Computer Communications 139 (2019) 91–102

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

# Universal framework for remote firmware updates of low-power devices

Ondrej Kachman [a,*], Marcel Balaz [b], Peter Malik [a]

[a] Institute of Informatics, Slovak Academy of Sciences, Bratislava, Slovak Republic
[b] R-DAS, s.r.o. Research company, Bratislava, Slovak Republic

ARTICLE INFO

*Keywords:*
Firmware update
Reprogramming
Low-power
Embedded
Multiplatform

ABSTRACT

Remote firmware updates are nowadays an important part of any firmware for embedded devices. In systems with dozens or even hundreds of devices, for example wireless sensor networks, cyber–physical systems, smart systems, etc., reprogramming each device manually just to fix a simple issue would be ineffective. We present a novel multiplatform framework for remote updates of low-power embedded devices. Our approach achieves its multiplatform nature through the availability of standard object files for almost every existing platform. We avoid any alteration of the source code and focus on thorough analysis of the firmware outline and generation of the linker commands. The framework tracks changes in the firmware, improves similarity between various versions and generates a differential file, called delta or patch. Reducing data shared between devices with delta files and minimizing the amount of memory operations improves the energy efficiency of an update. The process reduces the update size by approximately 80% compared to the full firmware transfer and improves over existing differencing solutions variously from 5% to 50%. The framework is designed for battery powered, physically inaccessible low-power devices in heterogeneous systems but can be used for any device connected to any network.

## 1. Introduction

Almost any computational device today has some type of a remote update mechanism coded into its base software. It creates the possibility to fix software problems, add, remove and modify features or change the software of the device completely. Different update strategies may be more suitable for different device types. Full firmware or software rewrite is easy to implement but does not have any special advantages. Modern systems often use differential updates. This kind of updates generates differential files, called deltas or patches. Delta files encode the differences between two files using operations that describe byte sequences that must be added, removed or shifted. Differential updates reduce the amount of data shared on the network. This prevents network congestion and reduces energy consumption of the update dissemination process for battery powered devices. The importance of the delta size reduction is even higher for low-power devices and further steps can be taken to improve the similarity of two firmware versions. A firmware update procedure can be split into multiple processes as shown in Fig. 1:

1. Similarity improvement — Aims to help a differencing algorithm detect more common sequences in firmware.
2. Delta file generation — A differencing algorithm encodes firmware differences into a delta file.
3. Network dissemination — Delta files are propagated from a base station to firmware update target devices.
4. Application of an update — A target device must apply operations from a delta file to rebuild new firmware.

The framework presented in this article is focused on the similarity improvement, delta file generation and application of an update. It is independent of a networking protocol used for the dissemination of delta files through the network. The reason for this is the spread of standard IP based protocols like CoAP or HTTP to embedded low-power devices. They can be used for the update purposes while being agnostic to what data are being shared. There is no need to implement an update specific protocol.

We propose a universal framework for the remote firmware updates. The main contribution of our work is the multiplatform nature of the proposed framework. Modern intelligent systems may consist of many devices based on different platforms. Implementation of the platform specific update mechanisms is inefficient and universal solutions are not common. Our framework does not alter machine code that is often platform or even device specific. The framework's algorithms can be used for any embedded platform that supports object files in the Executable and linking format (ELF). Port to a different object file format is also possible. Another contribution is our new version of the differencing algorithm Delta generator. We present a new delta

* Corresponding author.
*E-mail addresses:* ondrej.kachman@savba.sk (O. Kachman), marcel.balaz@savba.sk (M. Balaz), p.malik@savba.sk (P. Malik).

https://doi.org/10.1016/j.comcom.2019.03.014
Received 6 September 2018; Received in revised form 12 February 2019; Accepted 29 March 2019
Available online 2 April 2019

file format that improves delta size over existing algorithms. Smaller delta files result in less data shared on the network. The framework also supports different configurations. These can be used to adjust the algorithms to work the most efficiently for a chosen update scenario. Different configurations may be better for different devices and update cases. The final contribution is the development of a patch module for target devices that does not erase any page of a program memory more than once per update. The framework does not require external memory and does not rewrite pages that remain unchanged between versions. The changed pages are rebuilt in RAM memory and once finished, they are erased and written. This preserves the durability of program memories.

Section 2 provides insight into the state-of-the-art solutions used for firmware updates in embedded systems, our commentary on these solutions and a brief description of our previous work. Section 3 describes our framework for the remote firmware updates. Section 4 presents results of the experiments we carried out and Section 5 concludes this article.

## 2. Background of the study

This section provides a survey of the existing works in reference to the firmware update stages shown in Fig. 1. The first four subsections describe algorithms and methods related to the corresponding firmware update stage from the figure. The fifth subsection provides our commentary of the existing works and briefly sums up our previous works.

The developments in the area were originally centered around the TinyOS, an operating system developed for sensor devices. Deluge [2] is one of the most important solutions for the reprogramming of wireless sensor network (WSN) devices. It propagates changed blocks of code efficiently, reducing the amount of control messages between devices. Further research led to the development of similarity improving methods and differencing algorithms.

### 2.1. Improving firmware similarity

To make the differencing algorithms more efficient, it is possible to take steps to create more similar firmware images. The authors of Update conscious compilation (UCC) [3] preserve register allocations for variables between firmware versions by inserting MOV instructions into code at important addresses. This results in more common blocks of code. The disadvantage is the worse execution time caused by inserted instructions. Worst case execution time thresholds for various blocks of code were proposed in [4] , limiting the inserted operations but resulting in less common code.

Zephyr [5] uses the linking stage to improve the similarity of firmware. A call indirection table is created that holds the addresses of firmware functions. Every function call is routed through this table. This approach reduces the impact of function shifts. The disadvantage is an extra step between function calls. Hermes [6] was developed by the authors of Zephyr and builds on it by handling data shifts. It creates a gap in RAM between *.bss* section for uninitialized variables and *.data* section for initialized variables. This reduces shifts of these sections and the changes of references to these sections. However, it could not guarantee that the gap would not be crossed by the growth of either of these sections.

Authors of the live code update mechanism for internet-of-things (IoT) devices [7] also use memory redirections for various patches of the code. New or modified code is inserted at the end of memory space. References to this code are made using either the insertion of call instructions into the program memory or trampolines in the RAM, similar to the indirection table of Zephyr.

Pinning new code to a different part of the memory was introduced in [8] . This work presented a method to provide functions with slop regions. Slop region is a free space after a function. The function can grow on shrink in its slop region. This can greatly reduce memory shifts. The disadvantage is the memory fragmentation and less space for user data if a device stores some.

QDiff [9] handles similarity improvement using multiple techniques. New or modified code is appended at the end of the firmware and referenced using call instructions, similar to [7] . Deleted code is left out as a slop region. RAM sections are handled similar to Hermes, leaving free space between and expanding them in opposite directions. QDiff identifies that this change can influence indirect addressing in the program. It corrects all addresses in the instructions that access the RAM variables. QDiff also addresses the relative jumps by moving any inserted block that would influence them to the end and referencing it using a CALL instruction.

The authors of R2 [10] and its upgraded version, R3 [11] , use the relocatable code to improve firmware similarity. Relocation entries are instructions that reference different memory locations, other functions, variables, memory reserved for data storage, and more. These entries can change their values with every code shift. The authors propose to set the relocation entries to the same value for every firmware version. This helps differencing algorithms to detect more common code sequences. Real values are added to a delta file as compressed metadata.

### 2.2. Differencing algorithms

After taking steps towards more similar firmware, a differencing algorithm can be used to encode a delta file. This file encodes various operations that can be used to reconstruct new firmware version by a target device. Differencing algorithms also went through various improvements over the course of last decades. Initially, devices used in a WSN would be reprogrammed with full firmware image every time. This is inefficient, especially for the battery powered devices. Difference based approach is more energy efficient.

#### 2.2.1. Block-based and byte-based differencing algorithms

Presently, there are two most used types of differencing algorithms. The first type is represented by block-based algorithms, for example Deluge [2]. Such algorithms can be found even in more recent works [12]. The firmware is split into fixed-size pages. Only modified pages are disseminated through the network. The main advantage of block-based approaches is the easy implementation. The disadvantage is the higher amount of update data required.

