Aarno Toiviainen

Over-the-air firmware update for mission critical embedded devices

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Bachelor’s Thesis

6 November 2020

Author Title Number of Pages Date	Aarno Toiviainen Over-the-air firmware update for mission critical embedded devices 29 pages 6.11.2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Smart Systems
Instructors	Jussi Heiskanen, Project Manager Sami Sainio, Senior Lecturer

Due to lower cost and advancements in technology embedded devices have become widespread in the recent years. Embedded devices can be used in mission critical applications with requirements for predictable processing time and reliability. At the same time, they can also have requirements on power usage and cost. As a result of these requirements creating software for these devices can be challenging. The firmware on these devices often needs to be updated to fix bugs or add new features. However, due to the large number of these devices its often impractical to update them locally. For that reason remote firmware update systems are needed.

In order to transmit the update data reliably in a firmware update system the data needs to be validated with checksums. Encryption and authentication are also needed to make sure the data comes from a trusted source and hasn't been modified. The system also needs to be able to recover if the device loses power during the update process. This can be done by using multiple memory banks to store different firmware versions.

The purpose of the thesis was to create a working firmware update system that can be used in mission critical systems. The devices used for the system were an nRF52480 microcontroller for the device that was updated and a Raspberry Pi for fetching the firmware package wirelessly. The nRF52480 was used without an operating system and the Raspberry Pi used the Linux-based Raspbian operating system. The nRF52480 and the Raspberry Pi communicated through UART to transfer the update package.

As a result of the thesis a working firmware update system was created. A custom protocol was created to reliably transfer the firmware package through UART. The bootloader from the nRF software development kit was used to verify and install the update. Two memory banks were used for updates to make the system more reliable. Testing was conducted to verify that the system works correctly. The system was able to recover if the update process was interrupted. The system could not recover if a faulty firmware was sent to the system. At the end of the thesis improvements to make the system more reliable are discussed.

Keywords	Internet of Things, IoT, Bootloader, Firmware

Tekijä Otsikko Sivumäärä Aika	Aarno Toiviainen Langaton laiteohjelmistopäivitys missiokriittisille sulautetuille laitteille 29 sivua 6.11.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Älykkäät järjestelmät
Ohjaajat	Projektipäällikkö Jussi Heiskanen Lehtori Sami Sainio

Sulautetut laitteet ovat tulleet yleisiksi matalampien hintojen ja teknologisten kehitysaskelten johdosta. Sulautettuja laitteita voidaan käyttää kriittisissä systeemeissä, joissa on vaatimuksia ennustettavissa olevalle prosessointiajalle ja luotettavuudelle. Niillä voi myös olla samaan aikaan vaatimuksia virrankäytölle ja hinnalle. Tämän johdosta ohjelmiston luominen näille laitteille voi olla hankalaa. Sulautettujen laitteiden ohjelmisto tarvitsee usein päivityksiä virheiden korjaamisen tai uusien ominaisuuksien lisäämistä varten. Laitteiden suuresta määrästä johtuen voi kuitenkin usein olla epäkäytännöllistä päivittää niitä paikallisesti. Siitä syystä on tarve laiteohjelmiston etäpäivitysjärjestelmälle.

Jotta päivitysdata voidaan siirtää luotettavasti laiteohjelmistopäivitysjärjestelmässä, data pitää vahvistaa tarkistussummilla. Salausta ja todennusta tarvitaan myös, jotta voidaan varmistaa, että data tulee luotetusta lähteestä eikä sitä ole muokattu. Systeemin pitää myös pystyä toipumaan, jos laite menettää virrat päivitysprosessin aikana. Tämä on mahdollista kun käytetään useaa muistipankkia eri ohjelmistoversioiden säilyttämiseksi.

Lopputyön tarkoituksena oli luoda toimiva laiteohjelmistopäivitysjärjestelmä, jota voidaan käyttää kriittisissä järjestelmissä. Systeemissä käytettiin nRF52480-mikrokontrolleria päivitettävänä laitteena ja Raspberry Pi:tä ohjelmistopaketin hakemiseen langattomasti. nRF52480-mikrokontrolleria käytettiin ilman käyttöjärjestelmää ja Raspberry Pi käytti Linux-pohjaista Raspbian-käyttöjärjestelmää. nRF52480 ja Raspberry Pi kommunikoivat UART:in kautta päivityspaketin siirtämistä varten.

Lopputyön tuloksena luotiin toimiva laiteohjelmistopäivitysjärjestelmä. Ohjelmistopaketin siirtämiseksi luotettavasti UART:in kautta käytettiin itsetehtyä protokollaa. Päivityksen varmistamiseen ja asentamiseen käytettiin käynnistyslatainta nRF-ohjelmistokehityspaketista. Järjestelmässä käytettiin kahta muistipankkia, jotta se olisi luotettavampi. Systeemiä testattiin, jotta sen voitiin varmistaa toimivan oikein. Systeemi pystyi toipumaan päivitysprosessin keskeyttämisestä. Systeemi ei pystynyt palautumaan, jos sille lähetettiin viallinen ohjelmisto. Lopputyön lopussa kerrotaan parannuksista, joiden avulla systeemistä voisi tulla luotettavampi.

Avainsanat	Esineiden internet, Käynnistyslatain, Laiteohjelmisto

Contents

List of Abbreviations

BLE	Bluetooth Low Energy. A wireless communication technology.
Bootloader	A program that loads other programs when a computer boots.
CoAP	Constrained Application Protocol. An internet application protocol for constrained devices
CRC	Cyclic Redundancy Check. Error-detecting code for detecting changes in data.
DFU	Device Firmware Update. The process of updating the firmware on a device.
DTLS	Datagram Transport Layer Security. A communication protocol that allows secure communication.
Flash memory	A type of non-volatile computer memory.
IoT	Internet of Things. Physical objects with unique identifiers that are connected to the internet.
MBR	Master boot record. A program to allow updating the bootloader and the application on a system.
Microcontroller	A small computer on a single computer chip.
MQTT	Message Queuing Telemetry Transport. A network protocol based on the publish-subscribe paradigm.
OTA	Over-the-air

RTOS	Real Time Operating System. Operating system that fulfills fixed time constraints.
SHA	Secure Hash Algorithm. A set of cryptographic hash functions.
SoC	System on a Chip. A small computer on a single computer chip.
UART	Universal Asynchronous Receiver-transmitter. A computer device for asynchronous serial communication.

1 Introduction

1.1 Overview

Embedded devices have become widespread in the last few decades due to reductions in prices and advancements in technology. One example of this trend is the Intel 4004 which was the first microprocessor and had a 4-bit processor (shown in figure 1.). It was sold for $60 in 1972 [1], whereas today there are microcontrollers with 32-bit processors that cost less than $1. These trends have allowed embedded devices to be used for a wide variety of applications such as cars [2], home appliances [3] and factories [4]. The devices often need to be updated in order to fix bugs, add new features, or improve performance. In the past the devices needed to be updated locally using a programming device. However, the large number of devices makes updating each one manually impractical. Because of this there is a need for a convenient method to update the devices. Over-the-air updates are an efficient way to solve this issue. However, implementing them in a reliable and efficient way can be challenging due to the limitations of embedded devices.

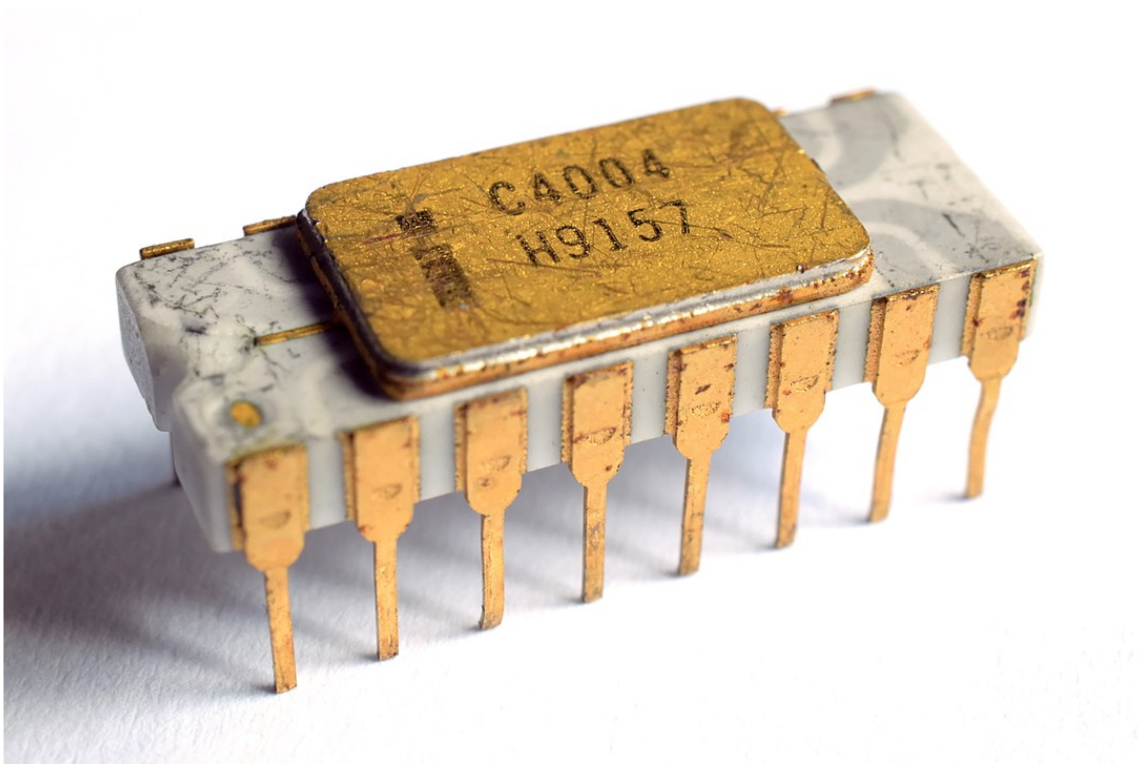


Figure 1: Intel 4004 microprocessor [5]

The aim of this thesis was to create a working prototype system for over-the-air (OTA) firmware updates for the nRF52480 microcontroller. The thesis also discusses design decisions that can be made in a device firmware update (DFU) system.

