CHALMERS
UNIVERSITY OF TECHNOLOGY

Delta Updates for Embedded Systems

An Implementation of Firmware Patching for Zephyr RTOS

Bachelor's thesis in Computer Science and Engineering

LINNÉA LINDH

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY / UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

DEGREE PROJECT REPORT

Delta Updates for Embedded Systems

An Implementation of Firmware Patching for Zephyr RTOS

LINNÉA LINDH

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of gothenburg
Gothenburg, Sweden 2021

Delta Updates for Embedded Systems
An Implementation of Firmware Patching for Zephyr RTOS
LINNÉA LINDH

Supervisors: Peter Malmberg, Endian Technologies AB; Jan Jonsson,
Department of Computer Science and Engineering
Examiner: Jonas Duregård, Department of Computer Science and Engineering

Department of Computer Science and Engineering
Chalmers University of Technology / Univerity of Gothenburg
SE-412 96 Gothenburg, Sweden
Telephone +46 31 772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden 2021

Abstract

Many resource-constrained Internet of Things units are used in a manner which makes them inaccessible via cable and at the same time unable to receive large amounts of data using radio transmissions. Upgrading firmware is hence difficult or in some cases impossible. Integrating support for delta updates in the unit is a potential solution to this problem, as it significantly reduces the payload during an upgrade scenario. However, as of now there is no open-source solution for delta upgrades in embedded systems. This thesis aims to show that such a solution is achievable and that it causes a significant payload reduction. This proof of concept is realized through the development of a Zephyr application with delta upgrade support. Using this application firmware upgrades are performed using only 2.6 percent of the amount of external data needed for standard upgrades.

Keywords: Firmware Upgrades, Zephyr, Resource-constrained Systems, Patching, Delta Encoding, Delta Updates, Data Differencing, Open-source.

Acknowledgements

This thesis would have been inconceivable without the sterling help I received from the developers at Endian Technologies. Not only in the form of stimulating discussions and competent feedback, but through presenting the ideas behind this work in the first place. It has been tremendous fun, and I have learnt a great deal, but most importantly, I feel like I got the opportunity to create something useful.

Thank you.

Linnéa Lindh, Gothenburg, February 2021

Contents

Glossary

Term	Definition
Binary	Data represented using the binary number system. In this project often referring to executable code.
Binary Differencing	Differencing using two binaries. This is in contrast to source code differencing, or other forms of differencing comparing text files, which is commonly used in version control software.
Booting	The process that starts the device operating system.
Block	In C, a block refers to an area in the source code delimited by curly braces.
Callback Function	A function which is passed as an argument to another function, which the other function calls/executes at a given time.
Delta	Data describing the differences between the contents of two pieces of data - the source and the target.
Data Differencing	The process of generating a delta. Also known as *differencing*.
Delta Updates	Updates which create new firmware locally by patching old firmware.
Device Tree	A data structure describing the hardware components of a system. Allows for kernel usage and management.
Distribution server	The server used for distributing the data needed for firmware upgrades. For delta updates this data is the delta.
Firmware	Computer software used for low-level control of device hardware.
Git	A version management system for repositories. Examples of online repositories using Git include *GitLab* and *GitHub*.
Image Signing	A tool used for confirming that firmware has not been corrupted or altered since it was signed. Uses cryptographic hash to validate identity and integrity.
Internet of Things	Items with embedded electronics and an internet connection. Often abbreviated IoT.
Makefile	A file mapping words to short scripts. Used for build automation.
Metadata	Data containing information about other data.
Partition	A division of the secondary storage of a unit. In this project, a labeled sector between two memory offsets in the flash.
Patch File	A file containing a delta.
Patching	The process of creating the target using the source and the delta.
Repository	A storage location for data. Primarily source code.
Resource-constrained System	An embedded system with limited hardware resources available for completing jobs. For example, limited memory.

Term	Definition
RTOS	Real-time Operating System.
Source	The old data that one wants to upgrade from. In this project, the old firmware.
Space Complexity	How much memory an algorithm requires.
Struct	In C, a composite data type that stores a list of variables.
Target	The new data that one wants to upgrade to. In this project, the new firmware.
Target Device/Unit	The device running the firmware we want to upgrade.
Time Complexity	The amount of time it takes to run an algorithm.

1
Introduction

Many resource-constrained Internet of Things (IoT) units are used in a manner which makes them inconvenient to access via physical cable. In these cases, firmware upgrades are preferably transferred using Firmware Over-the-air, FOTA. FOTA technology makes it possible to deliver firmware using radio technology[1]. However, using radio technology comes with the drawback of it being rather expensive. Units receiving large amounts of data via radio may accumulate high data costs and fast battery drain. The latter of which is a consequence of the — relative to other IoT components — high energy needs of communication devices[1]. Hence, many IoT units seldom or never perform firmware upgrades.

A potential way to get around this problem is using so-called delta updates. These reduce the data redundancy of distributing the upgraded system in its entirety, by replacing it with a *patch file* containing a *delta*. Deltas contain information about the differences between two pieces of data, the *source* and the *target*. This information can together with the source data be used to create an exact copy of the target data. Delta updates function by using the old firmware image as source data, and the desired firmware as target data. Managing deltas does, however, introduce new complexity to the firmware upgrade process.

On the distribution side, the server has to keep track of which version of the firmware every target unit is currently running and be able to create deltas using this information. On the target units side, the devices system has to be able to build bootable firmware images using the data contained in these deltas. There are several open-source solutions for version control and creating deltas, or *differencing* as this process is called, but none freely available for applying deltas, or *patching*, in resource-constrained systems.

Developers at Endian Technologies AB saw a need to integrate delta updates into some of their resource-constrained solutions, which gave rise to the objectives of this bachelor's thesis. Endian is a small engineering firm located in Mölndal. The firm specializes in industrial IoT and manages everything from designing hardware, to developing firmware, to creating cloud services. Endian has developed online solutions for embedded systems for almost 15 years, with a broad spectrum of applications, such as the transport sector, medical technology, and consumer electronics. The company specializes in open-source solutions for embedded systems, which heavily influence the tools and systems used in this theses as well.

1.1 Problem Statement

With the prospect of allowing developers to create products that previously have not been viable, we hope to contribute to the development of a free implementation of delta-updates for embedded systems. We aim to do this by *(i) demonstrating that it is possible to perform delta updates in a resource-constrained system and (ii) showing that this causes*

a meaningful reduction of data transmitted during a firmware upgrade.
More concretely, this will be achieved through the completion of the following objectives:

1. Choose a suitable algorithm for patch creation and application.
2. Tweak the patch application algorithm in order to make it compatible with the hardware and the operating system (Zephyr).
3. Create a demonstrator which can be used to run the solution and test its validity.
4. Document the reduction of amounts of data transferred.
5. Suggest improvements to the solution and topics of further research.

Note that the aim of the project is to realize an idea in order to demonstrate its feasibility. It is not to create a finished product or a generalized solution. Thus allowing for the following demarcations to be made:

- the implementation will be a Zephyr application,
- the board will be the nRF52840 Development Kit (DK),
- data will be transferred via the boards USB interface,
- image encryption will be turned off.

Though creating a generalized solution would be out of the scope of this work, the nature of embedded systems makes the problems present in one system likely to be common in another. Ergo, the results of the report will in all likelihood be useful for implementations of delta updates in many embedded systems, despite the demarcations made.

1.2 Problem Domain

In order to theorize about potential solutions, some prerequisite knowledge about the target system is required. In this case the target system is a combination of the operating system and the card on which it functions; Zephyr real-time operating system (RTOS) and the nRF52840 DK.

