Michael Astl, BSc.

Evaluation and Implementation of Reliable Over-The-Air Updates in Lossy, Low Bandwidth Sensor Networks

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger
Dipl.-Ing. Dr. techn. Tobias Rauter

Institute of Technical Informatics

Graz, May 2019

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present masters thesis.

....
Date Signature

Kurzfassung

Firmwareupdates werden immer wichtiger für Internet of Things (IoT) Applikationen. Aufgrund der beschränkten Energieressourcen und der limitierten Bandbreite der Drahtlosverbindung ist es notwendig, die Menge an gesendeten Daten zur Durchführung von Firmwareupdates zu reduzieren. Die Menge an Änderungen im Code ist üblicherweise gering, bezogen auf die Firmwaregröße. Daher kann die Datenmenge deutlich verringert werden wenn man nur die Änderungen im Code überträgt.

Das Ziel dieser Arbeit ist die Entwicklung einer effizienten und zuverlässigen Over-The-Air-Updatelösung, welche in Umgebungen mit beschränkten Ressourcen integriert werden kann. Es existieren bereits zahlreiche Algorithmen zur effizienten Kodierung von Änderungen der Firmware in ein Delta-Image. Die erreichbare Kompressionsrate solcher Delta-Algorithmen ist jedoch nicht ausreichend, da kleine Änderungen im Code meistens große Änderungen in der generierten Firmware verursachen. Im vorgestellten Konzept wird ein Ansatz zur Erhöhung der Ähnlichkeit verschiedener Firmwareversionen präsentiert. Erreicht wird dies durch eine Reduktion der Verschiebungen von Programmsektionen innerhalb der Firmware, welche durch Codeänderungen ausgelöst werden. Ein weiteres Problem bei Firmwareupdates in IoT Umgebungen sind die limitierten Ressourcen, welche am Sensor verfügbar sind. Dieser muss in der Lage sein sich selbstständig zu aktualisieren, ohne die Verwendung von zusätzlicher Hardware. Des Weiteren dürfen unvorhergesehene Ereignisse den Sensor nicht in einen undefinierten Zustand bringen. Diese Arbeit präsentiert ein Konzept, welches sicherstellt, dass am Sensor zu jedem Zeitpunkt zumindest eine valide Firmware verfügbar ist.

Die Evaluierung im Zuge dieser Arbeit wurde mit Firmware durchgeführt, welche in einer produktiven IoT Umgebung im Einsatz ist. Das präsentierte Konzept zur Erhöhung der Ähnlichkeit verschiedener Firmwareversionen verbessert die Kompressionsrate existierender Delta-Algorithmen um 88%. Das präsentierte Updatesystem stellt sicher, dass Endgeräte durch Fehlerfälle nicht unbenutzbar werden.

Abstract

Firmware updates are becoming more and more important in Internet of Things environments. Due to energy constraints and low-bandwidth wireless links, it is necessary to reduce the amount of transmitted data required to perform firmware updates. Since changes are usually small in terms of code size, transmitting only the changed parts of the firmware reduces the amount of data significantly.

The goal of this thesis is to develop an efficient and reliable Over-The-Air update solution that can be integrated into resource-constrained environments. There are already numerous algorithms that efficiently encode the changed parts of a firmware into a delta image. Nevertheless, the achievable compression rate with these delta algorithms is not sufficient for real world scenarios, because small changes in code often cause big changes in the generated firmware image. This thesis presents an approach that increases the similarity of different firmware versions by mitigating the effects of relocated program sections due to code changes. Despite the high efficiency requirements, the available resources of end devices are usually very limited. Sensors must be capable of updating themselves without any additional hardware. Furthermore, unexpected events should not cause the end devices to remain in an undefined state. This thesis also presents a concept for processing updates that ensures the presence of at least one valid firmware on the sensors at any point during operation.

In this thesis, an evaluation was done using firmware that is used in real environments. The presented similarity improvements concept increases the average compression rate of existing delta algorithms by 88%. The presented update system additionally ensures that end devices can recover from failures.

Acknowledgements

This master thesis was carried out at the Institute for Technical Informatics, Graz University of Technology.

First of all, I would like to thank Dipl.-Ing. Dr.techn. Christian Kreiner for the incredible mentoring, advice and the possibility to work on such an interesting topic in cooperation with the company smaXtec. Dipl.-Ing. Dr. techn. Tobias Rauter took over the role as my mentor after the tragic and unexpected death of Dipl.-Ing. Dr. techn. Christian Kreiner. I want to thank him for the great mentoring as well. He really provided huge effort and support as mentor. Not even a vacation in the beautiful mountains of Salzburg prevented him from correcting my thesis. Besides my mentors, I also want to thank Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger who guided me through my whole study and provided great technical and organizational support during writing this thesis.

Moreover, I would like to thank my whole family for the incredible support during my study over the past years. They always believed in me and gave me the opportunity to move into a another city for studying, far away from home. Furthermore, I want to thank all my great colleagues at smaXtec. They helped me a lot with the amendments of this thesis and they were very patient with me when company related tasks took a "bit" longer due to the master thesis. Last, but most certainly not least, I want to thank especially Paul Rouschal BSc. who helped me a lot with technical issues related with this thesis.

Graz, May 2019 Michael Astl

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The Internet of Things (IoT) is a rapidly growing industry field. The number of connected devices is growing exponentially, which means engineers have to face new challenges every day [AFGM+15, AIM10]:

- Requirements often change and new features may become necessary when products are already out in the field.
- When security weaknesses of IoT devices are exposed, they need to be fixed as quickly as possible.
- Time to market is a critical factor in the industry nowadays. The resulting shorter development cycles lead to more undiscovered bugs at the time of rollout.

The product's role, capabilities and functionality are mainly defined by the software running on the Embedded System (ES). This means that the possibility of updating this software while the device is in use at the customer's site would be a great benefit of the product and would provide much more flexibility [JS14]. IoT environments often consist of thousands of devices, which may not be physically reachable once deployed out in the field. Thus, approaches like standard serial programming, which requires a wired connection, are not suitable [MH13]. A solution capable of wirelessly updating many devices at once is required.

1.2 Goals

Existing optimization approaches are mainly focused on efficient and reliable image distribution in multi-hop Wireless Sensor Networks (WSNs). The goal of this thesis is to design an Over The Air Update (OTA) solution that is used in star topology networks, such as LoRa [All15], which reduces the complexity of the distribution challenges. However, reducing the transmission data needed to perform updates is a very important requirement for all types of WSNs, especially for LoRa networks. Sensor nodes often have limited power sources (for example, battery powered). The goal of this thesis is to develop an efficient and reliable OTA solution that can be deployed in a wide range of IoT environments,

which use resource-constrained devices and lossy, low-bandwidth network technologies for communication. For efficient delta image generation, the so-called Delta-Generator with Offset (DGO) algorithm [Ast19] should be integrated in the proposed OTA solution.

1.3 Problem Statement

There are several issues related with the defined goals that have to be solved within this thesis:

- **Resource Constrained Devices**: Sensors used in IoT applications mostly have limited energy resources. To provide a sufficient sensor lifetime, low-power MCUs, which have limited memory and computational power, have to be used. In order to keep the cost per device to a minimum, firmware updates should be possible without any additional hardware. Existing solutions use external memory or even separate hardware components to process firmware updates.

- **Reliability**: Once deployed in the field, sensors are often physically inaccessible. The possibility of the target device ending up in an undefined state after updates have been performed, has to be avoided. The sensor should be able to recover from invalid update data and unexpected events like a reboot during updates.

- **Efficiency**: Transmitting data causes high energy consumption for sensors. When firmware updates require a large amount of data to be transmitted, the lifetime of the sensor may be drastically reduced. The used wireless network technologies can only transmit a limited amount of data in a certain time, due to regulatory restrictions and low bandwidth. The distribution time for firmware updates should be kept to a minimum, additionally the network should be able to continue its normal operation mode while update data is distributed. Furthermore, the downtime of a sensor should be minimized during update in order to prevent a quality of service degradation.

- **Flexibility**: The OTA solution should provide the possibility of replacing every component of the firmware image that requires changes. Furthermore, the solution should be as platform independent as possible and should not be restricted to usage by a specific Operating System (OS). The solution should also be deployable on Microcontroller Units (MCUs) which do not use any OS.

1.4 Outline

Chapter 2 presents existing solutions for OTA. They are grouped into four basic approaches: Native Approach, Modular Design, Virtual Machines and Incremental Updates. Chapter 3 describes the structure and the concept of the presented OTA solution. At the beginning, an overview of the whole update system is provided. The following sections then describe the different components, such as similarity improvements, delta image encoding and target device processing. Chapter 4 provides further details on the implementation of the presented update components. Chapter 5 evaluates the OTA solution. Initially, the

developed simulation framework is presented, then the efficiency of the developed similarity improvements approach is evaluated in the rest of the chapter. Finally, Chapter 6 concludes this thesis.

Chapter 2

Related Work

A lot of research was already done regarding Over The Air Updates (OTAs). The primary focus in these publications was laid on how to minimize the necessary data required for processing updates on the target nodes and how to efficiently distribute the update data in the Wireless Sensor Network (WSN). Published dissemination protocols mainly focus on multi-hop sensor networks where code distribution is more complex compared to star-topology based sensor networks. The OTA solution presented in this thesis is intended for usage in Long Range Wide Area Network (LoRaWAN), a star-topology based sensor network. The focus in this chapter will be mainly laid on reducing the number of bytes to transfer for performing updates (efficient encoding) and existing software designs for resource constrained sensors, which contribute to efficient encoding. Existing solutions can be grouped into four basic approaches as shown in figure 2.1.

The most primitive solution is the *Native* approach. When a firmware update is desired, the whole image containing the new firmware is transmitted to the sensor and replaced with the old firmware. A more sophisticated approach is the *Modular Design*. The sensor firmware is grouped into a replaceable part containing the application code and a non-replaceable part containing system software such as OS, dynamic loader and other software parts, which do not need to be updated. Another approach is the usage of *Virtual Machines (VMs)* on resource constrained sensor devices. In this approach, the non-replaceable part is extended with a VM or agent system on top of the OS. This enables to execute more compact and higher abstracted code (bytecode) on the sensor node. When application updates are desired, only the bytecode containing the changed application logic needs to be transmitted to the sensor node. *Incremental* updates make use of the fact that under normal conditions, only small parts of the code actually change. In this approach, only the changes between the new and old firmware version are transmitted to the sensor node. After transmission, the sensor node reconstructs the new firmware version using the transmitted diff file and the old firmware-version, which already resides on the sensor node.

Sections 2.1 to 2.4 provide insights into existing solutions using the four basic approaches illustrated in figure 2.1. Some of these solutions also are mixing the approaches shown in figure 2.1. Section 2.5 compares the existing solutions with each other.

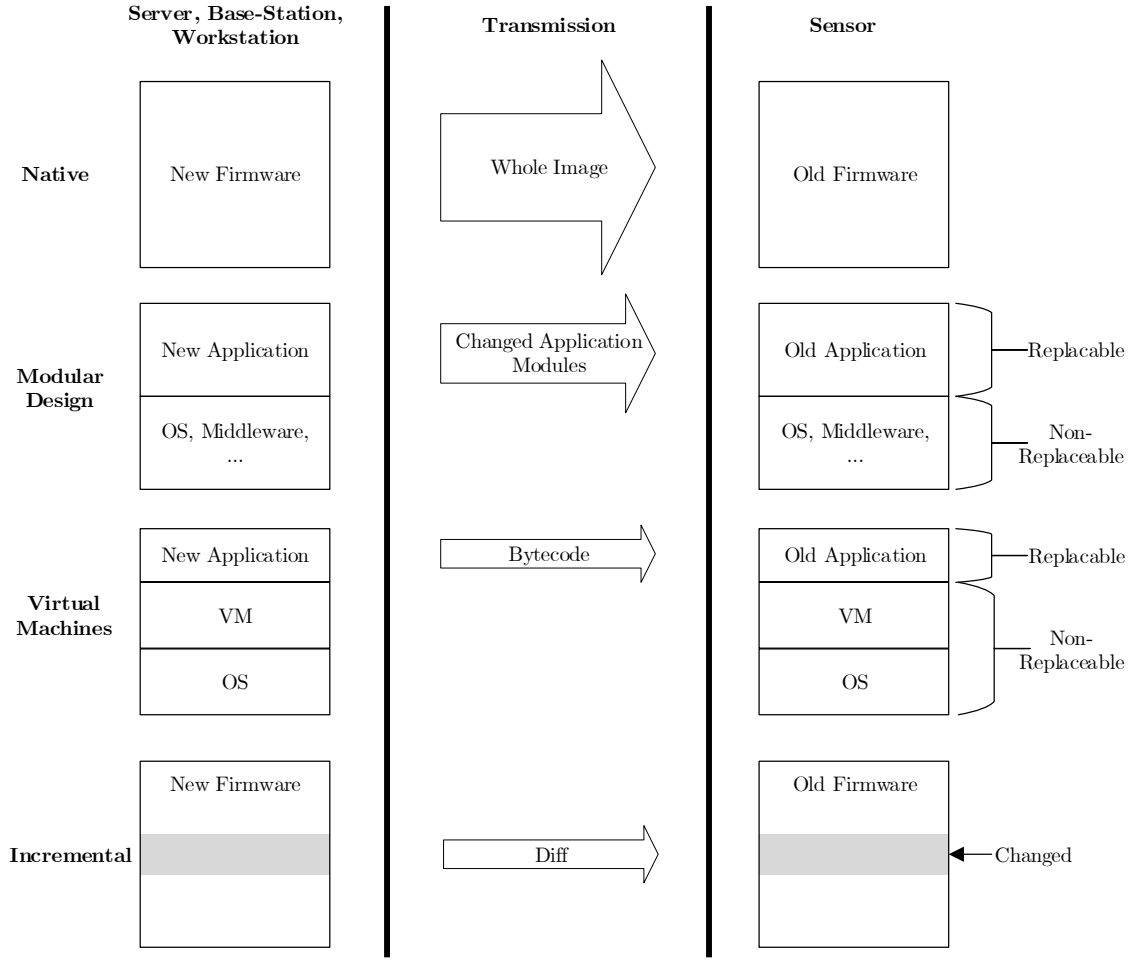

Figure 2.1: Basic Approaches for performing OTA in IoT Environments.

2.1 Native Updates

Updating sensor nodes by distributing the whole image is a simple but yet very flexible way to perform Over The Air Updates. The number of solutions where native updates are performed, is very scarce. This is not surprising due to the bad performance and efficiency compared to more sophisticated approaches as described in section 2.5. The first solution presented in this section is XNP [Incar, Manar]. XNP is the network programming implementation provided by the TinyOS 1.1 release [LMP+05]. It is the first solution that enables OTAs for devices equipped with TinyOS. Due to the missing support of multi-hop distribution in XNP, Deluge was introduced for TinyOS [HC04]. Deluge enables to disseminate the firmware update packets in multi-hop sensor networks. Another native update solution is Stream [PKB07]. Stream reduces the amount of update traffic by preinstalling the reprogramming protocol on the sensor nodes before they are used out in the field. In fact, Stream could also be seen as Modular Design approach because the reprogramming protocol is non-replaceable, which reduces the update size compared to Deluge. Another possibility to further reduce the update size is to apply code compression

on the transmitted image [TDV08]. All solutions presented in this section are part of Tiny OS or developed as extension for it. In the found solutions, the preparation mechanism for images to be distributed is always pretty similar as shown in figure 2.2, only the encoding step may differ in the existing solutions.

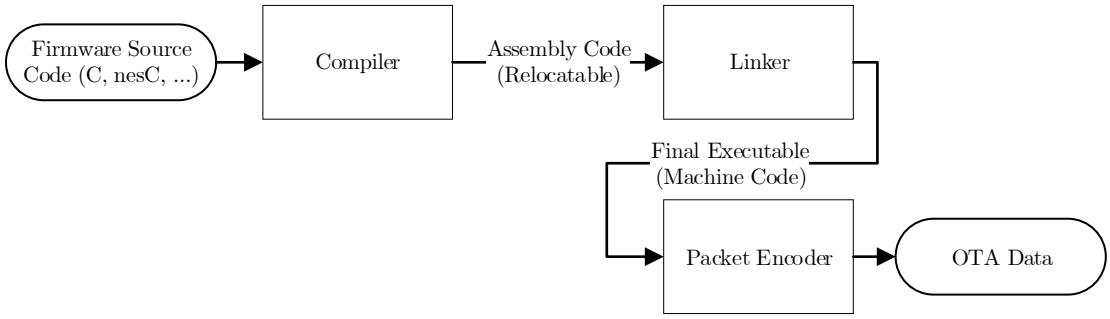

Figure 2.2: Firmware-Update Data Generation for the Native Approach.

2.1.1 Crossbow Network Programming (XNP)

XNP enables Wireless In-Network reprogramming for multiple target devices by sending the whole image using broadcast messages [Incar, Manar]. Following reprogramming steps are performed:

- Download Phase
- Query Phase
- Reprogramming Phase

In the *Download Phase*, the update data is sent from the base station to the sensors via broadcast messages. The sensor stores the received data on the EEPROM. The *Download Phase* is initialized from the base station by sending a "download start message" multiple times. The sensors can accept the update by sending back an acknowledge frame to the base station. In the *Query Phase* asks the base station the sensors if any packets are missing. Each sensor scans the EEPROM and requests retransmission in case parts are missing. Other sensors, which listen to the retransmission, can also use the packets for filling holes. In the Reprogramming Phase the sensor first checks the new firmware in the EEPROM for correctness. When the firmware is valid, control to the bootloader is transferred. The bootloader then copies the code from the EEPROM to the program memory and reboots the system.

2.1.2 Deluge

Deluge [HC04] is the standard reprogramming protocol included in TinyOS. Same as XNP, it also transmits the whole image for performing updates. The main improvement compared to XNP is the support of distributing firmware updates in multihop networks. Deluge consists of several software modules, which are shown in Figure 2.3. Deluge uses

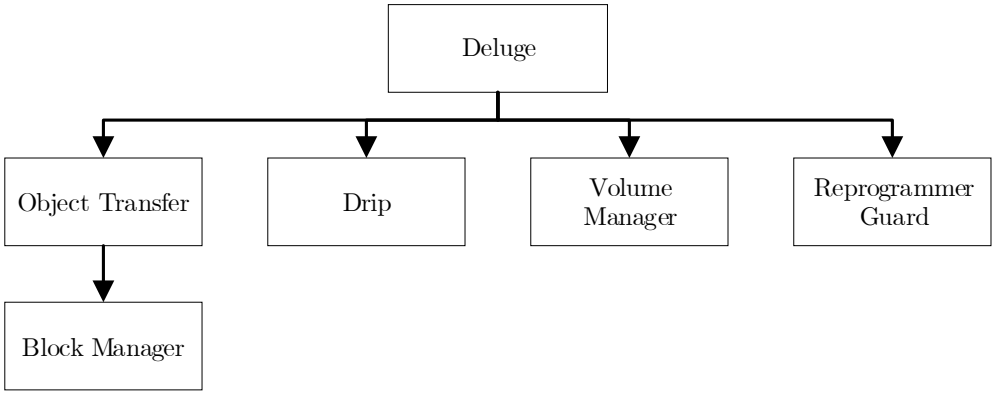

Figure 2.3: Code Structure of Deluge.

Drip [TC05] in order to disseminate command messages to the entire network for starting/stopping the distribution process of the image. *Drip* is based upon Trickle, a protocol for maintaining code updates in WSNs [LPCS04]. In Trickle, the sensors stay up-to-date by occasionally broadcasting a code summary to the neighbour nodes. In Deluge, this protocol is extended in order to support distribution of large data objects. Data objects (i.e., code image) are distributed by using a three way handshaking mechanism to ensure the complete delivery:

- Advertisement (ADV)
- Request (REQ)
- Code Transfer (DATA)

Several other solutions like MNP [KW05] and Freshet [KPB+08] use a similar ADV-REQ-DATA handshake mechanism for code distribution. In Deluge, distributed code images (data objects) are divided into a set of fixed-size pages. Furthermore, each page consists of a number of packets that are transmitted. Deluge uses a *Volume* and *Block Manager* for handling erase, read and write operations on the external and internal flash memory. The *Reprogrammer Guard* is responsible for ensuring that the sensor is capable of rebuilding and loading the received new image.

Due to the design of Deluge, the energy efficiency in real world applications where nodes are mainly battery powered, is suboptimal. Deluge is designed to operate over always-on links, the performance for distributing code over low-power links is very poor as shown in MobileDeluge [ZNV+14]. MobileDeluge was designed to overcome the above weaknesses of Deluge. At first, one-hop network reprogramming is processed instead of multi-hop. In order to reach all nodes in the WSN, the base-station is mobile. The mobile base communicates with the currently reachable nodes on a different channel in order to avoid interference with other nodes. The nodes in range and the mobile base also disable the low-power mode in order to allow a fast and efficient transmission of the new code image.

2.1.3 Stream

Stream [PKB07] was developed in order to reduce the transferred code size compared to Deluge by pre-installing the reprogramming protocol on the node before it's used out in the field. This is achieved by splitting the software into two images:

- Stream Reprogramming-Support (Stream-RS)
- Stream Application-Support (Stream-AS)

The Flash memory is segmented into multiple images, *Stream-RS* and *Stream-AS* are stored in two different image areas. These two images are compiled and linked completely independent from each other. When changes to the *Stream-AS* image are applied, no modifications to the *Stream-RS* image have to be done. The *Stream-RS* image contains the reprogramming protocol and is pre-installed on each node before deployment. *Stream-RS* builds on Deluge, it uses a three-way-handshake for hop-by-hop code dissemination. In Stream, the replaceable image contains *Stream-AS*, TinyOS and the application logic. Stream-AS is generic and can be added to any existing, Tiny-OS based application by just inserting two lines of code. The *Stream-AS* logic is responsible for switching to the *Stream-RS* image when any code update related message is sent to the sensor.

2.2 Loadable Modules (Modular Design)

Modular design solutions enable to replace only modules that have been actually modified instead of replacing the whole application when updates are performed. The firmware is grouped into a replaceable part and a non-replaceable part. The replaceable part consists of several modules that can be exchanged independent from each other when updates are performed. The non-replaceable part contains the core logic, such as the Operating System (OS), dynamic loader and other core components. The replacement of loadable modules is handled by the core logic. Solutions like Contiki [DGV04], FlexCup [MGL+06] and Elon [DLW+10] use runtime relocation in order to dynamically load modules into the existing application. Modules, transmitted for updates, contain relocatable code and additional metadata required for relocation. The system core dynamically links the transmitted modules into the running application using the transmitted relocatable code and the metadata. A lot of modular design solutions require to send a lot of metadata (relocation entries, symbol tables, etc.) for updating desired modules. In SOS [HKS+05] the amount of metadata is reduced by generating position independent binaries for each module. Another drawback in solutions like Contiki, SOS and Flexcup is the extensive usage of the flash memory. To address this issue, Elon places and executes the loadable modules in RAM. Some concepts in modular design solutions are also used for incremental updates. R2 [DLC+13] and R3diff [DMH+13] use relocatable code in order to improve the similarity between different firmware versions (further described in Section 2.4).

2.3 Virtual Machines

Interpretable code executed on Virtual Machines (VMs) is usually much more compact than corresponding native code. Thus, running virtual machines on sensor nodes would

reduce the energy cost for processing OTAs significantly. [CPS07] provides an overview of existing virtual machine solutions that are deployable on resource constrained devices. In this paper, the existing VM solutions are classified into two different types: Middleware level VMs (Mate [LC02], VM* [KP05b], ...) and system level VMs (TinyVM [Sol00], Squawk [SCC+06]). Middleware level VMs follow the classical model of VMs that are positioned between OS and application layer in the software stack (as shown in Figure 2.1). System level VMs substitute the entire OS. Even though processing OTAs on VMs is very efficient in terms of transmission cost, they have several drawbacks which makes them probably a bad choice in most WSNs. The execution of VM code is less efficient than native code. The energy consumption will always be higher in sensors equipped with a VM compared to sensors that execute efficient native code.

2.4 Incremental Updates

In this section, several approaches for performing incremental updates will be introduced. Incremental updates make use of already existing firmware running on the target device (sensor). Instead of transmitting the whole image, only the delta between the old firmware, already available on the target device, and the desired new firmware is transmitted.

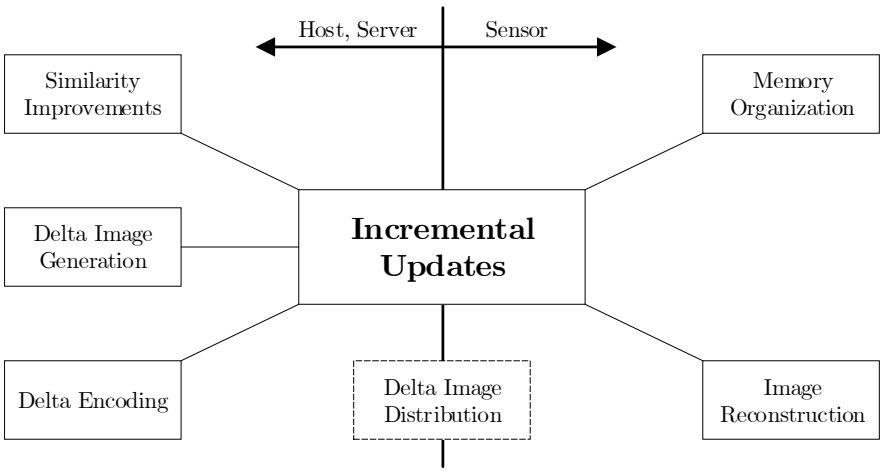

Figure 2.4: Components for applying Incremental Updates.

Figure 2.4 provides an overview which components are required for incremental updates. These components are further described in following subsections. Existing solutions for generating delta images are presented in [Ast19]. Similarity improvements are modifying the generated firmware in order to further reduce the resulting delta size. Existing solutions are presented in Section 2.4.1. After receiving the delta image, the sensor-node reconstructs the new firmware version using the transmitted delta image and the old version, which is already available on the sensor. The memory organization and the reconstruction process are highly dependent on the steps performed at the server. In Section 2.4.2, existing memory layouts and corresponding solutions for image reconstruction are further explained.

2.4.1 Improving Similarity

An efficient delta generation algorithm reduces the amount of bytes to be transmitted for performing OTAs significantly. However, a lot of existing solutions still claim that the resulting delta size is much bigger than expected in most cases. Even small changes in code often result in large delta size. This is caused by address shifts in the final executable code. Adress shifts are caused by relocated sections on Flash or RAM. A lot of instructions inside a firmware are vulnerable to address shifts. This was analyzed in Reijers [RL03]. The analysis showed that 16% of all instructions inside their example firmware were vulnerable to address shifts. This means that address shifts shouldn't be ignored when incremental updates are executed on resource constrained devices. The impact of address shifts on the resulting delta size can be reduced by applying similarity improvements between the version to update and the old version, which is executed on the target sensor. Table 2.1 compares existing solutions for improving similarity between different firmware versions. All the existing solutions can be grouped into four approaches:

- Slop Regions: Inserts spacings between sections.
- Indirection Table: Redirects function calls by using a fixed positioned table.
- Relocatable Code: Replaces addresses inside instructions with a constant value. Resolves addresses directly on the sensor by using the transmitted metadata, which contains the relocation entries.
- Assembly Modifications: Modifying executable assembly code in order to reduce address shifts.

Following subsections will provide a more detailed overview of these approaches including their advantages and drawbacks.

Indirection Table

Indirection tables are used for mitigating the effects of function shifts on the resulting delta image size. This approach is used in solutions like Zephyr [PBM09] and Hermes [PB12]. The concept behind indirection tables is shown in Figure 2.5. On the left side, the effects of function shifts on instructions is shown, *fun1* and *fun2* are relocated in the new version. Every *call* instruction in the new version, which points to one of the relocated functions, has a changed target address. A single relocated function can cause dozens of address changes when the function is called from multiple locations in code. In order to reduce these address changes, the *call* instructions are redirected using an indirection table. For each function inside the firmware, a fixed positioned entry in the indirection table is created. Each entry in the indirection table must be kept on the same memory location for the whole lifetime of the sensor. Each call target address is replaced by its corresponding indirection table entry. This is shown in Figure 2.5 on the right side: *Call* instructions pointing to *fun1* will be redirected to *loc1*, *call* instructions pointing to *fun2* will be redirected to *loc2*.

