School of Computer Science and Statistics

An Incremental Firmware Update System for Zephyr OS

Yue Yu

August 17, 2022

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MSc in Computer Science

Declaration

I hereby declare that this dissertation is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

Signed: Yue Yu Date: 17/08/2022

Abstract

The Internet of Things(IoT) edge devices has limited resources, such as limited storage, RAM, CPU performance, and bandwidth. However, the software that runs on them may occasionally need to be updated, and this kind of updating is a challenge. Since IoT devices are resource-constrained, this dissertation would like to explore an approach that reduces the update package size, improves the update efficiency, and considers this approach's robustness.

There are already many business solutions and research on this topic, but they mainly focus on monolithic updates, i.e., overwriting the previous firmware entirely. However, improving the current monolithic solutions is challenging to gain a better result because of many physical level constraints—for example, chips' frequency, Bluetooth bandwidth, and power supplies. Therefore, instead of working on current monolithic update solutions to approach the hardware performance limitation, we would like to find a different way to apply updates.

In this dissertation, we design three components to implement an incremental updating system. The first component is a reversed FAT file system called rFAT. This file system merges multiple device flash partitions into one, allowing one firmware to occupy more spaces and naturally support firmware downgrade. The next is a different algorithm, BSDiff-Inplace, to apply an incremental update. A different algorithm can generate a patch file, and the device can recover the new firmware by using this patch and the old firmware. Moreover, BSDiff-Inplace is explicitly designed for tiny devices, requiring less RAM than other difference algorithms like BSDiff, EXEDiff, and Courgette. Last, because flash chips only support page-wise read/write, we implement an EEPROM emulator to support random read/write.

Overall, this project accomplishes the main goals. For instance, users can generate a mini update patch, send it to a device, and apply it. However, the performance of BSDiff-Inplace can still be better, as its compression rate is only 50% of the compression rate of the original BSDiff. Besides, in the future, exploring modular updates or a hybrid of incremental and modular updates can improve the performance.

Acknowledgements

I would like to thank my supervisor, Dr.Jonathan Dukes, who gave me lots of advice while coding and writing this dissertation. Moreover, as a non-native speaker of English, he is always patient and enthusiastic. During this one year of study at Trinity College, I did not only learn technologies and knowledge from him but also learned English skills from him. Besides, I appreciate the Zephyr OS foundation and Nordic Semiconductor, which provided me the software and hardware to support my project.

Besides, I also want to show my gratitude to good friends I meet in Trinity, Nithin, Kevin, Marc, and Daanish. They helped me in both study and life, we had good teamwork, and I expect we have a chance to collaborate again in the future works. In addition, as we all experienced such a special and tough period, I would also like to show my respect to all Trinity College and HSE staff for guaranteeing our health in COVID-19.

Last but most importantly, I want to thank my mom. She always supports my computer science studies and believes I can be a good engineer. Even at this decreasing economic age, she entirely found my tuition fee and living cost in Ireland. We haven't met for more than one year, and I can't wait to come back home and see her again.

Yue Yu

University of Dublin, Trinity College

August 2022

Contents

1 Introduction 1
1.1 Context 1
1.2 Motivation 1
1.3 Challenges 2
1.4 Approach 3
1.5 Overview of structure 4

2 Literature Review 6
2.1 Firmware Similarity 6
2.1.1 Position-Independent Code (PIC) 7
2.1.2 Dynamic Linking 8
2.2 Difference Algorithms 8
2.2.1 Xdelta 9
2.2.2 EXEDiff 10
2.2.3 BSDiff 11
2.2.4 Courgette 12
2.3 Compression Algorithms 12
2.3.1 Lempel-Ziv family 12
2.3.2 Huffman family 13
2.4 Update Application 14
2.4.1 Inplace Updates 15
2.4.2 Reversible Updates 15
2.5 Summary 16

3 Design 17
3.1 System Architecture 17
3.2 Reversed FAT (rFAT) 18
3.2.1 Requirement 18
3.2.2 Design 19
3.2.3 Design Critique 20

3.3 BSDiff-Inplace 20
3.3.1 Requirement 20
3.3.2 Design 21
3.3.3 Design Critique 26
3.4 EEPROM Emulator 27
3.4.1 Requirement 27
3.4.2 Design 28
3.4.3 Design Critique 29

4 Implementation and Evaluation **30**
4.1 Prototype Implementation 30
4.1.1 rFAT backend 30
4.1.2 Bootloader 31
4.2 Workbench Implementation 32
4.3 Performance Evaluation 34
4.3.1 Metrics 34
4.3.2 Experiment Design 35
4.3.3 Result 35
4.3.4 Reflection 38

5 Conclusion **39**
5.1 Future Work 39

List of Figures

2.1 The essential stages in firmware updates [1] 7
2.2 Non-PIC vs. PIC 7
2.3 Dynamic linking example 8
2.4 The principle of EXEDiff. 10
2.5 Approximate Match Region in BSDiff 11
2.6 The Courgette workflow 13
2.7 The LZXX principle 14
2.8 A Huffman example of "helloworld!" 14
2.9 Devices that cannot load two firmware into memory or store on the disk. . . 15
2.10 Regular updates vs. Inplace updates 15
2.11 Reversible updates storage map for regular (top) and in-place (bottom) . . . 16

3.1 The system architecture overview 17
3.2 ROM map for nrf52dk running Zephyr OS without rFAT 18
3.3 Reversed FAT file system structure 19
3.4 Using file system to support rollback 19
3.5 Allocation list example 19
3.6 Code difference comparison 21
3.7 Code difference in ARM Assembly 22
3.8 The original BSDiff vs. the in-place BSDiff 23
3.9 Approximate match in BSDiff 24
3.10 Approximate match region details 24
3.11 An overlap causes forward reference happens 25
3.12 Block-wise compressing by LZ77 26
3.13 Block-wise decompressing by LZ77 26
3.14 Move data backward to insert extra string in BSDiff 27
3.15 Insert extra string into pre-reserved paddings 27
3.16 Flash vs. EEPROM 28
3.17 A design of EEPROM emulator for flash 28

4.1 The steps to launch applications in MCUBoot 32
4.2 The steps to establish a IPSP node 33
4.3 Transmit a patch to a device by TCP 33
4.4 LZ77 compression rate with different block size 37

List of Tables

2.1 The example of XDelta, applying a patch on an old file 10

3.1 The rFAT file system header format 20

3.2 An example of difference string in BSDiff 21

4.1 The size patches produced by BSDiff and BSDiff-Inplace 36

4.2 The size patches produced by BSDiff-Inplace with different block sizes 36

4.3 The MCUBoot memory map 36

4.4 The life cycle cost on different firmware 38

1 Introduction

1.1 Context

The Internet of Things (IoT) edges are frequently populated by small embedded computers with minimal resources (limited flash storage, limited RAM, CPU performance, and communication channels). The software that determines the behaviour of these devised may occasionally need to be modified, for example, to correct erroneous behaviour, improve performance, address new vulnerabilities, adapt to the introduction of new technologies or otherwise modify the device's behaviour. Modifying or updating the firmware on remote, resource-constrained devices remains a significant challenge. This dissertation will explore an approach for reducing the update package size, improving the update efficiency, and considering the robustness of this approach.

1.2 Motivation

Nowadays, firmware updates are becoming more critical than before. On the one hand, these devices are widely deployed and take an important position in society. For example, traffic lights, electronic bus timetables, smart home sensors, and manufacturing machines are derived from IoT chips. Thus, they are profitable to hackers, and security OTA updates are essential. Moreover, new real-time operating systems like FreeRTOS and ZephyrOS finish their hardware abstraction layer, allowing embedded device engineers to focus on functionalities rather than the hardware itself. Therefore, the lower development costs allow companies to offer products with continuous functionality updates.

However, firmware updates are complex. According to the study from Konstantinos Arakadakis [2], network bandwidth and ROM size are the two main bottlenecks. First, most SoCs equip 2.4GHz Bluetooth Low Energy (BLE) chip, allowing up to 256KB/s [3]. Nevertheless, in practical applications, it will be affected by various reasons, such as bidirectional transmission, protocol header, CPU capability, and RF hardware limitations. The speed will be much lower, around 10 to 50KB/s. Besides, the flash size also constrains firmware updates. For reliability reasons, developers usually implement a rollback function

on devices because most users are unlikely to use wired flashing to recover a device once an update fails. This rollback function is mainly implemented by storing an old copy on the device. It means a firmware can use only half of the flash, which is challenging as this kind of device usually have less than 1MB of storage. In this dissertation, the targeting platforms are Nordic nRF52DK-52832 and nRF52840DK, using 2.4GHz Bluetooth as the network access. The bit rate of which in a real environment is usually around 30KB/s, and the flash size are 512KB and 1MB respectively.

According to the above data, how to optimize the process of transmitting update packages to a group of devices and applying the update is a valuable topic. To achieve the goal, the following approaches are commonly used:

(1) **Monolithic updates**, which are the most widely used and most straightforward way, overwrites the old firmware.

(2) **Modular updates**, which place library codes on different places and libraries are compiled position independently. Therefore, instead of replace the firmware, we can only update the module we want to update.

(3) **Incremental updates**, which generate a difference patch from the old and new firmware and the old firmware runs on devices can generate the new firmware by this patch.

(4) **Hybrid updates**, which are also possible. For example, we can incrementally update a module, which has potentially better performance than modular or incremental updates.

In this dissertation, we will explore an approach based on incremental updates. Because the monolithic updates do not have much space to improve, we can get an instant benefit from reducing the update package by using incremental updates, and modular updates are more complex than incremental updates, requiring special executable format design. By following the executable binary difference research [4] from Colin Percival, making an incremental patch can reduce 90% the update size on average, and as a bonus, firmware can utilize more flash space than a reversible monolithic update.

1.3 Challenges

In production environment, most solutions are still based on monolithic updates, and the reasons can be attributed to the limited flash capacity, limited flash life cycles, limited RAM, limited CPU performance, and identifying suitable algorithms.

First, the limited flash capacity forces us to use in-place updating because putting the old and new firmware into two slots is already challenging; finding another place for the patch is much more challenging. However, we can gain much free space if we use in-place patching.

Nevertheless, the cost is flash life cycles as the in-place operating is always with memory moving.

Furthermore, the RAM size also limits us, i.e., we cannot read the full firmware into memory, patch it, then write it back to the flash. Thus, our algorithm need to support stream-like data I/O. Meanwhile, we must consider the CPU performance, avoiding complicated calculations.

Last, algorithm selections are also worth-talking. For example, the difference algorithm can be Xdelta [5], EXEDiff [6], and BSDiff [4]; the compression algorithm can be BZip2 [7], GZip [8], and LZ77 [9]. There are no perfect answers, and we need to trade the CPU and memory costs.

1.4 Approach

The current firmware update solutions are all monolithic update based. However, we need to integrate an incremental update feature for a bootloader to overcome the above issues. In this dissertation, we will take mcuboot and ZephyrOS as an example because they have official support from the manufacturer of the prototype devices used for this work, Nordic Semiconductor.

The following key requirements are expected to be in our scope, as the main contributions of this dissertation,

(1) **Incremental update**, which means users only need send a part of the new firmware and the device can generate the entire new firmware on the device.

(2) **Rollback**, which allows users downgrade a newly updated firmware to the previous version. It provides reliability to our firmware.

(3) **Flash optimization**, which aims to reduce the number of bytes to write to increase the flash life cycle.

Besides, there are some interests that we would like to investigate but not at this time since they are a bit out of the scope of incremental firmware update, such as network issues, security, and privacy.

