Figure 3. Requests are forwarded by components lower in the hierarchy to the final destination.

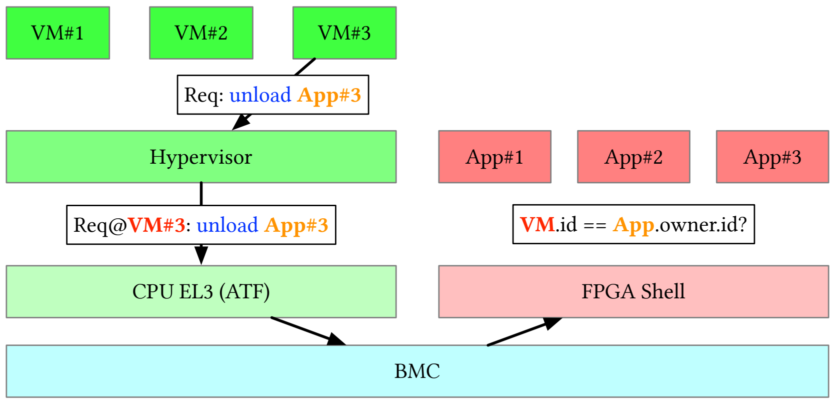


Figure 3: Example of how a request traverses a CPU-FPGA heterogeneous system. A request from the CPU VM to unload an application needs to go through the hypervisor, the CPU EL3 firmware, and the BMC, finally arriving at the FPGA shell. The FPGA shell checks if the initiator is authorised to unload the app by checking if it's the owner of the app.

Defining trust for security and virtualisation is difficult. The compartmentalisation of software components is very important; for example, a compromise of a kernel component, hypervisor, or the Arm Trusted Firmware (ATF), automatically means void of all trust above that level. In contrary, since the BMC and CPU are distinct devices that do not share memory, we can reason about trust of them separately. We define the attack model here to be that we trust all the *forwarding agents* with an external assurance, for example, various attestation schemes [Brickell et al.(2004), Seshadri et al.(2004), Microsoft(2022)]. The system designer can incorporate fail-safe mechanisms in case the attestation mechanism reports a violation.

The permission check process happens along the forwarding chain. First, the first forwarding agent tags the message with an appropriate *initiator* label. The actor then grants or denies access to a specific resource based on the trusted initiator label according to a policy database. Optionally, the forwarders can choose to drop a request according to its local policy database. We leave the exact format of the label and policy unspecified; they are orthogonal to the base protocol and can be specified in a separate work.

The protocol itself is agnostic of being implemented in-band or out-of-band, therefore does not necessitate any transport-level security. In the case of an in-band implementation, in a lot of situations, the *physical security* of the platform is enough: for example, a UART link as PCB traces is secure enough for most servers in managed environments. For clients exposed over the network, for example in a remote management scenario, user authentication and transport-layer encryption can be added independently.

3.4 Asynchronous Event Mechanism

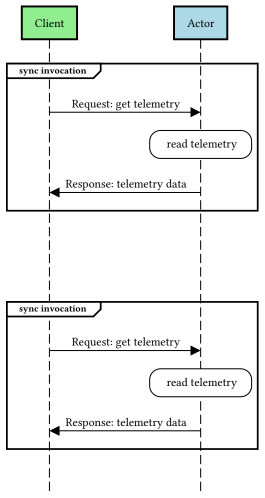
For applications such as telemetry data delivery, we often want to get periodical updates on a specific data source. Instead of polling the data source at the desired interval manually, a better pattern is to *subscribe* to the data source for update with the given interval, as shown in Figure 4B. This saves the client CPU cycles needed to generate the request for every update. More importantly, it also allows the collecting of temporally precise data points in comparison to synchronous invocation, where the exact time the sample is taken is unknown, as shown in Figure 4A.

Another important use case of the asynchronous event mechanism is for *asynchronous invocations*. Although the base protocol is defined to be synchronous, we recognise that some operations may inherently take a long time to finish. It would be wasting resources if we keep the client busy looping and waiting for the response. We support asynchronous invocations through the asynchronous event mechanism by registering the invocation synchronously with an ID and later issuing a *one-shot* event with the same ID to signal completion. The client can then handle this event and complete the call. This process is demonstrated in Figure 4C.

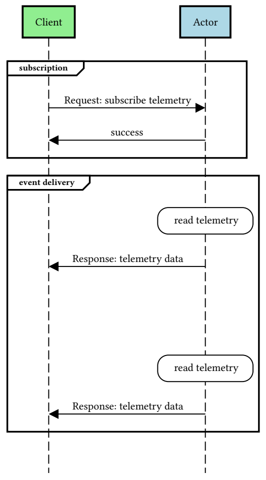
One point to note for implementation is that the event channel has to be a separate *logical* channel from the usual synchronous channel. This can be either a different physical channel or multiplexed over the same physical channel. The reason is straightforward: the synchronous channel does not expect event messages and would not be able to correctly handle such messages.

3.5 Symmetric Client/Actor Design

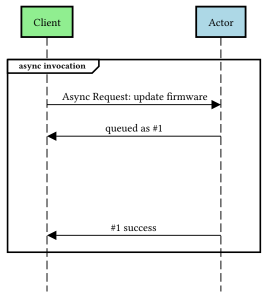
An important aspect in the design of EFRI is that the protocol is *symmetric* in regard of the hardware components in the system. In most traditional designs, the CPU is implicitly assumed to be the center of the entire system i.e. the



A The usual synchronous invocation model to acquire sensor samples periodically.



B Event subscription and subsequent delivery for periodic sensor samples.



C Asynchronous invocation implemented with the asynchronous event mechanism.

Figure 4: Demonstration of use cases of the asynchronous event mechanism.

initiator; the FPGA or BMC can only be the actor. This is not the case in EFRI, as the CPU can also implement services as actors to expose resources, such as performance counters or even OS-defined services. The BMC can also have clients, for example a shell to control resources in the system. This view provides unmatched flexibility for heterogeneous systems which previous solutions cannot offer.

4 Implementation

In this section, we discuss one implementation of the proposed protocol on the Enzian platform. Due to the time constraints of the project, we only implement the most essential components as a Proof-of-Concept (PoC) for the protocol. An overview of the reference implementation is shown in Figure 5. We document how these software components are implemented and the trade-offs. Nonetheless, we also discuss how the other components can potentially be implemented for future reference.