1.2 Company Background

The thesis was conducted for the company Wizense. Wizense creates industrial Internet of things (IoT) solutions for companies. Their products allow tracking the location of people or vehicles using wireless protocols, which helps improve safety of employees. Updating the tracking devices wirelessly allows bug fixes and new features to be delivered more conveniently. The prototype system developed in this thesis can be used as a basis to create a production ready system for these devices.

1.3 Objectives

The objective of the thesis was to design and implement a working DFU system. The system used the nRF52480 microcontroller and the Raspberry Pi single-board computer. The nRF52480 used the nRF software development kit with no operating system. The Raspberry Pi ran a Linux based operating system. One requirement was that the system should be able to recover if an update fails or the system crashes. An additional objective was to explain how remote firmware updates work, what features they can have and what existing systems there are.

1.4 Thesis Structure

The thesis is divided into five chapters. The first chapter introduces the background of the project and the need for the project. The second chapter describes existing studies and background information on the methods and technologies used. The third chapter describes the design and implementation. The fourth chapter explains the results of the project and discusses the current implementation. The fifth chapter discusses the conclusions from the project while the final chapter discusses possibilities for future development.

2 Theoretical Background

2.1 Prior Literature

As the Internet of things becomes more prevalent the need for reliable and efficient over-the-air firmware update systems increases. Due to the increased need research is being done regarding OTA firmware updates. There are multiple areas of research regarding OTA firmware updates. One area of research is addressing the challenges posed by the limited resources in typical IoT systems. IoT systems can have limitations in terms of power usage, processing power, RAM and flash memory capacity. This creates some challenges for implementing firmware update systems.

Another area of research is delivering the firmware update to devices. Due to the low power available to Internet of things devices their wireless signal may be strong enough to reach the update server. Because of this protocols have been designed to connect the devices as a network where the nodes in the network forward data to each other. Using this approach all of the devices do not have to be directly connected to the update server. The study [6] discusses wireless firmware updates in urban scenarios. A system was implemented for the study using a long-range wireless broadcast. The system uses asymmetric transmission similar to FM radio and digital television. The system is shown in figure 2. The study addresses reliability and security issues related to the transmission.

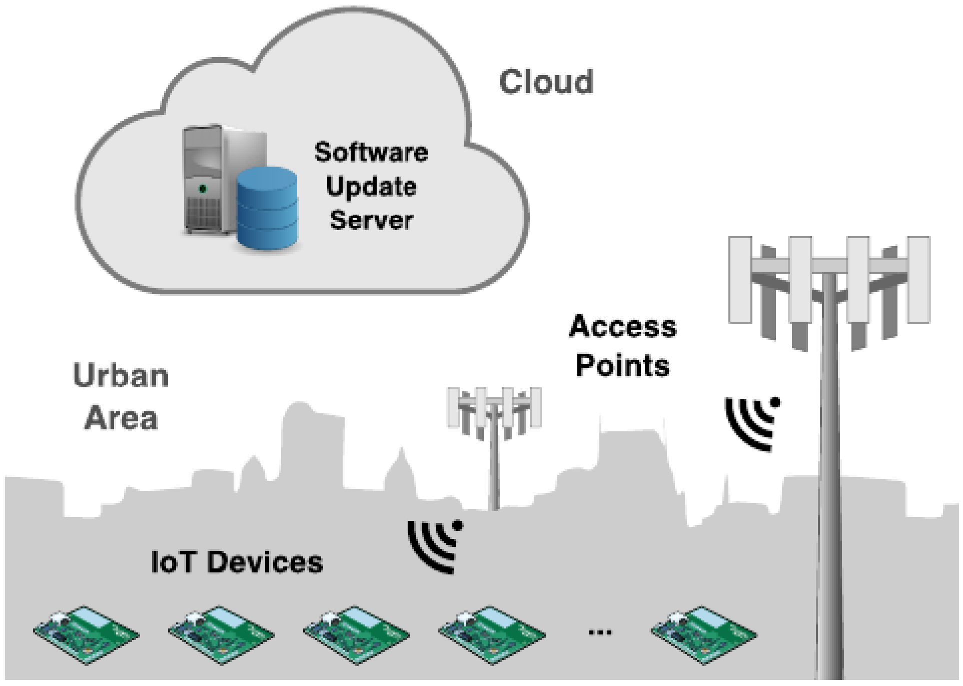


Figure 2: A Scalable Software Update Service for IoT Devices [6, fig. 1.]

Security is also a big concern regarding OTA firmware updates. If the update system is not secure there is a risk of a malicious update being pushed to the system. It is also important that any critical information is not leaked during the update process. The thesis [7] discusses creating an application for an Atmel microcontroller that sends air quality data from a sensor and a DFU system for it. The thesis also takes into consideration security aspects of the update. The thesis talks about encryption, authentication, and secure ways to transfer a firmware update.

Recovering from errors during the update is also important. If the device can't recover from a failed update the device will be left in an unusable state and will have to be reprogrammed manually. The study [8] describes an implementation of a fail-safe OTA programming system. Methods to detect and recover from errors are described in the study. The methods include using a hardware watchdog and beacon messages. The beacon messages are sent by a server and the bootloader checks that they are received. The study evaluates the system for scalability and reliability.

There have been surveys that looked at existing research regarding these areas. A survey [9] looks at research regarding embedded software design, back-end frameworks and network transport. The study provides an overview of how firmware updates on constrained IoT devices work. The study also looks at open standards for firmware updates and cryptographic libraries. A prototype firmware update system that was implemented for the study is described. Multiple versions of the prototype system were implemented using different standards and cryptographic libraries. The study compares the different standards and libraries based on security and resource usage.

Another survey [10] looks at research regarding performing firmware updates in efficient ways, distributing firmware updates and security. Some techniques such as differencing algorithms, delta script dissemination are discussed. These techniques aim to reduce the amount of data transferred during an update and increase the lifetime of the flash memory in the devices. The study discusses existing protocols focused on distributing firmware updates and security. It also gives an overview of different firmware update platforms and their features.

There are some theses and articles that describe implementing OTA firmware update systems. The thesis [11] discusses using the nRF52840 development kit to create a proof of concept over-the-air firmware update system using Bluetooth Low Energy. An OTA DFU system with an example application was implemented using the nRF SDK and nRF Connect mobile application. The application includes Bluetooth Low Energy (BLE) services for DFU, a CO2 sensor, and LEDs. The thesis is similar to this one in that both use the nRF bootloader and similar hardware. However, in this thesis the transmission of update data was implemented in the application instead of using the bootloader to transmit data, which made the system more flexible. A firmware update server was also created in this thesis.

The thesis [12] describes a bootloader program, a DFU protocol and an interface program on a tablet that were developed for the thesis. The update protocol created uses the controller are network (CAN) bus. In a CAN bus all data is sent to all of the devices connected to the bus. The design of the bootloader is explained in the thesis.

The article [13] discusses design decisions concerning OTA firmware updates and their trade-offs. The design decisions discussed are having a second-stage bootloader, caching, compression and protocols. The article describes implementing a DFU architecture based on the design decisions discussed. Results, security and performance of the implementation are discussed at the end.

2.2 Embedded systems

An embedded system is a computer system that does a specific task inside a machine or a device [14, ch. 1.0]. They are different from general-purpose computers that can be used for many tasks. Some examples of devices that contain embedded systems are a microwave oven or a vending machine. The hardware on embedded systems is typically chosen to minimize cost, size, and power usage. Because of this embedded systems often have tight constraints on system resources [15]. Some embedded systems also have specific timing requirements. These kinds of systems are called real-time systems [16]. An example of a real-time system would be a car that needs to react to the brake pedal being pressed within a certain time frame.

Programs on embedded systems are usually called firmware instead of software. Firmware is made for specific hardware whereas software can typically be used on many hardware configurations. Firmware is stored in flash- or read-only memory [17]. Embedded devices can run on bare-metal without an operating system or have an operating system. Real-time operating systems that have certain timing guarantees are also commonly used. Having an operating system requires more system resources, but also makes multitasking easier.

Most embedded systems run on microcontrollers [14, ch. 1.2.1]. Microcontrollers are a type of computer that runs on a single computer chip. They are different from microprocessors in that they contain additional peripheral functions. They are useful because it makes the system more compact and efficient. Microcontrollers typically have less processing power than microprocessor-based systems [18, ch. 1]. Microcontrollers contain a processor, program memory, data memory, timers, and peripherals for input/output. They can also contain other application specific peripherals. A block diagram of a typical microcontroller is shown in figure 3.

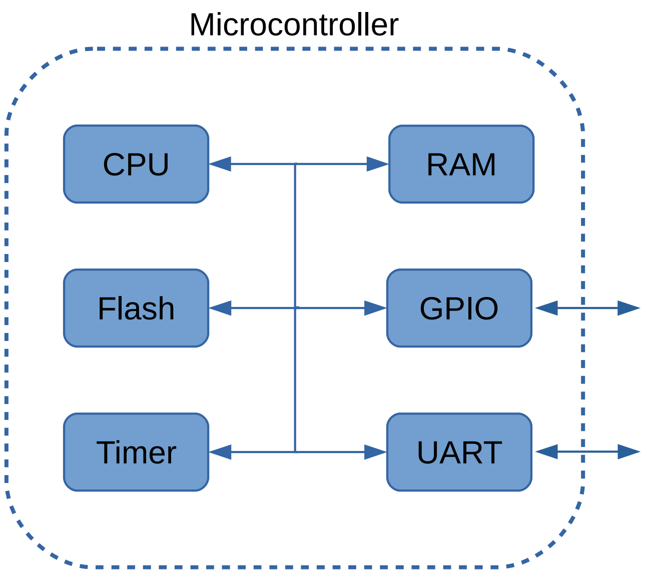


Figure 3: Block diagram of a typical microcontroller system

Some embedded systems also use systems on a chip (SoC) which are like microcontrollers but, they integrate even more functions into one chip. They can have peripherals like graphics processing units, wireless modules like Bluetooth or Wi-Fi, and coprocessors. Integrating the parts onto one chip makes the system more efficient and can increase performance [19].

2.3 Internet of Things

Internet of things has become an important concept during the past decade. Figure 4. illustrates how the interest in IoT has increased using data from Google Trends. The term means physical objects with unique identifiers that are connected to the internet [20, p. 72]. The devices can use sensors to collect data and use actuators to affect their environment. Some examples of Internet of Things devices are a light bulb that can be turned on or off remotely or a temperature sensor that sends temperature readings to a mobile device.