The below list is a brief of our approaches to implement the incremental update system,

(1) **Block Patch**. An incremental patch will be organized as blocks, allowing multiple blocks to be stored by a device simultaneously. Blocks are independent and can be applied separately. Besides, already applied blocks can be released from RAM and even the file system. In this way, we can utilize the limited RAM more efficiently. However, a

possible cost is that frequent block I/O will slow the patching progress and spend more flash lifetimes.

(2) **Dynamic Partition**. It is the key to incremental updates on IoT devices. Because the existing solutions from Zephyr OS require a two-partition on small devices, and they are also widely used by other real-time operating systems like freeRTOS [10] and RT-Thread [11] , even though the other partition is unused, the one partition still cannot exceed the partition boundary. The smaller update patch enables this dynamic partition solution to allow a higher firmware flash memory usage. In our solution, applying a patch will be in place to save the flash memory.

(3) **Decremental Patch**. The decremental patch is a part of reliability that can give the device a mini patch from the newer version to the older version, that is, a replacement of a full-size rollback firmware image. Usually, the incremental patch is generated by comparing the newer to the older image. The decremental patch can be generated in the reversed way, generated by the difference from the older to the newer image.

In this dissertation, the incremental update workflow on tiny devices will be illustrated, including the process of uploading a patch to a device and applying the patch, the process of rollback, and the process of resuming an uncompleted update. Moreover, the multiple metrics, e.g., the time of applying a patch, the compression rate of a patch, and the number of bytes being read/written, will also be described to measure and judge the performance of this work.

1.5 Overview of structure

This dissertation consists of five chapters as list shows:

(1) **Introduction**, Chapter 1, which is this chapter, presenting the context and motivation, approach, and structure overview.

(2) **Literature Review**, Chapter 2, which composes of surveys of key technologies in this dissertation, including the firmware similarities, binary difference algorithms, compression algorithms, and update applications.

(3) **Design**, Chapter 3, which illustrates the detailed system architecture. This chapter is organized with an introduction to the architecture, followed by the specially designed storage system, incremental patch algorithm, and prototype adapter for this system.

(4) **Implementation and Evaluation**, Chapter 4, which includes workbench implementation, prototype implementation, and performance evaluation.

(5) **Conclusion**, Chapter 5, which summarizes the work done and the result against the

original requirements and discussing the future works.

2 Literature Review

This chapter aims to make a review of crucial mechanisms in firmware updates, and sections will be organized as Figure 2.1 [1] shows:

(1) **Firmware similarity**, Section 2.1, which is a step to discover more chances to optimize the patch size by comparing the old and new source code.

(2) **Difference algorithm**, Section 2.2, which compares the binaries from the old and new source code and generates a patch file. With this patch, a device can build the new firmware locally without the new source code and binary. Generally, a good difference algorithm should minimize the patch size to reduce the transmitted bytes over the network.

(3) **Update application**, Section 2.4, which is the last stage for installing the new firmware. Moreover, update applications are also responsible for firmware reconstructing and verifying.

Besides, dissemination, which is also in Figure 2.1, will be skipped because it is out of scope. This stage is to make a reasonable choice from various types of networks based on availability, reliability, and mobility. However, we focus on the local side in this dissertation. Moreover, Section 2.3 about compress algorithms will also be included as an extension part of difference algorithms. Since the difference algorithms are not responsible for compressing data, a good compress algorithm can help difference algorithms achieve a better compression ratio.

2.1 Firmware Similarity

Firmware similarity is not a new topic nowadays [12]. In the last century, the age that people use limited memory and disk PC, the IBM System/360 (7 April 1964) was designed with truncated addressing which is the prototype of the modern position-independent code mechanism(Section 2.1.1). Furthermore, this idea also contributes to dynamic linking in Section 2.1.2, which targets to reuse of object files for different programs and to update a part of a program instead of the whole.

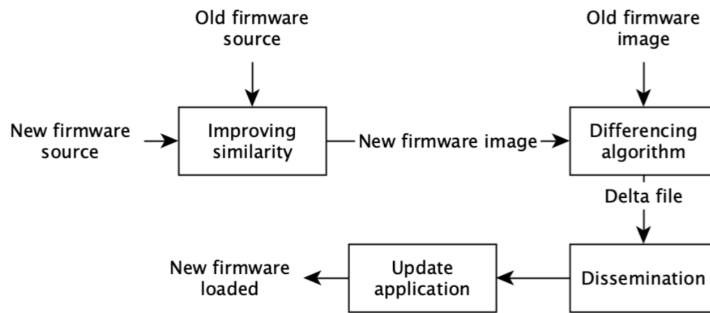

Figure 2.1: The essential stages in firmware updates [1]

2.1.1 Position-Independent Code (PIC)

Position-independent code is a compile-time mechanism to generate position-independent assembly code. This mechanism is initially designed for shared libraries because a compiler does not know where a shared library will be loaded. However, since other benefits of PIC are realized, like anti-hook and improving flexibility, compilers can also generate PIC for executable, so-called position independent executable (PIE).

The key to PIC is the global offset table (GOT). In general, it is an assembly section with shared objects information and placeholders for external symbols, e.g., external functions and variables. In this way, as Figure 2.2 displays, it is possible to convert a function call with absolute address like "call <addr-to-function>", to a PIC function call like "call <addr-to-got>". The GOT items are initially empty and will be filled by OS.

The most important is that PIC can also smaller the delta. In binary differencing, many changes are caused by address changes. Therefore, putting addresses into the global offset table and converting calls to PIC calls can remarkably reduce the address changes. The PIC is a mature technology and is enabled in default in mainstream compilers such as GCC, Clang, and MSVC.

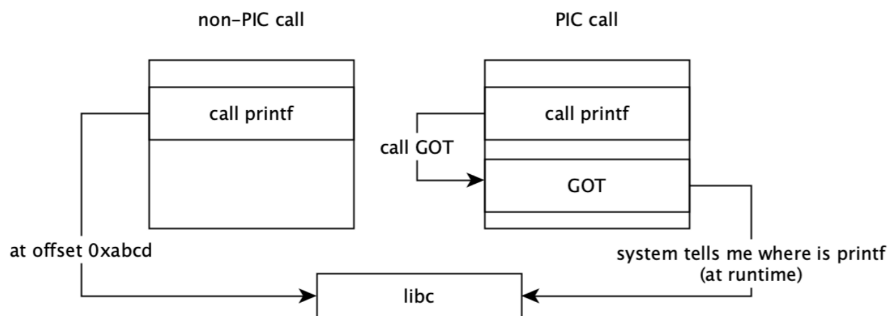

Figure 2.2: Non-PIC vs. PIC

2.1.2 Dynamic Linking

The legacy programs were statically linked, meaning that different compiling units (different .c files) are integrated into a monolithic output. It is simple and still widely used. However, two possible issues come with increasing software projects:

(1) **Inflated executable**, which asks users to update software monolithically, in the age that incremental update does not exist, the age that people use dial-up Internet access.

(2) **Limited storage**, which was about 10KiB in the mid-1950s. To save storage, programmers reused object files for different programs at runtime. This reuse technology is dynamic linking.

For instance, as Figure 2.3 shows, they can reuse the instance from the system to save the storage instead of owning their OpenSSL copy. Nevertheless, if there is any update of OpenSSL, users only need to update once for all applications. They only need to update the OpenSSL lib, e.g., libssl.so on Linux, rather than update many application bundles.

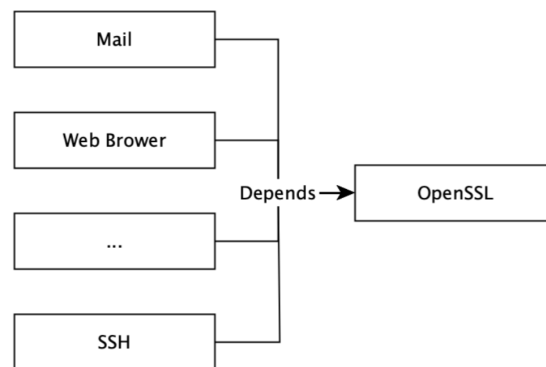

Figure 2.3: Dynamic linking example

2.2 Difference Algorithms

A difference algorithm is a kind of algorithm which allows users to generate a patch from a pair of an old and a new file. Besides, users can use this patch and the old file to recover the new file. Generally, this kind of algorithm is commonly used in a software update and document version management system, i.e., rsync, svn, and git.

A widely accepted opinion is that plain text and binary files are remarkably different, and we should treat them differently in difference algorithms. For instance, in git, instead of logging incrementally, binary files will be logged as "the new file replaces the old file monolithically". Although there are some plugins to support binary diff for git, this kind of behavior still

approves that the text-based-oriented difference algorithms are not suitable for binary files. At least, the performance is not as good as they work on text files.

In this section, we will explore the history of difference algorithms. First, section 2.2.1 introduces Xdelta, a simple and widely-used difference algorithm. Then Section 2.2.2 follows, talking about EXEDiff, a platform-related, executable-file-oriented difference algorithm. Besides, BSDiff, a binary-file-oriented, platform-unrelated algorithm, will be given in Section 2.2.3. The last part of this section, Section 2.2.4, will discuss Courgette, a difference algorithm based on disassembling the target new file.

2.2.1 Xdelta

Xdelta is a widely used differential algorithm. The first version, Xdelta 1, was based on rsync as a part of PhD thesis [13] of Andrew Tridgell. The latest release is Xdelta 3, organized in VCDiff [14] format.

Xdelta3 has three instructions, ADD, COPY, and RUN. Assuming O and T are the strings of the old file and target file, respectively, and the length of the old file is L_O, the three instructions can be explained as:

(1) $ADD(X, S)$: Copy a string S, the length of which is X, to the current position of T.

(2) $COPY(X, Y)$: When $Y < L_O$, copy the following X characters from $O[Y]$; When $Y \geq L_O$, copy X characters from $T[Y - L_O]$.

(3) $RUN(X, Z)$: Append character Z for X times on the current position of T.

Table 2.1 shows a detailed example. The old string is "abcdefghello," and the patch is listed, then we need to build the target, i.e., recover the new string. Firstly, we append "abcd" to the empty target because "COPY 4, 0" copies $O[0..4]$ to T. Next, we need to add "xdelta" to the target end since the "ADD" instruction. Then, the "COPY 5, 7" requires us to copy $O[7..12]$, that is, "hello," to T. Because 22 in the following "COPY 5, 22" is over L_O, we start to find the T itself and copy it, which is also "hello." Finally, we run "RUN 3, !" which means append three exclamation marks to T. Therefore, the final result is "abcdxdeltahellohello!!!".

Overall, Xdelta is a straightforward algorithm. It performs well in plain text because these three instructions can represent what we will do for a document. However, the changes in a binary file are more complicated. Usually, we only edit source codes, and a compiler generates a binary file. Therefore, even a tiny change in codes can cause a significant change in the output binary. Though Xdelta can work with binary files, it can still be better.

2.2.2 EXEDiff

To improve the performance on binary files, EXEDiff is raised [6] by Baker et al. This algorithm uses two operations called pre-matching and value recovery to reduce the influence caused by code changes. Pre-matching aims to find out the relationship between the old and new files. This process is heuristic, assuming the program does not know the source code. Moreover, the value recovery targets identifying changes that can be implicitly stored in the patch.

