



# **Hellenic Mediterranean University**

Department of Electrical & Computer Engineering

## **SMART FIRMWARE MANAGEMENT IN EMBEDDED SYSTEMS AND IoT DEVICES**

Student: Koutsourelakis Charilaos

Supervisor: Kornaros George, PhD



# **ΕΛΛΗΝΙΚΟ ΜΕΣΟΓΕΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ**

Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών  
Υπολογιστών

## **ΕΞΥΠΝΗ ΔΙΑΧΕΙΡΙΣΗ ΑΝΑΒΑΘΜΙΣΗΣ ΛΟΓΙΣΜΙΚΟΥ ΣΕ ΕΝΣΩΜΑΤΩΜΕΝΑ ΣΥΣΤΗΜΑΤΑ ΙoT ΣΥΣΚΕΥΕΣ**

Φοιτητής : Κουτσουρελάκης Χαρίλαος

Επιβλέπων : Κορνάρος Γεώργιος, PhD

## **DECLARATION OF AUTHORSHIP**

I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgement in this thesis. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged.

## **ACKNOWLEDGEMENTS**

I would like to thank my professor, Dr. George Kornaros, for providing guidance and insight during the course of my thesis. Also, I would like to thank Dimitris Mbakoyiannis for being there through all the brainstorming and providing immense help and support during this journey.

# TABLE OF CONTENTS

<b>DECLARATION OF AUTHORSHIP .....</b>	<b>III</b>
<b>ACKNOWLEDGEMENTS.....</b>	<b>IV</b>
<b>TABLE OF CONTENTS .....</b>	<b>V</b>
<b>TABLE OF FIGURES.....</b>	<b>VIII</b>
<b>ABSTRACT.....</b>	<b>IX</b>
<b>LIST OF ABBREVIATIONS .....</b>	<b>XI</b>
<b>CHAPTER 1 INTRODUCTION .....</b>	<b>1</b>
1.1.    MOTIVATION .....	1
1.2.    PROBLEM STATEMENT.....	2
1.3.    ORIGINAL WORK DESCRIPTION AND ACKNOWLEDGEMENT .....	3
1.4.    OBJECTIVE .....	3
1.5.    RELATED WORKS .....	5
1.6.    STRUCTURE.....	7
<b>CHAPTER 2 SOFTWARE .....</b>	<b>8</b>
2.1.    ZEPHYROS.....	8
2.2.    BOOTLOADER .....	9
2.3.    MCUBOOT .....	9

2.4.	ENCRIPTION AND SIGNATURES .....	10
2.5.	PARTITION MANAGEMENT .....	11
2.6.	GOLIOTH .....	13
<b>CHAPTER 3 DIFFERENTIAL TOOLS AND ALGORITHMS .....</b>		<b>14</b>
3.1.	BSDIFF .....	15
3.2.	DETOOLS .....	16
3.3.	HEATSHRINK .....	17
3.4.	DIFFERENTIAL BINARY .....	17
<b>CHAPTER 4 EQUIPMENT AND SPECIFICATIONS .....</b>		<b>19</b>
4.1.	MICROCONTROLLER FEATURES .....	19
4.2.	FLASH MEMORY .....	20
<b>CHAPTER 5 ARCHITECTURE MECHANISM AND LIMITATIONS.....</b>		<b>22</b>
5.1.	LIMITATIONS .....	24
<b>CHAPTER 6 APPLICATION ANALYSIS .....</b>		<b>26</b>
6.1.	APPLICATION SUMMARY .....	27
6.2.	BOOTLOADER .....	27
6.3.	SOURCE FIRMWARE .....	28
6.4.	TARGET FIRMWARE.....	30
6.5.	PATCH .....	31
6.6.	HEADER PADDING .....	32
6.7.	FIRMWARE MANAGEMENT .....	32
6.8.	DELTA UPDATE.....	34

6.9.	NEW FIRMWARE .....	35
6.10.	UPGRADE .....	35
<b>CHAPTER 7 FAULT-TOLERANCE AND AUTOMATION .....</b>		<b>37</b>
7.1.	AUTOMATION .....	37
<b>CHAPTER 8 SUMMARY AND FUTURE WORK .....</b>		<b>38</b>
<b>BIBLIOGRAPHY .....</b>		<b>39</b>

## TABLE OF FIGURES

FIGURE 1. FOTA VS DELTA UPDATES .....	2
FIGURE 2. ZEPHYROS MICROKERNEL OVERVIEW .....	9
FIGURE 3. MCUBOOT'S IMAGE TRAILER DESIGN .....	11
FIGURE 4. FOTA WORKFLOW CYCLE WITH GOLIOTH.....	13
FIGURE 5. REPRESENTATION OF DATA DIFFERENCING AND PATCH GENERATION.....	14
FIGURE 6. B-L475E-IOT01A .....	19
FIGURE 7. ESP8266 WIFI MODULE AND PINOUT.....	20
FIGURE 8. FLASH MEMORY MAP FOR STM32L475XX/476XX/486XX DEVICES [73, P. 80] ...	21
FIGURE 9. CUSTOM ARCHITECTURE MECHANISM.....	23
FIGURE 10. DIFFERENT APPROACHES FOR THE PATCH STORAGE .....	24
FIGURE 11. HEX OF THE SOURCE BINARY, SHOWING THE HEADER OF THE BOOTLOADER. ....	28
FIGURE 12. SOURCE APPLICATION (VERSION 1.0.0) SERIAL OUTPUT.....	29
FIGURE 13. BOOTLOADER INFORMATION OF THE SOURCE FIRMWARE. ....	30
FIGURE 14. PATCH INFORMATION.....	31
FIGURE 15. DETAILED INFORMATION OF THE PATCH BINARY, SHOWING THE PADDED HEADER. .	32
FIGURE 16. GOLIOTH DASHBOARD 1/2 .....	33
FIGURE 17. GOLIOTH DASHBOARD 2/2 .....	33
FIGURE 18. FLOW OF DELTA UPDATE .....	34
FIGURE 19. SOURCE AND TARGET FIRMWARE, RECOGNIZED BY THE MCUBOOT BOOTLOADER. .	35
FIGURE 20. VERSION 1.0.1 OF THE APPLICATION.....	36
FIGURE 21. SHORT PATCH HEADER ERROR.....	37



## ABSTRACT

The rapidly multiplying embedded devices on the Internet of Things (IoT) require remote management capabilities for constant updates of the embedded software. With a multitude of assets dispersed across geography, terrain, industrial environments, office spaces or customer locations in an IoT deployment, the task of managing remote software updates is critical to effective operations of an IoT cloud. Industry verticals such as automotive, manufacturing, mining, agriculture, e-health and connected spaces rely ever more on a high performing IoT cloud for desired outcomes. Remote Management of IoT Solutions involves firmware upgrades, diagnostics, basic troubleshooting, security patches, and configuration changes, all of which could be delivered remotely and securely to the IoT devices - wherever those may be - from a central location. For a technician to service each device for an update would be prohibitive. As a result, over-the-air (OTA) updates remain the only viable option. The software updates in certain key IoT segment verticals are delivered over limited bandwidth networks, which causes long wait times and increases failure rates. With connectivity and data consumption often at a premium, the delivery of over-the-air (OTA) updates to embedded devices on a cellular or other wireless network can quickly become a major financial burden. With IoT, as the number of devices grows and the frequency of software updates increases, the upsurge in network cost and idle time for upgrades gets in the way of user satisfaction, experience, and productivity. In addition, software updates are often delivered to memory constrained devices, which means the device agent that executes these updates must be lean enough to fit and operate within the constraints of the device. This thesis will develop methods for efficient OTA updates for resource constrained devices.

## ΠΕΡΙΛΗΨΗ

Οι ταχέως πολλαπλασιαζόμενες ενσωματωμένες συσκευές στο Διαδίκτυο των Αντικειμένων (Internet of Things) απαιτούν δυνατότητες απομακρυσμένης διαχείρισης για συνεχείς ενημερώσεις του ενσωματωμένου λογισμικού. Με ένα πλήθος στοιχείων διασκορπισμένα σε τοποθεσία, έδαφος, βιομηχανικά περιβάλλοντα, χώρους γραφείων ή τοποθεσίες πελατών σε μια IoT ανάπτυξη, το έργο της διαχείρισης απομακρυσμένων ενημερώσεων λογισμικού είναι κρίσιμο για την αποτελεσματική λειτουργία ενός IoT cloud. Οι κλάδοι όπως η αυτοκινητοβιομηχανία, η εξόρυξη, η γεωργία, η ηλεκτρονική υγεία και οι συνδεδεμένοι χώροι βασίζονται όλο και περισσότερο σε ένα IoT cloud υψηλής απόδοσης για επιθυμητά αποτελέσματα. Η απομακρυσμένη διαχείριση των IoT συσκευών περιλαμβάνει αναβαθμίσεις υλικολογισμικού, διαγνωστικά, βασική αντιμετώπιση προβλημάτων, ενημερώσεις κώδικα ασφαλείας και αλλαγές διαμόρφωσης, τα οποία θα μπορούσαν να παραδοθούν εξ αποστάσεως και με ασφάλεια στις συσκευές IoT - όπου κι αν βρίσκονται - από μια κεντρική τοποθεσία. Για έναν τεχνικό για να επισκευάσει κάθε συσκευή για μια ενημέρωση θα ήταν απαγορευτικό κόστος. Ως αποτέλεσμα, οι ενημερώσεις over-the-air (OTA) παραμένουν η μόνη βιώσιμη επιλογή. Οι ενημερώσεις λογισμικού σε συγκεκριμένους κλάδους βασικών τμημάτων IoT παρέχονται μέσω δικτύων περιορισμένου εύρους ζώνης, γεγονός που προκαλεί μεγάλους χρόνους αναμονής και αυξάνει τα ποσοστά αποτυχίας. Με τη συνδεσιμότητα και την κατανάλωση δεδομένων συχνά σε υψηλή τιμή, η παράδοση ενημερώσεων over-the-air (OTA) σε ενσωματωμένες συσκευές σε ένα κυψελοειδές ή άλλο ασύρματο δίκτυο μπορεί γρήγορα να μετατραπεί σε μια σημαντικά οικονομική επιβάρυνση. Με το IoT, καθώς ο αριθμός των συσκευών και η συχνότητα των ενημερώσεων λογισμικού αυξάνονται, η αύξηση του κόστους δικτύου και ο χρόνος αδράνειας για αναβαθμίσεις εμποδίζουν την ικανοποίηση, την εμπειρία και την παραγωγικότητα των χρηστών. Επιπλέον, οι ενημερώσεις λογισμικού συχνά παραδίδονται σε συσκευές με περιορισμένη μνήμη, πράγμα που σημαίνει ότι ο παράγοντας συσκευών που εκτελεί αυτές τις ενημερώσεις πρέπει να είναι αρκετά αδύνατος ώστε να χωράει και να λειτουργεί εντός των περιορισμών της συσκευής. Αυτή η διατριβή θα αναπτύξει μεθόδους για αποτελεσματικές ενημερώσεις OTA για συσκευές περιορισμένων πόρων.

## LIST OF ABBREVIATIONS

<b><u>Term</u></b>	<b><u>Description</u></b>
<b>IoT</b>	Internet of Things
<b>IIoT</b>	Industrial Internet of Things
<b>FOTA</b>	Firmware Over-the-Air
<b>MCU</b>	Microcontroller Unit
<b>OMA</b>	Open Mobile Alliance
<b>BLE</b>	Bluetooth Low-Energy
<b>RTOS</b>	Real-Time Operating System
<b>LoRa</b>	Long Range Wide Area Network
<b>LWM2M</b>	Lightweight Machine to Machine
<b>ECU</b>	Electronic Control Unit



# Chapter 1 Introduction

## 1.1. Motivation

Firmware Updates are very important in both home and industry automation (IoT and IIoT), whether the domain is that of automotive, industrial, health or other [1]. The requirement for firmware over-the-Air (FOTA/OTA) updates becomes necessary when using resource-restricted devices that may have constraints regarding energy consumption, limited connectivity, remote geographic location, etc.

Firmware over-the-air is used to remotely update the software on devices such as smartphones, embedded systems and IoT devices and allows distribution of updates and bug fixes without the need for physical access to the device, making it more convenient and efficient for both the end user and manufacturer. Additionally, FOTA can also be used to add new features or improve the performance of a device via software, without the need for a complete hardware upgrade.

Firmware management is essential when a group of devices requires a firmware upgrade. Simultaneously rolling out updates is crucial to prevent firmware-related performance disparities. By automating the distribution of updates to all devices and monitoring the health of the hardware globally, automated updates ensure that the firmware remains up to date.

Managing the software of a resource-constrained microcontroller unit can become a difficult task when major firmware updates are in order. Specifically, devices that communicate with a master node or computer database via protocols such as LoRaWAN [2], Cellular [3] or BLE [4], make the update of the microcontroller a time-consuming task that not only affects the developer, but also the limited resources of the device.

The automation of OTA updates, though, whether it is mass or single device oriented, brings to the table the security issue. Both the software and the hardware need to have a low-level communication so that the device can check the validity of the software. This can be accomplished by providing a secure and trusted hardware layer that can recognize whether the nature of the software is of a malicious or benevolent nature. On this level, the firmware may have a signature to verify its origin with the bootloader or, even better, share an encryption key, to provide an extra layer of protection.

The main contribution of this thesis is to implement firmware updating with the use of incremental binary differencing (also known as patching) that reduces both transmission time and energy consumption.