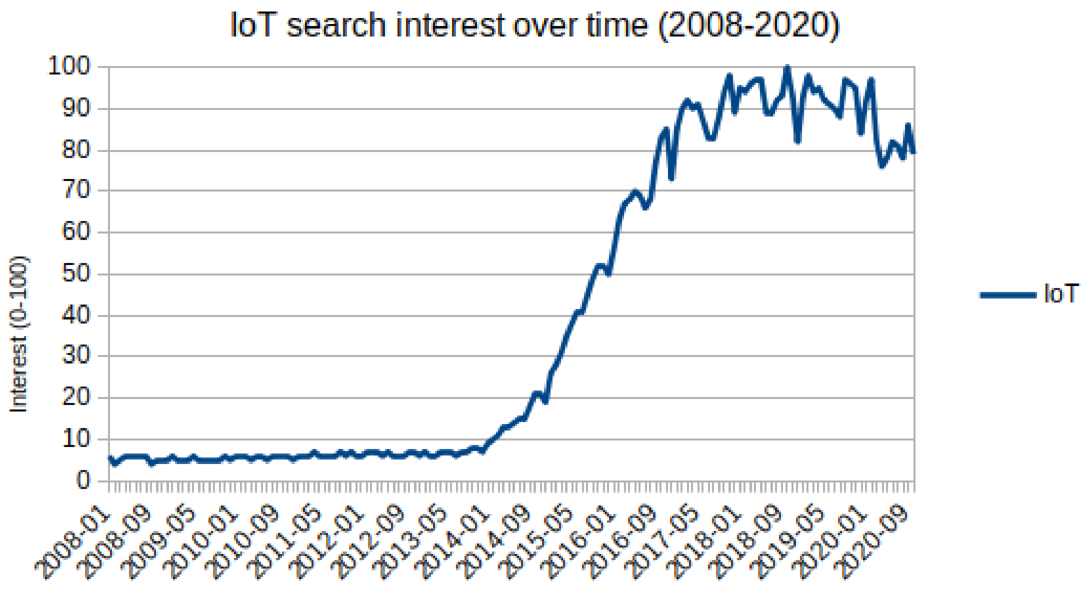


Figure 4: Search interest in the term "IoT" over time

The benefit of connecting objects to the Internet is that they can be accessed at any time using their unique identifier. Internet connected devices make it possible to automate and optimize tasks in factories, homes and cities [21, p. 4-5]. The data collected by the devices can be sent to cloud servers to be processed. The information can then be sent to other devices or users. [22, ch. 1.1.] It is estimated that there were 9.5 billion connected Internet of Things devices at the end of 2019 [23]. The large number of these devices makes updating them manually inconvenient, which is why OTA firmware updates have become commonplace for IoT devices.

2.4 Device Firmware Updates

Device firmware updates can be used to replace the firmware on a device. This makes it possible to fix bugs and add new features. In the past devices could only be updated locally, which made it impractical to update thousands of devices. However, with more devices having a wireless connection it is now possible to update them over-the-air. OTA updates allow devices to be deployed before the firmware development is finished. To update the firmware remotely the devices need to be connected to a server that sends the firmware data to the devices. In addition to sending the firmware binary some metadata about the firmware is commonly sent too. The metadata can contain things such as checksums to verify the integrity of the data, hardware version the firmware is meant for and the firmware version number. This information can help prevent incorrect firmware from being installed onto the device. After transferring the firmware binary a program called a bootloader is typically used to verify it and write it to flash.

There are some existing systems for OTA firmware updates such as UpdateHub [24], Mender [25] and SWUpdate [26]. These systems support security features such as encryption and have ways to easily manage updates. Mender and SWUpdate only support Linux while UpdateHub also supports the Zephyr operating system. Figure 5. illustrates how the Mender architecture works.

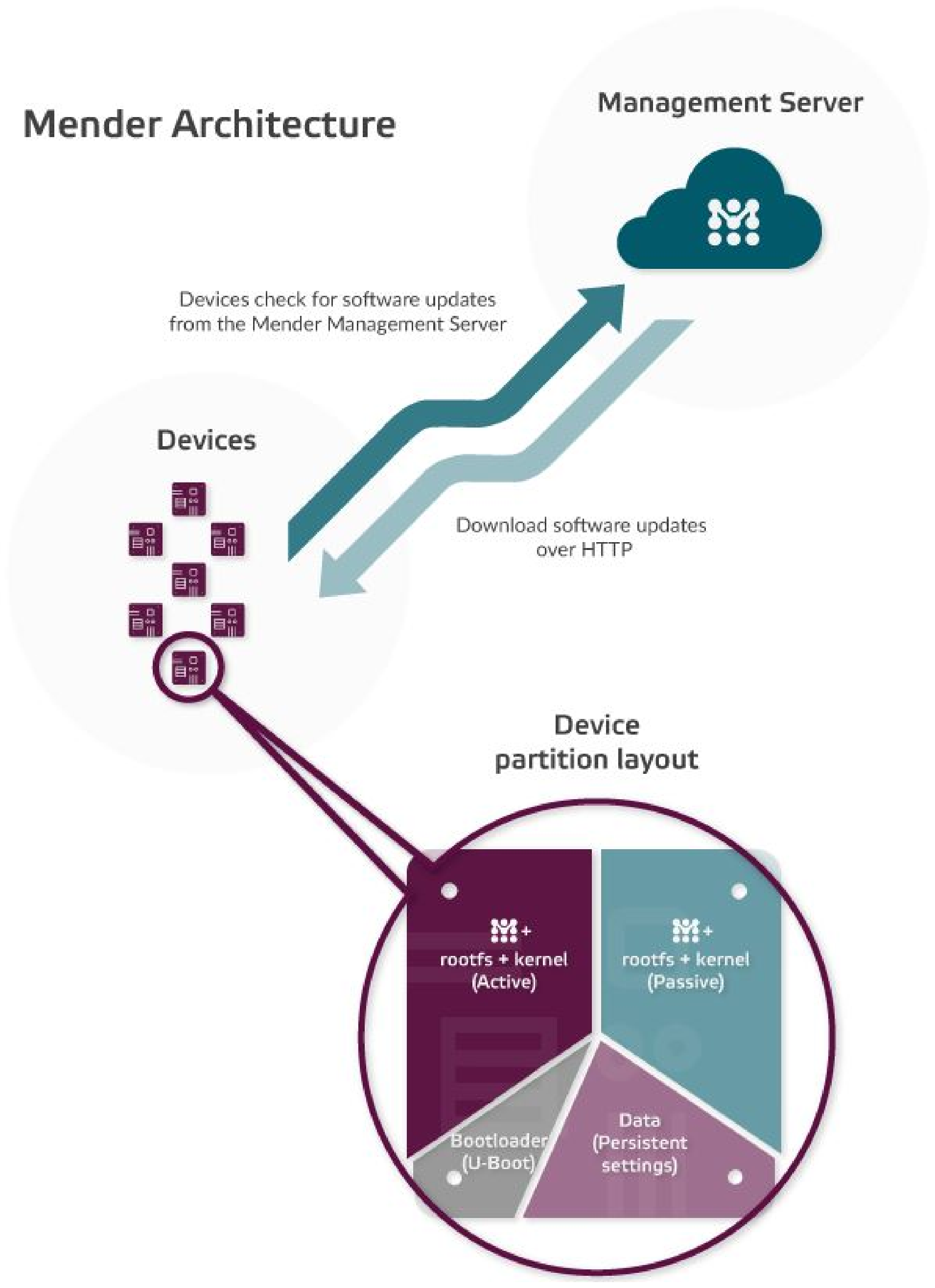

Mender Architecture
Management Server
Devices check for software updates from the Mender Management Server
Devices
Download software updates over HTTP
Device partition layout
rootfs + kernel (Active)
rootfs + kernel (Passive)
Bootloader (U-Boot)
Data (Persistent settings)

Figure 5: Mender architecture [31]

MCUboot is a secure bootloader that supports the Zephyr and Apache Mynewt operating systems. It includes support for cryptographic signatures and encrypted firmware images. [27.] There is an OTA firmware update framework for the ESP32 microcontroller. The framework includes a Python API to send commands to the device. The API allows reading, writing, and switching the OTA partition. [28.] The STM32 microcontroller has a software module called SBSFU which allows secure firmware updates using secure boot [29]. There are also protocols such as Lightweight M2M that support OTA firmware updates [30].

2.5 Bootloaders

Bootloaders are typically the first program that runs when a device boots. Their purpose is to load the operating system or firmware into memory. On embedded devices they are also used to update the firmware. When the bootloader runs it checks if an update is needed. This is often done using configuration parameters written by the application to non-volatile memory. [32, p. 9.] The update can either be loaded from flash or the bootloader can receive it from the update server. Once the firmware data is on the device it needs to be validated by the bootloader. The bootloader can check that the firmware version is correct, validate the data using a cyclic redundancy check (CRC) and check the cryptographic signature. If the firmware is valid the old firmware in flash memory is then overwritten with the new one, after which the new firmware is executed. If the new firmware does not work correctly a way to revert the firmware to a previous version is needed to get the device to a working state. or this purpose a previous version of the firmware is often stored in memory in addition to the current version [9, ch. 2A]. This has the downside of using more memory. An example of this is shown in figure 6. It is also possible to update the bootloader if multiple bootloader stages are used [33, ch.: Multiple Boot Stages].

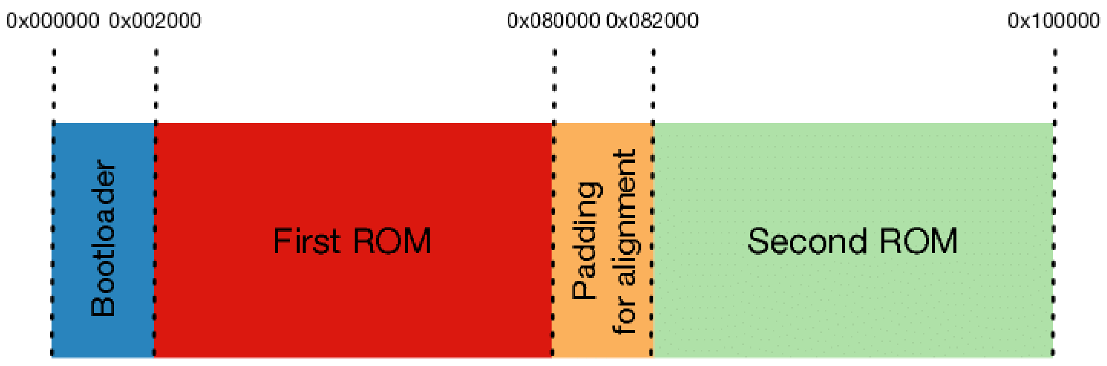


Figure 6: Example of a memory map with two application images [34, fig. 2.]