Due to the fact that usually functions are called from multiple locations inside a firmware, using indirection tables lead to smaller delta size. When functions are relocated, only the corresponding target addresses inside the indirection table need to be

Name	Year	Compared To	Approach
Reijers [RL03]	2003	None	Slop Regions Relocatable Code
XDelta [KP05a]	2005	None	Slop Regions
Zephyr [PBM09]	2009	Stream, RSync	Indirection Table
Hermes [PB12]	2012	Deluge, Stream, RSync, Zephyr	Indirection Table
Qdiff [SAH12]	2012	Deluge, Stream, RSync, Zephyr, Hermes, Elon	Slop Regions
R2 [DLC+13]	2013	Zephyr, Hermes, RMTD	Relocatable Code
R3diff [DMH+13]	2013	Stream, Zephyr, Hermes, R2	Relocatable Code
Delta Generator [KB16b]	2016	R3diff	Slop Regions
Trampoline [ZAZC16]	2016	RSync, Zephyr	Assembly Modifications
LiRep [QHQ16]	2016	Deluge, Hermes	Assembly Modifications

Table 2.1: Comparison of different Solutions for Similarity Improvements between different Firmware Versions.

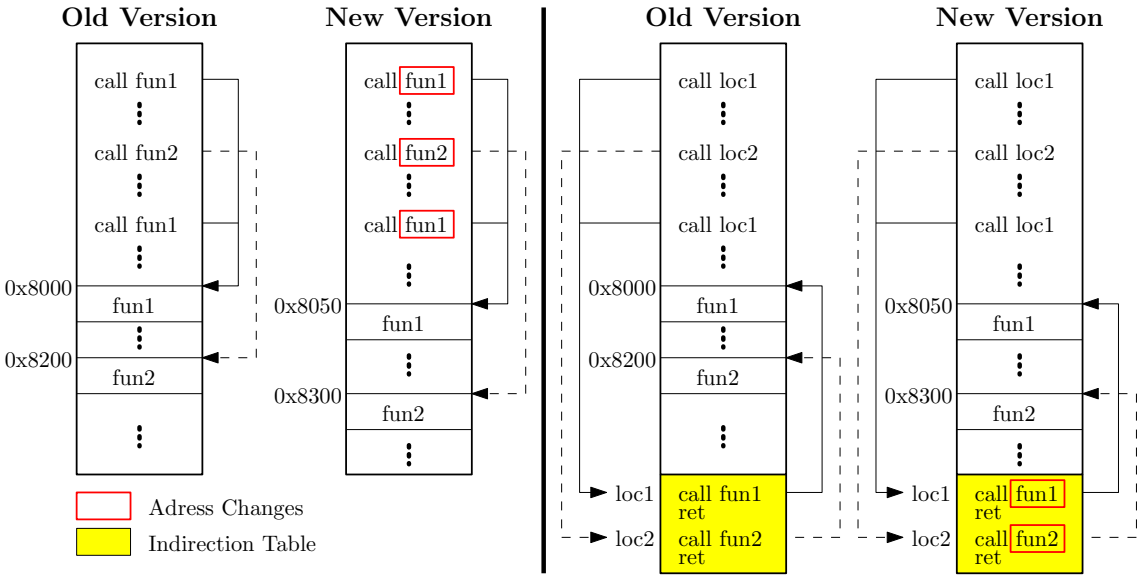

Figure 2.5: Improving Similarity of different Firmware Versions using Indirection Tables.

changed. In Zephyr [PBM09], the indirection of *call* instructions are applied in the linking stage. This means that no modifications in the source code are necessary. The indirection table approach presented in Zephyr has several drawbacks: Each call indirection creates execution overhead (e.g., 8 clock cycles on the AVR platform) for the MCU. This overhead accumulates for each function call that is executed on the running program. Another drawback is that only *call* instructions are handled by indirection tables. For example the effects of data shifts in the program memory or shifts of constants on flash wouldn't be

mitigated using indirection tables.

Hermes [PB12] is a solution that is built upon Zephyr including various improvements in order to eliminate the drawbacks mentioned before. Data shifts in the program memory are avoided in Hermes by organizing global variables into fixed positioned structs. This is achieved by applying software modifications before compilation stage. Another improvement compared to Zephyr is the removal of indirections before execution of the new firmware version. In Hermes, the bootloader is responsible for loading the reconstructed new image from external memory into the internal flash. The reconstructed new image is kept on external flash including its indirection table. During the image loading procedure, the bootloader eliminates all indirections by using the exact function address from the indirection table. Although Hermes includes several optimizations compared to Zephyr, there are still several problems arising with this approach: Hermes still only handles shifts on *call* instructions. Shifts on flash may have also effects on other instructions like relative jumps, accesses to constants and others. Another problem is the elimination of indirections by the bootloader, which requires the ability to locate call instructions inside the new firmware. This is not simple because the instruction-sets of most MCUs have varying length. The elimination of indirections is not further described in Hermes. The bootloader maybe has a simple disassembler implemented in order to locate *call* instructions.

Slop Regions

Using slop regions, is an approach in order to minimize shifts of sections in code. Indirection table approaches try to mitigate the effects of function shifts, slop region approaches try to avoid them by inserting placeholders between sections in code. The basic idea of slop regions is shown in Figure 2.6. Different change cases of the initial program are illustrated: Modification of existing sections, insertion of new sections and removal of existing sections. In the upper half of Figure 2.6, no slop regions are inserted between the sections. Each change case causes shifts of sections, which results in address changes for several *call* instructions. For example when section *fun1* is modified and it's size increases by 50 bytes, all sections placed below *fun1* are shifted down by 50 bytes respectively. These shifts of sections can be avoided by adding slop regions between the code sections in the initial program. This is shown in the lower half of Figure 2.6. Modified sections can either grow or decrease in size. In case of increasing size, the modified section grows into the following slop region. The size of the slop region decreases accordingly in order to preserve the location of all sections placed below. When the size of a modified section decreases, the following slop region increases it's size accordingly. New sections are either placed into existing slop regions or at the end of the memory when no slop region has sufficient space. Removed sections are simply replaced by slop regions.

Solutions that use slop regions for similarity improvements of different firmware versions have two major drawbacks:

- Inefficient usage of memory (memory fragmentation).
- Size of modified section increases beyond the slop region below.

Memory fragmentation cannot be prevented when slop regions are used. In the worst case, the new firmware doesn't fit into the memory anymore. In this case, the sections must be

rearranged and new slop regions must be created between the rearranged sections. This would lead to a large delta image because the sections get relocated and lots of address shifts occur. The second drawback, mentioned above, is solved by most solutions that use slop regions. In Reijers [RL03], sections get relocated when a modified section grows beyond the slop region. The effects of address shifts on the resulting delta size are reduced by adding an additional delta command called patch-list. In XDelta [KP05a], modified sections are relocated when they grow beyond the slop regions. This approach ensures that all sections below the modified sections retain their previous position. In XDelta, address shifts are not prevented for modified sections that become relocated, but still, the number of address shifts is very low compared to Reijers where all sections get relocated that are positioned below modified sections that grow beyond the slop space. Another approach that uses slop regions is Qdiff [SAH12]. In this solution, the part of the modified section that doesn't have sufficient space inside the following slop region is placed at the end of the memory. A *call* instruction is inserted into the modified section, which retains the same position. This *call* instruction points to the relocated part at the end of the memory. Qdiff is thus modifying the assembly in order to improve similarity. This approach is similar to the solution presented in Trampoline [ZAZC16]. Qdiff also changes the layout of the global variables in RAM (.bss and .data sections). This layout-change prevents shifts inside the Random-Access Memory (RAM) when variables are added or removed. Besides the inefficient usage of memory, slop regions efficiently improve the similarity of different firmware versions. Some publications claim that fragmented memory increases the energy consumption of MCUs. In [OK16] an analysis regarding this claim was done. The results show that fragmented memory in fact requires more energy, but not unacceptably more. A realistic scenario in this analysis shows an increase of only 1.4% compared to non-fragmented memory.

Relocatable Code

The idea of relocatable code is to make all references to symbols the same. The purpose of symbols is to resolve the position of functions, global variables, constants on flash and others. In R2 [DLC+13], the target addresses of all instructions, which are resolved using symbols and relocation entries, are set to a constant value. This is shown in Figure 2.7 where the target addresses of *fun1* and *fun2* are replaced with $0x000$ inside the call instructions. Relocatable code has a big advantage when used for incremental updates: Rearrangement of sections in code has no impact on the delta size because all absolute addresses are replaced by a constant and thus won't change in different firmware versions. Relocatable code is a common technique used in dynamic linking and loading solutions (described in section 2.2). In R2, this technique is used to improve the program similarity. Relocatable code is not executable because the target addresses are not resolved yet. When firmware updates are performed using relocatable code, additional metadata must be transmitted to the target sensor node in order to make the code executable. In R2, the metadata contains the relocation entries as shown in Figure 2.7. For each position in code where the target address was replaced by $0x0000$, a relocation entry is generated. Using the relocation entries, the bootloader is able to insert the actual target address when the reconstructed image is loaded from external memory into the internal flash. In Figure 2.7 for example, $0x8000$ is inserted at *pos1* and *pos2*, and $0x8100$ is inserted at *pos3*.

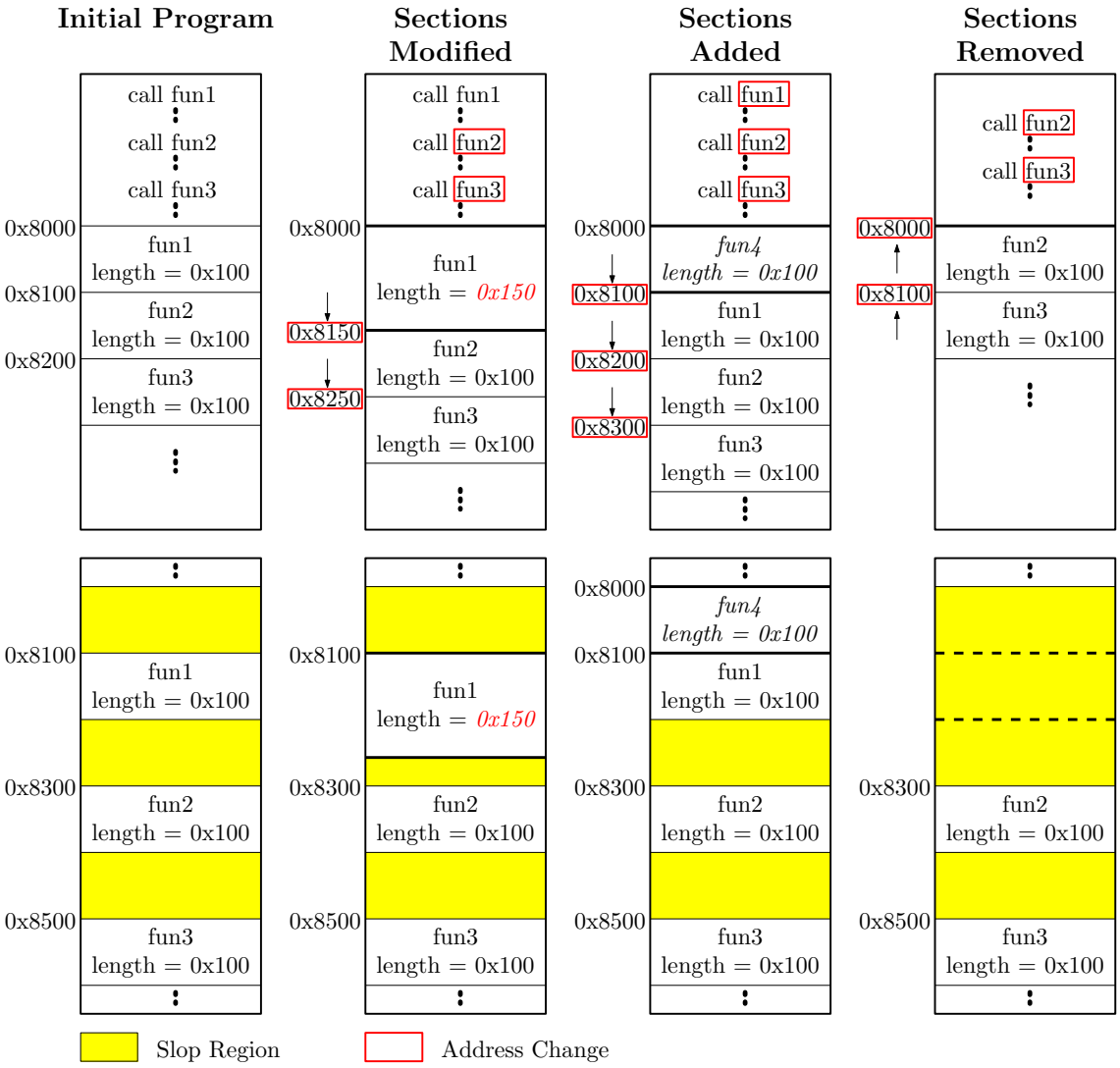

Figure 2.6: Improving Similarity of different Firmware Versions using Slop Regions.

Relocatable code covers more types of reference instruction types than indirection table based approaches. Compared to indirection tables, the size of metadata is much higher. For each reference instruction, a relocation entry is needed. This will result in a large amount of metadata for complex programs. In R2, this problem is addressed by using additional delta commands specifically designed for efficient delta encoding of relocation entries. Nevertheless, the amount of metadata that must be transmitted when performing updates is still too high as described in R3diff [DMH+13]. In order to further reduce the metadata size, the solution presented in R3diff only transmits the symbol table and a bitmap instead of the relocation entries. The approach of R3diff is shown in Figure 2.7: Instead of placing a constant into the reference instructions, the symbol table index is inserted. Using the symbol table index, the bootloader is able to locate the actual target address inside the symbol table. In Figure 2.7 for example, symbol index *#1* refers to address $0x8000$ *(fun1)* and symbol index *#2* refers to address $0x8100$ *(fun2)*. The bitmap

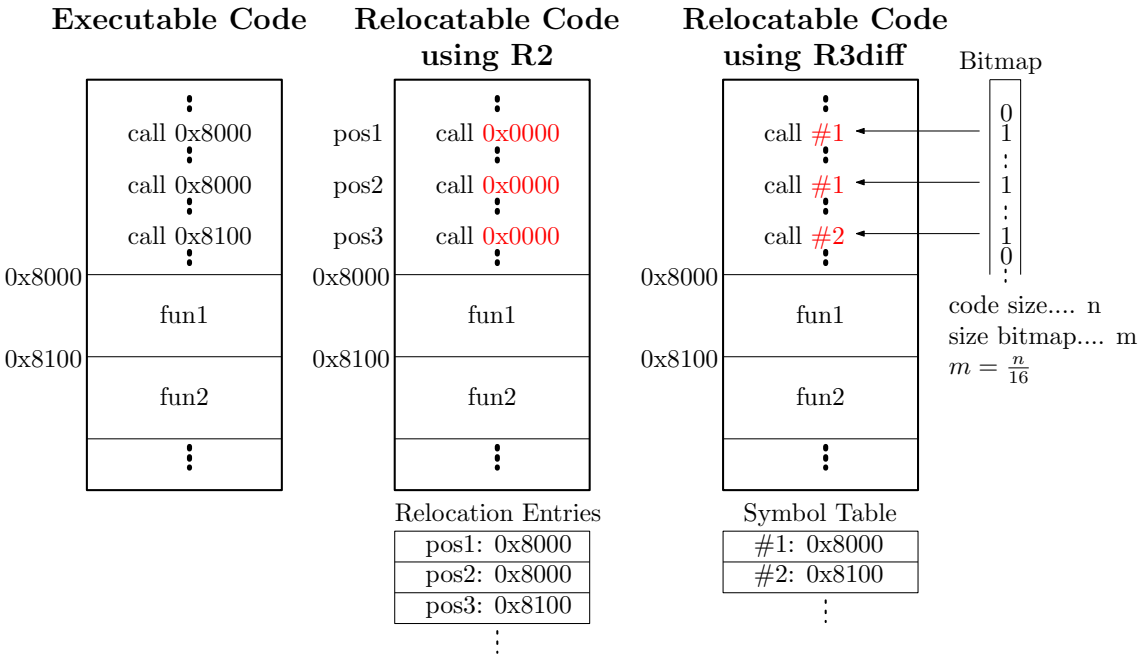

Figure 2.7: Improving Similarity of different Firmware Versions using Relocatable Code.

is required for the bootloader in order to locate the positions in code where symbol indexes have to be replaced by the real target address. Due to the 16-bit architecture of the used MCU in R3diff, each bit in the bitmap maps 2 bytes inside the memory. This leads to a bitmap size m of $n/16$, where n is the code size. Compared to R2, the amount of metadata is efficiently reduced in R3diff. Yet some aspects regarding the transmission of the symbol table and the bitmap are unclear in the presented paper: Does R3diff transmit the whole metadata for each update or is the delta algorithm applied to the metadata as well? If the metadata is inlcuded in the delta generation phase there are still some concerns. In relocatable code approaches, reference instructions often are relocated in different firmware versions. This causes a lot of changes inside the bitmap for every firmware version which leads to larger delta files.

Assembly Modifications

In LiRep [QHQ16] and Trampoline [ZAZC16], solutions are presented that preserve similarity by modifying the assembly executed on the target node. LiRep introduces the concept of in-situ code modifications where the code executed on flash is directly modified instead of reconstructing the image on the external flash. This reduces the necessary energy for rebuilding the new image compared to other solutions like R3diff [DMH+13] and others. LiRep defines three types of update blocks: code insertion, code removal and code modification. In case of code modifications, a single section on flash may is split: One part of the modified section remains on the same position and the other part is positioned at the end of the code. This requires to insert *jump* instructions in order to preserve the execution flow, otherwise the node would end in unexpected behaviour. The solution presented in [ZAZC16] uses a similar approach extended with trampolines. Trampolines are

used in order to avoid consistency problems when currently running code is updated. The main target of the solution presented in [ZAZC16] is to minimize the downtime of nodes while updates are performed. Although these approaches seem to be promising, they have very high complexity to implement and cause massive fragmentation (single sections are split). Also the reliability is questionable because code that is currently being executed, becomes modified. In LiRep for example, there was no explanation what happens if the code part, which performs the update, is modified using in-situ modifications.

2.4.2 Memory Organization and Image Reconstruction

Existing incremental update solutions can only be applied on platforms where corresponding memory storage is available. Some of the existing solutions require additional external memory while others only need the MCUs internal memory for processing updates. In Figure 2.8 different types of memory layouts are shown. In following subsections, the different memory layouts and required image reconstruction considerations are described in more detail. Existing incremental update solutions are mainly designed for one specific memory layout. Nevertheless, most of the presented approaches could also be modified in order to support different memory layouts while others can only be used when external memory is available. This is shown in table 2.2.

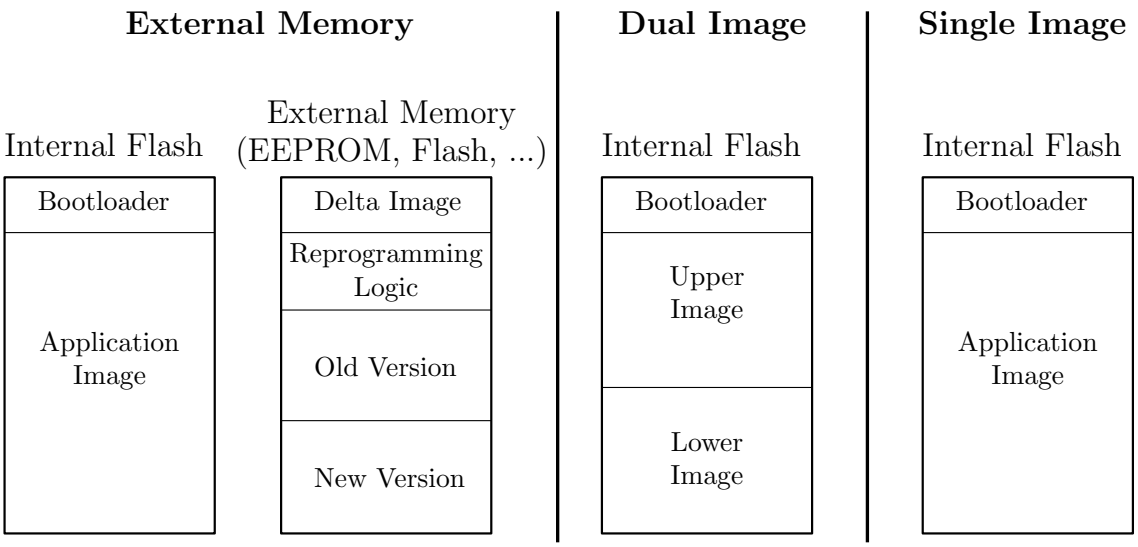

Figure 2.8: Types of Memory Layouts used in Incremental Update Solutions.

External Memory

Equipping sensors with external memory in order to process incremental updates has several advantages: The received delta image can be buffered in external memory. This enables the main application to continue execution while the delta image is received. Another advantage is that external memory is usually non-volatile. In case of an unexpected device reboot while receiving the delta, the transmission of the delta doesn't have to be restarted from the beginning in contrast to storing deltas in RAM. Solutions like Zephyr

Solution	External Memory	Dual Image	Single Image
Reijers [RL03]	Yes	Yes*	Yes*
XDelta [KP05a]	Yes*	Yes*	Yes
Zephyr [PBM09]	Yes	Yes*	Yes*
Hermes [PB12]	Yes	No	No
Qdiff [SAH12]	Yes	Yes*	Yes
R2 [DLC+13]	Yes	No	No
R3diff [DMH+13]	Yes	No	No
Delta Generator [KB16b]	Yes*	Yes*	Yes
Trampoline [ZAZC16]	Yes*	Yes*	Yes
LiRep [QHQ16]	Yes*	Yes*	Yes

Table 2.2: Supported Memory Layouts of different Incremental Update Solutions (Yes*.... Modifications necessary for support).

[PBM09] and Hermes [PB12] additionally place the reprogramming logic on the external memory. The reprogramming logic doesn't have to be included into the application image, the node receives a command to reboot from the reprogramming component when updates are pending. While adding the reprogramming logic to external memory reduces the size of the application image, the main application is halted for the whole update process. Furthermore, updating the reprogramming logic itself is not possible with approaches like Zephyr and Hermes. Using external memory enables to store multiple firmware versions simultaneously and provides simpler image reconstruction compared to other memory layouts. The new firmware can be reconstructed directly on the external memory. When the reconstruction of the new firmware is finished, the bootloader is used to copy the new firmware from external memory into the internal flash. Some of the incremental update solutions store non-executable firmware versions on the external memory. The bootloader additionally performs tasks in order to make the firmware executable when loaded into the internal flash. This approach is used for example in R2 [DLC+13] and R3diff [DMH+13]. The delta files for processing updates are much smaller by applying delta algorithms on the generated relocatable code. The old version of the relocatable code is necessary for reconstructing the corresponding new version. The bootloader transforms the relocatable code into executable code while loading the new version into the internal flash. This means that solutions, which need to transform the firmware before it's executed on the internal flash, need additional memory spaces for storing the non-transformed firmware that is used to generate smaller delta files.

As described in [LWS18], most byte-level delta generation algorithms do not consider the problems related with page-based erase operations in real flash memories. This problem can easily be avoided when for example an EEPROM or FRAM is used as external memory instead of flash. Block-level based approaches like FBC [Jeo03], $\text{RSync}_{\text{Berk}}$ [JC04], Zephyr [PBM09], Hermes [PB12] and others usually set the block size equal to the page-size of the external flash. For each delta command, simply a whole page is overwritten. On the other hand, Hirschberg [MH13] for example uses FRAM as external memory, which allows to erase and overwrite single bytes for image reconstruction. Although the use of

external memory has a lot of advantages, there are also some drawbacks that should be considered: Adding external memory to a sensor raises the cost per device, especially when expensive memory chips like EEPROM or FRAM are needed for the incremental update solution. The energy consumption for the image reconstruction process is high [QHQ16]. A lot of read and write operations are necessary on the external memory and usually the whole internal image is overwritten for loading the new firmware.

Dual-Image

The dual-image layout splits the available internal flash memory into two halves, the upper image and the lower image. This layout provides the possibility to execute the firmware from one half of the image and simultaneously process an update on the other half without creating consistency problems. The only incremental update solution that mentions this type of memory layout is Reijers [RL03]. In Reijers, always the firmware in the upper image is executed. The lower image is only used for reconstructing the new firmware by using the old firmware in the upper image together with the transmitted delta. When the reconstruction process is finished, a small piece of code is loaded into RAM that copies the code from the lower image to the upper image. When the sensor reboots unexpectedly while code is copied from lower image to the upper image, the sensor would result in undefined behaviour. The MCU cannot be restored into a defined state anymore. Another thing to consider in dual image layouts is how to buffer the delta image, which is not further described in Reijers. The delta file can either be transmitted completely before reconstruction process starts, or the operations encoded into each single delta commands are processed immediately after reception.

Single Image

Single image layouts require image reconstruction approaches that have a very high reliability. In contrast to dual image and external memory layouts, no rollbacks to previous firmware versions are possible in case the new firmware is invalid. The big challenge is that only one area to store the application image is available, the reconstruction process cannot use any buffers, such as external memory or RAM, to store the reconstructed image before it's loaded into the executed application image. XDelta [KP05a] solves this challenge by transferring the control to the bootloader before the transmission of the delta file starts. The bootloader is handling the communication and the reconstruction process. Delta commands are received until a single page can be reconstructed. The next step is to reconstruct the affected page. This means that the bootloader only needs buffers for storing a small amount of delta commands and a single reconstructed page. The drawback of this approach is that no application logic can be executed as long as the bootloader performs the update. This issue is solved in LiRep [QHQ16] and Trampoline [ZAZC16]. The system downtime is minimized by only stopping the parts of the application that are currently updated. This approach is complex to implement and very platform dependent because the update logic directly manipulates the code execution flow.

2.5 Comparison of Existing Solutions

Native update solutions provide maximum flexibility when sensors need to be updated. The whole firmware is replaced in every update, which allows to modify every component inside the firmware. The main drawback in native update solutions is the high amount of data to be transmitted for each update. In resource constrained, low-power environments, this approach would exceed the network bandwidth limits. Deploying Virtual Machines on sensor nodes reduces the transmission cost significantly compared to native updates. Virtual machine code is much more compact than native code. On the one hand, updates in VM solutions only contain the application logic, which further reduces the transmission cost. On the other hand, this is a big drawback in VM solutions: Updates can only be applied to the application logic, updates of core components such as communication stack, OS and others, are not possible. Another drawback of VMs is the high execution overhead, which leads to reduced energy efficiency compared to platforms that execute native code. Modular design approaches occupy a middle ground with more flexibility and lower execution overhead than VMs, but also lower energy efficiency. In summary, none of these three approaches meet the requirements in this thesis (Section 1.3). The trade-off between flexibility, execution efficiency and update cost is not satisfying in native, modular, or VM approaches.

In contrast to the other approaches, incremental updates provide good flexibility, small updates and depending on the implementation, very small or no execution overhead compared to standard native code running on a sensor. Similarity improvements help to further reduce the resulting delta size when updates are applied. In Table 2.3, the different approaches described in Section 2.4.1 are compared using following properties:

- Toolchain Changes: Provides information if modifications in the compiler, linker or executable code are necessary. Note that indirection tables could also be implemented in the source code directly but this isn't the case in the solutions presented in Section 2.4.1.

- Preserving Similarity: Affects the resulting delta size when updates are performed. For example using slop regions would reduce the delta size more than using indirection tables.

- Runtime Efficiency: Some of the listed approaches degrade the runtime performance when applied on sensors. For example indirection tables cause additional jumps, which leads to a high execution overhead.

- Memory Efficiency: Describes the amount of memory additionally needed when the approach is applied to a sensor. For example slop regions cause massive fragmentation on flash.

- Processing Complexity: Describes the complexity for processing the received firmware update on the sensor node. Higher complexity raises the risk that updates may fail or something goes wrong while processing the update on the sensor node. In the worst case, a device could get bricked.

- External Memory: Describes if the approach requires external memory

- Platform Independency: High independency means that porting the similarity improving method to a different platform requires little effort, low independency requires a lot of effort.

In Chapter 3 a solution for similarity improvements is presented that is based on the slop regions approach described in Section 2.4.1. Slop regions are very efficient in terms of similarity preservation and the processing complexity is low compared to other approaches. Furthermore, the runtime efficiency is only slightly reduced and more importantly, no external memory is required.

	Indirection Table	**Slop Regions**	**Relocatable Code**	**Assembly Modifications**
Toolchain Changes	Yes	No	Yes	Yes
Preserving Similarity	Poor	Very Good	Good	Very Good
Runtime Efficiency	Low	Medium	High	Low
Memory Efficiency	High	Low	High	Low
Processing Complexity	Medium	Low	High	Very High
External Memory	Yes	No	Yes	No
Platform Independency	Medium	High	Medium	Low

Table 2.3: Comparison of Similarity Improvement Approaches.

Chapter 3

Architecture and Concept

In this chapter, an incremental updates concept for resource constrained devices is introduced. The presented architecture is not limited to a single device type. It can be deployed in any environment where sensors equipped with MCUs meet following requirements:

- The Memory containing the firmware can be overwritten by the MCU itself.
- Sufficient memory is available for OTA. The memory requirements are further described in Section 3.3.

