Heikki Kilpeläinen

Dynamic firmware updating of an embedded system

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Bachelor's Thesis

23 February 2023

Abstract

Author:	Heikki Kilpeläinen
Title:	Dynamic firmware updating of an embedded system
Number of Pages:	49 pages
Date:	23 February 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Smart Systems
Supervisors:	Keijo Länsikunnas, Senior Lecturer

Software updates are important for maintaining functionality of an embedded systems. For a firmware update to take effect, the system's memory must be re-programmed, and its execution re-initialized through a power-cycle. However, it is not always a feasibly conductible operation due to the requirements set by the system's deployment environment. Autonomously and programmatically performed firmware update without external assistance could provide new use cases for systems which do not benefit from the conventional update methods.

The objective of this thesis was to design a method for conducting a dynamic firmware update on an embedded system. It was required to receive and apply updates to the program's functionality during runtime without a need for a power-cycle. To promote unbiased problem solving, the method was developed in an experimental and explorative research which was based on a progressive literature review and a current state analysis.

The specific topic was found to be rather unprecedented in its existing field of research. Throughout the process the work's requirements were observed increasing in complexity, which resulted in the time requirements for solving all the problems becoming unknown. Despite that, the essential problems were identified for further development.

As an outcome, the work resulted in a theoretical foundation for the method, which enables continuing development for a possible future implementation. Until a testable implementation the work's feasibility against the objective can not be reliably evaluated.

Keywords: ARM, DSU, Embedded systems, firmware, memory reconstruction

Tiivistelmä

Tekijä:	Heikki Kilpeläinen
Otsikko:	Sulautetun järjestelmän laiteohjelman dynaaminen päivittäminen
Sivumäärä:	49 sivua
Aika:	23.2.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Älykkäät järjestelmät
Ohjaajat:	Lehtori Keijo Länsikunnas

Ohjelmistopäivitykset ovat tärkeitä sulautetun järjestelmän toiminnallisuuden ylläpitämiseksi. Niissä laiteohjaimen päivittäminen vaatii uudelleenohjelmoinnin lisäksi järjestelmän nollaavan uudelleenkäynnistyksen. Se ei kuitenkaan ole aina käytännöllisesti toteutettavissa käyttökohteen aiheuttamien rajoitusten tai vaatimusten takia. Laitteen itsenäisen päivityksen suorittaminen ohjelmallisesti ilman ulkoista ihmis- tai laiteapua voi mahdollistaa ohjelmistopäivityksen käyttötarkoituksissa, joissa perinteiset menetelmät eivät olisi eduksi.

Opinnäytetyön tavoitteena oli suunnitella menetelmä sulautetun järjestelmän laiteohjelmiston päivittämiseksi dynaamisesti. Sen tuli mahdollistaa ohjelmiston toiminnallisuuden muuttaminen etäältä suorituksen aikana ilman tarvetta järjestelmän uudelleenkäynnistykselle.

Menetelmä kehitettiin käytännönläheisen kokeilevan tutkimuksen ja tutkitun kirjallisuuden yhteensovituksessa, jolla pyrittiin korostamaan intuitiivista ongelmanratkaisua. Sovellettu teoria kartoitettiin asteittain mukautetulla kirjallisuuskatsauksella ja analyysillä aiheen nykytilasta. Laiteohjaimen dynaaminen päivittäminen sulautetuissa järjestelmissä osoittautui odotettua vähemmän tutkituksi aiheeksi aihepiirissä.

Prosessin myötä vaatimusten havaittiin monimutkaistuneen. Sen takia ongelmien ratkaisuun vaadittavan ajan voitiin todeta laajentuneen ennalta määrittelemättömäksi. Siitä huolimatta toiminnallisesti oleelliset ongelmat tunnistettiin jatkokehitystä varten.

Työ tuotti menetelmälle teoreettisen pohjan. Se mahdollistaa jatkokehityksen kokonaisvaltaiselle käytännön toteutukselle, jota ilman tavoitellun menetelmän mielekkyyttä ei voi luotettavasti arvioida.

Avainsanat: ARM, DSU, sulautetut järjestelmät, laiteohjain, muistirekonstruktio

Contents

List of Abbreviations

API:	Application programming interface
CPU:	Central processing unit
DSU:	Dynamic Software Update
IAP:	In-application programming
IO:	Input and output
OTA:	Over-the-air
RAM:	Random-access memory
ROM:	Read-only memory
UART:	Universal asynchronous receiver-transmitter
WSN:	Wireless sensor network
XIP:	Execute in place

1 Introduction

This thesis is about designing a dynamic firmware update method for an embedded system. The objective is to design a method that can be applied in a microcontroller.

A firmware and its updates in an embedded system needs to be programmed to the system's persistent memory with external aid and a power cycle is generally required for it to take effect. Some systems can be applied in such tasks that do not benefit from a reboot or try to avoid losing their runtime memory state during maintenance. For example, sensors, subsystems, Internet of Things, machine learning and edge computing systems which are beyond practical reach either because of their physical distance, environmental restrictions, or hazards.

In such cases, redundant or additional hardware is not necessarily required for a dynamic firmware update if it can be conducted from within the running system without the need for a power cycle. The dynamic update functionality is a software feature in the target system's firmware which provides support for receiving and applying modifications to the functional program's executable code and current state during runtime.

The programmer's intention for a microcontroller firmware update can manifest in a variety of ways. The reason can range from a minimal patch to a completely exchanged firmware. Hence, different levels of understanding the problem and the target hardware are required to better utilize all the possible expressions of a firmware update.

However, the update method does not take a position on the means of possible over-the-air (OTA) data transfer or its encryption schemes. The method defines only the core interfaces and functionality, which an embedded system requires to achieve the dynamic firmware updating. This fundamental approach emphasizes the work's scope and prepares ground for later implementations to test and evaluate its feasibility.

Methodology for defining and solving the problem focuses on literature research on related topics and explorative experiments on a microcontroller software. The subject's origin comes from a personal interest to the dynamic firmware updating and no third parties are involved, nor has one commissioned the work.

The thesis first defines the foundation of the subject and the fundamentals of the problem solving that are required to meet the stated objectives (chapters 2 and 3). It then continues to the design process which walks through a research for the dynamic firmware update method proposition (chapter 4). Finally, the work is summarized and concluded in chapter 5.

2 Definitions

This chapter defines the essential concepts related to the work's subject, which provide sufficient prerequisites for understanding the detailed problem introduction in the next chapter. The concepts include definitions of the embedded systems elements, firmware, and its updating processes. From the definitions, a proper frame of reference can be established to outline the work's scope.

2.1 Embedded systems

Embedded systems are electronic programmed computing ensembles dedicated for specific tasks whose requirements cannot generally be met by conventional means. Usually, the tasks solve problems that are not efficient or meaningful to be carried out with general-purpose computers or pose otherwise special restrictions. Such as dimensional limitations, physical reach, requirement to be embedded into other systems or objects, or to interact with specialized peripheral devices and interfaces. Because of their target use cases, the systems can usually be expected to be reliable in terms of error resistance, self-recovery, and uptime. [1], [2, pp. 26-31], [3, pp. 5-13]

It is meaningful to distinct the concept from general purpose computer systems as certain characteristics will be expected throughout the work [4, p. 5]. Some generalizing examples of different industries among civil and public sectors where the embedded systems are applied are mobile and wearable electronics, production and infrastructural systems, agriculture, healthcare, and medical devices. Other examples include vehicles and robotics in terrestrial, aerospace, and maritime environments. They practically cover every electronic computing device with a function, which is not a personal computer or a server featuring a general-purpose operating system. However, some problems might require the system to manage many tasks concurrently. In such cases, embeddable customized operating systems can be used. [2, pp. 14-26], [5, pp. 1-2], [6], [7, p. 53], [3, pp. 5-7]

Apart from them, some systems implement real-time operating systems for managing multiple tasks that require hard real-time requirements for computational accuracy [1]. This is a widely applied property of embedded systems [8], which also must be considered in the dynamic firmware updating problem later.

2.2 Computer organization

A computer organization is the non-functional specification of the system hardware which describes the fundamental electronic components and the implementation of their interactive relations according to the functional requirements defined by an architecture. They can be defined in various ways, which ever works best for the different architectures. As the hardware ultimately represents the properties of a system, it is fair to identify what components are essential to be paid attention to, and to know how their onboard implementation can affect the software design. [7, p. 26], [4]

In modern computers, their functionality to perform interactive and programmable computations to produce output from an input, depends on five fundamental components. To begin with the functional point of view, the input and output (IO) devices can be perceived being the most relevant for establishing the interfaces for external interactions with the system. For the computation in between the two, a memory and a controller along with a data path are required to manage the information between the components. [4, pp. 16-17]

Electronic embedded systems, being computers, share these fundamentals with the other modern general-purpose computer systems [9, p. 6]. Yet they essentially perform computational tasks which control the system according to programmed instructions and make it interactive [4, p. 17]. While the technologies and implementations keep evolving, the fundamentals remain. Their computational function is usually implemented by a microcontroller architecture [10].

The diversity of engineering problems the embedded systems solve, causes non-uniform requirements for the microcontroller implementations, which manifests in a great variance of hardware configurations. Therefore, it is not necessarily practical to have a specified architecture for a common reference. [10], [3, pp. 11-13]

The components and interfaces together form the system's hardware, which can also be described as the hardware's resources. The next sections will introduce them and their relevance to this work in more detail.

2.3 Microcontroller

A microcontroller unit (MCU) is designed to provide all the functions required by its intended purpose. It is an ensemble of integrated hardware. Not only it implements the controller and data path, conventionally central processing unit (CPU), but many or all the other fundamental components, peripheral devices, and interfaces, on a single integrated circuit, a chip [10]. This increases conveniency, design and economy wise, since only the limited requirements apply, and less general-purpose functions need to be taken in consideration [7, p. 56]. Therefore, a microcontroller can implement a subsystem of an embedded system or define one itself.

Of the fundamental components, the CPU functions by conducting arithmetic and logic operations with instructions and data stored in the memory [4, pp. 62-63] [4, p. 68], which are specified by an instruction set architecture [3, pp. 131-135]. It can consist of one or more processor cores for controlling the operations, along with an arithmetic logic unit, data registers, intermediary cache memory and interfaces [7, p. 30], [3, p. 129].

To enhance the MCU's performance and applicability, additional peripheral controllers can be integrated to it to relieve some computational burden of specific functionalities from the CPU [7, pp. 56-63]. Such as, memory management, IO, timing, interrupts, data error checking and encryption. Some

components, usually types of extended memories, and various external peripheral devices can be interfaced separately off-chip, through electronic digital or analogue interfaces [10,11].

The logical relations of the chip hardware are defined by computer architectures. Despite the MCUs lack a standardized reference architecture, at least two fundamental approaches for managing the memory model are extensively applied [10,12]. Generally, a CPU is designed in two major architectures, referred as von Neumann and Harvard [3, p. 175].

They both describe the same fundamental components, but they manage the memory model for instructions and data differently. The former defines instructions and data to reside within a same memory, which can only provide access to one of them at a time. The latter defines separate memories and interfaces for instructions and data which a processor can access simultaneously. Depending on the task, MCUs can benefit from both architectures, or variants of them. [12]

2.4 Memory

Computations and programmability require storage in various forms, which are types of memories. The different types reflect their purpose in varying sizes and operating principles. They can be considered volatile or persistent non-volatile memories (NVM), along with dedicated access types, which allow read-only or read and write operations on them. They range from small, fast, and volatile processor registers and caches to random-access memory (RAM) modules, to usually larger, but comparatively slower, long-term NVM data storage mediums. Such as solid-state and hard disk drives. [3, pp. 166-178], [3, pp. 231-250]

While the NVM is primarily considered for reading and writing arbitrary data, they can alternatively be implemented as read-only storage. In such case, the intention implies that the memory only needs to be the size of the desired

content. The property is relevant to consider for the design and project implementation.