For a program to be able to be updated some modifications are needed. The program needs some way of knowing when an update is available. It also needs to be able to signal to the bootloader that an update is needed. Lastly the location of the application in memory needs to be modified so that it fits into memory along with the bootloader.

Sometimes ways to remotely check the firmware version a device has installed are added too.

In some cases it is necessary to minimize the amount of data written to flash memory in order to maximize the lifetime of the device and conserve power. In these cases a method called delta update can be used to calculate the differences of the new firmware compared to the old one. This minimizes the data that needs to be transmitted by only sending the differing parts. [35.] Another way to reduce update sizes is to use compression algorithms. Lempel–Ziv–Welch (LZW) is a popular compression algorithm that can be implemented on embedded systems [36].

2.6 Network Topologies

When designing a DFU system the network topology needs to be taken into account. In some cases the devices are spread out geographically and it is not feasible for all of them to connect to a central server. In these cases a multi-hop system where the devices closer to the server forward data to the devices further away is needed [37]. Multi-hop systems allow more devices to be reached but are more complicated. Figure 7. illustrates a partially connected mesh network.

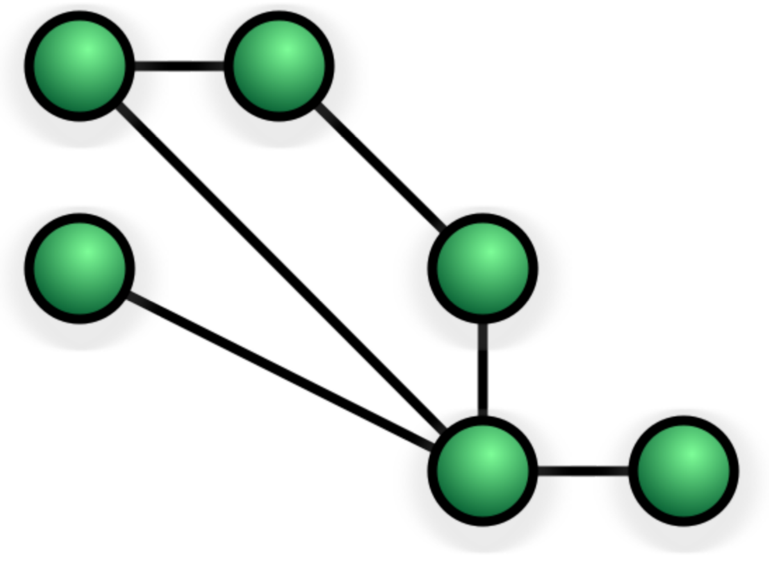


Figure 7: Partially connected mesh network [38]

There are a number of network topologies such as bus, star and mesh. Each type of topology has some advantages and disadvantages. In the star topology, for instance, each node is connected to one central node. This makes the network very simple, but also leaves the network susceptible to failure if the central node fails. Mesh networks are a common type of topology used for IoT devices. They can be either fully connected or partially connected. One benefit of mesh networks is that the network can keep functioning even if a node fails.

2.7 Protocols

The choice of protocol is important when designing a firmware update system. Different protocols have different features and properties. In some systems it is important to use protocols that use less energy or bandwidth. Other systems need low latency or reliability. Transmission distance can also be affected by the protocol. Multiple protocol layers can be used on top of each other. Some common protocols for IoT systems are message queuing telemetry transport (MQTT) and constrained application protocol (CoAP). Figure 8. compares typical IoT protocols to TCP/IP protocols.

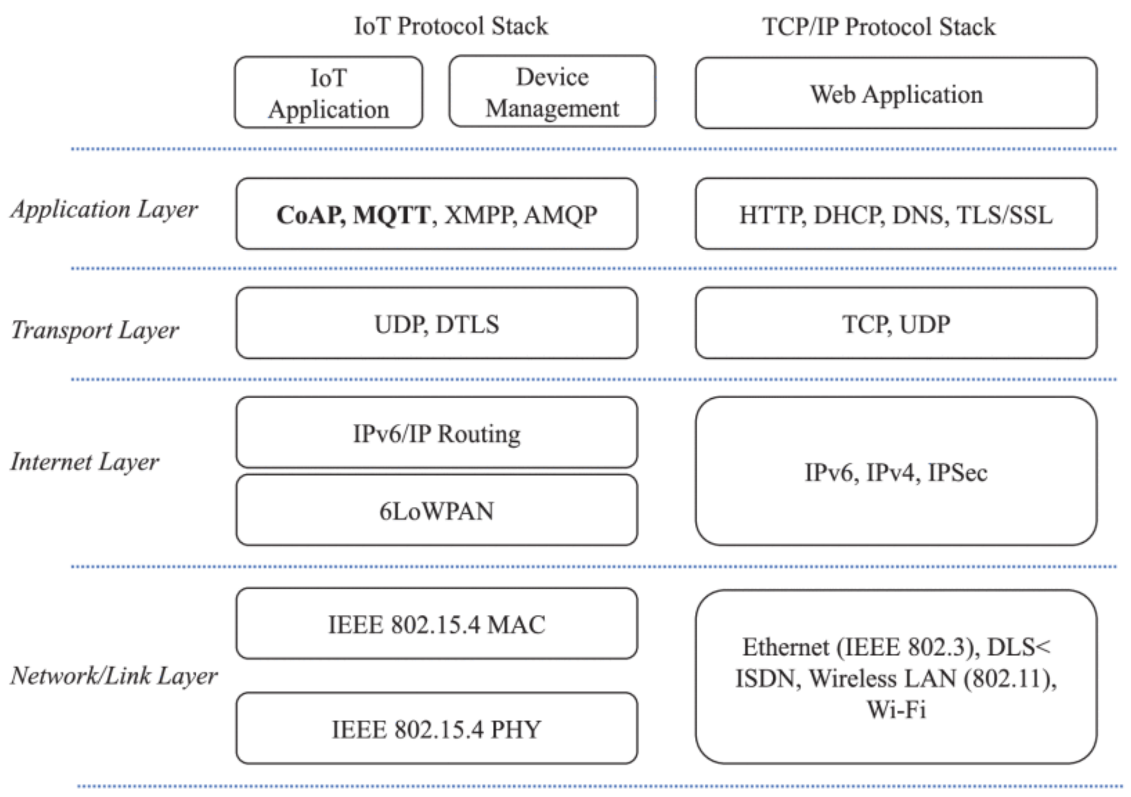


Figure 8: IoT protocols compared to TCP/IP protocols [39, fig. 3.]

MQTT is a publish/subscribe based protocol with low bandwidth usage [40]. Devices using MQTT can subscribe to different topics. Each message published to that topic is sent to the devices that subscribe to the topic through a broker. MQTT also supports a quality of service setting that configures how reliable the data transmission needs to be.

CoAP is a web transfer protocol for resource constrained devices [41]. Similarly to HTTP it works in a request/response-based format. There are different methods such as GET for getting data and PUT for sending data. There are also error codes for different error conditions. The protocol supports sending data to multiple devices using multicast and also has extensions such as encryption using datagram transport layer security (DTLS).

2.8 Security

There have been cases where vulnerabilities in IoT devices have been used to create botnets. One example is the Mirai malware that targeted IP cameras and home routers with weak login credentials [42]. Another example is the Ripple20 vulnerability [43]. It is important that these vulnerabilities are patched quickly before they can be exploited. Around 70% of the serious security bugs in the Chromium project were memory safety bugs [44]. Memory safety bugs can occur in languages with manual memory allocation such as C and C++ which are commonly used in embedded systems [45]. Static code analyzers can detect some of these issues.

DFU systems need to be implemented with security in mind or they risk creating security vulnerabilities in the system. If cryptographic signatures are not used for authenticating updates there is a risk that a malicious update could be installed by the device. With cryptographic signatures the device can only install updates from authorized sources. Using encryption for the firmware update process prevents a man-in-the-middle attack where the update package is tampered with or sensitive information is leaked during transmission. Using version checks prevents the firmware from being reverted to an older version with vulnerabilities.

2.9 File Formats

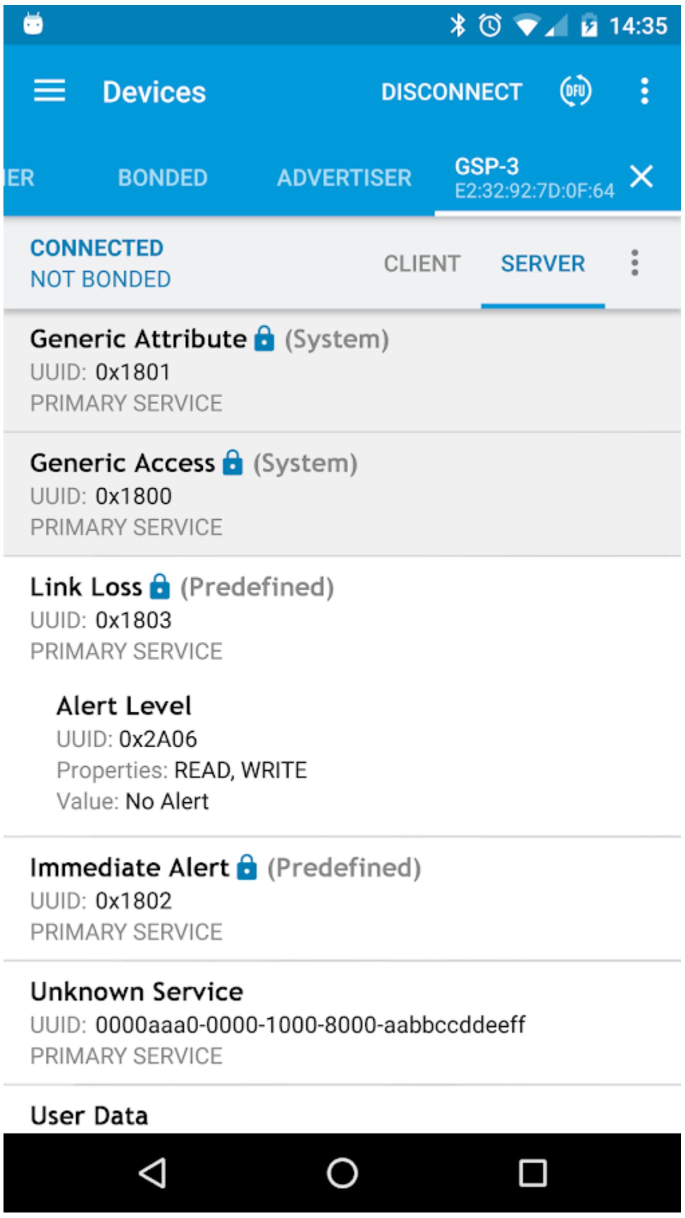
The firmware file can be in different file formats. Intel HEX and Motorola SREC are file formats that store binary data as ASCII text [46] [47]. These formats are commonly used for programming microcontrollers. Both of the formats use record structures where each record contains a type, a memory address, byte count, data, and a checksum. The records are represented as hexadecimal numbers. The files need to be converted to binary before they can be executed by the microcontroller.