## 1.2. Problem Statement

In cases that require the device to be in a remote location and have limited means of transmission with the node, the firmware may take quite some time to reach the end node. This may be due to the size of the firmware, or the communication protocol used by the device. A solution to this problem can be the implementation of firmware updates with incremental (delta) updates, which is the issue this thesis is focused on.

A delta update is a type of firmware or software update that includes only the changes made since the previous version of the software, rather than the entire updated software package. As a result, only the differences between the previous and present firmware are downloaded and installed, thus reducing the size of the update and the time it takes to download and install it [5].

Delta updates are often used for firmware over-the-air updates, not only for embedded devices, but also for smartphones, modems, smart IoT devices, etc. This is because they are a more efficient option than entire firmware updates since they consume less bandwidth, which can be vital for devices with limited storage, internet connectivity or energy consumption. Delta updates are also known as differential updates, incremental updates, or binary differentials.

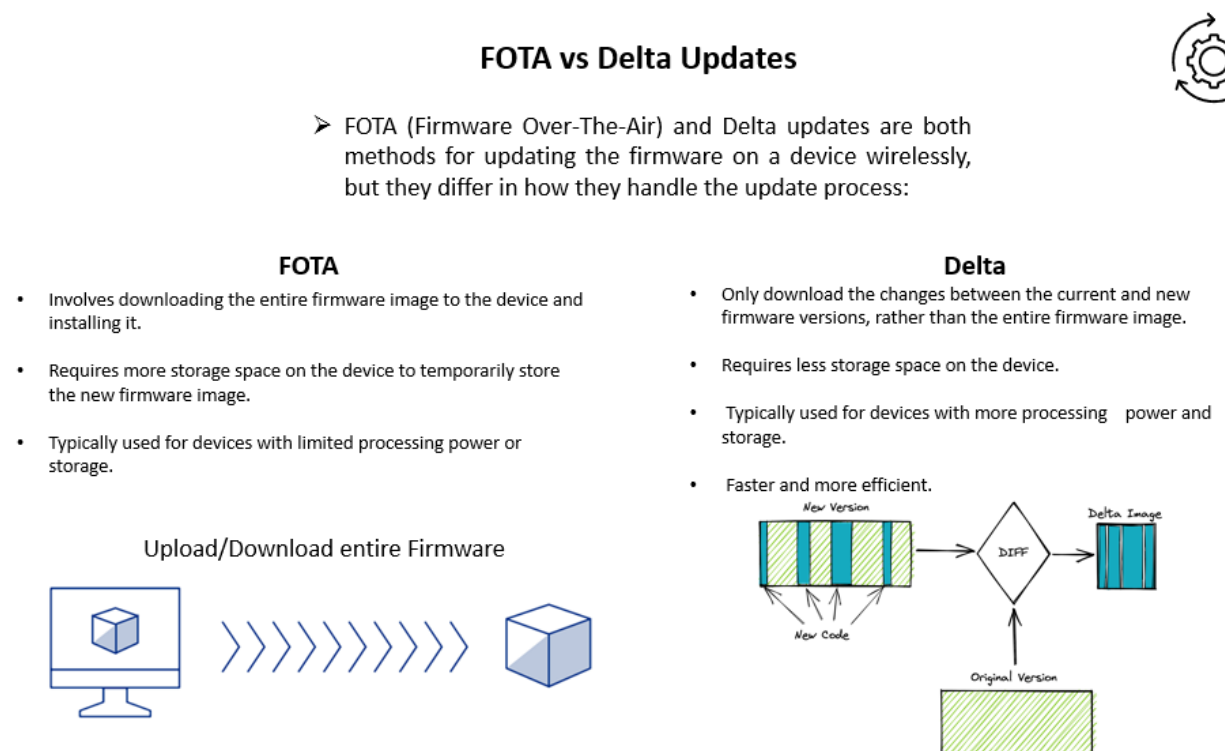


Figure 1. FOTA vs Delta Updates

### 1.3. Original Work Description and Acknowledgement

This thesis is based on the work of Linnéa Lindh's bachelor thesis, "**Delta Updates for Embedded Systems: An Implementation of Firmware Patching for Zephyr RTOS**" [6], which provided the original delta encoding algorithm.

Lindh's work was a stepstone for this thesis as it provided help and guidance in the delta encoding implementation for the Zephyr RTOS. Nonetheless, this work took her implementation a step further by providing several contributions to the overall algorithm.

Such contributions include:

1. Over-the-Air delta updates
2. Firmware version control
3. Documented experimentation with IoT management platforms, such as Golioth, Eclipse's Leshan [7] and hawkBit [8]
4. Port to B-L475E-IOT01A Cortex-M architecture

Lindh's original work is further explained in a later chapter, which demonstrates manual flashing of the patch in the storage partition and then executing the delta update algorithm on Nordic's nRF52840 Development Kit [9]. In the case of this thesis, though, the original source code of the delta algorithm was altered to match the specifications of the board used, the B-L475E-IOT01A, since there were important differences in the architecture of the devices. Details on the code alteration and architecture differences can be found in a later section.

### 1.4. Objective

The aim of this thesis is to present an application focused on firmware updates and version management with the implementation of incremental (delta) updates for the target-board used, the B-L475E-IOT01A. The application was developed specifically for this Arm Cortex-M4 core-based STM32L4-series board, but the mechanism developed for the delta algorithm can be altered to suit numerous microcontroller architectures with notable flash memory and pagination differences. The purpose is to provide further research, experimentation, and analysis for the firmware over-the-air branch of embedded devices with the usage of 'patching' the source code, rather than updating the whole firmware.

Firmware Overt-the-Air (FOTA) updates are a necessity today due to the increasing complexity and dependence on technology in our daily lives. With the rise of smartphones, tablets, and IoT devices, the need for quick and easy updates to firmware has become more critical than ever. FOTA allows manufacturers to address security concerns, fix bugs, and add new features and improvements to devices in a timely and efficient manner. The need for FOTA is further emphasized by the rapid pace of technological advancement and the constantly evolving security landscape, which necessitates regular firmware updates to ensure devices remain secure and perform optimally. Additionally, FOTA saves time and money by eliminating the need for

users to physically bring their devices to a service center for updates. Overall, FOTA is a necessity today as it enables devices to receive updates and improvements quickly, easily, and securely, ensuring that they continue to meet the needs and expectations of users.

FOTA is also an important component of fleet management. Fleet management involves the monitoring and maintenance of a fleet of vehicles or other assets, often across a wide geographic area. FOTA allows fleet managers to remotely update the firmware of devices, which include updating vehicle software or ensuring that IoT devices are running the latest version of their operating system.

FOTA in fleet management has several advantages. First, it can significantly reduce downtime as vehicles or other assets do not need to be physically brought to a service center for firmware updates. This can result in cost savings and increased productivity as units can continue to operate while updates are being performed. Second, FOTA can improve the security of units by enabling manufacturers to quickly address vulnerabilities or bugs. This is particularly important in fleet management as assets are often dispersed across a wide geographic area and can be more vulnerable to security threats. Finally, FOTA can ensure that fleet managers have access to the latest features and improvements, which can improve the overall efficiency and effectiveness of their operations.

In essence, FOTA is an important tool for fleet management, enabling fleet managers to remotely update the firmware of vehicles or other assets in a timely and efficient manner, resulting in improved productivity, security, and performance.

In cases where the device(s) are connected via communication protocols, such as LoRaWAN (Long Range Wide Area Network), BLE (Bluetooth Low Energy), Zigbee [10], CoAP (Constrained Application Protocol) [11], LWM2M (Lightweight Machine to Machine) [12], etc., updating the device from the source (current) to the target (updated) firmware can be time consuming for the device, energy-wise. This is because the device may take a long time to retrieve the firmware due to limited payload.

For example, in a scenario in which LoRaWAN is the chosen communication protocol between a master and end nodes, the transmission of a firmware update would be limited, since end-devices transmit only a few bytes per transmission time [13].

Overall, firmware update via any of the aforementioned communication protocols depends on multiple parameters, such as the use-case, device requirements, geolocation of the device, etc. In the use-case where the device is in a remote location, LoRaWAN can be a good option, especially since LoRa has typically lower data rate than other wireless communication technologies, such as Wi-Fi or 4G, which can make firmware updates slower. But, since LoRa is a low power technology, devices may have to be in close proximity for firmware updates, which may not be possible in certain situations.



## 1.5. Related Works

This section covers works related to the FOTA and incremental update processes in different scenarios and approaches. The topics covered refer to the automotive industry (Electronic Control Units), firmware hotpatching, also an object code optimization technique called “Feedback linking”, and finally, a fleet management of IoT devices used in apiaries.

Firmware over-the-air updates in Electronic Control Units (ECUs) are a way to remotely update the firmware of a device over a network. This allows manufacturers to release updates and bug fixes to devices in the field without requiring physical access to the device [14]–[16].

In the automotive industry, FOTA is becoming increasingly important as it allows for updates to be performed on cars' ECUs at the customer's location, rather than at a dealership. This can greatly reduce fleet management costs and minimize downtime for customers. Additionally, as cars become more connected, FOTA enables the updating of the software stack of the vehicle, including the infotainment, telematics, and other systems. However, it's important to note that as with any software update, security and integrity of the update must be ensured to prevent malicious actors from compromising the system. This can be done by using secure communication channels, digital signing of the update, and verifying the authenticity and integrity of the update before it is installed. Overall, FOTA in ECUs allows for improved maintenance and security of devices in the field and is becoming increasingly important in the automotive industry and other cyber-physical systems.

What this thesis presents, as opposed to the aforementioned work, is the ability of the microcontroller unit to detect and apply a differential update, and thus saving energy resources, time, and storage.

Another notable work of similar nature is that of Rapidpatch [17], which presents a technique for hotpatching firmware on real-time embedded devices. The authors argue that traditional methods for updating firmware on such devices, which involve stopping the device, flashing the new firmware, and then restarting the device, can result in significant downtime and potential loss of data.

RapidPatch works by dynamically updating the firmware of the device at runtime without stopping the device or interrupting its operation. The authors describe the design and implementation of RapidPatch and demonstrate its effectiveness through experiments on an embedded system. They show that RapidPatch can achieve low-latency updates, with the device's downtime reduced by up to 90% compared to traditional firmware update methods. RapidPatch is a promising technique for updating firmware on real-time embedded devices, particularly in safety-critical applications where downtime and interruption can have serious consequences and suggest that future work could focus on optimizing the performance of RapidPatch and exploring its use in other domains, such as the Internet of Things and cloud computing.

The major difference with the work presented in this thesis is the ability of RapidPatch to perform rebootless updates, meaning that a system reboot is not necessary to perform an upgrade of the firmware. Such a feature is promising, since the action of an update does not interrupt the process of the application, which in other cases, such as that of this thesis, a reboot is needed after an update in order to apply the target firmware as the primary one. Such an action requires the board to be idle for a few seconds, which can lead to serious security vulnerabilities.

Another technique that utilized incremental updates is presented under the name “Feedback Linking”, in a paper written by Carl von Platen and Johan Eker [18], that optimizes the layout of object code in a program so that updates can be applied efficiently. The authors argue that the traditional approach of updating entire program binaries is inefficient, and that optimizing the layout of code can help to reduce update times and improve performance.

Feedback Linking works by analyzing the behavior of a program at runtime and then using that information to generate a layout that optimizes the placement of frequently executed code. The authors demonstrate that Feedback Linking can reduce update times by up to 80% and can also improve performance in some cases. The authors conclude that Feedback Linking is a promising approach to optimizing the layout of object code for updates and that it could be useful in a variety of settings, including embedded systems and web browsers. They also suggest several directions for future research, including exploring the use of Feedback Linking in combination with other optimization techniques and investigating the tradeoffs between update time and code size.

Feedback linking is quite similar to the work presented in this thesis, as both works focus on the reducing the size of the firmware code and improving the delta update process. The difference lies in the fact that even the slightest alteration in the source code has the potential to impact almost every memory position, which in turn necessitates a complete rewrite of the memory, according to the authors. Although, via the use of the BSDiff algorithm, as a part of the DeTools library, the work presented here overcomes this challenge via the use of pointers, as further explained in section 3.1.

Finally, In the paper "IoT Apiary Fleet Management with Jenkins" [19] a system for managing a fleet of IoT devices used in an apiary is described. The system uses Jenkins [20], an open-source automation server, for managing the firmware updates and fleet management tasks.

FOTA is used in this system to remotely update the firmware of the IoT devices used in the apiary. Fleet management in this system involves monitoring the IoT devices used in the apiary, including their locations, health status, and other performance metrics. The system also enables the fleet managers to remotely diagnose and troubleshoot any issues that arise in the IoT devices.

Jenkins is used in this system for automating various fleet management tasks, including deploying new firmware updates using FOTA, running tests on the updated firmware, and monitoring the performance of the IoT devices. The authors suggest that the system could also be used to automate other tasks, such as scheduling maintenance activities, monitoring the environmental conditions in the apiary, and optimizing the operation of the IoT devices.

Overall, the paper demonstrates the potential for FOTA and fleet management to improve the management of IoT devices used in agricultural applications. The use of Jenkins for automating various tasks in the fleet management process enables more efficient and effective management of the IoT devices, reducing downtime, and minimizing maintenance costs.

This work, which describes the necessity and usage of fleet management in IoT and embedded devices in a real-world scenario, could be immensely improved with the incremental update process, as presented in this thesis. This is due to the fact that having a fleet of numerous devices that may need to be updated all at once, sending and receiving firmware updates can be

time and energy consuming, especially if more than a few devices are connected to the same network.