In Figure 3.1, an overview of the concept presented in this thesis is shown. The update process starts with improving the similarity between the old firmware, which is residing on the target sensors, and the new firmware. When this step is finished, a delta image is generated using the old and the new firmware image. The generated delta image is prepared for the transmission phase in the *Delta Image Encoding* step. The Firmware Update Manager (FUM) is responsible for storing and providing all the necessary information when updates are processed. When updating sensors to a new firmware version is desired, the FUM provides the necessary information of the old firmware image. After the *Similarity Improvement* step, the FUM stores all the necessary data of the adapted new firmware image for future updates. The FUM additionally provides metadata, such as version numbers and firmware checksums. This metadata is encoded into the transmitted delta image in the *Delta Image Encoding* step. Another task of the FUM is tracking the current firmware versions of the sensors in the WSN. The *Diff Image Reception* is the first step processed on the sensor. The received packets are buffered for the following *Image Reconstruction* step. The location of the delta image buffer is depending on the used memory layout of the sensor. After transmission is finished, the next step is to reconstruct the new image using the old image residing on the sensor and the transmitted delta image. The used memory layout specifies the target location of the reconstructed, new firmware and also the reconstruction logic. The memory layout is highly depending on the available memory on the platform. Platforms with less memory available may require a more complex reconstruction logic and reduce the reliability of the update process. After reconstruction of the new firmware, the sensor is rebooted. This triggers the execution of the bootloader, which checks if the new firmware is valid in the *Image Verification* step. The last step of the update process is the *Image Activation*. Depending on the used memory layout, multiple firmwares could be available on the sensor. The bootloader decides

which firmware image will be executed depending on the corresponding firmware version.

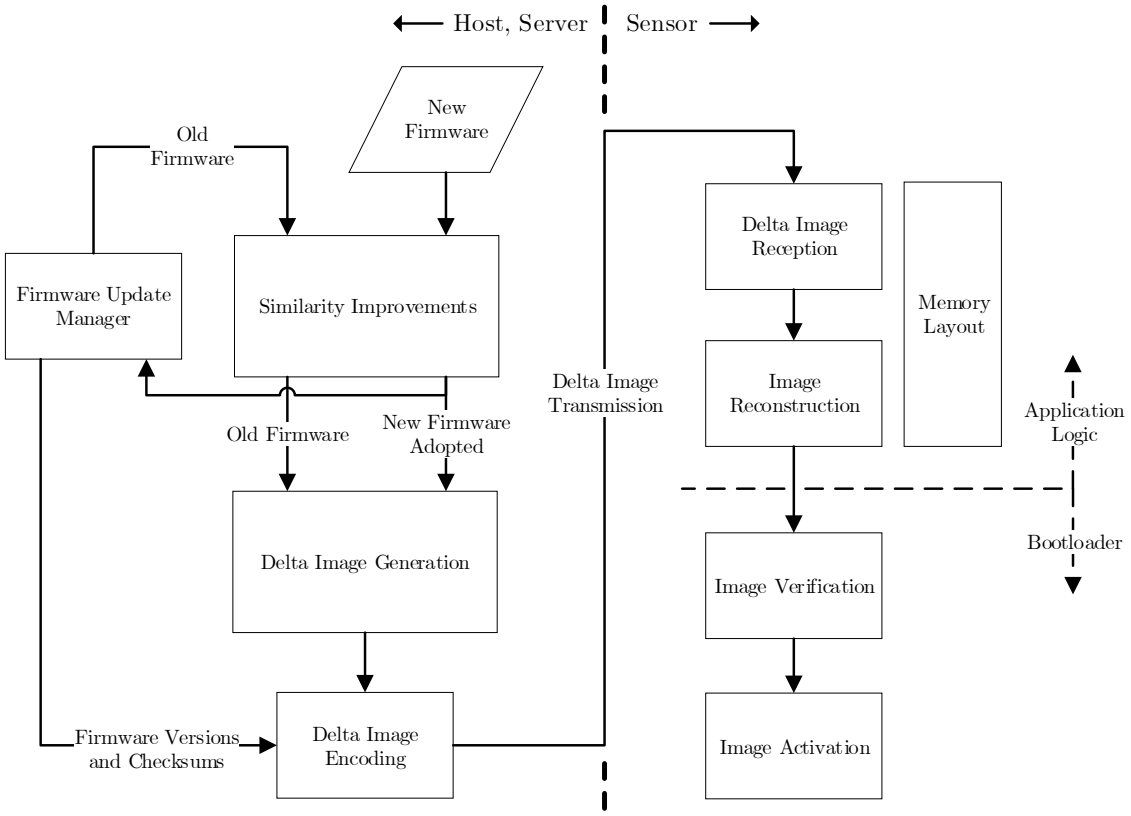

Figure 3.1: Overview of Incremental Update Concept.

The components, shown in Figure 3.1, are further described in the rest of this chapter and in Chapter 4. The algorithm used in the *Delta Image Generation* step is called DGO [Ast19], which is already described in a different thesis. The thesis, which presents this algorithm, uses a simple encoding scheme with fixed sized headers for delta messages. This chapter presents a more efficient and platform independent approach in Section 3.2.

3.1 Similarity Improvements

In this section, a concept for improving similarity between different firmware versions is shown. As already described in Section 2.4.1, similarity improvements can efficiently reduce the differences between different firmware versions, what leads to smaller delta files, no matter which delta algorithm is applied. In Section 2.5, Table 2.3, a comparison of the existing approaches is shown. The incremental update solution presented in this thesis should be deployable in environments where no external memory is available. Due to this requirement, only *Slop Regions* or *Assembly Modifications* are possible options for the similarity improvement concept presented in this thesis. When comparing the two remaining options, the *Slop Regions* concept is the only reasonable choice: First, changes inside

the toolchain should be avoided because they perform numerous optimizations specifically developed for the intended platform. Furthermore, *Assembly Modification* approaches are depending on the used instruction set of the platform, which leads to high platform dependency. Enabling reliable updates using *Assembly Modificatins* requires additional development effort and is hard to achieve. *Slop Regions* can be realized by simply modifying the linker script. Generating errors inside the firmware due to modifications of the linker script is not possible. When problems occur, the linker will throw an error and no firmware is generated. This means that applying *Slop Regions* does not require any additional validation of the generated firmware in contrast to modified firmware in *Assembly Modification* approaches. The executable code may becomes modified after the linking stage, thus validation mechanisms of the modified firmware are necessary. The linker script modifications are called Link Time Optimizations (LTOs) in this thesis.

The solution described in this section is based on the *Slop Region* approaches presented in Reijers [RL03], XDelta [KP05a], and QDiff [SAH12]. In Section 3.1.1, issues of these existing solutions are explained. Most of them will be avoided or minimized by the concept described in Sections 3.1.2 and 3.1.3. In Section 3.1.2, a placement strategy will be introduced that is intended to be used on the initial firmware, which will be programmed before roll-out of the sensors. This so called "major placement strategy" is optimizing the placement of code/data sections within the memory by analysing their dependencies. In Section 3.1.3, "the minor placement strategy" is introduced, which is used for creation of firmwares with maximized similarity based on an existing version. It uses concepts of existing solutions and additionally introduces new optimizations. The "minor placement strategy" generates firmware images used for incremental updates of already deployed sensors. Fragmentation of the memory is unavoidable when using the *Slop Region* approach for similarity improvements. Section 3.1.4 explains, why the major and minor placement strategies are optimized for efficient defragmentation. Partially defragmenting the memory minimizes the enlargement of the resulting delta file.

3.1.1 Analysis of Existing Solutions

The concepts in existing solutions assume that *Slop Regions* are already existing after each single section. There is no information provided what size for each *Slop Region* should be chosen in the initial firmware version. Several questions for initial placement of sections and *Slop Regions* remain unanswered:

- Which amount of memory should be kept unused? This affects the resulting size of each *Slop Region*.
- Should the size of each slop region be constant or should it be depending on the size of the corresponding section?
- Should the placed section have a certain order or should the default order, provided by the linker, be used?

Figure 3.2 shows an example where small *Slop Regions* are inserted between sections in order to preserve some free space in the initial program. Smaller *Slop Regions* lead to a higher probability that modified sections grow beyond the following *Slop Regions* boundary. In Reijers [RL03], this would lead to big delta files because the modified

sections are not relocated and thus numerous sections would be shifted (*Keep Position* in figure 3.2). In XDelta [KP05a], the modified section would be relocated to an unused position in memory (*Relocate Section* in Figure 3.2). The previous location of the modified section is filled with a *Slop Region*. Address shifts occur in both solutions whereas XDelta would lead to a smaller delta file. Small *Slop Regions* provide more free space in memory but they are often to small for modified sections in order to retain their position. They only have benefits in terms of delta generation when the size of modified sections slightly increases. Figure 3.3 shows an example where big *Slop Regions* are inserted between sections in order to reduce the number of modified sections that grow beyond the *Slop Regions*. A new section, which is larger than the *Slop Regions* between the sections, is inserted. Defragmentation is necessary in order to fit the new section into the memory. The resulting delta will have increased size due to the occurring address shifts caused by defragmentation.

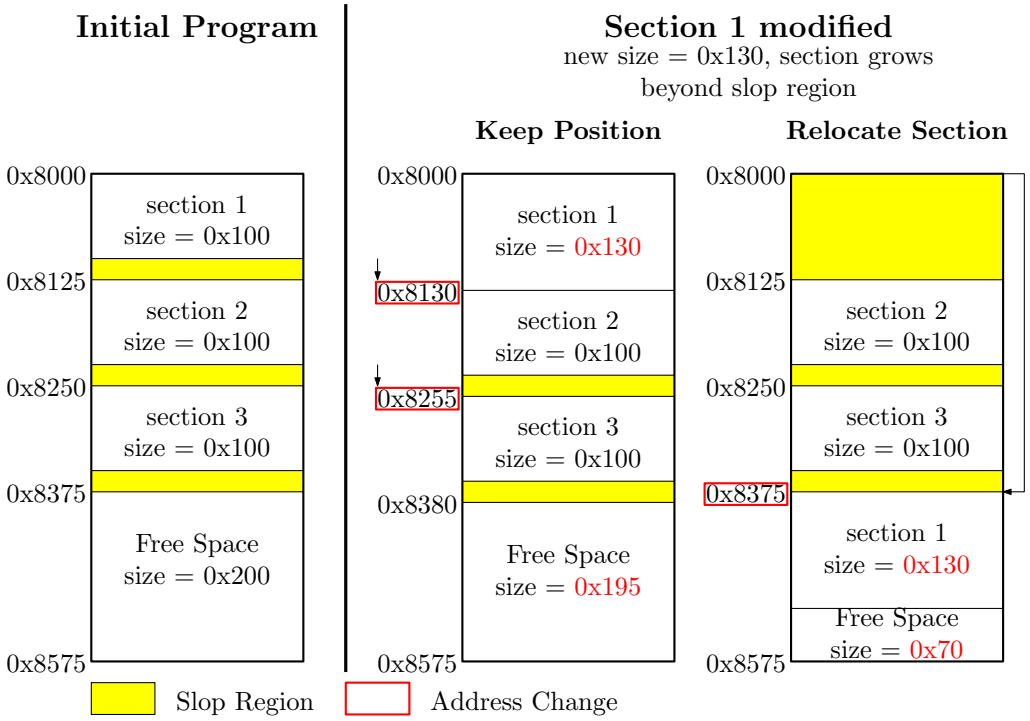

Figure 3.2: Initial placement of small *Slop Regions* between each section. The probability for modified sections which grow beyond the *Slop Regions* is high when firmware is updated.

Placing a *Slop Region* after every section is not efficient no matter what size for the *Slop Regions* is chosen. Small *Slop Regions* would cause a lot of relocations and waste of memory. Most of these small *Slop Regions* are probably never used. The sections above may never grow into the *Slop Region*. Furthermore is the size of the *Slop Regions* probably too small for most relocated and new sections. Big *Slop Regions* would probably require defragmentation in order to fit relocated and new sections into memory. Firmware typically contains hundreds of sections. Using all of the unused memory for creating *Slop Regions* would still result in relatively small *Slop Regions*.

Exisiting *Slop Region* approaches have several other drawbacks: A placement strategy for data memory (RAM) is only mentioned in XDelta [KP05a]. The placement strategy for RAM sections is different to the strategy used for flash. The strategy uses a memory map on the flash for preserving similarity in RAM. Shifts of sections in RAM have the same impact on the resulting delta size as shifts in flash. Thus, preventing this shifts and preserving similarity of the RAM layout between different firmware versions further reduces the delta size. The used placement strategies for performing updates have drawbacks as well: Reijers [RL03] does not relocate modified sections that grow beyond the *Slop Region*. Furthermore, it is not mentioned whether new sections are placed in existing *Slop Regions* or at the end of the firmware. QDiff [SAH12] splits sections that grow beyond the following *Slop Region*. Modification of code after the compilation and linking stage should be avoided. The reliability and efficiency of the system is reduced. XDelta [KP05a] relocates sections that grow beyond the *Slop Region*. On the one hand, the placement strategy minimizes the number of pages to be rewritten on the sensor, which reduces the cost for image reconstruction. On the other hand, is the placement strategy used in XDelta causing a higher amount of memory fragmentation.

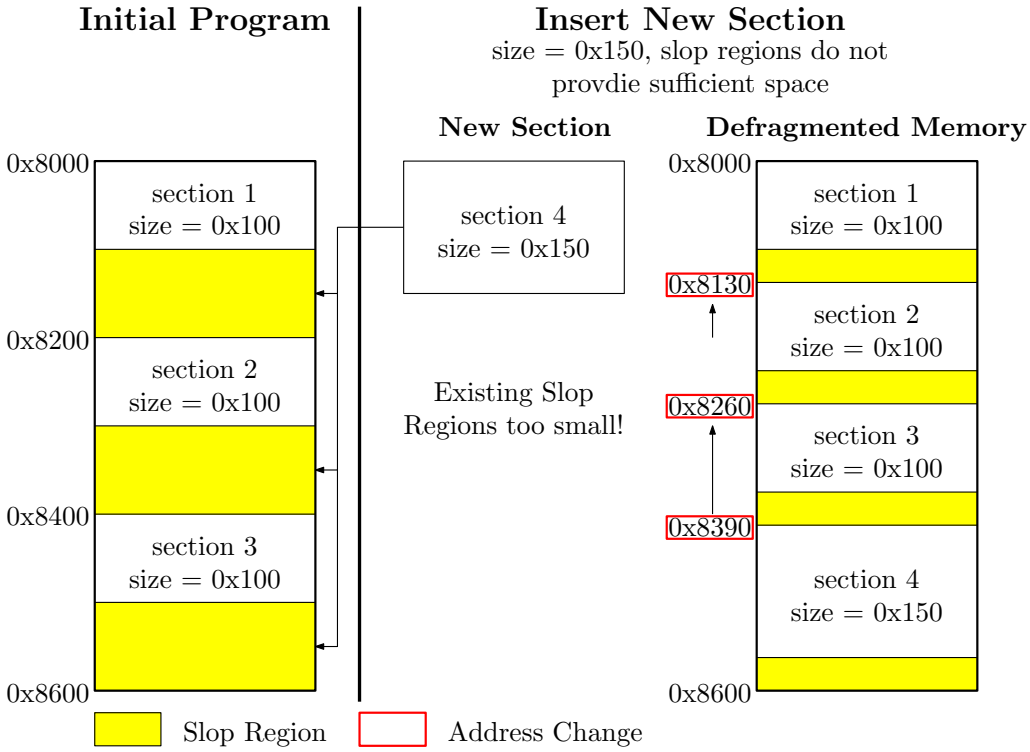

Figure 3.3: Initial placement of big Slop Regions between each Section. High Probability that Memory Defragmentation is required when Firmware is updated.

3.1.2 Major-Version Placement Strategy

In this section, a concept for creating major firmware versions is introduced. Figure 3.4 shows the changes in the flash layout when the presented approach is used. For example,

the GNU linker for MSP430 MCUs orders sections descending by their size in flash. Bigger sections are positioned at the beginning of the used flash region, smaller sections at the end. The presented concept first groups sections together before placing them on flash:

- Sections of a single object file (code file) are grouped together.
- Objects are grouped together into modules. Modules usually contain the objects of a single layer in the software architecture. Typical examples for modules are Hardware Abstraction Layer (HAL), Radio Protocol Stack, OS, Application Logic, etc.

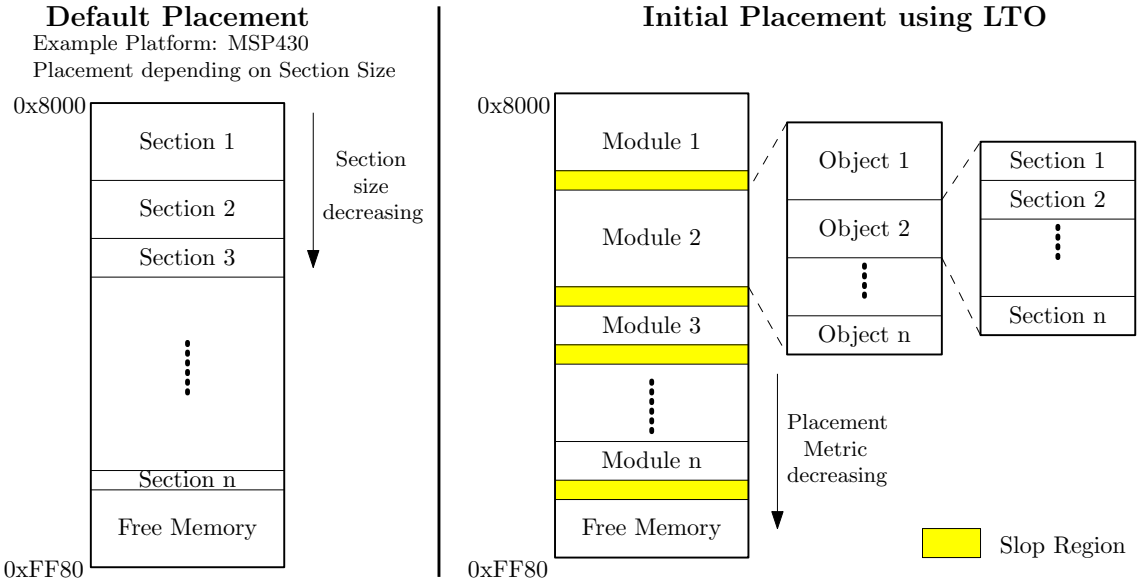

Figure 3.4: Major Placement Strategy on Flash.

After grouping the sections into objects and modules, the placement order on memory is calculated by analysing the inter-dependencies of the sections. For each section, object and module a so called placement metric ($place_{metric}$) is calculated:

$$place_{metric} = ref_{from} - ref_{to} \tag{3.1}$$

Parameter ref_{to} describes the number of references to other sections in code. Parameter ref_{from} describes the number of references from other sections pointing to the corresponding section in code. Existing slop region solutions use different approaches for preventing shifts in RAM and flash. In this thesis, the same approach will be used for both types of memory. Figure 3.5 compares the default placement with the initial placement using LTO. The default placement uses a MSP430 MCU as example. The default linker separates uninitialized (bss) and initialized variables (data) and puts them into the bss and data output section respectively. The linker orders the output sections descending by size except the stack region. The linker places the stack at the end of the RAM and the heap after the bss and data section. Most other linkers place the heap at the bottom and the stack above and leave some free space between them. The initial placement using LTO groups the sections placed on RAM the same way as done in flash. Sections are

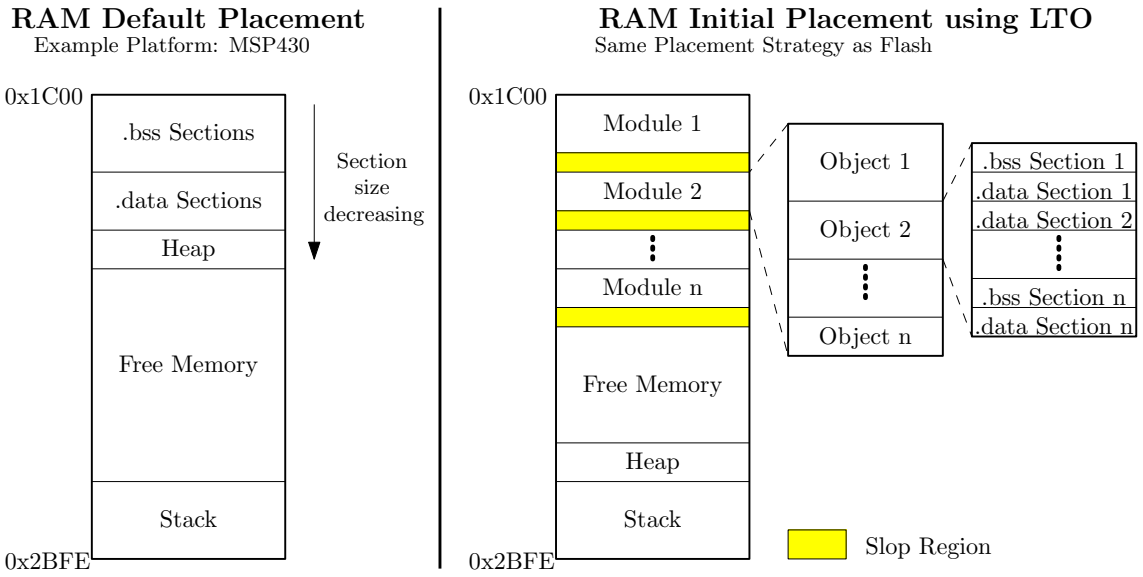

Figure 3.5: Major Placement Strategy on RAM. Sections containing initialized and uninitialized data are also grouped into objects and modules.

grouped into objects which are grouped into modules respectively. The placement metric calculation for RAM sections is identically to the calculation used for flash sections. The resulting placement order mixes data and bss sections on RAM, as shown in Figure 3.5. At startup, the MCU initializes the data variables in RAM. When data sections are grouped together on a certain location in RAM, the initialization logic can simply copy a sequence containing the init values from flash to the data region. This initialization logic only needs the start and end address of the data section in RAM and the start and end address of the flash region containing the values. When data sections are mixed with other sections in RAM, the data variables would be initialized incorrectly. The initialization logic needs knowledge of each data section address in order to copy the initial value to the correct position in RAM. Mixing of data and bss sections is supported on MSP430 platforms when the linker is configured to use the so-called "ROM model" for variable initialization [Tex13b]. When platforms require to group data and bss sections, two options are possible:

- Implement a custom logic for variable initialization.
- Split grouping of bss and data sections. This results in a series of modules containing the bss sections and a series containing the data sections. The modules containing the data/bss sections must be placed contiguous. Slop regions between bss or data sections are not a problem for variable initialization. The drawback is that fragmentation is increased because bss and data sections must be kept separate for each firmware version.

3.1.3 Minor-Version Placement Strategy

The minor placement strategy defines the memory layout based on an existing version. It minimizes the number of address shifts between the new and the old version. The changes

of a new firmware image compared to an existing firmware image can be broken down to three types:

- New Sections
- Modified Sections
- Removed Sections

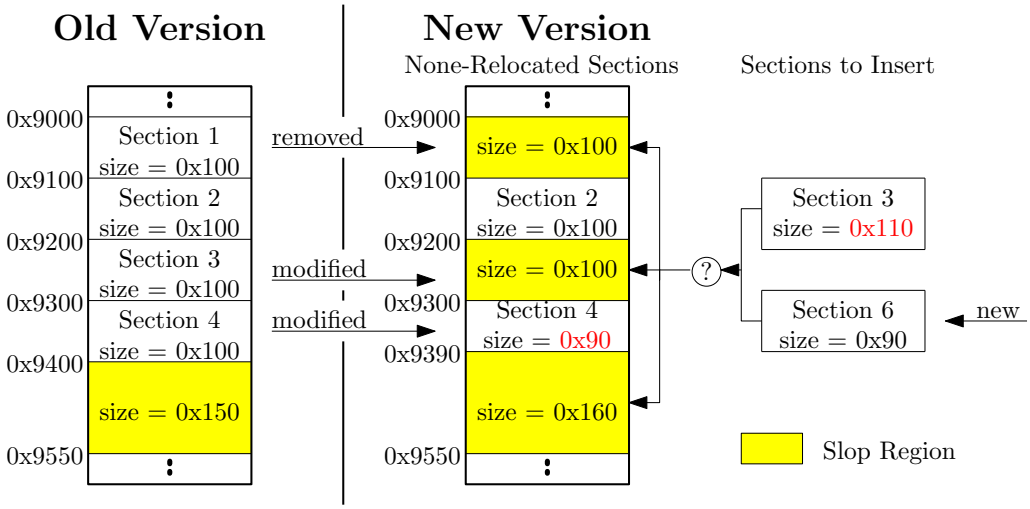

Figure 3.6: Minor Placement Strategy used for Similarity Improvements.

Figure 3.6 shows an example including these three types of changes. The minor placement strategy aims to preserve maximum similarity between two versions. Building the memory layout for the new version is divided into multiple steps:

- Fill the regions where removed sections were placed in the old firmware with *Slop Regions*.
- Handling of modified sections is depending on the changed size:
  - When the size of the section decreases, it remains on the same position in memory. In order to avoid shifts, a *Slop Region* is inserted after the modified section. *Section 4* in Figure 3.6 for example is padded with a *Slop Region* of 0x10 bytes.
  - When the size of the section is increasing and no *Slop Region* with sufficient size is placed below, the section must be relocated. When a *Slop Region* with sufficient size is below the modified section, it grows into the *Slop Region*. The size of the *Slop Region* must be reduced accordingly. When the modified section needs relocation, it is replaced by a *Slop Region*.

Performing these steps generates a memory layout where all existing sections, which do not require relocation, are kept on the same position relative to the old version. Figure 3.6 shows the temporary memory layout where non-relocated sections are placed. The next

step is to insert the remaining sections into the temporary memory layout. The remaining sections consist of modified sections that require relocation and new sections. Choosing the locations where the remaining sections should be inserted is a non-trivial task. The implemented algorithm should satisfy following requirements:

- Place sections inside the region of the module they belong to.
- Filling *Slop Region* with sections should result in minimized fragmentation. After inserting the remaining sections, the number of "small" *Slop Region* should be as small as possible.

Figure 3.7 shows an example for clarification of the second requirement. Example B shows the optimal placement for the shown problem case. In contrast, *Example A* places the sections in a way that three small *Slop Regions* remain in the memory. *Example A* causes more fragementation of the memory than *Example B*, which has a single *Slop Region* left after placement. The chance is higher that the single *Slop Region* in *Example B* can be filled with sections in future updates. The *Slop Regions* in *Example A* probably remain in the memory until defragmentation is performed. Filling the *Slop Regions* is actually a bin packing problem [bin06], which is NP-hard. This thesis introduces a heuristic approach to solve this problem. The developed algorithm is further described in Section 4.1.2.

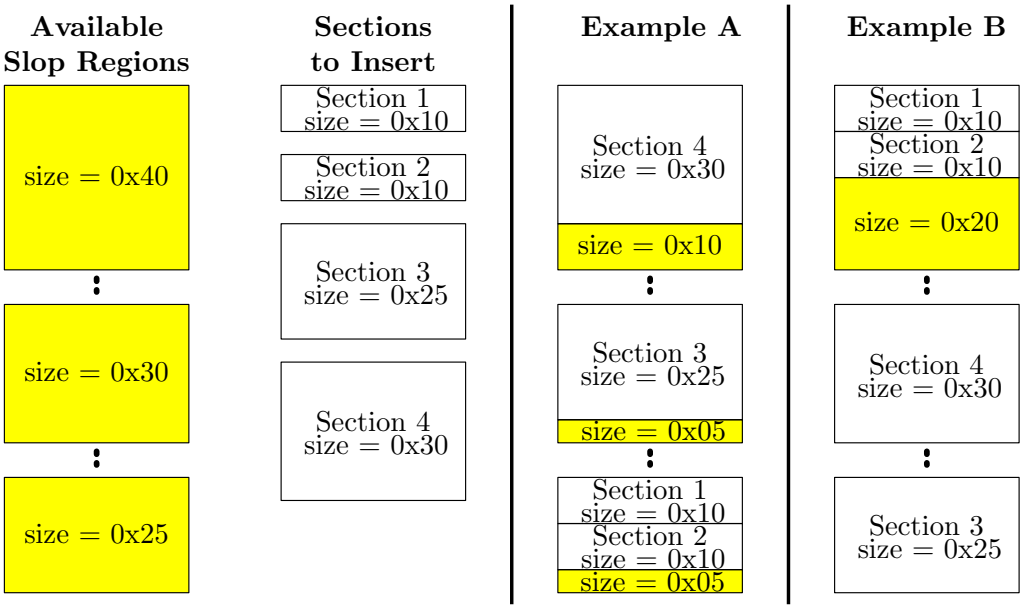

Figure 3.7: Placement of New and Relocated-Modified Sections into existing Slop Regions.

