FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of master's thesis

Title:	Robust flash memory bootloader for a microcontroller over near field communication
Student:	Bc. Jitka Seménková
Supervisor:	Ing. Jiří Hušák
Study program:	Informatics
Branch / specialization:	Design and Programming of Embedded Systems
Department:	Department of Digital Design
Validity:	until the end of summer semester 2022/2023

Instructions

Develop a robust flash memory bootloader for a proprietary RISC-V based microcontroller, which will allow the update of the internal flash memory over near field communication (NFC) on the fly.

Implementation:
- Get familiar with all needed technologies.
- Develop nonvolatile flash low-level firmware driver or use an existing one.
- Design and develop all necessary communication layers in the firmware and the bootloader which uses this communication stack.
- Develop an application for one of the mobile platforms (iOS or Android), which can be used to test and benchmark the whole solution.
- Document completely the results in terms of bootloader footprint, transfer time and power consumption.

The final solution must be able to recover from all possible use cases (loss of power, out of range...).

The design choices made in this thesis should support robustness, security, speed of communication and usability in large scale production inside a size constraint of a wearable device design.

Electronically approved by doc. Ing. Hana Kubátová, CSc. on 22 February 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Robust flash memory bootloader for a microcontroller over near field communication

Bc. Jitka Seménková

Department of Digital Design
Supervisor: Ing. Jiří Hušák

June 27, 2021

Acknowledgements

I would like to thank my supervisor Ing. Jiří Hušák for his valuable advice, comments and help during the whole process of writing this thesis. Also, I would like to thank the team members for introducing me to the project and for their code reviews. Furthermore, I would like to thank my family for their support. Finally, I would like to thank the correctors of this thesis, especially Bc. Martina Klimešová.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived its right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act. This fact shall not affect the provisions of Article 47b of the Act No. 111/1998 Coll., the Higher Education Act, as amended.

In Prague on June 27, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Seménková, Jitka. *Robust flash memory bootloader for a microcontroller over near field communication.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato práce obsahuje návrh a implementaci zavaděče flash paměti pro mikrokontroler s architekturou RISC-V. Zavaděč přenáší novou aplikaci přes rozhraní NFC, tvořené deskou NTAG5 link. Součástí práce je i návrh a implementace komunikace přes dané rozhraní. Vytvořený zavaděč je optimalizován na velikost a je navržen tak, aby byl robustní. V rámci práce vznikla i aplikace pro Android pro otestování zavaděče.

Klíčová slova Zavaděč, Flash paměť, NFC, RISC-V

Abstract

This thesis includes the design and implementation of a flash memory bootloader for a RISC-V microcontroller. Communication is done over the NFC board NTAG5 link. Design and implementation of the communication are also present in this thesis. The created bootloader is robust, and its memory footprint is small. As a part of this thesis, an Android application was created to test the bootloader.

Keywords Bootloader, Flash memory, NFC, RISC-V

Contents

List of Figures

List of Tables

Introduction

In recent years, the popularity of personal electronics, including wearable devices, has been rising. Many people wear a smartwatch as an extension of their smartphone. Smartwatch shows them notifications and can track their daily activities. There are many brands and models to choose from and the manufacturers need to make an effort to engage customers. The customers are considering multiple factors such as features of the device, compatibility with their other devices, and the length of battery life on one charge is also one of the main parameters they focus on because they do not want to charge the device every day.

The devices need to be updated from time to time to add new features or fix bugs. This is where a bootloader comes into play because it is a part of the device's code, enabling it to be updated on the fly. The bootloader needs to be robust so the devices do not get bricked. The bootloader's footprint also needs to be small, so it leaves as much space as possible to the user application, so there can be plenty of features the customers care about.

The assignment of this thesis comes from a real company, which wants to stay anonymous. The result of this thesis is going to be used in a real device in the future.

The applicability in real life and the opportunity to work on something new are the reasons why I chose this assignment.

The goal of this thesis is to create a robust flash memory bootloader for a proprietary RISC-V microcontroller over NFC, with the usage of NTAG5 link board. The aim is to have a robust solution with a small memory footprint and power consumption.

The thesis consists of 4 chapters. The first chapter includes a summary of all used technologies and a description of the target hardware. The second chapter describes the design of the communication stack and the bootloader. The third chapter describes the implementation of the bootloader and the example mobile phone application. The fourth chapter includes the testing results and their analysis.

Chapter 1

Analysis

This chapter consists of two parts. In the first one, I cover the principle of bootloader and I go through all the technologies that are used in this thesis. In the second part, I describe the hardware for which the implementation will be done.

The company specified the assignment and requested for the NFC communication to be done over NTAG5 link board. It is a NFC tag, that can be connected to a microcontroller over I^2C bus. The mobile phone communicates with the tag over NFC. The microcontroller and mobile phone can exchange data via memory of the NTAG5 link. Figure 1.1 shows the scheme of communication over an NFC tag, like NTAG5 link.