## 1.6. Structure

This thesis is organized as follows:

**Chapter 2** presents the software used for the implementation of this project, such as the Real-Time Operating System, the bootloader and the IoT management platform. **Chapter 3** presents the differential tools and algorithms used for the generation of the patch, along with the compression algorithm and description of differential binaries. **Chapter 4** presents the specifications of the equipment/microcontroller, along with a description of the flash memory layout of the target board. **Chapter 5** presents the custom mechanism that was developed for the architecture of the target board along with the limitations faced during the course of this thesis. **Chapter 6** presents the application analysis, the steps needed to reproduce the application's outcome, along with each step's results with visual representation. In **Chapter 7**, the subjects of fault-tolerance and automation are presented. Finally, **Chapter 8** summarizes the work that was done for the purpose of this thesis, along with potential future work material.

## Chapter 2 Software

The software that was used for this thesis consists of the operating system, the bootloader and the IoT management platform, all of which are described in the following sections of this chapter.

For the development of the application with firmware over-the-air capabilities, a real-time operating system (RTOS) was used, along with a bootloader, which is a necessary part of the operating system used if firmware updates are part of the target application.

Regarding the IoT management platform, there were multiple alternatives to the chosen one, Golioth, most notable Eclipse's Leshan [7], which is an OMA (Open Mobile Alliance) [21] Lightweight Machine To Machine (LWM2M) protocol from the Open Mobile Alliance for Machine to Machine, and Eclipse's hawkBit [8], which is a domain-independent back-end framework for rolling out software updates to constrained edge devices.

### 2.1. ZephyrOS

ZephyrOS [22] is an open-source real-time operating system (RTOS) [23] for connected devices and systems. It is designed for use in embedded systems and Internet of Things devices, and is intended to be highly modular, secure, and energy efficient. It is characterized by its small size, low power consumption, and high reliability, making it well-suited for use in resource-constrained devices such as sensors, actuators, and other IoT devices.

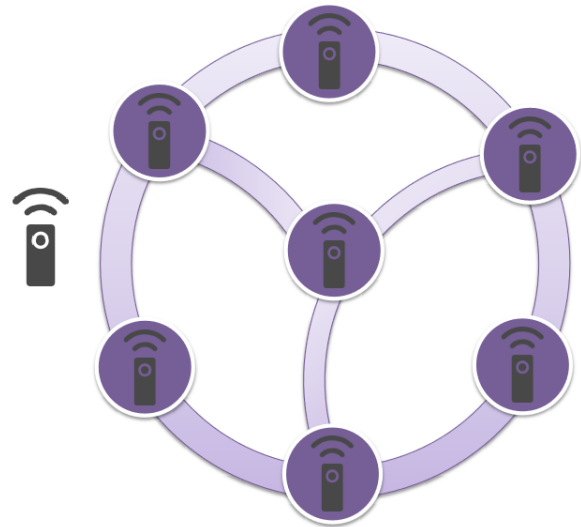
ZephyrOS is built using a micro-kernel architecture, which allows for a small footprint and minimal memory requirements. Being a micro-kernel operating system means that the kernel only contains the essential components required to manage the system's hardware and provide basic services, while other components run as separate processes outside the kernel. This design allows for a more modular, scalable, and secure operating system, as it minimizes the attack surface and reduces the risk of bugs and other security vulnerabilities in the kernel.

ZephyrOS also includes support for multiple architectures and a wide range of device peripherals. Additionally, the ZephyrOS supports a variety of communication protocols such as Bluetooth, Zigbee, Thread [24], CoAP, MQTT [25], LWM2M, and can be integrated with other software like Machine Learning and Edge Computing [26] frameworks. ZephyrOS is developed and maintained by the Zephyr Project, an open-source community that is governed by the Linux Foundation[27].

The Zephyr kernel supports multiple architectures [28], including ARM (Cortex-A, Cortex-R, Cortex-M), Intel x86, ARC, Nios II, Tensilica Xtensa, and RISC-V, SPARC, MIPS, and a plethora of supported boards [29].

## Zephyr Microkernel Overview

- Supplements the capabilities of the nanokernel to provide a richer set of kernel features
- Suitable for systems with heftier memory (50 to 900 KB)
- multiple communication devices (like Wi-Fi and Bluetooth® Low Energy) and multiple data processing tasks
- Examples of such systems include:
  - Fitness wearables
  - Smart watches
  - IoT wireless gateways



*Figure 2. ZephyrOS microkernel overview*

### 2.2. Bootloader

The Bootloader is the initial firmware that runs when an embedded system is powered on or reset. Its primary goal is to initialize the system and transfer control to the application/RTOS. The bootloader is responsible for initializing the hardware, checking the integrity of the operating system, and providing a user interface for selecting different operating systems or firmware images to boot. It also allows for easy updates and recovery of the operating system or firmware. Additionally, the bootloader also facilitates the loading of data.

### 2.3. MCUboot

MCUboot [30], [31] is an open-source, secure bootloader for microcontroller-based systems. It allows for secure firmware updates over-the-air and provides a secure boot process that verifies the authenticity and integrity of firmware images before loading them into the memory. MCUboot also supports multiple firmware images, allowing for easy rollback (downgrade) to previous firmware versions in case of issues. It is designed to be integrated into a wide range of microcontroller-based systems and supports a variety of communication protocols such as UART, I<sup>2</sup>C, and SPI.

## 2.4. Encryption and Signatures

MCUboot includes encryption to increase the security of firmware updates in IoT and embedded systems. This encryption protects the firmware image during both storage and transmission, ensuring that it is only accessible by authorized individuals. Before being stored on external storage or transmitted over a network, the firmware image is encrypted by MCUboot. The bootloader then decrypts the image before loading it into memory. This helps to prevent unauthorized access and tampering with, providing a secure way to distribute firmware updates to devices in the field.

Every bootloader generates a header at the beginning of the binary. The header file in MCUboot serves as a blueprint for the bootloader, specifying the constants, macros, and structures it requires, including information about the format of the image header, image type, version, and size. Additionally, the header file outlines the communication protocol between the bootloader and the host and sets forth the format of the firmware image that will be loaded and executed by the bootloader.

The image is encrypted using AES-CTR-128 or AES-CTR-256 [32], with a counter that starts from zero (over the payload blocks) and increments by 1 for each 16-byte block. In order to distribute the key, new type-length-value (TLVs) [33] records are defined. These records are part of the trailer of the image (metadata), which can be found at the end of the image binary. The key that is stored in the new TLVs can be encrypted using either RSA-OAEP [34] [35], AES-KW [36] (128 or 256 bits), ECIES-P256 [37] or ECIES-X25519 [38].

When a new upgrade process is executed, MCUboot checks that the target firmware, which is downloaded in the secondary slot, has the `ENCRYPTED` flag set and the required TLV with the encrypted key. The bootloader then uses its internal private key to decrypt the TLV containing the key. Assuming there are no errors, the validation process will be initiated. It involves decrypting the blocks, followed by the upgrade process that reads the blocks from the *secondary slot* and writes them, after decryption, to the *primary slot*.



Figure 3. MCUboot's image trailer design

## 2.5. Partition Management

In order for the bootloader to be included in a microcontroller unit, a partition table must be set for the flash-map of the board. Each board has its own unique specifications and flash memory size, so the partition table must be set according to the device's tree. ZephyrOS provides support for a plethora of boards and shields, and many of them provide a preconfigured partition table, if the device is officially tested by the MCUboot and ZephyrOS developers.

The partition layout must contain the partitions described below for the bootloader to work properly [39]:

- boot\_partition: The bootloader area contains the bootloader image itself.
- slot0\_partition: Contains the primary (source) image of the firmware.
- slot1\_partition: Contains the secondary (target) image of the firmware.
- scratch\_partition: Used to swap images for firmware upgrade.
- storage\_partition (optional): File system partition.

In the case of this thesis, since the board used was the B-L475E-IOT01A, the devices' partitions are divided, as per Zephyr's default device tree, in the internal and external flash. This is due to fact that the microcontroller has a 64-Mbit Quad-SPI external flash memory [40] alongside the internal 1MB internal flash memory.

However, since the original idea of this thesis was to develop a generic implementation of the delta algorithm, and not to be targeted to a specific architecture, the Quad-SPI external flash memory was disabled, since most boards are not equipped with a similar flash memory.

The default partition table of the bootloader can be seen below on **Table 1.1**, along with the custom alterations on **Table 1.2**.

**Table 1.1: Preconfigured internal and external flash partitions**

#### Internal Flash

Partition	Offset Start	Offset End	Size in Hex	Size in Kilobytes
boot_partition	0x00000000	0x00010000	0x00010000	65.536 KB
slot0_partition	0x00020000	0x000F8000	0x000D8000	884.736 KB
scratch_partition	0x000F8000	0x000FC000	0x0004000	16.384 KB

#### External Flash

slot1_partition	0x00000000	0x000D8000	0x000D8000	884.736 KB
storage_partition	0x000D8000	0x000E0D68	0x00008D68	7.168 KB

**Table 1.2: Custom flash partitions [41]**

#### Internal Flash

Partition	Offset Start	Offset End	Size in Hex	Size in Kilobytes
boot_partition	0x00000000	0x00010000	0x00010000	65.536 KB
slot0_partition	0x00020000	0x0008C000	0x0006C000	442.368 KB
slot1_partition	0x0008C000	0x000F8000	0x0006C000	442.368 KB



scratch_partition	0x000F8000	0x000FA000	0x00002000	8.192 KB
storage_partition	0x000FA000	0x000FFFFF	0x00006000	24.575 KB

## 2.6. Goliath

Goliath [42] is a cloud-based IoT management platform that offers control over multiple embedded devices of numerous architectures along with monitoring and management of IoT devices and sensors, as well as providing data visualization and analytics capabilities.

Goliath has collaborated with the Zephyr Project™, since it is an open-source, secure and architecture-wide option among the RTOS alternatives that are available under the Linux Foundation. Similar real-time operating systems, each one with unique characteristics, are listed as followed: Xenomai [43], OpenRTOS/freeRTOS [44], RISC-V Software Ecosystem [45], etc.

Goliath also has developed its own open-source client Software Development Kit (SDK) that is built on-top of ZephyrOS and provides its own libraries for extensive development options with the Goliath platform.

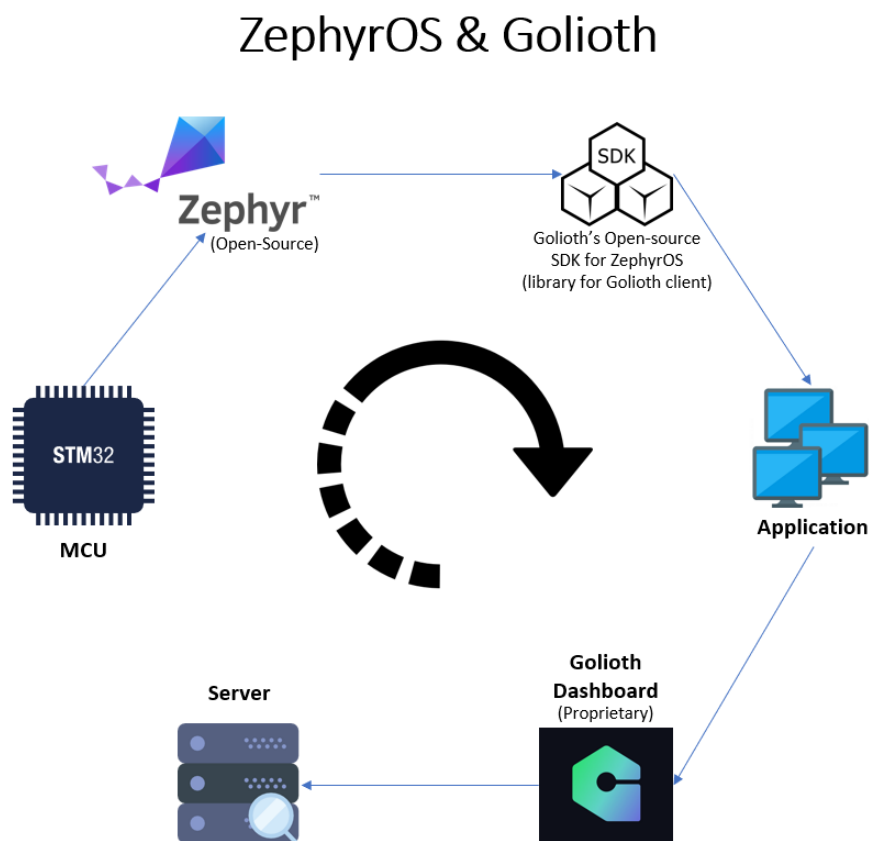


Figure 4. FOTA workflow cycle with Goliath

## Chapter 3 Differential Tools and Algorithms

Data differencing is the production of a file that consists of the differences between two datasets, called source and target. The produced file is considered a “patch” or a differential binary [46]. Such an example is that of the diff utility on Unix and Unix-like operating systems, which produces line-by-line differences between text files [47] .

Binary data differencing, also named delta encoding, is a technique used to efficiently store and transmit changes made to a binary file. This process involves identifying the differences between the original file (source) and the revised version (target), and then encoding only their differences rather than the entire file. By doing so, the amount of data that needs to be transmitted or stored is greatly reduced, leading to improved performance and reduced storage requirements [48].

Differential binaries are primarily meant to be constructed using two functions, copying and insertion. This is accomplished by techniques such as substring matching [49] or hashing techniques, where a comparison is done between the new and the old file and adding the different parts by indexing them via hash tables in the new patch file. The disadvantage of such techniques is the alternation of blocks that occurs in the code, even with a simple change of a few bytes, leading in changed addresses in the code that is located after the modified region. This happens when the case of comparison are two blocks of code that have not been changed but are located after the modified region.