Hardware The Enzian v3 platform, codenamed *zuestoll*, has a Cavium ThunderX-1 CPU and a Xilinx XCVU9P FPGA as the main application processors connected by the ECI interconnect; a block diagram can be seen in Figure 1. In addition, there is a Zynq-7000 BMC on board, whose Programmable Logic (PL) is connected to the CPU and FPGA via low-speed IO pins. A UART link is

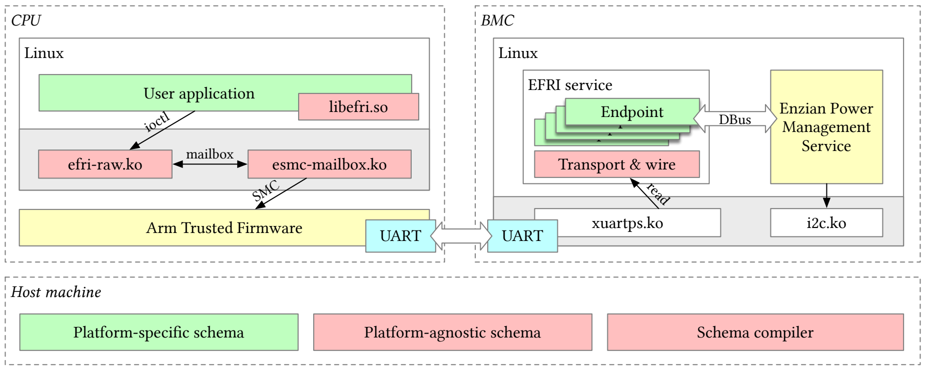


Figure 5: Overview of the reference implementation of EFRI on Enzian. Red denotes common EFRI infrastructure that is common between platforms; green denotes platform-specific bits from the system integrator; yellow denotes existing platform systems.

implemented between the CPU and BMC for EFRI, while a similar setup is possible between the BMC and FPGA. A link for EFRI between the CPU and FPGA can either be a direct one over ECI or forwarded by the BMC.

Terminology The *client* and *actor* discussed here refer to roles in the protocol, which may not entirely correspond to hardware devices in the platform. Each device can have more than one client of the protocol; for example, the CPU has a *user* client that invokes telemetry requests, while inside the *firmware* it has another PSCI client that issues power-down calls.

4.1 The schema compiler

EFRI specifies resources on platforms using *schemas* to reduce hand-written boilerplate and enforce consistency. The schema is written in YAML and defined in two parts: *platform-agnostic* and *platform-specific*. The platform-agnostic schema defines objects shared by all actors and clients. This includes basic types transmitted on the protocol, for example, error values, integers, floating point numbers, etc. It also defines the four common actions of endpoints, `put`, `get`, `subscribe`, and `list`. The common nodes are referenced by the platform-specific part of the schema to reduce redundancy.

The platform-specific part of the schema specifies various resources available on a platform, ranging from voltage and current readings of power rails, RPM of chassis fans, configuration parameters, etc. Each *endpoint* is denoted by the resource **name**, **actions** on the resource, a **class-object-subsystem** tuple to uniquely identify the resource, the **type** of the data, the implementation **backend** of the endpoint, and a human-readable **description** for a self-documenting implementation. Common resource types are defined as template nodes and exposed as YAML *anchors* for maximum reuse.

The schema compiler loads the platform-specific YAML schema. With a plugin, it also additionally loads platform-specific extra endpoints defined by external structural data, for example, voltage rails defined in the Enzian BMC Power Management project. It then checks all the endpoints to see if the required properties are defined. Finally, it generates marshalling and unmarshalling code with string templates, including type bindings, for all actors and clients. It also generates the *endpoint implementation database* for actors, which describes what concrete actions the actor will take for an incoming request. This will be described in detail in Section 4.3.

4.2 CPU

The CPU currently has three protocol-level roles proposed in the system, among which two are implemented. The PSCI client and the user-mode client (except for subscriptions) are implemented and tested to be functional. The CPU-side telemetry actor is proposed but not yet implemented.

Shared infrastructure The ATF codebase hosts most of the common software components shared by all roles on the CPU. The core component is the EFRI link layer, which performs two duties: 1) marshalling and unmarshalling an EFRI message into the *wire format* and 2) transmitting and receiving marshalled messages over the UART link. They are packaged into a library, `libefri`, for different clients to call into, either directly inside ATF or across the EL3 boundary via Secure Monitor Calls (SMC).

The wire format is a simple ASCII-based textual serialisation of an EFRI payload for quick prototyping; an example can be found in Figure 6. While it is possible to adopt a more sophisticated encoding, for a PoC implementation a human-readable wire format is a lot easier to implement and debug. Values are always typed, allowing the unmarshalling code to assign appropriate type tags, which the application logic can then check against. The clients and actors in EFRI never get hold of the raw wire format, but instead always the unmarshalled structure (or possibly a further encapsulated format). Thanks to this modular approach, a more compact format can be adopted in the future if needed. (*requirements* **(C)** **(E)**)

The most basic unmarshalled structure, and so far the only implemented one, uses the *argspage* convention. Specifically, it is a C structure with trailing data to store strings in the message. When a field of the structure contains a string, a `ptrdiff_t` pointing to the string in the trailing data, with the start of the structure as the pointer base, is stored. An example of the *argspage* convention can be seen in Figure 7. The *argspage* is used to pass a message between software components.

Multiple alternative designs of an unmarshalled data structure exists. Simple solutions include encoding strings directly inside the structure, either as a fixed-length buffer or with a length tag. The downside of a fixed-length buffer is that an artificial upper limit of length must be determined, which results in either too much limitation or waste of memory. Other complicated encoding formats

$$\underbrace{\texttt{==EFRI== =>}}_{\text{request preamble}}\ \underbrace{\texttt{platform:fans:CASE\_FAN\_0:rpm}}_{\text{FRN}}\ \underbrace{\texttt{get}}_{\text{action}}\ \texttt{\textbackslash n}$$

$$\underbrace{\texttt{==EFRI== <=}}_{\text{response preamble}}\ \underbrace{\texttt{platform:fans:CASE\_FAN\_0:rpm}}_{\text{FRN}}\ \underbrace{\texttt{get}}_{\text{action}}\ \underbrace{\texttt{Error\$0}}_{\text{error code}}\ \underbrace{\texttt{Rpm\$300\#RPM}}_{\text{value}}\ \texttt{\textbackslash n}$$

Figure 6: Example request and its response of a telemetry request for the RPM of a case fan. The wire format is tokenised with spaces and terminated via a newline character (`\n`). Notice the type tags for the error code and fan RPM data.

would require another layer of parsing, complicating code interacting with the structure. The *argspage* design is easy to implement with a simple reinterpret pointer cast, with its security and portability implications. If a bullet-proof design is desired, a mechanism such as ProtoBuf [Google LLC(2023)] can be adopted, thanks to the modular design of the protocol. (*requirement* **(D)**)

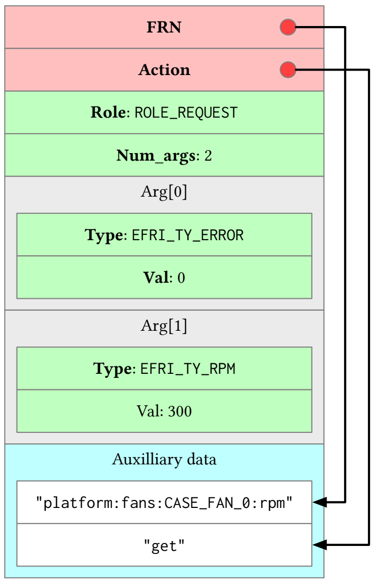


Figure 7: The unmarshalled *argspage* for the response `==EFRI== <= platform:fans:CASE_FAN_0:rpm get Error$0 Rpm$300#RPM\n` (from Figure 6).