The volatile memory, by definition, can only hold data when the component is powered, thus losing its content in the event of shut down or unexpected power loss. In NVM, the data remains without the same power requirements, making it possible to store long-term data. [3, pp. 166-178]

A CPU utilizes the former type as registers for computations and caches to temporarily store data closer than RAM, allowing faster access [3, pp. 174-176]. Caches can be classified in levels for their organizational implementation. By Noergaard's example, a level-1 cache is integrated directly on-chip, and the levels increment the further from the CPU they are. On system start up, application programs are usually loaded to the RAM from the NVM [3, p. 166], again for faster access. A firmware, which will be introduced in chapter 2.6, is immutable by design and is therefore stored in persistent read-only memory (ROM) types.

However, MCUs do often utilize the long-term memory differently by executing its program, the firmware, in-place instead of loading it to RAM first. This concept is referred as execute in place (XIP), which sets a few practical requirements for the storage medium and system organization. As the program execution tends to require speed, the memory must have a sufficient performance and ability to return the smallest data units, words, when read. Also, the code complexity can render the program's memory size requirements larger than what could be supported on-chip. In such case, the memory can be interfaced externally. [13,14]

A convenient type for the modern MCU semiconductor NVMs is the Fujio Masuoka's electrically erasable programmable read-only memory, which is commonly called a flash memory [15, pp. 5-9]. However, the flash does experience wear over quantitative use, which has negative effect on the component's physical durability by decreasing its lifetime [15, pp. 137.-141]. As introduced by Richter, the issue can be addressed with various methods of

wear levelling [15, pp. 141-144]. The concept is important to consider, as the firmware, and possibly numerous updates of it, are to reside on a flash memory.

On logical level, the flash works by managing the data in arrays of semiconductor memory cells, which are commonly based on implementations of Intel's NOR, or Toshiba's NAND logic gate architectures. They describe the cell interconnections which ultimately determine how the data is written or read. The former can manage the data with much more fine-grained precision than the latter, from single bits to full memory. NAND flash manage the cells in larger units, which are called pages. Furthermore, pages can be organized into sectors, which usually denotes the smallest erasable page-aligned unit. Yet, both require erasure of their corresponding memory before being able write to them. [15, pp. 32-46]

From a functional point of view, the NOR flash types are ideal for embedded program memory and XIP, as single data words can be reasonably written and read fast [15, pp. 35-39]. Also, modern programs can be complex, and therefore large in size, to which flash can be economically scaled. Both flash architectures implement various version for different use cases. With the MCUs in focus, the NOR flash versions can enhance the comparatively slow programming cycle rate furthermore by implementing the interconnections in serial rather than in parallel. In such configuration, the memory cells are organized and managed in larger chunks of cells, called pages. They are physically the smallest units of manageable data, whose smaller arbitrary word-size units can be virtually accessed through page buffers, which will manage the page read, erase, and write operations accordingly. [16,17]

A widely applied implementation is based on serial peripheral interface protocol (SPI), which defines the page size to 256 bytes. Despite the organizational compromises, the SPI NOR flash still benefits from the naturally fast and fine-tuneable read operations, making it an industry standard flash memory type among the embedded systems. [16]

The relevance of knowing the introduced properties and usage of various program memories will pose an important part in the design of the firmware updating for an MCU. Yet, memories are the components where the firmware resides and executes. For conveniency, regardless of the implementation-specific terminological derivations of the read-only memory for the firmware, it will be simply referred as a ROM type throughout the upcoming sections and chapters.

2.5 Peripheral devices and interfaces

Due to the interactive purpose of the embedded systems, their deployment environments can pose complex models from which the information is interacted with. Primarily, the environments a system is deemed interfacing are the analogous world and other digital systems. Electronic sensors along with integrated or external amplifiers and analogue-to-digital converters are such type of peripheral devices that translate the analogue information into processable digital data [18, pp. 327-330]. In both cases, the further data processing can introduce complexity, which might cause intolerable performance decrease, if conducted exclusively on the MCU's main processor. [7, pp. 53-57], [7, pp. 253-256], [3, pp. 253-256]

In addition to sensor applications, peripheral devices can be dedicated to processing the externally received data to appropriate format, while only interfering the system when necessary. This relieves load from the MCU's processor, and it can enhance the system's performance. The concept works the other way around as well, to generate appropriate output from internally processed or stored data. In the context, a process can include controlling and data management through algorithmic manipulation, filtering, buffering, streaming, timing, debugging, generation, comparison, or conversion. Another functionality for the process can be interrupt signal management, which is an essential concept for the devices to interact with the MCU's processor. [19], [3, pp. 129-130], [7, pp. 253-267]

Depending on the application, the MCU can integrate such devices directly on-chip or connect to them via various electronic interfaces [3, pp. 253-256]. While many aspects of an embedded systems are seemingly tailored for a specific purposes, the common interface specifications for the IO have become actual or de facto standards [10]. Common to engineering, the proven models and methods tend to be the most adopted, which is beneficial to the work when peripheral devices and interfaces are considered.

The peripheral devices come with great diversity and unrestricted complexity requirements, function-, and implementation wise. Yet, a rough functional classification can be drawn according to their varying purposes. While ignoring their organizational implementation in a system, they can represent devices which either exchange information through interfaces or process data which is related to the IO of the former, or internal computations. [7, pp. 253-255]

As an example, a few relevant, usually integrated, peripheral device types can be named. Such as, various timers and triggers, general-purpose input and output, switch matrix, converters from analogue to digital or vice versa, interrupt controller, direct memory access, cyclic redundancy check engine [20] for data integrity verification, data encryption engine, codec, and wireless radio. [7, pp. 60-63], [7, pp. 272-277], [19]

To establish a physical interactive connection to the peripheral devices, whether internal or external to the MCU, they require mutual electronic interfaces. They are specifications for physical analogue or digital medium interconnections which exchange the intended electronic information according to their communications protocol. The mediums are commonly referred to as buses, which along their protocols are designed for specific properties. Such as, being optimized for information transmission reliability, reach, scalability, or throughput. [3, pp. 254-255], [3, pp. 277-282], [19]

The interfaces with their respective communications protocols can be classified by their hardware implementations, which are based on serial- or parallel electrical topologies [3, 257-269], [7, pp. 33-39]. The former is more utilized in

modern MCU designs. Also, the protocols manage the data either with or without synchronization [3, pp. 184-187]. A few commonly integrated peripheral device interfaces are universal asynchronous receiver-transmitter (UART) with optional driver device, SPI, and inter-integrated circuit buses [10,19].

Regarding the software updating of an MCU, peripheral devices are especially utilized, as they provide the IO interactions to and from the system. Therefore, the distinction of various device types and interfaces among the identified common cases represents a more abstract set of factors that can affect the later design and cause possible limitations.

2.6 Firmware

Firmware is an architecture-dependent low-level system software of an MCU, which controls the system hardware. Its primary task is to initialize and configure the hardware environment for the components to interact, and therefore allow the functional application software to work properly. Since embedded systems tend to perform dedicated tasks, the application code with the device drivers, and possibly an operating system, are usually included in the firmware. It can represent the whole fundamental software stack which defines the embedded system functionalities introduced in above sections of the chapter. [21] [3, pp. 311-315]

As referred by Tan et al., the firmware is defined, by an international standard [22], to be read-only software. In practice, this implies that they are usually stored in ROM types, regularly flash memory. Therefore, programming the firmware to ROM might require dedicated tools for writing to the memory, depending on the memory type and its supported programming methods. The software is represented in the memory as machine-readable instructions, referred as machine code or binary [7, pp. 730-750].

Despite the firmware being considered low-level software, it is not unequivocally the only nor the lowest level of code the system might start with. Depending on

the application and system implementation, some components, for example RAM, might require initialization prior to firmware execution. Such program is a bootloader, or multiple of them for various initialization stages. In context of an MCU and embedded applications, the bootloader is commonly used to program the ROM with a firmware, either initially or when being updated. [21,23,24]

From the software developer's point of view, the firmware is programmed in source code in a suitable programming language. Because of the low-level placement and interactions with the hardware, the firmware's source code is required to be compiled into the machine code with a target system's architecture specific compiler. In the context, C and assembly programming languages remain common, as they can express the hardware, especially the memory related, operations efficiently. A program can be compiled on arbitrary host system architectures other than the target if a corresponding compiler toolchain, a cross-compiler, of the target is used. The practice is a common approach in embedded system firmware development. [3, pp. 30-33], [21]

Moreover, the process of compiling programs involves stages that are worth noting on a general level. The written and referred source code gets gathered and compiled into object code files, which contain the machine code in binary format. However, they don't yet have their remote code references or their relation to the target memory model resolved. The final stage is to link the compiled object code accordingly into an executable format, so that the included code is structured and resolved to be accordingly reachable by the target processor. [25, pp. 28-36], [26, pp. 251-259]

The structure of an executable code is processor architecture specific [25, p. 30]. It defines the code in memory mapped regions, which commonly includes the respective memory segments for the instructions and initial data [25, pp. 34-36]. Various executable formats exist, of which Intel hexadecimal object file format HEX [27,28] and executable and linkable format ELF [29] are relevant in context of the embedded systems [30, pp. 8-10], [25, pp- 30-31].

Overall, the section's relevancy is central, as the concept of a firmware will be wielded extensively. The remaining sections will introduce the principles of its update operations.

2.7 Firmware updating

Beyond seemingly successive deployment, the firmware is yet subject to many natural factors which can require maintenance for updating the software. Some of which are more critical in sense of safety or functionality. Such as, compilable technical- and semantic errors, bugs, as in any man-made software, which require patching. Other less dramatic reasons can involve software updating for functional improvements through tweaking and feature management.

The maintenance is conducted in accordance with the system's hardware properties, which is defined by the components and available interfaces. Overall, the conventional method for updating the firmware is inherently the same as for its initial programming to the memory, as introduced in the previous section. The system is brought to a halted state where the software can be written to the device's memory through an interface, which is configured for memory programming [23, p. 71908].

As an example, the programming interfaces can include the on-chip serial interfaces UART and SPI, which can be used for programming with aid of a bootloader. Further down the scope, the embedded system bootloaders can implement in-system programming and in-application programming (IAP) functionality [23,31]. The former requires an external input signal, and the latter can be managed completely from within the running software [32]. Other interfaces can involve Joint-Terminal-Action-Group standard (JTAG) [33] or its less featured alternative ARM implementation, Serial Wire Debug (SWD) [34], for the hardware debugger which both utilize the same operating protocol.

Although, the embedded systems, which are deployed in their production environment, tend to utilize the methods in more sophisticated and automated

procedures. As it is not always practical to maintain the systems manually by an operator, they favour additional software or hardware infrastructure to support remote management through a wired or wireless connection. The concept of wireless remote management is referred as an OTA method. Furthermore, an update event can minimize damage in case of a catastrophic failure by implementing resiliency with a software version rollback functionality to revert the defect update. [35–37]

As a process, the update through memory writing operations can have varying limitations due to the used hardware. Especially, with flash memory, the write cycle can be considered slow [25, pp. 39-43], [25, pp. 56-60], [15, pp. 127-131], [15, pp. 135-137]. As shown by Richter, the speed is limited by the used block size, the chunks of writeable binary, of the update and the physical writing speed. It can be concluded that the system's run-down, possible overhead of a remote update infrastructure, and memory's programming limitations correlate to the overall maintenance downtime directly.