1.2.1 Software

The Zephyr project is an open-source collaboration project hosted by Linux Foundation[2]. Zephyr RTOS is optimized for resource-constrained devices and is available across multiple architectures, which makes it a good fit for this project.
Updates in Zephyr are done via the Device Firmware Upgrade subsystem[3]. It provides the necessary frameworks to upgrade an application at run time. The subsystem consists of two modules: the boot loader and image management code.
A boot loader is a piece of software used for starting the operating system, or main application, of a unit[4]. It also enables device firmware upgrades, which makes it a focal point of this thesis. Zephyr uses a boot loader called *MCUboot*.
For MCUboot to work a set of fixed flash partitions must be defined, so that the boot loader knows where in the memory it ought to look to find certain data. The device tree labels for these partitions are: `boot_partition`, the partition containing the the MCUboot image; `slot0_partition`, the primary partition, containing the Zephyr image; `slot1_partition`, the secondary partition, which stores firmware upgrade images and therefore must be the same size as `slot0_partition`; and lastly, the scratch partition, which is used as temporary storage while swapping the contents of `slot0_partition` and `slot1_partition`[5].
The standard upgrade scenario is started by MCUboot[6] reading from the secondary partition and noticing that it contains an image marked as "pending". This triggers the process of physically swapping the contents of the two partitions with each other, and then

trying to boot from the upgrade image located on the secondary partition. If the upgrade does not boot, the boot loader swaps the images again and runs the old firmware, and if it boots, the device naturally runs the new firmware.
By not discarding the old firmware, MCUBoot is able to test upgrades before committing to running them. This is a process which makes upgrading firmware a lot safer. Moreover, it also leads the device having the old firmware stored until something new is written over it. Which could hypothetically be utilized for patching.
Before booting the new firmware, the MCUBoot performs an integrity check by reading some data placed at the end of the image - the image trailer. The trailer contains a cryptographic hash signature used for authentication and metadata about the image used for ensuring compatibility. MCUboot supports several options for signing an image, but in this work we will use a set of RSA keys.

1.2.2 Hardware

The system will run on the Nordic Semiconductor development kit nRF52840[7]. Key features are:

- an on-board SEGGER J-Link debugger
- support for Bluetooth, NFC, Thread and Zigbee
- user programmable LEDs and buttons
- 1 MB flash memory divided into 256 pages of 4kB each
- NVMC - Non-volatile memory controller
- 256 kB RAM on chip
- 64 MB external flash memory
- USB interface
- 32-bit, 64MHz Arm® Cortex™-M4F

More details on the significance of these properties will be presented in the theory chapter (chapter 2). A more detailed view of the device's memory layout can be found in figure 1.1. Zephyr has defined standard partitions for the nRF52840's flash memory, the addresses and labels of which can be found in table 1.1.

boot_partition	0x00000000 - 0x0000C000
slot0_partition	0x0000C000 - 0x00073000
slot1_partition	0x00073000 - 0x000DA000
scratch_partition	0x000DA000 - 0x000F8000
storage_partition	0x000F8000 - 0x000FFFFF

Table 1.1: Partition addresses[8].

1.3 Thesis Outline

The remainder of this report will be organised as follows. Chapter 2 introduces the theory, which focuses on presenting the theoretical foundations needed to understand the subsequent chapters. This includes the theory surrounding the subject of patches, and the theory surrounding embedded systems, and the the potential compatibility issues that may arise from combining these. Chapter 3 documents the methods used to create and test the implementation of the patching algorithm. Chapter 4 focuses on documenting the implementation created and the results of the tests. Chapter 5 evaluates the results and reassess the report as a whole in reference to the aims set up in section 1.1.

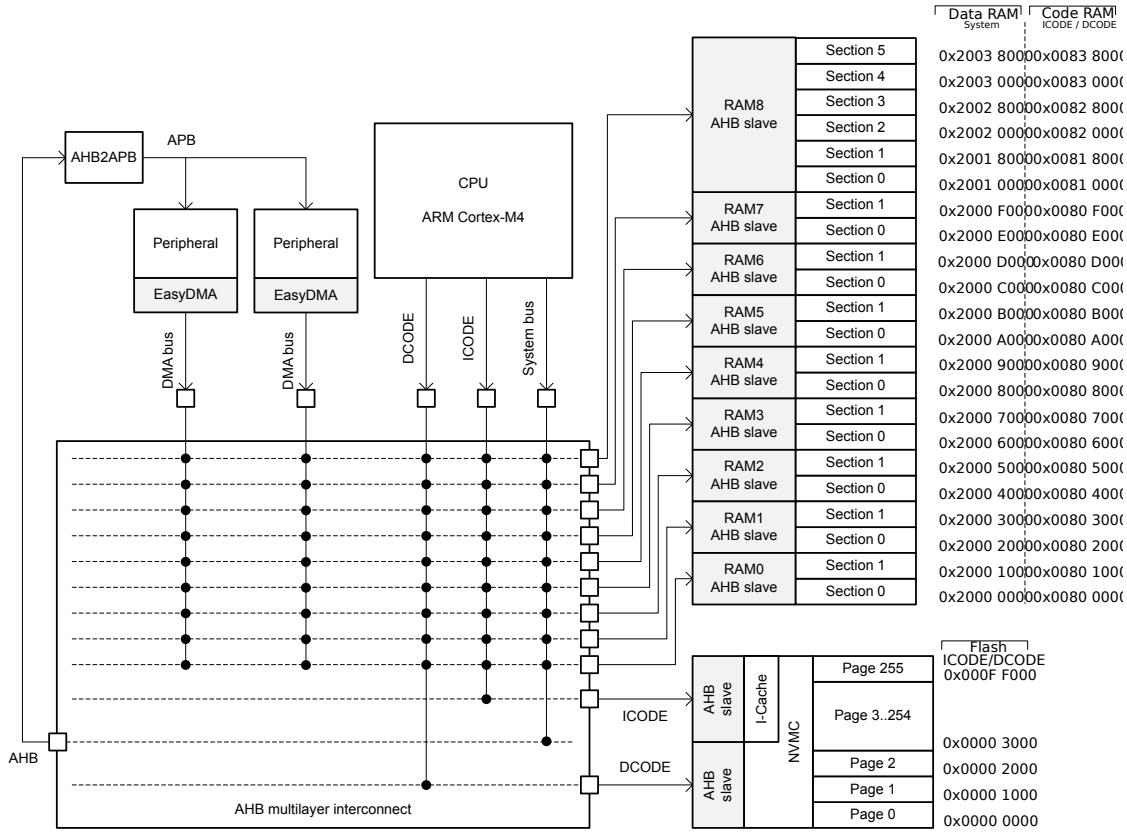

Figure 1.1: nRF52840 memory layout[7].

2
Theory

As mentioned in the introduction, the subject of this report is a combination of two topics: delta updates and embedded systems. Naturally, this chapter will focus on these, and what potential compatibility issues may arise from combining them. First, this aim is directed at the subject known as *differential compression*: what it is, how it is done, and which algorithms are used for this purpose. Second, the quirks of the device, or in particular its flash memory, will be detailed. The descriptions will not be covering the full topics, but will focus on the information needed to understand the design choices made in the following sections. The chapter will end on a concrete description of the compatibility issues and their potential solutions.