ATF client The Power State Coordination Interface (PSCI) handles power-down requests from the operating system in the firmware. On a ThunderX system, this is handled by sending an IPMI request to the BMC. We implement the power-down handler as sending a power-off request to the BMC, specifically by putting 0 to `cpu::power`. The implementation assembles the *argspage* statically and calls into `libefri` to transmit the request.

Another proposed use case inside EL3 is to host the DRAM parameters and UEFI variables on EFRI. This is currently difficult to implement, due to DRAM currently being initialised in the Board Development Kit (BDK) of ThunderX and not inside the ATF; a more detailed description of this situation can be seen in Appendix A. It should be easy to implement in BL1 of ATF after the convergence of the two codebases.

User-mode client Most use cases of EFRI originate from user-space applications on the CPU. We currently implement the simplest way to expose EFRI to the user by allowing them to construct the *argspage* directly, and then calling into the `efri-raw` Linux kernel module. It then in turn calls into the `esmc-mailbox` kernel module, which calls into the ATF with a SMC call. The ATF then uses the common EFRI infrastructure there to talk to the BMC. The entire call graph is shown in Figure 8. In realistic situations, the platform developer should expose a better encapsulated and limited interface and perform checks at each level.

The user application constructs the *argspage* with the help of a user-space `libefri`, pushing strings into the auxiliary buffer as shown in Figure 7. The *argspage* has in addition a small header to encode the destination device (BMC or FPGA) and length. The destination is not part of the protocol, as we do not implement forwarding of messages from one device to another at the moment. The length is an optimisation and also not part of the protocol, as most of the times we only need to copy part of the *argspage* from user-space into kernel memory. After constructing the *argspage*, the user application then calls `ioctl` on a device file exposed by the EFRI kernel driver, passing the pointer to the *argspage* with the header. Finally, the user application receives the response *argspage* inside the same buffer it passed to the kernel.

The Linux kernel module `efri-raw` handles the `ioctl` call by making use of the Linux Mailbox Framework [Brar(2023)]. A mailbox is a hardware component to deliver messages between processors, usually in a single System-on-Chip (SoC). The Linux framework allows part of the kernel to be registered as a *mailbox controller*, which handles transmitting and receiving messages to and from a hardware mailbox. A *mailbox client* in another part of the kernel then produces and consumes the messages. With this design of decoupling the transport from the user-facing interface, we can in the future provide better-encapsulated interfaces than `efri-raw` that, for example, exposes only a subset of all resources and performs permission checks, without having to rewrite the firmware-facing `esmc-mailbox` module.

Upon an `ioctl` call, `efri-raw` first copies the *argspage* from user space into the kernel buffer. Then, as a mailbox client, the kernel module passes the buffer pointer to the `esmc-mailbox` (**E**xtended **S**ecure **M**onitor **C**all, ESMC) module, which implements a mailbox controller. After finishing, the ESMC mailbox

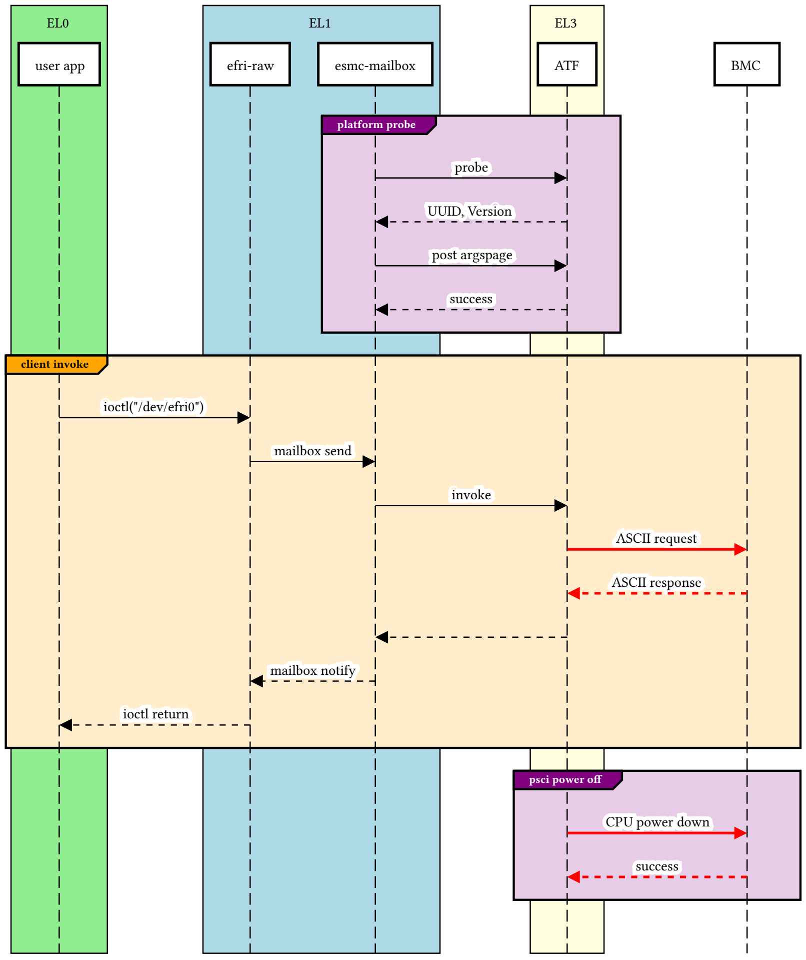


Figure 8: Call routine of three example CPU-side actions. The platform probe happens at load time of `esmc-mailbox`; client invoke happens when a client calls into EFRI; PSCI power off happens when the OS shuts the system down.

controller module passes the response *argspage* via the mailbox framework back to `efri-raw`, which in turn passes it back to the userspace in the `ioctl` buffer.

The `esmc-mailbox` module is the mailbox controller for EFRI requests, interfacing with the system firmware instead of actual mailbox hardware. It performs platform probing on load by trying to invoke various SMC calls to check for the vendor UUID and protocol version, to ensure that it invokes the right EL3 service. After probing, it allocates a physical memory page and posts this via an SMC call to the EL3 firmware as a shared buffer. This method is adopted since the standard SMC call convention does not support passing arbitrary amounts of data; at most 8 words (64 bytes) can be passed as arguments and 4 (32 bytes) as return values in one SMC call. Upon receiving the posted buffer, the EL3 firmware maps the page as Non-Secure Read-Write, allowing shared access from both EL1 and EL3.