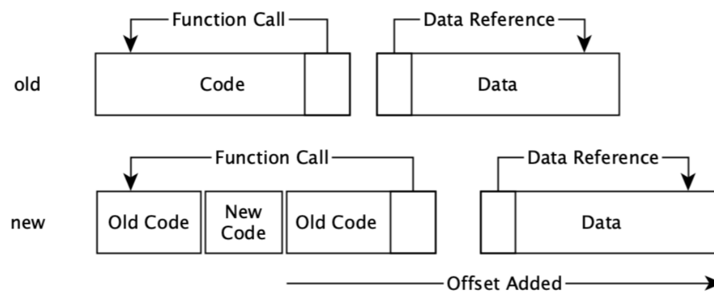

Figure 2.4: The principle of EXEDiff.

As Figure 2.4 shows, every new code block will influence the following address values, including functions and data references. Therefore, EXEDiff uses the pre-matching and value recovery mechanisms to calculate an approximate offset and try to minimize the delta size (address changes).

However, this process is highly platform depended, based on the prior knowledge of a specific machine architecture (like x86, ARM, and RiscV) and operating system. It was initially developed and tested on x86_64 UNIX Alpha and can be adapted to other platforms. Nevertheless, developing a new version for a new platform takes time.

To summarize, according to the research [15] by Giovanni Motta et al., both the patch size and the patch size after BZ2 compression of EXEDiff is 50% of XDelta. However, the cost is

Old	a b c d e f g h e l l o
Patch	**Instructions** COPY 4, 0 ADD 6, x d e l t a COPY 5, 7 COPY 5, 22 RUN 3, !
Target	a b c d x d e l t a h e l l o h e l l o ! ! !

Table 2.1: The example of XDelta, applying a patch on an old file

that we have to adapt EXEDiff for different platforms, which can be expensive.

2.2.3 BSDiff

BSDiff [4] is designed to solve the cross-platform issue of EXEDiff. It also follows the principle given in Figure 2.4 but sets the target to finding an approximate match region instead of counting offset and minimizing the delta size.

Figure 2.5 gives an overview of the approximate match. In each iteration, BSDiff will find one approximate match region, which contains over 8 bytes difference and is on the left side(smaller index) of an exact match region. Every approximate match region consists of three parts, as the figure shows. The forward and backward extensions are required to match over 50% content between the old and new strings. The middle part, which can be empty, will be skipped in the old string and is extra in the new.

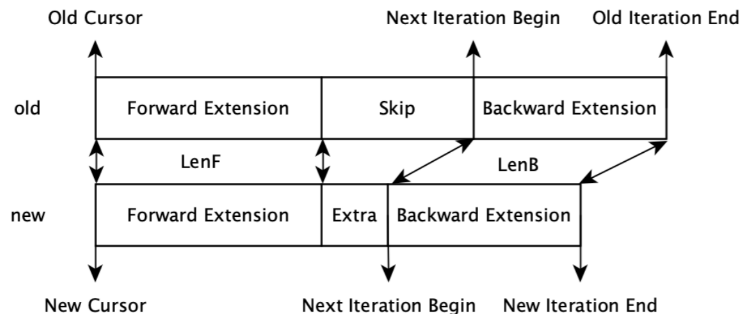

Figure 2.5: Approximate Match Region in BSDiff

Initially, the old and new cursor in Figure 2.5 starts from 0. After an iteration, they will be set to $OldIterationEnd - LenB$ and $NewIterationEnd - LenB$ respectively. *OldIterationEnd* is the position of the longest common string, which is from $Old[NewCursor]$, and the *NewIterationEnd* equals $NewCursor + Len$ where the *Len* is the length of the above longest common string. Besides, the difference threshold is 8 bytes because 8 bytes can be considered an address change on 64 bits machines.

BSDiff stores the difference values between the old and new for forward extension, i.e., it stores "001" if the old is "abc" and the new is "abd", because the 'd' is one greater than 'c' in ASCII. Moreover, the extra part will be saved in the patch file. As you may notice, BSDiff does not compress the delta since it tries to save every difference. However, most changes in a binary file are address values. Therefore, the diff strings generated by the old subtracting the new are mostly filled with zero, which means the diff strings are sparse and highly compressible.

In conclusion, BSDiff does not try to calculate an offset and use it to organize the patch file.

Instead, it tries to construct a sparse vector to represent the delta. According to Giovanni's research, BSDiff 4 is sightly worse than EXEDiff in compression rate. On the other hand, BSDiff 6, based on Colin's doctoral thesis, uses a sophisticated algorithm and offers about 20% smaller patches and almost the same compression rate as EXEDiff. Besides, the most important thing is that BSDiff solves the cross-platform issue in EXEDiff.

2.2.4 Courgette

Courgette [16] is a part of Chromium Projects used to generate patches for Google Chrome. Courgette follows the idea of EXEDiff and moves a step forward - the developers focused on solving the offset issues. They believed that, in source code, all the entities are symbolic until the assembly or linking stage, which means it is possible to avoid offset issues if we can revert an object file to an earlier stage.

Figure 2.6 illustrates the workflow of Courgette. The developers made a "disassembler" that can split an object file into three parts: a target address list of internal pointers, all the other bytes like resources and media data, and an instruction sequence. Once we have a disassembled result, we need to run the "adjust" step, which means reconstructing a symbolic table in Courgette. Then, we can use the disassembled and adjusted new to generate a patch. Google claims that the patch is 30% smaller than the directly generated by BSDiff.

Summarily, Courgette learns from EXEDiff and also uses BSDiff. Thus it gains an averagely 30% smaller patch file. However, it also accepts the backwardness of EXEDiff. Namely, this algorithm is highly architecture related. Because Courgette is for Chromium which runs on specific platforms, this trade is understandable but unsuitable for tiny devices. Furthermore, assembling the patch and old files on IoT devices is too expensive. Because this means that we must make an assembler, like GCC, run on a small device.

2.3 Compression Algorithms

Compression algorithms can be mainly distinguished into two types, the Lempel-Ziv family and Huffman family, which are dictionary and sliding window based, and Huffman tree based, respectively. The Lempel-Ziv family will be introduced in Section 2.3.1, and Section 2.3.2 discusses Huffman-based compress algorithms.

2.3.1 Lempel-Ziv family

Figure 2.7 demonstrates the principle of Lempel-Ziv family compress algorithms. The main idea is to find the longest common substring in a sliding window and use references instead of saving the literal data. For example, LZ77 [9], the most initial version of the LZXX

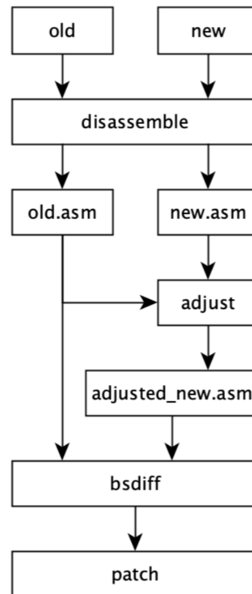

Figure 2.6: The Courgette workflow

family, uses a triplet (x, y, z) to mark a reference region, which means moving to the x-th byte in the sliding window, copying y bytes from here, and go to the next byte equal to z in the incoming data. However, we may not benefit from a reference because a reference tuple usually takes more than 3 bytes. Therefore, an improved algorithm called LZSS [17] was developed on 1982, compares the compressed and uncompressed data, and uses the smaller one to make sure the data is actually "compressed." Moreover, the reference tuple in LZSS becomes (x, y), meaning moving forward x bytes, copying y bytes from here, then stepping backward y bytes in the incoming data, which saves one byte for every reference tuple. Besides, there are other variants of LZ77, like LZ78 [18], LZW [19], and Deflate [8], which also hold the same principle and will not be introduced in detail here.

Overall, LZXX algorithms are sliding window based, so we do not have to have full access to the data, which is memory friendly. However, the limited buffer also limits algorithms to find more compressing chances. Therefore it averagely performs a lower compression ratio than Huffman families, which will be illustrated later.

2.3.2 Huffman family

Huffman encoding [20] is a variable length encoding method based on Shannon's information entropy theory [21]. It is a bit-wise character compression by the character frequency. For instance, as Figure 2.8 shows, in the string, "helloworld!", "l," which appears three times, is the most frequent character. Thus, it is encoded into two bits, i.e., we use two bits to represent "l" instead of 8 bits in ASCII.

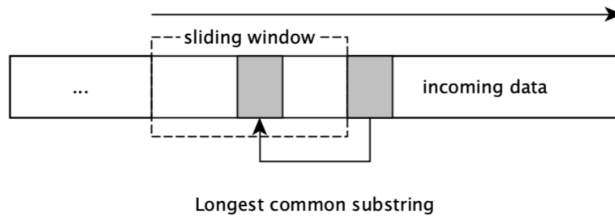

Figure 2.7: The LZXX principle

Because Huffman encoding is a global-wise compression method, it can have more compression changes than LZ-based methods. It is widely used by compression software like bz2, gzip, and 7-zip. However, the weak point of Huffman encoding is that we must go through the whole data set to generate the Huffman tree while compressing. Besides, when decompressing, we also need to load the whole tree, which is not friendly for tiny devices.

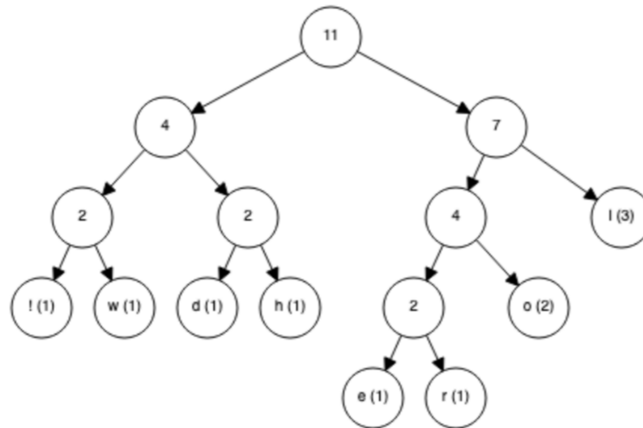

Figure 2.8: A Huffman example of "helloworld!"

2.4 Update Application

Updating application is the application that runs on devices to install new firmware. Generally, it is a part of the bootloader or operating system. The functionalities in update application are changing. In this case, we take inplace updates and reversible updates into account. Section 2.4.1 states inplace updates for IoT devices. Then, reversible update is in Section 2.11.

2.4.1 Inplace Updates

Inplace update is a way to apply updates on storage and memory-limited devices (Figure 2.9). Instead of creating a new firmware instance, the in-place update mechanism directly works on the old firmware. However, it is also dangerous since the old firmware will be unusable and hard to recover once an exception happens.

Besides, a challenge is in Figure 2.10 that a regular incremental update may read reference out of order, e.g., in the figure, the blue block can reference the data front of the green block (reference 1) in regular incremental update (left side). Unfortunately, it is incapable of in-place update because the new firmware already overwrites the data backward. As the right part of the figure, the old and new share the same memory, and the blue block can only reference data behind the index $len(ref1) + len(extra1)$.

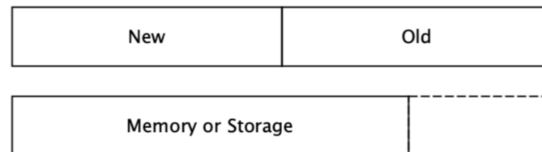

Figure 2.9: Devices that cannot load two firmware into memory or store on the disk.