2.10 nRF SDK

The nRF SDK contains libraries and example code for nRF devices. There is an example bootloader project in the SDK [48]. The bootloader handles running the application when the device boots. It can also download updates using the DFU protocol when it is set to DFU mode by the application. The bootloader can update the application firmware, the Bluetooth software stack, or itself. It can use one or two memory banks for updates. There is a secure version of the bootloader that requires that all updates are signed with a private key.

When building the nRF bootloader example there is an option in the configuration file to use one or two DFU banks. If one bank is used the previous firmware is overwritten when an update is performed. If the update validation fails the new firmware can’t be booted. If two banks are used the new firmware is first written in to the second bank and then validated. If the validation fails the old firmware is used. If it succeeds the old firmware is overwritten and the new firmware is booted.

There is also a DFU library in the SDK [49]. The DFU library contains functions for configuration, validation and DFU transport. There are example transport layers for Bluetooth and serial. It’s also possible to create a custom transport layer. A protocol is defined in the SDK for the DFU process. The protocol includes commands for initializing, transferring data and validating the update. The protocol sends required information such as the type and size of update. The DFU process requires that a DFU package is created with the nRF Util tool. The package is a zip file that contains the firmware binary and the initialization packet. The initialization packet is sent during the DFU process and contains information about the update. The nRF Connect mobile application can be used to update the device via Bluetooth [50]. Figure 9. demonstrates the nRF Connect mobile application.



Generic Attribute (System)
UUID: 0x1801
PRIMARY SERVICE

Generic Access (System)
UUID: 0x1800
PRIMARY SERVICE

Link Loss (Predefined)
UUID: 0x1803
PRIMARY SERVICE

Alert Level
UUID: 0x2A06
Properties: READ, WRITE
Value: No Alert

Immediate Alert (Predefined)
UUID: 0x1802
PRIMARY SERVICE

Unknown Service
UUID: 0000aaa0-0000-1000-8000-aabbccddeeff
PRIMARY SERVICE

User Data

Figure 9: nRF Connect for Mobile

The nRF SDK contains the master boot record (MBR) which is the first thing that runs when the device is booted. The MBR makes it possible to update the bootloader and to recover from unexpected resets during the update process [51]. Once the MBR is done executing it runs the bootloader if one is detected.

3 Implementation

3.1 Overview

The goal of the thesis was to implement a working DFU system using the nRF52840 microcontroller and the Raspberry Pi single-board-computer. The nRF52840 used a custom firmware and a bootloader from the nRF SDK. The Raspberry Pi used the Raspbian operating system and a Python program for the DFU server. The Raspberry Pi periodically checks for new firmware versions from a file server and downloads them through HTTP. The new firmware is transferred through universal asynchronous receiver-transmitter (UART) from the Raspberry Pi to the nRF52480. It is then written to flash memory, after which the system resets and the bootloader overwrites the old firmware if the new one is verified to be correct.

3.2 Hardware

The prototype system used an nRF52840 on a custom board made by Wizense for the device that is updated. The nRF52840 is an SoC made by Nordic Semiconductor. It contains 1MB of flash memory, 256 kB of RAM and includes support for Bluetooth 5. The processor is an ARM Cortex-M4F. [52.] Raspberry Pi was used to send the update package. Raspberry Pi is a single-board computer which means it works with peripherals such as a monitor. The Raspberry Pi model is Raspberry Pi 2B, which contains a 900 Mhz quad-core ARM Cortex-A7 CPU and 1GB RAM [53]. Figure 10. demonstrates the development setup used for the project.

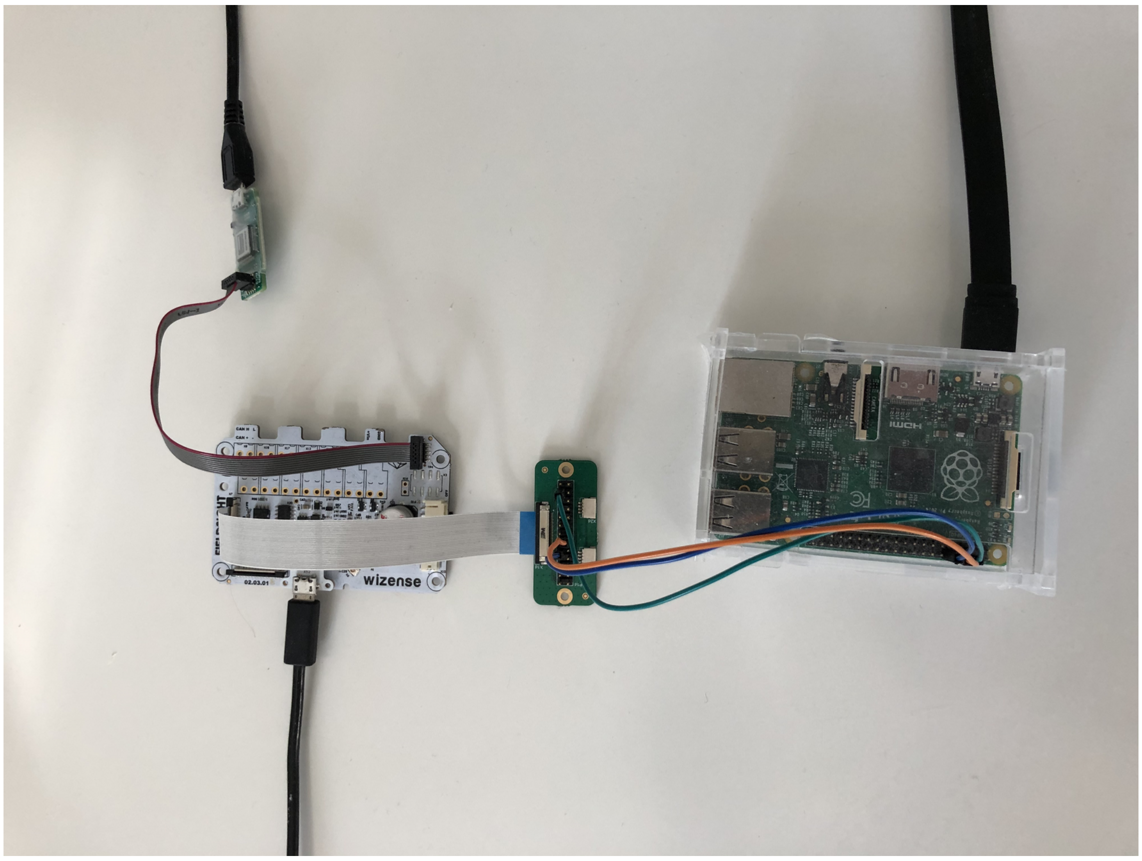


Figure 10: Development setup for the DFU system

The Raspberry Pi is not suitable for some industrial uses due to a lack of some connectivity interfaces and not being able to endure harsh conditions [54]. In future versions of the system another device could be used instead of the Raspberry Pi. Figure 10. demonstrates the development setup for the system.

3.3 DFU Protocol

The DFU transport layer from the nRF SDK was not used for the system. A custom protocol was instead designed by the author. The custom protocol makes the system more flexible and makes it easier to add custom features such as compression, encoding, and encryption. Table 1. demonstrates the firmware update protocol commands. When not using the tranport layer from the nRF SDK an initialization packet needs to be created by the device so that the bootloader can verify the update.

Table 1: Firmware update protocol commands

Command	**Description**	**Parameters**
0	Get firmware version number	None
1	Get number of firmware chunks	None
2	Get firmware SHA-256 hash	None
3	Get firmware chunk	Chunk number (0-255)

The firmware package is sent from the Raspberry Pi to the nRF52480 via a serial UART connection. The nRF52480 receives an interrupt when it gets the command to start the DFU. Then it sends a request to fetch the version number of the new firmware. If there is a new firmware available the client then sends a request to get the new firmware. The update is sent to the client in parts of 1024 bytes each with a CRC code for each part to check that it is valid. Sending the firmware in parts has the benefit of not having to resend the entire file if there is a transmission error. The update process is illustrated in figure 11.