Even though, the firmware update is considered being close to hardware, an operating system can be implemented for managing the system's functionality through application software [38]. In such case, the application-level software, or even some kernel modules, can be updated by the operating system [39]. This, however, does not apply to the actual firmware-level software but the scenario and its methods are worth taking in consideration.

2.8 Dynamic software updating

As the concept of updating a software dynamically is the work's subject it essential to be introduced on a functional level. The next chapter includes a current state analysis of the dynamic software update (DSU), which reviews the existing research and classifies related implementations in more detail. Despite the precise type of the involved software or its application domain, DSU can be scaled to operate on various levels of applications, including the firmware of an embedded system [40,41]. Hence, the section will refer to software generally.

The research for DSU dates back to mid-1970s [42,43] and the published implementations appeared in 1983 by Cook and Lee [44]. However, modern theory is based on intensively growing research from the last three decades, which address the requirements of the modern computing- and infrastructural evolution [40,45]. DSU theory and its application requirements do not appear to have a disclosed consensus or taxonomy. Although, various common concepts and problems are identified among the modern research [40,45,46].

From a user's point of view, a DSU method conducts a software update on a live system concurrently without halting and dispelling its volatile execution state. This results in the instructions and data in persistent memory and RAM being replaced according to a new version from within the system. Therefore, a system initialization or a power-cycle for the update to take effect is not necessarily required. The DSU event can be triggered as any conventional software update, which includes manual and automatic procedures being performed either locally or remotely. [40,42,47–50]

For achieving this functionality, the method must be able to manage the target application's underlaying levels of the software stack for concurrent operations. In case of a firmware, the hardware management is most certainly required as well. However, the method design introduces several problems which can require approaches with application- and hardware-specific distinctions. Such as, how to define suitable units of update information and how they are received concurrently during runtime. Also, how the received units can be transformed into executable format and applied to the running software in a timely manner. In other words, to update the system's execution state and program. [42,49,51]

Additionally, in the context of a DSU, a software roll-back functionality can be convenient for minimizing downtime in case of a catastrophic failure. Generally, the optional concept of reverting an update can be implemented to any kind of update. Yet, such functionality for a DSU would need to apply the dynamic update mechanisms recursively. If considered, it can therefore set additional requirements for the update method's architectural design. [37]

Similarly, as can be thought with conventional software updating, the DSU architecture would manifest in at least two logical entities. The updater which receives and applies the update on the target system and the external system which prepares and provides the update to the target.

The conveying medium and transfer methods between the entities for the update data do not fundamentally concern the DSU design. Yet, possibility for integration and interfacing of them should be taken in consideration in their design. Such as existing wireless capabilities and OTA-approaches [35].

In general, Frieder and Segal classify the approaches roughly by their functional implementations. The earliest hardware based DSU involves redundant computing hardware for switching the execution between the old and updated programs. It isn't necessarily practical for embedded systems which would benefit from the DSU in the first place. However, the modern approaches are based on completely software managed methods, which can be classified further depending on their intentions. [49]

Each fundamental or practical problem in the DSU design can spawn complexities of their own, as different target system configurations and DSU implementations can cause specific corner cases [52]. As an example, a system implementing a XIP paradigm for program execution likely utilizes different functionality for applying the updates than the ones with fully RAM-loaded software, as introduced earlier.

3 Scope

With the essential definitions at hand, this chapter provides a scope for the work to start with. It introduces the research problem in detail, continued by introductions to the methodology and materials which were chosen for solving a problem of its kind. They include the fundamental requirements, practicalities, and discuss about measurements to evaluate the later proposed design.

Also, the current state of the field of research is introduced in a cursory literature review. The identification of other applicable approaches to the technology can be further analysed to supplement the work's justifications with relevant comparisons.

3.1 The problem

The research problem revolved around the issue, that applying a firmware update in embedded systems requires external interaction. The update will also interrupt and initialize the current execution state because of a required system restart. Some systems either don't benefit from or tolerate such interruptive maintenances in all scenarios.

Under certain conditions the firmware updating could be implemented more efficiently, performance- and economy-wise, possibly allowing existence of new applications which do not tolerate excessive execution downtime or pose physical maintenance restrictions. The identified place for improvement for the problem formulated the idea into the following research question. How can an embedded system's firmware be updated dynamically during runtime?

Updates are important. Embedded systems can require software maintenance when it's firmware doesn't perform as intended, because of a flaw in the code, and lack correction or it is planned to have a new feature, semantic changes, adjustments, or the code is to be fully exchanged. Some of these operations can be required in times when the system is already deployed in production use and downtime through hard maintenance is not ideal or tolerated, either because of expenses or practicality.

Dynamic firmware update method could mitigate some of the named inconveniencies if it can be designed to meet the hardware requirements of the target systems. Hence, the objective addresses a design process for finding the method in such a way it would be applicable for a clearly specifiable group of targets which could be proven to benefit from it.

The work becomes relevant if such a method can be generalized for a group of systems which commonly share the ideal use cases and properties for adopting the method's non-functional requirements. Fundamentally, the method should avoid including unnecessary non-functional requirements, which can limit the general applicability.

A few concrete examples of the applications. Redundant hardware, co-processors, or subsystems are not necessarily required for applying the firmware update. For example, a remote sensor system could be adjusted while not losing too many samples. Also, remote-, practically unreachable-, or restricted systems could be updated with a new task while maintaining their autonomous responsiveness at some tolerable extent. Such as, probes or unmanned vehicles.

To clarify the scope more, this work doesn't take part in the means of transmitting the update to the target device. Them being physical mediums, communication protocols or data encryption schemes. The matter is ideally left implementation-specific to allow purposeful and flexible applicability.

Some requirements, especially the non-functional, are not practical to be specified in detail up-front, as designing is naturally a creative process which matures over iterations [53]. Therefore, they tend to refine over time. However, the initial functional requirements can be logically concluded from the research problem definition, some of which produce implicit restrictions for possible later implementation. They were not based on existing designs. Below are stated the functional requirements in a progressive dependency order.

- The system must receive and apply updates dynamically.
- The updates must be integral.
- The system must be able to retain its execution state during update.
- The system must be responsive after fatal execution failures.
- The design for meeting the requirements must be generalizable.

To introduce reasoning behind the initial functional requirements, the list items are broken down into explanatory paragraphs in the same order as they appear in the list.

By dynamicity, the system is meant to receive and apply firmware changes itself while executing. Thereby, it is required that the system can change its context of execution at some extent to manage the IO and updating independently from the application firmware. However, the system is not required to be applicable for time-critical applications, such as real-time operating systems, due to their hard time requirements.

In case of an externally caused transmission fault, the received update might not always be integral, and can therefore cause failure. The functional requirements do not take position on what transmission methods are to be used. Before being applied, the received update must contain the same information as was originally intended.

It is essential that the system can modify itself according to the update without hardware initialization, hence retaining the execution state when appropriate. This is intentionally specified as an ability. In some cases, an update might be intended to replace the whole firmware, therefore requiring a full system initialization.

Maintaining responsiveness in a controlled state on fatal execution failure is considered a mandatory requirement due to the intended use cases of an embedded system. A specific responsive action should not be dictated. The statement contributes to the requirement of the designed method being flexibly applicable. A practical implementation of the design would likely be expected to include self-recovery functionality for additional resilience. However, not all systems might have enough hardware resources for implementing it. Hence, it is better being left on an abstract level.

The design must result in a generalizable outcome to identify a group of target systems and applicable use cases. The more applicable the design becomes,

the more relevant it will be. In addition to the listed items, any other possible, mandatory, or optional, requirement that would amend or change the overall functionality will be discussed later in the design process.

To evaluate the outcome of the design, a practical implementation of it on a real hardware was found ideal for allowing consistent and well documented measurements. That is, a proof-of-concept. It would be unreliable to evaluate the design's outcome with other means. Also, an implementation could be tested on various levels of abstraction to identify suitable metrics from where the measurements can be recorded and observed. However, an implementation does not fit to the work's scope at practical extent.

3.2 Methods

The work focuses on solving the research problem by answering to the practice-based research question. Hence, the fundamental methods required for approaching the question are to be of empirical type. Despite the lack of fully functional implementation, the design inherently depends on researched knowledge on related topics, and how it can be applied to form an operating theory of a dynamic firmware update.

As the work's problem originated out of curiosity and an independent observation of the subject, initial hypotheses of various architectural designs were formed before further research was conducted. An important methodological decision was to continue working on the intuitive approaches before exposing them to external bias. The decision could potentially promote novelty and originality.

In case of a problem which is based on a system's function, it therefore depends on the system's properties. Hence, the intuition may not always be fully applicable on all levels of the design as is. It requires supplementary understanding of the domain. That is, fundamental and detailed technical knowledge of the system's operations. Therefore, the design process can be

classified as deductive research, as the design evolves from the established theory.

A work of such type, with an unknown degree of uncertainties, and on the other hand, being specifiable without strict restraints would benefit from being adaptable to unexpected solutions. For conducting the work, a combination of experimental and explorative strategy was chosen. As proposed by Oivo et al. in their paper [54], both strategies can enhance each other when applied in unison to a practice-based problem. The choice can be further justified by a conclusion from the comparison of other viable strategical approaches. While case studies, engineering method, and grounded theory of the existing dynamic firmware updating designs can give a good supplementary insight into the field, would they possibly alone limit the exploration. This choice commits to the previously justified intuitive-driven approach.

The methodological decisions face a few limitations, however. The experiments could end up producing false evaluations. As can be concluded from Wohlin, et al. [55] experiment process, if they were carried out with insufficient research or understanding of the background theory, that could lead not only to incorrect establishments but also to lacking analyses. Another shortcoming related to the strategies is to find a reasonable scope for the evaluation’s depth and time usage. Clear indicators for when an evaluation can be deemed sufficient and done, might be cumbersome to define if specifications and analysis methods lack determination.

In conclusion, the work is best conducted by deducting a design to answer the research question through an experimentally explorative approach, which combines unbiased hypotheses with researched information of the existing technology and technical properties of the target systems. The design’s exploration process is evaluated through controlled experiments based on the solution’s requirements.

3.3 Materials

The work began with a thought process and sketching to form initial unbiased hypotheses. Before introducing supplementary information and influence from the existing approaches to the technology, the feasibility of the initial ideas required fundament evaluation against the technical limitations of a selected target system. That information was available in forms of technical data sheets, and user manuals by the system manufacturer and community forum discussions. Data sheets are documents which provide the plain facts of the system properties and features. Applicable information on their intended usage is provided by the user manuals. Community forums, on the other hand, supplemented the information with the practical view of other developers on how and where to apply the system features in various implementation-related problems.

To converge the hypotheses into a single solid theory of operation through the design process, further information was first required to specify the requirements of the problem and plan the methodology. To apply the methodology as intended, a current state analysis of the technology was required to identify comparable approaches for the work's evaluation. The research continued for the background theory. Later, the experiments of the exploration relied heavily on the technical literature. More precisely, the material provided by the vendor of the target system, programming and assembly language specifications, and technical guides for the host operating systems which provided the development environments.