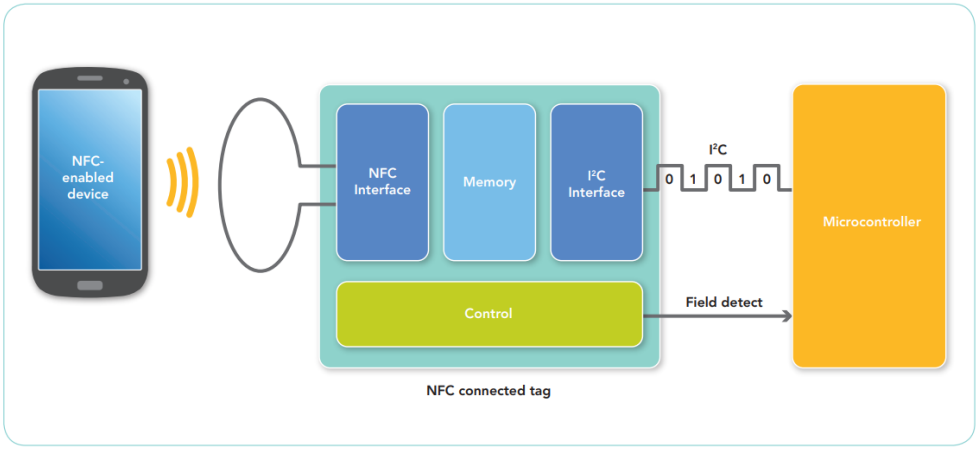

Figure 1.1: Scheme of communication [1]

1.1 Technologies and principles

1.1.1 Bootloader

The term bootloader can have different meanings in different contexts. In this work, bootloader is a part of code that is run on power-up of a microcontroller unit (MCU). It is able to reprogram user application and run it without an external programmer. This is very important in the embedded software, because it enables the user to update the firmware on the fly, which is needed in products, where the application is meant to be updated, fixed or expanded even after the product is in possession of a customer. However, it is also useful for developers because they do not need to use the external programmer at all times and they can test the application in a real environment. Therefore it is usually one of the first things implemented in a project.

1.1.1.1 Bootloader stages

A bootloader is an independent part of code that is run on reset of the controller. Its functionality can be summarized into the following steps:

1. Initialize the environment
2. Check state of the user application
3. Flash new application if needed
4. Run the application

Sometimes the process is theoretically divided into two stages – boot stage (boot manager), which includes steps 1, 2, 4 and load stage (flash loader), which corresponds to step 3. [2, 3, 4]

Possible operation scheme of a bootloader is displayed in figure 1.2 and boot manager internal scheme is in figure 1.3.

The initialization step includes initialization of stack and registers, setup of interrupt vector table, move of selected code section to RAM, setup of watchdogs and clock and initialization of HW drivers. This is partly done in startup code and partly in the actual bootloader code. [5]

After initialization, the bootloader checks if the user application can be run or device firmware update (DFU) needs to be performed. This usually consists of checking if there is a valid application in memory (e.g. CRC check) and if there was a request for an update. [4]

The actual update can be performed in multiple ways based on the options present in a specific circuit. There are two most common options. First consists of preloading a whole new application to external memory or part of internal memory, which is free. The application can be in form of a binary file or a more complex structure that needs to be unpacked or decrypted.

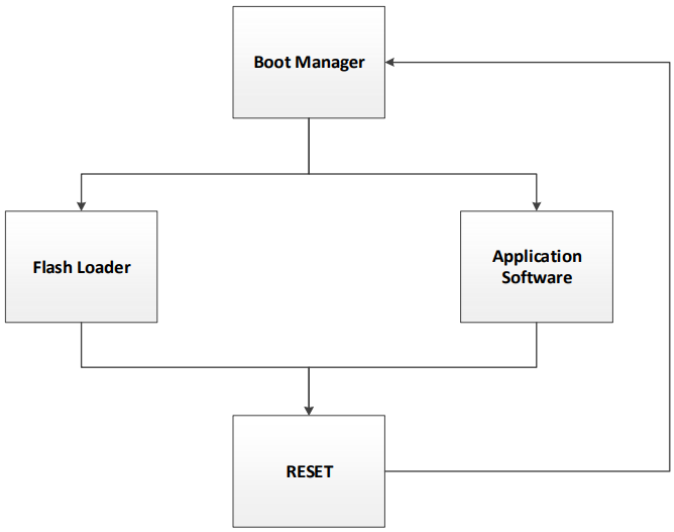

Figure 1.2: Architecture of the flash bootloader[4]