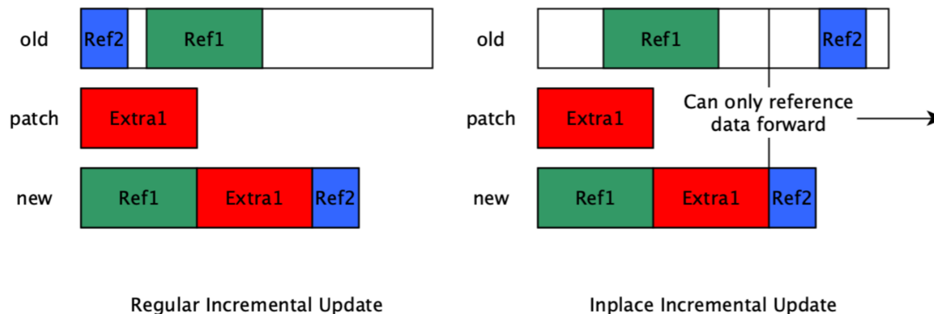

Figure 2.10: Regular updates vs. Inplace updates

2.4.2 Reversible Updates

Figure 2.11 demonstrates the storage map for non-inplace and inplace update mechanisms. The non-inplace version is achieved simply by saving two different versions on the disk and using a bootloader to select which firmware to boot. This design gives a guarantee to users. That is, when the new firmware does not work, instead of letting the device down, users can downgrade to the earlier version to recover the device.

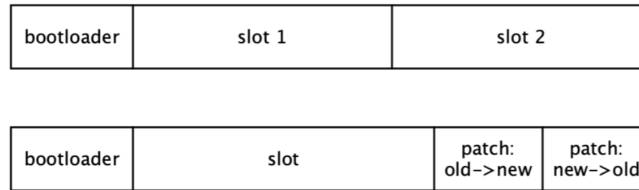

Figure 2.11: Reversible updates storage map for regular (top) and in-place (bottom)

However, challenges exist if we want both to build reversible updates and keep a sizeable continuous firmware space:

(1) **CPU performance**, the first thing we need to consider. Since complex calculations are complicated for IoT devices, choosing a lightweight patching algorithm is essential.

(2) **Memory limit** is another issue. In IoT devices, we cannot read all of the firmware data into RAM. For example, the sizes of firmware given in Section 4 are around 200KB, whereas the RAM size is 32KB. Therefore, using in-place incremental updates is critical.

(3) **I/O speed** becomes an issue after we decide to use in-place incremental updates because, in this kind of update, it is unavoidable to move blocks back, which is a massive task for the I/O system.

2.5 Summary

Overall, we have three main methods to enhance the firmware patching process, increasing firmware similarities, using better difference algorithms, and developing a better update application (the patcher). Besides, we go through the popular solutions in these three domains, and in this dissertation, we will focus on BSDiff, LZ77, and building a good update application.

3 Design

This chapter starts with the system architecture in Section 3.1, illustrating how components in this dissertation work together. Then, it introduces three components with trade-offs and critique considerations,

(1) **Reversed FAT**, in Section 3.2, is a file system designed for this dissertation. It merges slots to offer more space for the current firmware. Moreover, it natively supports saving incremental and rollback patches as they are treated as regular files.

(2) **BSDiff in-place**, in Section 3.3 , is the core, solving the problem of memory limit as it does not require loading the full firmware into RAM.

(3) **EEPROM emulator**, in Section 3.4 provides an adapter for the above two components to access the flash on devices. Because flash chips do not allow random access physically, we need a software emulator to support random access.

3.1 System Architecture

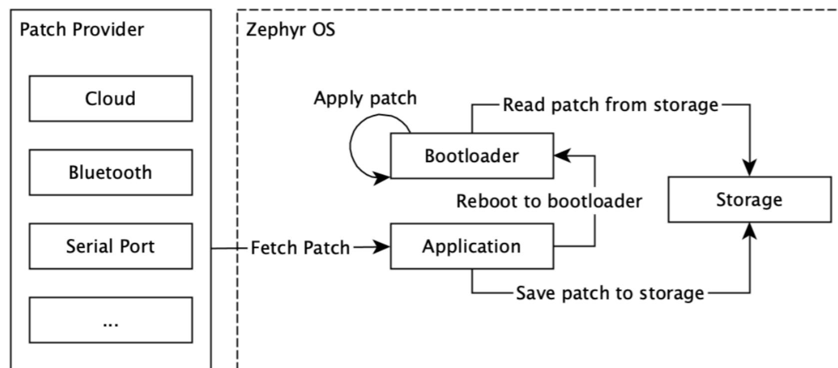

Figure 3.1: The system architecture overview

The system architecture is given in Figure 3.1. As the figure displays, the bootloader is only responsible for applying a patch and does not care about where to download the patch. Besides, downloading a patch is the task of the application. There are reasons to use this

design. First, we follow the K.I.S.S. rule, i.e., keep it simple and stupid. So, a bootloader should not care about things like what application runs on it, what functionalities the application has, and where to download the application updates. Furthermore, the applications are changing, but the bootloader is constant. In general, allowing the bootloader to download updates means we need to compile bootloaders for different applications, which is not a clever idea in most cases. Nevertheless, on the hardware side, the limited storage also does not allow us to save duplicated network libraries.

After downloading, the patch will be placed in the storage, which is a specially designed file system that will be explained in detail in Section 3.2. Once the application decides to update, it reboots to the bootloader, and the bootloader will discover the patch in the storage. Then, the bootloader will use BSDiff-inplace, an improved BSDiff algorithm on memory occupation, to update. In the last step, the bootloader reboots and jumps to the new application.

3.2 Reversed FAT (rFAT)

3.2.1 Requirement

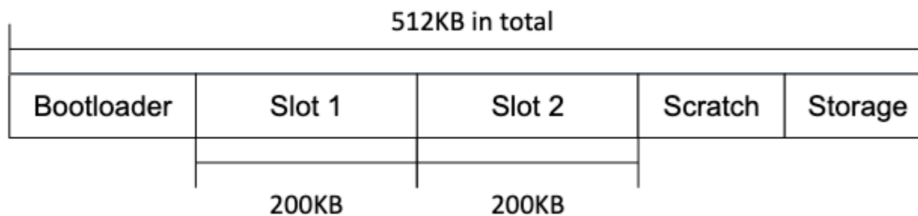

Figure 3.2: ROM map for nrf52dk running Zephyr OS without rFAT

As the bootloader we use, MCUBoot reads the firmware at the beginning of a firmware slot, and this behaviour is address-dependent, which means we cannot move the firmware in the slot. Thus, to achieve an elegant way to integrate slots like Figure 3.2, putting the FAT header at the end of the slot is a solution. This figure comes from the Zephyr, but it is also a typical flash ROM layout for other embedded devices on other operating systems. Because manufacturers want reversibility for their products, the most straightforward way is to keep two slots in its lash, one for the current firmware and the other for the old firmware. Besides, to reduce flash cost (the header size) and to decrease the write cost, we set the block size to 4KB (flash page aligned) because when the block size is less than a page, even if we only want to modify one block, we are going to change multiple blocks in a page. Additionally, we will not use the scratch and storage partitions as they are for the application.

3.2.2 Design

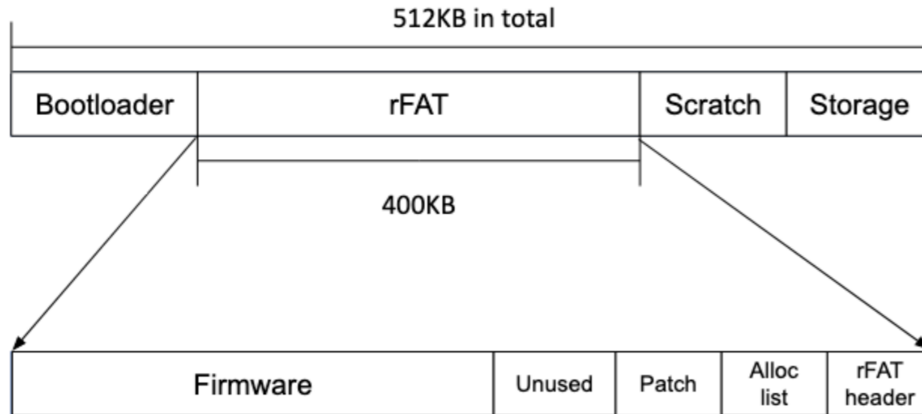

Figure 3.3: Reversed FAT file system structure

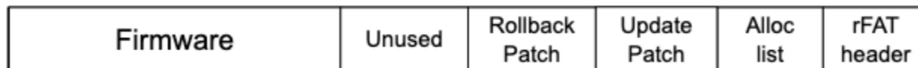

Figure 3.4: Using file system to support rollback

Alloc List						
Index	0	1	2	3	4	5
Next	0	0	4	5	3	0

Figure 3.5: Allocation list example

Figure 3.2 and 3.3 are the ROM structure on Nordic nrf52dk with and without rFAT respectively. As Figure 3.2 displays, the firmware runs without rFAT should be less than 200KB. But firmware runs on rFAT (Figure 3.3 can be up to 400KB. Though, in most cases, it is less than 400KB due to the header and allocation list occupations, the possible maximum size is still more than the firmware in Figure 3.2.

Moreover, using a file system can also bring us the rollback function without extra pay. The idea is given in Figure 3.4, which is putting the rollback patch file as a regular patch like the update patch, and the device can choose which patch it will use. The only difference between a rollback patch and an update patch is that the rollback patch is generated from the new firmware to the old firmware, and the regular update patch is reversed.

Besides, the structure of header is on Table 3.1. The magic number is used to identify whether this slot is treated as an rFAT file system. The block_cnt field saves how many

blocks this file system has (each block is 4KB). Then, the firmware_block_cnt logs the count of firmware blocks, which stands for firmware occupation and only exists at the beginning of the slot. The "file_entries" store files' entries of this file system, up to 16 files. Finally, the alloc_list is used as an array-based linked list, representing the file blocks. For example, as Figure 3.5 shows, the "index" is the index of the block in the file system, starting from the end of the slot. The "next" means the file's next block, and zero stands for "end of file" (EOF). Therefore, index 0 will never be used to remove the ambiguous meaning. We can find two files in this example, one starts from index one and takes only one block, and the other is a 4-block length file, $2 \rightarrow 4 \rightarrow 3 \rightarrow 5$.

3.2.3 Design Critique

Although the rFAT is an elegant way to support a bigger firmware slot and supports rollback naturally, it also uses flash size we do not need, i.e., the header and allocation list. Table 3.1 shows the flash occupation details, which uses 900 bytes on 400KB slots. Furthermore, deleting a patch can cause file system fragments. Though a simple clean-up algorithm can solve this, we also need to pay some costs, the flash life cycles. However, as IoT devices will not update frequently, and, we do not have to clean the fragments, this cost can be far fewer than the expected reliable flash write cycles.

3.3 BSDiff-Inplace

3.3.1 Requirement

BSDiff [4] is initially designed for PC binaries, and it is not suitable for IoT devices for two reasons. First, the author aimed for global optimization of compression rate. Thus, we may refer to a region before the current position while patching, which means we cannot do BSDiff in place. However, as the reason stated in Section 1.2, the ROM and RAM size we can use are limited. We highly prefer an in-place solution. Besides, since BSDiff does not

Table 3.1: The rFAT file system header format

Name	**Type**	**Size (bytes)**	**Annotation**
magic	uint32_t	4	must be "rFat" (0x54414672)
block_cnt	uint64_t	8	the number of total blocks
firmware_block_cnt	uint64_t	8	the number of firmware blocks
file_entries	array	480	file entries
alloc_list	array	400	file block allocation list
Total	N/A	900	total ROM occupation

compress data, the author chooses BZip2 to compress the BSDiff output. Nevertheless, BZip2 uses at least 100KB [7] memory (depending on the block size option), which is not acceptable for IoT devices.