In the next section, the *BSDiff* [50] algorithm will be explained along with the solution of the problem stated above.

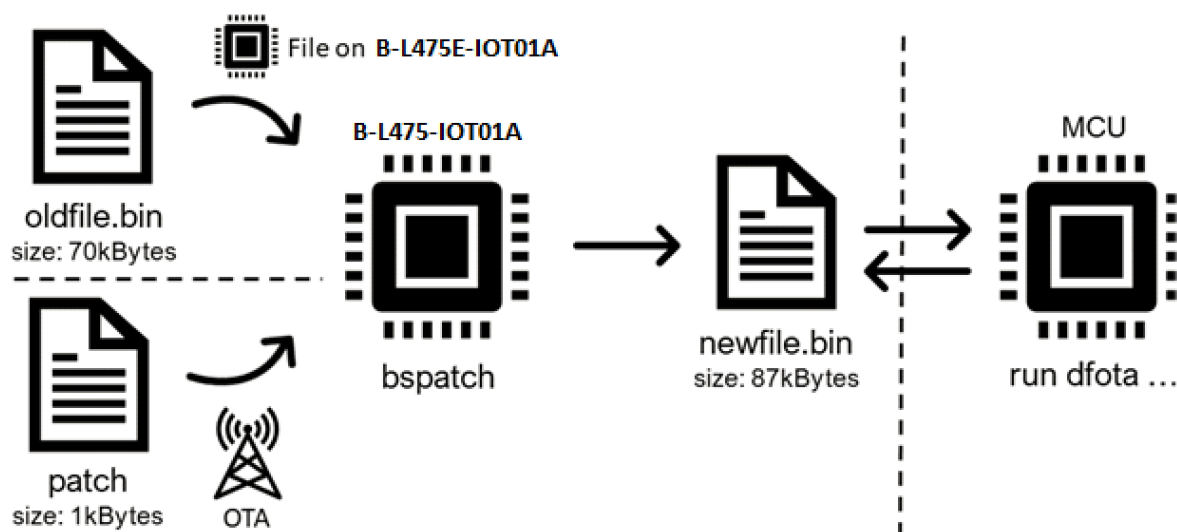


Figure 5. Representation of data differencing and patch generation

### 3.1. BSDiff

BSDiff (named after binary differential) is a command-line utility and library for creating and applying binary patches. The purpose of this tool is to calculate the variances between two binary files and generate a patch file that can be utilized to modify the initial file into a new binary format. This allows for time-efficient transmission and storage efficiency in firmware updates without the need to send the entire new firmware.

The algorithm uses a technique called suffix sorting, specifically qsufsort [51], to compare the two files and identify the matching blocks. By comparing the matching blocks on a byte-by-byte basis, the algorithm can determine which parts of the two files have changed, resulting in a highly compressible string of differences, as most bytes will be identical and the differences that do exist will often take on a small number of values. This allows BSDiff to achieve a high compression ratio and efficient storage of the patch file.

Suffix sorting [52] is an algorithm that sorts the suffixes of a given input string in lexicographic order. A suffix is a substring of a given string that starts at a certain position and continues to the end of the string. Suffix sorting is used in various applications such as text compression, text search, and pattern matching. The most common use of suffix sorting is the construction of the suffix array, which is an array of integers representing the starting position of each suffix in the sorted order. This array is then used in other algorithms such as the LCP (longest common prefix) array, which allows for efficient pattern matching and text search.

Qsufsort [51] is one of the suffix sorting algorithms, it is based on the quick-sort algorithm [53], and it was proposed by Larsson and Sadakane. It is an efficient, fast, and space-saving algorithm that can be applied to large datasets. We can create a straightforward illustration of this process by considering the string "update" in C programming.

The string is indexed as follows [54]:

<b>i</b>	0	1	2	3	4	5
<b>C[i]</b>	u	p	d	a	t	e

and if given ascending order yield the following suffixes:

<b>i</b>	3	2	5	1	4	0
<b>Suffix</b>	ate	date	e	pdate	te	update

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

$i$	0	1	2	3	4	5
$A[i]$	3	2	5	1	4	0

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 BSDiff algorithm works by comparing two binary files and identifying the differences between them. The regions of the files that only differ by a small amount are saved in a separate file as a byte-wise representation of the differences. This file is highly repetitive and easily compressible. Instructions for copying and inserting the changes are also saved in another file, which also has high redundancy. The non-matching data, which corresponds to the modified code, is saved in a separate 'extra' file. These output files can be compressed using the Burrows-Wheeler transform [55], a technique that rearranges the order of characters in a string to concentrate character recurrences, which leads to higher compression. The combination of these steps makes the BSDiff algorithm highly efficient in creating small patch files.

When using BSDiff for differencing, the amount of time it takes to process the data increases as the size of the old file ( $n$ ) and the size of the new file ( $m$ ) increases. Additionally, the amount of space required also increases proportionally to the size of the files. The exact complexities are  $O((n + m) \log n)$  for time and  $\max(17n, 9n + m) + O(1)$  for space. When applying patches using BSDiff, the time and space complexities are less demanding. The time complexity is  $O(n + m)$  and the space complexity is  $n + m + O(1)$  [56]. This means that the time and space required to apply the patch increases linearly with the size of the files.

### 3.2. DeTools

DeTools [57], [58] is a binary delta encoding utility library for ARM architectures. The supported data formats of the library are ARM Cortex-M4 [59] and AArch64 [60]. The DeTools library consists of multiple differential algorithms, patch types, compressions and can create differential binaries (patches) based on the following algorithms:

- BSDiff
- HdiffPatch [61]
- Match-blocks [62]

Since each of the aforementioned algorithms has a unique nature, the approach used to generate the patch from each algorithm is distinct. For instance, the BSDiff algorithm creates a sequential patch type [63], whereas the HdiffPatch generates an HdiffPatch [63] patch type, and the Match-blocks algorithm generates an in-place (resumable) patch type [63] .

A sequential patch uses two memory regions, one containing the from-data and the to-data is written to the other. The patch is accessed sequentially from the beginning to the end when applying the patch. Further details can be found in section 2.4.

The algorithm implemented in this thesis is the BSDiff differential algorithm, which generates a sequential patch, and is compressed with the heatshrink compression, which is described in the next section.

### 3.3. Heatshrink

Heatshrink [64] is a lossless data compression algorithm that is designed for embedded systems with limited resources. It is a combination of two algorithms: an entropy coder [65] and a context-adaptive dictionary coder [66]. The entropy coder uses a range coder [67], which is a form of arithmetic coder, to compress the input data. The context-adaptive dictionary coder uses a sliding window algorithm to compress repeated patterns in the data.

Heatshrink is designed to be efficient in terms of both memory usage and computational resources, making it well-suited for embedded systems with limited memory or processing power. It is also designed to be easy to implement, making it a good option for embedded systems with limited development resources. Heatshrink is available as an open-source library and can be easily integrated into a wide range of systems.

Heatshrink is similar to other compression algorithms such as LZ77 [68], LZ78 [63], and LZW[69] . However, it differs from these algorithms in that it is more efficient in terms of memory usage and computational resources, making it more suitable for embedded systems.

### 3.4. Differential Binary

Patching, as in creating and applying a patch, is related to data compression and decompression. It involves the application of compressed data and instructions for copying and inserting it. The method used for patching depends on the target system and specific preferences. In this context, we will only examine the sequential and in-place patching.

Sequential patching uses two memory regions, one for the source and one for the target. It also uses patches that are divided into repeating patterns or sequences. Each sequence is composed of the parts: diff, extra, and adjustment, which together make up the instructions for a small section of the target image. It is applied in a loop, repeating the same three steps until the patch is fully applied.

In-place patching requires less memory as it only uses one memory region for both source and target. It works by moving the source image upwards in the memory to make room for the target image. Similarly, to sequential patching, it applies patches incrementally, however, it also requires that the source image can be moved during runtime.

## Chapter 4 Equipment and Specifications

### 4.1. Microcontroller Features

For the implementation of this thesis, the microcontroller used was the B-L475E-IOT01A [70] which features the following specifications:

- On-board ST-LINK/V2-1 debugger/programmer features re-enumeration capability: mass storage, Virtual COM port and debug port.
- Bluetooth, Wi-fi, NFC
- Up to 1 MB Flash, 2 banks read-while-write, proprietary code readout protection. Each bank contains 256 pages of 2 Kbyte.
- 64-Mbit Quad-SPI External Flash memory
- 32-bit, 80 MHz, Arm Cortex-M4 CPU
- Capacitive digital sensor for relative humidity and temperature (HTS221)
- High-performance 3-axis magnetometer (LIS3MDL)
- 3D accelerometer and 3D gyroscope (LSM6DSL)
- 260-1260 hPa absolute digital output barometer (LPS22HB)
- Time-of-Flight and gesture-detection sensor (VL53L0X)

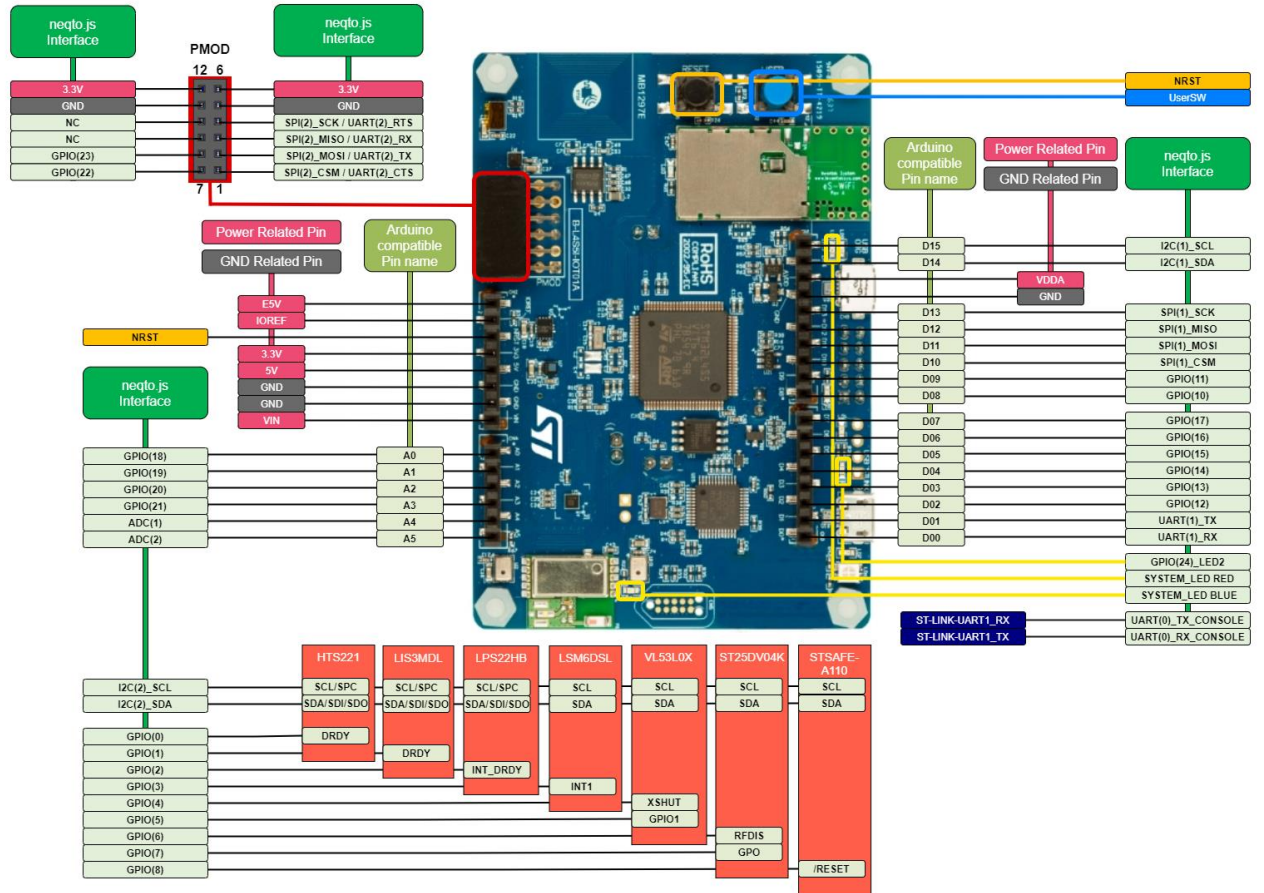
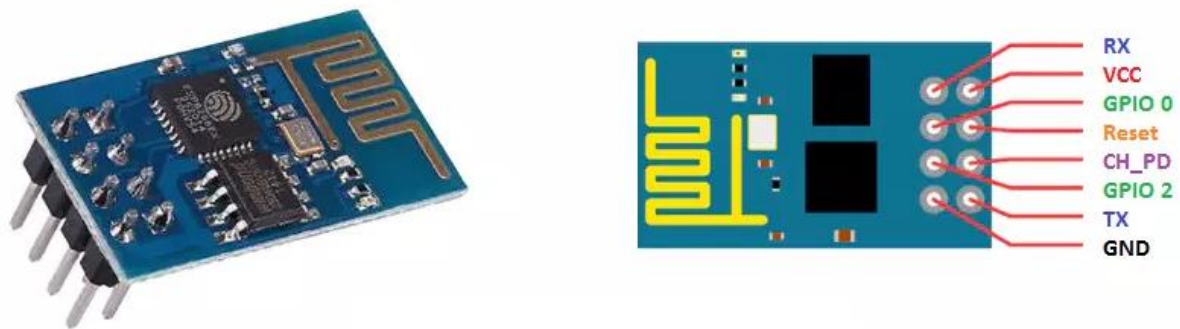


Figure 6. B-L475E-IOT01A

Although the device comes with an integrated WiFi module, Inventek system ISM43362-M3G-L44 [71], it wasn't always responsive with Golioth's SDK libraries and thus in its place, an external WiFi shield module was used.