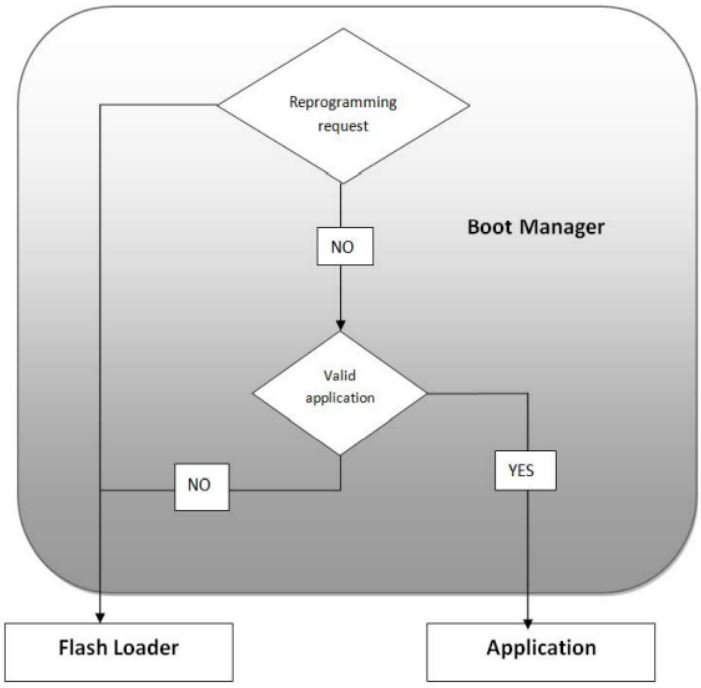

Figure 1.3: Boot manager concept [4] (flowchart starts from the top condition - Reprogramming request)

Finally, the application is copied to the final place at one go, from where it can be run. The second option is to receive the application part by part and

store it in the final place continuously. The first option enables to receive and check the new application before the old one is deleted, so if the transfer of the new application fails, the old one can still run. The second option is required if the memory is too small to store both applications at the same time. [2, 6] There are also special techniques like using only an update file instead of downloading the whole application [7].

The steps, before the application is started, depend on the state in which the application expects the processor to be. If the application has a separate start-up code and it initializes everything by itself, the bootloader needs to deinitialize everything it initialized before starting the app. If the application expects everything to be prepared beforehand, the bootloader needs to take care of that. The bootloader will most probably need to take care of the change of the interrupt vector table in both cases, because switch to application is frequently done by jumping to its reset handler [5].

1.1.1.2 Memory layout

The bootloader is usually located in a different part of memory than the user application, so it can be write protected in the re-programming process. This is important because if the bootloader was overwritten by mistake, the device would get bricked. The bootloader needs to be put to place in memory, so it is automatically run after reset. This location is processor specific and it is mostly defined by the default address of the interrupt vector table (IVT) after reset. Its default address is most frequently at the beginning of the memory. Example, how the memory can look, if the default position is set to the beginning of the memory, is displayed in figure 1.4.

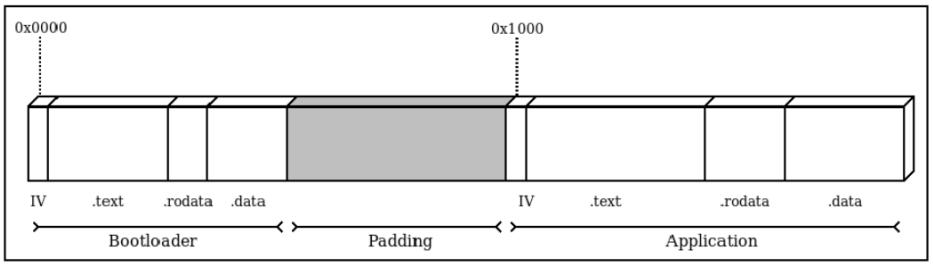

Figure 1.4: Boot manager concept [5] (IV refers to IVT here)

The bootloader, that is stored in flash and also writes the new application to flash, in most cases needs to be at least partly relocated from flash to RAM, because processors mostly do not allow to read flash while erase or write operations are ongoing. Behavior, what happens if a user tries it, depends on a specific processor.

Next important thing is the management of interrupt vector tables. If it is possible for the specific processor to set the location of the IVT on run-time,

it is convenient to have separate vector tables for the bootloader and for the application. If it is not possible, special arrangements need to be implemented. Solution for this single vector table design is presented in article [8]. Also if there is a need to use interrupts in the part of bootloader, that needs to be relocated to RAM, the vector table and handlers also need to be put there. Otherwise the interrupts need to be disabled in this part of code.

1.1.1.3 Communication interface

During the load stage, there is no standard way, how to transfer the code of the new application. Bootloader can use any interface present in the device. The firmware update is divided into two types based on the type of the used interface [9]:

- OTW (over-the-wire)
- OTA (over-the-air)

Even though both options are widely used, the OTA is more discussed recently as the fields that use it are expanding. One of its main advantages is the possibility of an easy reprogramming of boards that are difficult to access. This can be the case for lots of sensors and other parts of smart homes or IoT in general or systems supporting agriculture or in automotive. Another advantage is the lack of need for any connector, which may be needed for some water-resistant designs or it is just used to reduce costs. The OTA option can also be used to update multiple nodes at once. Lately, it is frequently connected to the automotive industry. [10]

1.1.2 NFC

Near-field communication (NFC) technology enables wireless half-duplex data exchange between two NFC-enabled devices in close proximity (around 4 cm and less). The communication uses radio waves and it operates on 13.56 MHz. It was developed by Philips (from which NXP Semiconductors spun-off) and Sony in 2002 and it was built on RFID and smart card technology. The companies together with Nokia founded NFC Forum in 2004. Since then, the NFC Forum promotes usage and covers the standardization of the NFC to ensure compatibility of the devices. Its standards are built on ISO standards. [11, 12]