The second type is represented by more efficient byte-based algorithms that started to surface soon after block-based solutions. These algorithms are developed to determine various-sized byte sequences that match and differ between two firmware versions. These sequences are then encoded as delta operations. One of the more referenced works that uses byte-based differencing is RMTD [13]. Authors of [10,11] also used its altered version and compared it to their own algorithm, R3diff. Another works that use or propose byte-based differencing algorithms are [5,6,9,14,15], each being used for a specific device or designed to reduce delta calculation times. We developed a word-based differencing algorithm [16], called Delta generator, as many of today's devices with NAND flash program memories use word as the smallest possible write unit. Since the publication, we changed the structure of a delta file to reduce the amount of required flash erase operations. These changes will be described later in Section 3.2.

#### 2.2.2. Operations encoded within a delta file

Operations encoded within the delta files are translated to memory operations. These operations are used to create new firmware by mixing existing data from its older versions with new data. Based on the related works, there are four basic operation types:

- COPY <old address> <new address> <amount of bytes/words> – copies existing data to a different location

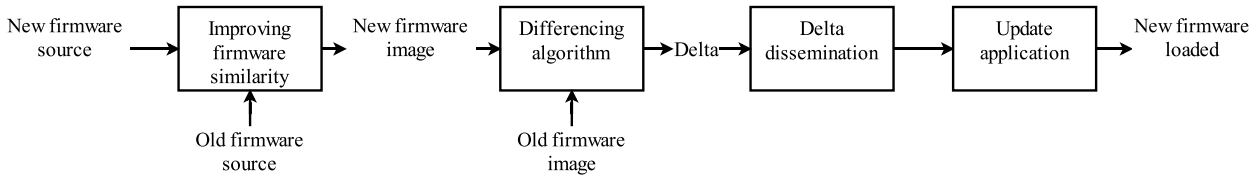


**Fig. 1.** Overview of a remote reprogramming mechanism for low-power devices [1].

- ADD <new address> <amount of bytes/words> <data> – inserts a sequence of data into a specified location
- INSERT <new address> <byte/word> – inserts a single byte or word into a specified location
- PAD <new address> <value> <amount of bytes/words> – pads a memory sequence with the same value

Every algorithm may use different variations of these operations or introduce new more specific operations. R2 [10] and R3 [11] rebuild the firmware in an external memory and use only ADD and COPY operations. QDiff [9] adds DELETE and REPEAT operations. DELETE deletes a sequence of bytes. REPEAT repeats a copy of an instruction. REPEAT is also used in Zephyr [5] and Hermes [6]. They also use a copy with insert (CWI) operation. CWI can be used to move large amounts of data with insertion of small chunks of data somewhere within the moved sequence.

### 2.3. Delta file dissemination

In this stage, our framework relies on the standard protocols available in the modern systems. However, there are works that develop their own solutions. Original remote update mechanism proposals were more focused on the propagation of updates through a network. Deluge was designed primarily as a remote update protocol for TinyOS. Mobile Deluge [17] is a newer implementation of Deluge for mobile updates of various sensor devices running TinyOS. Zephyr and Hermes use modified version of an older R-sync algorithm [18]. A newer algorithm ULTRA [19] reduces the dissemination process time by 34.8%.

With recent developments in smart systems, cyber–physical systems, IoT technologies, etc., the proprietary protocols are being replaced with standard ones. The recent works try to build on the existing standards. Authors of [20] propose that IoT devices do not have all code in their program memory, but instead download it after powering up. In [21], the authors propose a method for detection of the changed code that is securely transferred to all devices connected to a sensor cloud. A lot of research also focuses on determining the update trees and the most efficient path to disseminate an update between all the connected devices [22,23]. The authors of [24] use GSN network to propagate the update data.

### 2.4. Application of an update on a target device

Once a delta file is received by a target device, it must be processed to finish an update. This is also a very important part of any reprogramming mechanism. This can be executed by a bootloader, small module or a single function. This piece of code is also called an update agent or a patch module. A lot of energy can be saved on the memory operations. Erase operation is the most energy hungry for widely used NAND flash memories. The minimum erase unit is a page. This means, that to change a single non-empty byte (not 0xFF), the whole page must be completely erased and then written back with the changed byte. Modern patch modules must take this into account [25] .

R2 [10] and R3 [11] are designed for devices with an external memory. Firmware is rebuilt there and copied over to a program memory. This causes every page of firmware to be rewritten, even if a delta file is small. It increases energy consumption by additional writes in the external memory. QDiff [9] is designed to rebuild new firmware page by page in the program memory. The authors of [25] pointed out the lack of page-based approach in our original solution and proposed a heuristic using graph algorithms to find the best sequence of the pages for more efficient update. RePage [26] is designed to group similar functions on the same pages in the memory and to reduce shifts. Only changed pages are rewritten with RePage.

### 2.5. State-of-the-art discussion and previous work

Out of the 4 firmware update stages shown in Fig. 1, our framework is focused on the firmware similarity improvements, delta file generation and application of an update. As mentioned in the introduction, we rely on the standard protocols to handle the dissemination stage.

The focus of our work is to develop a multiplatform solution while preserving some of the advantages of the device specific solutions. Works like UCC, Zephyr, Hermes or even QDiff often mention inserting instructions, changing references to variables or creating jump tables in a memory. Methods like these must understand instruction sets that may greatly vary between different platforms. They also counter the optimizations made to the code by a compiler. From analyzed works, only remote linking mechanism [8] and R2 [10] use an approach that does not change machine code or fills the code back to its original state. The aim of our work is to not alter machine code as well. Code fragmentation can be useful to reduce code shifts in the memory. However, it can cause problems once the memory is heavily fragmented. Also, as shown in the study of energy consumption of NAND flash memories [27] , jumps between more pages consume slightly more energy than jumps between consecutive pages. Our conclusion is that the fragmentation can be useful, but the memory should be defragmented and cleaned up after the end of incremental updates.

We developed our own differencing algorithm [16] . It is a word-based algorithm. In its original version, the delta file size was small but the number of erases and writes was too high. Since the updates were to be executed right in the program memory, the delta included all COPY operations first and then all ADD operations. The structure of NAND flash memories that can only erase data with page granularity was not considered then. Each operation was executed individually and would cause erase and write of at least one page. Byte-write enabled ferroelectric RAM non-volatile memories are still not used widely so this old approach is very ineffective. This disadvantage was pointed out in [25] . To address this, we changed the structure of the delta file and designed a new patch module. This module does not execute delta operations individually but rather builds every flash page from operations that target it. The page is then erased and written only once.

The base of our framework is a reprogramming mechanism that was published in [1]. It introduces various configurations to different stages of the reprogramming process. The main experiments presented were for three firmware fragmentation types — no fragmentation, partial fragmentation and full fragmentation. They were carried out only on a single platform, AVR, more precisely ATmega32u4 microcontroller. In [28], we presented an architecture of our software tool we developed with our reprogramming mechanism. This article goes more into depth of the algorithms and methods that we use, as well as proves its applicability to different platforms. We apply it to an AVR device with 8-bit ATmega32u4 MCU, 16-bit MSP430 based device and more powerful 32-bit ARM Cortex-M4 based system-on-chip (SoC).

## 3. The design of our framework

We designed a framework for remote firmware updates of low-power devices. It can analyze, slightly modify, link firmware and generate a delta file for it. Furthermore, it tracks firmware versions, different linked binaries for these versions and delta files that are used to update from one version to another. One firmware version can be linked into multiple different binaries using various linker commands and directives. Delta file is different for each version update or downgrade. The framework keeps track of all version updates, rollbacks and currently uploaded firmware version. A server or a base station that takes care of the dissemination stage must determine which device is running which firmware version. With this information, it can query our framework for the correct delta to update required devices to the desired firmware version. Our framework is designed to manipulate firmware layout and generate the smallest delta files possible. It does not track network topology or firmware versions that are running on specific devices.