3.3.2 Design

Old Code

```
#include <stdio.h>

int main() {
    char name[] = "Alice";
    printf("hello, world!\n");
    printf("hello %s!\n", name);
    return 0;
}
```

New Code

```
#include <stdio.h>

int main() {
    int flag = 0;
    char name[] = "Alice";
    printf("hello, world!\n");
    printf("hello %s!\n", name);
    return 0;
}
```

Figure 3.6: Code difference comparison

The in-place version follows the main idea of the original version, that is, constructing difference strings to represent the difference between two files. For instance, assuming we already have a pair of approximately matched regions, Table 3.2 shows how to get the difference string. Because most changes in binary tend to be gentle address changes (4 or 8 bytes) or a breaking change (new function or code deletion), the result mostly consists of zeros; in other words, it is highly compressible. Figure 3.6 raises an example. The green line on the right side is the newly added code and will be considered an extra string in BSDiff.

Old char	Old ASCII value	New char	New ASCII value	Difference
H	72	H	72	0
e	101	a	97	-4
l	108	l	108	0
l	108	l	108	0
o	111	o	111	0
Result	0 -4 0 0 0			

Table 3.2: An example of difference string in BSDiff

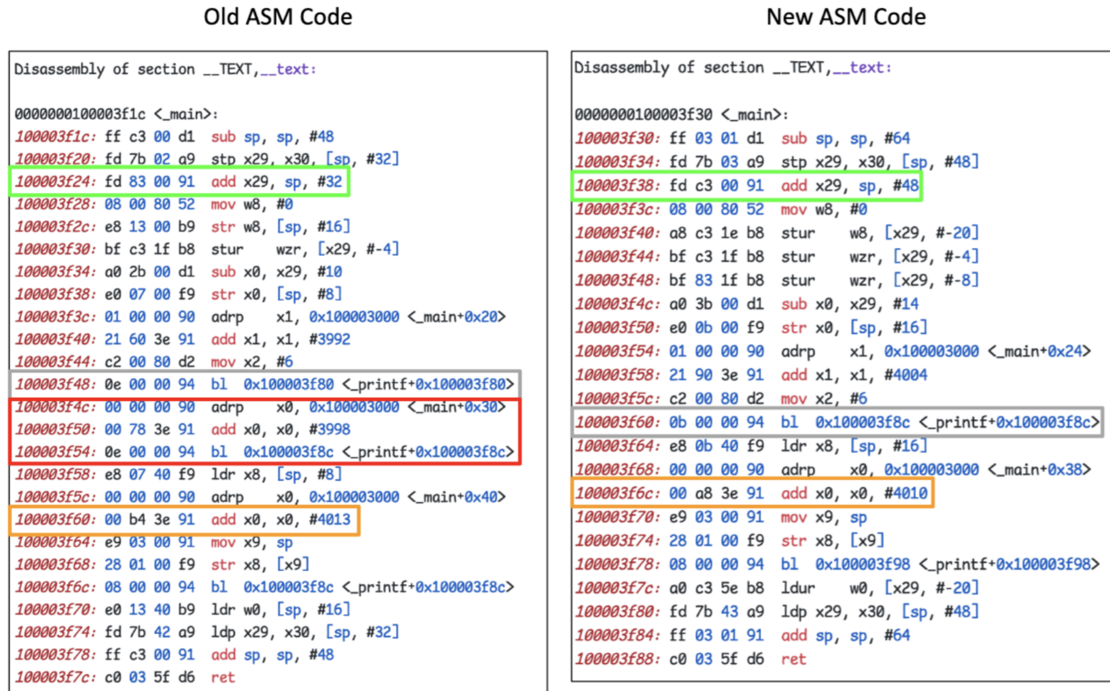

Figure 3.7: Code difference in ARM Assembly

Besides, the variable "name" will be taken place by the new variable "flag", e.g., if the address of "name" was 0x0000, the address now is 0x0004 if int is 32 bits. Therefore, any reference to "name" will be influenced as an 8-byte change, a gentle change marked as orange. Moreover, the red line means the removed code, which will be ignored while patching.

Though the main idea is the same, the steps are still different from the origin, shown in Figure 3.8. First, we do not need to handle the overlap for reasons will be given later to solve the forward reference issue, which is already illustrated in Section 2.4.1. Besides, BZip2 is used as the compression module of the original BSDiff as BSDiff does not compress data directly. However, BZip2 memory demands highly exceed the memory on our devices, nrf52DK.

Figure 3.7 shows the code difference in Figure 3.6 in the view of ARM Assembly. The two codes are obtained by compiling with ARM Clang and disassembling with *objdump* command. The "volatile" keyword decorates the variable declarations for demonstration because they would be optimized as two constants, and we cannot see this comparison. It will not influence our conclusion, as in a true project, most variables involve calculations and cannot be considered a constant. The green rectangle refers to the declaration of "int flag", and it takes an integer variable of more than 4 bytes because of the byte alignment in C. Besides, the orange square shows the address difference when referencing the "name"

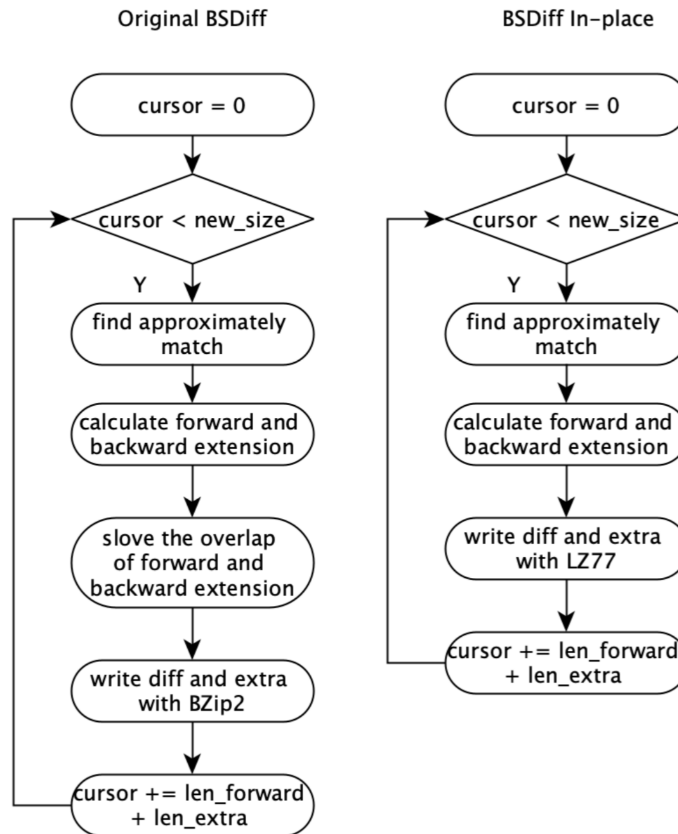

Figure 3.8: The original BSDiff vs. the in-place BSDiff

variable, and the offset difference is 4 bytes. Furthermore, the "hello world" printf call in the red on the left is removed in the new ASM code. Last, as an extra note, the grey region is a stub call generated by the compiler, but not a printf call.

To solve the above issues, we must go through the BSDiff first. As Colin's paper [4], we find out the an approximate match, $new[x'...x'+k'] = new[y'...y'+k']$, for an exact match, $new[x...x+k] = old[y...y+k]$. This approximate match should contains more than 8 bytes difference that $new[x'+i] \neq old[x'+i+(y-x)]$. Then, we extend this match region on both forward and backward sides to get the final approximate match, the orange part in Figure 3.9.

Next, the orange region details are in Figure 3.10. The part, length of which is *lenf*, will be saved as a difference string by $new[i] - old[i]$, the extra part will be stored as an extra string, and the length of the skip part will be saved to let the patch program know the number of bytes to be skipped. Finally, we need to move our cursor back *lenb* bytes, i.e., the *lenb* part will be a part of *lenf* of the next iteration until the cursor moves to the end.

Now, we roughly know how BSDiff works. The process of determining the *lenf* and *lenb* trys

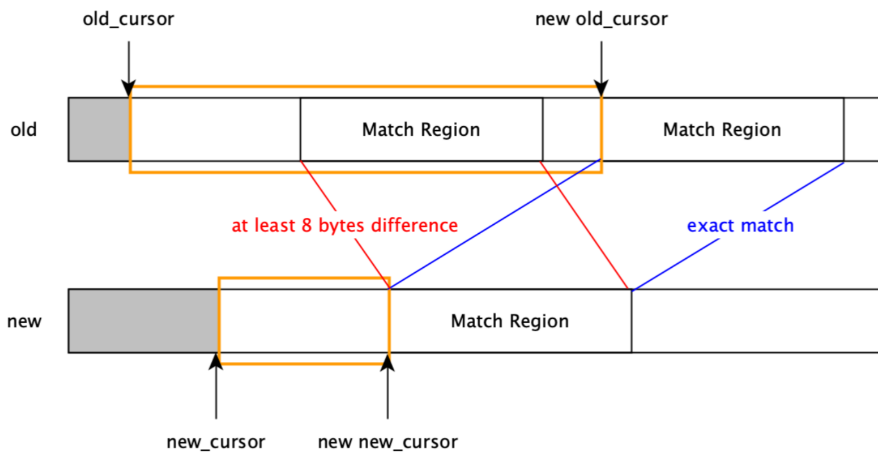

Figure 3.9: Approximate match in BSDiff

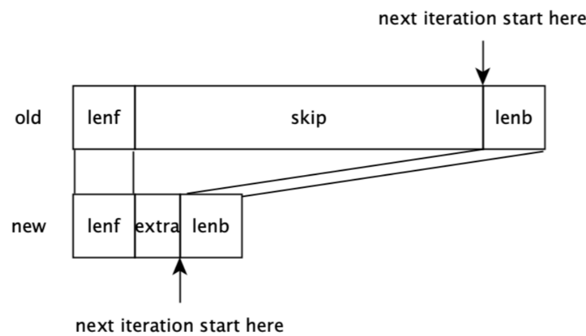

Figure 3.10: Approximate match region details

to find the optimal result needs to be adjusted for in-place BSDiff. This optimal result can be represented as

$$score = \frac{match(old, new, x)}{x}$$

, *match* counts the number of matched bytes in the way mentioned above. We are trying to find the highest *score* and the *lenf* equals the x when the *score* peaks. Also, the *lenb* is determined in the same way, but the match starts from the end of the approximate match. Whereas a case may happen like Figure 3.11 describing when the length of approximate match in the old is shorter than it in the new, and there is not enough extra string, $lenf + lenb$ can be greater than the total length of the approximate match in the old. In this case, forward reference happens. That is, we need the old data before recovering the following new data, but in in-place patches, it is challenging. Because the number of

skipped bytes is calculated by

$$len_skip = (old_cursor - lenb) - (new_old_cursor + lenf)$$

, two-pointer algorithm can solve this issue. Instead of always looking for the best result, we end the search when an overlap occurs, no matter in the old or new. Then, the cost is that some bytes belonging to a different string in the original version may be put in an extra string. Therefore, the compression rate may be slightly lower than the original implementation, and the data will be given in Chapter 4. This compression rate downgrade is necessary, supporting the BSDiff in-place running on tiny IoT boards.