Upon receiving a mailbox send request from the `efri-raw` client, `esmc-mailbox` copies the buffer contents (without the `ioctl` headers) into the shared *argspage* page and issues an SMC call into EL3 firmware. The EFRI runtime service inside the firmware handles the call in a blocking fashion. It first copies the contents of the shared *argspage* into a private buffer, to mitigate potential Time-of-Check-to-Time-of-Use (TOCTTOU) attacks [Wei and Pu(2005)]. After that, the firmware marshals the request, transmits it over the UART link to the BMC, and spin on the UART link to receive the response from the BMC. Finally, it unmarshals the request back into the shared *argspage* page. The firmware also returns time measurements taken in EL3 as the SMC call's return values, which are described in detail in Section 5.2.

A proposed but not yet implemented aspect of the user-mode client on the CPU is the asynchronous event mechanism for the subscription model. We need an *upcall* [Barton-Davis(1998)] from the EL3 firmware into the EL1 OS. This is standardised by Arm on ARM platforms as the Software Delegated Exception Interface (SDEI) [Arm Limited(2023)] for delivering platform events to the operating system. We can implement the routine by routing the interrupt from the UART controller to EL3, which then generates a software exception into the kernel. The kernel could either post the data into some buffer and wait until the client application polls, or deliver it directly as a signal. We would also need to implement channel multiplexing on the UART link, including dispatching incoming messages to the corresponding channel and associated buffering.

CPU-side actor As discussed in the protocol design in Section 3.5, the CPU can potentially expose various resources for other clients in the system, for example, performance counters of the CPU and the OS. Some requests, such as CPU performance counters or status registers, can already be served inside the ATF. Other more complicated requests would need to be serviced with an upcall into the OS. This can be implemented on the CPU side with the asynchronous event mechanism (once it is implemented): the EL3 can generate a software exception for an incoming request for handling in the OS. The exact semantics of such OS-driven services are left for future work to explore.

4.3 BMC

The BMC currently has the actor part facing the CPU implemented. In general, the implementation is written in Python and runs as a standalone process on the BMC. The Python script will eventually be integrated into the OpenBMC distribution as a `systemd` service.

As an actor The Python service receives marshalled requests in the EFRI ASCII wire format as shown already in Figure 6. After unmarshalling the message, the service tries to match the FRN and action name against the *endpoint implementation database.* If a matching one is found, the service then executes the corresponding action and generates the response. Finally, the response is marshalled and sent back on the UART link, before proceeding to blocking-receive the next request. Any errors during this process will result in an error response being sent back, and the service should never crash.

The endpoint implementation database is generated by the schema compiler to select a concrete backend for all endpoints. The compiler generates instantiations of backend implementation classes according to the node of the resource in the schema. Different implementation classes consume different properties in the resource definition. A concrete example can be seen in the case study in Section 5.1.

We abstract the concept of *datasources* that support `get` actions and optionally `put` and `subscribe` actions. The actual data source is defined by further specialisation, either from a *dummy* data source during development that simply holds the last stored value or a data source that talks with the BMC power management service over DBus for real telemetry data. Another common endpoint implementation is for binary switches backed by Linux commands to flip the switch on or off. This is used to implement the power-on/off actions since the high-level functionalities of the power management service are not exposed over DBus yet, but possible with a shell command.

An important subsystem for evaluating the performance of the action implementations is the *benchmarking* subsystem. A global `Benchmark` singleton class in Python allows the entire processing pipeline to inject timestamps for a detailed breakdown of the request processing procedure. The benchmark database control is exposed over EFRI to the client as `bmc:benchmark::control` for starting and stopping timestamp collection. The collected timestamps are dumped onto the BMC as a JSON file for later inspection.

As a client The BMC currently does not have a client role implemented, but we propose the possible use cases here. Note that multiple client roles on the BMC still make sense even if the only actor device in the system. An important use case is for out-of-band management (OOB). For example, we can expose EFRI actions as commands for a shell interface to the BMC. This will allow simplification and unification of the existing administrative interfaces. We could potentially also expose a Redfish-style RESTful API for standardised remote management.

4.4 FPGA

We currently do not have a universal shell on the FPGA side. When this is implemented in the system, many services can be exposed to EFRI from the FPGA; for example performance counters, tenant status and control in a multi-tenancy setup, and possibly a reconfiguration interface. The shell can further provide EFRI client and/or actor interfaces to the applications for further integration with EFRI. Another possibility is to use the FPGA to instrument the CPU over ECI, for example with the Arm CoreSight infrastructure [Arm Limited(2021)]. EFRI can also provide a nice, standardised interface for this.

5 Evaluation

In this section, we evaluate the proposed design and reference implementation. We first present two case studies of the implementation of resources in EFRI to showcase the extensibility and usability claims. We then show two performance evaluations to give a rough impression of the overall runtime cost of the protocol implementation.

Experiment setup We run all the experiments on Enzian v3 (*zuestoll*) machines. Specifically, the CPU is a 48-core Cavium ThunderX-1 Processor. The BMC, at the time of writing, is a Xilinx Zynq 7000 SoC. The exact commit hashes and repo URLs can be found in Appendix C. The UART link between the BMC and CPU is configured with a 115200 baud rate, 8 data bits, no parity bits, and one stop bit (115200-8-N-1).

Toolchain overview The EFRI toolchain builds the implementation on a development machine for deployment onto a *target* machine i.e. an Enzian. The schema compiler runs on the development machine and consumes the input YAML schema and any additional structured data, as described in Section 4.1. It then produces generated source files for each hardware component, C files for ATF on the CPU, and Python files for the service on the BMC. The C sources are compiled with ATF into the fimware image to flash onto the CPU. The Python files are deployed to the BMC to run as a `systemd` service.

5.1 Case studies

We show case studies of how the developer would extend the functionality of a system implementing EFRI. We present the changes needed for adding a resource to an actor. We then show the corresponding client change to call the newly-added resource.

CPU power-down The resource needed for controlling the binary power state of the CPU is a binary switch. This is defined first in the platform-agnostic schema, `efri-schema.yml`, as a shared anchor as shown in Listing 1a. A node for the CPU power switch is then defined in the platform-specific schema for Enzians, `enzian-efri.yml`, as shown in Listing 1b.

The `switch-cmdline` implementation backend specifies that this action is implemented by executing a shell command. The schema compiler generates three entries in the endpoint implementation database: the endpoint `cpu::power`, and the containing namespace for the two actions ::power:cpu:power and the parent `::power:cpu`. The developer extends the codegen logic to correctly generate the new endpoint entry, as shown in Listing 2. The entries are shown in Listing 3.

Finally, the developer defines the `CmdlineSwitch` class as a specialisation of the `DataSource` endpoint backend. The implementation overrides behaviour for the `put` and `get` python methods to execute commands accordingly. This is demonstrated in Listing 4.

The client of this resource is in the ATF PSCI `system_off` handler as shown in Listing 5. The client code constructs the *argspage* statically and then invokes the EFRI routines to transmit the request to the BMC. Since the constructed *argspage* does not cross the ESMC boundary, we do not have the requirement to fit the entire structure into a single page. Thus, we directly subtract the start of the frame from the static string pointer to calculate the required `ptrdiff_t` values.