Our algorithms are focused on the work with relocatable and executable ELF files. We analyze these files before and after linking. The goal is to generate a linker command that will link firmware into a more similar version. The framework is multiplatform due to performing no real changes to the machine code. Our differencing algorithm supports variable word size. Two of the tested platforms, AVR and MSP430, use 2-byte word addressing. ARM uses 4-byte word addressing.

### 3.1. Improving firmware similarity

We developed a universal set of tools which support various configurations and aid a linker to produce more similar firmware images. The supported configurations are [1]:

- Code sections manual/auto split — a programmer can split *.text* section function into different sections manually, or use provided tool to split all functions automatically, e.g., *.text.function1* is assigned to a section *.text1.function1*. A GCC linker is able to place this section into any part of the memory (e.g., 0x7000) using –ssection-address=.text1=0x7000 parameter.
- No fragmentation, partial fragmentation and full fragmentation — Firmware is not fragmented in no fragmentation mode. Selected sections are provided with slop regions in partial fragmentation mode. Every function is assigned its own slop region in full fragmentation mode, as based on [8].
- Relocation entries extraction — We support the configuration to extract relocatable entries like R2 [10]. However, we only extract those that changed between versions. Instead of appending them as metadata, we generate INSERT delta operations that are later processed by the optimization stage of our Delta generator. Some of them might be optimized by merging into ADD operations. Our experiments show no positive effect of this configuration option.

This stage uses many different processes and tools, as well as many different file types. The outline is shown in the diagram presented in Fig. 2. The figure is split into 4 rows. The first row represents standard tools used to produce firmware image — linker, objcopy tool and hex2bin tool. The second row consists of files that represent the firmware image from the ELF format through its Intel HEX representation to the final output binary. The third row represents our custom algorithms and tools described in the list below. The fourth row represents our custom files generated by the custom tools. The directed lines show the flow of data between tools and files, some files may be input into multiple tools. Linker info extractor is executed separately before and after linking. The flow before linking is highlighted by the dashed line, the flow after linking by the solid line. Description of the custom tools in the third row:

- ELF file analyzer and editor — parses a whole ELF file into a JSON file for faster analysis and can edit the string tables of ELF files
- Linker info extractor — an algorithm that generates a JSON file that holds various information about sections stored in ELF files. It can be run independently before and after linking, it does not require both input files from Fig. 2 at once. The path of the input file before linking is marked with the dashed line in Fig. 2, the path of the input file after linking is marked with the standard solid line.
- Relocation info extractor — creates a JSON file that holds an array with section names, locations, sizes and values of all firmware relocation entries
- Firmware layout analyzer and linker command generator — can analyze physical address assignments of *.text* sections including their growth, shifts, etc., and can generate a linker command that causes the linker to partially or fully fragment firmware

The main output of the whole process is a binary image of firmware that is compared to an older version by the differencing algorithm. A file with the relocation entries info can also be passed to the differencing algorithm provided it is configured to deal with the relocations. These two output files are marked gray in Fig. 2.

#### 3.1.1. Splitting the .text section

Each compiled code section is put by linker into the *.text* section by default. Naming sections *.text1*, *.text2*, etc., enables us to generate a linker command that puts these sections to the addresses specified by us. To automatically rename code sections like this, the compiler must be configured to put every function into its own ELF section of the relocatable ELF file. The names of these ELF sections for each function are held in a special ELF section called the string table, usually one of the last sections of a relocatable ELF file. It is composed of null terminated ASCII strings. Fig. 3 illustrates the basic outline of a relocatable ELF file on the right. An ELF header includes the offset that points to the end of the file where the section header table is located. The section header table includes offsets of the individual sections that are located before it in the file. The described algorithm works with the string table section illustrated just above the symbol table. The edits performed are illustrated on the left side of Fig. 3. Numbers are added to each code section that starts with "*.text.*" string. For example *.text.uart_init* is changed to *.text**1**.uart_init*, *.text.appStart* is changed to *.text**6**.appStart*. This addition causes the string table to grow, shifting the offsets of the symbol table and the section header table. Therefore, the section header table offset must be corrected in the ELF header and the symbol table offset must be corrected in the section header table. Note, that there are no changes to the actual code produced by a compiler.

#### 3.1.2. Extracting linking related information from the ELF files

Linker info extractor is used both before and after linking. For every generated section, we extract the following information:

1. Section name.
2. Information about section's presence in a previous version, used by pre-link info files.
3. Physical address of the section, used by post-link info files.
4. The section's content and size. Content is copied from an ELF file.
5. The section's slop region size, used by post-link info files.
6. Shifted pages, a number of memory pages between section's old and new location. Unused for new sections.

#### 3.1.3. Extracting the info about relocatable code

Relocation info extractor scans *.rel* or *.rela* sections. These sections are generated by the compiler for every *.text* section that has relocatable code within its contents. This algorithm runs after linking triggered by our custom command. It creates a simple JSON file that holds an array with information about every relocation entry including the section it belongs to, its physical address, size and value.