3.1.4 Partially Defragmentation

Performing numerous updates leads to fragmentation of the memory. The used minor placement strategy creates *Slop Regions* for relocated modified sections. At some point, the memory may has insufficient free slots for placing the sections required for the update. Defragmenting the memory creates free space but the delta size increases significantly

because a lot of address shifts are generated when the sections are rearranged. The major placement strategy described in Section 3.1.2 reduces the delta size when defragmentation is required. Grouping sections into modules and ordering them descending by their placement metric results in an initial placement order where most references point from higher memory addresses to lower addresses. In Figure 3.8 for example, most references point from *Module 3* to *Module 1* and *Module 2*. Modules with low placement metric typically contain application logic, modules with higher placement metric are typically system libraries, HALs, drivers, and others. Firmware updates usually include more changes in modules, such as the application logic for example. Changes to modules containing system libraries for example are less often required. Therefore, modules placed at higher addresses usually become more fragmented due to updates than modules placed at lower addresses.

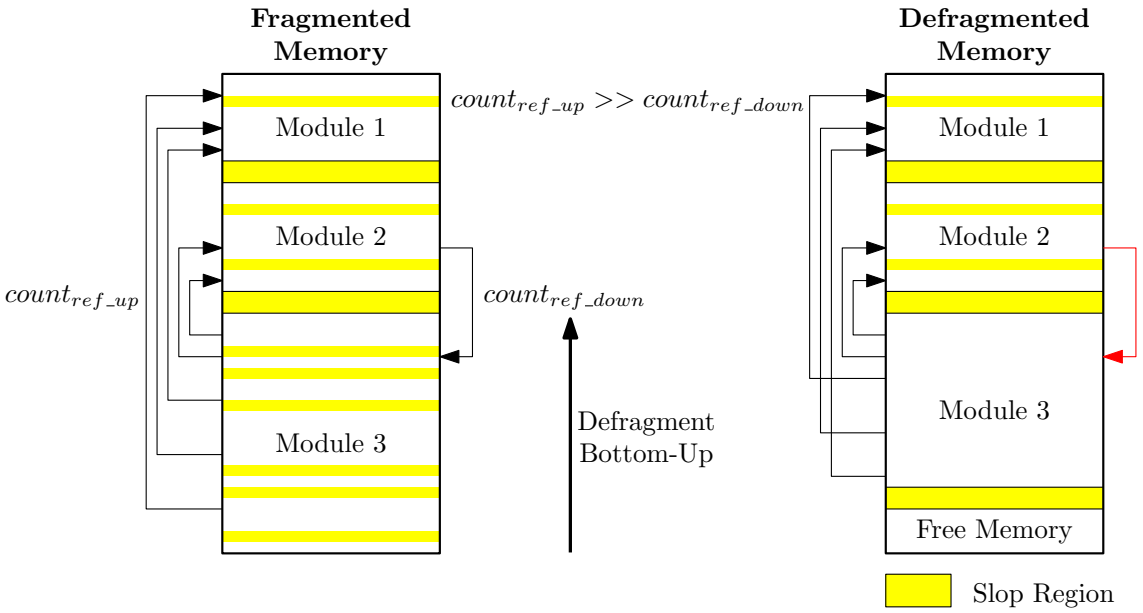

Figure 3.8: Bottom-Up Defragmentation of Memory.

The placement strategies introduced in this chapter enable to partially defragment the memory when additional space is required for placing sections. Defragmentation is started at the bottom (high address) and performed towards the top of the memory (low address) until sufficient free memory is available. Figure 3.8 shows an example where only *Module 3* is defragmented. The sections inside *Module 2* and *Module 1* retain on their previous position. The number of address shifts (changed target addresses) in these modules is minimized, only references pointing to the defragmented *Module 3* will create address shifts which increase the delta size. Absolute references pointing from *Module 3* to the other non-defragemented modules remain unchanged as well. Relative references on the other hand are changed due to the modified distance of the reference. Another advantage of the bottom-up approach is that defragmentation of modules at higher addresses also creates more free space due to higher degree of fragmentation.

3.2 Delta Image Encoding

The DGO algorithm, presented in [Ast19], creates a list of delta messages. This section presents a concept for efficient encoding of the given message list. The presented concept encodes numbers, such as addresses and lengths, with a variable length encoding called LEB128 [UNI93]. LEB128 enables to encode small numbers with a few bytes and at the same time enables to encode arbitrarily large numbers. These properties are very suitable for encoding delta messages. Another advantage of LEB128 is the simple decoding. Figure 3.9 shows an overview of the different encoding steps. Section 3.2.1 describes the content and the encoding of the delta image header. The encoding step groups COPY, ADD and OFFSET messages. This enables more efficient encoding because including the message type into each message header is not necessary. The relative addressing approach also enables more efficient encoding by using LEB128. Further information about message encoding is provided in section 4.2. The next step after encoding is to split the delta image into fixed sized packets. The chosen packet size depends on the amount of data that is transmitted in a single frame. After packaging, the delta image is ready for transmission to the target nodes. The *Update Info Packet* is used to inform the sensors that an update is pending. With the information provided in this packet, the sensor prepares for the upcoming update.

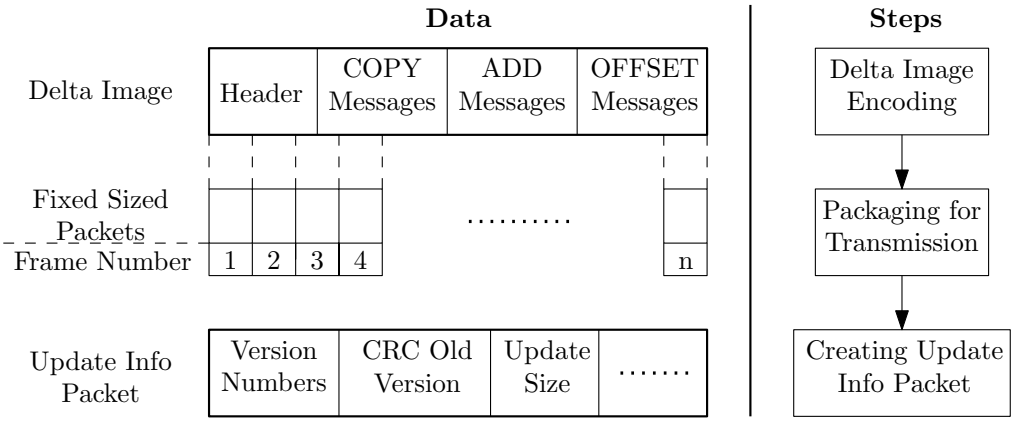

Figure 3.9: Encoding Overview.

3.2.1 Header Encoding

Figure 3.10 shows the delta image header structure. The header has variable length except the checksum, which has fixed length.

Checksum

The checksum is positioned at the beginning of the header, its calculation includes the rest of the image header and the delta image content (delta messages). The calculation ensures that the transmitted delta is valid. Due to the delta image check, the sensor is able to prevent reconstruction using an invalid delta image. This would lead to a corrupt new firmware with undefined content.

Figure 3.10: Delta Image Header Structure.

Base Image

The sensor reconstructs the *New Image* from the *Base Image*. Depending on the reconstruction approach, the sensor may has several options what image is used as reconstruction base. The *Base Image* field contains necessary information for reconstructing from the intended image:

- $base_{absolute}$: Absolute start address of the *Base Image*.
- $base_{size}$: Size of the *Base Image*.
- $base_{checksum}$: Checksum of the *Base Image*.

The *Base Image* checksum, which is included in the *Update Info Packet*, enables the target sensor to verify that the reconstruction base is identical to the base firmware (old firmware) used in the delta generation phase.

New Image

The *New Image* field contains the necessary information for reconstructing the new image to the desired location:

- $new_{absolute}$: Absolute start address of the new image.
- new_{size}: Size of the new image.

The *New Image* checksum is encoded directly into the firmware header. The delta image includes the firmware header, which means that the checksum information is sent with the delta messages. Section 4.8 provides further information about the firmware header.

Message Structure

The *Message Structure* provides information to find the start of each message group when reconstruction is processed. This is necessary because each message group has variable length and the type is not encoded into the messages directly. The *Message Structure* field includes following information:

- $size_{COPY}$: Size of the COPY messages block.
- $size_{ADD}$: Size of the ADD messages block.
- $size_{OFFSET}$: Size of the OFFSET messages block.

For page-level target memories, the delta messages are placed inside the delta image in a fixed order, ascending by a certain field inside the message depending on the message type.

Page Update Order

Each memory type has a certain granularity for overwriting data:

1. Page-level granularity (e.g., FLASH, etc.).
2. Byte-level granularity (e.g., EEPROM, FRAM, etc.).

When the reconstruction target has byte-level granularity, any message order inside their group can be used. The messages can be reordered arbitrarily for avoiding copy conflicts. The *Page Update Order* field is obsolete for byte-level memory types. To avoid copy conflicts at reconstruction phase, the *Page Update Order* field is included. This field tells the reconstruction logic on the sensor in which order the pages should be reconstructed. The header encodes three types of update orders:

- Ascending order from page 0 to last page.
- Descending order from last page to page 0.
- Custom update order.

In most update scenarios, ascending or descending order probably causes no copy conflicts. These orders can be encoded using a small header. When a custom update order is necessary, the header includes an additional array. The array's size depends on the number of pages available in the target image. A custom update order is only necessary when the base image and the new image are the same, otherwise no copy conflicts are possible because no source regions of COPY messages can be overwritten. Section 4.3 describes an approach to find a custom page order that prevents copy conflicts.

3.2.2 Update Info Packet

The gateway sends an *Update Info Packet* to each sensor that should be updated. The sensor uses the information inside this packet in order to prepare for the upcoming update. In order to prevent the sensor from performing an update using a wrong base image, the old version number and the CRC of the base firmware are included into the packet. In case that either the version number or the CRC is not matching with the existing base image, the sensor ignores the upcoming update. The *Packet Size* is mandatory for the sensor to

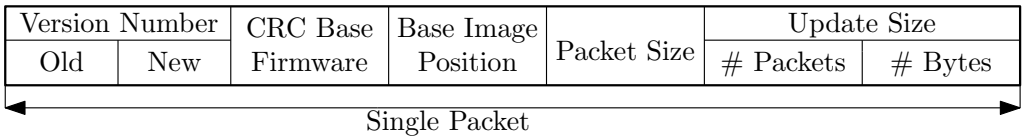

Figure 3.11: Content of Update Info Packet.

store the received update fragments correctly. Especially when packets get lost, the sensor

still can store the received packets into the right position in the delta image buffer. The device calculates the correct position by using the packet size in combination with the frame number. The *Packaging for Transmission* step adds the frame number to the fixed sized packets (see Figure 3.9). The update info packet encodes the size of the upcoming update including the number of packets and number of bytes. In the reception phase of the delta image, the *# Packets* info is used to determine the completion of the delta update transmission. Due to the fixed packet size, the last packet containing the delta image is usually filled with a constant value at the end. The decoder in the image reconstruction phase needs information at which position the end of the delta image is reached. Since delta messages have variable length, correct encoding is not possible without the delta size information. The update info packet could also include additional transmission related information. Depending on the used transmission technology for example, the planned update start time must be transmitted to the sensor.

3.3 Memory Layout

The presented incremental update approach requires numerous components that must be available in the memory of the target device. Section 3.3.1 describes these components. Section 3.3.2 provides an analysis of possible failure modes in the update system. Since the presented solution is intended for resource constrained devices, the available memory for those components is scarce. The used memory layout defines where the update components are stored. This layout is always a tradeoff between efficiency, reliability and memory cost. Section 3.3.3 compares some of the possible layout options. The possible options, which can be used on the target platform, mainly depend on the available memory types and their sizes. This section is intended to provide the reader some insights, which tradeoffs for a given platform have to be accepted, or which requirements must be fulfilled by the platform in order to meet the defined requirements for the OTA implementation.

3.3.1 Required Update Components

Figure 3.12 shows the update components for the presented OTA solution. The two *Application* components contain the firmware to be updated:

- *Reconstruction Target*: Defines the destination region where new firmware version is reconstructed.
- *Reconstruction Source*: Defines the source region that is used to reconstruct the new firmware together with the delta image.

The two images can either be placed on different locations or at the same location (*New Image = Base Image*). The *Reception Logic* describes the code that processes the reception of the *Update Info Packet* and the delta image packets. The *Reception Logic* performs following tasks:

- Writing the received delta packets into the *Delta Image* buffer.
- Handling of lost packets. Depending on the protocol used for delta image transmission.

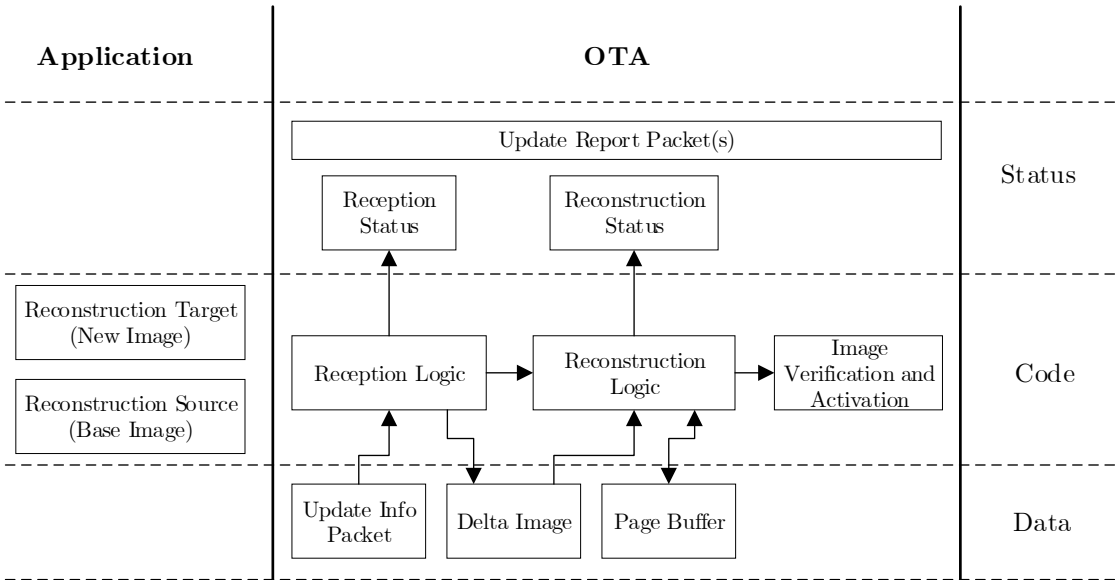

Figure 3.12: Components included in presented OTA Approach.

- Tracking the current *Reception Status*, which contains info whether reception of a delta image is in progress or not. Further information included is to track, what delta image packets are already received and what packets are still missing.

- CRC Check of the delta image when reception is complete.

- Notifying the *Reconstruction Logic* that the *Delta Image* is ready in case the delta image is valid. Otherwise writing the occurred error into the *Update Report Packet(s)*.

The *Reconstruction Logic* uses the *Delta Image* and the old firmware, which is residing in the *Base Image*, to reconstruct the new firmware into the *New Image*. When the *New Image* is positioned in a page-level granularity memory, a *Page Buffer* is required for reconstruction. The *Reconstruction Status* tracks following information:

- Reconstruction currently in progress.

- Page or address that is currently reconstructed.

After finishing the reconstruction process, the logic informs the system about completion.

3.3.2 Failure Analysis

This analysis assumes that the hardware of the MCU has no errors and works as expected. Only possible failures when writing into non-volatile memory are considered. Figure 3.13 shows the possible failure roots divided into software-, hardware- and transmission-errors.

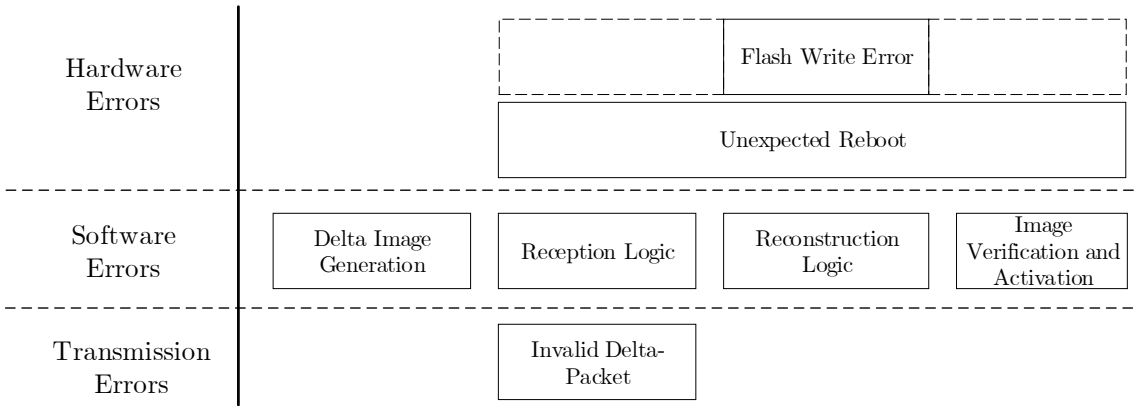

Figure 3.13: Possible Failure Roots in the Update Process.

Software Errors

Software Errors are caused by unknown bugs or wrong handling in the update system. Erroneous *Delta Image Generation* at the host side causes failures at the sensor side as well. Wrong values in the delta image header cause errors in the update process. The following list describes which errors are caused depending on the header field:

- Base Image: *Reconstruction Logic* uses a different base image as the *Delta Image Generation* logic. The reconstructed new firmware will be invalid.
- New Image: *Reconstruction Logic* writes to unintended memory regions. In the worst case, the device gets bricked.
- Message Structure: Invalid decoding of delta messages in the *Reconstruction Logic*. The reconstructed new firmware will be invalid.
- Page Update Order: Depending on the used layout option, a wrong order can lead to a corrupt new firmware due to copy conflicts.
- Delta Image CRC: The update process aborts when the *Reception Logic* calculates the checksum of the delta image.

Invalid delta messages can occur due to errors in the delta generation algorithm or the message encoding logic. The *Reconstruction Logic* creates an invalid new firmware, which is handled in the *Image Verification and Activation* step. The severity of *Software Errors* on the target sensor depends on the used layout option. Implementing countermeasures for the possible failure roots prevents the target sensors from getting bricked when updates are distributed. Intensive testing of the update logic can prevent unknown bugs beforehand.

Hardware Errors

An *Unexpected Reboot* of the target sensor can happen at any stage during the update process. The reliability and resilience of the sensor are highly depending on the used layout option. A high reliability is achieved, when the sensor is able to continue the update process after an *Unexpected Reboot*. A high resilience is achieved, when the sensor

is recovered to a defined state after reboot and updates can be restarted. A *Flash Write Error* prevents that the sensor is able to successfully process any upcoming updates. Depending on the used layout option, the sensor may is able to recover to a defined state and continues execution with the latest valid firmware version available. The erase cycle limitations on flash memories are probably the main cause for *Flash Write Errors*. Thus, the probability of flash errors is reduced when the number of writes to a single page is kept at a minimum when updates are processed.

Transmission Errors

The reception of *Invalid Delta-Packets* should be handled by the communication stack implemented on the sensor. When an invalid packet is still passed to the *Reception Logic*, the update process will be aborted in the delta image verification step.

3.3.3 Analysis of Layout-Options

The used layout option is always a tradeoff between efficiency, reliability and memory cost. This section provides an overview over the layout options *Dual-Image*, *Single-Image* and *External Memory*. A combination of these options is also possible. The *Data* and *Status* components shown in figure 3.12 can either be placed in non-volatile or volatile memory.

Volatile vs. Non-Volatile Memory

Storing update components in non-volatile memory increases the reliability of the update process on the target devices. The process can be continued after an unexpected reboot. Recovery from an unexpected reboot during the delta image reception phase requires to store following data and status components inside non-volatile memory:

- *Update Info Packet*
- *Reception Status*
- *Delta Image Buffer*

In case of an unexpected reboot the transmission of the delta image does not need to be restarted from the beginning. Recovery from an unexpected reboot during the image reconstruction phase requires to store following data and status components inside non-volatile memory:

- *Reconstruction Status*
- *Delta Image Buffer*
- *Page Buffer*

Placing these components into non-volatile memory also has drawbacks: More non-volatile memory is necessary and the complexity of the reception logic and/or the reconstruction logic increases. When using flash memory for example, the erase cycle limitations can lead to *Flash Write Errors*. Thus, a byte-level granularity memory (e.g., EEPROM) should be used for storing data and status components. Another drawback of non-volatile memory usage is the higher amount of energy consumption compared to volatile memory.

Dual-Image

The *Dual-Image* layout splits the internal flash memory into two halves with fixed size, called *Lower Image* and *Upper Image*. This layout has several advantages:

- No additional hardware costs.
- High energy efficiency: Only modified pages inside the target image need to be overwritten.
- High resilience: There is always at least one image with valid firmware available. The reconstruction logic does not allow to overwrite pages inside the currently executing image.
- High flexibility and low complexity: Each component inside the firmware can be updated, including the OTA components.
- Rollback to previous version possible.

The *Dual Image* layout also has several drawbacks:

- Inefficient memory usage. The maximum size of the firmware is only half of the available flash.
- The update flow (Section 4.5) is more complex.
- Requires a MCU with vector table remapping functionality. Some platforms only support vector tables at fixed position. In this case, a software solution for remapping would have to be implemented.

Single-Image

Using the *Single-Image* memory layout for the presented OTA solution is not recommended because bringing the device into an undefined state due to an update cannot be completely avoided. The reconstruction logic must either be loaded into RAM before it's executed, or it has to be separated from the firmware image. This would reduce the flexibility because the reconstruction logic cannot be updated. When the reconstructed image is invalid, the sensor has no possibility to boot an older, valid firmware version.

External Memory

This layout option uses external memory for reconstructing the new image. The *External Memory* layout has following advantages :

- Efficient usage of internal memory. The firmware image can use the whole flash.
- High resilience.
- High flexibility and low complexity.

The *External Memory* layout also has several drawbacks:

- Additional hardware cost due to external memory.
- Lower energy efficiency: The new version is reconstructed on the external memory. After reconstruction, the content of the external memory is loaded into the internal flash. Compared to the *Dual Image* layout, more write operations on non-volatile memory are required.
- Rollback only possible when multiple versions available on the external memory.

Chapter 4

Design and Implementation

This chapter refines the components of the presented OTA solution and provides a detailed view on the actual implemented system. Following subsections describe the implementation details of the different components illustrated in Figure 3.1. Section 4.3 presents the solution for the "Page Ordering Problem", which occurs in page-level granularity memory. Section 4.4 presents the used memory layout for the sensors. This layout increases the complexity of generating delta images and building different firmware versions. Thus, in Section 4.5 the "Update Flow" is presented.

4.1 Link-Time Optimizations

This section describes the implementation of the similarity improvement concepts presented in Section 3.1. Section 4.1.1 provides details about the implemented major-placement algorithm. Section 4.1.2 provides details about the developed minor-placement algorithm. Figure 4.1 provides an overview of the implemented LTO solution. The presented implementation requires two types of inputs:

- *Object Files*: Relocatable ELF-Files generated by compiler.
- *Map-File*: Generated by the linker together with the executable ELF-File.

The *Map-File* provides information, which sections are used and at which position they are placed in the final executable. The *Object Files* provide information about dependencies between sections. The only platform-dependent components are the *Memory-Map Parser* and the *Linker Script Writer*. The rest of the LTO implementation is platform independent, provided that the compiler of the target platform generates the relocatable code using the ELF format. The *Linker Script Writer* generates the placement order of sections and holes (*Slop Regions*), which can be interpreted by the platform-specific linker. After inserting the output of the *Linker Script Writer* into the linker script, the firmware needs to be linked again in order to generate an optimized major- or minor-version. The required input for the LTO implementation is depending on the used placement strategy. Building a major version only requires the input files (*Map-File*, *Object Files*) from the new version. A minor version requires the input files from the new and the old version. The FUM provides the necessary input files of the old version. Generating a firmware

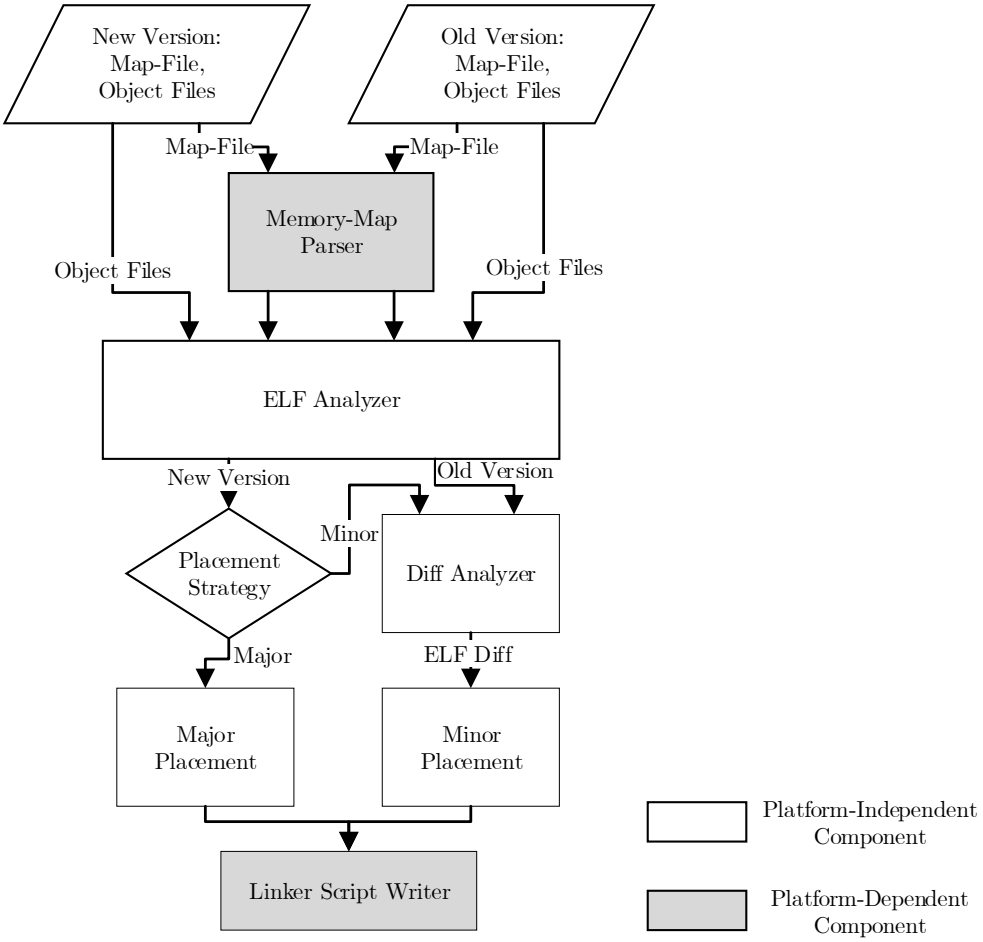

Figure 4.1: Link-Time Optimizations Overview (Similarity Improvements).

with optimized placement, requires to initially build the new version without LTO. This is necessary to create the required input for the LTO algorithm.

The *ELF Analyzer* determines the dependencies between sections and assigns them to their corresponding objects and modules. By using these dependencies, the placement metric ($place_{metric}$) can be calculated. The *Diff Analyzer* compares the firmware versions by using the data provided by the *ELF Analyzer*. The determined new, modified and removed sections are passed to the *Minor Placement* algorithm, which is further described in Section 4.1.2.

4.1.1 Major-Placement Algorithm

The *Major Placement* algorithm generates the section placement order for major versions. It uses the calculated placement metric and following input parameters for ordering the sections:

- *group_objects*: Setting this flag enables grouping of sections into objects.
- *group_modules*: Enables grouping of sections and/or objects into modules.

- *interspacing*: Defines the size of holes (slop regions) inserted between objects and/or modules.
- *memory_roi*: Defines which memory directives (FLASH, RAM, etc.) and corresponding output sections (.text, .const, .bss, etc.) inside these directives should be used for auto-placement.

The placement metric calculation of modules does not include references occurring inside the module. Only references across different modules are used for calculation. The same pattern is used for objects and sections respectively. Figure 4.2 shows an example how the placement metric is calculated for sections, objects and modules. For example *Section 4* has a reference pointing to *Section 1*. This reference raises the ref_{from} counter in *Section 1* and the ref_{to} counter in *Section 4* by one. *Section 1* could be a function that is called in *Section 4*. It could also be a constant that is used in *Section 4*. The calculation of the placement metric is not depending on the section type (code, constant, global variable, etc.). A section can also have multiple references to another section. In Figure 4.2 for example, *Section 5* has two references pointing to *Section 3*. The placement metric for objects/modules excludes references that occur inside the same object/module:

- The placement metric calculation of *Object C* excludes the reference from *Section 6* to *Section 5*.
- The placement metric calculation of *Module A* excludes the references from *Section 4* to *Section 3*, *Section 3* to *Section 1* and *Section 4* to *Section 1*.