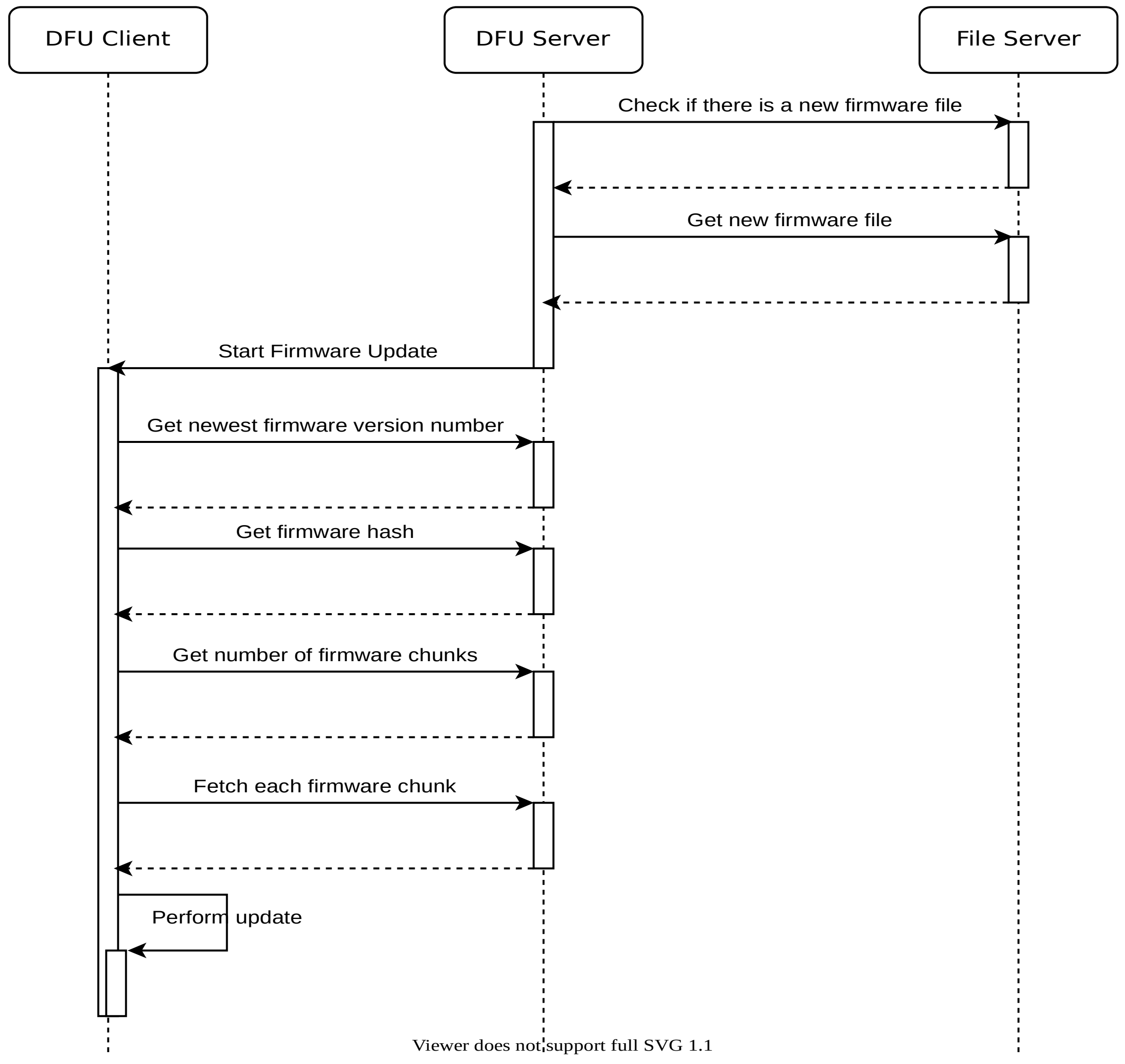

DFU Client
DFU Server
File Server
Check if there is a new firmware file
Get new firmware file
Start Firmware Update
Get newest firmware version number
Get firmware hash
Get number of firmware chunks
Fetch each firmware chunk
Perform update
Viewer does not support full SVG 1.1

Figure 11: Diagram of the DFU process

The Raspberry Pi software was implemented using Python. The program first sends a command to the device to start the update process and then waits for commands from the serial port. There are commands for getting the version number of the newest firmware, getting the number of chunks of the firmware file, getting a specific chunk of the firmware file, getting the 256-bit secure hash algorithm (SHA-256) hash of the firmware file. Code from the nRF Util program by nRF was used as an example for calculating the update hash. The firmware chunks and the SHA-256 hash are sent with a CRC code to make sure they are not corrupted during the transfer. The firmware file needs to be converted from the Intel HEX format to a binary format before sending it to the nRF52480. Code for sending an update chunk to the device is shown in listing 1.

The code calculates a CRC code and a length for the data and sends it through the serial port.

```
def send_update_chunk(chunk, dfu_data, ser):
    try:
        if chunk < num_chunks:
            data = dfu_data[chunk * chunk_size: (chunk + 1) * chunk_size]
        else:
            data = b''
        crc = crc32(data) & 0xffffffff
        crc = '{0:0>8x}'.format(crc)
        len1 = (len(data) & 0xff00) >> 8
        len2 = len(data) & 0xff
        lenbytes = bytes((len1, len2, len1 ^ 0xff, len2 ^ 0xff))
        crcbytes = bytes(int(crc[n * 2 : n * 2 + 2], 16) for n in range(4))
        ser.write(lenbytes + data + crcbytes)
    except serial.SerialTimeoutException as e:
        print('timed out')
```

Listing 1. Python code from the server program to send an update chunk to the device

3.4 DFU Process

The existing bootloader from the nRF software development kit was used for booting and validating the firmware. Using an existing bootloader saves development time and avoids the risk of creating bugs in the bootloader. The bootloader is programmed onto the device along with the application, the bootloader settings file and the master boot record. The memory map of the system is shown in table 2.

Table 2: System memory map

Memory area	**Purpose**
0x00000-0x01000	Master boot record
0x01000-0xDFFFF	Application
0xE0000-0xFDFFF	Bootloader
0xFE000-0xFEFFF	MBR parameters
0xFF000-0xFFFFF	Bootloader settings

The settings file needs to be programmed into the flash memory before starting the application because some of the values in the settings can’t be written from the application. The settings file is generated using the nRF Util command line program (Listing 2.).

```
nrfutil settings generate --application application.hex --family NRF52840 --
application-version 0 --bootloader-version 0 --bl-settings-version 2
settings.hex
```

Listing 2. The command for creating the bootloader settings file

When the application updates it sends commands to the Raspberry Pi through UART. It checks if there is a new firmware available and then fetches the firmware in chunks. Each firmware chunk is validated with a CRC code. If one of the CRC checks fails the application retries 20 times before stopping. Once the firmware binary has been received the application writes the bootloader settings to the global variable s_dfu_settings. The bootloader settings have the location and offset of the firmware binary and an initialization packet. The initialization packet is in the protobuf format and the structure of the packet is defined in the nRF SDK. Protobuf is a format defined by Google for serializing structured data [55]. The initialization packet contains information about the firmware binary, its SHA-256 hash and CRC code. The code for creating the initialization packet is shown in listing 3. The flash memory has two memory banks for application images. After writing the bootloader settings the application then writes the new firmware from RAM into the second memory bank. The system then reboots.

```
memset(s_dfu_settings.init_command, 0xFF, INIT_COMMAND_MAX_SIZE);

dfu_packet_t packet = DFU_PACKET_INIT_DEFAULT;
packet.has_command = true;
packet.command.has_init = true;
packet.command.init.has_type = true;
packet.command.init.type = DFU_FW_TYPE_APPLICATION;
packet.command.init.has_app_size = true;
packet.command.init.app_size = 0;
packet.command.init.has_is_debug = true;
packet.command.init.is_debug = true;
packet.command.init.has_hash = true;
packet.command.init.hash.hash_type = DFU_HASH_TYPE_SHA256;
packet.command.init.hash.hash.size = 32;
uint8_t hash_bytes[32] = {0};

packet.command.init.app_size = data_received;
memcpy(packet.command.init.hash.hash.bytes, hash_bytes, sizeof(hash_bytes));
uint8_t buffer[DFU_INIT_COMMAND_SIZE] = {0};
pb_ostream_t stream;
stream = pb_ostream_from_buffer(buffer, sizeof(buffer));
bool success = pb_encode(&stream, dfu_packet_fields, &packet);
if (!success) loop_forever();
memcpy(s_dfu_settings.init_command, buffer, stream.bytes_written);
```

Listing 3. Code for creating the initialization packet

When the device boots the MBR is the first thing that runs. The MBR checks if a bootloader is present and executes it [56]. When the bootloader runs it checks if there is a new firmware that can be activated. If a new firmware is found the bootloader then performs post validation. In post validation the bootloader checks that the SHA-256 hash, CRC code and other information in the bootloader settings match the firmware binary [57]. If the validation succeeds the firmware is then activated. During activation the bootloader overwrites the old firmware in the first memory bank with the new one and updates the bootloader settings page [58]. The bootloader then executes the new firmware. Figure 12. illustrates the system boot procedure.

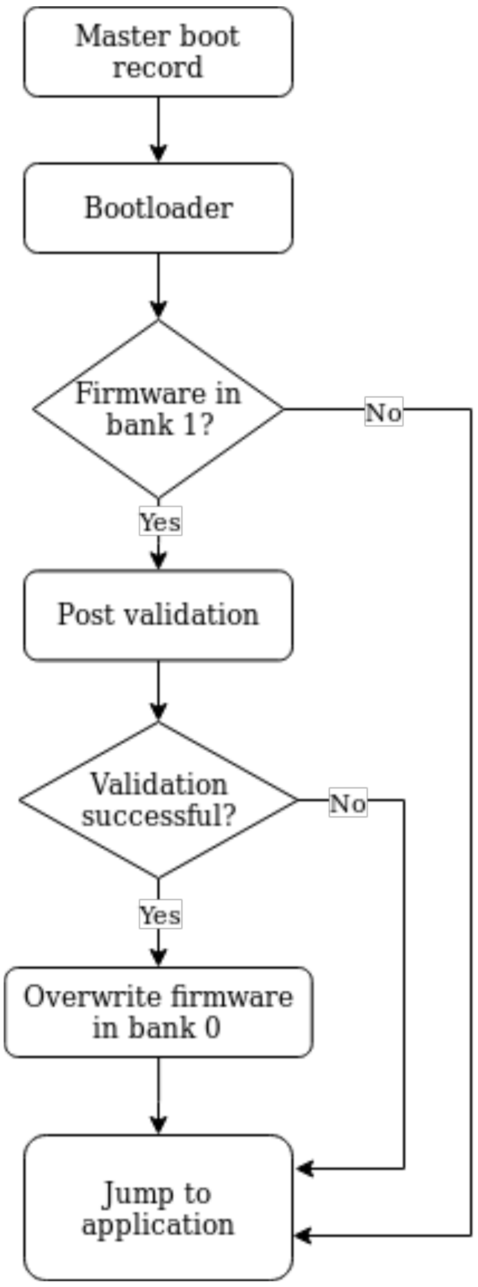


Figure 12: System boot procedure

When the application runs it can receive another update from the server. The application can also be updated to a completely different application. If the update process is interrupted the system is able to recover to a working state.