2.1 Differential Compression

Differential Compression is the process of creating and applying deltas. As the name implies, differential compression has a close relationship with *data compression*. In fact, data compression can be seen as a special case of differential compression in which the source data is empty[14]. As indeed, differencing with empty source data would be nothing more than condensing the target image into a more compact form, which is by definition data compression.
The data record containing the delta is in the context of software upgrades often referred to as a *patch*. The ideal patch is minimal in space complexity and has high usability, where the specifics of what usability entails varies depending on the domain of the problem. Usability is however, often to some degree tied to a time complexity. This leads to a space-time complexity trade-off, as lower time complexity leads to higher space complexity and vice versa. This implies that finding an algorithm for differential compression includes finding a global minimum, where the patches created are sufficiently small but time complexity is not consuming important computational resources from either the server creating the patch or the device applying it.

2.1.1 Differencing

Data compression is a process used to encode information using fewer bits than the original representation. It comes in the forms of lossy and lossless compression. Lossless data compression means that the compressed data will contain the same information as the uncompressed version, which enables the compression to be reversed. For delta updates, this is obviously needed, as the point of the process is to recreate the target. Differencing algorithms are ergo implicitly lossless data compression algorithms.
The theoretical foundations of lossless compression are found in algorithmic information theory, which concerns itself with measuring the irreducible information content of a string. In relation to binary differencing the string is the difference between the contents of two firmware binaries, or what could be called "the uncompressed patch".

Most differencing algorithms use a combination of data compression and differential compression, where the patches are post-compressed using regular data compression.
Delta encoding algorithms fall into two classes depending on what type of differencing algorithm is used[16]. The first of these is the *copy/insert* class, which typically use string matching to locate matching blocks of code. The second is the *insert/delete* class, which makes use of the longest common sub-sequence or shortest edit distance algorithms to create a sort of ”edit script” that modifies the source file into the target.
In general, insert/delete algorithms are inferior to copy/insert algorithms in terms of size complexity, but a lot easier for humans to read. Because of this, the former is used for version control in Git for example[18], and the latter is used for binary differencing. Consequently, we will only be concerned with algorithms in the copy/insert class in this thesis.
The term "copy/insert" comes from the instruction stored in the patches the algorithm class creates. "Copy" refers to instructions regarding which source blocks ought to be kept and at which offset they should start. "Insert" refers to instructions regarding where the new byte stored in the patch should be inserted.

2.1.1.1 BSDiff

BSDiff is a tool for building patches based on binary files, created by Colin Percival[10]. A major advantage of the BSDiff algorithm is that it offers a solution to the pointer problem[10], which is a commonly occurring problem when creating binary patches. The pointer problem arises from the nature of the executable file. Small, one-line, source code adjustments to the code or data changes the relative positions of blocks, which makes all pointers jumping over the modified regions outdated. A problem which, if not dealt with, creates unnecessarily large patches.
BSDiff exploits two facts to solve the pointer problem. Firstly, that the changes outside the modified region generally will be rather sparse, and mostly concern the least significant one or two byte. Secondly, that data and code usually moves around in blocks, which leads to a large number of nearby pointers having to be adjusted by the same amount. This information can be used to deduce that the byte-wise differences between old binary and the new are highly compressible.
The algorithm creates binary patches as follows. The algorithm starts by applying suffix sorting to the source file, specifically qsufsort by Larsson and Sadakane[15]. Suffix sorting is an algorithm which creates a suffix array, a lexicographically sorted array containing all suffixes of a string. A simple example of what this entails can be constructed using the string $\mathbf{S} =$ ”*delta*”. The string is indexed:

i	0	1	2	3	4
S[i]	d	e	l	t	a

(2.1)

and would given ascending order yield the following suffixes:

i	4	0	1	2	3
Suffix	a	delta	elta	lta	ta

(2.2)

where **i** is the starting index of the suffix. The suffix array, **A**, is an array containing **i**:

i	0	1	2	3	4
A[i]	4	0	1	2	3

(2.3)

This array can be used to find the largest possible regions in the new file which either approximately or exactly match with regions in the old file. The regions that are only

off by a smaller amount are saved in a separate file containing the byte-wise difference between the two files, which should be very repetitive and easily compressed. Copy and insert instructions are saved in another file, and should also have high redundancy. Lastly the non-matching data, which approximately should correspond to the modified code, is saved in an 'extra' file.
These output files can be considerably compressed using the *Burrous-Wheeler transform*, also known as block-sorting compression or BWT[15]. BWT uses permutation of the order of the characters in a string to concentrate character recurrences, which can be abbreviated (for instance "EEEEE" could be written as "5E"). Together these steps make BSDiff remarkably good at creating small patches.
Differencing using the BSDiff algorithm has a time complexity of $O((n+m)\log n)$ and a space complexity of $\max(17n, 9n+m)+O(1)$, where n is the size of the old file and m is the size of the new file[10]. Patching using BSDiff has a time complexity of $O(n+m)$ and a space complexity of $n+m+O(1)$.

2.1.2 Patching

In the same manner that the creation of a patch is related to data compression, the application of a patch is related to decompression. Patching consists of data compression and applying the instructions copy and insert. Exactly how this is done depends on the target system and general preferences. In this work, we will only look at sequential application and in-place application.
Sequential patching uses two memory regions, one containing the source and one containing the target[12]. It also uses patches that are divided into repeating patterns, or sequences. Each sequence consists of the parts: *diff*, *extra*, and *adjustment*, which together make up the instructions for a small section of the target image. Sequential patching is hence a loop which repeats the same three steps until the patch is fully applied.
In-place patching is slightly more memory constrained, as it only uses one memory region for both source and target. It uses a sort of rotation, where the source image is moved upwards in the memory to make room for the target image. Similarly to sequential patching, this algorithm also applies patches incrementally. However, it does also require that the source image can be moved during run-time.

2.2 System Limitations

A resource constrained system has several notable features that sets it apart from the conventional system most differential compression algorithms have been developed for. This implies that a large part of what signifies a good solution has to do with its portability. So, in order to choose a good algorithm, it ought to be determined which aspects of the system are likely to be incompatible with regular patching algorithms.

2.2.1 Limited Memory

As mentioned in 1.2.1 the unit's internal memory is divided into 4 partitions, where the primary partition contains the Zephyr application with the ability to decipher a downloaded patch. When performing a delta update a reasonable approach would be to build the new firmware in the secondary partition and then proceed with the normal update maneuver. However, this leaves no room for the patch. In a worst case scenario both the current and the new Zephyr application image take up the full primary or secondary partition, in which case the patch itself would either have to be stored somewhere else or

progressively discarded in order to make room for the new image being built in the second partition.