The foundation for the material consists electronic and printed literature of books, conference papers and whitepapers, which were sourced through library catalogues and internet search engine results. The results were analysed in a form of an adaptive literature review. It consisted of different granularities, of which first was to locate the proper research material by progressively reducing the levels of abstraction of the reviewed topics to integrate them in a scope.

Manual and partly automated methods were used for traversing the results. For the latter, a text analysis software Voyant-tools [56] was used, which was originally developed by Stéfan Sinclair and Geoffrey Rockwell. It can provide simultaneous analysis and comparison of text properties of multiple material in various visual and textual formats. The software can be either self-hosted or accessed online from domain voyant-tools.org. The self-hosted off-line option can be important for respecting the terms of licensed material when applicable.

First, a concept analysis was conducted to identify which terminology, and what kind of scope was associated with target literature. In addition to the work's topic, the material was also searched with more abstract words, to provide some offset from the exact title to uncover yet unfamiliar results. In general, the initial search patterns consisted of subjects of software and firmware, which were appended around verbs the concepts of updating and upgrading. As newfound terminology appeared, the pattern was altered along with more specific descriptions for the adjective of the key functionality, the dynamicity. Such as, concurrent update, hot swap [47,57, p. 30], hot patch, dynamic patch [58, p. 5], runtime patching [59], hot load, live patch [60], on-line version change [48], runtime updating [61], and live synthesis [62]. After the trends were identified, the subject in the search pattern was also extended with the target context of the embedded systems.

The most common sources for the results were digital libraries and publication sites of various communities in the field of the technology or universities. The selected material was scoped and filtered with further content analysis, similarly to the first phase. In general, the literature's scope of interest was based on the identified properties and concepts utilized by conventional and dynamic firmware updating.

The process for researching the other related background theory was similar. Although, the main material was based on physical and electronic books sourced from library catalogues, which was extended with occasional research publications on the topics. The searched topics contained memory

manipulation, information security, peripheral device interfaces and their protocols, operating system theory, microcontroller architecture, program execution, software compilation, programming languages, and other, sometimes vendor specific, features in their hardware.

3.4 Current State Analysis

The objective of the analysis was to gain understanding of the current state of dynamic firmware updating among embedded systems and DSU methods in general, which could be used to justify the relevance of this work, and to identify comparable implementations for evaluation. The raw material and research methods for the analysis, consisted of same types of literature and methodology as the for other theory. However, it was extended with lecture notes, code repositories and recorded videos of seminars or educators to extend the sources further. The queries were likely affected by a bias from the scope.

The research for this analysis was supplemented with various existing studies to support the conclusion, introduce more material, and to work as references for possible results comparison. The studies consisted of a comprehensive systematic mapping study [40] by Ahmed et al., a survey [52] by Miedes and Muñoz-Escoí, an analysis [63] by Mugarza et al., and a review [64] by Ilvonen et al. Some of the studies were partly based on same sources and publications that were also screened for other chapters.

Many works [41,45–47,50,51,57,65–72] of the selected theory, including a few [73–77] hinted by the study of Ahmed et al., had cited other within the same sampling frame. Few authors and works emerged repetitiously referred or the authors were co-authors of multiple works, which implies their creditability. This realization was verified by a Voyant-tools term analysis for the occurring authors.

The scope-wise appropriate results over the analysed material were typically whitepapers, conference papers and journal publications. The minority types

were various educational presentations, source code repositories, or patents. However, only a fraction of the overall material was relevant to the work. They mostly introduced research in favour of dynamic, or similar adjective, updating of higher-level software, rather than firmware in the specific sense of this work. While the terminology is not problematic itself, they seemed to cover system- and application software updating significantly more often in other contexts than among the embedded systems of the scope.

Some selected works, which could have been considered off-scope because of their target domain, provided otherwise applicable theory that had been extensively utilized in the relevant material. The resulting material was divided into theory- and implementation-related works.

The observation of the scarce results was also supported by the mapping study [40], in table 14, where the classification referred to resource constrained systems which represented less than seven percent of their publication samples. Although, some other potential targets such as distributed- and time-critical systems could have been partly included, their applications implied requirements or use cases mostly outside of the scope. Such as implementations for real-time operating systems, Java virtual machines, or not-targeted application programming languages. It is likely, however, that some cut work could be modified and ported to work for the embedded system firmware updating.

The reference studies, and the mapping study by Ahmed et al. especially, suggested that dynamic software updating in general is rather extensively researched field with a decent history of theory and application. Yet, the field tends to address applications which are fundamentally higher in the technical abstraction level of software than a firmware of an embedded system, thus rendering them mostly irrelevant to the scope. Their implementations and theories focused mainly on updating software dynamically on server- and wireless sensor network (WSN) environments. Mostly with POSIX- or TinyOS-

compliant operating systems, as the mapping study's table 16 of target programs also implies. [40, pp. 474-476], [52,63,64]

Review of the less represented theory and implementations, that indicated more suitable context, were refined with further analysis to validate their relevancy. The most prominent domains regarding the scope of interest were time- and safety-critical systems and WSN [40,63,78,79]. The former relied heavily on embedded real-time operating systems, which were scoped out due to their architecturally higher level in the software stack. The WSN seemed natural as they target resource constrained MCUs intended for instrumentation applications [40,74,79–81]. However, almost every DSU implementation for them was also based on an operating system or a virtual machine platform. Mainly TinyOS and its derivatives, with various network reprogramming protocols or Java virtual machine [82–85]. Many accompanying words and conclusions of the material implied that the update methods in general for both fields are under intensive research to enhance security and performance.

In addition to dedicated system designs, some programming languages support DSU rather comprehensively by design, such as some LISP dialects, Smalltalk, Erlang and various bytecode interpreter versions of Java, .NET, and Python [63]. Despite their potential and flexibility, they were not analysed any further due the set scope.

From the observations of the theoretical material and suggested implementation, a few recurring trends in design problems and approaches were identified. Despite the varying taxonomy and non-uniform classifications among the material, their concepts of the DSU converged into the following fundamental design problems which are ordered according to their logical dependency on each other.

- What application parts are to be updated dynamically?
- How are the parts updated?
- How is the update formed?

The first problem seems definitive for the whole DSU system design, as it renders practical implications to further design problem approaches. It essentially defines the update granularity, the kind of software entities that are to be subject to the dynamic updates, which is directed by the specified target application [52]. The material covered the granularity ranging from arbitrary changes at byte level up to full program changes, while many limited the changes only at function level.

Being the broadest, the second problem was often referred as a process of state transformation. It introduces questions for when and how the old program is changed to its updated version. The approaches for them manifests in update timing- and state transformer methods, which ultimately define the update safety and performance of the DSU design. Ahmed et al. had mapped various classifications of approaches with several distinct techniques, of which most referred ones were represented in their tables nine, ten and eleven [40, pp. 472-475]. The last table included complete models for expressing the DSU design theories.

Despite the rather wide range of classifications, they all seemed to answer the same questions of this fundamental problem in practice and could therefore be deemed bound to it. Although, the designs for the update timing and state transformations would affect to the update safety and performance, at least implicitly, some works were based on such aspects.

In general, the update timing was usually controlled in various forms of indicator points or other artificially added execution state observation techniques, which initiates the state transformation for applying the changes with a proper timing [73, p. 6]. In other words, a valid execution state of the program to receive changes. The timing methods could be defined in the target program’s source code or in the updater, either statically or dynamically [40, pp. 472-475].

The state transformation, which is being referred as an abstract concept, would cause the old program state to manifest updated. This includes updating of all memories involved with the program. On fundamental level, common methods

for it introduced various forms of execution indirection and redirection techniques. They involved use of dynamic binding, swap buffers, reserved dummy memory, proxies, and others that utilize symbol mapping [40, pp. 472-475], [52]. Also, forms of memory reconstruction and binary rewriting techniques, often referred as dynamic patches, were distinctively generalizable [52].

Without considering the number of changes, the effect on update performance could be thought twofold. It is evident that the computational time requirement of the state transformation method halts the program execution for a known period. However, the time before the execution state becomes transformable is not necessarily deterministically reachable, nor are the old parts after the update [86, pp. 22-46], [86, pp. 53-54]. This problem was widely discussed among the works and addressed by many designs. A comprehensive and often referred theory for the problem was studied by Gupta and later by Gupta et al. and Stoyle [47,48,86].

Along with the update granularity, this appeared to be the key issues when update safety was explicitly studied or touched on by a discussion. Various later works adopted concepts of type-safety for addressing the determinism problems. It defines update safety mechanisms by limiting valid update operations so that the changes would not need to interfere with the old program in any way [57]. That is also why the update granularity correlates strongly with the update safety, as it can control the flexibility of the program mutability. Yet, as stated by Gupta and Hayden et al., the safety cannot be guaranteed to be solid for arbitrary or even for all limited update cases [57, pp. 1-5], [86, pp. 22-46].

Naturally, the approaches to the final fundamental problem, about how the update would be formed, is bound to the previous ones. Their design and methods would essentially depend on the done choices for update granularity, the methods for update timing, and especially the state transformation. The analysed works that suggested a practical model or an implementation for a

DSU system, typically included a customized approach for generating the update data. Of the reference studies, Miedes and Muñoz-Escoí's survey and the analysis of Mugarza et al. included comparable information on the topic.

The methods had distinctive differences, which lead to a cursory classification based on their related state transformation methods. They can be divided into ones involving programming languages with in-built DSU support, modified compilers, and binary or bytecode post-processing. The first two would specifically introduce additional information in the resulting program binary or bytecode to control the state transformation process when loaded by the DSU system on the target. The post-processing approach seemed common among dynamic patch forming. It utilizes static analysis techniques on compiled or interpreted programs to identify changes and prepare them in a format which the DSU system could receive.

Approaches to the fundamental problems resulted in further observations and deductions to identify a common architectural trait and a classification of DSU-applicable environments. It was clear, that at least two tightly coupled entities must have been defined for the design and implementation of a functional DSU system. In addition to the dynamic updating functionality on the target system, an infrastructure with the ability to produce the updates in applicable format must exist. Practically, a communications medium would be required between the two to transfer the update information.

Common architecture, especially on physically distributed systems such as WSN, was a client-server model where the target DSU system represented the client and server would prepare and provide the update. The information transferring for both seemed to have a natural tendency to rely on existing methods and infrastructure, such as internet- and inter-process protocols.

The environments, in terms of target hardware and software, for utilizing a DSU functionality had a predictable variety. They could be classified according to their level of abstraction from the hardware. To start with the obvious, hardware implemented DSU or reprogramming with redundant hardware were referred to

be the oldest and original methods. However, they remain a valid approach [58,63]. That would be followed by a so called bare-metal implementation, which manages the updating on firmware level, and either exists besides the main application program or provides a runtime environment for it. Further would be the DSU which rely on an operating system, of either an embedded or a conventional type, and exists as a kernel driver or process to update other application processes.

An additional implementation independent class could be defined for virtual machine runtime environments for both the DSU and the application. More specifically addressing virtual machine derivations, which would not necessarily require an operating system backend, but could also be implemented on bare-metal. Systems, such as Java virtual machine, which are known to have this property speaks for the classification.

Although, the domain appeared barely evident in the raw material, its existence was eventually supported by a virtual-machine centric approach [87] by Subramanian et al. Especially the Java virtual machine seemed extensively researched and referred in the overall material and reference studies. This was clearly indicated by Ahmed et al. in their tables 7 and 12 for representing their results for the most cited programming languages and approaches.