The shield used in the integrated WiFi module's place was the ESP8266 WiFi Module [72]. ESP8266 is a low-cost WiFi module that allows microcontrollers to connect to a WiFi network and make simple TCP/IP connections. A shield is a piece of hardware that can be connected to a microcontroller board via the, almost universal, ARDUINO® Uno V3 pinout connectors, and give additional functionality to the board



*Figure 7. ESP8266 WiFi module and pinout.*

## 4.2. Flash Memory

The heart of the delta encoding process in the system is the flash memory, although it does have some limitations in terms of low-level access compared to other memory types utilized in patching algorithms. The Non-Volatile Memory Controller (NVMC) manages the flash memory, performing duties such as erasing and writing to the memory. It's important to note that when the NVMC is engaged in these tasks, it can temporarily halt the CPU if it is currently accessing code stored in the flash. Another limitation of flash memory is that the NVMC can only change bits to '0' during writing, and to change a bit to '1', the memory must be erased. Also, flash memory has a finite number of write cycles before a page erased is necessary, called flash wear. This can result in challenges when updating firmware, as even small changes may require erasing and reprogramming of large sections of memory. To address this, an efficient page erasure management system is crucial.

A representation of the STM32L475xx/476xx/486xx series' flash memory layout can be seen on the following figure.



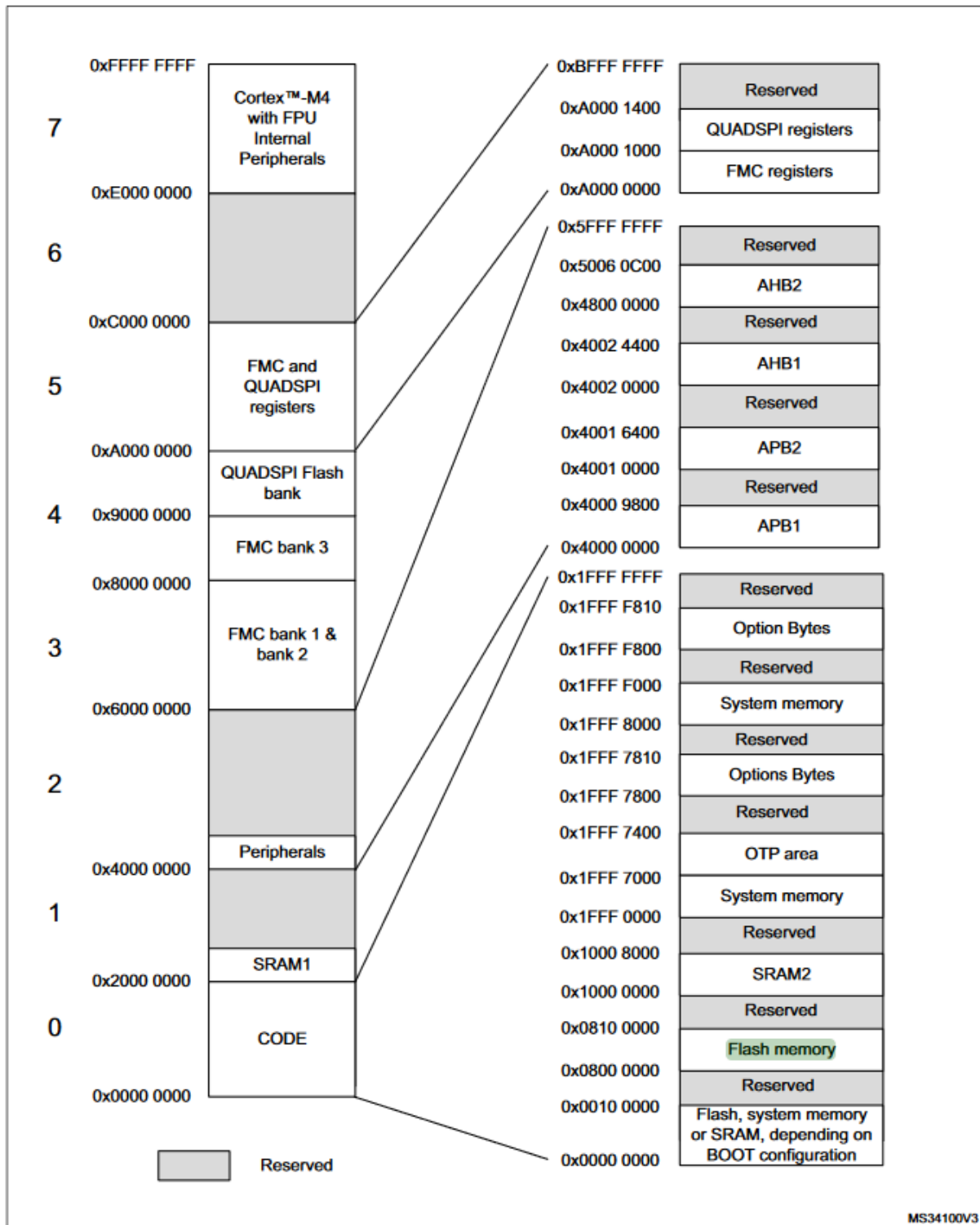


Figure 8. Flash memory map for STM32L475xx/476xx/486xx devices [73, p. 80]

## Chapter 5 Architecture Mechanism and Limitations

The process of porting the application from the nRF52840 Development Kit to the B-L475E-IOT01A involved the creation of a custom mechanism to enable sequential writing on the flash memory, which consists of 256 pages with a size of 2KB per page instead of 4KB. This was the most significant challenge, as it required modifying the code from the original project to accommodate the read and write capabilities of a microcontroller unit with smaller page size. The following paragraphs describe the pattern used to modify the code and achieve this outcome.

The execution of the delta shell command initiates the process of reconstructing the target firmware. The delta function starts by verifying that the patch has the correct header, which is necessary for it to be recognized as a new patch. To do this, it reads the first page of the patch. However, after the header is recognized, the delta function needs to remove it so that it does not overwrite data and cause software malfunctions. To achieve this, the function "flash\_erase" is called, which deletes the page containing the header.

Here, the differences in architecture between the microcontrollers becomes apparent. The original function deletes a page that is twice the size required for the target board B-L475E-IOT01A. To overcome this issue, a temporary buffer was created to store the data before it is deleted from the flash partition offset and then rewritten back after each write, making sure that it is aligned in 8-byte increments. This is crucial because writing unaligned data (not 8-bytes multiples) to the flash results in an error.

The mechanism operates as follows:

Initially, the delta function determines whether the flash-write offset is 8-byte aligned.

- If the alignment is confirmed, it then verifies if the write-size is also aligned with 8 bytes.
  - If the write-size is aligned, the data is then accurately written.
  - Otherwise, if the write-size offset is not aligned, a calculation is performed to determine the required padding size. This padding, along with the write-data, is written to fill an entire page. This allows the subsequent block of data to be written starting from the beginning of the next page.
- Otherwise, the flash-write offset is unaligned, then, the page offset where the data must be written is calculated.
  - A single page is retrieved from the calculated page offset and transferred to the temporary buffer during each iteration.

- Next, the location within the temporary buffer where the data needs to be written is determined by computing the byte-offset.
- Following this, the updated data is copied into the temporary buffer, beginning at the determined byte-offset.
- Prior to writing the data to the flash memory, two pages are erased from the calculated page offset to ensure that the data can be correctly converted within a maximum size of two pages.
- Finally, the data in the temporary buffer is committed to the flash memory.

This mechanism is repeated for every byte that needs to be written in the slot that will host the reconstructed target firmware.

If this application needs to be ported to another board, then the same steps must be followed, changing the values of the flash page size for the target board.

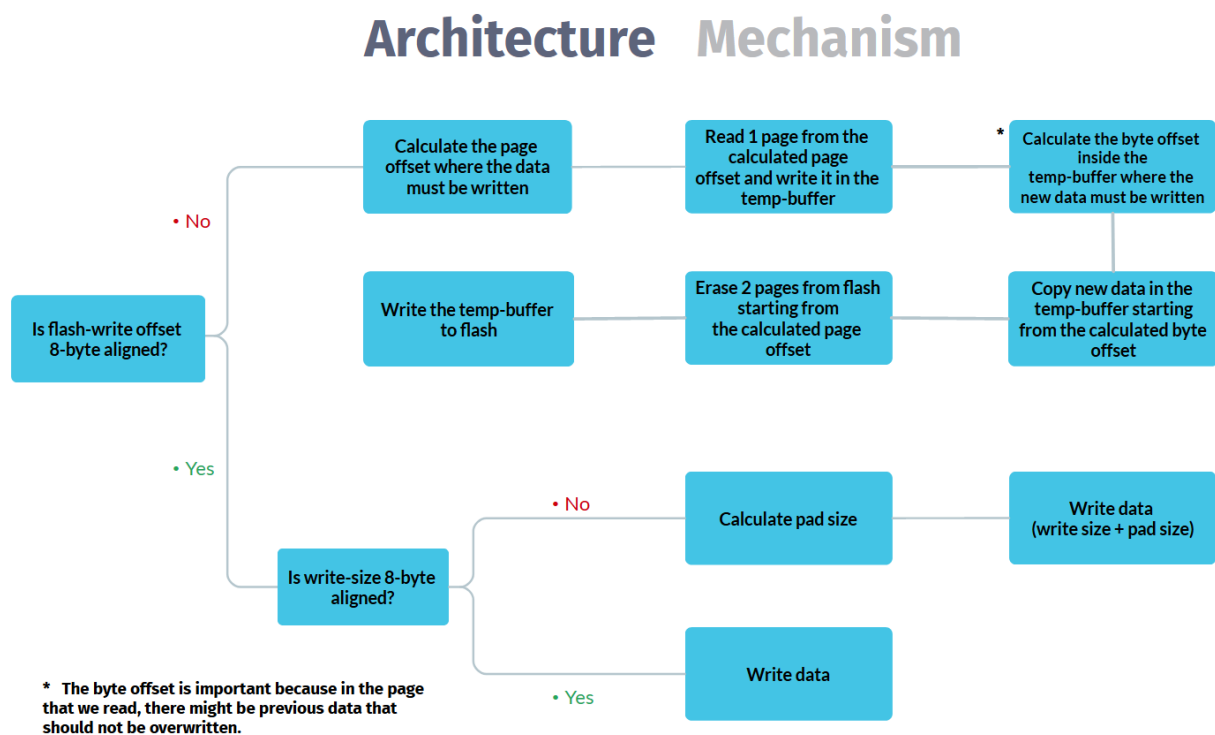


Figure 9. Custom architecture mechanism

## 5.1. Limitations

The limitations that were noted and documented during this thesis concern the application's modularity on different boards and architectures, the size of the patch partition, along with the maximum patch size for successful delta updates.

The first limitation regards the modularity of the application. The original project was developed for the architecture of the memory layout of the nRF52840 Development Kit [9], as stated in section 1.2, which consists of 256 pages of 4KB in size for each page. In order to port the application to the B-L475E-IOT01A, a custom mechanism had to be developed that allowed the sequential writing on the flash memory on also 256 pages, but of the size of 2KB each page. As a result, the code had to be dissected and reconstructed in order to be tailored for the needs of this thesis.

If the same application were to be used for a board of similar architecture but different flash memory layout, the mechanism needs to be altered accordingly. Further work is suggested in Chapter 10, which discusses future plans.

The second limitation regards the patch partition (named storage partition, as per the ZephyrOS's default device tree for the B-L475E-IOT01A [36]). This limitation rises due to the fact that if a different partition is created, in the device tree, so that the patch is downloaded there, that partition sets the maximum size of the patch that can be downloaded in the device. The alternative approach to this limitation would be to store the patch in the second partition (target firmware partition), at the expense of losing the the firmware rollback, or downgrade, feature.

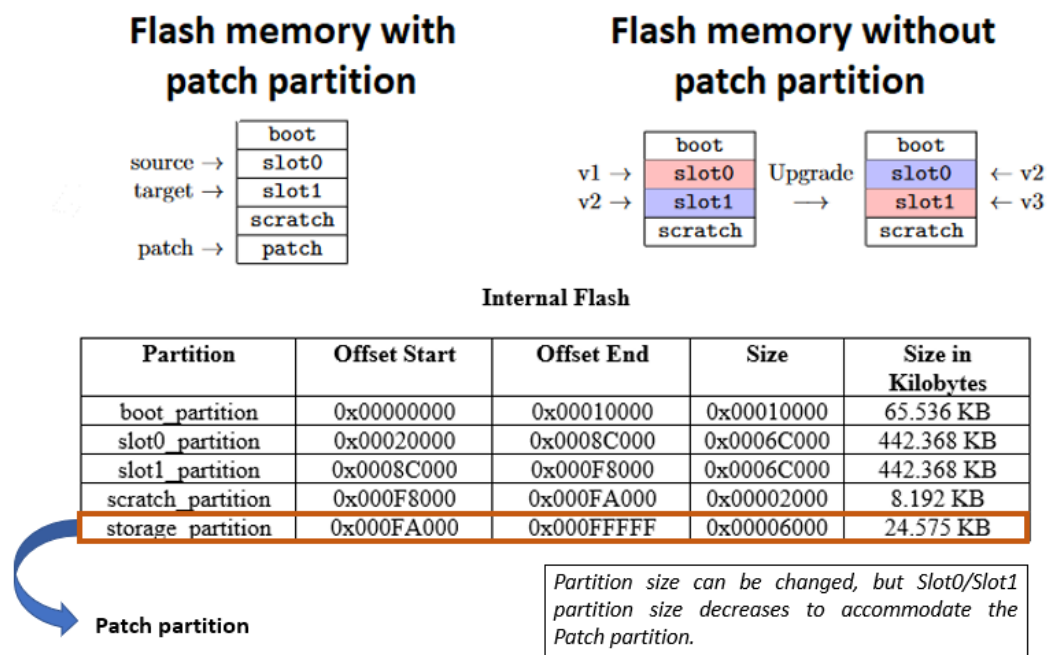


Figure 10. Different approaches for the patch storage

The third and most important limitation that was discovered while developing this application, is that the patch cannot exceed the size of 4 KB. The reason for this obstacle has not been discovered and is to be resolved in future work.