In NFC communication there are two types of devices based on their actions with the RF field:

- active device - can generate RF field (NFC readers, smartphones)
- passive device - only modulates RF field that it is in (NFC tags)

Active devices always have their own power supply. Passive devices can be powered from the RF field of the active device if the device is small and the field is strong and stable enough (this action is called energy harvesting). [11, 13]

There are two positions in which a device can be in the NFC communication based on who starts the communication. The device that initiates and directs the NFC communication is called initiator. The second device, that answers to this communication, is called target. Initiator always has to be an active device. The target device can be either passive or active. [11]

There are 3 defined NFC communication operating modes [11, 12]:

- reader/writer - used for reading and writing data to NFC tags by mobile phones or NFC readers
- peer to peer - used for data exchange between two mobile phones
- card emulation - mobile phone emulates a smart card while communicating with a NFC reader

NFC tags can be either standalone devices, where they can store a small amount of data in the form of an NFC data exchange format (NDEF) messages, which is transmitted to the active device on tapping mobile on the tag. Or the tag can be connected to an IC and be used as an NFC interface, for communication with active devices. [13]

To this day, there are 5 different types of tag specifications released by NFC Forum. The names are "Type X Tag", where X goes from 1 to 5. Where the 5th one is the youngest, released in 2015. Basic info about each type is in table showed in figure 1.5. The values for memory size and data rates are only informative, and real values depend on a specific device.

Further in this thesis I will pursue only type 5, because it is the type of my target device.

1.1.2.1 Tag 5 type

The tag 5 type is based on ISO/IEC 15693, which was originally developed for vicinity cards. Both specifications (ISO/IEC 15693 and tag 5 type) are available only after purchase.

The standards, among other things, specify commands, that the device must and can support. For example, it includes commands for controlling and manipulation with tag's memory. More information about the commands is included in section 1.2.1.1, which also discusses specific commands supported by the tag, which is used in this thesis.

Communication based on these standards have an integrated CRC. It uses CRC16 to secure every single command.

Mobile phone operating systems have an included support for NFC to be used in the applications. In Android, it is possible to use the Android Java

Property	Type 1	Type 2	Type 3	Type 4	Type 5
Standard	ISO/IEC 14443A	ISO/IEC 14443A	ISO/IEC 18092 JIS X 6319-4 FELICA	ISO/IEC 14443A ISO/IEC 14443B	ISO/IEC 15693
Memory	96 bytes to 2 Kbytes	48 bytes to 2 Kbytes	2 Kbytes	32 Kbytes	64 Kbytes
Data rate	106 kbit/s	106 kbit/s	212 kbit/s, 424 kbit/s	106 kbit/s, 212 kbit/s, 424 kbit/s	26.48 kbit/s
Capability	Read Re-write Read-only	Read Re-write Read-only	Read Re-write Read-only	Read Re-write Read-only Factory- configured	Read Re-write Read-only
Anti-collision	No	Yes	Yes	Yes	Yes
Notes	Simple, cost effective	-	Higher cost, complex applications	-	Vicinity area

Figure 1.5: NFC tag types [13]

package android.nfc. It includes NfcV class, which provides functionality for communication with type 5 tags. Android activity can be notified when a tag is detected, so it can start the communication with it. NfcV automatically adds CRC to every message, so no action from the user is needed. Also if a message is received and the CRC is not correct, it raises an exception. [14]

1.1.3 I^2C brief summary

I^2C is a 2-wire bus. It operates on master-slave principle. By default, it can always be multi-slave and if extra logic is added to masters, it can also be multi-master. Neither option requires adding additional wires like it is for SPI. Both wires are usually connected to pull-ups and nodes operate them by pulling them down via an open-drain output. One wire is used for clock transfer and is operated by the master. The second wire is used for transferring data in both directions. Synchronization is done by using start and stop conditions, which mark the start and the end of one transaction and they are generated by the master. The bus can operate at data rates up to 100 kbit/s (Standard-mode), 400 kbit/s (Fast-mode), 1 Mbit/s (Fast-mode Plus), or 3.4 Mbit/s (High-speed mode). [5, 15]

Each transaction starts with an address frame, which most frequently consists of 7 bit slave address (there is also a 10 bit version) and 1 bit mode select (1 for read, 0 for write mode). Slave with given address acknowledges the frame by pulling the data wire down. [5] Structure of the address frame is displayed in figure 1.6.

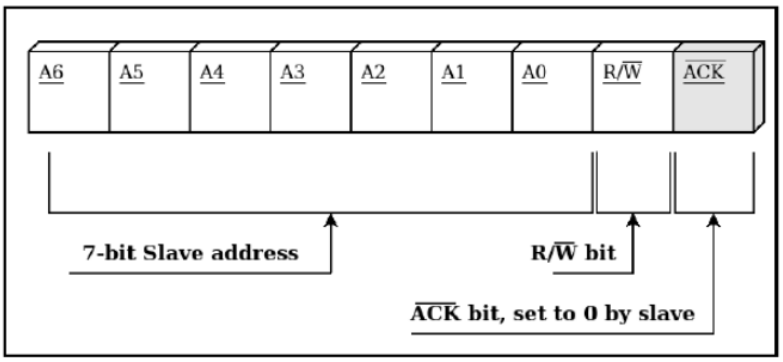

Figure 1.6: I^{2}C address frame structure [5]

The rest of the transaction consists of data frames. The direction of the transfer depends on the mode selected in the address frame. In read mode, the data is sent from slave to master – slave puts data on the bus and master acknowledges them. In write mode, the data is sent from master to slave – master puts data on the bus and slave acknowledges them. [5]

Figure 1.7 includes example of one bus transaction, that sends 1 byte of data from master to slave.