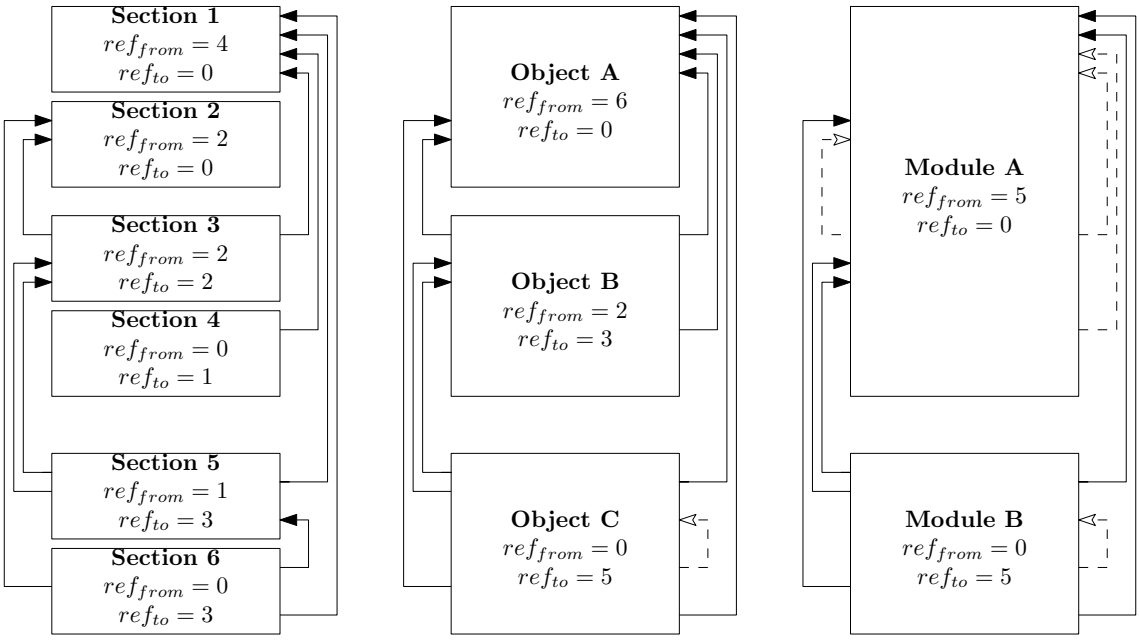

Figure 4.2: Placement Metric calculation using the directed Dependencies between Sections.

The calculated placement metric values define the placement order of each module in the firmware. *Module 1* in Figure 3.4, has the highest placement metric, *Module n*

has the lowest placement metric. Inserting *Slop Regions* after each module increases the probability that sections of a single module remain in a contiguous area on flash when updates are performed. The placement metric of the objects defines the placement order of the objects inside their corresponding modules. The placement metric for sections defines their order inside the object they belong to.

4.1.2 Minor-Placement Algorithm

Figure 4.3 shows the basic flow of the implemented minor-version placement algorithm. Processing of removed and modified sections is already described in Section 3.1.3. The

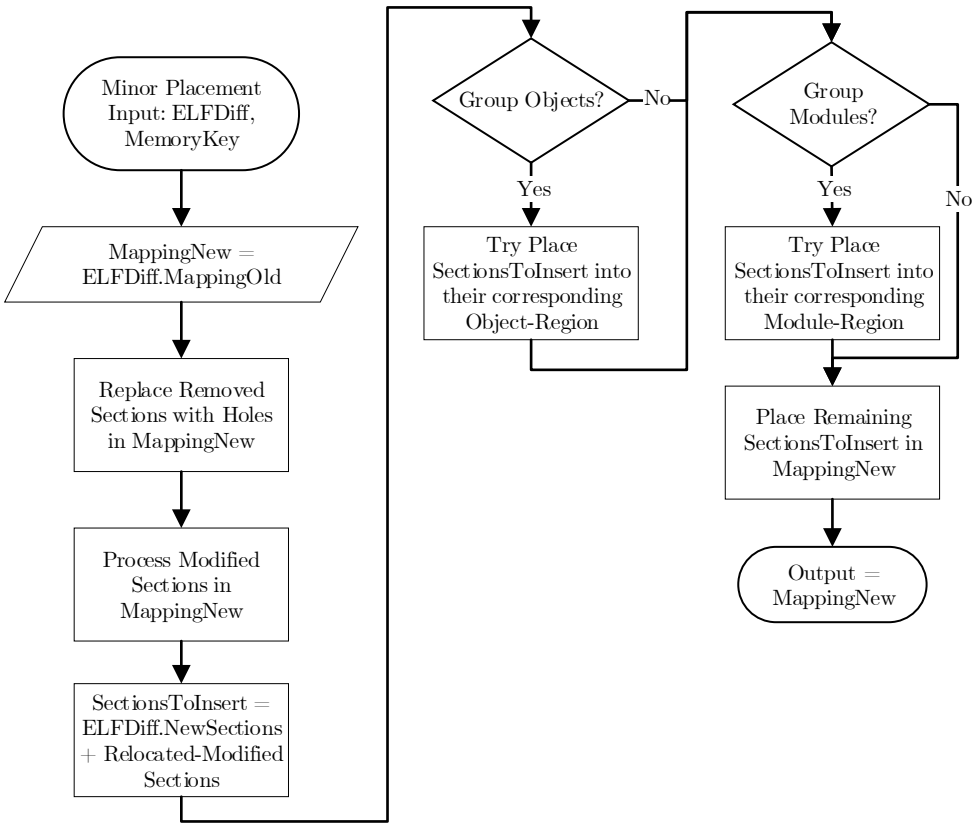

Figure 4.3: Minor-Placement Algorithm Overview.

remaining sections to insert (*SectionsToInsert*) consist of *New Sections* and *Relocated-Modified Sections*. The next step is to determine whether the sections should be grouped into objects or not. This parameter (*group_objects*) is set when the initial major version is created. When the sections are grouped into objects, the next step is to iterate over each existing object and determine the corresponding region inside the firmware. The intended region of each object is saved after generating a major version. For each object, Algorithm 1 is executed. This algorithm tries to efficiently insert the remaining sections, which belong to the corresponding object, into the defined region. When grouping of objects is disabled, or sections to insert remain, the same approach is applied by iterating

over each existing module. The corresponding region of each module is determined and Algorithm 1 is executed for each module. In this case the algorithm tries to efficiently insert the remaining sections into the module region they belong to. When grouping of modules is disabled and/or sections to insert remain, Algorithm 1 is called again with the whole image as valid region for inserting the remaining sections.

Algorithm 1 Try Place SectionsToInsert into given Region

```
1:  procedure TryPlaceInRegion(SectionsToInsert, MappingNew, Region)
2:      SectionsInserted ← 1
3:      while length(SectionsToInsert) > 0 and SectionsInserted > 0 do
4:          SectionsInRegion ← GetSectionsInRegion(SectionsToInsert, Region)
5:          HolesInRegion ← GetHolesInRegion(MappingNew, Region)
6:          SectionsInserted ← 0
7:          for Section in SectionsInRegion do
8:              for Hole in HolesInRegion do
9:                  if Section.Length ≤ Hole.Length then
10:                     Margin = Hole.Length − Section.Length
11:                     Hole.Candidates.add(Tuple(Section, Margin))
12:                     Section.Candidates.add(Tuple(Hole, Margin))
13:         for Section in SectionsInRegion do
14:             Section.Candidates.sort(by ← Margin, mode ← Ascending)
15:         for Hole in HolesInRegion do
16:             Hole.Candidates.sort(by ← Margin, mode ← Ascending)
17:         for Section in SectionsInRegion do
18:             BestHoleCandidate ← Section.Candidates[0].Hole
19:             if BestHoleCandidate.Candidates[0].Section == Section then
20:                 IndexToInsert ← MappingNew.IndexOf(BestHoleCandidate)
21:                 MappingNew.InsertAt(IndexToInsert, Section)
22:                 Margin ← BestHoleCandidate.Candidates[0].Margin
23:                 BestHoleCandidate.Length ← Margin
24:                 if BestHoleCandidate.Length == 0 then
25:                     MappingNew.Remove(BestHoleCandidate)
26:                 SectionsToInsert.Remove(Section)
27:                 SectionsInserted ← SectionsInserted + 1
28:     return [MappingNew, SectionsToInsert]
```

4.2 Delta Message Encoding

The presented encoding approach groups the different message types together as shown in Figure 3.9. The order of messages inside their groups is depending on the target memory granularity. For byte-level memories, the message order can be defined arbitrarily. Page-level memories require a strict message order. The presented encoding approach is optimized for page-level memories, byte-level memories with custom message order would require different encoding for certain delta message fields.

4.2.1 COPY Messages

COPY messages are ordered ascending by their absolute destination address *dest_abs*. Each message contains following information:

- *length_copy*: Number of bytes to copy, the absolute value is encoded as unsigned LEB128.
- *dest_copy*: Destination start address of COPY message. The address is encoded relative as unsigned LEB128.

$$\begin{aligned} dest\_copy_i &= dest\_abs_i - dest\_abs_{i-1}, \\ dest\_copy_1 &= dest\_abs_1 - new_{absolute} \end{aligned} \tag{4.1}$$

  $dest\_copy_1$ is the destination start address of the first COPY message in the delta image.

- *move_dist*: Move distance of COPY message, encoded relative as signed LEB128.

$$\begin{aligned} src\_rel &= src\_abs - base_{absolute}, \\ dest\_rel &= dest\_abs - new_{absolute}, \\ move\_dist &= dest\_rel - src\_rel \end{aligned} \tag{4.2}$$

Figure 4.4 shows that relative destination start addresses lead to smaller numbers to encode. LEB128 encodes small numbers with fewer bytes, the *dest_copy* field inside the COPY message needs less bytes for encoding. The defined COPY message order additionally enables to use unsigned LEB128 for encoding *dest_copy*. The calculation of *dest_copy*,

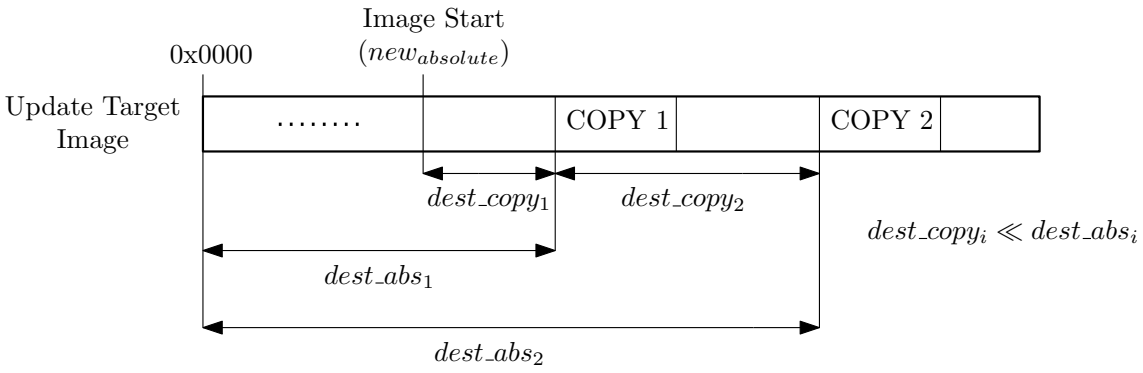

Figure 4.4: Improved Encoding Efficiency when using Relative Target Addresses.

as shown in Equation 4.1, also allows to encode nested COPY messages. Nested COPY messages only bring benefits when the messages fully overlap (see Figure 4.5). Furthermore, is the generation of nested COPY messages a non trivial task for a delta generation algorithm. The DGO algorithm [Ast19] does not support them. When the delta algorithm creates no nested COPY messages, *dest_copy* can be calculated even more efficiently using following formulas:

$$\begin{aligned} dest\_copy_i &= dest\_abs_i - dest\_abs_{i-1} - length\_copy_{i-1}, \\ dest\_copy_1 &= dest\_abs_1 - new_{absolute} \end{aligned} \tag{4.3}$$

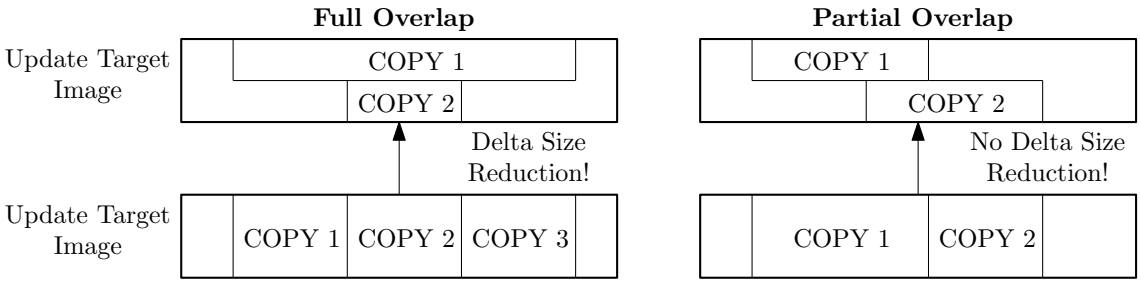

Figure 4.5: Nested Copy Messages. Efficiency Improvement only when Messages fully Overlap.

The *move_dist* efficiently encodes the source start address of a COPY message. The value is independent of the used images for performing updates. Another advantage of *move_dist* is the small resulting value, which can be efficiently encoded using signed LEB128. Signed encoding is necessary because COPY operations are possible in both directions. The high probability for small values of *move_dist* can be explained with the similarity improvement approach presented in Section 2.4.1: COPY messages are mainly generated due to modified sections. When no relocation is required, *move_dist* is small because the section remains at the same position. In case of relocation, the minor placement strategy tries to place the section inside the modules region. The distance to the previous location of the section is kept at a minimum.

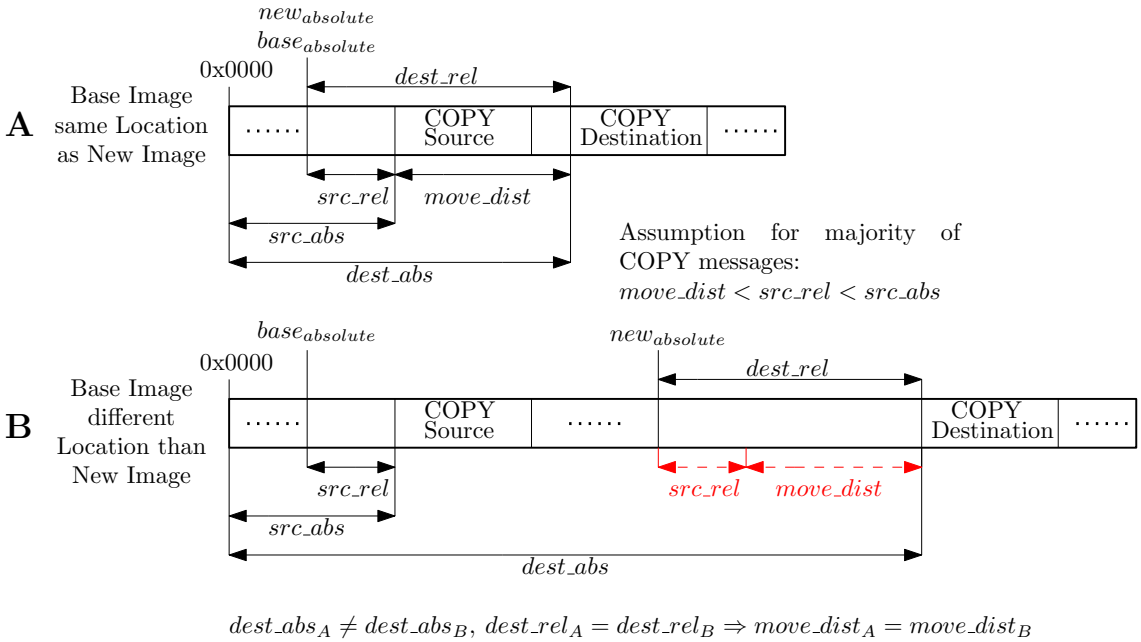

Figure 4.6: Efficient Encoding of Copy Source Address by using the Move Distance.

4.2.2 ADD Messages

ADD messages are ordered ascending by their absolute destination address *dest_abs*. Each message contains following information:

- *length_add*: Number of bytes to add, the absolute value is encoded as unsigned LEB128.
- *dest_add*: Destination start address of ADD message. The address is encoded relative as unsigned LEB128:

$$\begin{aligned} dest\_add_i &= dest\_abs_i - dest\_abs_{i-1} - length\_add_{i-1}, \\ dest\_add_1 &= dest\_abs_1 - new_{absolute} \end{aligned} \tag{4.4}$$

  $dest\_add_1$ is the destination start address of the first ADD message in the delta image.
- *data_add*: Array that contains the data to be written into the encoded memory region. *length_add* defines the size of the array.

The encoding approach of *dest_add* has the same benefits as *dest_copy* for COPY messages. Since nested ADD messages don't bring any advantage in terms of delta size, the length of the previous ADD message is included for calculating *dest_add*. This further reduces the value and increases the encoding efficiency.

4.2.3 OFFSET Messages

OFFSET messages are ordered ascending by their offset value. Each offset message contains following information:

- $value\_off$: Offset value of the message. Encoded as difference to the previous offset value as unsigned LEB128. The first OFFSET message encodes the value absolute as signed LEB128.
- add_list_off: List of addresses where offset is applied in the new image. The addresses are ordered ascending and encoded relative as unsigned LEB128. The calculation of address i (add_off_i) inside the address list of size n is performed with following formulas:

$$\begin{aligned} dest\_align_i &= \frac{dest\_abs_i}{arch\_align}, \\ add\_off_i &= dest\_align_i - dest\_align_{i-1}, \\ add\_off_1 &= dest\_align_1 - \frac{new_{absolute}}{arch\_align}, \\ add\_off_{n+1} &= 0 \end{aligned} \tag{4.5}$$

The delta algorithm generates offset messages that are intended to be applied on addresses in code. For example, the offset is applied to the address field of a *call* instruction. The positions of these address fields are usually aligned inside the code. The alignment value

$arch\_align$ depends on the MCUs architecture. A 16-bit MCU aligns addresses to 2, a 32-bit MCU aligns them to 4 for example. In order to further reduce the value of add_off_i, the absolute addresses $dest\_abs_i$ are divided by $arch\_align$. The delta algorithm checks whether found addresses are aligned to $arch\_align$. Unaligned addresses are changed to the next lower, aligned address entry, the offset must be adopted accordingly (left shift). The encoding approach does not need to include the length of the address list. The decoder on the sensor recognizes the end of the list when an address entry is zero (add_off_{n+1}). Except the first address list entry, which can be zero. In this case the offset is applied to address $new_{absolute}$.

4.3 Page Ordering Problem

In page-level granularity memories, the reconstruction logic of the sensor processes the delta messages page-wise. The sensor completes the reconstruction of a single page before the next page is processed. Dependencies between two pages exist when following conditions are true:

- Source and destination region of a COPY message are located in different pages.
- Source region is modified by other delta messages (COPY, ADD, OFFSET).

Figure 4.7 shows how dependencies between pages are found. For given page dependencies, a custom page update order is necessary. This avoids that copy conflicts lead to invalid reconstruction of the new firmware. To find an optimal order, pages are modelled by a directed graph. A vertex represents a page, an edge represents N COPY messages from page A to B that meet the dependency conditions mentioned above. An optimal page update sequence is equivalent to a vertices sequence $(v_{e0}, ..., v_{en})$, where each vertex v_{ei} is a sink in the vertices subsequence $(v_{e0}, ..., v_{ei-1})$. This statement is only true when the graph is cycle-free. Circular dependencies require to remove edges in the directed graph until the graph is cycle-free. Finding the best candidates to remove cycles in the graph is a non-trivial task. This thesis presents an approach that is based on SUSPIRe [LWS18], whereas the method for calculating the weights of edges is modified. The weight w of an edge (A, B) describes the additional bytes needed in order to replace the N COPY messages with a list of messages that remove the dependency from page A to B. A simple approach is to replace the "Copy Conflict Regions" (Figure 4.7) of the N COPY messages with M ADD messages. This replacement removes the dependency of page A to B. The copied areas, which cause no conflicts, can be replaced by L smaller COPY messages. Equation 4.6 shows the calculation of the weight w:

$$\begin{aligned} w &= \sum_{i=1}^{N} rem\_cost_i - \beta_i, \\ rem\_cost_i &= \sum_{k=1}^{M} (\alpha_k + length\_add_k) + \sum_{j=1}^{L} \beta_j \end{aligned} \tag{4.6}$$

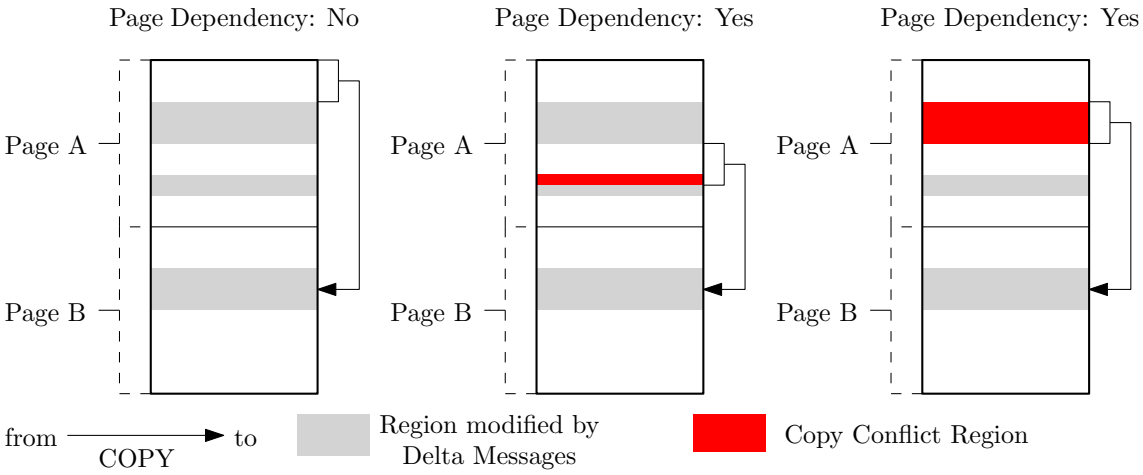

Figure 4.7: Finding Dependencies between Pages.

where:
- N = number of COPY messages that meet dependency conditions from page A to B
- β_i = header size of COPY message i
- rem_cost_i = number of bytes required to remove the dependency caused by COPY message i
- M = number of ADD messages required to remove the dependency caused by COPY message i
- α_k = header size of ADD message k, $\alpha_k = 0$ when ADD message can be merged into preceding or following ADD message.
- $length\_add_k$ = number of bytes to add by message k
- L = number of COPY messages required to remove the dependency caused by COPY message i
- β_j = header size of COPY message j
- $length\_copy_i$ = number of bytes copied by message i from page A to page B

The sum of delta messages, used to replace COPY message i, have to process the same number of bytes as COPY message i. This is shown in Equation 4.7.

$$length\_copy_i = \sum_{k=1}^{M} length\_add_k + \sum_{j=1}^{L} length\_copy_j \tag{4.7}$$

4.4 Dual-Image Layout

The target platform, where the update solution should be deployed, has no external memory available. The internal memory provides sufficient space for using the dual-image layout. Achieving high resilience and reliability for dual-image layout is much simpler than for single-image layout. Figure 4.8 shows three different dual-image layout options:

- **Option 1**: Provides high efficiency because update status and data components are stored in RAM. Provides low reliability because retransmission of delta image is

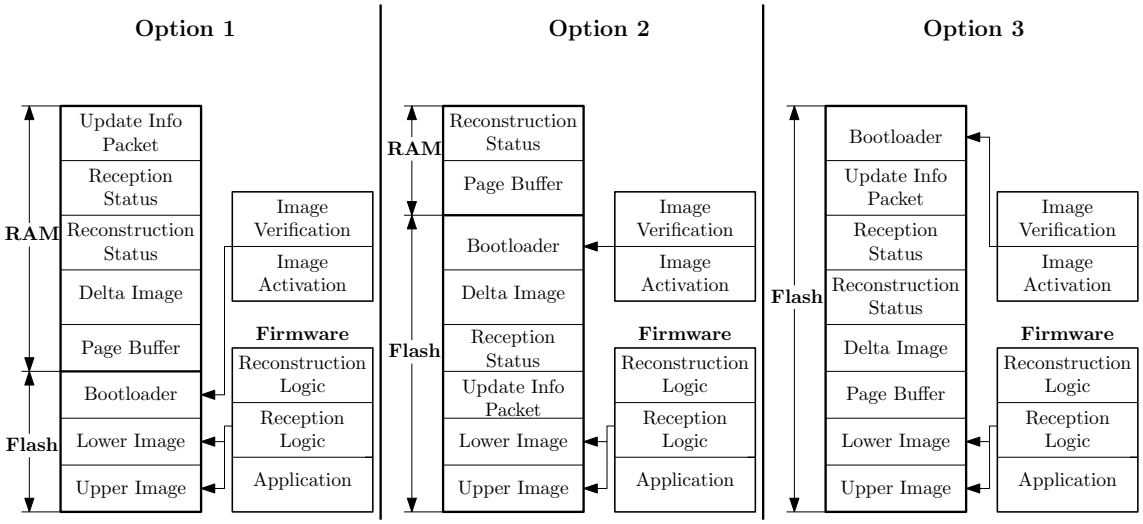

Figure 4.8: Dual-Image Layout Options.

required in case of an unexpected reboot during the reception phase. Additionally, an unexpected reboot during the reconstruction phase leads to a corrupt target image (*Lower Image* or *Upper Image*). In this case, a delta image that uses a base image with defined state must be generated at the server and transmitted to the sensor for a successful update (further described in Section 4.5)

- **Option 2**: Provides lower efficiency than *Option 1*, but higher reliability. The data and status components, required for the reception logic, are placed inside the non-volatile memory. The device is able to continue the delta image reception phase after an unexpected reboot. No retransmission of the whole delta image is necessary.

- **Option 3**: Provides lowest efficiency but highest reliability. In case of an unexpected reboot during the delta image reception or the reconstruction phase, the sensor can continue the update process.

The probability for unexpected reboots during the update process is very low. Thus, *Option 1* will be used on the target platform.

4.5 Dual-Image Update Flow

The dual-image memory layout increases the complexity of building firmware images and generating delta images. Figure 4.9 shows the update flow. The sensor can execute code from the *Lower Image* and the *Upper Image*. Before rollout, the initial firmware version is built for both images. A firmware update can be performed with two different delta images:

- **Option A**: The base image and the target image are the same. In most cases, this option provides smaller delta files. The drawback is that copy conflicts are possible (Section 4.3).

- **Option B**: The base image and the target image are different. The advantage is that no copy conflicts are possible.

The presented OTA solution always uses the option that generates the smaller delta image. When the update leads to a corrupt image, the delta generated with *Option B* is transmitted repeatedly to the device until the update is succesful. When using a MCU that supports dual bank memory [STM16], the firmware images built for the *Lower Image* and the *Update Image* are equal (e.g.: v1L == v1U). In this case, the update flow is simplified. Both options, which are shown in Figure 4.9, generate the same delta image.

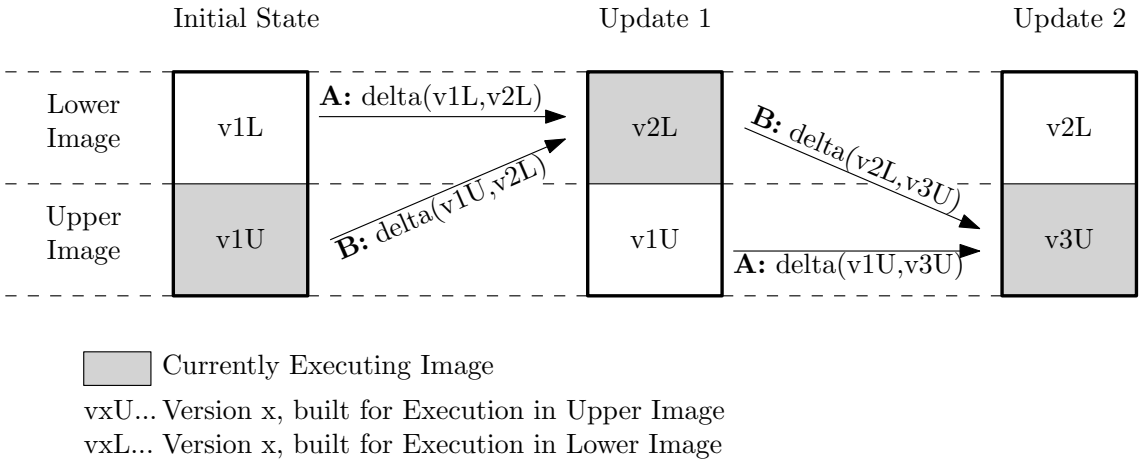

Figure 4.9: Dual-Image Update Flow. Two possible Options for generating Delta Images.