## Chapter 6 Application Analysis

The following section presents an in-depth explanation of the application developed for the delta update use-case, along with visual representations of the results from each stage. The steps described in the upcoming sections are arranged in the following sequence to showcase the application's demonstration:

### *Application Summary*

- *Details on the developed application*

### *Bootloader*

- Building the bootloader.
- Flashing the bootloader to the target board.

### *Source Firmware*

- Building the source firmware (image).
- Signing the image.
- Flashing the image to the target board.

### *Target Firmware*

- Building the target firmware.
- Signing the new image.

### *Patch*

- Creating the patch.
- Padding the header of the patch.

### *Firmware Management*

- Uploading the patch onto the database.
- Downloading the patch on the target board.

### *Delta Update*

- Initiating Delta Update.

### *New Firmware*

- Target firmware reconstruction inside the MCU.

### *Upgrade*

- Upgrading to the new firmware.

## 6.1. Application Summary

The application that was created for the purpose of this thesis outputs live feedback via serial communication, from the Time-of-Flight and gesture-detection sensor, VL53L0X [74], along with the firmware version number. The VL53L0X sensor is a laser-ranging module that provides accurate distance measurements, which can be seen via serial output with a very small refresh interval rate. The version number can also be seen with the *'mcuboot'* shell command but is added alongside the sensor output for easy visual confirmation.

The application has an established communication with IoT development platform Golioth, which controls the firmware update, version, and streaming of data, if there is such need. There are numerous alternatives to Golioth, such as Eclipse's Leshan and Hawkbit which are both open-source and containerized, as opposed to Golioth, which is proprietary and cloud-based. At the time, Golioth was the most viable option, though, since it provided good documentation, extensive driver support, and also the developers were eager to help with hands-on assistance.

The delta update, that is initiated via a custom shell command, upon completion makes a very simple change in the code, such as a couple of words that are printed on the serial output, which in this case are the version of the firmware that is manually inserted for visual aid and demonstration purposes.

The original idea was to alter the Time-of-Flight sensor with another (e.g., HTS221 – temperature and humidity sensor [75]) and print the data in the same manner as before, so as to demonstrate a greater and better scenario. Unfortunately, after the code was developed and tested, the problem of the patch size came to light, and thus compromises had to be made.

## 6.2. Bootloader

The first step to of the application is to build the bootloader and flash it onto the target board. Upon completion, the bootloader shows serial output feedback and waits for a signed firmware image to be flashed.

A bootloader is a program that runs when a device starts up and is responsible for loading the operating system or other system software. The bootloader header is a portion of the bootloader that contains information about the bootloader itself, such as its size and location in memory. The header may also contain information about the system, such as the type of processor or memory configuration. The header sequence refers to the order in which the bootloader header information is stored and read by the device. This sequence is often determined by the device's hardware and can vary between different types of devices [76]. In the image below, the header, that is part of the firmware image, can be seen in the first 16 bytes of the binary.

OFFSET	1.0.0.bin
0x00000000	3D B8 F3 96 00 00 00 00 00 00 02 00 00 E4 CE 02 00   =.....
0x00000010	00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00   .....
0x00000020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
0x00000030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
0x00000040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
0x00000050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
0x00000060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
0x00000070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
0x00000080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
0x00000090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
0x000000A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
0x000000B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
0x000000C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
0x000000D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....
0x000000E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .....

Figure 11. Hex of the source binary, showing the header of the bootloader.

The following commands were used to build and flash the bootloader:

```
west build -s boot/zephyr -b disco_l475_iot1 -p
west flash
```

### 6.3. Source Firmware

The source firmware is the first version of the application. This firmware is built with the function of serial output of the data collected from the VL53L0X sensor every 5 seconds and indicates the sum of the readings on the bottom right corner, as seen in *figure 12*.

After the firmware building is complete, the image needs to be signed. The process of signing images in MCUboot, as in most bootloaders, is important for ensuring the integrity and authenticity of the firmware updates that are installed on the target device. By signing the images, it is possible to verify that the firmware update has not been tampered with or altered in any way during transmission from the update server to the target device.

The signing process uses a cryptographic signature, which is generated using a private key. The signature is then included as part of the firmware update image, along with the corresponding public key. When the target device receives the firmware update, it uses the public key to verify the signature and ensure that the image has not been modified. If the signature is valid, the target device can proceed with the installation of the update.



By signing the firmware updates, MCUboot helps prevent compromising the security and stability of the target device. Signing also helps ensure that the firmware updates are delivered in a trustworthy and secure manner, which is critical for systems that require high levels of reliability and security, such as IoT devices and other embedded systems, especially on the edge.

MCUboot itself provides the ability to generate a key using multiple signing algorithms, such as RSA-2048 [34], RSA-3072 [34], ECDSA-P256 [37] and ED25519 [77]. In the case of this application, the algorithm used is the RSA-2048.

The following commands were used to build, sign, and flash the source firmware to the target board:

```
west build -b disco_l475_iot1 Thesis_Delta_Update/ -- -DSHIELD=esp_8266_arduino

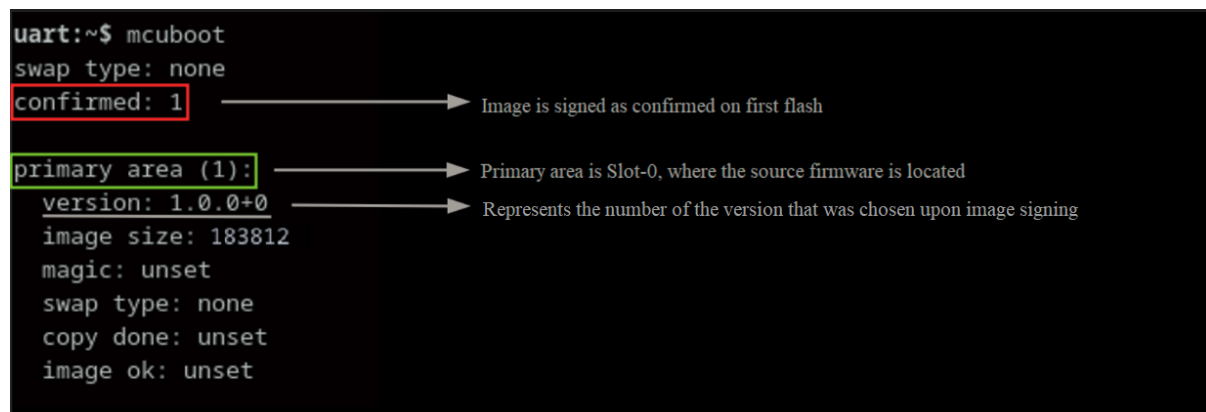
west sign -t imgtool -H 1.0.0.hex -B 1.0.0.bin -- --header-size 512 --slot-size 0x6c000 --align 4 --key ~/zephyrproject/bootloader/mcuboot/root-rsa-2048.pem --version 1.0.0

west flash --bin-file 1.0.0.bin --hex-file 1.0.0.hex
```

```
=====
|                               Time-of-Flight Sensor Firmware: Delta v1.0.0                               |
|-----|
|                               VL53L0X Data                               |
|-----|
| Proximity is:      0      |
|-----|
| Distance is :  8.190m  |
|-----|
|                               Sensor Reading: 3                               |
|-----|
```

*Figure 12. Source application (version 1.0.0) serial output.*

The size of the source firmware is **183812 KB**, as seen in *figure 13* below.



```

uart:~$ mcuboot
swap type: none
confirmed: 1
primary area (1):
version: 1.0.0+0
image size: 183812
magic: unset
swap type: none
copy done: unset
image ok: unset

```

Annotations:

- `confirmed: 1` → Image is signed as confirmed on first flash
- `primary area (1):` → Primary area is Slot-0, where the source firmware is located
- `version: 1.0.0+0` → Represents the number of the version that was chosen upon image signing

Figure 13. Bootloader information of the source firmware.

## 6.4. Target Firmware

The updated version of the application is referred to as target firmware. After the board is flashed with the bootloader and the source firmware, the serial output of the sensor's data and the firmware version number are displayed.

For this demonstration, a slight modification is made to the application, resulting in a different target firmware from the source. This modification involves changing a few characters in the serial output, specifically the version number, which is updated from '1.0.0' to '1.0.1'. As a result, the target firmware is of similar size to the source firmware, with only a few dozen or hundreds of bytes difference.

To ensure the security of the update, the target firmware is signed using the same algorithm. This allows the bootloader to recognize the new firmware as a secure update from a trusted source.

The following commands were used to build and sign the target firmware:

```

west build -b disco_l475_iot1 Thesis_Delta_Update/ -- -DSHIELD=esp_8266_arduino

west sign -t imgtool -H 1.0.1.hex -B 1.0.1.bin -- --header-size 512 --slot-size 0x6c000 --align 4 --key ~/zephyrproject/bootloader/mcuboot/root-rsa-2048.pem --version 1.0.1

```

The target firmware is **183800 KB** in size, which is quite similar to the source firmware since the changes were minor. Unlike the source firmware, we do not flash the target firmware since a patch will be created next.

## 6.5. Patch

The patch can be created as both the source and target firmware are already accessible and prepared. To create the patch, the Detools delta encoding tool will be utilized. Detools offers various compression algorithms like BZ2 [78], LZ4 [79], LZMA [80], Zstandard [81], heatshrink [64], and (C)RLE [82] for generating patches. In this demonstration, heatshrink compression is selected as it has been specifically designed for embedded solutions and a basic implementation for the Zephyr RTOS on a different board and architecture has already been ported.

```
[harrkout@rth]~/zephyr-sdk-0.15.2/zephyr
$ detools patch_info patch.bin
Type: sequential
Patch size: 4.02 KiB
To size: 180.32 KiB
Patch/to ratio: 2.2 % (lower is better)
Diff/extra ratio: 72028.1 % (higher is better)
Size/data ratio: 0.0 % (lower is better)
Compression: heatshrink (window-sz2: 8, lookahead-sz2: 7)
Data format size: 0 bytes

Number of diffs: 3
Total diff size: 180.07 KiB
Average diff size: 60.02 KiB
Median diff size: 27.92 KiB

Number of extras: 3
Total extra size: 256 bytes
Average extra size: 85 bytes
Median extra size: 0 bytes
```

*Figure 14. Patch information*

The following commands were used to generate the patch with the DeTools library:

```
e.g., detools create_patch --compression [compression type] [source binary] [target
binary] [output patch binary]

detools create_patch --compression heatshrink 1.0.0.bin 1.0.1.bin patch.bin
```

As depicted in *figure 14*, the patch's size is **4.02 KB**, which is significantly smaller compared to the entire firmware update, accounting for only 2.23% of the total firmware size.

## 6.6. Header Padding

Once the patch has been successfully generated, the final step is to add padding to the binary's header so that it can be identified by the application's delta function. This padding includes metadata at the start of the patch, known as the header. The metadata is a 16-byte string consisting of a message and the size of the patch.

The message, "**NEWP**", is placed in the first 8-bytes and serves as a recognition marker for the delta function. The last 8-bytes hold the size of the patch, represented as a padded hexadecimal number string, such as "NEWP0x000008c4".

Please note that once the delta function is executed, the header will be replaced with null values and will not be available for later use. The padding process is carried out manually using a Python script.

OFFSET	patch.bin
0x00000000	4E 45 57 50 A0 0F 00 00 04 B0 CC 1C 44 80 6C 39  NEWP.....D.19
0x00000010	11 C0 C1 58 0A 67 F8 03 F8 03 F8 03 F8 03 F8 03  ...X.g.....
0x00000020	F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03  .....
0x00000030	F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03  .....
0x00000040	F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03  .....
0x00000050	F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03  .....
0x00000060	F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03  .....
0x00000070	F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03  .....
0x00000080	F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03  .....
0x00000090	F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03 F8 03  .....

Figure 15. Detailed information of the patch binary, showing the padded header.

## 6.7. Firmware Management

For the firmware management, the platform selected is Golioth. Upon booting the device and starting the application, communication is established between the board and Golioth's cloud-based platform. Golioth offers a multitude of features, including firmware management (FOTA), data streaming, security enhancements, and massive firmware updates over a fleet of devices.