3.5 Testing

In order to verify the functionality of the system testing is required. In order to test the system the correct functionality needed to be defined first. The DFU process needed to be interruptible in order to be reliable. If the system is not interruptible the system can be left in a non-working state after an update. The system needed to be able detect if the firmware is corrupted during transmission. If this feature was not implemented the corruption could cause errors in the system. The device also needed to be able to recover if a faulty firmware was sent to the device. These requirements are listed in table 3.

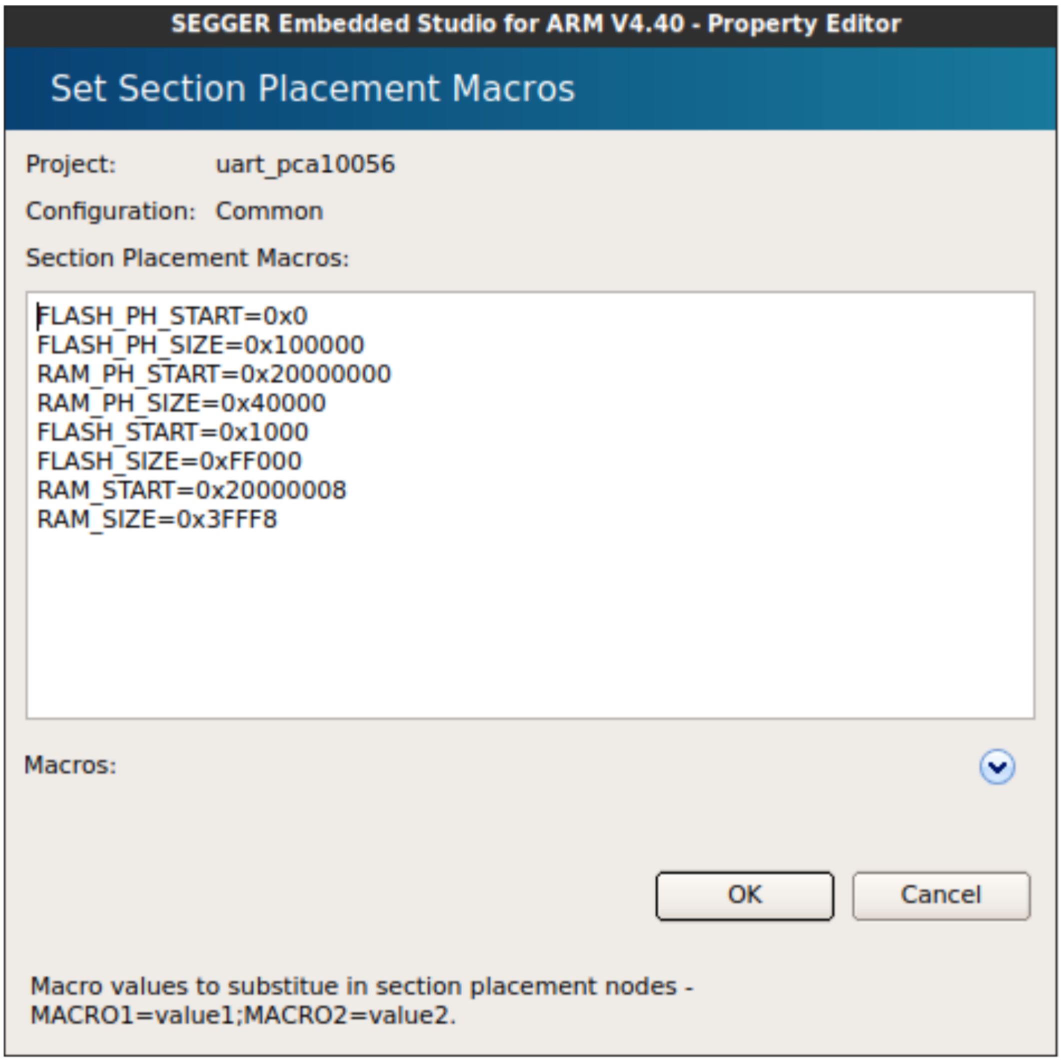
Table 3: Testing results

Requirement	**Result**
Device can recover if the DFU process is interrupted during transmission.	Device kept trying to receive transmission for 20 seconds and then stopped the DFU process.
Device can recover if the firmware is corrupted during transmission.	Device recovered after rebooting.
Device can recover if a faulty firmware is sent to the device.	Device got stuck and didn’t recover.

The system was tested based on the requirements. The first two requirements were met while the last one was not. Possible ways to meet the last requirement are discussed in chapter 6. The system was also tested by updating to firmwares with different file sizes and functionalities.

3.6 Challenges

There were a number of challenges during the implementation. There was a problem where the system would reboot during the DFU process, but not update to the new firmware. Using a debugger it was noticed that the system didn't execute the bootloader. The solution was to add the master boot record to the device's flash memory. To do this the memory address for the application needed to be changed (Figure 13.). After adding the MBR the bootloader executed correctly.


SEGGER Embedded Studio for ARM V4.40 - Property Editor
Set Section Placement Macros

Project: uart_pca10056
Configuration: Common
Section Placement Macros:

```
FLASH_PH_START=0x0
FLASH_PH_SIZE=0x100000
RAM_PH_START=0x20000000
RAM_PH_SIZE=0x40000
FLASH_START=0x1000
FLASH_SIZE=0xFF000
RAM_START=0x20000008
RAM_SIZE=0x3FFF8
```

Macro values to substitue in section placement nodes - MACRO1=value1;MACRO2=value2.

Figure 13: Modifying the firmware's placement in memory in SEGGER Embedded Studio

There was an issue with getting the UART transmission to work. The UART connection worked very unreliably and sometimes sent random data. When the UART connection was analyzed with a logic analyzer it was noticed that the RX pin on the Raspberry Pi was stuck on high logic level even when it was not connected. It was also noticed that there was a warning on the Raspberry Pi console about undervoltage. The issue was fixed by using a shorter power cable with less resistance. Similar issues could be avoided in the future by testing what happens when the system doesn’t get enough voltage.

4 Results

The thesis explained how over-the-air firmware updates work and reviewed prior literature on the subject. A working OTA firmware update system was created for the thesis. Figure 14. illustrates an overview of the system. DFU client and server programs were created. The client program runs on the nRF52480 and communicates with the server program through UART. The server program is written in Python and runs on a Raspberry Pi. New updates can be deployed by uploading them to the file server.

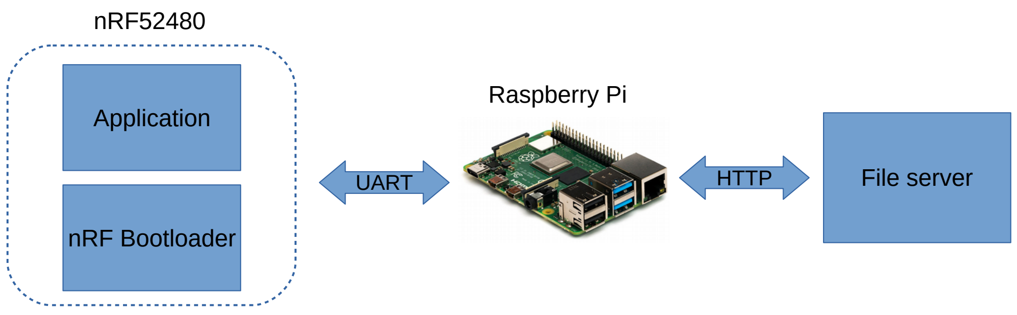


Figure 14: Diagram of the firmware update system

A way to transmit firmware updates to the device in application code was developed for the thesis. A custom protocol that includes CRC codes to validate the data was designed to transfer the update. The system used a bootloader from the nRF SDK to validate and apply the update. The system can be easily extended to add new features. Because the system transfers the update in application code it is easier to modify it to work with a different bootloader. The system is scalable to multiple devices. The system was tested to validate its functionality. The system can survive a power-off during the update process due to the bootloader being power fail-safe [58]. One potential reliability issue was discovered based on the testing.

5 Conclusions

DFU systems are important for embedded systems that are deployed at large scale. To update device firmware a program called a bootloader is used. It checks the validity of the new firmware and overwrites the old one. For DFU systems to be reliable the update process needs to be interruptible. Multiple application images can be stored so that the system can revert to an older version if an update fails. There are a number of ways the update process can be optimized to reduce power usage and increase device lifetime. Different protocols such as MQTT and CoAP can be used for firmware updates. The protocols can have different features such as quality of service and multicast.

A working DFU system was created for the thesis. The basic functionality of the system works correctly. The system is power fail-safe, which makes it more reliable. The system can verify that the update file is correct by calculating CRC codes for the data. It's possible that the system can get into a nonworking state if a faulty firmware update is installed onto the system. There are possible changes that can be made in the future to mitigate this problem. Using an existing bootloader made the development process easier. Using a custom protocol to transfer the update made the system more flexible.

6 Future Development

In the future features such as automating update deployment could be added to make the update process more convenient. New firmware could be built and deployed automatically whenever a new version of the source code is checked in to version control. The system could also monitor the state of the devices. The protocol used for the system could also be improved to be more flexible. Features that make the update process more efficient could also be added. Some examples are only sending parts of the firmware that are different from the previous version and using a compression algorithm on the data. The system could also be modified to work with a network module instead of a Raspberry Pi. Support could also be added for new transfer protocols such as Bluetooth. Other bootloaders could also be supported.

There are a number of improvements that could be made to the reliability of the firmware update system, for instance a watchdog feature could be added that detects if a new firmware is faulty and reverts to an older version. Errors could be detected using a flag that needs to be set by new firmware when it’s first booted to tell the bootloader the firmware is working. Another option could be to modify the bootloader to not overwrite the old version of the firmware. Exception handlers could also be implemented to detect errors in the firmware.

Currently the update has some flaws in terms of security. The system does not currently use cryptographic signatures to authenticate update data, which can make the system vulnerable if the update server is compromised. The nRF bootloader supports cryptographic signatures so they could be added with minimal effort. The system also does not currently encrypt update data, which allows the update data to be modified by a man-in-the-middle attack. There are existing libraries that could be used to implement this feature.