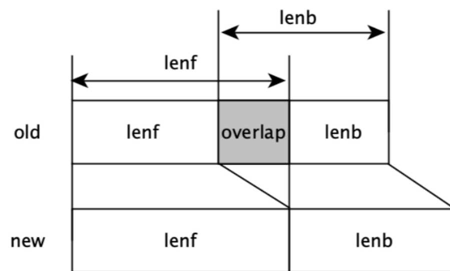

Figure 3.11: An overlap causes forward reference happens

On the other hand, the compressing algorithm issue comes. According to the BZip2 official manual [7], we need $100k + (4 \times blocksize)$ or $100k + (2.5 \times blocksize)$ to decompress, costing more memory and less time or less memory more time. The default block size is 900KB, and the minimal size is 100KB. Unfortunately, both options cost too much memory for our boards, and there is no low-memory low-level API from BZip2. Hence, we need a replacement of BZip2, the LZ77, briefly mentioned in Section 2.3.1.

LZ77 is designed as a lightweight, dictionary-based compression algorithm. It uses a sliding window, called history, to save the data already read. While new data comes, put in another buffer area, the LZ77 finds a reference in the history and uses the reference to replace the original data. This process is an ideal method for our scenario for two reasons:

(1) **Flexibility**. Because the principle is simple and flexible, we can choose our own buffer size and windows size to fit our devices.

(2) **Stability**. The memory cost of LZ77 is stable, i.e., once the buffer size and window size are decided, the memory cost is mathematically decided. Because there is no uncertainty, we can use only the memory on the stack, avoiding risks from memory allocation on the heap such as memory leak and memory overflow.

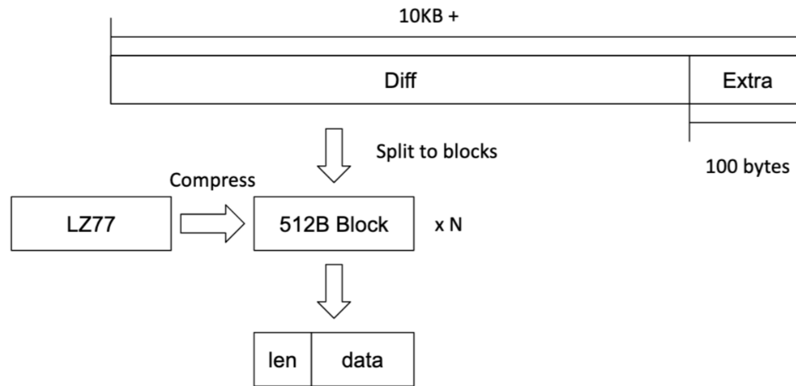

Figure 3.12: Block-wise compressing by LZ77

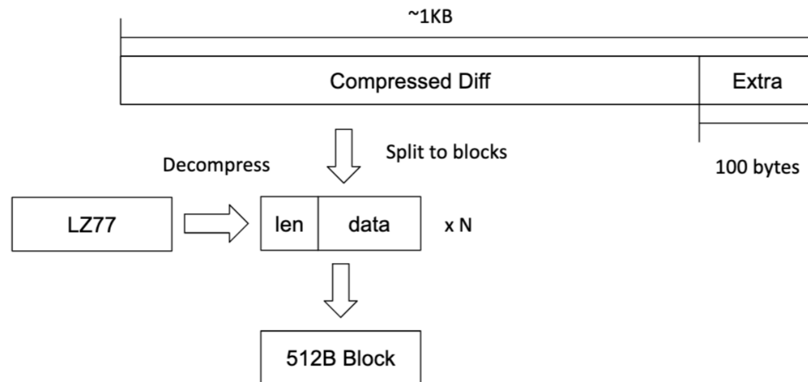

Figure 3.13: Block-wise decompressing by LZ77

Because of experiments discussed later, difference strings are maximally around 10KB. So, it is unlikely to load the whole difference string in memory. Instead, it will be split into blocks, and the LZ77 compresses each block and outputs. Figure 3.12 and 3.13 shows the details. The "len" hints to the decompressor how many bytes it needs to read, and the "data" part contains that number of bytes. Besides, blocks are independent. Therefore, we do not need to save the history from other blocks.

3.3.3 Design Critique

Because of the in-place implementation, as Figure 3.14 illustrates, we have to partially move firmware backward for times (about five times for a 20KB patch). This operation costs flash life cycles, which can be reduced by padding zeros between blocks while compiling and linking the firmware as Figure 3.15. However, because of the schedule limit of this project, this padding solution is over-complicated. More detailed, we need to design a new linker script, splitting an assembly section and padding with zeros. For example, there is a DATA section, the size of which is 4KB. We can split it into two 2KB sections and link them to

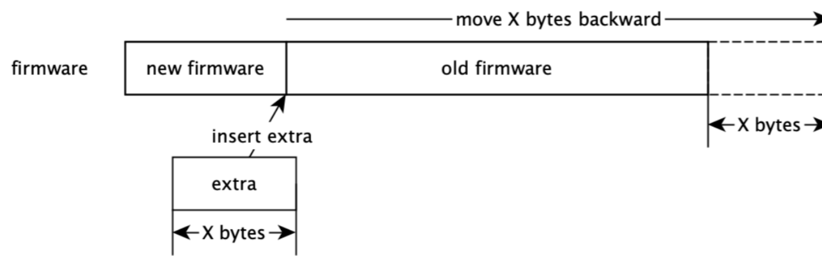

Figure 3.14: Move data backward to insert extra string in BSDiff

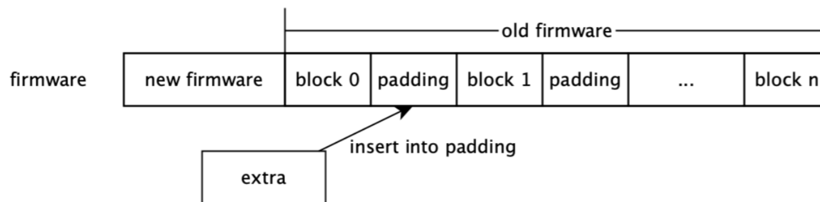

Figure 3.15: Insert extra string into pre-reserved paddings

two different positions, and the middle space is zeros.

Moreover, we only do this the first time and at the time when the middle space is out. Besides, a sequence of challenges comes. For instance, what is a reasonable ratio between the firmware and padding; how to distinguish which part of the code should be in the padding without knowing the previous code? Therefore, we will not investigate it in-depth this time, but it is an inspiring future work.

3.4 EEPROM Emulator

3.4.1 Requirement

Figure 3.16 shows a downside of flash memory. Flash memory has three I/O operations, read, write, and erase. The read operation follows the literal meaning, given an offset and a size and reading the "size" bytes from the offset to a buffer. However, the write operation differs from other common memory like RAM or EEPROM. The write operation can only set 1 to 0 in a flash. For example, if we write 0b10101010 to a region that was 0b11110101, the result is 0b10100000. To set 0 to 1, we need to use "erase".

Nevertheless, "erase" is a physically page-aligned operation, i.e., we must set a page (e.g. a 4KB aligned continuous memory) to 1. We cannot only set a part of a page to 1. So, we need to develop an EEPROM emulator for flash, and the design will be introduced in the

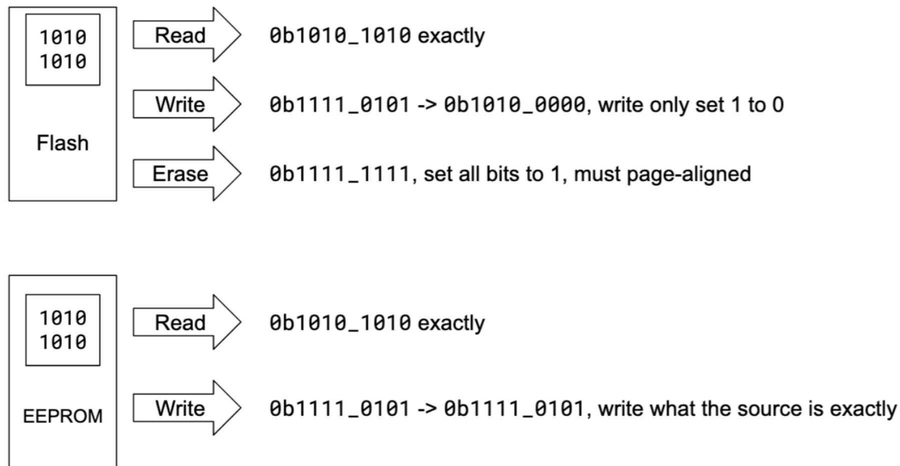

Figure 3.16: Flash vs. EEPROM

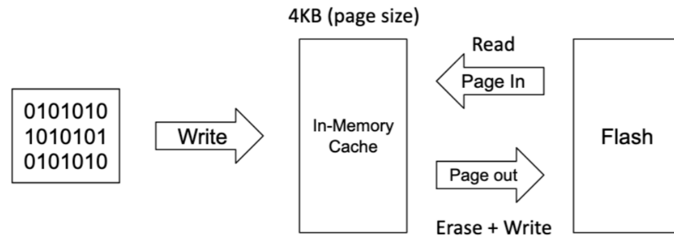

Figure 3.17: A design of EEPROM emulator for flash

next section.

The EEPROM emulator hides from the programmer the necessity to erase an entire page to modify just part of that page. It does this by maintaining a page in memory, and all read/write operations are converted to access this page in memory.

3.4.2 Design

An EEPROM emulator design is given in Figure 3.17. This design is like a simplified page-replacement algorithm, and it works in this workflow: once a batch of data is written to the flash. Instead of directly writing it into the flash, we write it into a buffer in RAM. Besides, the cache also holds a page ID. If the page ID of the data we want to write is different from the page ID of the current in-memory page buffer, we first erase and write the current buffer to the corresponding page in a flash, then read the page's content that the data want to write.

3.4.3 Design Critique

This design is simple and easy to implement. However, flash chips, namely Read Only Memory (ROM), are not designed for frequent writing, but firmware updates and, hence, flash erase-rewrite operations are expected to be infrequent. Moreover, the EEPROM emulator can be unreliable as it keeps changes in memory until they are committed to flash. Though all changes will be saved in a flash when we call a "close" function, it can still lose data, i.e., if an error happens and the system is halted. Therefore we have no chance to save the changes.

4 Implementation and Evaluation

Currently, embedded system development is challenging because of the leak of toolchains, i.e., visual debugging tools, variable watchers, and single-step debugging tools. It is worth splitting the implementation into two parts, workbench implementation and prototype implementation. The workbench implementation allows developers to develop, debug and test the software without the complexities of the embedded system. Then, programmers can port them to the embedded system.

Therefore, this chapter distinguishes contents into three parts: workbench implementation, prototype implementation, and performance evaluation. The workbench and prototype implementation show details of implementations close to the hardware and software side in Section 4.1 and 4.2 respectively. Besides, in Section 4.3, the performance evaluation first shows the metrics we care about, then explains how the experiments will be executed and list the test cases. Finally, the results and reflections will be given.

4.1 Prototype Implementation

4.1.1 rFAT backend

The rFAT file system is designed in frontend/backend mode. The backend side defines multiple interfaces, requiring developers to support random access for the flash chip, and they are listed below.

(1) **fs_area_open(id, fs_area)** accepts two parameters. One is the device ID, which allows users access to a unique area of the flash chip, and the type of ID is implementation-defined. In Zephyr OS, it is the partition ID, image-0, which is declared in the device tree file(*.dts* file), as Listing 4.1 shows. The other one is the pointer to a file system area as the output, and users need to use this pointer to call other functions below.

(2) **fs_area_close(fs_area)** closes a file system, representing the termination of occupation to a file system by the pointer.