To summarize the current state from the conducted literature review and analysis, it could be concluded that DSU is not as extensively researched or utilized in firmware updating on low levels of resource-limited embedded systems as it is in system-level applications on operating system or virtual machine environments. It is worth noting, that the existing research and theory rarely made clear distinctions between applicable target domains, which would help in their evaluation. However, compact operating systems for embedded WSN nodes or bare-metal virtual-machines seemed to be the closest implementations to provide appropriate DSU functionality of interest.

The existing methods involved approaches, which eventually seemed to solve same problems regarding update safety and performance with varyingly

emphasized priorities. They posed mutually exclusive limitations which rendered a universally applicable approach non-existent, as was often concluded by the material and reference studies. In general, the concept of DSU is not fully disclosed for a formal consensus, although certain aspects recur throughout the requirements and definitions of implementations and research. Yet, the material did not include any clear hints on applying software centric DSU for firmware on embedded systems.

4 Design

This chapter walks through the applied methodology and process which took place for the implementable design development of the intended DSU system. The design's purpose was to prepare creation of specifications and descriptions for a possible implementation by establishing a theoretical system model and algorithms for its functional components. The subchapters break down the steps that were taken throughout the development phases of each individual problem in the order their solving was conducted.

Overall, the design process was driven by systematic problem-solving methods and observations from experimental research which induced the discovered theory with the initial idea of operating principles. The approach for orientating the design towards the stated objective began by identifying the required logical building blocks, the entities. This phase was conducted with a top-down method to start with the abstract goal-oriented information to understand the entity relationships. The relationships formed practical candidates for the system architecture where the details and more specific problems could be iterated towards a better insight. With a confident high-level model of the logic, the theoretical system functionalities were easier to be prepared for practice-based experiments with a sample target hardware.

In contrary to the previous phase, the experiments through implementation were conducted from target-oriented and detailed pieces towards larger collective functionalities. In other words, from bottom to top to identify inconveniencies in

the architecture earlier. The inconveniencies refer to actual errors rather than places for optimization, as the latter would be done preferably after the system works on all required levels to proof the concept. Yet, potential improvements were documented for further development as they were discovered.

Other benefits of the practical target-oriented experiments for the design were related to the identification of possible limitations and common factors of the target. The information streamlined the architectural design by binding together functionalities whose affiliations could be managed by logically meaningful system entities.

Although, several hardware could have been potential for the experimentation, an NXP LPCXpresso evaluation board with an LPC1549 MCU, based on ARM Cortex-M3 processor, was chosen for practical experimentation. The evaluation board provided many properties and functionalities which seemed suitable for experimenting with both the core- and auxiliary DSU functionalities. Therefore, they would enable possibility for comprehensive testing throughout and after the experimentation phase. Also, the vendor provided extensive development tools and documentation. For other practical matters, the author’s existing experience with the hardware and tools provide advantage to the design and experimentation process.

4.1 Architecture

To begin with a problem of conducting software based DSU on an embedded system, an initial and intuitive approach was to introduce an intermediary program underlaying the user defined firmware. An external entity was obviously required for transmitting the update data as the system was intended for remote- and peripheral applications. Therefore, similarly to other existing distributed systems, a client-server model was chosen for representing the relationship of the two entities. However, a dedicated model had to be designed for the client to meet the requirements for flexible applicability. The next client chapter will introduce the respective design process.

The system design began with an in-depth exploration of the client to identify and prioritize relevant functional problems that could be potential blockers for further work. Throughout the exploration, the server-side's design was gradually prepared to adapt to the choices made for the client. Especially to the updating method. Eventually, the identified interoperable requirements were outlined into notes for a DSU draft specification.

The later design of the client resulted in a requirement for the server-side to processing the firmware updates into a specific patch format. It was considered more convenient to manage the processing as an independent problem, outside the maintenance control. Therefore, an entity for a DSU patch generator was identified. Its operation was deemed as an extended software build phase, requiring more computationally capable hardware resources than the server or client.

A consideration for modularizing the update server, similarly to the approach of Felser et al., resulted in a decision of separating the patch generator into a dedicated remote entity. Fundamentally, the server could be deployed closer or embedded into a network of clients instead of the remote development host the patch generator would have to reside on.

The architecture recognized entities for a client composition of an abstract DSU logic bound to a variable target, a server for controlling the maintenance, and a patch generator for preparing the update data. Each of them operates on their respective deployment environmental level.

4.2 Client

Due to the functional requirements, the design had to consider varying targets having unpredictable hardware configurations and usage. It might not be possible or ideal to design a universal client software that could apply to all of them. Although, target specific standards for some central functionalities could

streamline the design. The problem was approached by separating it into two individually manageable modules and to promote their configurability.

The main problem boiled down in organizing the client components in a way that allows independent targets to conveniently adopt the DSU software and on the other hand, the DSU system to utilize the target hardware and application programming interface (API) for its operations. Both had to support module-wise shareable configurations, as different targets and applications might have requirements for specific hardware configurations or peripheral usage. The configurability posed various non-functional requirements for the design, some of which could not be readily disclosed or were subject to change.

Experiments with practical trials resulted in a dynamically defined client concept where the DSU functionality utilizes target hardware through a software interface which the target modules are expected to implement. This way a monolithic client is formed by including an implemented target module of interest as a submodule with a specified namespace into the DSU client hierarchy. The interface and configurations were designed to be exchanged between the modules by including cross-references to specified header file paths.

This decision can potentially limit the backwards compatibility of further structural changes to the client architecture. For now, it provides a manageable method but requires a clear specification which must be strictly followed by both modules. Hence, a target module can not only be expected to implement the interface, but it is also required to establish a specific file structure and fixed namespace for exposed configuration headers to comply with the inclusion scheme.

Although the modules could be considered mutually opaque to each other due to the abstraction intended by the DSU interface, some hardware-specific configurations were found to be required by the update algorithms prior to compilation. Such configuration of initially detached modules was discovered being easiest to achieve by coupling C-programming language preprocessor

definitions with the fixed inclusion scheme. The decision to relying heavily on compile-time configurations and preprocessor validation rather than usage of shared memory resulted in a smaller binary footprint and possibility for specifiable compile-time validations. Due to the structure being subject to such specifics, its scheme indicates a non-functional requirement.

Implementation-wise, the module development could benefit from separate version history, as the number of potential target implementations is not constant, nor immutable. The separation of the client module source repositories did not result in inconveniencies when implementing a new target as the developer can exclude the DSU-specific parts prior to release.

4.2.1 Updater

As the work revolved around the idea of conducting DSU for an MCU firmware, the actual method was the first problem to gain attention. To begin with it, an initial and intuitive approach was to introduce an intermediary program underlaying the user defined firmware. The functionality would utilize the existing flash IAP- and inter-device communication capabilities of a target.

The approach was iterated before the comprehensive research on the existing DSU methods, and no specific ones were considered for the algorithm. However, the context and intended usage with target research provided a practical scope for the theoretical exploration. For finding a functional method, update- and system safety along with performance were not concerned, as they can be addressed once the method can be deemed conceivable. Only the firmware data on flash memory was taken in account, leaving also stack management as a separate problem.

First candidate considered swap buffer for switching between firmware versions, also enabling further possibility for performing automatic firmware recovery offline. Also, the flash operations could have been managed consistently during execution context switches, similarly to operating system

processing. However, it would have likely consumed flash space for at least two full binary image versions. Therefore, it would have required impractical amount of flash memory to be reserved, limiting the applicable size of the initially small user firmware space.

Flash reconstruction emerged as an alternative method. It could potentially utilize the update data more efficiently resource-wise as less data would be required to be received and buffered. The method was based on a concept of utilizing the already present data by reading it in a specific order into a page-sized write buffer on stack which is then modified according to the changes in the new version of the firmware before being rewritten back.

The sample target provided synergetic properties to it as the flash supported relatively small erase- and write operations, which could be conducted a single page of 256 bytes at a time. Evidently, small-granularity updates can result in smaller input buffer requirements. In some updates the property can also become beneficial in reducing the number of required flash operations when compared to the effort required for cycling a full firmware image. On the contrary, small, or fragmented flash erasures and writes can naturally result in increased overhead caused by the reconstruction algorithm and input handling. Therefore, the algorithm could potentially benefit from a configurable update granularity, although strict target-specific flash limitations would apply.

The flash reconstruction method was researched by visually experimenting with the modeled logic on a Microsoft Excel spreadsheet. The steps and required components for the algorithm were identified by working out backwards multiple iterations of various states of a visually represented flash reconstruction. Three mandatory change operations under several update data related conditions were identified. They consist of performing byte-level substitutions, removals, and insertions for the flash in a specific order and operating direction. The condition definitions play a significant role in preventing the yet unreconstructed flash pages from being overwritten.

To mitigate information lost during flash reconstruction the change operations were discovered to require a specific order and processing direction starting either from the low or the high end of the affected firmware pages. The direction of reconstruction must respond to the resulting overall byte shift of the completely updated data, which can happen when information is being removed or inserted. Under certain conditions, all changes could not be conducted during a single write buffer's read-write cycle. The evident algorithm overhead of costly heuristic runtime data analysis for finding the appropriate operations had to be reduced. It was decided to be managed with specific update instructions in the update data, which could be generated on a more computationally capable patch generation host.

Therefore, the update data was ideal to be specified in a format to be usable by the other system entities. The data format went through multiple iterations of theoretical and practical exploration. It settled in a two-part structure consisting of a header and a payload. A dedicated header was considered convenient for the task by providing the functional information in fixed byte fields for integrity verification, intention validation, data pointers, and selecting a proper behavior for the maintenance. The payload would only include the binary changes in a form of update instructions and the actual byte data.

An equally important part of a fully functional DSU is to also update the stack to ensure consistent resumption of the execution through state transformation. There was fundamentally no problem if the firmware's main function or earlier stacked data, relative to current execution, was not modified. In the opposite case the stack would have required updating as well. This problem seemed to be a limiting factor, blocking the design until a possibility of already utilized reconstruction was considered again.

The required information could possibly be extracted from the compiled data during the patch generation process by identifying the known stack manipulating patterns in the binary instructions. Speculatively, stack reconstruction could provide a flexible but relatively slow method. The most

cumbersome problem was to adopt a mechanism for gaining insight of the currently stacked data reliably and without excessive interference to the user firmware development. So far, the methods for doing so during runtime, without a debugger, were found being limited to analyzing the CPU registers. The lack of knowing the taken paths of the execution could break the reconstruction's consistency quickly upon branching, as that information might not be tracked indefinitely in a feasible manner.

Perhaps, an internal finite state machine for tracking execution paths could be utilized by the reconstructor for deciding a suitable starting point from the stack. Yet, a type of update safety with such methods could not be reliably estimated as the update information can essentially be considered a non-deterministic input. On the other hand, the afterwards researched update points could provide reliability to some extent, as demonstrated by the existing DSU methods, but they might require additional expertise from the user and interfere intrusively with the firmware development. Therefore, while being important aspect of the system, the complex problem had to be excluded from the experiments due to its un-estimable research time requirements.

4.2.2 Process

Once the update method was theoretically eligible, the client design was continued by considering the overall update process on a high-level. The target research drew guidelines for the required steps and order. Experimentations with the individual mechanisms for IO, execution interruption, and resumption after completed update resulted in a natural set of individually manageable steps. They consisted of communication, execution interrupts, input processing, and resumption.