However, for this demonstration, the use of the storage partition was changed to download the patch. After the delta function is executed, the target firmware is reconstructed in Slot-1, effectively using Slot-1 to store a backup firmware in case of any malfunctions. This is because MCUboot, after an update is performed, stores the previous firmware as a backup in case a rollback is necessary. It is possible to disable this mechanism and gain more storage space in

either Slot-0 or Slot-1, but with the current state of technology, this defense mechanism is deemed important and worth the small cost.

Software (4/5)

## Golioth IoT Platform



- **Golioth** can be connected to numerous embedded devices and manage their Firmware, monitor data and export them in WebHooks if needed.
- Multiple devices can be controlled at the same time for uses such as **Massive Firmware Update**.

**Device Hardware ID**

- Contains also User-Created Credentials.

**Device Status**

**Data Log and Streaming**

**Firmware Version**

- Complete Management from GUI and CLI

Figure 16. Golioth Dashboard 1/2

Software (5/5)

## Golioth IoT Platform



- Firmware **Releases** can be created independently from Artifacts
- **Rollout** allows firmware to exist in the database, but not be released until needed
- If a Release Rollout is disabled, the Firmware automatically **Downgrades** to exact previously available version

**Release Version**

- Can control when to deploy via **Rollout**

**Artifact**

- Contains the firmware binary

**Rollout**

- Manages whether the new Firmware is deployed.

Figure 17. Golioth Dashboard 2/2

## 6.8. Delta Update

After being downloaded from Golioth, the patch is stored in the Storage Partition, ready to be used to update the Target firmware. When the "delta" command is entered in the serial output, the algorithm starts by reading the header of the patch. If the header has the correct metadata, the delta update continues by copying data from the source firmware in Slot-0 and applying it to Slot-1, with the patch directing which code blocks need to be changed.

Once the delta function is completed, the application displays the message "Delta OK", and the header metadata is deleted so that the patch cannot be reused and potentially overwrite important code data.

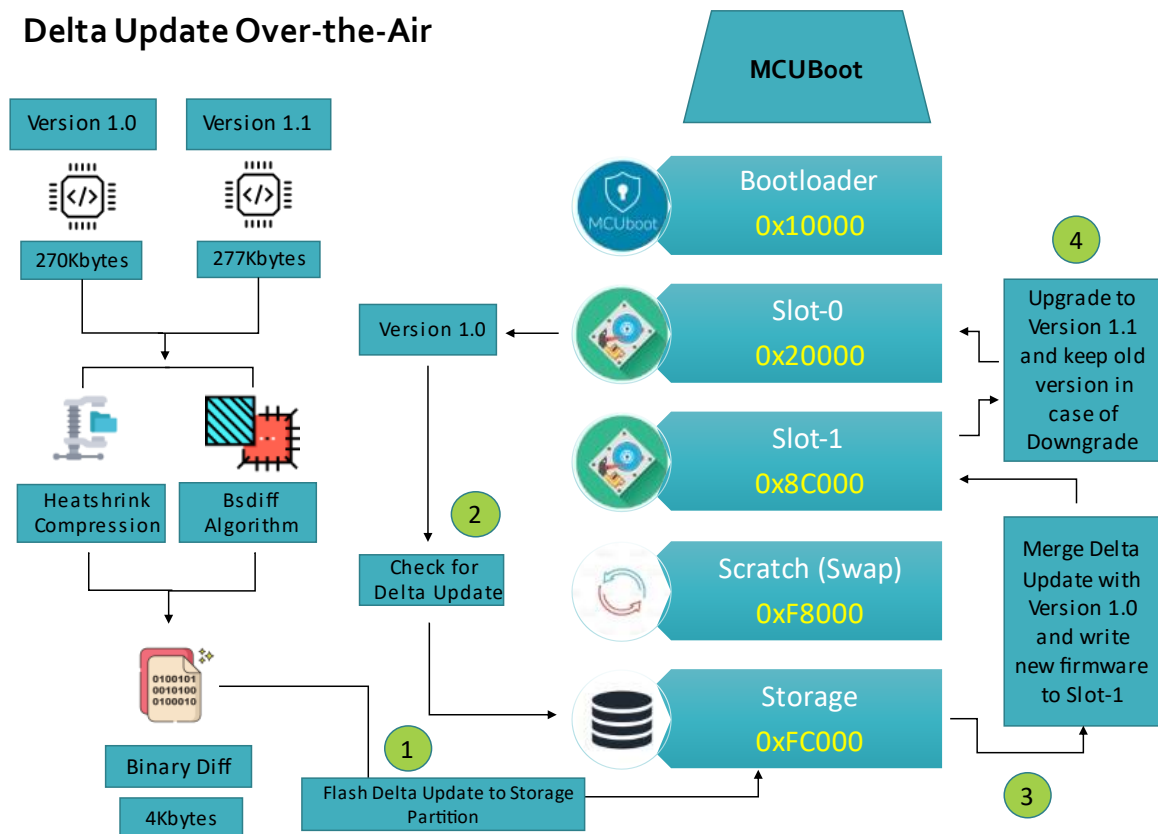


Figure 18. Flow of delta update

## 6.9. New Firmware

After the delta update is completed, MCUboot recognizes two types of firmware in its storage: the source (version 1.0.0) and the target (1.0.1), which can be verified by using the shell commands provided by MCUboot, as shown in *figure 19* below. To update the firmware, the board can either be rebooted, which will automatically upgrade to the newer firmware version, or the upgrade can be initiated manually through the shell command 'mcuboot request\_upgrade'.

This manual upgrade feature was originally turned off for debugging purposes but was retained because it provides better control over the firmware update process during the phases of development and experimentation.

```

uart:~$ mcuboot
swap type: none
confirmed: 0

primary area (1):
  version: 1.0.0+0
  image size: 183812
  magic: unset
  swap type: none
  copy done: unset
  image ok: unset

secondary area (2):
  version: 1.0.1+0
  image size: 183800
  magic: unset
  swap type: none
  copy done: unset
  image ok: unset
  
```

Primary area represents Slot-0, in which the source firmware is located

Secondary area represents Slot-1, in which the target firmware is located

*Figure 19. Source and target firmware, recognized by the MCUboot bootloader.*

## 6.10. Upgrade

Once the board has rebooted, the bootloader will take advantage of the Scratch partition to swap data between Slot-0 and Slot-1 so that the target firmware will take charge as the leading firmware, keeping the former one in Slot-1 as a rollback option in case of a malfunction, which

can be either executed manually if needed, or automatically if the application is not running as expected.

[illegible]

Figure 20. Version 1.0.1 of the application



## Chapter 7 Fault-tolerance and Automation

In the context of delta updates, fault tolerance refers to the system's capability to persist or recover in the occurrence of an error or breakdown. This encompasses techniques like error detection and correction, transmission of lost or damaged data, and the possibility to revert to a prior version of the software in case the update process fails.

For instance, some delta update systems use checksums or other forms of error detection to confirm that the update data hasn't been impacted during transmission. Others may have the option to pause and resume the update if the device loses its connection. These methods guarantee that the update process will conclude successfully, and the software remains operational even if an error occurs. Though delta updates are not immune to faults, these mainly occur when the delta function is not appropriately tailored to the target board's architecture. Except for the previously mentioned case, there are few documented faults in the update process of an unprecedented nature.

One example that was documented during this thesis regards the case of an error in the patch header. If, for any reason, the patch was lacking the padding header then the error shown in the following figure would occur. This error indication, among others in the DeTools library source code, exists to provide information of possible malfunctions during the delta update. The loading bar was added manually for visual aid since a timestamp was not available at the time to provide execution time measurements.



*Figure 21. Short patch header error*

### 7.1. Automation

Regarding automation, delta updates can be fully integrated in an application designed for a real-time operating system, allowing IoT management services such as Golioth, or alternatives, as tested and described in a previous section, to manage mass firmware updates or downgrades in numerous devices at once with extreme ease. Devices can be compartmentalized with the use of filters or flags in the IoT management services' dashboard and can be used to select devices that are running the same firmware to be updated to a newer version or downgraded to an older one if needed.

Such an operation makes incremental updates a feature of extreme importance, since it reduces the time a resource-constrained device needs to be active in order to download the firmware.

## Chapter 8 Summary and Future work

In summary, this thesis details the implementation of Over-the-Air updates, specifically incremental updates, on the Zephyr Real-Time Operating System, utilizing the MCUboot bootloader and the Golioth IoT development platform.

The microcontroller used was the ST B-L475E-IOT01A from the ultra-low-power STM32L4 MCU series, which are based on the Arm Cortex-M4 processor. The developed application provided serial output through UART, which featured readings from the VL53L0X Time-of-Flight and gesture-detection sensor along with the firmware version number. Delta updates were implemented using the BSDiff algorithm to generate a patch containing the differences between a source and target firmware, which was then compressed using the heatshrink compression. This allowed the microcontroller to reconstruct the target firmware by combining the source firmware and the patch data, thus saving time and resources since the patch size was significantly smaller to receive than a whole firmware image.

The results of the delta function were highly encouraging, with the algorithm completing in only a few seconds and successfully reconstructing the target firmware from the patch data and source firmware. The delta update process proved to be a more efficient alternative compared to downloading an entire firmware image, which could take approximately ~45 seconds for a firmware of median size of 180 KB. A patch of median size of 4 KB, could be downloaded in just a few seconds. Although, while a patch of such small size may not reflect a real-world scenario, the time difference between downloading a patch and a complete firmware remains significant.

### 8.1. Future work

As a result of limited time, various adaptations, tests, and experiments have been postponed for further investigation. Conducting experiments with real data can be exceedingly time-intensive, often requiring multiple days to complete a single trial. Upcoming pursuits will focus on more comprehensive evaluations of particular mechanisms, exploring novel proposals to assess alternative approaches, or satisfying pure curiosity.

In the upcoming work agenda, the primary objective is to resolve the limitation of the patch size that can be written in flash memory. Currently, the maximum size is 4 KB, which restricts the seamless implementation of incremental updates with a target of updating large sections of firmware via patching the original source firmware.

Additionally, another crucial goal is to enhance the delta algorithm so that it can be universally adopted across microcontroller devices with similar architecture, regardless of the flash memory's pagination and size.

## Bibliography

- [1] N. Mthethwa, P. Tarwireyi, A. Abu-Mahfouz, and M. Adigun, “Secure Firmware Updates in the Internet of Things: A survey,” Nov. 2019, pp. 1–7. doi: 10.1109/IMITEC45504.2019.9015845.
- [2] “LoRa,” *Wikipedia*. Jan. 30, 2023. Accessed: Feb. 08, 2023. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=LoRa&oldid=1136468709>
- [3] “Cellular network,” *Wikipedia*. Jan. 24, 2023. Accessed: Feb. 08, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Cellular\\_network&oldid=1135381812](https://en.wikipedia.org/w/index.php?title=Cellular_network&oldid=1135381812)
- [4] “Bluetooth Low Energy,” *Wikipedia*. Jan. 27, 2023. Accessed: Feb. 08, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Bluetooth\\_Low\\_Energy&oldid=1135882552](https://en.wikipedia.org/w/index.php?title=Bluetooth_Low_Energy&oldid=1135882552)
- [5] “Delta update,” *Wikipedia*. Sep. 14, 2022. Accessed: Feb. 08, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Delta\\_update&oldid=1110285871](https://en.wikipedia.org/w/index.php?title=Delta_update&oldid=1110285871)
- [6] L. Lindh, “Delta Updates for Embedded Systems,” 2021, Accessed: Feb. 08, 2023. [Online]. Available: <https://hdl.handle.net/20.500.12380/302598>
- [7] “Eclipse Leshan.” <https://www.eclipse.org/leshan/> (accessed Feb. 08, 2023).
- [8] T. E. hawkBit Project, “Eclipse hawkBit.” <https://www.eclipse.org/hawkbite/> (accessed Feb. 08, 2023).
- [9] “nRF52840 DK.” <https://www.nordicsemi.com/Products/Development-hardware/nRF52840-DK> (accessed Feb. 08, 2023).
- [10] “Zigbee,” *Wikipedia*. Jan. 24, 2023. Accessed: Feb. 08, 2023. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=Zigbee&oldid=1135335140>
- [11] “CoAP Protocol: What is Meaning, Architecture and Function.” <https://www.wallarm.com/what/coap-protocol-definition> (accessed Feb. 08, 2023).
- [12] “What is LwM2M? Lightweight M2M Protocol Overview.” <https://www.avsystem.com/blog/lightweight-m2m-lwm2m-overview/> (accessed Feb. 08, 2023).
- [13] A. Augustin, J. Yi, T. Clausen, and W. M. Townsley, “A Study of LoRa: Long Range & Low Power Networks for the Internet of Things,” *Sensors*, vol. 16, no. 9, Art. no. 9, Sep. 2016, doi: 10.3390/s16091466.
- [14] D. Mbakoyiannis, O. Tomoutzoglou, and G. Kornaros, “Secure over-the-air firmware updating for automotive electronic control units,” in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, New York, NY, USA, Apr. 2019, pp. 174–181. doi: 10.1145/3297280.3297299.