Power rail telemetry Power rail telemetry data include voltage and current readings for all rails defined by the Enzian power management service. Since there exists already a structured representation of all the rails in the system and their topology data, we do not duplicate these in the schema. The schema compiler directly loads them from the Python modules and generates the nodes. Nevertheless, we have the shared anchor for voltage rails defined as shown in Listing 6, and the compiler generates the nodes by overriding them.

The implementation backend for power rails, `datasource-dbus-power`, talks to the Enzian power management service over DBus. The backend is read-only and accepts a (device, monitor) tuple to read from a monitor node in the power service. The mapping from the power rail name to the tuple is acquired through the topology database in the `enzianbmc` Python package. Similar to the case of power switches, the developer extends the codegen logic as shown in Listing 7. An example of generated endpoint implementation database entries is shown in Listing 8.

The developer defines the `DBusPowerDataSource` Python class to implement these endpoints. It connects to the system bus, issues a `read_device_monitor` message on the Enzian power service, and returns the value. A simplified implementation can be seen in Listing 9.

The client application for acquiring telemetry data runs in Linux userspace and calls into EFRI through the `efri-raw` and `esmc-mailbox` kernel modules,

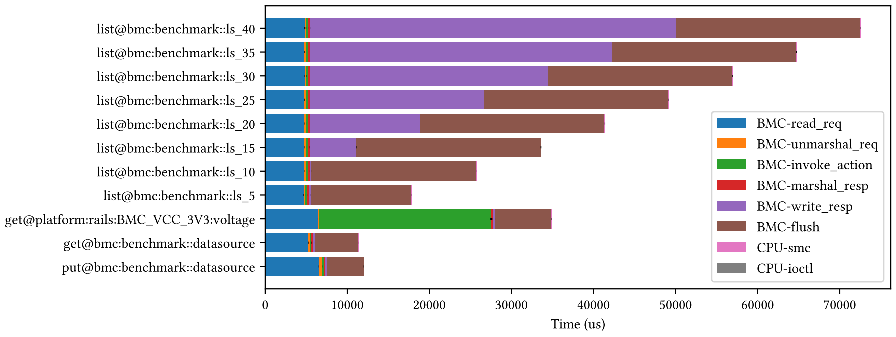


Figure 9: Timing results for invoking different endpoints. The bars with caption *CPU-* denote latency measured on the CPU side; bars with *BMC-* are on the BMC side.

as discussed in Section 4.2. The client code only has to `open` the device file, `ioctl` the *argspage* buffer, and read the response back. An example is shown in Listing 10.

5.2 Performance

We present a basic performance test of the reference protocol implementation between the CPU and BMC, as described in Section 4. The round-trip time is measured by calculating the time difference between invoking the action on the client side and the return of the `ioctl` call. We turn off all debug outputs.

Timing measurements Multiple points in the entire invocation chain take timestamps to measure the time elapsed during various stages of processing. On the CPU side, the user-mode client times the time of the `ioctl` call to get the end-to-end latency of a call. This does not include further user-space processing, such as assembling the *argspage* and consuming the response. The EL3 firmware times the entire SMC handling time, as well as the time spent transmitting the request and receiving the response. It returns these time measurements in the SMC return values to the OS, which will then pass them back to user-space. The BMC actor side further uses the benchmarking framework as described in Section 4.3 to collect fine-grain timestamp data. The timing measurements from both sources are analysed and plotted off-line on the host machine.

Latency breakdown We measure the time consumption of invoking data-source endpoints and break down the latencies introduced by each stage of processing. We present three types of endpoint operations: a dummy data source without real backend processing (FRN `bmc:benchmark::datasource`), a dummy namespace endpoint with synthetic children (FRN `bmc:benchmark::ls_*`), and a real data source with DBus backend (FRN `platform:rails:BMC_VCC_3V3:voltage`).

As shown in the legend of Figure 9, the timestamps mean the following:

- *read_req*: time spent for BMC to read the request over UART
- *unmarshal_req*: time spent for BMC to unmarshal the request
- *invoke_action*: time spent for BMC to execute the given action
- *marshal_resp*: time spent for BMC to marshal the response
- *write_resp*: time spent for BMC to *enqueue* the marshalled response (not flush!)
- *flush*: time spent for BMC to flush UART buffer onto the wire
- *smc* and *ioctl*: time spent due to SMC/`ioctl` call overheads

We observe that the most significant overhead of the protocol is transmitting and receiving the ASCII-format message over the UART link (*read_req* for request and *write_resp* plus *flush* for response). This overhead is linear to the length of the marshalled request and response. This is more significant when listing larger namespaces. This shows that the implementation still has much potential for optimisation to reduce latency if a more compact wire format is used, or if a higher baud rate is used on the UART link.

Another observation is made from the comparison of a real data source over a dummy one. In calling `get` on the platform rail voltage, significant latency comes from the *invoke_action* phase, which is absent in the dummy data source. This corresponds to invoking the power management service over DBus, which in turn reads the voltage data from I2C. The ratio of this latency compared to other latencies shows that the wire format overhead is comparable to the necessary latency for doing useful work.

An additional interesting observation is that the UART implementation on the BMC has internal buffering corresponding to approximately 20 milliseconds of UART activity; data less than this buffer would be enqueued immediately and only transmitted during buffer flush. If the client pushes through more data, the writing process would then block, resulting in latency under the *write_resp* category. This provides a perspective of the speed of the UART link between the CPU and BMC, as well as internal buffering of the Python serial library.

6 Discussion

While the design presented here is complete and can accommodate most use cases of the protocol, some aspects remain unimplemented due to time constraints on the project. Some design choices have been made towards simplicity instead of efficiency, which leaves large room for optimisations. The codebase is also at the time of writing not yet integrated fully into the Enzian ecosystem. Nevertheless, we believe that this work and its artefacts should serve as a good starting point for future efforts in these directions.

Another important topic is how the design and implementation would evolve while the rest of the Enzian software ecosystem rolls forward. We already mentioned a merge between the BDK and ATF codebases and tracking upstream TF-A. As the ATF implementation of EFRI is structured modularly with the library and runtime service interfaces, which still exists in upstream TF-A, a migration should not require too much effort. The new codebase would also bring much useful new features such as SDEI to our utility.

The BMC-side software and hardware also expects to see drastic changes with the overhauling to the new Zynq UltraScale+ MPSoC platform, as well as an ongoing effort to switch the software stack to seL4. The switch to MPSoC should be easy to accomodate once the CPU-facing UART port is set up correctly for Linux, since the current EFRI codebase on the BMC is purely in Python. The adoption of seL4, especially if we decide to get rid of Python, would be a more involved task; we should be able to reuse most of the ATF codebase for the base protocol (wire format, etc.). But since we would not have DBus, the IPC scheme between the EFRI and power management services remains to be figured out. This then needs to be adapted into the EFRI codebase.

7 Conclusion

In this work, we introduced the Enzian Firmware Resource Interface (EFRI). As presented in the motivation in the work, EFRI is a better fit for platform-level management tasks on heterogeneous systems in comparison to existing industrial standards that had similar but different goals targeting conventional PC, server, or SoC systems. EFRI, being designed specifically with extensibility and flexibility in mind, is a great fit as the firmware protocol on heterogeneous systems such as Enzian.