2.2.2 Flash Memory

One of the most notable properties of the system is that it uses flash memory. Indeed, memory management is quite the quintessential part of the delta encoding process. Flash memory has several limitations to low-level access that memory types assumed in patching algorithms do not have.
Erasing and writing to the memory is done using the *Non-Volatile Memory Controller* (NVMC)[7]. The NVMC has several features of interest. To start with, the operations of erasing or the writing to the flash will halt the CPU if it is currently running code located on it. Second, the NVMC is only able to set bits to '0' when writing. To achieve bits set to '1' a section of the memory has to be erased, creating a 'clean slate'. Additionally, there are a few features which vary between systems, and are characteristics of the target device.
Writing to the flash can be done down to the 32-bit word. This means that only word aligned writes are allowed. Byte or half-word-aligned writes will result in a fault. Each writing cycle takes 41 μs. To put this number into perspective it would take about 10 seconds to program the entire memory. The precision when erasing a page is significantly smaller. The smallest memory area that can be erased is called a *page*. As mentioned in section 1.2.2 the flash is divided into 256 pages of 4 kB each (one thousand words per page that is). Erasing a page takes 85 ms, which when put into perspective means that erasing the entire memory page by page would take around 22 seconds (though just erasing the entire memory normally, that is not page by page, would only take about 169 ms).
Furthermore, the write cycles per erase cycle are also limited. A 32 bit word can only be written 2 times before a page erase must be performed, because of something called *flash wear*. This means that even if two write cycles are used to flip nothing more than two bits it is still not possible to write anything more to that area before erasing the page. Exactly what flash wear is and how to avoid it is a whole subject by itself, and we will not dive any deeper into it in this thesis than just asserting that flash memories have limitations on how many times they can be reprogrammed before they start to lose their integrity.
To clarify why this matters, let us imagine a situation where a user wants to replace old firmware with new firmware that is identical but for one word, which has a bit flipped from '0' to '1'. This could potentially mean that 4 kB would have to be erased and then reprogrammed with almost identical data, instead of a simple one word write. Even more devastating, it would likely be impossible to keep track of when this is needed, making it the default case. Ergo, every page that is to have parts of the target image on it would likely have to be erased before anything can be written. This indicates that an implementation would need some sort of erase page management.

2.3 Solutions

Given the information in the preceding sections a series of different options for solutions can be imagined. These options concern the placement of data in the memory, which because of the device's constrained memory and the general nature of the flash memory has to be somewhat optimally selected. They also involve picking the algorithm used for compression, which has to be compatible with the resources provided by Zephyr. This section will describe what these options are and what picking them entails.

2.3.1 Patch Location

There are two appealing options for where the patch could be located. Either reserving a whole new partition for patches (see table 2.2), or storing it on the second partition (see table 2.1). An advantage of creating a new partition is that the patch and the target image will not ever risk obstructing each other. However, a major disadvantage is that this partition will further restrict the already very restricted memory of the device, or necessitate the use of an external memory. Putting the patch at the end of the second partition has the advantage being more memory efficient, but has the drawback of potentially getting very complex when the target or source image is close to the secondary partition in size. Unfortunately, this is rather likely to be the case in a memory restricted devices.

	boot	Lower addresses
source →	slot0	
patch →	slot1	
	scratch	Higher addresses

Table 2.1: Patch located on the primary partition.

	boot	Lower addresses
source →	slot0	
	slot1	
	scratch	
patch →	patch	Higher addresses

Table 2.2: Patch located on a patch partition.

2.3.2 Source image location

At first glance it might seems as only one option exists when it comes to the location of the source image - the primary partition. This is not necessarily the case (see table 2.3), as the secondary image actually also contains a version of the firmware - the version of the firmware that was replaced during the last firmware upgrade. If a patch was to be created based on the firmware located on the second partition this would likely mean a slightly larger patch, but also that nothing would have to be copied from the primary partition, which would both lower the time complexity of the solution and the flash wear.
It would also be possible to combine this with the previous section by having the source, the patch, and the target all located on the secondary partition. This would enable use of the in-place patching algorithm.

	boot		boot	
v1 →	slot0	Upgrade	slot0	← v2
v2 →	slot1	⟶	slot1	← v3
	scratch		scratch	

Table 2.3: Two versions of firmware swapping places during a firmware upgrade. A delta upgrading the system to v3 can be created based on either v2 or v1.

2.3.3 Compression

A suitable compression algorithm has to live up to two criteria: firstly, it has to be implemented into the differencing tool used to create patches; and secondly its corresponding decompression algorithm has to be implemented into our Zephyr application. At this moment Zephyr does not have any support for decompression[17], which means that the easiest solution would be to port a decompression algorithm corresponding to the compression options available for the delta encoding tool.

3
Method

The methods used to develop the product can be divided into: design, procedure, and environment. Design pertains to the choices made regarding the solutions presented in the previous chapter. It also pertains to the documentation of the algorithms that where chosen. Procedure refers to all steps performed in order to generate the results, both in terms of components needed for the solution and in terms of tests needed to validate it. Development environment includes all tools used outside the target system to test and deploy solutions. Together these make up a foundation which we hope to gives reliability to the results presented in the following section.

3.1 Algorithm Design Choices

The design choices made regarding the solution are: choice of algorithm, data locations, and patch metadata. Note that the project is a proof of concept, and thus does not have to be optimal in any manner. This makes it likely that improvements could be made to each of these.

3.1.1 Selecting an Algorithm

The algorithm chosen is an implementation of BSDiff created by Erik Moqvist called Detools [12]. Detools is a delta encoding tool primarily based on Python, but with a C based patching implementation created for embedded solutions. Detools has six different types of compression implemented: BZ2, LZ4, LZMA, Zstandard, heat-shrink and CRLE. However, only one of these are included in the embedded solution - heatshrink.
The solution is highly portable to Zephyr, with two exceptions. The first of these being that the solutions assumes that the target device has file system support. Which zephyr as a standard has not. However, data manipulations (read/write/erase operations) in Detools are performed using call back functions. The functions could easily be replaced by functions manipulating flash directly. They could also easily be developed to manage erasing pages, which would solve the problems stemming from the nature of the flash memory. The second of these is that the solution uses the size of the patch as input. The sizes of data records are difficult to know without a file system or meta data, but this can easily be solved by adding metadata to the patches.
For these reasons we selected Detools patching in combination with heatshrink decompression.

3.1.2 Data Locations

The data locations where heavily influenced by the introduction of Detools. The algorithm uses three separate memory locations, which means the target, patch, and source all have their own partitions. Although a more complex solution would likely require less space,

the significant cut to development time and the reduced risk of bugs made this appear the most sensible choice.
This solution uses the devices storage partition (see table 1.1) as a patch partition, limiting the size of the patch to 24 576 byte. The source image is naturally located on the primary partition, making the patch based on the currently running version of the firmware.
In summary, all data locations and their partitioned areas are thus as displayed in table 3.1.

	boot	0x00000000 - 0x0000C000
source →	slot0	0x0000C000 - 0x00073000
target →	slot1	0x00073000 - 0x000DA000
	scratch	0x000DA000 - 0x000F8000
patch →	patch	0x000F8000 - 0x000FFFFF

Table 3.1: Data locations in the implementation.

3.1.3 Patch Metadata

The patch created by Detools does, as most patch files, normally just contains instructions for which part of the source image binary needs to be changed and how. However, as mentioned above, there is a need for some extra information. This information is included in the form of metadata at the start of the patch. This metadata is composed of the size of the patch and a small message signifying that the patch is new. The message exists so that the device may edit once it has implemented the patch, and in this way keep track of what is currently located in the patch partition.
The metadata consists of a 24 byte long string organized as displayed in table 3.2. The first 8 byte say `"NEWPATCH"`, which is the message that tells the device to upgrade. The last 16 byte are made up of the size, encoded as a string representing a padded hexadecimal number. In practice this string could be, for example `"NEWPATCH0x000000000008c4"`.

Content	`"NEWPATCH"`	Size	Patch contents
Offset	`0x0`	`0x8`	`0x18`

Table 3.2: Patch contents and their offsets.

3.1.4 Summary

The selected algorithm is the Detools BSDiff implementation with heat-shrink compression. The source image is located in the primary partition, the target image is written to the secondary partition, and the patch is located on a patch partition. The patch has a 24 byte header which contains the word `"NEWPATCH"` and the patch size.