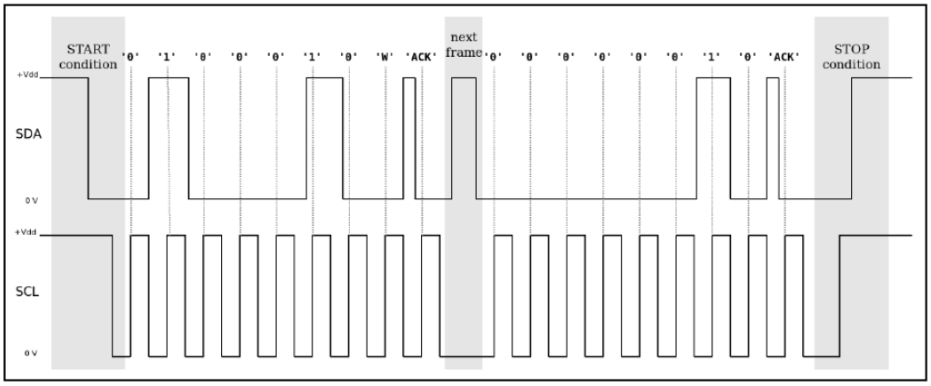

Figure 1.7: Example of I^{2}C transaction [5]

1.2 Given Hardware

In this section, I will describe hardware (HW) for which the program will be developed. First, I will write about NTAG5 link board, which is a NFC tag and then I will describe the target processor, including already implemented modules I will use.

1.2.1 NTAG5 link

Board NTAG5 link OM2NTP5332 is a development kit made by NXP Semiconductors, for designing systems with NFC interface. Image of the board can be seen in figure 1.8. The NFC included is of tag type 5 and is based on ISO/IEC 15693 specification. The board offers multiple wired connection possibilities: I^2C, PWM, GPIO. I will only discuss I^2C interface further in this thesis. [16, 17]

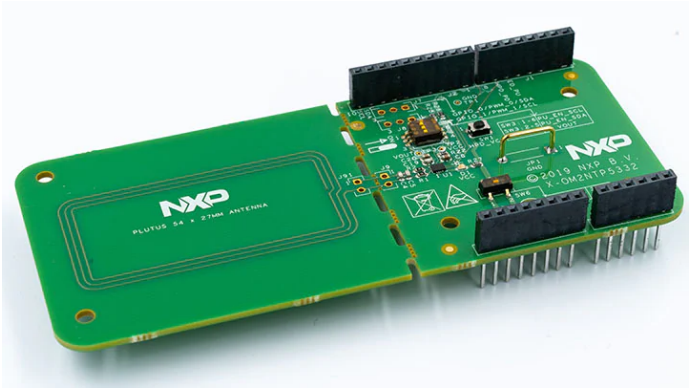

Figure 1.8: NTAG 5 link development board OM2NTP5332 [17]

Table 1.1 includes basic parameters of NTAG5 link.

Table 1.1: NTAG5 link parameters [16]

Parameter	**Value**
NFC Tag Type	5
SRAM memory size	256 B
EEPROM memory size	2 KB
Maximal interface speed	53 kbps
Supply voltage	1.8-5.5 V

1.2.1.1 NFC interface

The NFC interface of NTAG5 link is based on ISO/IEC 15693 and Type 5 Tag norms. It can communicate in close proximity with common NFC-enabled active devices and powerful industrial readers can reach it from a range up to 60 cm. [18]

The base of command set of NTAG5 link is built on mandatory and optional commands defined in ISO/IEC 15693 and Type 5 Tag specifications. On top of that NXP added custom commands to provide better and faster options for specific NTAG5 use cases. [18]

The board can operate either in selected or addressed mode. In selected mode, the initiator sends select command with tag's address mostly only at the beginning of the communication session and then the following commands do not include address of the tag in the header. In addressed mode, the mobile phone needs to include the address in every command.

Memory is writable and readable only by blocks (4 B) from NFC interface.

Table 1.2 includes a list of NFC commands that will be used in the example mobile application. Detailed documentation of each command can be found in [18].

Table 1.2: NTAG5 NFC commands [18]

Command	**Description**
SELECT	Transfer to selected mode
READ SINGLE BLOCK	Read a block from memory
WRITE SINGLE BLOCK	Write a block to memory
READ SRAM	Read blocks of data from SRAM (only in Pass-through mode)
WRITE SRAM	Write blocks of data to SRAM (only in Pass-through mode)
READ CONFIGURATION	Read blocks of data from configuration memory or session register
WRITE CONFIGURATION	Write a block to configuration memory or session register

The maximal length of one command from a phone is 253 B. For the command that writes SRAM, which has 5 B header (selected mode), it only leaves 248 B for data to be written. Therefore if we wanted to use the whole SRAM (256 B) for transferring data, we would need to use at least two separate commands to send the data.