We further demonstrated that the proposed protocol is practical to implement and satisfies the requirements we put forward in the motivation. As part of the evaluation, we presented a reference implementation on Enzian. We further showed the practicality using two extensive case studies and performance characterisation of the reference implementation. They should establish a sound basis for a future where EFRI is the real *common substrate* of heterogeneous platforms.

Acknowledgements

The author would like to thank Daniel Schwyn, David Cock, Ben Fiedler, and Michael Giardino for their extremely helpful comments and discussions in shaping the design of the protocol, as well as practical help with the implementation. The authors would also like to thank the Enzian team for reviewing the report and providing valuable feedback.

References

[Agron et al.(2006)] Jason Agron, Wesley Peck, Erik Anderson, David Andrews, Ed Komp, Ron Sass, Fabrice Baijot, and Jim Stevens. 2006. Run-Time Services for Hybrid CPU/FPGA Systems on Chip. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. 3–12. https://doi.org/10.1109/RTSS.2006.45

[Andrews et al.(2004)] D. Andrews, D. Niehaus, and P. Ashenden. 2004. Programming models for hybrid CPU/FPGA chips. *Computer* 37, 1 (2004), 118–120. https://doi.org/10.1109/MC.2004.1260732

[Arm Limited(2013)] Arm Limited. 2013. *ARM Trusted Firmware Design*.

[Arm Limited(2021)] Arm Limited. 2021. *Learn the architecture - Introducing CoreSight debug and trace*. https://developer.arm.com/documentation/102520/0100.

[Arm Limited(2022a)] Arm Limited. 2022a. *Arm Server Base System Architecture Platform Design Document*. Version 7.1.

[Arm Limited(2022b)] Arm Limited. 2022b. *Arm System Control and Management Interface Platform Design Document*. Version 3.2.

[Arm Limited(2022c)] Arm Limited. 2022c. *SCP-firmware - version 2.11*. https://github.com/ARM-software/SCP-firmware.

[Arm Limited(2022d)] Arm Limited. 2022d. *Trusted Firmware-A*. https://github.com/ARM-software/arm-trusted-firmware/.

[Arm Limited(2023)] Arm Limited. 2023. *Software Delegated Exception Interface (SDEI) Platform Design Document*. Version 1.1.

[Barton-Davis(1998)] Paul Barton-Davis. 1998. *upcalls*. https://lkml.iu.edu/hypermail/linux/kernel/9809.3/0922.html.

[Belwal et al.(2015)] Meena Belwal, Madhura Purnaprajna, and Sudarshan TSB. 2015. Enabling seamless execution on hybrid CPU/FPGA systems: Challenges & directions. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2015.7294022

[Brar(2023)] Jassi Brar. 2023. *The Common Mailbox Framework*. https://docs.kernel.org/driver-api/mailbox.html.

[Brickell et al.(2004)] Ernie Brickell, Jan Camenisch, and Liqun Chen. 2004. Direct Anonymous Attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (Washington DC, USA) *(CCS '04)*. Association for Computing Machinery, New York, NY, USA, 132145. https://doi.org/10.1145/1030083.1030103

[Cavium Inc.(2016)] Cavium Inc. 2016. *Cavium CN88XX Evaluation Base Board User's Guide.*

[Cisco Systems Inc.(2020)] Cisco Systems Inc. 2020. *Cisco Integrated Management Controller (IMC) Data Sheet.* https://www.cisco.com/c/en/us/products/collateral/servers-unified-computing/ucs-b-series-blade-servers/data_sheet_c78-728802.html.

[Cock et al.(2022)] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. 2022. Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 434451. https://doi.org/10.1145/3503222.3507742

[Corbet(2014)] Jonathan Corbet. 2014. *ARM, SBSA, UEFI, and ACPI.* https://lwn.net/Articles/584123/.

[Dell Inc.(2022)] Dell Inc. 2022. *Redfish API with Dell Integrated Remote Access Controller.* https://www.dell.com/support/kbdoc/en-us/000178045/redfish-api-with-dell-integrated-remote-access-controller.

[Dell Inc.(2023)] Dell Inc. 2023. *Integrated Dell Remote Access Controller (iDRAC).* https://www.dell.com/en-us/dt/solutions/openmanage/idrac.htm.

[DMTF(2022)] DMTF. 2022. *Redfish Specification.* Version 1.17.0.

[Frazelle(2020)] Jessie Frazelle. 2020. Opening up the Baseboard Management Controller: If the CPU is the Brain of the Board, the BMC is the Brain Stem. *Queue* 17, 5 (jan 2020), 512. https://doi.org/10.1145/3371595.3378404

[Free Software Foundation, Inc.(2021)] Free Software Foundation, Inc. 2021. *GNU GRUB.* https://www.gnu.org/software/grub/.

[Google LLC(2023)] Google LLC. 2023. *Protocol Buffers.* https://protobuf.dev/.

[Herdt et al.(2017)] Vladimir Herdt, Hoang M. Le, Daniel GroSSe, and Rolf Drechsler. 2017. Towards early validation of firmware-based power management using virtual prototypes: A constrained random approach. In *2017 Forum on Specification and Design Languages (FDL)*. 1–8. https://doi.org/10.1109/FDL.2017.8303898

[Hewlett Packard Enterprise Development LP(2022)] Hewlett Packard Enterprise Development LP. 2022. *Redfish API implementation on HPE servers with iLO RESTful API.* https://www.hpe.com/psnow/doc/4AA6-1727ENW.

[Hewlett Packard Enterprise Development LP(2023)] Hewlett Packard Enterprise Development LP. 2023. *HPE Integrated Lights-Out (iLO).* https://www.hpe.com/ch/de/servers/integrated-lights-out-ilo.html.

[Intel et al.(2013)] Intel, Hewlett-Packard, NEC, and Dell. 2013. *Intelligent Platform Management Interface Specifciation v2.0.* Revision 1.1.

[Iorga et al.(2021)] Dan Iorga, Alastair F. Donaldson, Tyler Sorensen, and John Wickerson. 2021. The Semantics of Shared Memory in Intel CPU/FPGA Systems. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 120 (oct 2021), 28 pages. https://doi.org/10.1145/3485497

[Microsoft(2022)] Microsoft. 2022. *Measured boot and host attestation.* https://learn.microsoft.com/en-us/azure/security/fundamentals/measured-boot-host-attestation.

[Minyard(2023)] Corey Minyard. 2023. *The Linux IPMI Driver.* https://docs.kernel.org/driver-api/ipmi.html.

[Muralidhar et al.(2012)] R Muralidhar, H Seshadri, V Bhimarao, V Rudramuni, I Mansoor, S Thomas, B Veera, Y Singh, and S Ramachandra. 2012. Experiences with power management enabling on the Intel Medfield phone. In *Proc. of Linux Symposium.* 35–46.