References

1 Dufresne, Steven. 2018. Inventing The Microprocessor: The Intel 4004. Online material. <https://hackaday.com/2018/01/29/inventing-the-microprocessor-the-intel-4004/>. Read 28.8.2020

2 Electronic Control Unit. Online material. Wikipedia. <https://en.wikipedia.org/wiki/Electronic_control_unit>. Read 28.8.2020

3 What is a Smart Refrigerator?. Online material. Lifewire. <https://www.lifewire.com/smart-refrigerator-4158327>.

4 Industrial internet of things. Online material. Wikipedia. <https://en.wikipedia.org/wiki/Industrial_internet_of_things> Read 28.8.2020

5 Intel C4004. Picture. Wikipedia. <https://en.wikipedia.org/wiki/Intel_4004#/media/File:Intel_C4004.jpg>

6 Toro-Betancur, Verónica; Zamora, José Viquez; Antikainen, Markku; Di Francesco, Mario. 2019. Article. A Scalable Software Update Service for IoT Devices in Urban Scenarios. <https://doi.org/10.1145/3365871.3365880>

7 Riissanen, Pasi. 2016. Remote Firmware Updating. Thesis. <http://urn.fi/URN:NBN:fi:amk-2016113018404>

8 Unterschütz. Stefan; Turau, Volker. 2012. Fail-safe over-the-air programming and error recovery in wireless networks. Article.

9 Zandberg, Koen; Schleiser, Kaspar; Acosta, Francisco; Tschofenig, Hannes; Baccelli, Emmanuel. 2019. Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check. Article. <https://doi.org/10.1109/ACCESS.2019.2919760>

10 Arakadakis, Konstantinos; Charalampidis, Pavlos; Makrogiannakis, Antonis; Fragkiadakis, Alexandros. 2020. Firmware over-the-air programming techniques for IoT networks -- A survey. Article. <https://arxiv.org/abs/2009.02260>

11 Nguyen Anh, Tuan. 2019. Over-the-Air Firmware Update for Bluetooth Low Energy Devices. Thesis. <http://urn.fi/URN:NBN:fi:amk-2019111321136>

12 Korpelin, Kimmo. 2018. Boot loader and firmware update protocol for embedded devices. Thesis. <http://urn.fi/URN:NBN:fi:amk-2018052410185>

13 Brown, Benjamin Bucklin. 2018. Article. <https://www.analog.com/media/en/analog-dialogue/volume-52/number-4/over-the-air-ota-updates-in-embedded-microcontroller-applications.pdf>

14 Canton, Maria P.; Sanchez, Julio. 2017. Embedded Systems Circuits and Programming. E-book. <https://www.oreilly.com/library/view/embedded-systems-circuits/9781439879313/>

15 Embedded System. Online material. Wikipedia. <https://en.wikipedia.org/wiki/Embedded_system>. Read 16.9.2020

16 Real-time Computing. Online material. Wikipedia. <https://en.wikipedia.org/wiki/Real-time_computing>. Read 16.9.2020

17 Firmware. Online material. Wikipedia.<https://en.wikipedia.org/wiki/Firmware>. Read 16.9.2020

18 Bishop, Owen. 2013. Microelectronics – Systems and Devices. E-book. <https://learning.oreilly.com/library/view/microelectronics-systems/9780750647236/>

19 System on a chip. Online material. Wikipedia. <https://en.wikipedia.org/wiki/System_on_a_chip>. Read 21.9.2020

20 Towards a definition of the Internet of Things (IoT). 2015. Article. IEEE. <https://iot.ieee.org/images/files/pdf/IEEE_IoT_Towards_Definition_Internet_of_Things_Revision1_27MAY15.pdf>

21 Ramgir, Mayur. 2019. Internet of Things. E-book. <https://learning.oreilly.com/library/view/internet-of-things/9789353941529/>

22 Veltri, Luca; Picone. Marco; Cirani, Simone; Ferrari, Gianluigi. 2018. Internet of Things. E-book. <https://learning.oreilly.com/library/view/internet-of-things/9781119359678/>

23 Lueth, Knud. 2020. IoT 2019 in Review: The 10 Most Relevant IoT Developments of the Year. Online material. <https://iot-analytics.com/iot-2019-in-review/>. Read 17.4.2020.

24 What is UpdateHub?. Online material. UpdateHub. <https://docs.updatehub.io/what-is-updatehub/>. Read 20.10.2020

25 Mender. Online material. <https://mender.io/>. Read 20.10.2020

26 SWUpdate. Online material. <http://sbabic.github.io/swupdate/swupdate.html>. Read 20.10.2020

27 mcuboot. Online material. <https://mcuboot.com/>. Read 26.10.2020

28 Over the air updates (OTA). Online material. Espressif Systems. <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/ota.html>. Read 26.10.2020

29 X-CUBE-SBSFU. Online material. STMicroelectronics. <https://www.st.com/en/embedded-software/x-cube-sbsfu.html>. Read 26.10.2020

30 OMA LWM2M. Online material. Wikipedia. <https://en.wikipedia.org/wiki/OMA_LWM2M>. Read 20.10.2020

31 Mender architecture. Picture. Mender. <https://mender.io/user/pages/blog/mender-management-server-test-version-released/Mender Architecture.JPG>

32 Beningo, Jacob. 2015. Bootloader Design for Microcontrollers in Embedded Systems. Article. <https://www.beningo.com/insights/white-papers/bootloader-design-for-microcontrollers-in-embedded-systems/>

33 Lacamera, Daniele. 2018. Embedded Systems Architecture. E-book. <https://learning.oreilly.com/library/view/embedded-systems-architecture/9781788832502/>

34 Frisch, Dustin; Reißmann, Sven; Pape, Christian. An Over the Air Update Mechanism for ESP8266 Microcontrollers. 2017. <https://www.researchgate.net/figure/The-flash-layout-used-for-two-ROM-slots_fig2_320335879>

35 Delta updates. Online material. Wikipedia. <https://en.wikipedia.org/wiki/Delta_update>. Read 25.10.2020

36 Lossless Data Compression for Embedded Systems. Online material. Embedded. <https://www.embedded.com/lossless-data-compression-for-embedded-systems/>. Read 5.11.2020

37 Stathopoulos, Thanos; Heidemann, John; Estrin, Deborah. A Remote Code Update Mechanism for Wireless Sensor Networks. Article. <https://apps.dtic.mil/sti/citations/ADA482887>

38 Mesh networking. Picture. Wikipedia. <https://commons.wikimedia.org/wiki/File:NetworkTopology-Mesh.svg>

39 Thantharate, Anurag ; Beard, Cory; Kankariya, Poonam. CoAP and MQTT Based Models to Deliver Software and Security Updates to IoT Devices over the Air. 2019. Article. <https://doi.org/10.1109/iThings/GreenCom/CPSCom/SmartData.2019.00183>

40 MQTT Version 5.0. 2019. Online material. OASIS. <https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html>. Read 29.9.2020

41 The Constrained Application Protocol (CoAP). 2014. Online material. IETF. <https://tools.ietf.org/html/rfc7252>, Read 29.9.2020

42 What is the Mirai Botnet. Online material. Cloudflare Inc. <https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/>

43 Grey, Mishka. 2020. Zero-Day Ripple20 Vulnerability Puts Millions of IoT Devices at Risk. Online material. <https://www.hackreports.com/zero-day-ripple20-exploit-iot-vulnerability/>. Read 14.8.2020

44 Memory safety. 2020. Online material. Chromium. <https://www.chromium.org/Home/chromium-security/memory-safety>. Read 5.11.2020

45 The Top Programming Languages 2020. 2020. Online material. IEEE Spectrum. <https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020>. Read 5.11.2020

46 Intel HEX. Online material. Wikipedia. <https://en.wikipedia.org/wiki/Intel_HEX>. Read 29.9.2020

47 SREC (File format). Online material. Wikipedia. <https://en.wikipedia.org/wiki/SREC_(file_format)>. Read 29.9.2020

48 Bootloader. Online material. Nordic Semiconductor Inc. <https://infocenter.nordicsemi.com/topic/sdk_nrf5_v16.0.0/lib_bootloader.html>. Read. 29.9.2020

49 Device Firmware Update Process. Online material. Nordic Semiconductor Inc. <https://infocenter.nordicsemi.com/topic/sdk_nrf5_v16.0.0/lib_bootloader_dfu_process.html>. Read. 29.9.2020

50 nRF Connect for Mobile. Online material. Nordic Semiconductor Inc. <https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Connect-for-mobile>. Read 5.11.2020

51 Master boot record. Online material. Nordic Semiconductor Inc. <https://infocenter.nordicsemi.com/topic/sds_s140/SDS/s1xx/mbr_bootloader/mbr.html>. Read 11.8.2020

52 nRF52480 Product Specification. Online material. Nordic Semiconductor Inc. <https://infocenter.nordicsemi.com/topic/ps_nrf52840/keyfeatures_html5.html>. Read 29.9.2020

53 Raspberry Pi 2 Model B. Online material. Raspberry Pi Foundation. <https://www.raspberrypi.org/products/raspberry-pi-2-model-b/>. Read 5.11.2020

54 The Future of Raspberry Pi in IoT-enabled Industrial Applications. 2019. Online material. Thomasnet. <https://www.thomasnet.com/insights/can-raspberry-pi-be-used-for-industrial-applications/>. Read. 24.8.2020

55 Protocol Buffers. Online material. Google Inc. <https://developers.google.com/protocol-buffers>. Read 10.8.2020

56 Master boot record and SoftDevice reset procedure. Online material. Nordic Semiconductor Inc. <https://infocenter.nordicsemi.com/topic/sds_s140/SDS/s1xx/mbr_bootloader/mbr_sd_reset_behavior.html>. Read 10.8.2020.

57 Validation. Online material. Nordic Semiconductor Inc. <https://infocenter.nordicsemi.com/topic/sdk_nrf5_v16.0.0/lib_bootloader_dfu_validation.html#lib_bootloader_dfu_init_validation>. Read 10.8.2020

58 Firmware activation. Online material. Nordic Semiconductor Inc. <https://infocenter.nordicsemi.com/topic/sdk_nrf5_v16.0.0/lib_bootloader.html#lib_bootloader_firmware_activation>. Read 10.8.2020