4.6 Delta Image Receipton

The received packets, containing the *Delta Image Fragments*, have a fixed size. The packet size is given via the update info packet (Section 3.2.2). Figure 4.10 shows the handling of a received delta image packet. The *Delta Image Buffer* is a static array located in the RAM. Due to the fixed packet size, the reception logic can store each *Delta Image Fragment* to the correct position in the buffer instantly. When packets are missed, the rest of the received fragments are still stored at the correct position. The reception logic determines the start position for the received fragment with following equation: $(frame_{nr} - 1) \cdot size_{packet}$. Another advantage of the fixed packet size is the simple handling of missed packets. The reception logic tracks the received packets with a *Lookup Table*. The reception of a packet leads to setting a bit at position $frame_{nr}$. Missing packets can be easily identified with the *Lookup Table* by looking for zero bits. Depending on the transmission approach, the sensor can act accordingly to receive the missing packets. For example, sending a retransmission request frame to the gateway, which contains the missing frame numbers. The reception logic determines the completion of the reception by checking if every bit position in the *Lookup Table* is set. After completion, the reception logic checks if the delta image in the buffer is valid (Section 4.6.1). Using static arrays for the *Delta Image Buffer* has several drawbacks compared to dynamic memories:

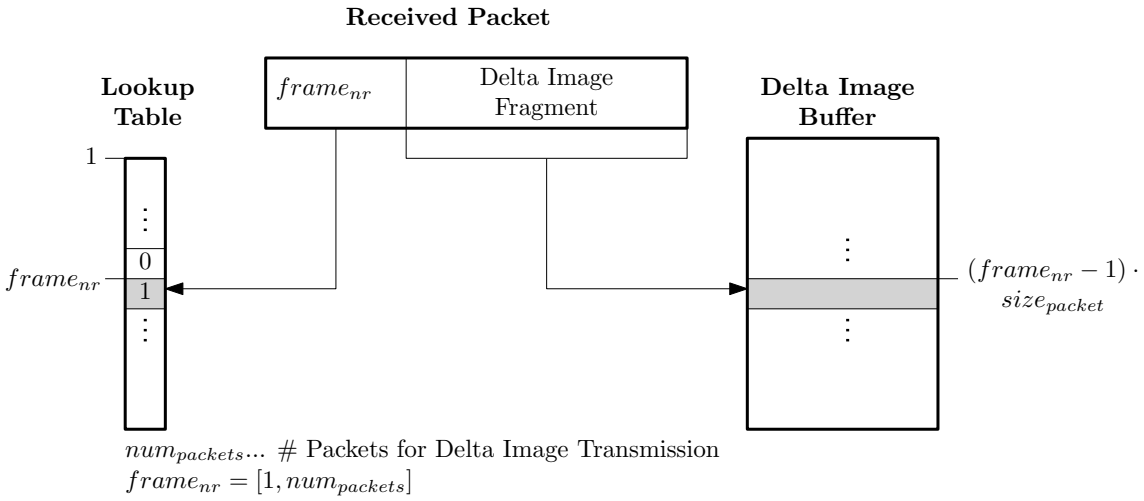

Figure 4.10: Handling received Delta Image Packet.

- A maximum delta image size ($size\_delta_{max}$) for future updates must be defined at compile time.

- When a delta image exceeds $size\_delta_{max}$, the update cannot be processed. The user has to split the desired firmware update into smaller sub-updates.

- A minimum packet size ($size\_packet_{min}$) must be defined at compile time. The maximum size of the *Lookup Table* ($size\_lookup_{max}$) is depending on $size\_delta_{max}$ and $size\_packet_{min}$. The *Lookup Table* is also stored as static array, thus $size\_lookup_{max}$ defines the size of the array at compile time. The calculation of the array size for the *Lookup Table* is shown in Equation 4.8. Each table entry requires a single bit, $size\_lookup_{max}$ is provided as number of bytes.

$$size\_lookup_{max} = \lceil \frac{size\_delta_{max}}{size\_packet_{min}} \cdot \frac{1}{8} \rceil \tag{4.8}$$

- The *Delta Image Buffer* occupies a lot of RAM that cannot be used by the firmware for any other purpose. Defining $size\_delta_{max}$ is a tradeoff between the avoidance of performing sub-updates and preserving sufficient RAM for the firmware.

Placing the *Delta Image Buffer* inside the dynamic memory (heap) would allow the firmware to use more memory during normal operation mode. In this case, the maximum size of the heap must be set according to $size\_delta_{max}$ at link time. The reserved heap cannot be used for any other components in RAM, such as stack or global variables. But in contrast to the proposed implementation, the heap required for the *Delta Image Buffer* could be used for other dynamic structures that are needed during normal operation. The major problem when using dynamic memory for image reception is that the guarantee for sufficient memory at runtime is very hard to achieve. The update system becomes more error-prone and complex.

4.6.1 Delta Image Verification

After completing the reception of the delta image, the sensor executes the verification step. Figure 4.11 shows how the sensor processes this verification. The reference checksum (CRC) resides at the beginning of the delta image header and has fixed length. Thus, the checksum is located at the beginning of the *Delta Image Buffer*. The sensor excludes the region of the reference CRC for the checksum calculation of the received delta image. The delta image size is usually smaller than the maximum size ($size\_delta_{max}$) of the buffer. Thus, the sensor must only use the region that is filled with the delta image for calculating the checksum. Otherwise, the checksums wouldn't match because the unused areas (usually filled with zeros) would change the result of the calculation. The region for the checksum calculation is determined with the *Update Size* field inside the *Update Info Packet*. When the reference CRC and the calculated CRC are equal, the sensor starts with the image reconstruction step. Otherwise the CRC missmatch is reported to the gateway and the sensor aborts the update process.

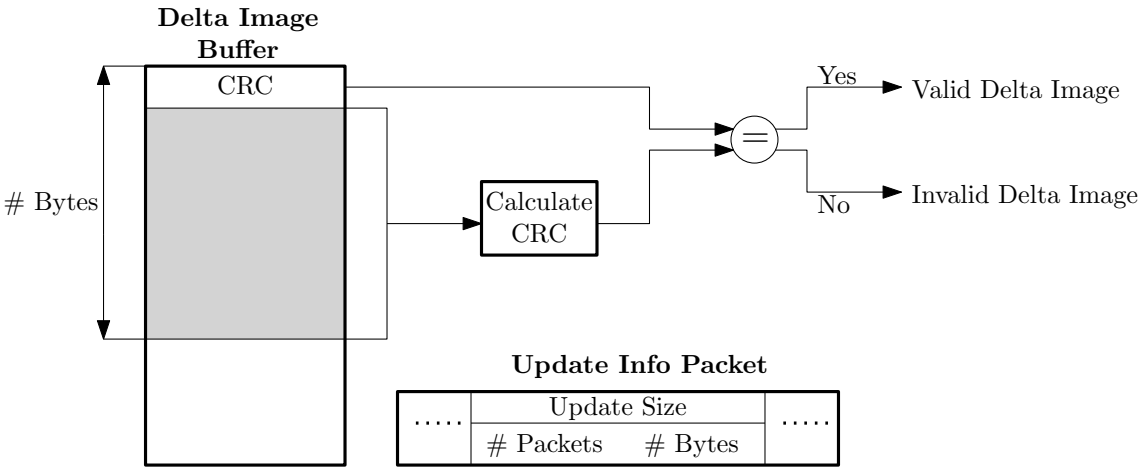

Figure 4.11: Checksum Calculation for Delta Image Verification.

4.7 Image Reconstruction

The sensor starts reconstructing the new image after verification of the delta image. The implemented reconstruction logic supports dual-image memory layouts where both images (Upper Image, Lower Image) are executable by the MCU. Placing one of the images on external memory is not possible on most platforms, because code cannot be directly executed from there. The logic currently supports two reconstruction approaches:

1. *Base Image = Executing Image, New Image = Non-Executing Image*: No copy conflicts possible, this approach can be repeated as often as necessary in case the update process fails.

2. *Base Image = Non-Executing Image, New Image = Non-Executing Image*: Copy conflicts are possible. When update process fails during the first try, approach 1

must be used because the *Non-Executing Image* results in an undefined state. The main advantage compared to approach 1 is the smaller delta file because it's created with firmwares built for the same memory region (same image).

Another possible approach is to generate the delta using firmwares that are both built for execution in the currently *Executing Image*. The reconstruction is equal to approach 1. After reconstruction, the bootloader copies the whole *New Image* into the *Base Image* because the new firmware is built for execution in the previously *Executing Image*. This approach would generate small deltas and could be repeated as often as necessary. The major disadvantage of this approach is the higher reconstruction cost due to the higher amount of pages overwritten. When the target platform supports dual-bank memory [STM16], this approach would not require the bootloader to copy the whole *New Image* into the *Base Image*. Dual-bank memories enable to map two images in the internal memory to the same address space. This MCU feature simplifies the whole update process. The generated firmwares are always positioned at the same memory region no matter which image is used as reconstruction target. Figure 4.12 shows the different steps of the

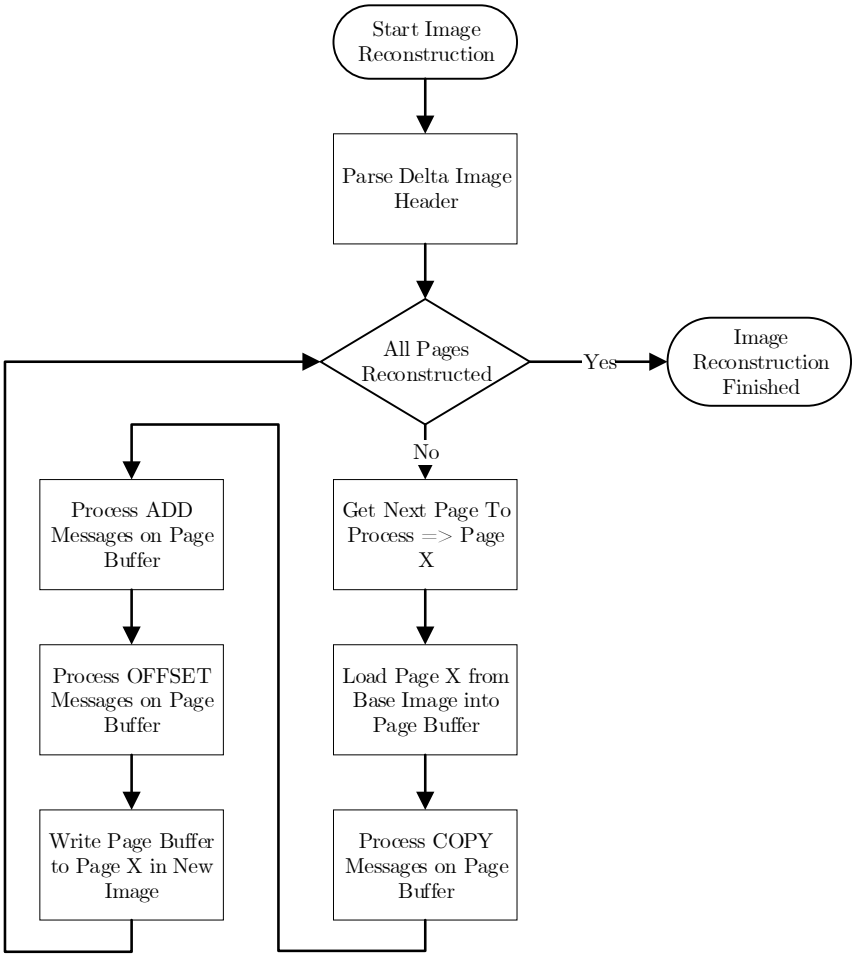

Figure 4.12: Reconstruction Logic Flowchart.

reconstruction logic. The delta image header contains information necessary for processing the reconstruction. Thus, parsing the header is the first step of the reconstruction logic. Since the implementation is intended to be used for flash memories, the reconstruction is processed page-wise. The page update order is included in the delta image header, it tells the reconstruction logic what page is the next to be reconstructed. The next step is to load the current page from the *Base Image* into the *Page Buffer*. Afterwards, the delta messages, which intersect with the current page, are applied to the page buffer. The last step is to write the processed page from the *Page Buffer* into the *New Image*.

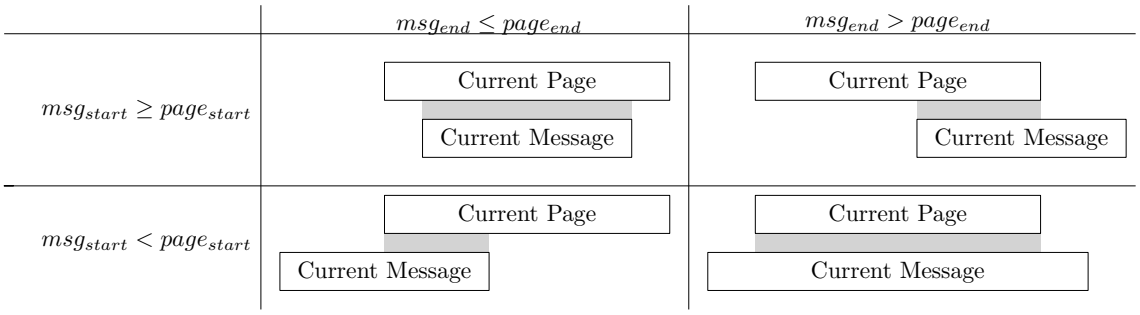

Figure 4.13: Processing Delta Messages on Single Flash-Page.

Delta messages ignore page boundaries. Thus, the reconstruction logic is responsible for processing intersections of pages and messages only. Figure 4.13 shows the possible intersections of pages with delta messages. Processing the delta messages on the current page is done block-wise per message type. Due to the block size infos in the delta image header, the beginning of each message block can be calculated without decoding all the delta messages. Due to the presented encoding scheme (Section 4.2), forward-decoding is necessary to find intersecting messages for the current page. For each page to reconstruct, the decoding starts from the beginning of each message block. The COPY and ADD message block is decoded until $msg_{start} \geq page_{end}$. When this condition is true, decoding of the next message block for the current page is started immediately because COPY and ADD messages are sorted ascending by their destination address $dest_{abs}$. No further message inside the block can intersect with the current page. The OFFSET block has to be decoded completely for each page due to the encoding scheme. An alternative to repetitive decoding for each page would be to decode the delta messages only once and store them in an additional buffer. The problem is that this would require a lot of additional memory for the delta image.

4.8 Bootloader Implementation

The bootloader is responsible for checking and activating the firmware images residing in the lower and upper image on the sensor's flash memory. Figure 4.14 shows the layout

of a firmware image. The *Interrupt Vector Table* resides at the beginning of the firmware image. Every firmware image needs it's own vector table. Thus, the used MCU must support relocatable vector tables in order to execute code in the lower and upper image. The bootloader reads the *Image Header* of both images at the beginning for comparison of the *Version Numbers*. The header is located at a fixed position in both images. The checksum, generated at the host side, is located at the end address img_{end} of the firmware. Figure 4.15 shows the steps performed by the bootloader. The implementation of the

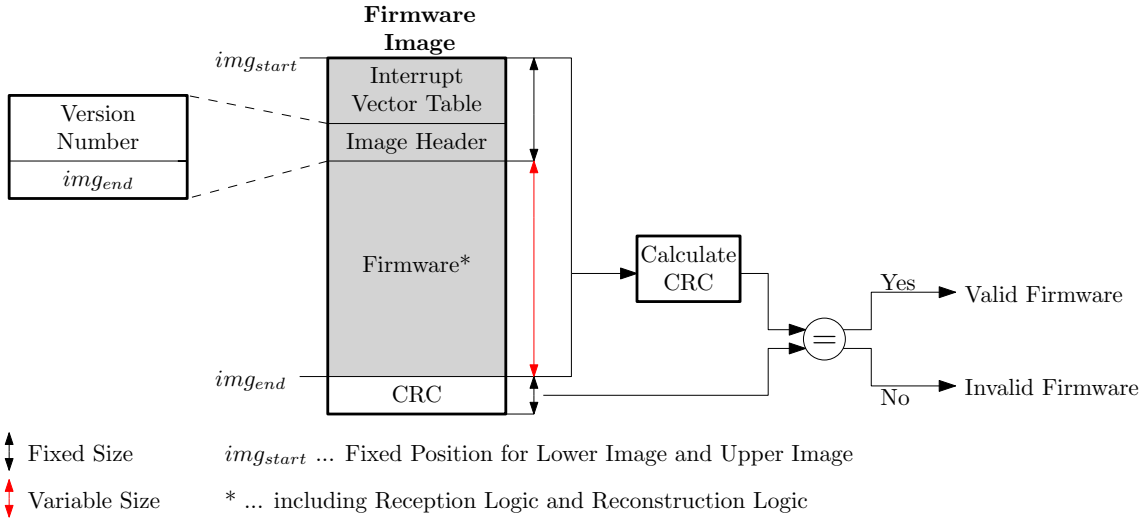

Figure 4.14: Firmware Image Layout.

different steps is depending on the used MCU: When there is no hardware unit available for CRC calculation, a software implementation inside the bootloader logic is necessary. Setting the *Vector Table Entry*, *Stack Pointer* and *Program Counter* is also depending on the used hardware.

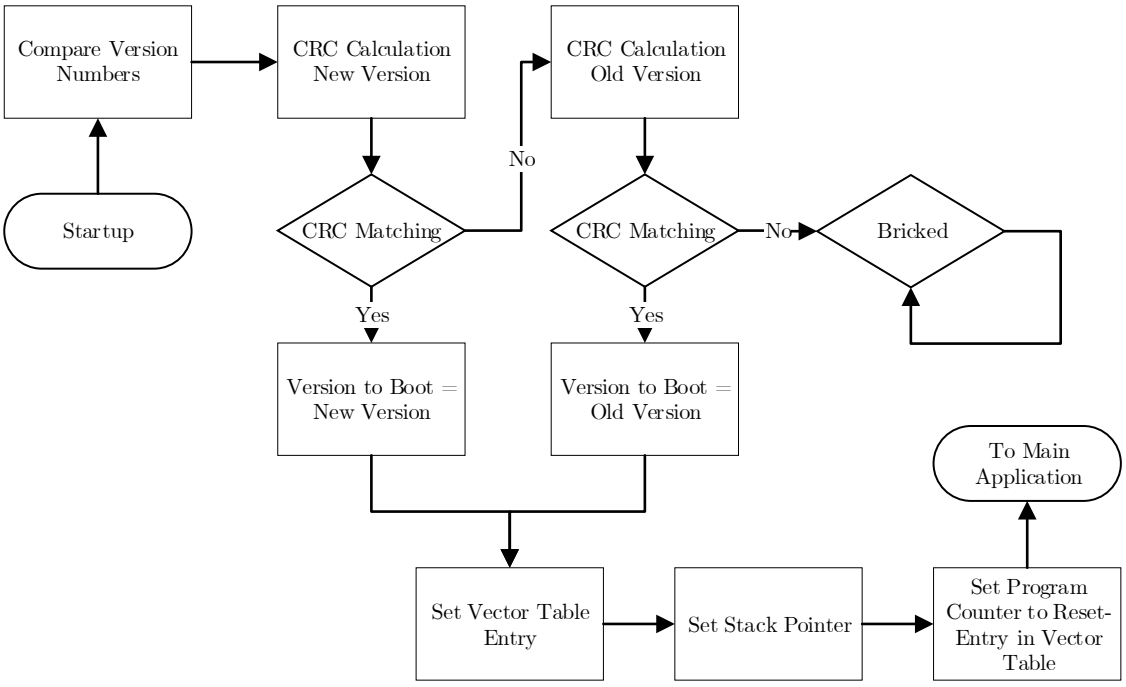

Figure 4.15: Bootloader Logic Flowchart.

Chapter 5

Results

This chapter evaluates the efficiency of the incremental update concept presented in this thesis. Section 5.1 introduces the simulation setup. Section 5.2 evaluates the *Similarity Improvements* approach presented in section 2.4.1. The evaluation presents among others the significant delta size reduction and drawbacks due to fragmentation of memory.

5.1 Simulation Setup

This section introduces the simulation setup used for evaluating the presented concepts in this thesis. In order to automate the firmware building and delta generation process, a simulation framework was developed, which is presented in Section 5.1.1. Section 5.1.2 introduces the firmware used for evaluating the performance of the presented incremental update solution.

5.1.1 Simulation Framework

Figure 5.1 shows an overview of the developed simulation framework. This framework enables to automatically build firmware versions for desired LTO configurations and to generate delta images for various DGO configurations. The configuration file enables to define the desired build-runs and diff-runs. Each build-run defines how to build the given firmware. Following parameters need to be set for each build-run:

- *key*: Unique name of the build-run. Required to identify each build-run in the evaluation step.
- *enable_autoplacement*: Defines whether LTO is used for the build-run or not. When LTO is disabled, the built firmware contains the standard placement defined by the linker of the platform.
- *versions*: A list that defines the versions to generate for the given build-run. Each list entry contains following parameters:
  - *placement_strategy*: Defines the placement strategy of the LTO logic (major or minor).

  - *version_old*: Version number of the old firmware. Only required when creating the new version using the minor placement strategy.
  - *version_new*: Version number of the new firmware.

- LTO configuration parameters: *group_objects*, *group_modules*, *interspacing* and *memory_roi*. Further described in Section 4.1.

Each diff-run contains following parameters:

- *build_run*: The target build-run to generate delta images. The delta generator uses the built firmwares of the build-run to create delta-images.
- *versions*: A list that defines the used versions for creating delta images. Each list entry contains following parameters:
  - *version_old*: Version number of the old firmware.
  - *version_new*: Version number of the new firmware.
- *config_list*: List of configuration parameters used for the DGO algorithm.

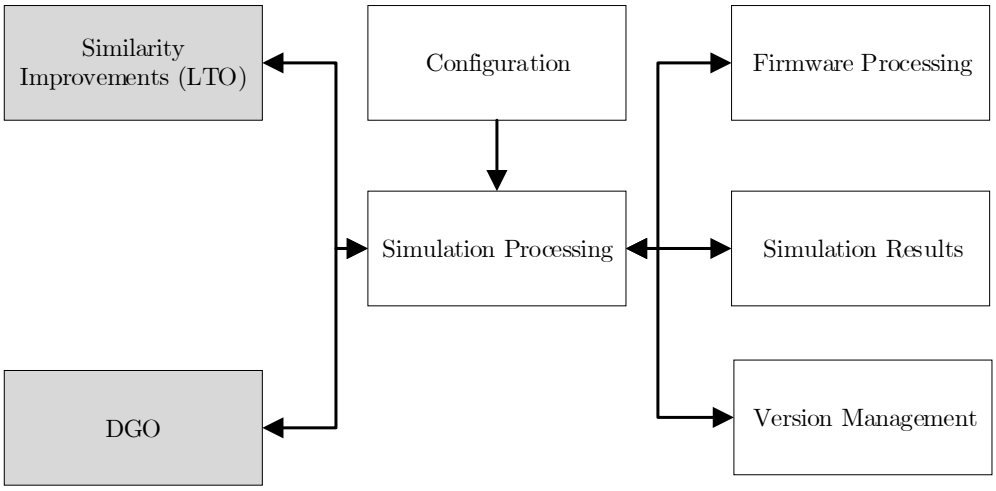

Figure 5.1: Simulation Setup used for Evaluation.

Each diff-run creates a delta image for each version list entry and diff-config entry combination. Each version-number entry in the build-run configuration exists as tag in the firmware repository. The *Version Management* component is responsible for checking out the desired version in the firmware repository. The *Firmware Processing* component builds the desired firmware version. Simulations are executed by the *Simulation Processing* component. This component generates the firmware versions as described in the build-runs and generates the delta images as described in the diff-runs. The *Simulation Results* component is used to store all necessary information for evaluating the developed LTO and DGO implementations. All results presented in this chapter were generated using the simulation framework.

5.1.2 Used Firmware

This section introduces the firmware used for evaluating the developed LTO and DGO. The firmware is executed on a sensor that was developed by the company smaXtec animal care [sacG]. Its purpose is to measure the movement activity and the temperature inside the rumen of a cow. The firmware is running on a CC430F5137 MCU [Tex13a] and contains several modules:

- System Libraries: Functionality provided by the manufacturer. Contains code such as initialization logic of .bss and .data sections for bare metal firmware, standard libraries (memcopy, memset,...) and others.
- Com Stack: Communication protocol for narrowband radio device.
- Measure: Measurement logic and drivers for peripherals.
- Application Logic: Contains logic for compression of measurement data, buffering of measurement data, data interpreters, sensor activation logic, state machines and others.

For build-runs, where module grouping is enabled, the developed LTO solution groups the sections of the firmware into the modules mentioned above. Figure 5.2 shows the

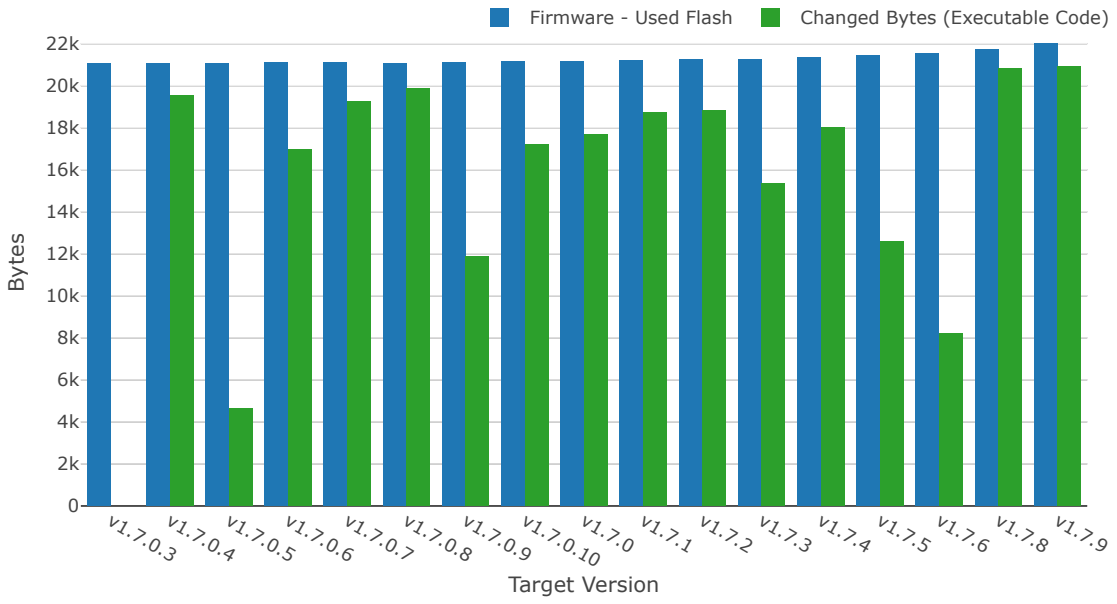

Figure 5.2: Flash Usage and Changed Bytes of Firmware Versions used for Evaluation.

flash usage and changed bytes of the different firmware versions used for evaluation. This figure should help to understand the amount of change between different firmware versions. Based on the amount of change, the performance of the presented LTO approach can be evaluated more accurate. The parameter *Changed Bytes* describes the total number of non-matching bytes between two firmware versions. The publication, which presents the existing Delta Generator [KB16a], also uses this parameter for describing the amount of

change between different versions. The *Changed Bytes* parameter can be misleading. The number of changed bytes is very high for almost every version. Figure 5.3 shows that the high number of changed bytes is not caused by a high amount of firmware changes. The number of changed bytes is high because a lot of sections get shifted inside the firmware. Figure 5.3 shows the firmware changes before the linking step. Analysing firmware changes

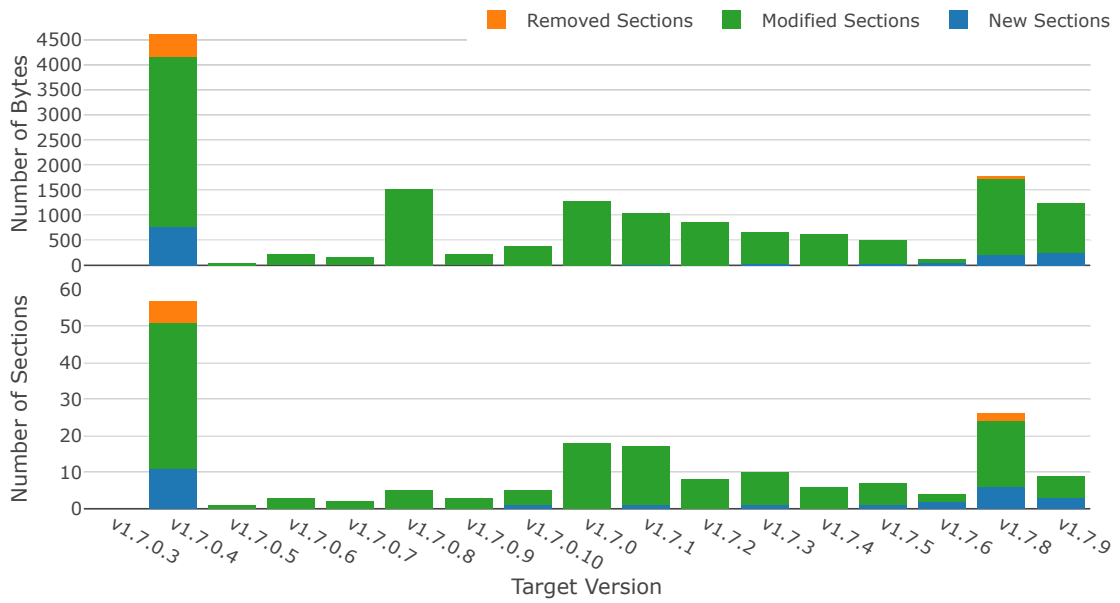

Figure 5.3: Firmware Changes before Linking Step.

before linking step represents the actual amount of change more accurate. The firmware is compared per section and the final position of the section in the firmware has no influence on the resulting amount of firmware changes. The information provided in Figure 5.3 is extracted with the developed *Diff Analyser* (Section 4.1). The actual amount of firmware changes is much smaller than the *Changed Bytes* parameter.