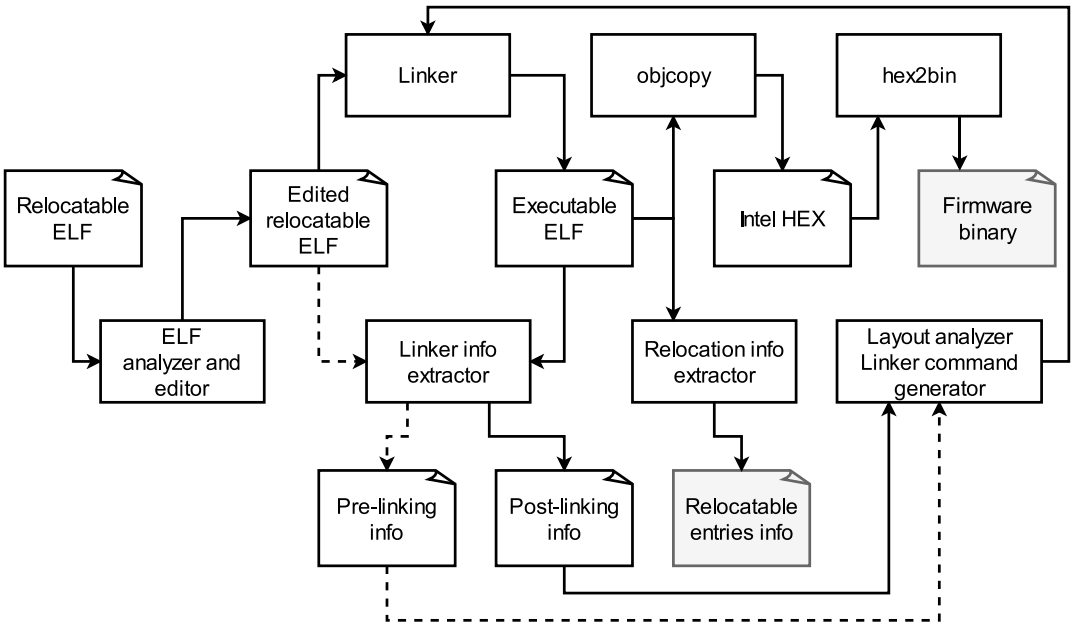


**Fig. 2.** File types and processes used in the firmware similarity improvement stage.

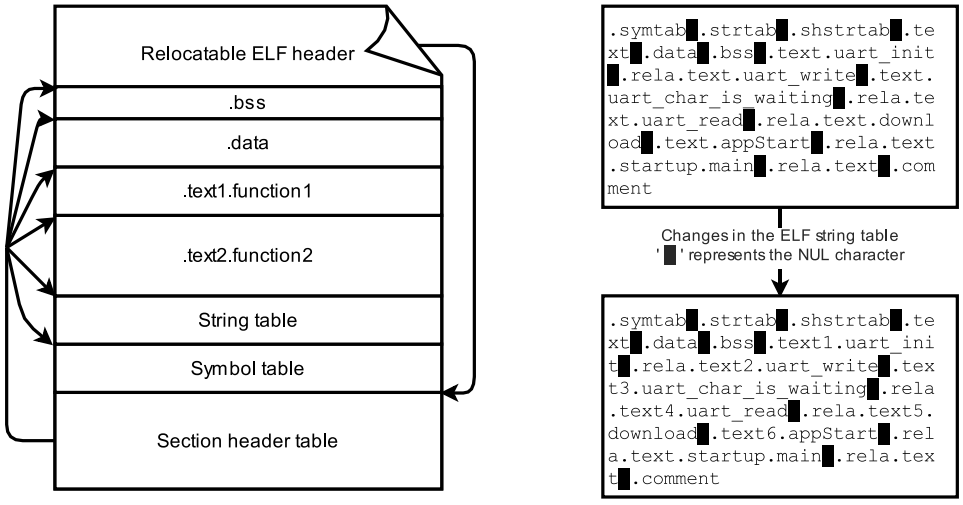


**Fig. 3.** Basic outline of a relocatable ELF file (left) and edits to the string table (right)

### *3.1.4. Generating a custom linker command*

The three previous algorithms generate information files as their output. This algorithm is used for the fragmentation of firmware with utilization of the files generated by the linker info extractor. By comparing post-link info file of an old version and pre-link info file of a new version, it calculates new physical addresses for each section. The addresses are appended to a linker command and the linker is started. Linked executable ELF file is used to create a binary firmware image that is passed to the differencing algorithm. The rules for fragmentation and their best use cases are following:

- No fragmentation
  - Old section preserves its order, no slop region is added.
  - Deleted section causes its neighboring sections to join.
  - New section is addressed at the end.
  - Best used for single or sporadic firmware updates.
- Partial fragmentation
  - Old section that reduces or preserves its size will preserve its address.
  - Deleted section will leave a gap.
  - New section is addressed at the end.
  - Old section that increases its size is addressed after new sections and provided with a slop region.
  - Best used for a short sequence of incremental updates (smaller bug fixes, addition of new features)
- Full fragmentation
  - Every section is provided with a slop region.
  - Order of old sections is preserved between versions.
  - Best used for frequent updates (major changes in the firmware functionality).

Linking configurations are illustrated in Fig. 4. The left side illustrates standard linking with *function 2* deleted, *function 6* inserted into the middle of the code, and *function 4* that increased its size. The right side of the figure represents our approach to changes in the firmware

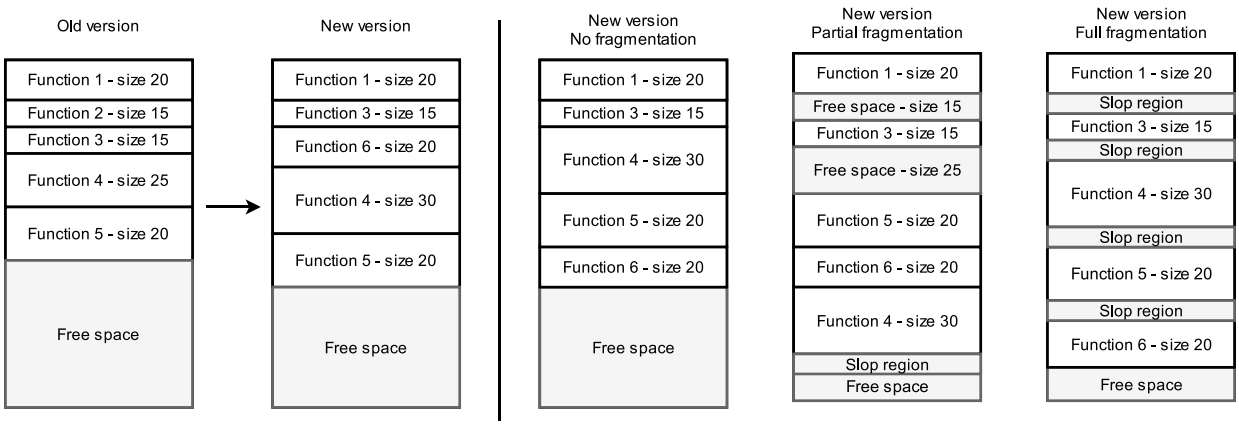


**Fig. 4.** Firmware update example. Left — standard linking, right — custom linking with various fragmentation configurations.

function layout with respect to the fragmentation rules listed before in this subsection:

- No fragmentation — instead of inserting *function 6* into the middle of the firmware we instruct the linker to place it at the end. Functions 3–5 are kept together and shifted to the address where *function 2* resided. The approach is not necessarily more efficient immediately, but may reduce memory operations for future updates to the new function, *function 6*.
- Partial fragmentation — The space of *function 2* remains free and serves as a slop region for *function 1*. This way *function 3* retains its original address. Since *function 4* grows and would cause *function 5* to shift, it is placed at the end instead. Its previous space serves as a slop region for *function 3*. *function 5* stays in its place. New function, *function 6*, follows without slop region. *function 4* is placed last.
- Full fragmentation — Each function is provided with a slop region in which it can grow and shrink. Initially, the fragmentation process causes a lot of memory shifts. However, memory shifts are significantly reduced for future updates. New function, *function 6*, is placed at the end.

### 3.2. Differencing algorithm — delta generator

After firmware is linked and converted from an executable ELF file to a binary image, it is compared to an older image by the Delta generator. Its original version presented in [16] had some flaws. The main flaw was the high amount of page erases required to complete the update process. The original version supported only COPY and ADD delta operations. All COPY operations were executed in the program memory before any new data were added. The rest of the data were filled in later using ADD operations. The structure of the delta file corresponded with this approach. COPY operations were placed consecutively at the beginning of a delta file, the rest of the file was filled with ADD operations. The presented new version adds support of the INSERT and SKIP operations. The structure of delta files also changed. No multiple page rewrites are required with the new version. The additional improvement is switching from the byte-based comparison to the word-based comparison and addressing. The delta operations are:

- ADD: 3+$n$ bytes
  - 1 byte — operation code (1)
  - 2 bytes — length of the data in words
  - $n$ bytes — data to be added to flash
- COPY: 3 bytes + word or 2 bytes + word
  - 1 byte — operation code (3) and the number of consecutive COPY operations, the op code is determined using lower 3 bits and the number of consecutive operations using upper 5 bits, consecutive COPY operations leave this byte out
  - 2 bytes — length of the data that will be copied in words
  - word — source address from which to copy the data
- INSERT: 1 byte + word
  - 1 byte — operation code (5)
  - word — the word to be inserted into the memory
- SKIP: 1 word
  - word — word sized skip to the target address of the next delta operation, the word is in LSB format and the lowest byte is always even, so no operation code is required

The SKIP operation does not need an operation code due to our trick we used. The operation codes are specially designed to be odd numbers (1, 3, 5) which combined with the word-based addressing enables to detect the SKIP operation when the first byte is an even number. This results in the reduced delta file size.

The algorithm starts by filtering out the non-matching segments between two versions. Next, it locates all common sequences between these segments and the old firmware. Found common sequences are encoded as COPY operations. The remaining unmatched sequences are encoded as ADD operations. The single word new sequences are encoded as INSERT operations. If the similarity improvement stage included extraction of the information about relocation entries, changed relocation entries are encoded as INSERT operations. The algorithm sorts all generated operations by their target addresses in ascending order and proceeds to the optimization stage. Some operations may operate on a small amount of data. They can be merged into another operation and thus reduce the delta file size. The example of the process is shown in Fig. 5. The delta operations do not include target addresses. Target address for each operation is calculated from the previous operations by summing up all skips, copied or inserted data. The illustration of the delta file structure is in Fig. 6. Delta generator supports two different configurations:

1. Clean mode — the delta file includes all operations generated by the algorithm, including padding operations. The firmware image after an update has no data leftover from the previous versions.
2. Dirty mode — padding operations which fill memory with empty symbols, are removed. Some data from an old firmware version may be leftover, but the resulting delta file size may be further reduced.