3.2 Algorithm descriptions

This project utilizes code written in two other projects, Detools and heat-shrink, to a large degree. The goal of the following section is to provide a short but sufficient explanation of all the external code used, so that the reader may more easily understand the solution as a whole.

3.2.1 Detools

The easiest way to understand the Detools algorithm is by understanding the composition of the patch. The patch created by the Detools differencing algorithm is a sequential patch, and thus has a repeating layout (see table 3.3). Consequently, the patching algorithm consists of the looping of several stages. These are: reading a chunk (512 byte), getting the size of the diff, applying the diff, getting the size of the extra, applying the extra, and adjusting the position of the buffer reading from the source image. In this context the "diff" is data created through additions of the patch data and the source image data, and the "extra" is data created simply by reading patch data.
Algorithm 1 provides an abstract description of the Detools algorithm, meaning that some parts of it, such as initiation of structs and error management, has been omitted for the sake of readability. If the reader wishes to dive deeper into the details of the Detools algorithm - the full source code exists on the code repository[12], along with some documentation.

header	diff 1	extra 1	adj. 1	diff 2	extra 2	adj. 2	...

Table 3.3: Patch layout without metadata (table 3.2) added. The first part of the header is uncompressed and the remainder of the patch is compressed.

3.2.2 Heat-shrink

In the original implementation of BSDiff, the compression algorithm used is suffix sorting, as explained in section 2.1.1.1. However, the Detools implementation of BSDiff does not make use of suffix sorting at all, but rather a completely different algorithm.
The compression library used by Detools is called "Heat-shrink" and was created by a developer at Atomic Object LLC. The project's source code is located on the company GitHub page[19]. It features both static and dynamic memory use, low memory usage, and bounded CPU usage, which is perfect in a resource constrained environment. The algorithm itself is based on the Lempel-Ziv-Storer-Szymanski (LZSS) algorithm for lossless data compression. LZSS is a dictionary coding technique, which means it replaces a string with a reference to a dictionary location of the same string. The idea is to record the locations of previously unseen sub-strings. Sub-strings which, once seen in the text again again, are replaced by a pointer to the first occurrence of this sub-string[20].
Table 3.4 shows an example of how LZSS compression changes a text. The example is taken from an article by A. Ozsoy and M. Swany [21] published in the 2011 IEEE International Conference on Cluster Computing.

0	I meant what I said	0	adj. I meant what I said
20	and I said what I meant	20	and(12,7)(7,8)(2,5)
44		30	
45	From there to here	31	From there to (51,4)
64	from here to there	47	f(46,4)(51,8)(50,5)
83	I said what i meant	55	(24,19)

Table 3.4: LZSS encoding of a small piece of text. The original number of characters is 102, which is encoded into only 56 characters.

3.3 Procedure

In order to create the desired application we needed a ported version of Detools, an idea for a demonstrator used to test the validity of the solution, an idea for a main application design, and a strategy for which tests are to be performed. This section will be the foundation of the results, as it details the strategy behind them.

3.3.1 Detools Porting Strategy

The initializing function of the Detools patching algorithm takes six arguments. Which are:

1. a callback to a function used from the source image into a buffer,
2. a callback to a function used to "jump" forward to an offset in the source image.
3. a callback to a function used for reading from the patch into a buffer,
4. an integer containing the size of the patch,
5. a callback to a function used for writing from a buffer into the target image,
6. a pointer to a struct containing all the resources needed by the functions above, for example file pointers, or in our case a pointer to the flash device and some information about the different partitions.

Given these arguments the following had to be implemented:

- A function which reads the patch header, determines if the patch is new, sets it to "not new", and returns the size of the patch.
- A struct which contains the necessary resources the four functions will need in order to access the flash and read or write to the correct places.
- A function which internalizes this struct.
- four functions which handles interactions with the device's flash memory.
- A main function which starts the patching process, reboots the device if it was successful, and manages any errors that may spawn if it is not.

The development of these callback functions were assisted by the use of code from the Zephyr sample application "Flash Shell", an application which provided examples of functions used for flash manipulation.

3.3.2 Demonstrator Setup

The "demonstrator" refers to the setup surrounding the implementation that is used to demonstrate its functionality. It is a collection of sample applications that together makes up the functionalities included in the main application that are not a part of the delta encoding process, but rather the testing of it.

The first sample application integrated into the solution, that also served as the base for the whole project, was an application called "Blinky" that turned one of the LED lights on the device, LED 0, on or off every second. The program was a good starting point as it was rather short and therefore easy to understand, but also communicated information about the software outwards. It could also easily be modified into blinking LED 1 on the board, creating a new application that was visibly distinguishable from the original. This characteristic could be used to demonstrate that a delta upgrade had been successfully implemented, as the source image currently running could be blinking LED 0 and the target image could be blinking LED 1.

Second, we wanted to integrate some user input functionality into the software. The purpose of this was to control when the patch was applied, which would make it easier to supervise the events relating to an upgrade scenario. The easiest way to do this was using

the device's buttons. Hence, we utilized the sample program "Button". Button executes a function every time a button on the device is pressed. This function could be modified so that pressing the button led to a flag being set to one, indicating to the main loop, running the LED-blinking, that it was time to stop blinking and look for a patch.
The resulting demonstrator was a application blinked LED 0 or LED 1, and when button 0 was pressed started the patch application process. This was the foundation of the main application loop, see algorithm 2, which could be used to launch the implementation of Detools (in the algorithm referred to as "checkForPatch").

3.3.3 Main Application Setup

The finished main application needed functionality concerning the upgrade process added to the demonstrator setup mentioned in the previous section. This functionality was:

- Reading a patch and marking it as read.
- Initialing the data structure.
- Starting patch application.
- Marking the new firmware as an upgrade and rebooting the system.
- Managing potential errors.

In combination with the main loop from the demonstrator and the Detools implementation, these functions would complete the full solution for the product.

3.3.4 Testing Strategy

Testing the validity of the solution is done in two ways. First, using the demonstrator, meaning seeing which LED is blinking. Second, by dumping the contents of the primary partition to a binary file after an upgrade and preforming a byte-wise comparison to the target image.
Reduction of payload can simply be shown through looking at the ratio between target image and patch.

3.4 Development Environment and Tools

This section details the environment and tools used to develop the application. This includes some standard software which will be named but not explained, as their functionality is assumed to be obvious. These are:

- IDE: Visual Studios Code 1.52.1
- Operating System: Ubuntu 20.04.1 LTS (x64)
- C standard: 11
- Compiler: GCC 9.3.0
- Python version: Python 3.8.5
- Board OS: Zephyr 2.4.0
- SDK: Zephyr SDK 0.11.3

In the section below the less known tools used will be detailed. The last section provide some documentation of the python scrips created for automation.

3.4.1 Tools

The Zephyr Project includes a meta tool named West. West manages building, signing, flashing and debugging Zephyr applications. It also has several repository management

tools. In this project West is used for keeping the Zephyr modules updated, as well as building Zephyr applications and signing them.
Building and flashing the boot-loader binary is done using ninja, which is a small build system with similar functionality as the Make tool. Ninja is also what west per default uses in the background for its flashing, building, and debugging operations.
Flashing the firmware image and the patch is done using pyOCD, which is a python based tool used for programming and debugging ARM Cortex-M microcontrollers.
Connecting to the device's console is done with J-Link Real-Time transfer, JLinkRTT-Logger, from Segger.
The script used for dumping flash contents into files utilizes the pynrfjprog API. pynrfjprog is a python wrapper around the nrfjprog dynamic link libraries, which allows for the development of python scripts programming and debugging a nRF device.