[Oracle(2014)] Oracle. 2014. *Oracle Integrated Lights Out Manager (ILOM) 3.0 HTML Documentation Collection.* https://docs.oracle.com/cd/E19860-01/E21549/toc.html.

[OSDev Wiki(2023)] OSDev Wiki. 2023. *AML — OSDev Wiki.* https://wiki.osdev.org/AML. [Online; accessed 25-January-2023].

[Ottaviano et al.(2022)] Alessandro Ottaviano, Robert Balas, Giovanni Bambini, Corrado Bonfanti, Simone Benatti, Davide Rossi, Luca Benini, and Andrea Bartolini. 2022. ControlPULP: A RISC-V Power Controller forăHPC Processors withăParallel Control-Law Computation Acceleration. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Alex Orailoglu, Marc Reichenbach, and Matthias Jung (Eds.). Springer International Publishing, Cham, 120–135.

[Richardson and Ruby(2008)] Leonard Richardson and Sam Ruby. 2008. *RESTful web services.* " O'Reilly Media, Inc.".

[Seshadri et al.(2004)] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. 2004. SWATT: softWare-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004.* 272–282. https://doi.org/10.1109/SECPRI.2004.1301329

[Super Micro Computer Inc.(2023)] Super Micro Computer Inc. 2023. *Supermicro Server Management (Redfish API)*. https://www.supermicro.com/en/solutions/management-software/redfish.

[The Linux Foundation(2021)] The Linux Foundation. 2021. *OpenAPI Specification v3.1.0*. Version 3.1.0.

[The OpenBMC Community(2022a)] The OpenBMC Community. 2022a. *IPMI Architecture*. https://github.com/openbmc/docs/blob/master/architecture/ipmi-architecture.md.

[The OpenBMC Community(2022b)] The OpenBMC Community. 2022b. *Redfish*. https://github.com/openbmc/bmcweb/blob/master/Redfish.md.

[The OpenBMC Community(2023)] The OpenBMC Community. 2023. *OpenBMC*. https://github.com/openbmc/openbmc.

[The TianoCore Community(2021)] The TianoCore Community 2021. *EDK II*. The TianoCore Community.

[The U-Boot Development Community(2021)] The U-Boot Development Community 2021. *The U-Boot Documentation*. The U-Boot Development Community.

[UEFI Forum Inc.(2022a)] UEFI Forum Inc. 2022a. *Advanced Configuration and Power Interface (ACPI) Specification*. Release 6.5.

[UEFI Forum Inc.(2022b)] UEFI Forum Inc. 2022b. *Unified Extensible Firmware Interface (UEFI) Specification*. Release 2.10.

[Wei and Pu(2005)] Jinpeng Wei and Calton Pu. 2005. TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4* (San Francisco, CA) *(FAST'05)*. USENIX Association, USA, 12.

A Overview of the ThunderX Boot Process on Enzian

The ThunderX-1 CPU on the Enzian platform has ARMv8 AArch64 cores and has four Exception Levels (EL): the *EL3* for the secure monitor, the Arm Trusted Firmware (ATF) in this case; the *EL2* for the hypervisor, if any; the *EL1* for the operating system kernel, for example Linux; and the *EL0* for user-space applications. On Enzian we have control of the EL3 software by flashing the `bootfs`[1] from the BMC[2].

[1] https://unlimited.ethz.ch/display/sgnetoswiki/Enzian+Software

[2] https://unlimited.ethz.ch/display/sgnetoswiki/Enzian+BMC#EnzianBMC-FlashingfromLinux

The CPU boots from the NAND flash upon power-on, loading the Board Development Kit (BDK)[3] and runs it in EL3. The BDK then initialises various hardware devices, including the UART used by EFRI between the CPU and the BMC (this is called the UAA on ThunderX [Cavium Inc.(2016)]) and the DRAM timing parameters (currently this is hard-coded). After finishing, the BDK then chain-loads the Arm ATF Boot Loader stage 1 (BL1) in EL3. At the time of writing, there is an on-going effort[4] to make the BDK and ATF code-bases converge, in order to reduce redundant and often contradictory hardware initialisation, as well as making use cases of EFRI such as storing DRAM parameters possible. This will also bring new upstream features in ATF (renamed to TF-A), such as the SDEI infrastructure.

The ATF BL1, after finishing its platform early initialisation, loads BL2, which in turn loads BL31, the *EL3 Runtime Firmware* [Arm Limited(2013)], in EL3. BL31 sets the EL3 exception vector to allow processing of service calls (SMC), loads the next-stage boot loader, the TianoCore EDK II UEFI[5], and returns to it in EL1. The UEFI supplies the device tree in `ArmPlatformPkg` and loads the GNU GRUB [Free Software Foundation, Inc.(2021)], which in turn loads the Ubuntu Linux kernel.

B Code Listings

We show the code listings in this section. Note that the listings are simplified for clarity; full resource repositories can be found in Appendix C.

List of Code Listings

[3]`https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-bdk`
[4]`https://gitlab.inf.ethz.ch/PROJECT-Enzian/arm-trusted-firmware-enzian-port/-/tree/v2.6-enzian`
[5]`https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-uefi`

```
actions:
    put: &put
        name: put
        args: [p]
        retval: [err]
    get: &get
        name: get
        args: []
        retval: [err, T]
aliases:
    rw: &rw [*put, *get]
    ro: &ro [*get]
binary_switch: &binary_switch
    type: binary
    actions: *rw
```

(a) Shared anchor for binary switches.

```
power_switch: &power_switch
    name: power
    subsystem: power
    <<: *binary_switch
bmc:
    - class: [cpu]
      impl: switch-cmdline
      do_off:
↪   /home/root/pengxu/shell.py bringup
↪   'cpu_power_down()'
      <<: *power_switch
```

(b) Enzian-specific CPU power node. Notice that the power switch is specialised into a regular binary switch by setting the action name and subsystem properties.

Listing 1: Platform-agnostic and platform-specific schema for the CPU power switch. Irrelevant nodes are removed for clarity.

```
def gen_ep(ep: Endpoint):
    ...
    elif ep.impl.startswith('switch'):
        if 'cmdline' in ep.impl:
            ep_impl = f'CmdlineSwitch("{frn}", {actions}, '
            for attr in ['on', 'off', 'query']:
                a = f'do_{attr}'
                if hasattr(ep, a):
                    ep_impl += f'{a}="{getattr(ep, a)}", '
            ep_impl += ')'
        else:
            assert False, 'unknown switch impl ' + ep.impl
    ...
```

Listing 2: Codegen logic for generating endpoint database entries for binary switches. The generator only accepts `switch-cmdline` here.

```
FRN_EP_MAP["cpu::power"] = CmdlineSwitch("cpu::power", [
    ActionDesc("put", 1, 1),
    ActionDesc("get", 0, 2)
], do_off="/home/root/pengxu/shell.py bringup 'cpu_power_down()'", )
FRN_EP_MAP["::power:cpu:power"] = NamespaceImpl("::power:cpu:power", [
    ("put", FRN_EP_MAP["cpu::power"]),
    ("get", FRN_EP_MAP["cpu::power"])
])
FRN_EP_MAP["::power:cpu"] = NamespaceImpl("::power:cpu", [
    ("list", FRN_EP_MAP["::power:cpu:power"])
])
```