(3) **fs_area_read(fs_area, offset, dst, len)** reads raw data from the opened file system. It starts from the *offset*, and reads the following *len* bytes to a buffer, *dst* , given by the caller.

(4) **fs_area_write(fs_area, offset, src , len)** writes raw data to the file system. The parameters are the same as the read interface, but only the *src* differs. It indicates where to read the data to write.

(5) **fs_area_get_size(fs_area, size)** can get the total size of the file system, and the return value is assigned to the *size*, which is a pointer as an output.

To implement these functions, we wrap the Zephyr Flash Map APIs [22] with our EEPROM emulator. The Zephyr Flash Map APIs are similar to the rFAT backend definitions but have an API called *flash_area_erase*, setting all bytes in a flash page to 1 for reasons in Section 3.17.

```
&flash0 {
    partitions {
        slot0_partition: partition@c000 {
            label = "image-0";
            reg = <0x0000C000 0xd000>;
        };
        /* other partitions ... */
    };
};
```

Listing 4.1: flash memory declaration

4.1.2 Bootloader

The bootloader steps are given in Figure 4.1—first, the hardware boot MCUBoot from the beginning of the flash. Then, we check the device status from the flash. If this device needs to update or roll back, it will check the existence of the corresponding patch file, i.e., "update.patch" and "rollback.patch". If it exists, we will open this patch and apply it. Otherwise, the board will print an error message and halt.

Moreover, users can use the frontend APIs of rFAT, which will be demonstrated later, to manage patches on the device, and set the device status to "need update," "need rollback," or "just boot." For example, we make an Internet Protocol Support Profile (IPSP) [23] based patch receiver (Figure 4.2 and 4.3). This protocol is a Bluetooth profile that utilizes 6LoWPAN [24], i.e., provides IPv6 connectivity over BLE. We first need to integrate the IPSP library in our application and flash it into a device with MCUBoot. Then, the device

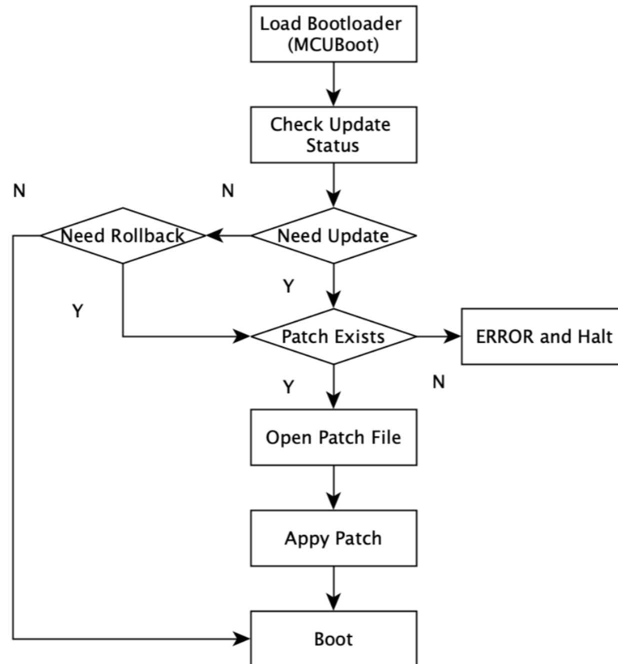

Figure 4.1: The steps to launch applications in MCUBoot

boots, broadcasts its Bluetooth signal, and waits for a connection. Next, we use our PC (or other devices) to connect to this board. Because we do not enable Dynamic Host Configuration Protocol (DHCP) [25] on board, which can help our PC auto-configure our IP table, we also need to manually configure the local IP table. A possible command for configuring is "ip address add 2001:db8::2/64 dev bt0". Last, we can access our device over IP-stack protocols, such as TCP [26], UDP [27], and HTTP [28], if the device implements them and listens to them.

On the TCP part, we can use any tools supporting TCP transmission to send a patch to the device. In our case, we use "nc 2001:db8::1 4242 < patch" to send the patch, where "nc" is the abbreviation of the command line tool, net cat [29], and the left angle means that this command takes the file, "patch," as the input.

4.2 Workbench Implementation

This section will introduces the rFAT frontend APIs, which are designed for applications to save patches and for MCUBoot to read and apply patches. These APIs are platform-independent, i.e., they only call the rFAT backend APIs. Therefore, an advantage is we can easily adapt them to other platforms. The APIs are defined in the following way,

(1) **rfat_fs_open(id, fs_area)** opens a rFAT file system, and the ID is implementation

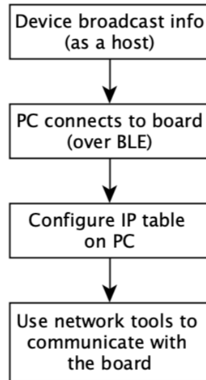

Figure 4.2: The steps to establish a IPSP node

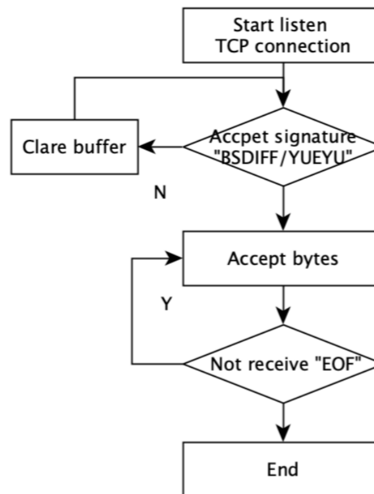

Figure 4.3: Transmit a patch to a device by TCP

defined. In this case, it is the partition ID, image-0, the same as above.

(2) **rfat_fs_close(fs_area)** closes a opened rFAT file system by the given fs_area pointer.

(3) **rfat_fs_init(fs_area, image_size)** initializes a file system, and it needs the size of current image to decide the firmware region. Once the file system is initialized and changed with rFAT APIs, the file system knows the change of firmware.

(4) **rfat_fs_validate(fs_area)** validates whether a opened fs_area pointer refers to a valid file system or not. The file system must start with a specific magic number.

(5) **rfat_fs_zip(fs_area)** can sort block fragments for reasons in Section 3.2.

(6) **rfat_open(fs_area, name, fd)** opens a file by a given name, and returns a file descriptor, *fd*. It is a stream-like I/O interface.

(7) **rfat_close(fs_area, fd)** closes a opened file and commit all changes to the flash storage.

(8) **rfat_create(fs_area, name, fd)** creates a new file and returns the *fd*. If a file already exists, this function returns error(-1).

(9) **rfat_delete(fs_area, name)** deletes a file in the file system. If this file does not exist, then returns error(-1).

(10) **rfat_read(fs_area, fd, dst, size)** is the stream-like read interface. Therefore, we do not need to set an offset to indicate where to read the data. The destination and the number of bytes to read are the parameters *dst* and *size*, respectively.

(11) **rfat_write(fs_area, fd, src, size)** is the stream-like write interface. We still do not need to pass an offset to this function. Besides, this function will read *size* bytes data from *src*.

(12) **rfat_read_firmware(fs_area, offset, dst, size)** is designed because the BSDiff-Inplace requires random access APIs to modify the firmware. It reads the *size* bytes from *offset* to *dst*.

(13) **rfat_write_firmware(fs_area, offset, src, size)** is designed because the BSDiff-Inplace requires random access APIs to modify the firmware. It writes the *size* bytes from *src* to *offset*.

(14) **rfat_ls(fs_area, entries)** can list all files in the file system and put it in the entries, which refers to file entries.

4.3 Performance Evaluation

After implementing the system functions, it is necessary to validate its reliability and evaluate the system performance. This section first introduces the metrics in section 4.3.1. Next, section 4.3.2 shows experiment designs, including requirements, steps, and limitations. Then, the experiment results are shown in section 4.3.3 and reflections are placed in section 4.3.4.

4.3.1 Metrics

Because this dissertation mainly focuses on the incremental update on tiny devices, the following metrics are critical:

(1) **Compression Rate**. Since one of the main reasons to use an incremental update is to save the limited bandwidth on small devices, compression rate can stand for the extent of the bytes we save while transmitting. Moreover, it is unrelated to a specific network

environment, i.e., we do not consider things out of scope, like multi-cast, signal interference, and the detailed bit rate. Besides, in this dissertation, the compression rate stands for how small the output file is, i.e., the lower the compression rate, the better it is.

(2) **Memory and Storage Occupation**. The targeting boards run on small flash and memory. Therefore, the peak memory and flash usage can be a reference to show how small a board this system can run on.

(3) **Flash Life Cycle**. This dissertation uses the in-place incremental update, which is unavoidable. Because many devices in a production environment need to work for years, reducing the bytes to write while doing the in-place incremental update is also vital.

4.3.2 Experiment Design

In this project, we test the above three metrics separately. For compression rate, we will compare the original BSDiff and our BSDiff-Inplace. Specifically, we will use the different versions of the examples from the official Zephyr OS repository to make incremental patches and observe the size of patches. Furthermore, as section 3.3 says, the bigger the block size, the more chances we have to find references while compressing. Therefore, we will also try different LZ77 block sizes to watch the relationship between the LZ77 block size and the compression rate and memory usage.

Regarding the memory and storage occupation, it is hard to watch the dynamic data in real-time. Fortunately, the official Zephyr OS tool-chain offers a tool to estimate the two values, and we will take them as our results.

Last, the flash life cycle is data we cannot get an exact value. However, we can get an approximate result by counting the written bytes and dividing them by the page size, which is 4KB in this case.

4.3.3 Result

We select six test samples from another version listed in Table 4.1. The version is given in the git commit hashtag format. The first column is the program name, the second is the size of the new firmware, the third column displays the patch generated by BSDiff and BZip2, and the last column gives the patch size generated by BSDiff-Inplace and LZ77 with 512 bytes block size. The output size of BSDiff is 25.03%, and the value of the in-place version is 52.39% which is two times than the original. Though, it is still smaller than the uncompressed monolithic updates.

To go more in-depth, we also want to explore the relationship between the LZ77 block size and the compression rate. The samples are the same as above, and the results are in Table

Program	Version	Uncompressed	BSDiff	BSDiff-Inplace
blinky	0b286d7304 → e18fcbba5a	15176 B	2551 B	5471 B
button	fff2644189 → be2e6a0850	18196 B	2726 B	5589 B
threads	5be0d00d41 → e18fcbba5a	29460 B	5535 B	11530 B
central	a1b77fd589 → 8e1682d1ea	113680 B	58638 B	110214 B
peripheral	6eb7574076 → 8e1682d1ea	169040 B	20806 B	58523 B
mqtt publisher	fcd392f6ce → aa5187ecde	106852 B	38187 B	82140 B
Average Compression Rate		100%	25.03%	52.39%

Table 4.1: The size patches produced by BSDiff and BSDiff-Inplace

Program	Version	Block Size			
		128 B	**256 B**	**512 B**	**1024 B**
blinky	0b286d7304 → e18fcbba5a	7207 B	6058 B	5471 B	5153 B
button	fff2644189 → be2e6a0850	7807 B	6337 B	5589 B	5229 B
threads	5be0d00d41 → e18fcbba5a	15062 B	12779 B	11530 B	10900 B
central	a1b77fd589 → 8e1682d1ea	118120 B	112438 B	110214 B	109626 B
peripheral	6eb7574076 → 8e1682d1ea	80006 B	66029 B	58523 B	54383 B
mqtt publisher	fcd392f6ce → aa5187ecde	91700 B	85141 B	82140 B	80958 B
Average Compression Rate		63.1%	55.96%	52.39%	50.68%

Table 4.2: The size patches produced by BSDiff-Inplace with different block sizes

4.2. In this table, with the increase in block size, LZ77 gets more chances to use a reference to replace literal data. However, the change in block size has a bottleneck. Figure 4.4 illustrates that the cost and gain do not have a linear relationship. For example, for the columns in 128 and 256 bytes, we pay only extra 128 bytes and gain about 7% better compression rates. Conversely, when we use 1024 bytes block size instead of 512 bytes block size, the compression rate is less than 2% better. In other words, the efficiency, increasing compression rate per byte, of changing 128 bytes block size to 256 bytes block size is 14 times the data of changing 512 bytes block size to 1024 bytes block size.