5.2 Results including Link-Time Optimizations

Figure 5.4 compares the achieved delta size of the DGO delta algorithm [Ast19] with the firmware changes before the linking step. Even though the DGO achieves a higher compression than the Delta Generator [KB16a], still much more data must be transmitted to the target nodes than actually changed inside the firmware image. This section presents the results including LTO. Section 5.2.1 evaluates how efficient the presented similarity concept reduces address shifts. Section 5.2.2 evaluates the reconstruction cost. It is shown that using LTOs additionally raises the chance that less pages have to be overwritten. Thus, the energy consumption during reconstruction decreases. Section 5.2.3 evaluates the advantages of grouping modules and objects when applying the major placement strategy on a firmware image. Section 5.2.4 evaluates the limitations caused by memory fragmentation. Section 5.2.5 finally presents the resulting delta size achieved with the concepts presented in this thesis.

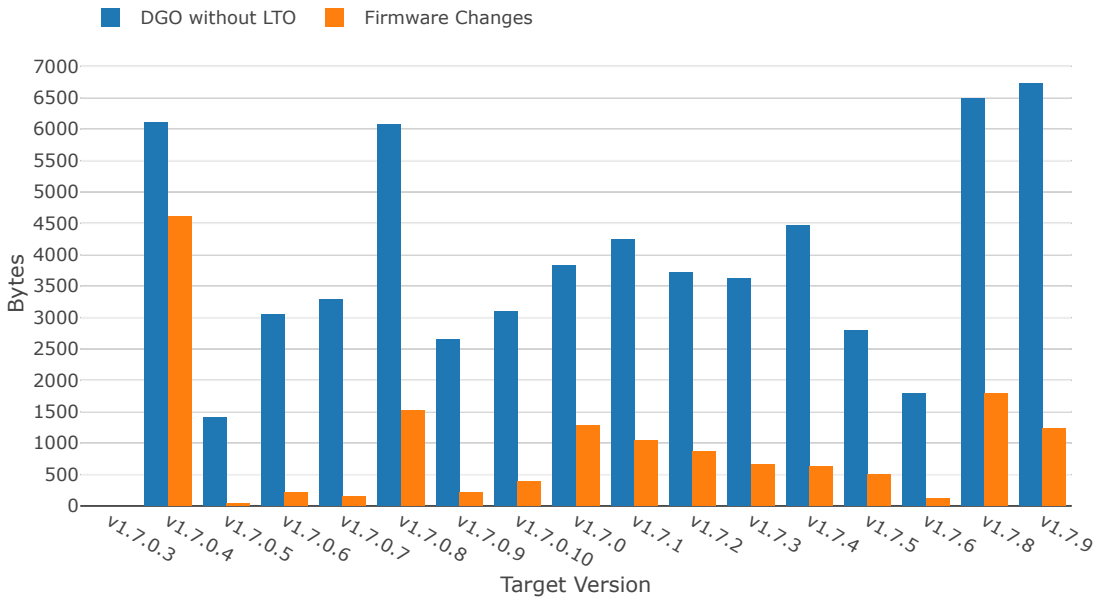

Figure 5.4: Delta Size when using DGO compared with the Firmware Changes before Linking Step.

5.2.1 Reduction of Address Shifts

The main target of the similarity improvement concept, presented in Section 3.1, is to preserve maximum similarity between different firmware versions. The presented concept aims to reduce the number of address shifts because they have a significant impact on the resulting delta size. Figure 5.5 compares the number of address shifts for following build-runs:

- *No Placement*: LTO disabled ($enable\_autoplacement = False$, described in Section 5.1.1).
- *Flash*: LTO applied to flash sections only ($enable\_autoplacement = True$, $memory\_roi = [FLASH]$).
- *Flash+RAM*: LTO applied to flash and RAM sections ($enable\_autoplacement = True$, $memory\_roi = [FLASH, RAM]$).

The presented LTO solution efficiently reduces the number of address shifts. Figure 5.5 additionally confirms the necessity of applying similarity improvements to RAM sections. Some of the target versions produce a lot of address shifts due to shifted RAM sections. In this case the number of address shifts in the *Flash* build run is significantly higher compared to the *Flash+Ram* build-run.

5.2.2 Reconstruction Cost

The amount of used energy for reconstructing the new image is mainly depending on the number of flash pages that need to be overwritten. The reconstruction approach, presented

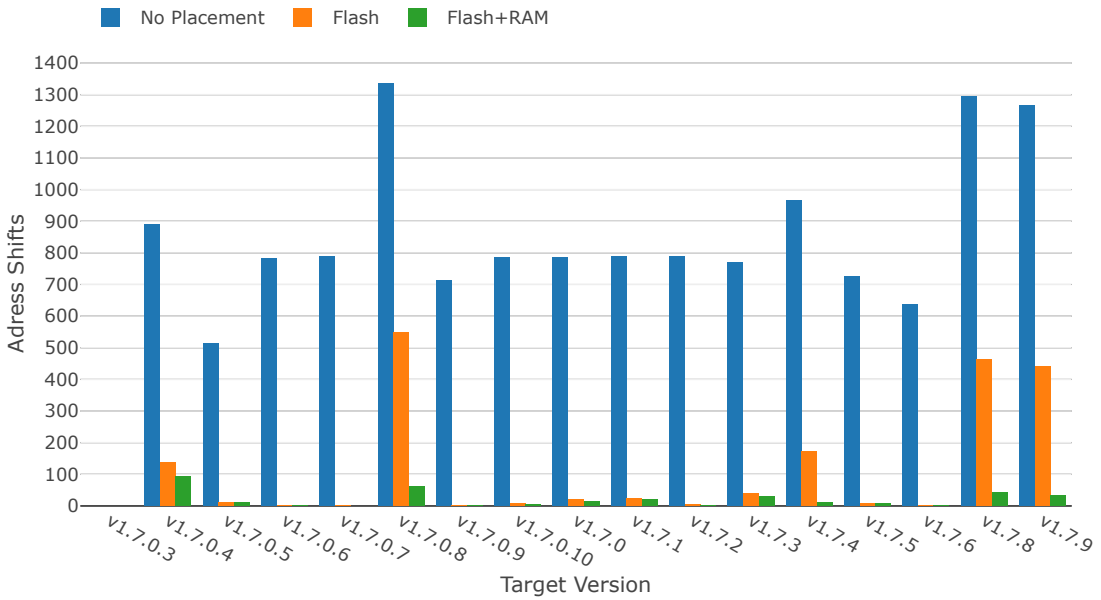

Figure 5.5: Reduction of Address Shifts when using Link-Time Optimizations.

in Section 4.7, only requires to overwrite pages that are modified by delta messages. Thus, minimizing the number of shifted sections results in less pages to overwrite. Figure 5.6

Figure 5.6: Reconstruction of Target Memory. Target Version = v1.7.5.

and 5.7 shows the reconstruction of two target versions on flash with delta messages using three different approaches:

- *DG without LTO*: Delta Image created with the Delta Generator algorithm without

LTO.

- *DGO without LTO*: Delta Image created with the DGO algorithm without LTO.
- *DGO with LTO*: Delta Image created with the DGO algorithm including LTO.

Each x-axis tick in both figures represents a flash page boundary. The size of a flash page on the CC430 is 512 bytes. Page 1 starts at memory address $0x8000$ and ends at $0x8200$, page 2 starts at address $0x8200$ and ends at $0x8400$. Using LTO reduces the number of pages to overwrite from 31 to 7 for the update to target version $v1.7.5$. For the update to version $v1.7.0.4$, the number of pages to overwrite is not significantly reduced. This is due to the high number of modified, removed and new sections.

Figure 5.7: Reconstruction of Target Memory. Target Version = v1.7.0.4.

5.2.3 Advantages of Grouping Modules and Objects

This section evaluates the effects of grouping object and modules. Furthermore, the advantages of *Slop Regions* between objects and modules are presented. Following build-runs (described in Section 5.1.1) were processed for evaluation:

1. *No Grouping, No Interspacing*:
   - *enable_autoplacement*: True
   - *group_objects*: False
   - *group_modules*: False
   - *interspacing*: None
2. *Group Modules, Interspacing Modules*:

- *enable_autoplacement*: True
- *group_objects*: False
- *group_modules*: True
- *interspacing*: 2048 bytes between modules

3. *Group Modules + Objects, Interspacing Modules + Objects*:

   - *enable_autoplacement*: True
   - *group_objects*: True
   - *group_modules*: True
   - *interspacing*: 1024 bytes between modules, 128 bytes between objects

Evolution of Memory Layout

Figure 5.8 illustrates the evolution of the memory layout for build-run 2 (*Group Modules, Interspacing Modules*), Figure 5.9 shows the evolution for build-run 3 (*Group Modules + Objects, Interspacing Modules + Objects*). The fragmentation (holes between sections) in

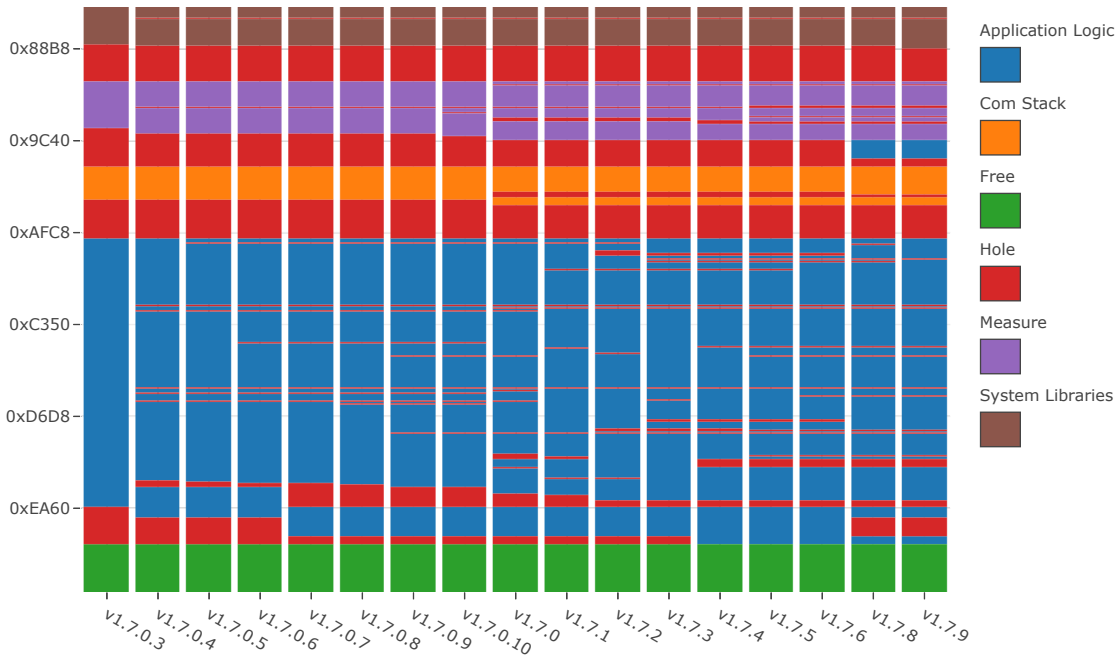

Figure 5.8: Evolution of Memory Layout when using Slop Regions between Modules.

modules at higher memory regions is bigger than in modules at lower memory regions. The *Application Logic* is the most fragmented module. Figure 5.8 and 5.9 prove the hypothesis from Section 3.1.4 that modules at higher memory regions usually become more fragmented. In case the MCU runs out of memory after several further updates, the additional delta size increase would be limited by applying bottom-up defragmentation (Section 3.1.4).

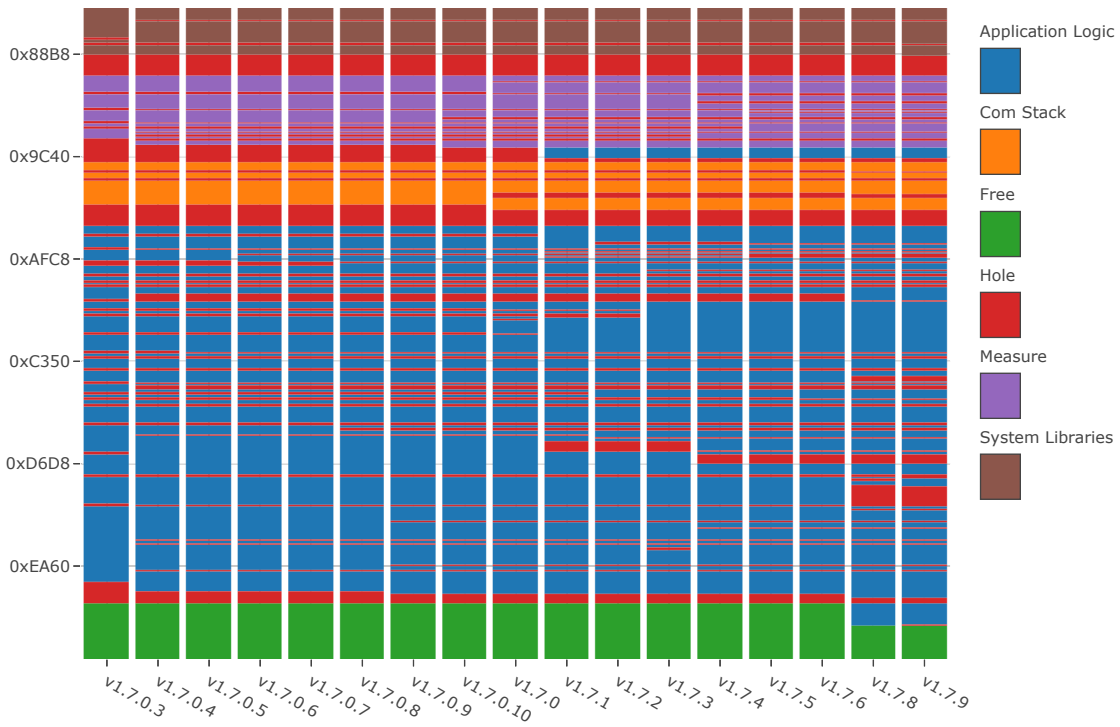

Figure 5.9: Evolution of Memory Layout when using Slop Regions between Objects and Modules.

Move Distance of Relocated-Modified Sections

Figure 5.10 shows the mean and standard deviation of move-distances for relocated-modified sections between the generated firmware versions. Equation 5.1 shows how to calculate the move-distance ($distance_{reloc}$) for a single relocated-modified section.

$$distance_{reloc} = abs(origin_{new} - origin_{old}) \tag{5.1}$$

The average move-distance is minimized in build-run 3 (*Group Modules + Objects, Interspacing Modules + Objects*) and maximized in build-run 1 (*No Grouping, No Interspacing*). This behaviour occurs due to the implementation of the minor-placement algorithm presented in Section 4.1.2: When sections are grouped into objects + modules and interspacing between them is configured while creating the major version, the minor-placement algorithm tries to place relocated-modified sections into their intended object regions. Small values of $distance_{reloc}$ can further reduce the resulting delta size when using the message encoding approach presented in Section 4.2. The move distance (*move_dist*) of COPY messages is encoded relative as signed LEB128. Smaller move distances require less bytes for encoding. Another advantage of smaller move distances is the higher probability that source and destination regions of COPY messages reside in the same flash page. That in turn reduces the probability of page conflicts (Section 4.3).

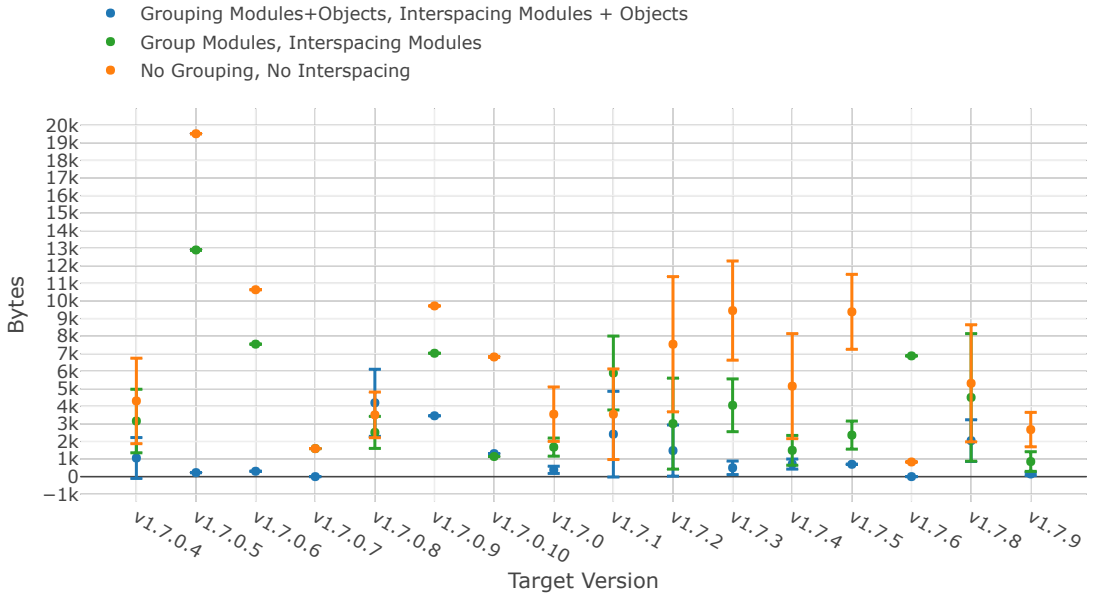

Figure 5.10: Mean and Standard Deviation of Move-Distances for Relocated-Modified Sections.

Delta Size

Figure 5.11 compares the resulting delta sizes of the different build-runs. They have

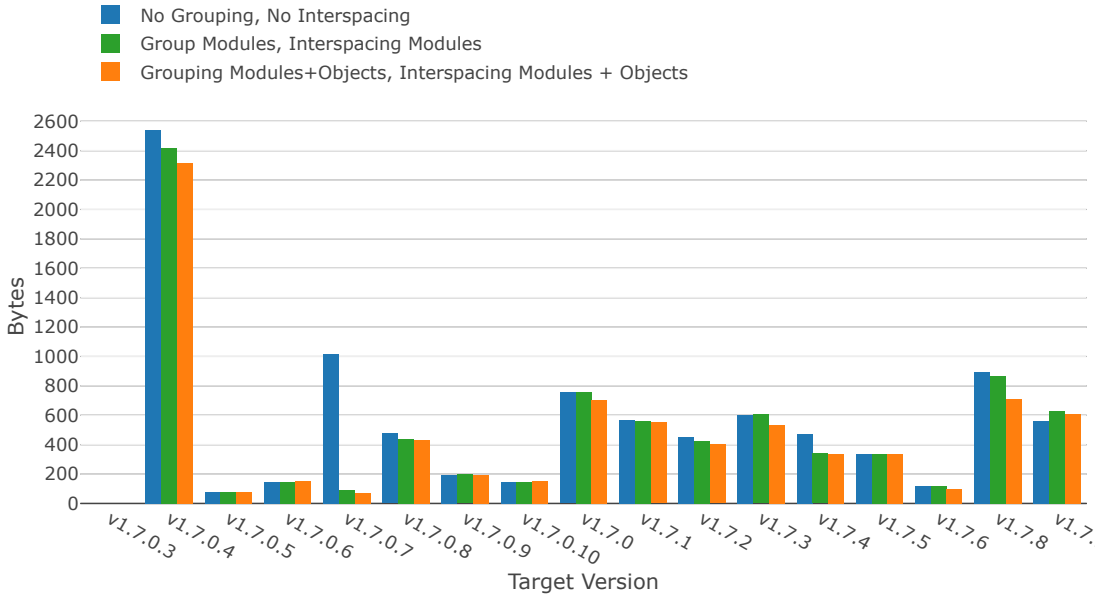

Figure 5.11: Comparing DGO Delta Size for different Grouping- and Interspacing Approaches.

following average delta size: build-run 1 = 585 bytes, build-run 2 = 510 bytes, build-

run 3 = 479 bytes. Build-run 3 has a slightly smaller average delta size than build-run 2. This results from the higher number of holes in build-run 3. The probability that modified sections can grow into following slop-regions (holes) and subsequently not have to be relocated is higher. The relocation of modified sections increases the resulting delta size. Build-run 1 has the highest average delta size. On the one hand this results from the low number of holes compared to the other build-runs, on the other hand the average delta size raises due to the update to target version *v1.7.0.7*: The delta of build-run 1 is much larger than the delta of the other build-runs. The large difference occurs because a large section is relocated in build-run 1. The previous position of this relocated-modified section is filled with a hole, this means the previous position is filled with a constant value. Due to the used placement strategy for build-run 1, no large holes exist before the update to target version *v1.7.0.7*. The DGO cannot use any existing holes for copying this large sequence containing a constant value and thus generates a large ADD message for the created hole.

5.2.4 Memory Fragmentation

This section evaluates the fragmentation of memory which is unavoidable when using the presented *Slop Regions* approach for improving similarity.

Total Fragmentation

Figure 5.12 shows the progress of total fragmentation for the build-runs described in Section 5.2.3. The total fragmentation conforms to the summed up size of all holes in

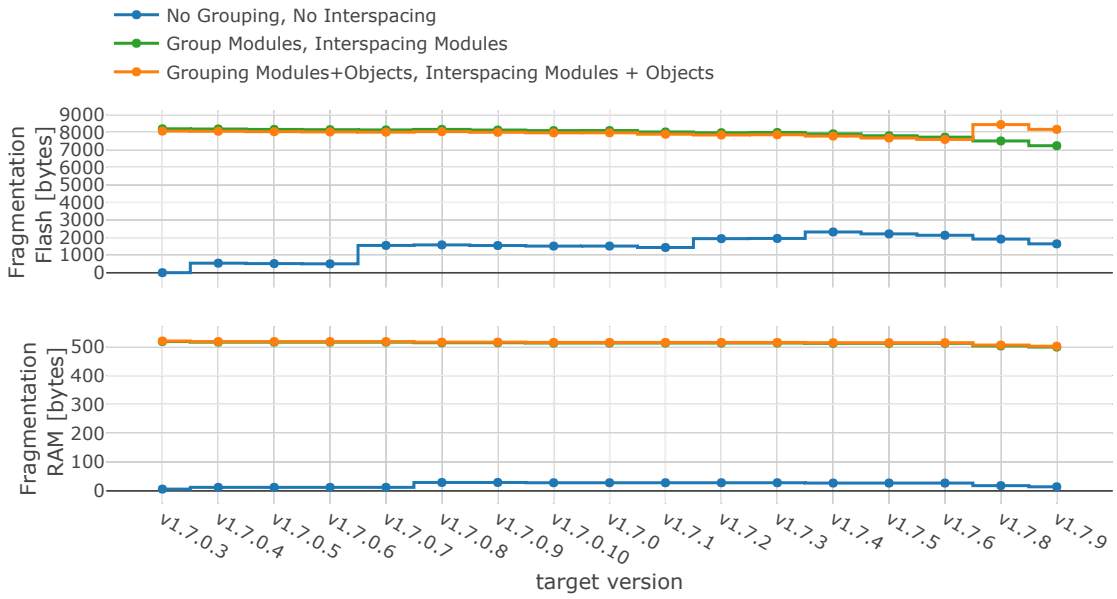

Figure 5.12: Progress of Total Fragmentation for different Grouping- and Interspacing Approaches.

the flash and RAM of the firmware image. For build-run 2 (*Group Modules, Interspacing*

Modules) and build-run 3 (*Group Modules + Objects, Interspacing Modules + Objects*), the total fragmentation is already very high at the initial version *v1.7.0.3*. This version is created using the major placement strategy. The progress of their fragmentation remains almost constant. This means that existing holes are efficiently used for relocated-modified and new sections. For the total number of 16 updates simulated, the total fragmentation does not increase significantly in build-run 2 and build-run 3. The total fragmentation in the initial version of build-run 1 (*No Grouping, No Interspacing*) is zero because no slop-regions (holes) are created in the major placement step. The total fragmentation increases during the processed updates because sections are removed and some of the modified sections are relocated.

Comparison Flash Usage

Figure 5.13 compares the flash and RAM usage of different build-runs. Without LTO (*No LTO* in Figure 5.13), the memory usage is minimized because no holes are created except paddings used for alignment of sections. These paddings are created by the linker of the used platform. Build-run 2 and build-run 3 have a lot of initial fragmentation. Thus, the memory usage is already high at version *v1.7.0.3*. Due to the efficient usage of holes for relocated-modified and new sections, the memory usage is nearly constant for all firmware versions. Figure 5.13 states that the initial interspacing between objects and modules should be chosen with respect to the available memory on the target device. When the resulting memory usage is already very close to the device's limits, the risk that defragmentation is necessary increases. In build-run 2 and build-run 3 for example, the flash usage is already very close to the maximum. When a large section needs to be inserted or relocated in a future update, defragmentation is unavoidable.

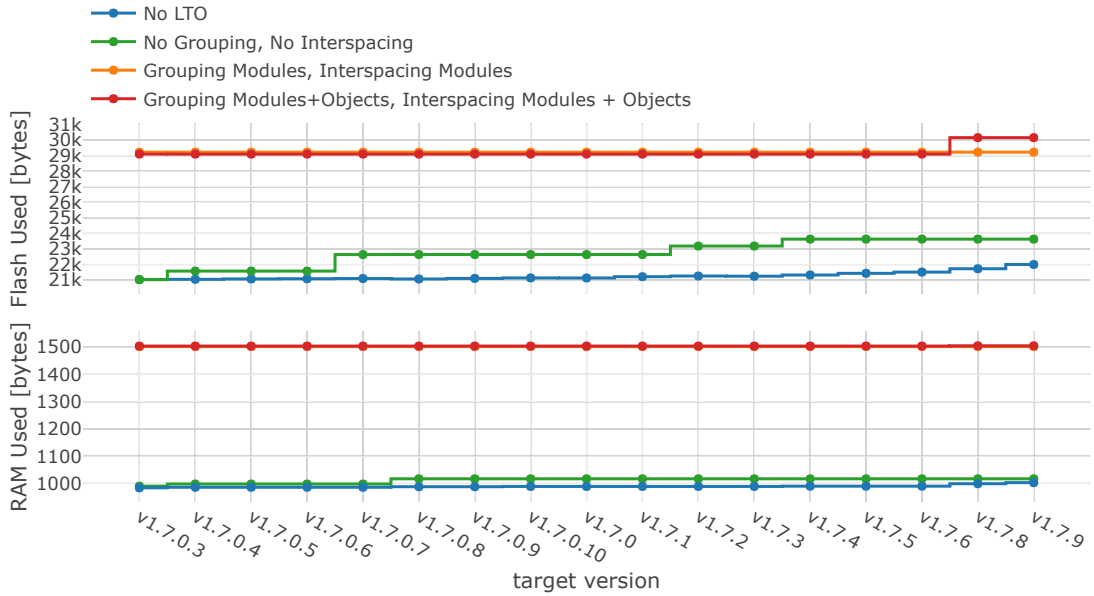

Figure 5.13: Progress of Flash Usage for different Grouping- and Interspacing Approaches.