To initiate a maintenance session with the client, it naturally requires an external request sent by the server. The system would not benefit from an extensive communications protocol but rather a compact set of rules to describe behavior according to the received data at certain states of execution. Easiest model for

it was a finite state machine whose states included normal firmware execution, maintenance session, and a generic fault state.

Upon receiving data from the server, the client enters an interrupt service routine caused by an interrupt from its configured two-way IO interface. The raw input must be handled by writing it into a read buffer. For each input, the routine checks if enough bytes were received for them being processed further. Once positively evaluated, a blocking communications routine would be executed, otherwise the execution would return to the user firmware. The communications routine would have to verify the input data for integrity and then evaluate the intention of the header. With a specification- and state-wise valid header and payload, the requested maintenance operation could be acknowledged back to the server and proceeded.

Successfully initiated sessions would have to change a communication state accordingly if they were relevant for further communications. Such cases include multi-part updates which would disruptively affect to the flash content's consistency. They would affect to more bytes than what could be managed by a single flash reconstruction cycle. As it was designed for managing the changes at a write buffer granularity, the most safe, yet acceptable, method ensuring correct resumption was to halt the firmware execution until all data was successfully updated.

Upon successfully completed maintenance, other resumption modes were found practical besides resuming the user firmware, such as maintaining the halted execution or performing a software reset. Also, hardware power management was considered useful, although it likely requires a dedicated hardware support, which naturally could not be subject to non-functional requirements. To expand the thought, a reserved mode for invoking a user defined action was specified instead. Therefore, the resumption method would depend on the information of the conducted maintenance operation. Hence, the resumption mode was designed to be explicitly included in the header.

Without a disclosed stack updating method, the lacking capability would limit the updater to only expecting rather rudimentary changes excluding any reference to code that could affect the already stacked data. Otherwise, a dedicated phase for updating the execution state, a state transformer, would be required for updating the stack memory and possibly corresponding CPU registers as well.

For maintaining responsiveness in an event of failure, the execution was designed to be handed over to the communication process to inform the server about the system's status. It allows the client to expect an update session for a recovering firmware version. However, in such case, the execution state could end up being unknown, thus requiring the firmware being exchanged fully and execution to be restarted. Ideally, communication- or update related errors would be best handled by simply replying to the server with a resend request. This behavior indicates that the IO must support and be able to initiate two-way communication. It led to a description of a non-functional requirement for the IO capabilities and to a specific format being designed for the returned data, including acknowledgements and error codes. Natural format for both replies was to define them in a form of return code for representing the client status.

4.2.3 Target

The target module's design introduced a few non-functional requirements for interfacing the DSU functionalities. They reflect to the decisions made for the DSU process, which expect some specific architectural-, flash programming-, and IO-related properties from the target. The intention was to maintain the hardware adaptable by the party who implements the target module.

As discussed earlier, configurability of individual target resources was identified being practical for providing flexibility to the user firmware development. However, it could not be considered a requirement for the target module if the DSU interface could be implemented sufficiently. The information of the selected peripheral interfaces, their possible configurations, data verification

method, and the memory limits, however, are essential to the server and patch generation.

The server-side modules were considered being responsible for maintaining the information related to their operability. For minimizing static data on the deployed client, its configurations were designed to ideally be shared prior to compilation. Not only did it make the multi-module client building convenient, but also made it possible to configure the server and patch generator builds at the same time. Although considering flexibility, some applications could benefit from the information being preserved on the client for later access in case client configurations shall be identified. The matter was left open to be specified as a configurable future feature.

For complying with the specified digital byte-level communications, the target must be able to dedicate a peripheral IO device with configurable interrupt handling, and to declare a method for performing data verification for input and output. The latter was required to ensure data integrity during communication to address pathological bitflip patterns, that could be characteristic to the system’s deployment environment.

While most of the resource configurations can be considered opaque to updater logic, exceptions would be those functionalities which can have an off-board alternative to be implemented by other DSU modules. Such would be the case for a user-defined data verification method if the target did not provide a dedicated hardware for it. A decision was made to leave the method generally specified and require the DSU interface to manage it abstractly. However, the method with its configuration remained necessary information to be shared between the client and the server.

For various target resources being externally referenceable by the DSU interface, such as devices interrupt handlers and peripheral devices, the ARM Common Microcontroller Software Interface Standard namespace was considered practical. Yet, experiments on a generic method for aliasing vendor specific namespace with the module configuration scheme was found possible.

Although, the introduced complexity could lay the software interface's implementation more open to errors, it could promote flexibility further if both methods were being configurable.

The IAP usage was found posing restrictions which apply to both the client and the firmware update building process. The deployed DSU client with the included target API would naturally exist at the lower addresses of the flash memory. Despite the resulting size of the DSU client, the IAP operations likely affect to the user firmware placement. Such was the case with the explored target device. In addition to a required RAM offset and stack usage reservation, it required the first flash sector to be left un-erased because the IAP functions would be called from there [20, pp. 554-555]. This implied that the compiled firmware update must not be expected to start from its default flash address, but it was required to be linked to a specific flash offset to mitigate erroneous object references.

The requirements for the linking also resulted in a realization of the firmware having to correctly reference the client's already present target API. A firmware providing the targe API redundantly did not seem ideal and was not explored further to verify whether it was practical alternative. In general, the requirements for the compiled object file linkage remained undisclosed.

During any interrupt service routine, the system must prevent further application- or communication interrupts from occurring to maintain operational consistency. Another fatal risk in addition to such inconsistencies are incomplete CPU operations during full-word memory transactions, as pointed by Ganssle [26, p. 325]. Although, the targeted ARM Cortex-M3 processors can operate with 4-byte data encoding and alignment, their Thumb instruction set architecture and Thumb-2 enhancement can utilize variable length encoding [88]. According to Ganssle, for certain 4-byte instructions the CPU can require two cycles during which enabled interrupts can fatally corrupt the incomplete memory transaction. Hence, the target must be expected to implement the

DSU-related interrupts with appropriate priority and to disable further interrupt generations upon the DSU process to mitigate these problems.

Considering the use cases, only three mandatory resumption modes were identified, but fourth was considered optional. Them being, firmware resumption, maintaining the halted execution, firmware restart and a reserved user-defined mode, such as for the client's power management. For the first, if the next executed program address would remain unchanged, a function call with standard C semantics was found being sufficient. Otherwise, if the stack was updated, an explicitly instructed jump into an updated position with accordingly manipulated CPU registers was speculated. Various methods for conducting the jump were researched, but not experimented with due to the lack of a proven state transformer method.

Software restarts, either directly or after a full user firmware exchange, could also utilize a function call to a fixed firmware restart address. This would be in addition to clearing the related stack entries and updating CPU registers accordingly. Although, the targets might provide built-in functionality for full software restart, it was not deemed worthy for a dedicated requirement.

4.3 Server and patch generation

Design for the server-side modules including the server and patch generator entities were left undisclosed due to limited resources for conducting sufficient research and exploration. However, fundamentals and various aspects were identified from the client's requirements through deduction. Some of them had to be considered further before being able to continue with the client design.

The fundamental idea of the server was to act as an interface to one or more clients and to provide an automatic update dispatching service. It was designed to be responsible for maintaining client-related end-to-end communications, configurations, build options, and version history of the deployed firmware. Generally, it can be considered as a function which would consume

maintenance options, and the patch data if it was provided, as an input and produce formatted byte stream as output. The formatting was designed to be responsible for header formation, along with the verification checksum generation.

A decision was made to leave the communications protocols, medium and their security unaddressed and open for user implementations. It might result in a trade-off between DSU coverage over the embedded system and the hardware requirements. Also, it was not found necessary for the interface to be bi-directionally active, although, unexpected error messages would be ideal to be responded to. Such as, with an automated recovery update.

Various configurations for the server-side model emerged as different use cases were considered. An operating environment for buffering and dispatching the update data does not have to reside on conventional host. In some systems, such as vehicle and control environments, it could be feasible for the service to be hosted within the target for controlling other subsystems locally.

In such case, the server-side model should also establish a separated entity for maintaining the information required by the patch generator on development host. Due to the potential placement choices, the server would be required to support modular implementations for the possibly separated entities. In any configuration, the most important aspect was identified to be the adoption of a user-defined communications medium. Remote connections might not always be wired, such as for OTA systems. Depending on the applied module placement, the communication can require additional hardware or software, which may not necessarily be covered by the DSU. The exact model for defining the server components remained subject to architectural changes, which required further research.

The patch generation was designed to be operated as an individual program on a development host. The operation for forming the patches from binary differences requires at least the latest deployed version as a reference, which is why the version history must be maintained. The practical storage model for the

information was considered being best left user definable if the binary format was explicitly specified.

The firmware binaries were expected to go through a conventional firmware build process without unnecessary interference of the patch generator, the binary differentiation could be considered a post-process operation. Hence, the resulting patches were considered being subject to changes to the firmware build options and toolchain version. Such changes could result in update failure or unknown client behaviour as the reconstruction algorithms would expect a specific composition of the new bytes, bit- and alignment-wise. To mitigate the problem, a firmware update would be expected to be tested on a twin DSU target system before dispatching to ensure its consistent semantics and behaviour.

Generally, the workflow from update development to its deployment was identified to include a few distinct phases. A development phase for implementing the application changes in a firmware source code and building it into a binary image. It would be continued by a patch generation phase, after which the application changes and patch data would ideally be tested on a twin target system. The testing would naturally require a setup of the production server and a twin client. Once verified, a maintenance phase with the patch data can be initiated with the production client to begin an update according to specified communications protocol.

With the reasoned fundamentals and general design, many aspects remained undisclosed or required further research before being able to be explored further in practice. Such variety of approaches also resulted in various details which were not feasible to be described only on a general level.

5 Conclusion

This chapter represents the deductions of the compiled information which was gained throughout the work. It will orderly conclude the central achievements

and take aways of the embedded DSU research, its general resolution, challenges, and further thoughts, along with retrospective reflection about the experienced insights.

The work was based on a research question for finding an applicable method for conducting a dynamic firmware update on an embedded system. The approach for solving the question was projected in an objective for designing such dynamic update method within a scope of embedded system firmware.

Actions for the objective consisted of three main phases. First, prior to researching, an initial hypothesis for the dynamic firmware updating on an embedded system was established to give chance for novelty through intuitive and unbiased thinking. It was followed by current state analysis in a form of a progressive literature review on the DSU field of research to identify topic-specific theoretical and technical information. The results were used to refine the initial hypothesis and research methodology. Also, finding any existing works was considered beneficial for establishing a comparative evaluation for this work against a known reference. In the final phase, all the information was used in an experimental exploration for finding a DSU method defined in the objective.

The research and experimental exploration resulted in many miscellaneous and unrepresented collection of notes, bibliography, and development artifacts being produced, of which the latter included work for testing and solving the design-related algorithm-, architectural-, inter-module information transfer problems.

As a conclusive result, the essential aspects and theory of a dynamic firmware update on an embedded system could be deemed sufficiently identified to the extent where the research question can be answered. Although, future amendments regarding evaluation of the requirements fulfillment and practical implementation were left still pending. On a logical yet theoretical level, the researched design suggests plausible approach to conducting dynamic firmware updating on an embedded system.

As concluded in the current state analysis, the specific subject of applying DSU methods on embedded system firmware it is rather unrepresented in its field of research, if at all. At the time of writing, designing such DSU system for a rather uncharted application domain resulted in gaining promoting insight, which can be leveraged by further works to draw consensus on its feasibility.