1.2.1.2 I^2C interface

The I^2C interface of NTAG5 board supports both master and slave mode. The master mode can be used for collecting data from sensors without using MCU. In this work there is an external processor, which will be used as a bus master, so slave mode will be used for NTAG5.

NTAG5 link I^2C interface uses standard 7-bit addressing. It can operate on 100 kHz or 400 kHz speed in slave mode. Board's default address is 0b1010100 (0x54). The interface is used to read and write memory or registers. Start and stop conditions and address frame are omitted in the communication description below for better readability, as they are mandatory. Brief summary of I^2C bus communication was done in 1.1.3 section.

The communication scheme for working with EEPROM memory can be seen in figure 1.9. For SRAM, there are only different constraints, but the principle is the same. Reading from memory requires two I^2C transactions. The first one is a write of higher and lower byte of address from where the data will be read. The second one is the actual read of N bytes from memory. The number of bytes to be read is not limited on the side of NTAG5.

Writing memory is handled by one I^2C transaction. The first two transmitted bytes are the higher and lower byte of the memory address, where the data will be written. After that N bytes of data is sent. It is only possible to write exactly 4 bytes to EEPROM. Writing to SRAM is limited to a maximum of 256 bytes at once and the number should be a multiple of 4.

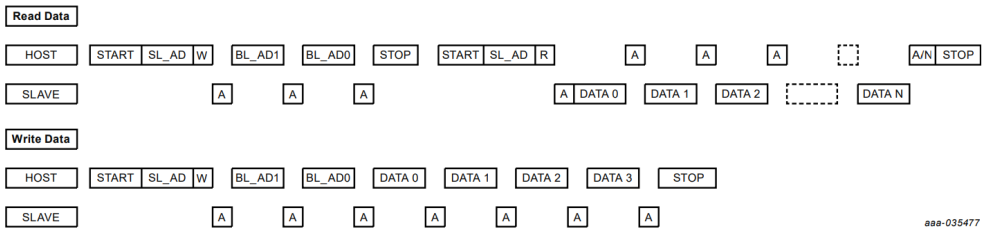

Figure 1.9: NTAG5 link - I^2C read and write memory commands [18]

The communication scheme for working with session registers can be seen in figure 1.10. The interface allows to manipulate only one byte of the register block at a time (one block has 4 bytes). For most operations, this is sufficient and when there is a need to modify more bytes at one time, the time overhead is not that high. Register value reading communication consists of two I^2C transactions. The first one writes higher and lower byte of register block address followed by index of wanted byte in the block. The second transaction is used to actually read the one byte of the register's value.

Writing a value to a register is done by one I^2C transaction. The transaction consists of writing higher and lower byte of register block address, index of the specific byte in the block, mask of bits that will be written from the value and it ends with the actual value to be written.

1.2.1.3 NTAG5 configuration

There are two ways how the NTAG5 can be configured:

- Configuration (memory) setting
- Session registers

The data from the configuration setting is always copied to session registers on power on reset (POR). While the NTAG5 is operating, the configuration

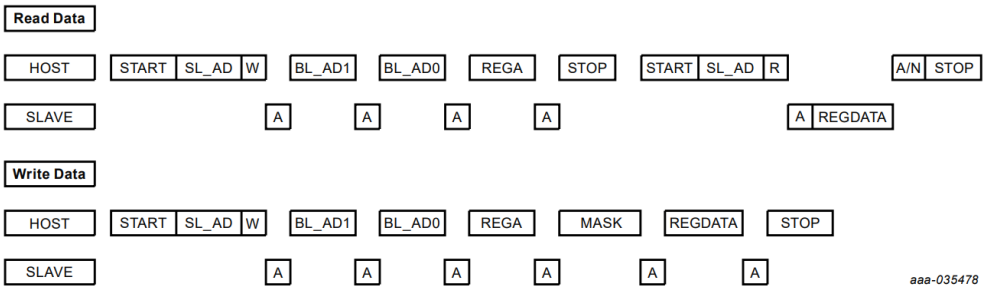

Figure 1.10: NTAG5 link - I^{2}C read and write register commands [18]

is taken from session registers. This means that using configuration setting enables the user to set long-term config that will persist after reset, but will not take effect immediately. On the other hand, setting session registers affects current communication, but will be overwritten on the next reset. So configuration that should be used permanently or at the very beginning of communication, when NFC field is entered, should be set in configuration setting and any other configuration should be set in session registers. A good example of what needs to be set in the configuration setting is the use case config - operation mode of the NTAG5 board (in this work, it is I^{2}C slave option). Also the security information should be set there. [18]

The configuration memory can be set by READ CONFIG and WRITE CONFIG from NFC interface and by memory manipulations from I^{2}C interface. The session registers can be also set by READ CONFIG and WRITE CONFIG from NFC interface, but from I^{2}C interface the register manipulations are needed instead of the memory ones. Note that I^{2}C interface addresses memory (starting from 0x1000), where NFC interface addresses only blocks in the commands (0x00 is the first block). [18]

1.2.1.4 Event detection pin

NTAG5 link board includes an event detection (ED) pin, that is used for signaling external devices that an event has occurred. The tracked event can be chosen by modifying bits 0-3 in ED_CONFIG (configuration memory) or ED_CONFIG_REG (session register). The ED pin has active low implementation (ON = 0, OFF = 1). [18]

Table 1.3 contains possible configurations of ED pin, that are important for this thesis.