### 3.3. The update module

The update module is the code that executes operations encoded within a delta file once the file is stored on a target device. The update

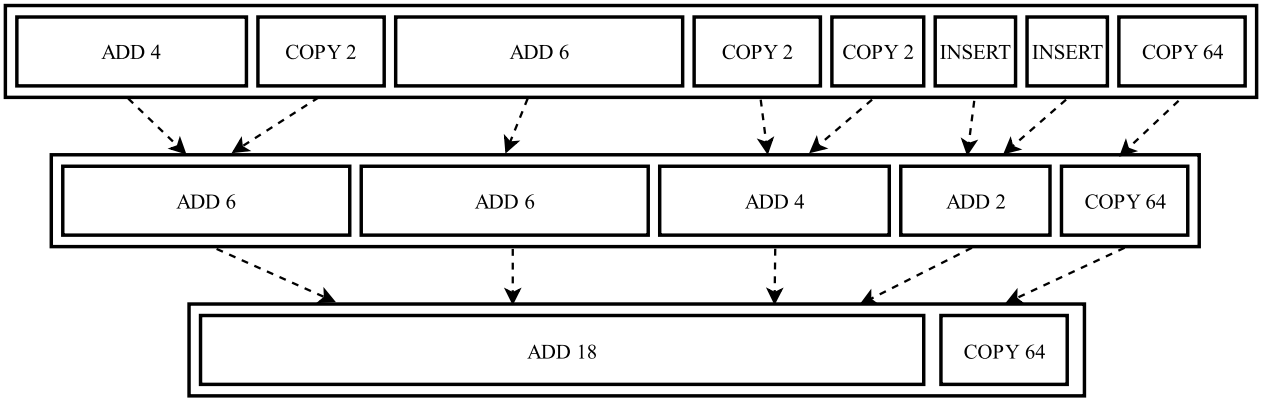


**Fig. 5.** Example of the delta operation optimization process.

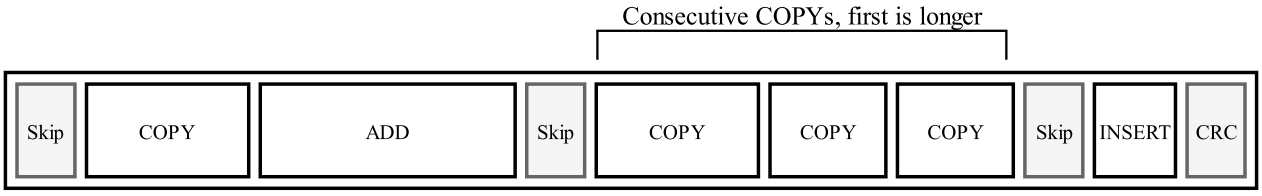


**Fig. 6.** Example of operations encoded within a delta file.

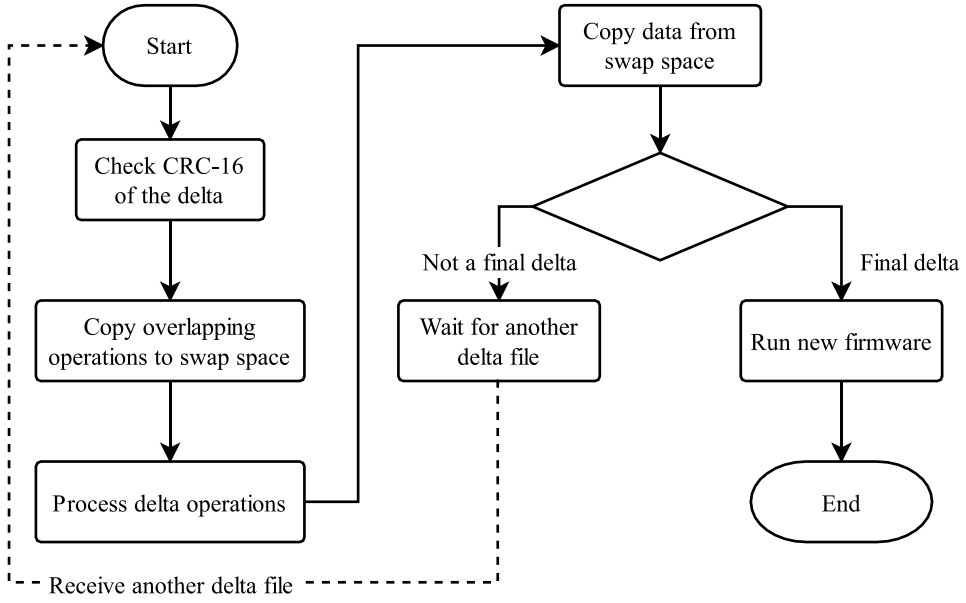


**Fig. 7.** Flowchart of the update module.

module can be a part of a bootloader or an independent code at a standalone position. It requires the following information:

- memory space reserved for firmware
- memory space reserved for the update process
  - memory reserved for the delta file
  - memory reserved as a swap space for overlapping data

This information is determined before the update process. If there is not enough space, the update cannot be applied. In that case, the delta file must be split into multiple files so that the memory reserved for them and swap operations does not overflow. The update module cannot run a new firmware version until the last delta file has been processed. Splitting of the delta is done only when the memory space is insufficient.

The flowchart of the update module is shown in Fig. 7. The first process in the flowchart after an update starts represents the integrity check. We use CRC-16 code for our delta files. This code is also depicted by the last block in Fig. 6. If successful, the module starts processing the delta file. The update module iterates through the delta file twice. In the first loop, depicted by the second process in Fig. 7, it detects overlapping operations. If the data source of a COPY operation is overwritten by any other operation, these data must be copied to the swap space. The copied segment is then rerouted to its address through the swap space. In the second loop, the third process in the figure, the pages are being reconstructed. Every page is rebuilt in RAM. Each delta operation that targets the currently processed page at least partially is

**Table 1**
Basic test results on the ATmega32u4.

| ATmega 32u4 | Firmware | | RMTD delta (bytes) | R3 diff delta (bytes) | Delta generator delta | | |
|---|---|---|---|---|---|---|---|
| Change case | Size (bytes) | Change (bytes) | | | Size (bytes) | % of firmware reduced | % reduction compared to R3 diff |
| Base | 688 | n | n | n | n | n | n |
| 1 | 692 | +4 | 256 | 217 | 207 | 70% | 4% |
| 2 | 978 | +286 | 629 | 548 | 559 | 42% | −2% |
| 3 | 1348 | +370 | 767 | 678 | 657 | 51% | 3% |
| 4 | 1080 | −268 | 186 | 169 | 148 | 86% | 12% |
| 5 | 1840 | +760 | 1002 | 937 | 896 | 51% | 4% |
| 6 | 1884 | +44 | 367 | 315 | 290 | 84% | 8% |
| 7 | 1630 | −254 | 101 | 96 | 72 | 95% | 15% |
| 8 | 1630 | 0 | 55 | 55 | 41 | 97% | 15% |
| 9 | 1698 | +68 | 359 | 318 | 285 | 83% | 10% |
| 10 | 1734 | +36 | 493 | 430 | 400 | 76% | 7% |

**Table 2**
Basic test results on the MSP430.

| MSP430 | Firmware | | RMTD delta (bytes) | R3 diff delta (bytes) | Delta generator delta | | |
|---|---|---|---|---|---|---|---|
| Change case | Size (bytes) | Change (bytes) | | | Size (bytes) | % of firmware reduced | % reduction compared to R3 diff |
| Base | 638 | n | n | n | n | n | n |
| 1 | 638 | 0 | 15 | 15 | 7 | 99% | 53% |
| 2 | 812 | +174 | 455 | 360 | 316 | 61% | 12% |
| 3 | 718 | −94 | 80 | 73 | 61 | 91% | 16% |
| 4 | 638 | −80 | 219 | 180 | 134 | 79% | 25% |
| 5 | 2768 | +2130 | 2894 | 2360 | 2366 | 14% | −1% |
| 6 | 4584 | +1816 | 2880 | 2256 | 2245 | 51% | 0,5% |
| 7 | 3954 | −630 | 513 | 477 | 463 | 88% | 3% |
| 8 | 3954 | 0 | 40 | 38 | 27 | 99% | 29% |
| 9 | 4540 | +586 | 1479 | 1094 | 1097 | 75% | −1% |
| 10 | 3306 | −1234 | 175 | 155 | 143 | 95% | 7% |