Valuable knowledge was also gained in other parts of the work, and not only on the research question's topic but on meta level as well. The selected research methods appeared effective for managing and solving the kind of unforeseeable issues encountered in the work. Also, requirements for working on the topic were comprehensively identified and noted in a way that can aid developing the academic institution's contribution and supervision for further works on the field.

Originally, the research question projected a two-fold objective for the theoretical design and its demonstrative implementation for evaluating the resulting feasibility against the research question. However, as the research advanced further and practical explorations were involved in the methodology, the growth of beforehand unknown complexity resulted in explosively increasing time requirements for the work. Therefore, the objectives had to be adjusted to match a more feasible scope of a bachelor's thesis.

The change resulted in the state transfer method's design being left for future work, which was perhaps the most time consuming, but important, part of the updater. Due to the available scope, walkthroughs of some core functionality designs were best being normalized to make them match the general level of reporting and structural consistency. Architecturally, the DSU system's server-side had to be limited to a rather superficial design stating only the generally deductible aspects.

Also, the user firmware was not distinctively discussed in the design due to the lack of a working implementation. Only few aspects of the limitations and capabilities could be deducted or speculated from the client and patch generation designs, leaving further analysis on an unreliable premise. Hence, the overall impact of the unexpected complexity affected the design process

and reporting dramatically as many important or resulting topics and artifacts had to be limited or excluded.

With hindsight, it can be concluded that the work's requirements were too vast to fit in the ideal scope of a thesis of this level. Perhaps just designing and implementing individual parts of the update system would have been efficiency-wise more feasible to produce more technically dense results. Although, it could be speculated whether the scope or objective should have been limited more extensively immediately when the first indications of research uncertainties were confronted.

Despite the adversities, the work can be consistently continued. For evaluating the proposed DSU design and its practical functionality, it would require a measurable implementation, which preferably was based on a complete design. Therefore, solving the state transfer method would be crucial to make the implementation more practical by increasing the system's update case coverage to match intended and realistic use. Hence, next steps would bring the design to a state where the implementation could be conducted. Continuation for achieving the researched dynamic firmware update system on an embedded system is desirable.

Overall, the work laid a valuable basis for its continuation to solve the remaining issues regarding the system components and their operating methods, which are required for initiating the implementation. So far, all the identified aspects of a dynamic firmware update on an embedded system together with the explorations orientated the design's potential use cases towards practical real-life fields which hinted demand for such functionality. During the research and design, improvements in subsystem updating of various vehicle applications was found having an emerging discussion, especially in automotive industry articles.

Although, the originally considered main target of sensor systems or networks of them have seemingly settled on robust and existing operating system based DSU mechanisms, could they speculatively improve further with a lower-level

application management. Also, DSU methods could be re-purpose for flash memory wear levelling in embedded software development. On the other hand, predictions for the work’s practical utilizations at this point are merely hopeful speculations until a complete implementation can be prepared for further testing.

References

1 Abbott D. Linux for Embedded and Real-time Applications, 4th Edition, Chapter 1. The embedded and real-time space [Internet]. 4th ed. Newnes; 2018 [cited 2022 May 23]. Available from: https://learning.oreilly.com/library/view/-/9780128112786/xhtml/chp001.xhtml

2 National Research Council, Division on Engineering and Physical Sciences, Computer Science and Telecommunications Board, Committee on Networked Systems of Embedded Computers, National Academy of Sciences. Embedded, Everywhere : A Research Agenda for Networked Systems of Embedded Computers [Internet]. Washington, D.C., UNITED STATES: National Academies Press; 2001. Available from: http://ebookcentral.proquest.com/lib/metropolia-ebooks/detail.action?docID=3375453

3 Noergaard T. Embedded Systems Architecture : A Comprehensive Guide for Engineers and Programmers. Burlington: Elsevier/Newnes cop.; 2005.

4 Patterson DA, Hennessy JL. Computer Organization and Design, Fifth Edition: The Hardware/Software Interface. 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2014.

5 Heath S. Embedded Systems Design [Internet]. Oxford, UNITED KINGDOM: Elsevier Science & Technology; 2002. Available from: http://ebookcentral.proquest.com/lib/metropolia-ebooks/detail.action?docID=294113

6 Lutkevich B. What is an Embedded System? [Internet]. IoT Agenda. [cited 2022 May 23]. Available from: https://www.techtarget.com/iotagenda/definition/embedded-system

7 Stallings W. Computer Organization and Architecture. Pearson; 2016.

8 Yanbing Li, M. Potkonjak, W. Wolf. Real-time operating systems for embedded computing. In: Proceedings International Conference on Computer Design VLSI in Computers and Processors. 1997. p. 388–92.

9 Ganssle J, Perrin B, Noergaard T, Eady F, Edwards L, Katz DJ, et al. Embedded Hardware: Know It All [Internet]. Oxford, UNITED STATES: Elsevier Science & Technology; 2007. Available from: http://ebookcentral.proquest.com/lib/metropolia-ebooks/detail.action?docID=534852

10 Lacamera D. Embedded Systems Architecture: Embedded Systems - A pragmatic Approach [Internet]. Packt Publishing; 2018. Available from: https://learning.oreilly.com/library/view/-/9781788832502/713537a0-ebf8-4ce6-818a-57b150fc1507.xhtml

11 Panda PR, Dutt N, Nicolau A. On-Chip vs. Off-Chip Memory: Utilizing Scratch-Pad Memory. In: Panda PR, Dutt N, Nicolau A, editors. Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration [Internet]. Boston, MA: Springer US; 1999. p. 117–36. Available from: https://doi.org/10.1007/978-1-4615-5107-2_5

12 Ledin J. Modern Computer Architecture and Organization [Internet]. Packt Publishing; 2020 [cited 2022 May 27]. Available from: https://learning.oreilly.com/library/view/modern-computer-architecture/9781838984397/

13 Benavides T, Treon J, Hulbert J, Chang W. The Implementation of a Hybrid-Execute-In-Place Architecture to Reduce the Embedded System Memory Footprint and Minimize Boot Time. In: 2007 IEEE International Conference on Information Reuse and Integration. 2007. p. 473–9.

14 Garcia D, Chien J. THE NEED FOR EXECUTE-IN-PLACE (XIP). [Internet]. NXP Semiconductors; 2021 [cited 2022 May 31]. Available from: https://www.nxp.com/docs/en/white-paper/SMART-VEHICLES-XIP-WP.pdf

15 Richter D. Flash Memories [Internet]. Dordrecht: Springer Netherlands; 2014 [cited 2022 Apr 20]. (Springer Series in Advanced Microelectronics; vol. 40). Available from: http://link.springer.com/10.1007/978-94-007-6082-0

16 Lutkevich B. What is NOR Flash Memory and How is it Different from NAND? [Internet]. SearchStorage. [cited 2022 Jun 1]. Available from: https://www.techtarget.com/searchstorage/definition/NOR-flash-memory

17 Vignesh R. An Introduction to SPI-NOR Subsystem [Internet]. The Linux Foundation; 2022 [cited 2022 Jun 1]. Available from: http://events17.linuxfoundation.org/sites/events/files/slides/An Introduction to SPI-NOR Subsystem - v3_0.pdf

18 Labrosse JJ. Embedded systems building blocks. 2nd ed. Lawrence, Kansas: CMP Books; 2000.

19 Lacamera D. Embedded Systems Architecture: General-Purpose Peripherals [Internet]. Packt Publishing; 2018. Available from: https://learning.oreilly.com/library/view/-/9781788832502/713537a0-ebf8-4ce6-818a-57b150fc1507.xhtml

20 UM10736 LPC15xx User manual Rev. 1.2 [Internet]. NXP Semiconductors; 2017 [cited 2022 Jun 9]. Available from: https://www.nxp.com

21 Tan CJ, Mohamad-Saleh J, Zain KAM, Aziz ZAAbd. Review on Firmware. In: Proceedings of the International Conference on Imaging, Signal Processing and Communication [Internet]. New York, NY, USA:

Association for Computing Machinery; 2017. p. 186–90. (ICISPC 2017). Available from: https://doi.org/10.1145/3132300.3132337

22 SO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes. IEEE STD 12207-2008. 2008 Jan;1–138.

23 Zandberg K, Schleiser K, Acosta F, Tschofenig H, Baccelli E. Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check. IEEE Access. 2019;7:71907–20.

24 Earl B. Bootloading Basics [Internet]. 2021 [cited 2022 Jun 11]. Available from: https://cdn-learn.adafruit.com/downloads/pdf/bootloader-basics.pdf

25 Barr M. Programming embedded systems in C and C++. Sebastopol, CA: O'Reilly; 1999. 174 p. (O'Reilly Series).

26 Ganssle J. The Firmware Handbook [Internet]. [cited 2022 Jun 9]. Available from: https://learning.oreilly.com/library/view/the-firmware-handbook/9780750676069

27 Intel Hexadecimal Object File Format Specification Revision A, 1/6/88 [Internet]. 1988 [cited 2022 Jun 11]. Available from: http://www.interlog.com/~speff/usefulinfo/Hexfrmt.pdf

28 Intel HEX File Format [Internet]. [cited 2022 Jun 11]. Available from: https://developer.arm.com/documentation/ka003292/latest

29 Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification [Internet]. TIS Committee; 1995 [cited 2022 Jun 11]. Available from: https://refspecs.linuxfoundation.org/elf/elf.pdf

30 Di Jasio L. Programming 32-bit microcontrollers in C: exploring the PIC32. 1st ed. Newnes; 2008. (Embedded Technology Series).

31 Catsoulis J. Designing Embedded Hardware, 2nd Edition [Internet]. [cited 2022 Jun 17]. Available from: https://learning.oreilly.com/library/view/-/0596007558/ch01s02.html

32 GENERAL: Difference Between ISP and IAP [Internet]. [cited 2022 Jun 17]. Available from: https://developer.arm.com/documentation/ka002966/latest

33 IEEE Standard Test Access Port and Boundary Scan Architecture. IEEE Std 11491-2001. 2001 Jul;1–212.

34 ARM® Debug Interface v5 Architecture Specification [Internet]. ARM Ltd; 2006 [cited 2022 Jun 17]. Available from: https://documentation-service.arm.com/static/5f900a61f86e16515cdc0610?token=

35 Jurkovic G, Sruk V. Remote firmware update for constrained embedded systems. In: 2014 37th International Convention on Information and

Communication Technology, Electronics and Microelectronics (MIPRO). 2014. p. 1019–23.

36 Jaouhari SE, Bouvet E. Secure firmware Over-The-Air updates for IoT: Survey, challenges, and discussions. Internet Things. 2022;18:100508.

37 Gu T, Zhao Z, Ma X, Xu C, Cao C, Lü J. Improving Reliability of Dynamic Software Updating Using Runtime Recovery. In: 2016 23rd Asia-Pacific Software Engineering Conference (APSEC). 2016. p. 257–64.

38 Wang KC. Embedded and Real-Time Operating Systems [Internet]. Cham: Springer International Publishing; 2017 [cited 2022 May 23]. Available from: http://link.springer.com/10.1007/978-3-319-51517-5

39 Shen BY, Chiang ML. A Server-Side Pre-linking Mechanism for Updating Embedded Clients Dynamically. In: Kuo TW, Sha E, Guo M, Yang LT, Shao Z, editors. Embedded and Ubiquitous Computing. Berlin, Heidelberg: Springer Berlin Heidelberg; 2007. p. 146–57.