None-Reusable Holes

The target of the minor placement strategy (Section 3.1.3) is to avoid the creation of small holes when sections are inserted or relocated. This section evaluates if the presented LTO approach creates small holes. Furthermore, the hypothesis that small holes lead to unusable memory for future updates is verified. Figure 5.14 verifies that small holes

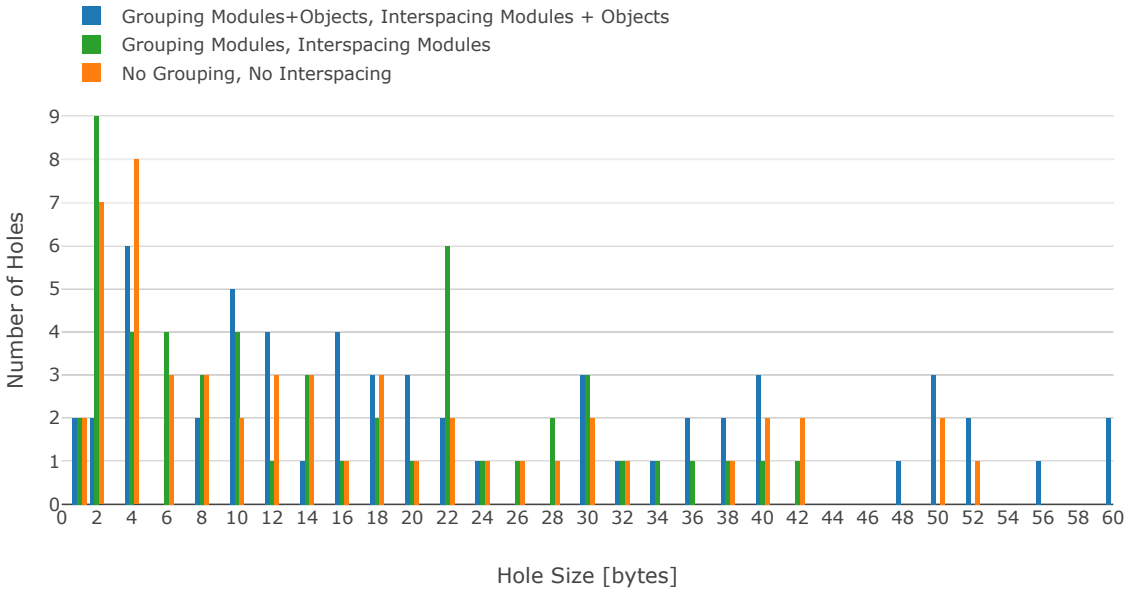

Figure 5.14: Distribution of small Holes (Slop Regions) at Version v1.7.9.

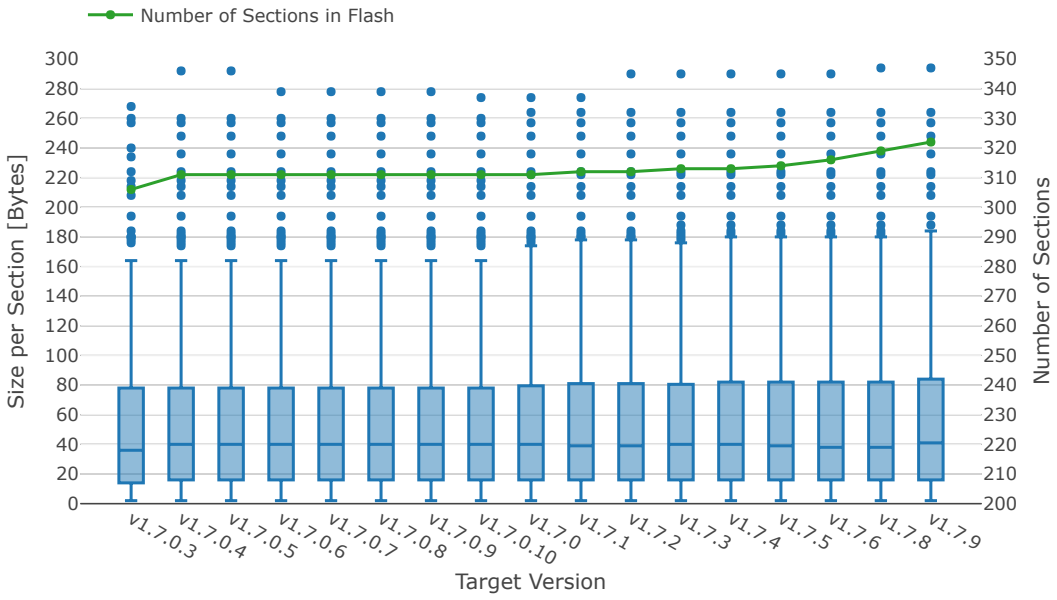

Figure 5.15: Distribution of Section Size for different Versions.

are existing after the update to version 1.7.9. It shows the distribution of small holes for the different build-runs. For evaluating the hypothesis of unusable holes, it's necessary to define the size of a so called "small" hole. Thus, the distribution of the section size is shown in Figure 5.15. The plot shows that 50% of the existing sections have a size between 20 and 80 bytes in all versions. 25% of the sections are larger than 80 bytes and 25% are smaller than 20 bytes. For further evaluation, a small hole is defined with a size between 0 and 20. This corresponds to the position of the lower quartile in the boxplots shown in Figure 5.15. Figure 5.16 compares the memory usage by small holes with the memory usage by holes

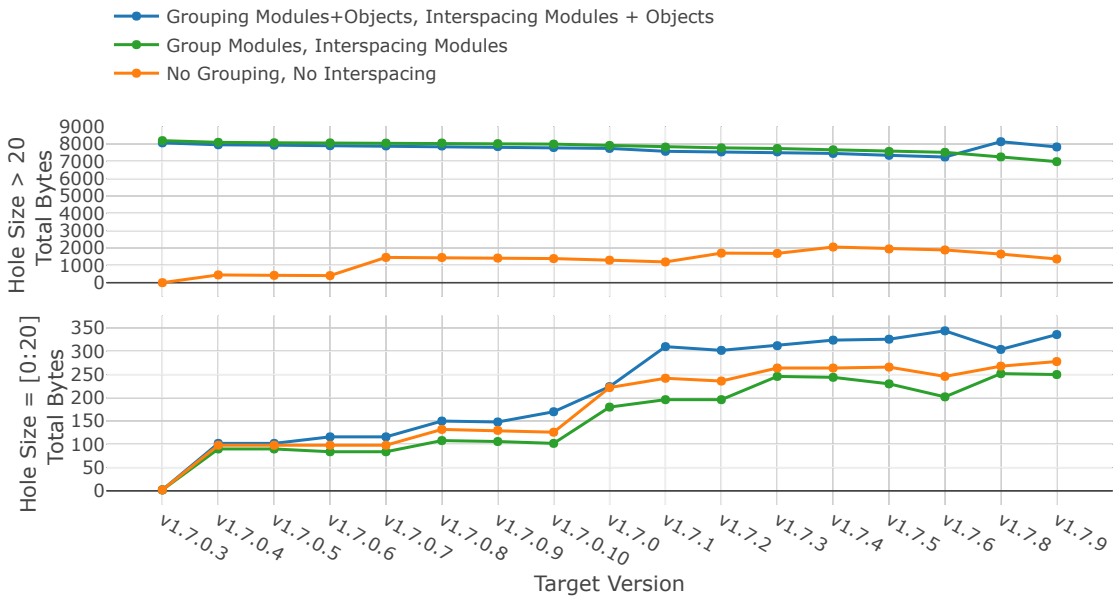

Figure 5.16: Increasing Memory Usage by small Holes.

bigger than 20 bytes. The memory usage of bigger holes remains almost constant for the different updates in build-run 2 and build-run 3. With these configurations, the existing larger holes are efficiently reused for relocated-modified and new sections. In build-run 1, the memory-usage by bigger holes increases more significantly compared to the other build-runs. This happens due to relocation of modified sections that do not fit into already existing holes and thus are placed at the end of the flash memory. The memory usage by small holes, which is shown in Figure 5.16, increases with the number of minor updates performed. This progress can be observed for each build-run. Thus, Figure 5.16 verifies the hypothesis that the presented LTO approach causes small holes that cannot be used for placing sections in future updates. Even though the total amount of unusable memory is relatively small after 16 updates, defragmentation will be unavoidable at some point when a lot of updates are performed.

5.2.5 Resulting Delta Size

This section evaluates the resulting delta size when using the presented LTO approach in combination with the DGO. Figure 5.17 shows the reduction of the delta size when using LTO. The significant improvement is achieved due to the reduction of address

shifts, as described in Section 5.2.1. Figure 5.18 compares the achieved delta size with

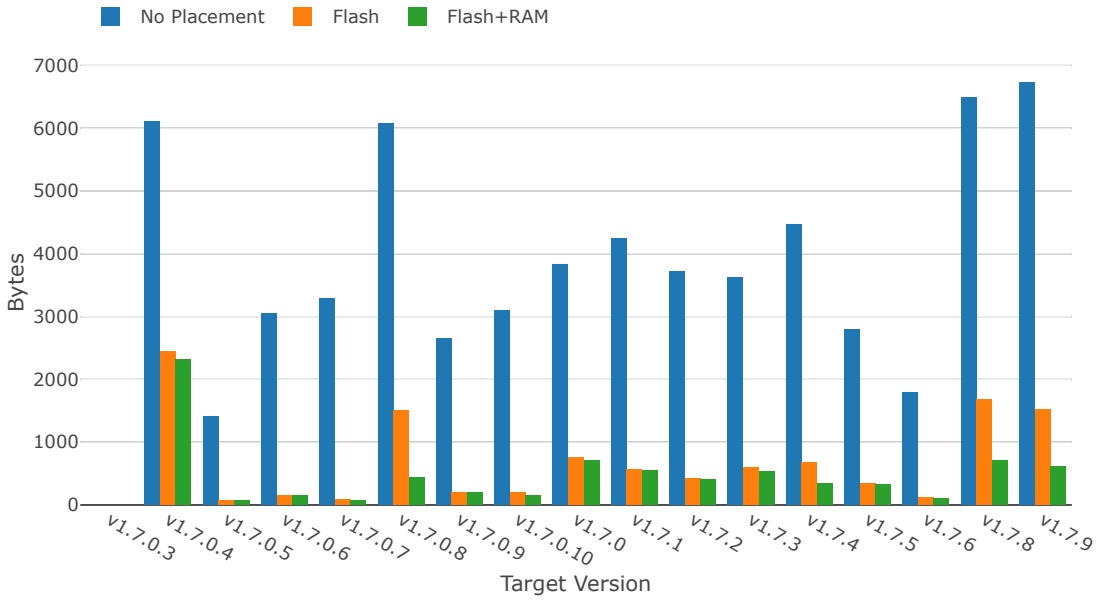

Figure 5.17: Reduction of Delta Size when using Link-Time Optimizations.

the firmware changes before linking step. For almost every update, the resulting delta size is significantly smaller than the amount of firmware changes. Table 5.1 summarizes

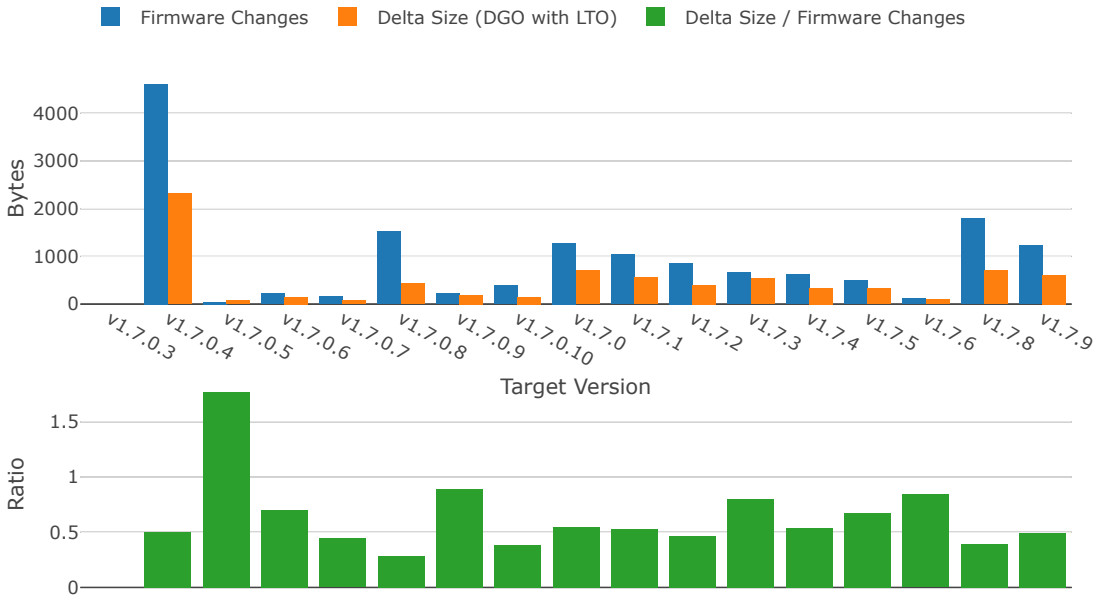

Figure 5.18: Comparison of Delta Size (DGO) including LTO with Firmware Changes before Linking Step.

the results presented in this chapter. Using the presented LTO approach leads to efficient

results for both delta algorithms evaluated in this Chapter. The improvement of the DGO compared to the existing Delta Generator is less significant when using LTO. This can be explained with the reduced number of address shifts. They are handled more efficiently by the developed DGO.

	Without LTO		With LTO	
	DG	DGO	DG	DGO
Mean Delta Size [bytes]	8660	3961	779	480
Deviation Delta Size [bytes]	2670	1633	868	537
Mean Compression [%]	59.59	81.53	96.34	97.75
Deviaton Compression [%]	12.31	7.54	4.12	2.55
Mean Improvement [%]	-	55.36	-	34.53

Table 5.1: Comparison of Delta Generator (DG) and DGO including LTO.

Chapter 6

Conclusion

This final chapter concludes the thesis. Section 6.1 summarizes the contributions presented in the previous chapters. Section 6.2 describes limitations and potential future work on the topics covered.

6.1 Contributions

The goal of this thesis was to implement and evaluate an efficient and reliable OTA solution for resource-constrained devices that communicate using a lossy, low-bandwidth network technology. Several issues relating to this goal had to be solved:

- The solution should support platforms with limited resources, where no additional hardware for processing firmware updates, such as external memory, is available.
- The amount of transmitted data should be kept to a minimum. The similarity of different firmware versions needs to be improved, otherwise the delta algorithm used does not achieve the required efficiency.
- Bringing a target device into an undefined state due to firmware updates must be prohibited.
- The OTA solution should have low platform dependency.

In order to address the issues stated above, this thesis examined following topics:

Similarity Improvements

Although the DGO algorithm achieves significant efficiency improvements compared to transmission of the whole firmware image, the efficiency is still not sufficient. Thus, improving the similarity of different firmware versions is necessary in order to further reduce the amount of data needed for applying updates. The goal of similarity improvements is to mitigate the effects of shifted code sections. These cause a high amount of address shifts inside the firmware image, which leads to delta files being much larger than the actual code changes themselves.

The solution presented in this thesis is based on the existing *Slop Region* approach, which inserts small pace-holders (holes) after each section. Several publications have mentioned that slop regions cause massive fragmentation of memory and thus do not present a practicable solution for similarity improvements. This thesis evaluates these drawbacks and presents a solution in order to avoid them. The presented *Major Placement* algorithm generates an optimized placement order of code/data sections within the memory by analysing their dependencies. This algorithm is intended to be used for creating initial firmware images before devices are shipped and firmware updates are applied. The presented *Minor Placement* algorithm is used for creating firmwares with maximized similarity based on an existing version.

Some of the other similarity improvement approaches modify the building toolchain or the executable binary. With the presented approach, only the linker script, which defines the placement of sections, is modified. This additionally reduces platform dependency. Only the linker script writer and the memory map parser may require adaptations for different platforms. Another advantage is that the approach does not require specific memory layouts. Other solutions require external memory in order to process the updates on the sensor. The evaluation was done with firmware that is actually used in productive environments. Without LTO, the used DGO algorithm achieved an average delta size of 3961 bytes (compression = 81.53%). Including the presented LTO solution, the average delta size was reduced to 480 bytes (compression = 97.75%). The major drawback of the presented approach is the fragmentation of memory. The evaluation in this thesis showed that the fragmentation is efficiently limited with the presented concepts. Only about 1% of memory was unusable due to fragmentation after processing 16 updates. The evaluation showed that the *Minor Placement* algorithm is able to efficiently reuse big holes in future firmware versions. Only small holes have a high probability of not being reused.

Update Processing on Sensor

Existing publications regarding OTA mainly focus on the efficient distribution of firmware updates, similarity improvements and delta generation. Besides the development of an efficient similarity improvement approach, this thesis also focussed on processing the transmitted update data on the sensor. The main goal of the sensor processing is to prevent an undefined state of the sensor due to firmware updates. Furthermore, the sensor should be able to process updates without additional hardware.

The developed concept presents several update components, which are included directly into the updateable firmware image. The *Diff Image Reception* logic is responsible for writing the received delta packets into the *Delta Image Buffer*, which is located in RAM. When all packets are received, the reception logic validates the delta image. The *Image Reconstruction* logic reconstructs the new version by using the received delta image and the old version. When reconstruction is finished, the sensor performs a reset. The bootloader performs image verification at startup. This ensures that no corrupt firmware is executed. After verification, the bootloader then starts execution of the newer firmware version.

The presented solution does not disturb normal operation on the sensor while receiving the delta image. Furthermore, the dual-image memory layout ensures that at least one working firmware image is always available on the sensor. This provides high reliability for the whole update system. The used memory layout does not require any external memory

on the sensor. However, the main drawback is that the maximum size of the firmware is only half of the available memory.

6.2 Future Work

With the conclusion of this work, some additional tasks remain:

- **Further reduction of platform dependency:** The current LTO implementation contains two platform dependent components. The information provided by the memory-map parser could also be extracted from the executable ELF file. This would further reduce the platform dependency of the LTO implementation and would decrease the effort involved in adapting the presented solution to different platforms. The ELF file format is standardized in contrast to memory-map files, which are often platform specific.
- **Minor-placement algorithm improvements:** The minor-placement algorithm currently relocates every modified section, which causes shifts of sections placed below. In some cases the resulting delta size would be smaller if the modified section is kept in the same position. This depends on the number of sections that are shifted below the modified section.
- **Further evaluations:** Carrying out an evaluation with more than 16 different firmware versions could provide a more detailed insight into the fragmentation of memory due to the presented LTO approach. Additionally, the delta size enlargement caused by memory defragmentation could also be evaluated. Another component that could be evaluated is the presented message encoding approach. This implementation should deliver smaller delta files, especially for 32-bit platforms.

Appendix A

Acronyms

ASCII	American Standard Code for Information Interchange
CS	Common Subsequence
DGO	Delta-Generator with Offset
EEPROM	Electrically Erasable Programmable Read-Only Memory
ELF	Executable and Linking Format
ES	Embedded System
FBC	Fixed Block Comparison
FRAM	Ferroelectric Random Access Memory
FUM	Firmware Update Manager
HAL	Hardware Abstraction Layer
IoT	Internet of Things
LCS	Longest Common Subsequence
LoRaWAN	Long Range Wide Area Network
LPWAN	Low-Power Wide Area Networks
LTO	Link Time Optimization
MCU	Microcontroller Unit
NMS	Non-Matching Segment
OS	Operating System
OTA	Over The Air Update
RAM	Random-Access Memory
RMTD	Reprogramming with Minimal Transferred Data
VM	Virtual Machine
VM	Virtual Machine
WSN	Wireless Sensor Network
XNP	Crossbow Network Programming

Bibliography

[AFGM+15] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, Fourthquarter 2015.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Computer Networks*, 54(15):2787 – 2805, 2010.

[All15] LoRa Alliance. *A Technical Overview of LoRa and LoRaWAN*. `https://www.tuv.com/media/corporate/products_1/electronic_components_and_lasers/TUeV_Rheinland_Overview_LoRa_and_LoRaWANtmp.pdf`, November 2015.

[Ast19] Michael Astl. Evaluation and Implementation of an Ecient Delta Algorithm for Incremental Firmware Updates. May 2019.

[bin06] *Bin-Packing*, pages 426–441. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[CPS07] N. Costa, A. Pereira, and C. Serodio. Virtual Machines Applied to WSN's: The State-Of-The-Art and Classification. In *2007 Second International Conference on Systems and Networks Communications (ICSNC 2007)*, pages 50–50, Aug 2007.

[DGV04] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Nov 2004.

[DLC+13] W. Dong, Y. Liu, C. Chen, J. Bu, C. Huang, and Z. Zhao. R2: Incremental Reprogramming Using Relocatable Code in Networked Embedded Systems. *IEEE Transactions on Computers*, 62(9):1837–1849, Sept 2013.

[DLW+10] Wei Dong, Yunhao Liu, Xiaofan Wu, Lin Gu, and C Chen. Elon: Enabling Efficient and Long-Term Reprogramming for Wireless Sensor Networks. volume 38, pages 49–60, 01 2010.

[DMH+13] W. Dong, B. Mo, C. Huang, Y. Liu, and C. Chen. R3: Optimizing Relocatable Dode for Efficient Reprogramming in Networked Embedded Systems. In *2013 Proceedings IEEE INFOCOM*, pages 315–319, April 2013.

[HC04] Jonathan W. Hui and David Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 81–94, New York, NY, USA, 2004. ACM.

[HKS+05] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani B. Srivastava. A Dynamic Operating System for Sensor Nodes. pages 163–176, 01 2005.

[HXHS09] J. Hu, C. J. Xue, Y. He, and E. H. M. Sha. Reprogramming with Minimal Transferred Data on Wireless Sensor Network. In *2009 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, pages 160–167, Oct 2009.

[Incar] Crossbow Technology Inc. XNP: Crossbow In-Network Programming. `https://www.eol.ucar.edu/isf/facilities/isa/internal/CrossBow/PresentationOverheads/Day2_Sect13_XNP.pdf`, Unknown year.

[JC04] Jaein Jeong and D. Culler. Incremental Network Programming for Wireless Sensors. In *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004.*, pages 25–33, Oct 2004.

[Jeo03] Jaein Jeong. Node-level Representation and System Support for Network Programming, 2003.

[JS14] G. Jurkovic and V. Sruk. Remote Firmware Update for Constrained Embedded Systems. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1019–1023, May 2014.

[KB16a] O. Kachman and M. Balaz. Optimized Differencing Algorithm for Firmware Updates of Low-Power Devices. In *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 1–4, April 2016.

[KB16b] Ondrej Kachman and Marcel Balaz. Effective Over-the-Air Reprogramming for Low-Power Devices in Cyber-Physical Systems. In Luis M. Camarinha-Matos, António J. Falcão, Nazanin Vafaei, and Shirin Najdi, editors, *Technological Innovation for Cyber-Physical Systems*, pages 284–292, Cham, 2016. Springer International Publishing.

[KB17] O. Kachman and M. Balaz. Firmware Update Manager: A Remote Firmware Reprogramming Tool for Low-Power Devices. In *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 88–91, April 2017.

[KP05a] J. Koshy and R. Pandey. Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.*, pages 354–365, Jan 2005.

[KP05b] Joel Koshy and Raju Pandey. VMSTAR: Synthesizing Scalable Runtime Environments for Sensor Networks. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, pages 243–254, New York, NY, USA, 2005. ACM.

[KPB+08] Mark D. Krasniewski, Rajesh Krishna Panta, Saurabh Bagchi, Chin-Lung Yang, and William J. Chappell. Energy-efficient On-demand Reprogramming of Large-scale Sensor Networks. *ACM Trans. Sen. Netw.*, 4(1):2:1–2:38, February 2008.

[KW05] S. S. Kulkarni and Limin Wang. MNP: Multihop Network Reprogramming Service for Sensor Networks. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 7–16, June 2005.

[LC02] Philip Levis and David Culler. Mate: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002.

[LMP+05] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. *TinyOS: An Operating System for Sensor Networks*, pages 115–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[LPCS04] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A Self-regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[LWS18] K. Lehniger, S. Weidling, and M. Schlzel. Heuristic for Page-Based Incremental Reprogramming of Wireless Sensor Nodes. In *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 61–66, April 2018.

[Manar] Saurabh Mangal. Network Reprogramming. `https://www.cs.wmich.edu/~gupta/teaching/cs603/wsnSp04/lec10a%20xnp%20Network%20Reprogramming%20021704.pdf`, Unknown year.

[MGL+06] Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. In Kay Römer, Holger Karl, and Friedemann Mattern, editors, *Wireless Sensor Networks*, pages 212–227, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[MH13] B. Mazumder and J. O. Hallstrom. An Efficient Code Update Solution for Wireless Sensor Network Reprogramming. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–10, Sept 2013.

[OK16] Marcel Bal Ondrej Kachman. Configurable Reprogramming Scheme for Over-the-Air Updates in Networked Embedded Systems. Technical report, Institute of Informatics, Slovak Academy of Sciences, 2016.

[PB12] R. K. Panta and S. Bagchi. Mitigating the Effects of Software Component Shifts for Incremental Reprogramming of Wireless Sensor Networks. *IEEE Transactions on Parallel and Distributed Systems*, 23(10):1882–1894, Oct 2012.

[PBM09] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. Zephyr: Efficient Incremental Reprogramming of Sensor Nodes Using Function Call Indirections and Difference Computation. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 32–32, Berkeley, CA, USA, 2009. USENIX Association.

[PKB07] R. K. Panta, I. Khalil, and S. Bagchi. Stream: Low Overhead Wireless Reprogramming for Sensor Networks. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, pages 928–936, May 2007.

[QHQ16] J. Qiu, L. Hunag, and L. Qian. LiREP: Lightweight Incremental Reprogramming of Sensor Nodes based on In-Situ Modification. In *2016 International Conference on Computing, Networking and Communications (ICNC)*, pages 1–6, Feb 2016.

[RL03] Niels Reijers and Koen Langendoen. Efficient Code Distribution in Wireless Sensor Networks. In *Proceedings of the 2Nd ACM International Conference on Wireless Sensor Networks and Applications*, WSNA '03, pages 60–67, New York, NY, USA, 2003. ACM.

[sacG] smaXtec animal care GmbH. Inside Monitoring. `https://www.smaxtec.com/en/inside-monitoring/`.

[SAH12] N. B. Shafi, K. Ali, and H. S. Hassanein. No-Reboot and Zero-Flash Over-The-Air Programming for Wireless Sensor Networks. In *2012 9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 371–379, June 2012.

[SCC+06] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java™on the Bare Metal of Wireless Sensor Devices: The Squawk Java Virtual Machine. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 78–88, New York, NY, USA, 2006. ACM.

[Sol00] Jose H. Solorzano. Tinyvm. `http://tinyvm.sourceforge.net/`, 2000.

[STM16] STMicroelectronics, `https://www.st.com/content/ccc/resource/technical/document/application_note/group0/ab/6a/0f/b7/1a/84/40/c3/DM00230416/files/DM00230416.pdf/jcr:content/translations/`

en.DM00230416.pdf. *Optimized Usage of the Dual Bank Structure of Flash Memory in STM32 Microcontrollers - Software Expansion for STM32Cube*, October 2016.

[TC05] G. Tolle and D. Culler. Design of an Application-Cooperative Management System for Wireless Sensor Networks. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.*, pages 121–132, Feb 2005.

[TDV08] N. Tsiftes, A. Dunkels, and T. Voigt. Efficient Sensor Network Reprogramming through Compression of Executable Modules. In *2008 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, pages 359–367, June 2008.

[Tex13a] Texas Instruments, http://www.ti.com/lit/ug/slau259e/slau259e.pdf. *CC430 Family User's Guide*, January 2013.

[Tex13b] Texas Instruments, http://www.ti.com/lit/an/slaa534/slaa534.pdf#page=70&zoom=auto,0,701.3. *MSP430 Embedded Application Binary Interface*, 2013.

[UNI93] UNIX International Waterview Corporate Center, http://dwarfstd.org/doc/dwarf-2.0.0.pdf. *DWARF Debugging Information Format*, July 1993.

[YMCH08] S. Yi, H. Min, Y. Cho, and J. Hong. Adaptive Multilevel Code Update Protocol for Real-Time Sensor Operating Systems. *IEEE Transactions on Industrial Informatics*, 4(4):250–260, Nov 2008.

[ZAZC16] C. Zhang, W. Ahn, Y. Zhang, and B. R. Childers. Live Code Update for IoT Devices in Energy Harvesting Environments. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, Aug 2016.

[ZNV+14] X. Zhong, M. Navarro, G. Villalba, X. Liang, and Y. Liang. MobileDeluge: Mobile Code Dissemination for Wireless Sensor Networks. In *2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems*, pages 363–370, Oct 2014.