3.4.2 Scripts

To automate the setting up of the demonstrator, a make file was created. The make file manages building the boot loader and the application, signing applications, creating patches, and flashing these images to the target device. It also manages connecting to the devices shell, dumping the contents of the flash into files, and installing the necessary dependencies.
Most of these functions are supported using the tools mentioned in section 3.4.1, but a few of them rely on scripts written in Python. These functions are: keeping track of which version of the firmware the device is currently running, adding metadata to the patches, and dumping the contents of certain partitions into files.
The version tracking system is for the sake of simplicity incredibly rudimentary and only keeps track of two versions of the firmware: the currently running one and the most recently built version. It works by running a script before flashing a firmware image and after flashing a patch. The script asks the user if they wish to update the source image by replacing it with the contents of the target image, and does so if the user presses 'y'. This is a far from perfect system, but it serves the purpose of this report by providing an easy system for storing the files needed to create a patch.
The script adding metadata appends metadata to the start of the patch file in accordance with section 3.1.3.
The script for dumping flash contents are, as the name suggests, used for saving flash contents on binary files. The make file has three commands for this: `dump-slot0`, `dump-slot1`, and `dump-flash` (which dumps both 0 and 1). All three of these use a python script which takes a start offset, a slot size, and a target file as input and outputs the slot into the target file. The purpose of this is to make it easier to troubleshoot if a patch for some reason is not successfully applied. For example, the UNIX command `cmp` may be used to get a byte by byte comparison between the desired firmware image with the one created by the patching algorithm on slot one.

Algorithm 1 Detools

```
while patch_offset < patch_size do
  chunk_size = min(patch_size - patch_offset, 512)
  chunk = read_chunk_from_patch(chunk_size)
  while chunk_available(chunk) do
    if STATE == init then
      target_size = read_patch_header();    STATE = diff_size
    end if
    if STATE == diff_size then
      chunk_size = unpack_diff_size(chunk);    STATE = diff_data
    end if
    if STATE == diff_data then
      to_size = min(128,chunk_size)
      if chunk_size == 0 then
        STATE = extra_size
      end if
      to[to_size] = decompress_patch_data(to,chunk)
      from[to_size] = read_from_source(to_size)
      for i = 0 ; i < to_size ; i++ do
        to[i] = to[i] + from[i]
      end for
      write_to_target(to)
      chunk_size -= to_size
      target_offset += to_size
    end if
    if STATE == extra_size then
      chunk_size = unpack_extra_size(chunk);    STATE = extra_data
    end if
    if STATE == extra_data then
      to_size = min(128,chunk_size)
      if chunk_size == 0 then
        STATE = adjustment
      end if
      to = decompress_patch_data(to,chunk)
      write_to_target(to)
      chunk_size -= to_size
      target_offset += to_size
    end if
    if STATE == adjustment then
      source_offset = unpack_offset(chunk)
      move_source_reader_buffer(source_offset)
      if target_size == target_offset then
        STATE = done
      else
        STATE = diff_size
      end if
    end if
    patch_offset += chunk_size
  end while
end while
```

Algorithm 2 Main loop

```
loop
  turn led on/off
  sleep for 1 second
  if buttonPressed then
    return ← checkForPatch(flash)
    buttonPressed ← FALSE
    if return then
      print errorMessage(return)
    end if
  end if
end loop
```

4
Results

This chapter will contain the full documentation of the patching solution that was developed, as well as the tests that were performed in order to determine how successful this solution was at reaching the aims of this work. The body of work surrounding the implementation mainly consists of solutions for porting the Detools algorithm. It also consists of the structure around it which manages device specific things, for example triggering an update and restarting the system. The following section details what this entails.

4.1 Data structure

In the original implementation of Detools a struct containing FILE pointers existed. These file pointers where the only data records needed by the callback functions (see section 3.3.1) to read and write data. Obviously this struct needed to be replaced by a new one in our implementation, as ours do not use files, but rather, offsets in the flash memory. Thus in its place a struct with the following data fields was created:

- `*device`: a pointer which points to the device used to access the flash memory,
- `patch_current`: the current position in the patch partition,
- `patch_end`: the end of the patch partition,
- `from_current`: the current position in the primary partition,
- `from_end`: the end of the primary partition,
- `to_current`: the current position in the secondary partition,
- `to_end`: the end of the secondary partition,
- `write_buf`: how much has been written to the secondary partition since a page was erased.

The fields keeping track of "current" positions might not be intuitive, and their use stems from the original implementation reading data from files using I/O streams. When new data from a data stream is read it will be starting from the last unread byte, meaning there is no need to keep track of positions. Consequently, using the original struct, the callback functions do not have any information about offsets. This is naturally needed in order for the functions to know where the relevant data starts. In the same manner the ends of the relevant memory areas are not marked with a "end of file" or similar, but instead one needs to keep track of this information manually.
The `write_buf` data field is used to keep track of when a full page has been written to the secondary partition, and a new page needs to be cleared in order to make room for more data.

4.2 Main Function

The main function serves to do five things: read the patch header, initialize the data structure, start the Detools patching process, inform the MCUBoot that slot 1 contains

a firmware upgrade, and restart the system. Algorithm 3 shows how this is done in more detail.

Algorithm 3 Main patching function

```
patchSize ← readPatchHeader(flash)
if patchSize < 0 then
  return patchSize
else if patchSize > 0 then
  init(flash)
  detoolsApplyPatch(func1, func2, func3, patchSize, func4, flash)
  requestUpgrade()
  rebootSystem()
end if
return 0
```

4.3 Initialization Functions

The first initialization function is the function reading the patch header, in algorithm 3 referred to as `readPatchHeader`. This function reads the 24 first byte of the patch partition, figures out if the first 8 byte say `NEWPATCH` or not (if not returns 0), and what size the last 16 byte say the patch is. It then writes the word `FFFFFFFF` over the place where `NEWPATCH` used to be. Lastly it returns the size of the patch, or an error code if something went wrong.
Initialization functions also refer to the two functions which run when the function referred to as `init` in algorithm 3 is called. These functions erase the first page of the second partition and initialize all the values in the flash memory struct.

4.4 Callback Functions

There are four callback functions in total. Their uses are: reading from the source image, reading from the patch, writing to the target destination, and "jumping" to another offset in the source image. The two functions used for reading are rather similar and will therefore be explained by the same algorithm (algorithm 4). The two other functions for writing and seeking ("jumping", as it were) are explained by algorithms 5 and 6.
All of these four functions start with a casting of the contents of a struct pointer. In our implementation this struct is the one detailed in section 4.1, which among other things contains a pointer to the device used to manipulate the flash memory and data relating to buffer positions. In the original implementation of Detools these functions cast to a FILE-pointer.
The read functions are rather simple. They use the flash device to read at some offset into a buffer, increase the offset by the buffer size, check if the buffer is outside the partition bounds, and return an error or a message saying the operation succeeded.
The write function does all the same things that the read functions do, but instead of reading into a buffer, it writes from one. However, there are a few additional features which make the write function slightly more complicated - it also contains some management of erasing data and write protections.
Write protection management consists of removing the write protection when writing to the flash, and then applying it again after the write. Erasing page management consists of

keeping track of how many byte have been written to the target area since the last erase, and erasing a new page every time the amount of byte becomes larger than the size of a page. This way we keep flash wear down, as we only have to erase enough pages to make room for the new image.
The seek function is likely the most simple function as it only adds an offset to the data field keeping track of the current offset used to read from the source image.