**Table 3**
Basic test results on the ARM Cortex-M4.

| ARM Cortex-M4 | Firmware | | RMTD delta (bytes) | R3 diff delta (bytes) | Delta generator delta (bytes) | | |
|---|---|---|---|---|---|---|---|
| Change case | Size (bytes) | Change (bytes) | | | Size (bytes) | % of firmware reduced | % reduction compared to R3 diff |
| Base | 18647 | n | n | n | n | n | n |
| 1 | 18647 | 0 | 15 | 15 | 9 | 99.9% | 40% |
| 2 | 19146 | +499 | 1620 | 1329 | 1321 | 93% | 0.5% |
| 3 | 19414 | +268 | 2383 | 1955 | 2045 | 89% | −4.6% |
| 4 | 18674 | −740 | 1974 | 1683 | 1666 | 91% | 1% |
| 5 | 57250 | +38576 | 52417 | 37810 | 37785 | 34% | 0.1% |
| 6 | 57742 | +492 | 5056 | 4006 | 3487 | 94% | 13% |
| 7 | 30822 | −26920 | 4934 | 3974 | 3782 | 87% | 4.5% |
| 8 | 30822 | 0 | 288 | 236 | 209 | 99% | 11.5% |
| 9 | 30230 | −592 | 2202 | 1802 | 1626 | 95% | 10% |
| 10 | 30570 | +340 | 2529 | 2018 | 1929 | 93% | 4.5% |

executed. If some of the currently processed page data are in the swap space, the target location is left blank for future write. After the page is rebuilt, it is physically erased in the program memory and written back from RAM. The fourth process includes copying of the data stored in the swap space to their locations. These were previously blanked and can now be written without the erase operation. After the delta operations have been processed, the update agent checks with the server or a base station if the delta file was final. If not, it will download the next delta and repeat the described processes. If yes, the new firmware is ready. Using the presented update module, every page is erased at most once. Pages with matching segments are left untouched.

## 4. Experimental results

The aim of our framework is to generate the smallest delta files possible to reduce the amount of energy consumed by the networking interfaces of the target low-power devices while receiving or transmitting the deltas. We performed experiments with the presented framework on three different platforms: 8-bit AVR microcontroller ATmega32u4 (2-byte word addressing), 16-bit MSP430 microcontroller (2-byte word addressing) and SoC with 32-bit ARM Cortex-M4 CPU (4-byte word addressing). We used the GCC cross-compiler ported for each platform. We compare delta file sizes generated by the Delta generator with the ones generated by R3diff [11] and RMTD [13] . The source code of the R3diff was published by the authors. We use different base firmware for each platform. The presented change cases build upon the base firmware. The change cases are the same for each platform. We tested our algorithms on 10 different firmware change cases:

1. Change a constant.
2. Add a function.
3. Alter a function.
4. Remove a function.
5. Add multiple functions.
6. Alter multiple functions.
7. Remove multiple functions.
8. Change order of some functions.
9. Add multiple functions and remove multiple functions.
10. Add functions, remove functions, change functions and change order of some functions.

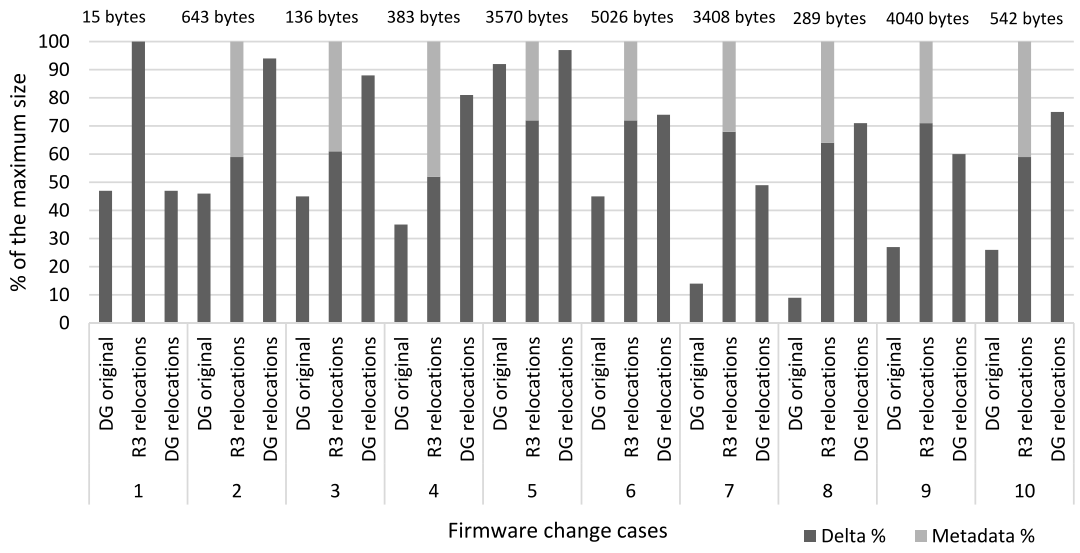


**Fig. 8.** Comparing standard differencing to methods with extraction of relocation entries.

### 4.1. Basic tests

Basic tests for each platform do not include any available optimizations. Firmware was compiled and linked with the default settings and no alterations to the linking command. Smaller delta files reduce energy consumption during the delta dissemination stage, especially if a single device needs to transfer one delta file multiple times. These tests show the improvement of R3diff and Delta generator over RMTD. In the most cases, Delta generator (DG) further improved over R3diff. The results of the tests for each platform are shown in the Tables 1–3. Delta generator is represented by three columns on the right side of each table. These columns represent:

- Delta file size generated by the Delta generator
- Delta size reduction compared to the full firmware image (often significant size reduction)
- Delta size reduction compared to the R3diff algorithm (mostly improved size reduction)

The results are following:

- ATmega — Firmware sizes under 2 KB. Up to 15 functions. DG performs 15% better in cases 7 and 8. It performs worse in the single case 2 and only by a small 2% margin. In all other cases, DG performs 3%–10% better.
- MSP430 — Firmware sizes between 0.6 KB–4.5 KB. Approximately 30 functions. In 8 cases, DG improves over R3diff with the improvement varying between 0.5%–53%. In two cases, 5 and 9, DG performs worse but only by less than 1%.
- ARM — Firmware sizes between 18 KB–57 KB. More than 5000 functions. In the last 5 cases, DG improves over R3diff by 4.5%–13%. In some cases, the results are very close with DG being better. R3diff performs better by 4.6% only in case 3.

### 4.2. Extracting relocation entries

We simulated the process described by R2 for testing of the configuration that extracts relocation entries from the firmware. We wrote a script that sets all relocation entries to the same value (empty word) and creates a metadata with relocation structures proposed in R2 [10]:

typedef struct{ int16_t r_offset, r_addr; } rela_t;

The results for the 10 firmware changes for MSP430 are shown in Fig. 8. We use percentages to compare delta sizes to the worst-case scenario which is always the result of R3diff with metadata. The worst-case size is shown above the columns. For each change case, we show the results of our algorithm (DG original, results from Table 2), the results of R3 (R3 relocations, delta + metadata) and the results of our algorithm that used the relocation extraction approach (DG relocations). This experiment brought no improvements, the results for relocation extractions are significantly worse. We concluded that the firmware change cases we used were not able to show the benefits and more experiments are necessary. We will focus on this in our future work.

### 4.3. Partial and full fragmentation

To further reduce the energy required to transfer the delta files, their sizes can be further reduced by using partial or full fragmentation configurations. We compared the delta files sizes produced by the Delta generator from fragmented firmware with the original sizes where no fragmentation is used. The results for each platform are shown in Figures Figs. 9–11. The worst-case delta size is represented as 100% and the other fragmentation configurations are compared to it in percentages. In most cases, partial and full fragmentation resulted in the delta size reduction.

Note, that the full fragmentation mode was unsuitable for the ARM CPU. Providing 5000 functions with slop regions would consume too much memory. The results represent our additional improvement over R3diff, where the authors considered memory fragmentation as inefficient. Once the incremental updates are done, the programmer can configure our framework to generate a standard file and generate a delta that will defragment and clean the firmware.