- [15] G. Kornaros *et al.*, “Towards holistic secure networking in connected vehicles through securing CAN-bus communication and firmware-over-the-air updating,” *J. Syst. Archit.*, vol. 109, p. 101761, Oct. 2020, doi: 10.1016/j.sysarc.2020.101761.
- [16] D. Bakoyiannis, O. Tomoutzoglou, G. Kornaros, and M. Coppola, “From Hardware-Software Contracts to Industrial IoT-Cloud Block-chains for Security,” in *2021 Smart Systems Integration (SSI)*, Apr. 2021, pp. 1–4. doi: 10.1109/SSI52265.2021.9467030.
- [17] Y. He *et al.*, “RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices”.
- [18] C. von Platen and J. Eker, “Feedback linking: optimizing object code layout for updates,” in *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, New York, NY, USA, Jun. 2006, pp. 2–11. doi: 10.1145/1134650.1134653.
- [19] C. Campell, J. Graber, R. Tashakkori, and W. O’Brien, “IoT Apiary Fleet Management with Jenkins,” in *2022 IEEE 13th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, Oct. 2022, pp. 0328–0335. doi: 10.1109/UEMCON54665.2022.9965635.
- [20] “Jenkins,” *Jenkins*. <https://www.jenkins.io/> (accessed Feb. 22, 2023).
- [21] “Open Mobile Alliance,” *Wikipedia*. May 23, 2022. Accessed: Feb. 08, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Open\\_Mobile\\_Alliance&oldid=1089396174](https://en.wikipedia.org/w/index.php?title=Open_Mobile_Alliance&oldid=1089396174)
- [22] “Zephyr Project,” *Zephyr Project*. <https://zephyrproject.org/> (accessed Feb. 08, 2023).
- [23] “Why RTOS and What is RTOS?,” *FreeRTOS*. <https://www.freertos.org/about-RTOS.html> (accessed Feb. 08, 2023).
- [24] “Thread protocol — Zephyr Project Documentation.” <https://docs.zephyrproject.org/latest/connectivity/networking/api/thread.html> (accessed Feb. 08, 2023).
- [25] “MQTT — Zephyr Project Documentation.” <https://docs.zephyrproject.org/1.14.0/reference/networking/mqtt.html> (accessed Feb. 08, 2023).
- [26] “Edge computing,” *Wikipedia*. Feb. 01, 2023. Accessed: Feb. 08, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Edge\\_computing&oldid=1136788470](https://en.wikipedia.org/w/index.php?title=Edge_computing&oldid=1136788470)
- [27] “Linux Foundation - Decentralized innovation, built with trust.” <https://www.linuxfoundation.org> (accessed Feb. 08, 2023).
- [28] “Zephyr Project | Architecture,” *Zephyr Project*. <https://www.zephyrproject.org/learn-about/architecture/> (accessed Feb. 08, 2023).
- [29] “Supported Boards — Zephyr Project Documentation.” <https://docs.zephyrproject.org/3.2.0/boards/index.html> (accessed Feb. 08, 2023).
- [30] “MCUboot - Secure boot for 32-bit Microcontrollers,” *Linaro*. <https://www.mcuboot.com/> (accessed Feb. 08, 2023).

- [31] “mcu-tools/mcuboot: Secure boot for 32-bit Microcontrollers!”  
<https://github.com/mcu-tools/mcuboot> (accessed Feb. 08, 2023).
- [32] “xilinx.github.io/Vitis\_Libraries/security/2020.1/guide\_L1/internals/ctr.html.”  
Accessed: Feb. 08, 2023. [Online]. Available:  
[https://xilinx.github.io/Vitis\\_Libraries/security/2020.1/guide\\_L1/internals/ctr.html](https://xilinx.github.io/Vitis_Libraries/security/2020.1/guide_L1/internals/ctr.html)
- [33] “Bootloader — MCUboot 1.9.99 documentation.”  
[https://developer.nordicsemi.com/nRF\\_Connect\\_SDK/doc/latest/mcuboot/design.html#image-trailer](https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/mcuboot/design.html#image-trailer) (accessed Feb. 08, 2023).
- [34] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978, doi: 10.1145/359340.359342.
- [35] M. Bellare and P. Rogaway, “Optimal asymmetric encryption,” Berlin, Heidelberg, 1995, vol. 950, pp. 92–111. doi: 10.1007/BFb0053428.
- [36] “Key wrap,” *Wikipedia*. Jul. 21, 2022. Accessed: Feb. 08, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Key\\_wrap&oldid=1099476216](https://en.wikipedia.org/w/index.php?title=Key_wrap&oldid=1099476216)
- [37] M. Adalier and A. Teknik, “Efficient and Secure Elliptic Curve Cryptography Implementation of Curve P-256,” 2015. Accessed: Feb. 08, 2023. [Online]. Available: <https://www.semanticscholar.org/paper/Efficient-and-Secure-Elliptic-Curve-Cryptography-of-Adalier-Teknik/62fc2cb6e4874dbd84759e466f286b695098008e>
- [38] D. Bernstein, “Curve25519: New Diffie-Hellman Speed Records,” Apr. 2006, vol. 3958, pp. 207–228. doi: 10.1007/11745853\_14.
- [39] “Building and using MCUboot with Zephyr,” *mcuboot*.  
<https://docs.mcuboot.com/readme-zephyr.html> (accessed Feb. 08, 2023).
- [40] “Macronix - MX25R6435F.” <https://www.mxix.com.tw/en-us/flash-memory-solutions/extended-temperature/Pages/spec.aspx?p=MX25R6435F&m=Ext+Temperature&n=PM2425> (accessed Feb. 08, 2023).
- [41] “zephyrproject-rtos/zephyr.” Zephyr Project, Feb. 08, 2023. Accessed: Feb. 08, 2023. [Online]. Available: [https://github.com/zephyrproject-rtos/zephyr/blob/45d5fb7492e0bd58b95ae3985660a5c7cd0252db/boards/arm/disco\\_l475\\_iot1/disco\\_l475\\_iot1.dts](https://github.com/zephyrproject-rtos/zephyr/blob/45d5fb7492e0bd58b95ae3985660a5c7cd0252db/boards/arm/disco_l475_iot1/disco_l475_iot1.dts)
- [42] “Golioth: the straightforward commercial IoT development platform built for scale,” *Golioth*. <https://golioth.io/> (accessed Feb. 08, 2023).
- [43] “Home · Wiki · xenomai / xenomai · GitLab,” *GitLab*, Jan. 01, 2023.  
<https://source.denx.de/Xenomai/xenomai/-/wikis/home> (accessed Feb. 10, 2023).
- [44] “OpenRTOS,” *FreeRTOS*. <https://www.freertos.org/openrtos.html> (accessed Feb. 10, 2023).

## Bibliography

- [45] “RISC-V Software Ecosystem - Home - RISC-V International.” <https://wiki.riscv.org/display/HOME/RISC-V+Software+Ecosystem> (accessed Feb. 10, 2023).
- [46] “Patch (computing),” *Wikipedia*. Jan. 27, 2023. Accessed: Feb. 10, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Patch\\_\(computing\)&oldid=1135934251#Source\\_code\\_patches](https://en.wikipedia.org/w/index.php?title=Patch_(computing)&oldid=1135934251#Source_code_patches)
- [47] “diff,” *Wikipedia*. Feb. 07, 2023. Accessed: Feb. 20, 2023. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=Diff&oldid=1138026269>
- [48] “Delta encoding,” *Wikipedia*. Jan. 21, 2023. Accessed: Feb. 10, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Delta\\_encoding&oldid=1134939359](https://en.wikipedia.org/w/index.php?title=Delta_encoding&oldid=1134939359)
- [49] “String-searching algorithm,” *Wikipedia*. Feb. 03, 2023. Accessed: Feb. 10, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=String-searching\\_algorithm&oldid=1137152883](https://en.wikipedia.org/w/index.php?title=String-searching_algorithm&oldid=1137152883)
- [50] C. Percival, “Naive Differences of Executable Code,” Aug. 2003.
- [51] Z. Peng, Y. Wang, X. Xue, and J. Wei, “An Efficient Algorithm for Suffix Sorting,” *Int. J. Pattern Recognit. Artif. Intell.*, vol. 30, no. 06, p. 1659018, Jul. 2016, doi: 10.1142/S0218001416590187.
- [52] S. J. Puglisi, W. F. Smyth, and A. Turpin, “The performance of linear time suffix sorting algorithms,” in *Data Compression Conference*, Mar. 2005, pp. 358–367. doi: 10.1109/DCC.2005.87.
- [53] C. A. R. Hoare, “Quicksort,” *Comput. J.*, vol. 5, no. 1, pp. 10–16, Jan. 1962, doi: 10.1093/comjnl/5.1.10.
- [54] “Suffix Arrays.” <http://www.allisons.org/ll/AlgDS/Strings/Suffix/> (accessed Feb. 08, 2023).
- [55] G. Manzini, “The Burrows-Wheeler Transform: Theory and Practice,” Sep. 1999. doi: 10.1007/3-540-48340.
- [56] “Binary diff.” <https://www.daemonology.net/bsdiff/> (accessed Feb. 10, 2023).
- [57] W. Seiler, “DETools: A Library for Differential Equations,” Apr. 1999.
- [58] E. Moqvist, “eerimoq/detools.” Feb. 04, 2023. Accessed: Feb. 08, 2023. [Online]. Available: <https://github.com/eerimoq/detools>
- [59] “Cortex-M4.” <https://developer.arm.com/Processors/Cortex-M4> (accessed Feb. 08, 2023).
- [60] “A64 - ARM - WikiChip.” <https://en.wikichip.org/wiki/arm/a64> (accessed Feb. 08, 2023).
- [61] housisong, “HDiffPatch.” Feb. 07, 2023. Accessed: Feb. 08, 2023. [Online]. Available: <https://github.com/sisong/HDiffPatch>

- [62] “google/diff-match-patch.” Google, Feb. 08, 2023. Accessed: Feb. 08, 2023. [Online]. Available: <https://github.com/google/diff-match-patch>
- [63] E. Moqvist, “dertools Documentation,” pp. 15–18.
- [64] “heatshrink.” Atomic Object, Jan. 19, 2023. Accessed: Feb. 08, 2023. [Online]. Available: <https://github.com/atomicobject/heatshrink>
- [65] “Entropy coding,” *Wikipedia*. Jan. 10, 2023. Accessed: Feb. 10, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Entropy\\_coding&oldid=1132723947](https://en.wikipedia.org/w/index.php?title=Entropy_coding&oldid=1132723947)
- [66] “Context-adaptive binary arithmetic coding,” *Wikipedia*. Aug. 17, 2021. Accessed: Feb. 10, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Context-adaptive\\_binary\\_arithmetic\\_coding&oldid=1039296243](https://en.wikipedia.org/w/index.php?title=Context-adaptive_binary_arithmetic_coding&oldid=1039296243)
- [67] G. N. N. Martin, “Range encoding: an Algorithm for Removing Redundancy from a Digitised Message”.
- [68] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, May 1977, doi: 10.1109/TIT.1977.1055714.
- [69] Welch, “A Technique for High-Performance Data Compression,” *Computer*, vol. 17, no. 6, pp. 8–19, Jun. 1984, doi: 10.1109/MC.1984.1659158.
- [70] “B-L475E-IOT01A - STM32L4 Discovery kit IoT node, low-power wireless, BLE, NFC, SubGHz, Wi-Fi - STMicroelectronics.” <https://www.st.com/en/evaluation-tools/b-l475e-iot01a.html> (accessed Feb. 08, 2023).
- [71] “ISM43362-M3G-L44-E/U Serial-to-WiFi Module – Inventek Systems.” <https://www.inventeksys.com/buy-online/product/ism43362-m3g-l44-eu-embedded-serial-to-wifi-module/> (accessed Feb. 08, 2023).
- [72] “ESP8266 WiFi Module | Sensors & Modules.” <https://www.electronicwings.com/sensors-modules/esp8266-wifi-module> (accessed Feb. 08, 2023).
- [73] “en.DM00172872.pdf.” Accessed: Feb. 12, 2023. [Online]. Available: <https://www.st.com/content/ccc/resource/technical/document/datasheet/90/51/73/d2/53/60/4f/5f/DM00172872.pdf/files/DM00172872.pdf/jcr:content/translations/en.DM00172872.pdf>
- [74] “VL53L0X - Time-of-Flight ranging sensor - STMicroelectronics.” <https://www.st.com/en/imaging-and-photonics-solutions/vl53l0x.html> (accessed Feb. 08, 2023).
- [75] “HTS221 - Capacitive digital sensor for relative humidity and temperature - STMicroelectronics.” <https://www.st.com/en/mems-and-sensors/hts221.html> (accessed Feb. 11, 2023).
- [76] “Image tool,” *mcuboot*. <https://docs.mcuboot.com/imgtool.html> (accessed Feb. 11, 2023).

## Bibliography

- [77] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, “High-speed high-security signatures,” *J. Cryptogr. Eng.*, vol. 2, no. 2, pp. 77–89, Sep. 2012, doi: 10.1007/s13389-012-0027-1.
- [78] “bzip2,” *Wikipedia*. Nov. 08, 2022. Accessed: Feb. 11, 2023. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=Bzip2&oldid=1120739638>
- [79] “LZ4 - Extremely fast compression.” <https://lz4.github.io/lz4/> (accessed Feb. 11, 2023).
- [80] “Lempel–Ziv–Markov chain algorithm,” *Wikipedia*. Feb. 07, 2023. Accessed: Feb. 11, 2023. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Lempel%E2%80%93Ziv%E2%80%93Markov\\_chain\\_algorithm&oldid=1137974981](https://en.wikipedia.org/w/index.php?title=Lempel%E2%80%93Ziv%E2%80%93Markov_chain_algorithm&oldid=1137974981)
- [81] “Zstandard - Real-time data compression algorithm.” <https://facebook.github.io/zstd/> (accessed Feb. 11, 2023).
- [82] A. H. Robinson and C. Cherry, “Results of a prototype television bandwidth compression scheme,” *Proc. IEEE*, vol. 55, no. 3, pp. 356–364, Mar. 1967, doi: 10.1109/PROC.1967.5493.