Algorithm 4 Read function

```
flash ← (flashStruct) *structPointer
read(flash.device,flash.from_current,buf,bufSize)
flash.from_current ← flash.from_current + bufSize
if flash.from_current >= flash.from_end then
  return ERROR
end if
return OK
```

Algorithm 5 Write function

```
flash ← (flashStruct) *structPointer
flash.writeBuf ← flash.writeBuf + bufSize
if flash.writeBuf >= PAGESIZE then
  erasePage(flash,flash.to_current + bufSize)
  flash.writeBuf ← 0
end if
setWriteProtection(flash.device,FALSE)
write(flash.device,flash.to_current,buf,bufSize)
setWriteProtection(flash.device,TRUE)
flash.to_current ← flash.to_current + bufSize
if flash.to_current >= flash.to_end then
  return ERROR
end if
return OK
```

Algorithm 6 Seek function

```
flash ← (flashStruct) *structPointer
flash.from_current = flash.from_current + offset
if flash.from_current >= flash.from_end then
  return ERROR
end if
return OK
```

4.5 Tests

The aims of this thesis are to (i) demonstrate that it is possible to perform delta updates on resource constrained embedded systems and (ii) that these updates cause a meaningful reduction of data transmitted during a firmware upgrade. To test that these aims have been reached we need to verify that the application created can perform a delta update,

Source Info	**Target Info**	**Target Size**	**Patch Size**	**P/T Ratio**	**Notes**
LED 1	LED 0	31.27 kB	845 B	2.6 %	
LED 0	LED 1	31.27 kB	845 B	2.6 %	
LED 1	LED 1	31.27 kB	515 B	1.6 %	Same signature.
LED 1	LED 1	31.27 kB	800 B	2.5 %	New signature.
LED 1 and print	LED 0 and no print	30.47 kB	1.4 kB	4.6 %	
LED 0 and no print	LED 1 and print	31.27 kB	2.0 kB	6.4 %	

Table 4.1: Test results. Where: *Source/Target Info* refers to information about the image; *Target Size* refers to the size of the target image; *Patch Size* refers to the size of the patch; and *P/T ratio* refers to the size ratio between patch and target image. The unit for size is byte.

and that the patch used in this process is significantly smaller than the target image. All the test results can be found in compact form in table 4.1.
The first of these aims was acknowledged by the successful upgrade from an application blinking LED 1 on the board to an application blinking LED 0. It was also possible to flash a new patch "upgrading" the image back to its original state. The validity of this was further confirmed through dumping the upgraded image on the primary partition of the board and comparing it to the target binary used to create the patch.
The second of these aims was acknowledged by noting that the patch created was 845 byte, while that target image was 31 270 byte. This meant that in this case only 2.6 percent of the amount of data needed without delta updates had to be transferred with delta updates. The target and the source file differ by 291 byte, 286 of which are changes to the image trailer.
Additionally, a few extra tests were performed using other types of upgrade images.
The first of these tests was performed using an upgrade image that was based on identical source code, but had been built and signed on a different day. The signed images differed by 256 byte, all located in the image trailer. The generated patch was 802 byte and had a 2.5 percent patch/target ratio.
The second test was performed using a firmware upgrade with error printouts disabled and the LED switched. This yielded a large difference to the binary (over half the image, due to shifts resulting from missing blocks of code) and a generated patch of about 1439 byte byte. The target image was in this case 30 470 byte, which means that the patch/target ratio was 4.6 percent. The patch could be applied without problem.

5
Conclusion

This chapter reflects on the methods and results presented in the previous chapters by returning to the aims presented in the problem statement. The analysis is intended to point out both how the thesis results may be utilized, but also in which way the method might be improved in order to generate even more utility. We also intend to describe what this utility can add to the industry at large. Lastly we suggest some topics which might be suitable for future work.

5.1 Discussion

This sections contains the full analysis of the methods and results in the report. This is divided into application design and tests. Where the first discusses the choice of algorithm and the second the outcome of the tests used to show its validity.

5.1.1 Application Design

This project was made considerably easier through the usage of the Detools package. Not only did the use of the package contribute to reducing the amount of time needed to develop an algorithm by ourselves, it also saved a lot of time otherwise needed for testing the validity of it. Though it is possible that another solution would have offered more control, the package proved to leave little to wish for in terms of portability. Hence, it seems rather unnecessary to reinvent the wheel.
The choices regarding data locations were selected for reduced development time rather than constraining memory usage. This was a consequence of the limited scope of the project, and might have limited the amount of systems for which the results are applicable. On the other hand, a robust system is one of the most essential parts of a unit that cannot receive firmware upgrades easily. Adding extra complexity to the solutions would have increased the risk for bugs, and thereby perhaps deemed the results in this report applicable for even fewer systems.

5.1.2 Test Results

The tests performed in section 4.5 acknowledged that the aims of the report had been fulfilled. A Zephyr application that was able to perform delta updates had been created, and it had been used to reduce the amount of data needed to be transferred. Though, there is still a lot of room for there to edge cases where the solution does not function as intended. In order for our solution to be ready for deployment, this would have to be investigated.
The solution developed has two natural limits to patch sizes. These are: when the patch gets larger than the patch partition (24 576 byte) and when the patch is larger than the target image. Though, it is indeed possible that there are other limits that we have yet to

discover. This could, for example, be at some arbitrary point where the time consumed by the device to apply the patch becomes too large, or at a point where the patch simply consumes too much memory. It is possible that there exists a better size for the patch partition. To discover these things, some further investigations would have to be made.
Notable is also the fact that there seems to exist a minimal patch size at around 800 byte. Given that the target and the source do not share the same signature, a patch cannot be any smaller than that. This seems to stem from the use of cryptographic hash, which generates a different signature every time. The signature has high information density, and can therefore not be compressed much. The patch header and information about the location and size of the signature also adds to patch size.

5.1.3 Ethical and Environmental Perspectives

After careful consideration, we have decided that there are no ethical aspects to consider in this project. However, delta updates do have an environmental impact, as they decrease battery drain.
Energy consumption takes up 60 per cent of total global greenhouse gas emissions, according to the United Nations[22]. Additionally, battery production is associated with several human rights abuses and large scale environmental destruction[23].
While IoT devices likely do not contribute the lions share to any of these issues, an increasing amount of IoT devices are currently entering the market, which means that ways to use less energy for data communication in IoT devices might become a progressively more important issue.

5.2 Implications and Consequences

As mentioned in section 1.1 one of the prospects of this project is to contribute to the development of a free implementation of delta updates in embedded systems. The current iteration of the created application is far from ready for deployment, and does not have much practical use. However it serves as a base for further development.
One such development would be to add support for radio transmissions. As of now, the solution requires a cable for information transfer, which does not solve the problem delta updates solve, namely to allow for cheap updates *without* the need for a cable. Zephyr has several sample applications which demonstrate information transfer via Bluetooth, which would likely be very easy to integrate into our application. Another possible solution could be to transfer data via SMS, which is useful for devices positioned at remote locations.
The long term goal with this type of addition, in combination with some additional testing of the current application, would be to submit the solution to the Zephyr project in the hopes of it becoming a sample application. A sample application is a small application which is integrated into the Zephyr repository, making it easily accessible. Having a full, freely available, implementation of delta updates via SMS would significantly lower the threshold for creating some IoT units. Units that had previously been unfeasible due to the cost that would have to be sunken into their development.