Listing 3: Generated endpoint implementation entries for the power switch in the database, according to the schema. Note that the parents `:` and `::power` are generated later due to presence of other siblings.

```
E_RUNCMD = TypedValue(Error, 6) # EFRI_INTERNAL_ERROR
class CmdlineSwitch(DataSource):
    def _run_cmd(self, action, cmd):
        if not hasattr(self, f'do_{cmd}'):
            raise UndefinedActionError(action, self.frn, f'do_{cmd} undefined
            ↪  for CmdlineSwitch')
        return subprocess.run(getattr(self, f'do_{cmd}'),
        ↪  shell=True).returncode
    def put(self, val: TypedValue):
        super().put(val)
        cmd = 'on' if val.val == 1 else 'off'
        try:
            ret = self._run_cmd('put', cmd)
            if ret == 0:
                return E_OK,
            else:
                return E_RUNCMD
        except subprocess.SubprocessError:
            return E_RUNCMD,
```

Listing 4: `CmdlineSwitch` class for executing a command to implement an action. Note that the querying state (aka `get`) is not supported for the power switch yet due to limitations of the power service. Irrelevant code is removed for clarity.

```c
void psci_system_off(void) {
    char link_buf[PAGE_SIZE]; size_t bytes_written; efri_error_t err;
    efri_frame_t frame = {
        .frn = "cpu::power" - (char *)&frame,
        .action = "put" - (char *)&frame,
        .role = ROLE_REQUEST,
        .num_args = 1,
        .args = { { .ty = EFRI_TY_BINARY, .val.opaque = "0" - (char *)&frame },
↪   },
    };

    err = efri_frame_marshal((char *)&frame, link_buf, &bytes_written);
    if (err != EFRI_OK)
        for (;;);
    efri_transmit(link_buf, bytes_written);
    for (;;);
}
```

Listing 5: Handler for PSCI power off request. Note that the *argspage* is constructed by directly computing the pointer differences.

```yaml
power_rail: &power_rail
    actions: *ro
    impl: datasource-dbus-power
    class: [platform, rails]
    subsystem: telemetry
rail_voltage: &rail_voltage
    name: voltage
    type: voltage
    <<: *power_rail
rail_current: &rail_current
    name: current
    type: current
    <<: *power_rail
```

Listing 6: Schema anchors for power rail telemetry.

```python
def gen_ep(ep: Endpoint):
    ...
    if ep.impl.startswith('datasource'):
        ds_common = f'"{frn}", {ep.type.name.capitalize()}, {actions}'
        if 'dummy' in ep.impl:
            ep_impl = f'DummyDataSource({ds_common}, readonly={"ro" in
            ↪  ep.impl})'
        elif 'dbus-power' in ep.impl:
            # ep should have device and monitor properties set
            assert hasattr(ep, 'device') and hasattr(ep, 'monitor'), \
                'datasource-dbus-power requires device and monitor set'
            ep_impl = f'DBusPowerDataSource({ds_common}, "{ep.device}",
            ↪  "{ep.monitor}")'
        else:
            assert False, 'unknown datasource impl ' + ep.impl
    ...
```

Listing 7: Codegen logic for generating endpoint database entries for DBus data sources. The generator accepts `datasource-dummy` and `datasource-dbus-power` here.

```python
FRN_EP_MAP["platform:rails:VDD_DDRCPU13:voltage"] =
↪  DBusPowerDataSource("platform:rails:VDD_DDRCPU13:voltage", Voltage, [
    ActionDesc("get", 0, 2),
    ActionDesc("subscribe", 2, 1)
], "pac_cpu", "VMON9")
FRN_EP_MAP["platform:rails:VDD_DDRCPU13:current"] =
↪  DBusPowerDataSource("platform:rails:VDD_DDRCPU13:current", Current, [
    ActionDesc("get", 0, 2),
    ActionDesc("subscribe", 2, 1)
], "ina226_ddr_cpu_13", "CURRENT")
```

Listing 8: Generated endpoint implementation entries for the power rail `VDD_DDRCPU13` in the database, according to the schema. Notice the different (device, monitor) tuples for the voltage and current endpoints.

```python
E_DBUS = TypedValue(Error, 6) # EFRI_INTERNAL_ERROR
class DBusPowerDataSource(DataSource):
    @classmethod
    def _get_service(cls):
        return dbus.SystemBus().get_object('systems.enzian.Power',
        ↪  '/systems/enzian/Power')
    def get(self):
        try:
            raw_val = self._get_service().read_device_monitor(self.device,
            ↪  self.monitor)
        except dbus.DBusException:
            return E_DBUS, None
        return E_OK, TypedValue(self.ty, raw_val)
```

Listing 9: Simplified `DBusPowerDataSource` class for the power rail endpoints.

```c
#include "libefri.h"
int main(int argc, char *argv[]) {
    int fd = open("/dev/efri_raw0", O_RDWR);

    // fill argspage
    struct efri_raw_invoke_msg msg;
    efri_frame_t *frame = &msg.frame.frame;
    char *cur = msg.frame.aux;
    frame->frn = push_string(msg.buf, &cur,
↪   "platform:rails:BMC_VCC_3V3:voltage");
    frame->action = push_string(msg.buf, &cur, "get");
    frame->role = ROLE_REQUEST;
    frame->num_args = 0;
    msg.dest = EFRI_DEST_BMC;
    msg.size = cur - msg.buf;

    // invoke and print result
    ioctl(fd, EFRI_RAW_INVOKE, &msg);
    describe_frame(frame);
}
```

Listing 10: Simplified userspace application for invoking the voltage endpoint `get` action. Error checks are omitted for clarity.

C Locations of Implementation

The implementation code for EFRI on Enzian is separated into two repositories, the Enzian ATF repository[6] (`2f95648`), which contains the CPU-side EFRI ATF service, and the EFRI repository[7] (`6997f37`), which contains the schema compiler, the Enzian EFRI schema, kernel modules and userspace utility on the CPU side, and the EFRI service on the BMC side. There are minor changes to the `bringup` script in the Enzian BMC Power Management Tools[8], as well as the UEFI repo[9] (`cbc60ff8cf`) for the device tree.

[6]https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-atf

[7]https://gitlab.inf.ethz.ch/OU-ROSCOE/Students/2022-prj-pengxu/enzian-firmware-resource-interface

[8]https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-bmc-powermgmt

[9]https://gitlab.inf.ethz.ch/PROJECT-Enzian/enzian-uefi

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

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

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

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

Titel der Arbeit (in Druckschrift):

Enzian Firmware Resource Interface

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

Name(n):	**Vorname(n):**
Xu	Pengcheng

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

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

Ort, Datum	**Unterschrift(en)**
Zürich, 17.02.2022	[illegible]

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