Name	Memory Region	Used Size	Region Size	Used (%)
MCUBoot	FLASH	39104 B	48 KB	79.56%
	SRAM	23616 B	64 KB	36.04%
MCUBoot-Incremental	FLASH	34808 B	48 KB	70.82%
	SRAM	34 KB	64 KB	53.12%

Table 4.3: The MCUBoot memory map

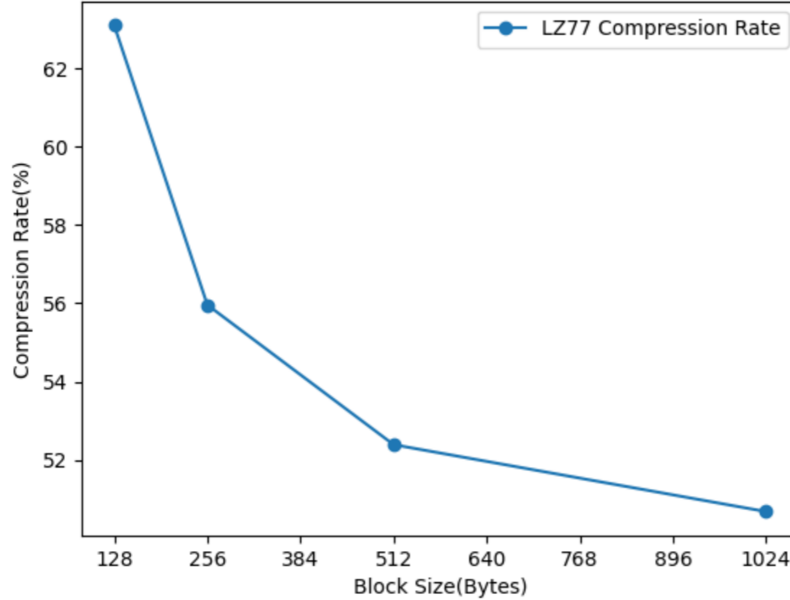

Figure 4.4: LZ77 compression rate with different block size

Next metric, the memory occupation, is in Table 4.3. The first row shows the memory map of the official release, using about 38KB flash and at most 23KB RAM. Besides, our incremental update supported MCUBoot costs around 34KB flash and at most 34KB RAM. The flash occupation of our version is 4KB less than the origin because our version removes some cross-slots validation as we do not need it in a single slot device. However, the monolithic MCUBoot needs this to decide which partition to update, rollback, or boot. Besides, we spend 11KB more than the origin, which performs as we expect. 8KB out of 11KB is used for the EEPROM emulator, i.e., 4KB is for cached data, and the other 4KB is for data swapping. Moreover, LZ77, with a 512KB block size, takes 1KB for maintaining a decompressing queue and incoming data byte queue. Besides, the left 3KB is taken by other functions on the stack.

Program	Version	Uncompressed	Move	Life cost
blinky	0b286d7304 → e18fcbba5a	15176 B	193466 B	12
button	fff2644189 → be2e6a0850	18196 B	227621 B	12
threads	5be0d00d41 → e18fcbba5a	29460 B	1045760 B	35
central	a1b77fd589 → 8e1682d1ea	113680 B	46576547 B	409
peripheral	6eb7574076 → 8e1682d1ea	169040 B	26265095 B	155
mqtt publisher	fcd392f6ce → aa5187ecde	106852 B	33302950 B	311

Table 4.4: The life cycle cost on different firmware

The flash life cycle can be estimated by counting the byes to write in Table 4.4. The "Move" column shows the number of bytes that have been moved back, which can be observed that the more the firmware size is, the more bytes need to be moved. The writing times on the same page are 12, 12, 35, 409, 155, and 311, respectively. The life cycle is 10,000 to 50,000, depending on the flash quality.

4.3.4 Reflection

The two main downsides of our implementation are the compression algorithm and the moving back operation in the bsdiff-inplace. For the compression algorithm, LZ77 does not give a remarkable compression rate. Though it is the cost to use less memory, it can still be better. For example, virtual memory can help, as it allows us to use a more expensive compression algorithm to earn a better compression rate. Furthermore, if we update a device every month, a device may only be used for 2 to 3 years. However, it is unlikely to update such frequently. A possible solution already said in Section 3.3 is reserving zero paddings to prevent too many byte movements.

5 Conclusion

The BSDiff-Inplace, a difference algorithm that offers two interfaces to generate and apply a patch and balance the RAM and flash cost. People can use a small patch and the old firmware on the device to generate the new firmware, avoiding transmitting a massive new firmware entirely. To adapt this algorithm to Zephyr OS, we designed a reversed FAT (rFAT) file system. Unlike the typical FAT file system, we put the FAT header and block allocation list at the end of the flash because the bootloader, MCUBoot, loads the application from the beginning of a flash partition.

Furthermore, we made an EEPROM emulator due to the physical feature of flash chips, which only natively support page-wise read/write operations. Thus, we can use a flash as an EEPROM, i.e., it supports random access. This implementation is like a page replacement algorithm on modern PCs.

Overall, the results achieved the scope in the Introduction,

(1) **Incremental update** is supported by BSDiff-Inplace, which generates patches with about a 50% compression rate, where the firmware is usually around 250 KB, so it can reduce 125 KB data transmission while updating. Moreover, each update averagely costs 200 flash life cycles. If a device is updated seasonally, a device can be used for over ten years, and the flash life cycle is still in a safe range.

(2) **Rollback** is natively supported by the rFAT because we can treat a downgrade patch as a regular file in this file system. Besides, this patch just "updates" the new version to the old by generating from the new to the old comparison.

(3) **Flash optimization** is accomplished by the EEPROM emulator. With this component, we do not have to refresh an entire page when we change one byte on this page.

5.1 Future Work

However, two issues influence the results. First, the compression rate of the original BSDiff is around 75%, i.e., our version is 25% worse than it. Though it is caused by the trade between memory and time cost, we still have many zones to improve. For example, we can

explore a hybrid patching system with modular and incremental updates. This approach can potentially decrease the patch size remarkably, as it solves two problems. One is that the current LZ77 compression is a local compression algorithm because we do not have enough memory to load the full firmware in memory. Once we update firmware modularly and update modules incrementally, it is possible to load a module totally in memory. Therefore, we can use a global compression algorithm for more compression chances.

On the other hand, the next issue, the backward movement in the BSDiff-Inplace, takes time, and the hybrid algorithm can also solve a flash life cycle. Because we can preserve zero padding between different modules, we can add new data into the padding rather than moving blocks back and spending many flash life cycles. Most importantly, it is also possible to patch a module in memory and overwrite the result to the flash, i.e., the backward movement issue can be ignored.

Summarily, this project is not perfect, and we would like to explore the more in-depth side of firmware updates.

Bibliography

[1] Ondrej Kachman and Marcel 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, 2017. doi: 10.1109/DDECS.2017.7934581.

[2] Konstantinos Arakadakis, Pavlos Charalampidis, Antonis Makrogiannakis, and Alexandros Fragkiadakis. Firmware over-the-air programming techniques for iot networks - a survey. *ACM Comput. Surv.*, 54(9), oct 2021. ISSN 0360-0300. doi: 10.1145/3472292. URL https://doi-org.elib.tcd.ie/10.1145/3472292.

[3] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9): 11734–11753, 2012. ISSN 1424-8220. doi: 10.3390/s120911734. URL https://www.mdpi.com/1424-8220/12/9/11734.

[4] Colin Percival. Naive differences of executable code. 08 2003. URL http://www.daemonology.net/bsdiff.

[5] David Korn, J MacDonald, J Mogul, and K Vo. The vcdiff generic differencing and compression data format. Technical report, 2002.

[6] Brenda S Baker, Udi Manber, and Robert Muth. Compressing differences of executable code. In *ACMSIGPLAN Workshop on Compiler Support for System Software (WCSS)*, pages 1–10. Citeseer, 1999.

[7] Julian Seward. bzip2 and libbzip2. 1996. URL http://www.bzip.org.

[8] Peter Deutsch. Deflate compressed data format specification version 1.3. RFC 1951, 1996. URL https://www.rfc-editor.org/info/rfc1951.

[9] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi: 10.1109/TIT.1977.1055714.

[10] FreeRTOS contributers. Market leading rtos (real time operating system) for embedded systems with internet of things extensions, Aug 2022. URL https://www.freertos.org/.

[11] RT-Thread contributors. Rt-thread iot os, Aug 2022. URL https://www.rt-thread.io/.

[12] Wikipedia contributors. Position-independent code — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Position-independent_code&oldid=1063379139, 2022. [Online; accessed 14-July-2022].

[13] Andrew Tridgell et al. Efficient algorithms for sorting and synchronization. 1999.

[14] David Korn, Joshua P. MacDonald, Jeffrey Mogul, and Kiem-Phong Vo. The VCDIFF Generic Differencing and Compression Data Format. RFC 3284, July 2002. URL https://www.rfc-editor.org/info/rfc3284.

[15] Giovanni Motta, James Gustafson, and Samson Chen. Differential compression of executable code. In *2007 Data Compression Conference (DCC'07)*, pages 103–112, 2007. doi: 10.1109/DCC.2007.32.

[16] Software Updates: Courgette — chromium.org. https://www.chromium.org/developers/design-documents/software-updates-courgette/. [Accessed 13-Jul-2022].

[17] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, oct 1982. ISSN 0004-5411. doi: 10.1145/322344.322346. URL https://doi-org.elib.tcd.ie/10.1145/322344.322346.

[18] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978. doi: 10.1109/TIT.1978.1055934.

[19] Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(6): 8–19, 1984. doi: 10.1109/MC.1984.1659158. URL https://doi.org/10.1109/MC.1984.1659158.

[20] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. doi: 10.1109/JRPROC.1952.273898.

[21] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x.

[22] Zephyr Project members and individual contributors, Feb 2022. URL https://docs.zephyrproject.org/3.0.0/reference/storage/flash_map/flash_map.html.

[23] Johanna Nieminen, Teemu Savolainen, Markus Isomaki, Basavaraj Patil, Zach Shelby, and Carles Gomez. Ipv6 over bluetooth (r) low energy. Technical report, 2015.

[24] Geoff Mulligan. The 6lowpan architecture. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 78–82, 2007.

[25] Ralph Droms. Dynamic host configuration protocol. RFC 1541, 1993. URL https://www.rfc-editor.org/info/rfc1541.

[26] Postel John. Transmission Control Protocol. RFC 793, September 1981. URL https://www.rfc-editor.org/info/rfc793.

[27] Postel John. User Datagram Protocol. RFC 768, August 1980. URL https://www.rfc-editor.org/info/rfc768.

[28] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. RFC 2616, 1999. URL https://www.rfc-editor.org/info/rfc2616.

[29] Hobbit. New tool available: Netcat, Oct 1995. URL https://seclists.org/bugtraq/1995/Oct/28.