1.2.1.5 NTAG5 link communication modes

This section describes different communication modes that are available on NTAG5 link for bidirectional communication between the NFC interface and

Table 1.3: NTAG5 event detection pin configurations [18]

Name	Value	State	Description
Disable ED	0000b	OFF	Event detection pin disabled
NFC Field detect	0001b	ON	NFC field present
		OFF	NFC field absent
I^2C to NFC pass-through	0011b	ON	Last byte of SRAM data has been read via NFC; host can access SRAM again
		OFF	Last byte written by I2C, or NFC off, or VCC is off
NFC to I^2C pass-through	0100b	ON	Last byte written by NFC; host can read data from SRAM
		OFF	Last byte has been read from I2C, or NFC off, or VCC off
Arbiter lock	0101b	ON	Arbiter locked access to NFC interface
		OFF	Lock to NFC interface released

the slave I^2C interface. The modes differ in the memory they use and the style of arbitration, which interface can access the memory. Some of the modes support extra commands.

NTAG5 link can operate in 4 communication modes for exchanging data between NFC and slave I^2C interface [19]:

- **Normal Mode** - EEPROM is used for main data storage, but the SRAM can also be accessed if requirements are matched. The memory access arbitration is based on first come first serve principle.
- **SRAM Mirror Mode** - the same as normal mode, but SRAM is mirrored on EEPROM (EEPROM underneath is inaccessible), and it can be used for functionalities restricted for that area (NDEF messages)
- **SRAM Pass-Though Mode** - SRAM is used as the only data storage. There are two parts of arbitration - the direction setting (changed manually by writing to session registers) and the interface lock (changed automatically based on reads and writes to the last data block). It also includes additional commands for reading and writing multiple blocks of data at once from NFC interface, which dramatically affects the time needed to transfer bigger data.

- **PHDC Mode** - special mode for Personal Health Device Communication - similar to normal mode, but special commands are supported

Pass-Through mode allows the smoothest data transfer for an application, where a large amount of data is sent in one or both directions. This is the case for the bootloader, where new application needs to be loaded. This is mostly caused by the ability to write multiple blocks of memory at once from the NFC interface, because this has a big effect on the transaction time.

As mentioned above, the Pass-through mode synchronizes the devices behind the NFC interface and I^2C interface by default. For the Normal and SRAM mirrored mode, it is possible to synchronize the devices by using signalization on SYNCH_BLOCK. NTAG5 detects and remembers when the synchronization block has been read or written. This is somewhat similar to Pass-Through mode detection on the last memory block, but the address of the block can be specified on runtime by modifying SYNCH_DATA_BLOCK register and the devices need to implement the arbitration mechanism themselves.

To be able to use Pass-Through mode or SRAM Mirror Mode, it is required to have the NTAG5 VCC supplied. This is required so that SRAM memory can be used. The same goes for using SRAM in Normal mode. [19]

Example, how the manufacturer suggests to use the Pass-through mode in document [19], is present in appendix A.

1.2.1.6 Example applications

NXP Seminductors developed example applications for NTAG5 usage. There is a set of an embedded project and an android app. Together they can demonstrate different use cases for the NTAG5, like Pass-Through mode, GPIO, PWM and I^2C master mode. The applications in the embedded project are designed for NXP's FRDM-KW41Z board. [17]

Authors in [20] describe the different features mainly from the view of the android app.

There is also an iOS version of the example application. There are two Windows applications. One enables the user to configure NTAG5 from NFC interface with an NFC reader connected via USB to the PC. The second one enables the user to configure NTAG5 from I^2C interface via USB-I2C bridge. [17]

Only the first set is interesting for this thesis, and especially the android app, which I will use as a base for the testing application.

All the applications are available from [17].

1.2.2 Target processor

The bootloader will be implemented for a proprietary processor with RISC-V architecture. The company, which gave me the assignment, wishes for the

name of the processor to remain hidden, so I will refer to the processor as "the target processor".

The target processor is still in development. I am using the processor emulated on an FPGA and once in a while, I receive an updated bitstream. This means that some things are still changing, while I am working on this thesis, and there are some things that do not work correctly. This makes the work harder than it would be if the processor was done and its specification was complete and not changing.

Basic information about the target processor is included in table 1.4.

Table 1.4: Target processor parameters [21]

Parameter	**Value**
Architecture	32b RISC-V
Clock frequency	8 MHz max
Supply voltage	1-3.6 V
SRAM memory size	8 kB
Flash memory size	128 kB

The processor is designed for usage in ultra low-power IoT projects.