40 Ahmed BH, Lee SP, Su MT, Zakari A. Dynamic software updating: a systematic mapping study. IET Softw. 2020 Oct;14(5):468–81.

41 Felser M, Kapitza R, Kleinöder J, Schröder-Preikschat W. Dynamic Software Update of Resource-Constrained Distributed Embedded Systems. In: Rettberg A, Zanella MC, Dömer R, Gerstlauer A, Rammig FJ, editors. Embedded System Design: Topics, Techniques and Trends [Internet]. Boston, MA: Springer US; 2007 [cited 2022 May 20]. p. 387–400. (IFIP – The International Federation for Information Processing; vol. 231). Available from: http://link.springer.com/10.1007/978-0-387-72258-0_33

42 Fabry, R. S. How to design a system in which modules can be changed on the fly. In Proceedings of the 2nd international conference on Software engineering (ICSE ’76). In: How to design a system in which modules can be changed on the fly [Internet]. San Francisco, California, USA: IEEE Computer Society Press, Washington, DC, USA; 1976. p. 470–6. (ICSE ’76). Available from: https://dl.acm.org/doi/10.5555/800253.807720

43 Goullon H, Isle R, Lohr KP. Dynamic Restructuring in an Experimental Operating System. IEEE Trans Softw Eng. 1978 Jul;SE-4(4):298–307.

44 Cook RP, Lee I. DYMOS: a dynamic modification system. ACM SIGPLAN Not. 1983 Aug;18(8):201–2.

45 Smith EK, Hicks M, Foster JS. Towards standardized benchmarks for Dynamic Software Updating systems. In: 2012 4th International Workshop on Hot Topics in Software Upgrades (HotSWUp). 2012. p. 11–5.

46 Bierman G, Hicks M, Sewell P, Stoyle G. Formalizing Dynamic Software Updating. Proc Second Int Workshop Unanticipated Softw Evol USE. 2003 Jun;

47 Gareth Paul Stoyle. A Theory of Dynamic Software Updates [Internet] [Degree of Doctor in Philosophy]. [Hughes Hall]: University of Cambridge; Available from: https://www.cl.cam.ac.uk/~pes20/GarethStoyle_thesis.pdf

48 Gupta D, Jalote P, Barua G. A formal framework for on-line software version change. IEEE Trans Softw Eng. 1996 Feb;22(2):120–31.

49 Frieder O, Segal ME. On dynamically updating a computer program: From concept to prototype. J Syst Softw. 1991 Feb;14(2):111–28.

50 Miedes E, Mu FD. Dynamic Software Update. 2012.

51 Hayden CM, Smith EK, Hicks M, Foster JS. State transfer for clear and efficient runtime updates. In: 2011 IEEE 27th International Conference on Data Engineering Workshops. 2011. p. 179–84.

52 Miedes E, Muñoz-Escoí F. A Survey about Dynamic Software Updating [Internet]. Technical Report ITI-SIDI-2012/003; 2012. Available from: https://www.researchgate.net/publication/267369142_A_Survey_about_Dynamic_Software_Updating

53 Dorst K, Cross N. Creativity in the design process: co-evolution of problem–solution. Des Stud. 2001 Sep;22(5):425–37.

54 Oivo M, Kuvaja P, Pulli P, Similä J. Software Engineering Research Strategy: Combining Experimental and Explorative Research (EER). In: Bomarius F, Iida H, editors. Product Focused Software Process Improvement [Internet]. Berlin, Heidelberg: Springer Berlin Heidelberg; 2004 [cited 2022 May 2]. p. 302–17. (Kanade T, Kittler J, Kleinberg JM, Mattern F, Mitchell JC, Nierstrasz O, et al., editors. Lecture Notes in Computer Science; vol. 3009). Available from: http://link.springer.com/10.1007/978-3-540-24659-6_22

55 Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, Anders Wesslén. Experimentation in Software Engineering - An Introduction [Internet]. 1st ed. Springer New York, NY; 2002 [cited 2022 May 16]. 204 p. (International Series in Software Engineering; vol. 2002). Available from: https://link.springer.com/book/10.1007/978-3-642-29044-2

56 Voyant Tools [Internet]. Voyant Tools; 2022 [cited 2022 May 19]. Available from: https://github.com/voyanttools/Voyant

57 Hayden CM, Smith EK, Hardisty EA, Hicks M, Foster JS. Evaluating Dynamic Software Update Safety Using Systematic Testing. IEEE Trans Softw Eng. 2012 Nov;38(6):1340–54.

58 Hicks M, Nettles S. Dynamic Software Updating. ACM Trans Program Lang Syst. 2005 Nov;27(6):1049–96.

59 Buck, Bryan and Hollingsworth, Jeffrey K. International Journal of High Performance Computing Applications. API Runtime Code Patching. 2000 Nov 1;14(4):317–29.

60 Protecting Linux systems with Oracle Ksplice zero-downtime updates [Internet]. Oracle; 2022. Available from https://www.oracle.com/a/ocom/docs/linux/ksplice-datasheet-487388.pdf

61 Holmbacka S, Lund W, Lafond S, Lilius J. Lightweight Framework for Runtime Updating of C-Based Software in Embedded Systems. In: 5th Workshop on Hot Topics in Software Upgrades (HotSWUp 13) [Internet]. San Jose, CA: USENIX Association; 2013. Available from: https://www.usenix.org/conference/hotswup13/workshop-program/presentation/holmbacka

62 Finkbeiner B, Klein F, Metzger N. Live synthesis. Innov Syst Softw Eng [Internet]. 2022 Mar 31; Available from: https://doi.org/10.1007/s11334-022-00447-5

63 Mugarza I, Parra J, Jacob E. Analysis of existing dynamic software updating techniques for safe and secure industrial control systems. Int J Saf Secur Eng. 2018;8(1):121–31.

64 Ilvonen V, Ihantola P, Mikkonen T. Dynamic Software Updating Techniques in Practice and Educator's Guides: A Review. In: 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET). 2016. p. 86–90.

65 Seifzadeh H, Kazem AAP, Kargahi M, Movaghar A. A Method for Dynamic Software Updating in Real-Time Systems. In: 2009 Eighth IEEE/ACIS International Conference on Computer and Information Science. 2009. p. 34–8.

66 Hayden CM, Saur K, Smith EK, Hicks M, Foster JS. Kitsune: Efficient, General-Purpose Dynamic Software Updating for C. ACM Trans Program Lang Syst [Internet]. 2014 Oct;36(4). Available from: https://doi.org/10.1145/2629460

67 Wahler M, Richter S, Oriol M. Dynamic Software Updates for Real-Time Systems. In: Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades, HotSWUp '09. 2009.

68 Makris K, Bazzi RA. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In San Diego; 2009. p. 14.

69 Niesler C, Surminski S, Davi L. HERA: Hotpatching of Embedded Real-time Applications. In: NDSS. 2021.

70 Hicks M. Dynamic Software Updating: Introduction and Foundation [Internet]. PowerPoint slides presented at: Oregon Summer School 2006; 2006 [cited 2022 Jul 20]. Available from:

https://www.cs.uoregon.edu/research/summerschool/summer06/lectures/oregon-dsu.pdf

71 Neamtiu I, Hicks M, Stoyle G, Oriol M. Practical Dynamic Software Updating for C [Internet]. Association for Computing Machinery; 2006. Available from: https://doi.org/10.1145/1133981.1133991

72 Hayden CM, Magill S, Hicks M, Foster N, Foster JS. Specifying and Verifying the Correctness of Dynamic Software Updates. In: Joshi R, Müller P, Podelski A, editors. Verified Software: Theories, Tools, Experiments. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012. p. 278–93.

73 Panzica La Manna V. Dynamic Software Update for Component-Based Distributed Systems. In: Proceedings of the 16th International Workshop on Component-Oriented Programming [Internet]. New York, NY, USA: Association for Computing Machinery; 2011. p. 1–8. (WCOP '11). Available from: https://doi.org/10.1145/2000292.2000294

74 Liu J, Tong W. A Framework for Dynamic Updating of Service Pack in the Internet of Things. In: 2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing. 2011. p. 33–42.

75 An S, Ma X, Cao C, Yu P, Xu C. An Event-Based Formal Framework for Dynamic Software Update. In: 2015 IEEE International Conference on Software Quality, Reliability and Security. 2015. p. 173–82.

76 Anderson G, Rathke J. Dynamic Software Update for Message Passing Programs. In: Jhala R, Igarashi A, editors. Programming Languages and Systems. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012. p. 207–22.

77 Lv W, Zuo X, Wang L. Dynamic Software Updating for Onboard Software. In: 2012 Second International Conference on Intelligent System Design and Engineering Application. 2012. p. 251–3.

78 Mugarza I, Amurrio A, Azketa E, Jacob E. Dynamic Software Updates to Enhance Security and Privacy in High Availability Energy Management Applications in Smart Cities. IEEE Access. 2019;7:42269–79.

79 Brown S, Sreenan CJ. Software Updating in Wireless Sensor Networks: A Survey and Lacunae. J Sens Actuator Netw. 2013;2(4):717–60.

80 Galos M, Mieyeville F, Navarro D. Dynamic reconfiguration inwireless Sensor Networks. In: 2010 17th IEEE International Conference on Electronics, Circuits and Systems. 2010. p. 918–21.

81 Das ML, Joshi A. Dynamic Program Update in Wireless Sensor Networks Using Orthogonality Principle. IEEE Commun Lett. 2008;12(6):471–3.

82 Levis P, Culler D. Maté: A Tiny Virtual Machine for Sensor Networks. SIGPLAN Not. 2002 Oct;37(10):85–95

83 Munawar W, Alizai MH, Landsiedel O, Wehrle K. Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks. In: 2010 IEEE International Conference on Communications. 2010. p. 1–6.

84 Chi TY, Wang WC, Kuo SY. uFlow: Dynamic Software Updating in Wireless Sensor Networks. In: Hsu CH, Yang LT, Ma J, Zhu C, editors. Ubiquitous Intelligence and Computing [Internet]. Berlin, Heidelberg: Springer Berlin Heidelberg; 2011 [cited 2022 May 20]. p. 405–19. (Lecture Notes in Computer Science; vol. 6905). Available from: http://link.springer.com/10.1007/978-3-642-23641-9_33

85 Han CC, Kumar R, Shea R, Kohler E, Srivastava M. A dynamic operating system for sensor nodes. In: Proceedings of the 3rd international conference on Mobile systems, applications, and services - MobiSys '05 [Internet]. Seattle, Washington: ACM Press; 2005 [cited 2022 May 20]. p. 163. Available from: http://portal.acm.org/citation.cfm?doid=1067170.1067188

86 Gupta D. On-Line Software Version Change [Internet] [PhD Thesis]. [Kanpur, India]: Indian Institute of Technology, Kanpur; 1994. Available from: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.3064&rep=rep1&type=pdf

87 Subramanian S, Hicks M, McKinley KS. Dynamic Software Updates: A VM-Centric Approach. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation [Internet]. New York, NY, USA: Association for Computing Machinery; 2009. p. 1–12. (PLDI '09). Available from: https://doi.org/10.1145/1542476.1542478

88 Cortex-M3 Devices Generic User Guide ARM Rev. A Version 1.0 [Internet]. ARM Ltd; 2010 [cited 2022 Jun 11]. Available from: https://developer.arm.com/documentation/dui0552/a/