5.3 Topics of Further Work

Working on this report generated quite a few questions that were out of the scope of this thesis to answer. Two of them are left to the reader as suggestions for future work.

5.3.1 Effects of Compiler Optimizations

Like many other embedded systems Zephyr is built using the C programming language. When one compiles code written in C, it is common to let the compiler optimize the code to some degree, to achieve a faster or smaller application. Without optimization all statements are independent. That means that if one stops the program on a break point and changes the program counter to jump to another statement in the function - one would get *exactly* the results one expected based on the source code[9]. In the context of delta encoding this means that if the source code of a firmware upgrade differs by only a few statements the binary will only differ on the instructions representing these specific statements, but no more than that. However, the higher the degree of optimization gets, the more this ceases to be the case. Indeed, the optimization might make it so that even small changes to the source code result in changes to the binary executable so large that it becomes pointless to generate a delta.
Like many other resource constrained systems, Zephyr does as a standard use size optimizations rather than performance optimizations. This might make the optimization problem a bit less of a problem than in the default case. Whether this is the case or not is highly interesting for the field.

5.3.2 Encryption Management

To provide confidentiality of image data while in transport to the unit MCUBoot supports using encrypted images[11]. This means that an encrypted firmware upgrade may be downloaded to the secondary partition to be decrypted when moved to the primary slot. It also means the image located in the primary partition will be re-encrypted once moved to the secondary partition. The purpose of this is to secure confidentiality of the image while not residing on the device.
While it is probably possible to obfuscate the delta via encryption and then decrypt it on the device, doing that was outside the scope of this thesis. However, IoT applications are often used in contexts where cyber security is important, and thus there would probably be great utility to finding such a solution.

5.4 Summary

In summary a proof of concept was successfully created and it was able to significantly reduce the payload needed for firmware upgrades. This solution is far from ready for deployment, but could likely with the addition of radio transfer support have several use cases. An aspiration is that the solution could be expanded upon to create a sample application that can be submitted to the Zephyr project.

Bibliography

[1] F. Firouzi, K. Chakrabarty, S. Nassif, *Intellegent Internet of Things: From Device to Fog and Cloud*, Charm, Switzerland: Springer, 2020. [Online]. Available: https://link-springer-com.proxy.lib.chalmers.se/book/10.1007%2F978-3-030-30367-9#about, Accessed on: 2020-10-08.

[2] Zephyr Project, "About the Zephyr Project", 2020. [Online]. Available: https://www.zephyrproject.org/learn-about/, Accessed on: 2020-10-05.

[3] Zephyr Project, "Device Firmware Upgrade", 2020. [Online]. Available: https://docs.zephyrproject.org/latest/guides/device_mgmt/dfu.html, Accessed on: 2020-10-05.

[4] Wikipedia, "Booting", 2020. [Online]. Available: https://en.wikipedia.org/wiki/Booting#Boot-loader, Accessed on: 2020-10-06.

[5] Zephyr Project, "Legacy Devicetree macro", 2020. [Online]. Available: https://docs.zephyrproject.org/2.3.0/guides/dts/legacy-macros.html#mcuboot-partitions, Accessed on: 2020-10-05.

[6] "MCUboot", GitHub Documentation, 2020. [Online]. Available: https://github.com/JuulLabs-OSS/mcuboot/blob/master/docs/design.md, Accessed on: 2020-10-05.

[7] *nRF52840 DK: Development kit*, Trondheim, Norway, Nordic Semiconductios, 2020. [Online]. Available: https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF52840-DK, Accessed on: 2020-10-06.

[8] "Zephyr Project", 2020, GitHub Repository. [Online]. Available: https://github.com/zephyrproject-rtos, Accessed on: 2020-11-16.

[9] R. Stallman, "Options That Control Optimization" in *Using The Gnu Compiler Collection: A Gnu Manual For GCC Version 4.3.3*, CreateSpace, 100 Enterprise Way, Suite A200, Scotts Valley, CA, USA, 2009. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, Accessed on: 2020-10-08.

[10] C. Percival, "Naive differences of executable code", Aug. 2003. [Online]. Available: http://www.daemonology.net/bsdiff/, Accessed on: 2020-10-21.

[11] "Encrypted Images", MCUboot, GitHub Documentation, 2020. [Online]. Available: https://juullabs-oss.github.io/mcuboot/encrypted_images.html, Accessed on: 2020-10-21.

[12] "Detools", GitHub Repository, 2020. [Online]. Available: https://github.com/eerimoq/detools, Accessed on: 2020-10-14.

[13] J.W. Hunt, M.D. McIlroy, "An Algorithm for Differential File Comparison", Bell Laboratories, Murray Hill, USA, Department of Electrical Engineering, Stanford University, Stanford, USA, 1975. [Online]. Available: https://www.cs.dartmouth.edu/doug/diff.pdf, Accessed on: 2020-10-30.

[14] D. Korn, J. MacDonald, J. Mogul, K. Vo, "The VCDIFF Generic Differencing and Compression Data Format", RFC 3284, Internet Engineering Task Force, 2002. [Online]. Available: https://tools.ietf.org/html/rfc3284, Accessed on: 2020-10-30.

[15] J. Larsson, K. Sadakane, "Faster Suffix Sorting", , vol. 387, nr. 3, ss. 258-272, Nov. 2007 doi: doi:10.1016/j.tcs.2007.07.017. [Online]. Available: http://www.larsson.dogma.net/ssrev-tr.pdf, Accessed on: 2020-11-03.

[16] J.P. MacDonald, "File System Support for Delta Compression", master thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, USA, 2000. [Online]. Available: http://pop.xmailserver.net/xdfs.pdf, Accessed on: 2020-11-04.

[17] "Data compression module integration", GitHub Thread, 2020. [Online]. Available: https://github.com/zephyrproject-rtos/zephyr/issues/26648, Accessed on: 2020-11-08.

[18] Y.S. Nugrogo, H. Hata, K. Matsumoto, "How different are different diff algorithms in Git?", *Empirical Software Engineering*, vol. 25, nr. 1, pp. 790-823, 2020, doi: 10.1007/s10664-019-09772-z. [Online]. Available: https://link.springer.com/article/10.1007/s10664-019-09772-z, Accessed on: 2020-11-04.

[19] "Heatshrink", GitHub Repository, 2015. [Online]. Available: https://github.com/atomicobject/heatshrink, Accessed on: 2021-02-12.

[20] Du, Ke-Lin Swamy, M. N. S, "Wireless Communication Systems - From RF Subsystems to 4G Enabling Technologies - 14.1 Basic Definitions", 2010. [Online]. Available: https://app.knovel.com/hotlink/pdf/id:kt009BKH15/wireless-communication/basic-definitions, Accessed on: 2021-02-12.

[21] A. Ozsoy and M. Swany, "CULZSS: LZSS Lossless Data Compression on CUDA", 2011. IEEE International Conference on Cluster Computing, Austin, USA, 2011, pp. 403-411. doi: 10.1109/CLUSTER.2011.52. [Online]. Available: https://ieeexplore.ieee.org/document/6061071, Accessed on: 2021-02-12.

[22] "Ensure access to affordable, reliable, sustainable and modern energy", 2020. United Nations Sustainable Development Goals. [Online]. Available: https://www.un.org/sustainabledevelopment/energy/, Accessed on: 2021-05-17.

[23] "UN highlights urgent need to tackle impact of likely electric car battery production boom", 2020. United Nations News [Online]. Available: https://news.un.org/en/story/2020/06/1067272, Accessed on: 2021-05-17.