The results for each platform show further delta size reduction for most of the change cases. In most scenarios, the additional reduction is at least ~10%. For ATmega, the reduction for some cases reaches 15%–35%. Full reduction is achieved for the case 8 with reordered sections. For MSP430, the achieved reduction if often around 15%. In case 10, the reduction is 60%. The reduction is the best for the ARM platform, often reaching 35% and more, 65% in case 6.

A delta for firmware fragmentation can be generated prior to any updates in the full fragmentation mode. A delta for firmware defragmentation can be generated after the incremental updates with partial or full fragmentation configuration are finished. We used these fragmentation/defragmentation delta files and calculated total sum of the delta files sizes used to complete the presented update process. The data are shown in Table 4. Row *Fragmentation* shows the sizes of the delta files required to fully fragment the firmware before updates with full fragmentation. Row *Update files sum* shows the total sizes of the delta files required to complete the firmware update cases. Row

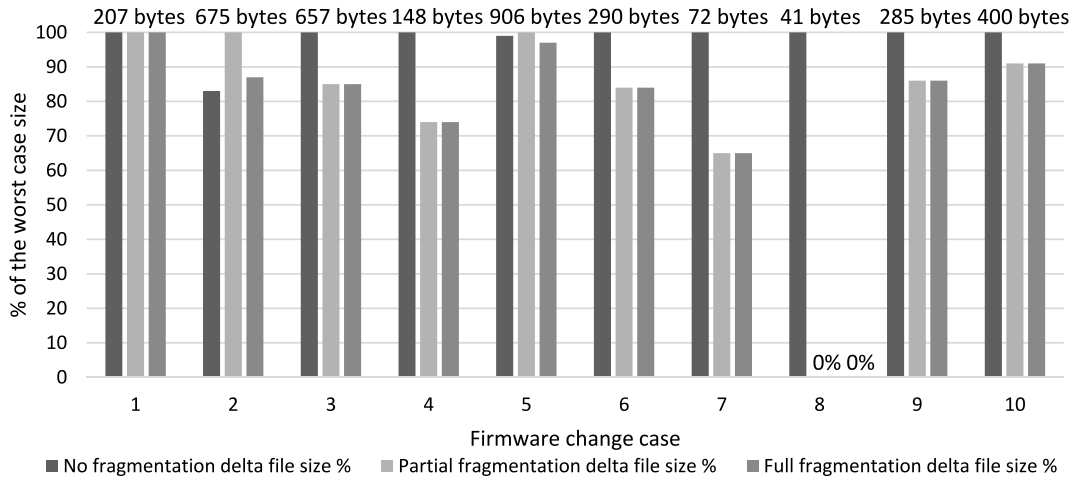


**Fig. 9.** Partial and full fragmentation configurations (ATmega32u4)

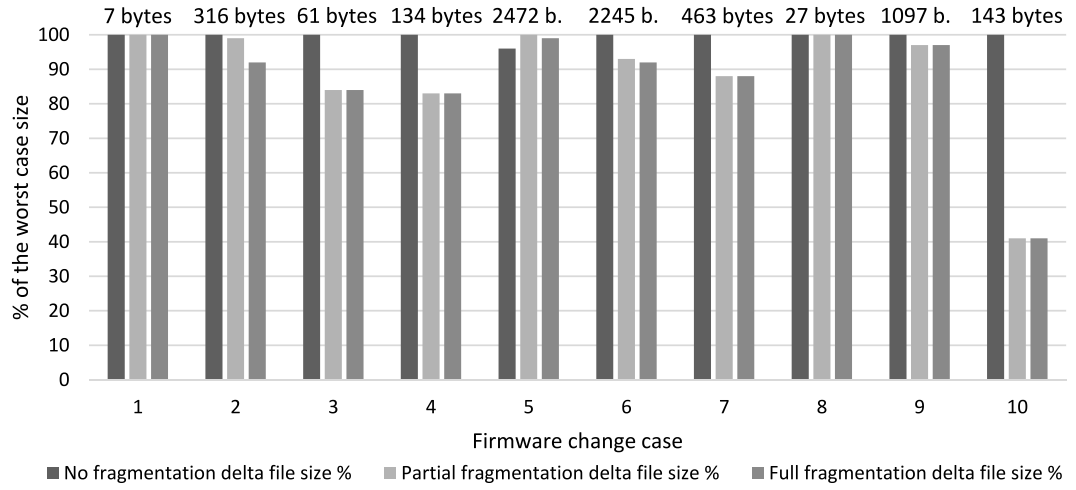


**Fig. 10.** Partial and full fragmentation configurations (MSP430)

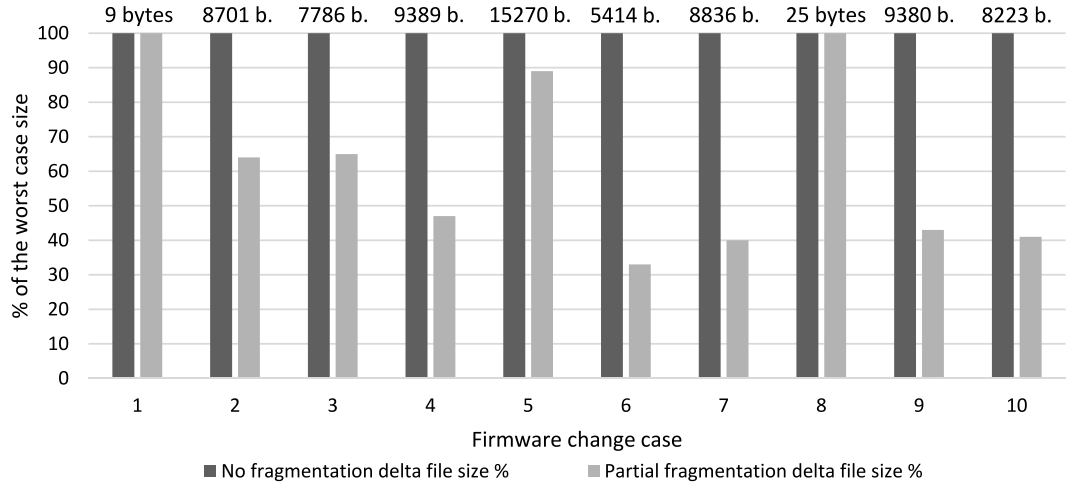


**Fig. 11.** Full fragmentation configuration (ARM Cortex-M4)

*Defragmentation* shows the sizes of the delta files required to defragment the firmware after the updates are done.

The results presented in Table 4 show, that for ATmega, the configuration without fragmentation is the best. For MSP430, the configuration with full memory fragmentation is the best. For ARM, partial fragmentation approach is the best achieving more than 45% reduction. This shows the advantage of our configurable approach. Different configurations may suit different platforms better.

**Table 4**
Sums of delta file sizes for different fragmentation configurations required to finish the updates (values are in bytes)

| Delta files | ATmega No frag. | ATmega Partial frag. | ATmega Full frag. | MSP430 No frag. | MSP430 Partial frag. | MSP430 Full frag. | ARM No frag. | ARM Partial frag. |
|---|---|---|---|---|---|---|---|---|
| Fragmentation | – | – | 286 | – | – | 50 | – | – |
| Update files sum | 3555 | 3348 | 3233 | 6859 | 6605 | 6531 | 73033 | 40816 |
| Defragmentation | – | 306 | 310 | – | 227 | 231 | – | 6285 |
| Total | **3555** | 3654 | 3829 | 6859 | 6832 | **6812** | 73033 | **47101** |

**Table 5**
Comparison between clean and dirty delta configurations for different platforms.

| FW change case | ARM (no fragmentation) | | | MSP430 (partial fragmentation) | | | ATmega (full fragmentation) | | |
|---|---|---|---|---|---|---|---|---|---|
| | Clean | Dirty | Reduced % | Clean | Dirty | Reduced % | Clean | Dirty | Reduc. % |
| 1 | 9 | 9 | 0% | 7 | 7 | 0% | 207 | 207 | 0% |
| 2 | 8701 | 8684 | 0,20% | 313 | 313 | 0% | 589 | 589 | 0% |
| 3 | 7786 | 7710 | 1% | 51 | 46 | 10% | 555 | 555 | 0% |
| 4 | 9389 | 9382 | 0,10% | 111 | 107 | 4,70% | 109 | 109 | 0% |
| 5 | 15270 | 15270 | 0% | 2472 | 2472 | 0% | 877 | 877 | 0% |
| 6 | 5414 | 5414 | 0% | 2089 | 2089 | 0% | 242 | 242 | 0% |
| 7 | 8836 | 8829 | 0,01% | 409 | 358 | 12,50% | 47 | 47 | 0% |
| 8 | 25 | 25 | 0% | 27 | 12 | 54% | 0 | 0 | 0% |
| 9 | 9380 | 9380 | 0% | 1067 | 1067 | 0% | 244 | 244 | 0% |
| 10 | 8223 | 8216 | 0,01% | 59 | 47 | 20% | 363 | 363 | 0% |

### 4.4. Removing padding operations

The additional size reduction of a delta file can be achieved by removing all operations that copy or add empty sequences. These are padding operations used to clean up the leftover data from the previous firmware versions. Our framework can be configured to look for any padding operations and to replace them with skips. The configuration produces interesting results achieving a significant reduction 4.7%–54% for MSP430, very small size reduction for ARM, and no delta size reduction for ATmega. The results are shown in Table 5.

## 5. Conclusion

The ability to update firmware of modern devices used in large-scale systems is a necessity. Benefits include easier management, increased security by security updates, bug fixes or addition of new features. We presented a new universal framework for remote firmware updates of networked embedded devices. Our work is oriented on the small embedded devices with limited power supply where the more efficient update process improves the battery life. Our framework introduces multiple configurations that reduce the sizes of delta files shared on a network during an update. This reduces the energy consumed by the networking interfaces of the updated devices. Additional improvement is the update module that rewrites each page of a program memory at most once per update. This preserves the durability of widely used NAND flash memories and reduces energy consumption caused by the erase operations. The main contribution of our framework is its universal use. The fact, that our algorithms work with the widely used standard ELF files and we tested it with the GCC compiler available for almost every platform, makes it suitable for modern heterogeneous systems that may use devices based on various platforms in their structure. We make no alterations to the machine code generated by the compiler and therefore the optimizations made by the compiler are preserved in the final firmware image.

## Acknowledgments

This work has been supported by Slovak national project VEGA 2/0155/19.

This work has been supported by the ECSEL Joint Undertaking (JU) under grant agreement No 737434.

## Conflict of interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References

[1] O. Kachman, M. Balaz, Configurable reprogramming methodology for embedded low-power devices, in: Technological Innovation for Smart Systems, Lisbon, 2017.
[2] J.W. Hui, D. Culler, The dynamic behavior of a data dissemination protocol for network programming at scale, in: Proc. of the 2nd Int. Conference on Information Processing in Sensor Networks (IPSN '08), ACM Press, Baltimore, 2004, pp. 81–94.
[3] W. Li, Y. Zhang, J. Yang, J. Zheng, UCC: update-conscious compilation for energy efficiency in wireless sensor networks, in: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2007.
[4] Y. Huang, M. Zhao, C.J. Xue, WUCC: joint WCET and update conscious compilation for cyber-physical systems, in: 18th Asia and South Pacific Design Automation Conference (ASP-DAC), Yokohama, 2013.
[5] R. Panta, S. Bagchi, S. Midkiff, Zephyr: Efcient incremental reprogramming of sensor nodes using function call indirections and difference computation, in: Proc. USENIX Ann. Technical Conf. 2009.
[6] R.K. Panta, S. Bagchi, Hermes: fast and energy efficient incremental code updates for wireless sensor networks, in: IEEE INFOCOM 2009, Rio de Janeiro, 2009.
[7] C. Zhang, W. Ahn, Y. Zhang, B.R. Childers, Live code update for IoT devices in energy harvesting environments, in: 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA), Daegu, 2016.
[8] J. Koshy, R. Pandey, Remote incremental linking for energy-efficient reprogramming fo sensor networks, in: Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.
[9] N.B. Shafi, K. Ali, H.S. Hassanein, No-reboot and zero-flash over-the-air programming for wireless sensor networks, in: 9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON), Seoul, 2012.
[10] W. Dong, Y. Liu, C. Chen, J. Bu, C. Huang, Z. Zhao, R2: incremental reprogramming using relocatable code in networked embedded systems, in: IEEE Transactions on Computers, vol. 62, IEEE, Shanghai, 2012, pp. 1837–1849.
[11] W. Dong, B. Mo, C. Huang, Y. Liu, C. Chen, R3: optimizing relocatable code for efficient reprogramming in networked embedded systems, in: IEEE INFOCOM Proceedings, IEEE, Turin, 2013, pp. 315–319.
[12] D. Wu, M.J. Hussain, S. Li, L. Lu, R2: over-the-air reprogramming on computational RFID's, in: 2016 IEEE International Conference on RFID (RFID), Orlando, 2016.
[13] J. Hu, C.J. Xue, Y. He, E.H.-M. Sha, Reprogramming with minimal transferred data on wireless sensor network, in: IEEE 6th International Conference on Mobile Adhoc and Sensor Systems (MASS '09), IEEE, Macau, 2009, pp. 160–167.
[14] H.U. Park, J. Jeong, P. Mah, Non-invasive rapid and efficient firmware update for wireless sensor networks, in: Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication, UbiComp '14 Adjunct, Seattle, 2014.

[15] B. Mo, W. Dong, C. Chen, J. Bu, Q. Wang, An efficient differencing algorithm based on suffix array for reprogramming wireless sensor networks, in: IEEE International Conference on Communications (ICC), IEEE, Ottawa, 2012, pp. 773–777.
[16] O. Kachman, M. Balaz, Optimized differencing algorithm for firmware updates of low-power devices, in: 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Kosice, 2016.
[17] X. Zhong, M. Navarro, G. Villalba, X. Liang, Y. Liang, MobileDeluge: mobile code dissemination for wireless sensor networks, in: 11th IEEE International Conference on Mobile Ad Hoc and Sensor Systems (MASS), Philadelphia, 2014.
[18] A. Tridgell, Efficient Algorithms for Sorting and Synchronization, The Australian National University, Canberra, 1999.
[19] Z. Zhao, Z. Wang, G. Min, Y. Cao, Highly-efficient bulk data transfer for structured dissemination in wireless embedded network systems, J. Syst. Archit. 72 (2017) 19–28.
[20] D. Park, M. Jung, J. Cho, Area efficient remote code execution platform with on-demand instruction manager for cloud-connected code executable iot devices, Simul. Model. Pract. Theory 77 (2017) 379–389.
[21] V. Kumar, S. Madria, Efficient and secure code dissemination in sensor clouds, in: IEEE 15th International Conference on Mobile Data Management (MDM), Brisbane, 2014.
[22] H. Asahina, I. Sasase, H. Yamamoto, Efficient tree based code dissemination and search protocol for small subset of sensors, in: IEEE International Conference on Communications Workshops (ICC Workshops), Paris, 2017.
[23] Z. Zhao, J. Bu, W. Dong, T. Gu, X. Xu, Coco+: exploiting correlated core for energy efficient dissemination in wireless sensor networks, Ad Hoc Netw. 37 (Part 2) (2016) 404–417.
[24] G. Jurković, V. Sruk, Remote firmware update for constrained embedded systems, in: 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), IEEE, Opatija, 2014, pp. 1019–1023.
[25] K. Lehniger, S. Weidling, M. Schölzel, 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), Budapest, 2018.
[26] J. Qiu, S. Li, B. Cao, Repage: a novel over-air reprogramming approach based on paging mechanism applied in fog computing, Wirel. Commun. Mob. Com. (2940952) (2018) 2018.
[27] J. Pallister, K. Eder, S.J. Hollis, J. Bennett, A high-level model of embedded flash energy consumption, in: International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), Jaypee Greens, 2014.
[28] O. Kachman, M. Balaz, Firmware Update Manager: A remote firmware reprogramming tool for low-power devices, in: IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Dresden, 2017.