1.2.2.1 RISC-V ISA

RISC-V is an instruction set architecture (ISA) with an integer base of 32b or 64b. There are multiple prepared extensions and the ISA is opened for customization and specialization. [22]

The RISC-V ISA base includes instructions for fundamental operations with integers (load, store and computation) and code flow-control instructions. Standard extensions add integer multiplication and division operations, single and double precision floating-point operations and atomic operations. [22]

Names of RISC-V implementations are based on the base and extensions that are included. The name starts with ether "RV32" or "RV64" depending on the integer register width and followed by "I" standing for the base. Then the one-letter identifications of included extensions are added. [22]

The target processor is of type RV32IMC. It is built on a 32b base with addition of standard extension of compressed instructions and partial support of integer multiplication and division. [21]

1.2.2.2 Flash memory

Flash memory of the processor includes two sections. There are 128 kB of user memory and 4 kB of flash is dedicated for configuration data. The flash's sector size is 4 kB and row (line) size is 128 B. This means that one sector consists of 32 rows and the whole memory has 32 sectors (1024 rows). Lines are

the minimal unit for erase operations and the only unit for write operations. [21]

The memory can be programmed either from JTAG interface or by in-application programming, where the data is received from any other present interface. [23]

The architecture expects that the bootloader will be located in the first N sectors at the beginning of the flash memory (N goes from 0 to 32) [21]. There is no particular functionality based on this except for the mass erase function (viz. 1.2.2.3).

Figure 1.11 shows the flash memory scheme of the target processor.

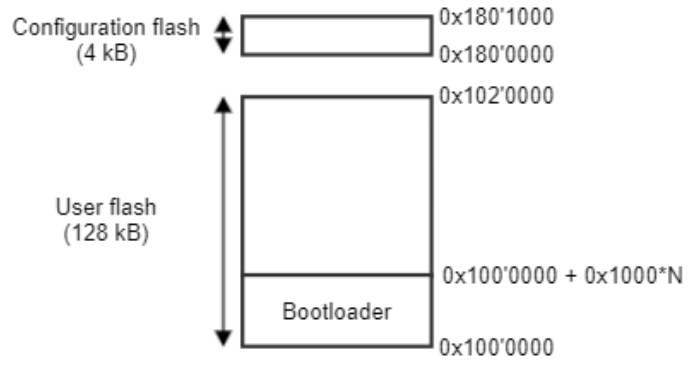

Figure 1.11: Flash memory scheme of target processor

Table 1.5 includes times of flash operations. These times are just an estimate to be corrected once the real version in silicon is characterized. Measured times for current configuration will be included in section 4.3.

Table 1.5: Maximal times of flash memory operations on the target processor [21]

Operation	**Time**
Memory erase (mass, sector or line)	4-10 ms
Line write	3.5 ms

1.2.2.3 Flash driver

The driver is required for erasing and writing the memory. It is not needed for reading operations. The flash memory is not accessible for reading while erase or write operation is being executed, so it is mandatory to move the flash driver and any other code that should be executed in the meantime from

flash to SRAM. Any attempt to access the flash will stall the processor until the operation is done. [23]

The flash driver has been implemented by the MCU manufacturer before this thesis. That is why the implementation is not included in it. However, I was actively involved in testing and debugging the driver and I have made some minor changes in it, so it could be included in the project.

The driver implements following operations for the flash memory [24]:

- **Locking and unlocking the memory** - Locking the memory prevents an unwanted modification or corruption of the memory. It is essential to unlock the memory only for the time necessary to perform erase and write operations.
- **Mass erase** - This function is meant to be used for erasing the whole memory except for sectors where the bootloader is located. It accepts an address of the first block, from where the erase will be performed.
- **Sector erase** - This function is used to erase a single sector (4 kB) of the memory.
- **Line erase** - This function is used to erase a single line (128 B) of the memory.
- **Line write** - This function writes data to one line (128 B) of the memory. It is required to write only previously erased memory.
- **Waiting for an operation to end** - The erase and write functions mentioned above only order the HW layer below to perform the specified operation. This function is used to wait until that operation is done. The user can specify a timeout for this function.

A rewrite of a part of memory is meant to be performed by executing following steps:

1. Unlock the memory
2. Erase a part of memory
3. Write all needed lines
4. Lock the memory

It is possible to split the erase and the write operations or not to perform all the write operations at the same time, but in that case, it is needed to lock the memory in between them.

It is necessary to avoid reset while the erase or write operation is ongoing (watchdog should be kicked, software reset should not be called, etc.).

The state of the battery should be checked before performing erase or write operation and voltage monitoring should be enabled. If one of the operations fails due to a voltage drop, it is necessary to erase the part of memory to which the operation was connected. [23]

1.2.2.4 I²C driver

The I²C driver provides API for operating I²C. The processor can act only as a master. It can be either used with busy waiting on every operation or with interrupts. DMA support has not been added yet. [21]

I²C interface in the target processor supports 7b addressing and data rates up to 100 kbps, 400 kbps, or 1 Mbps (requires an external pull-up resistor). The I²C controller includes a 2 B FIFO for storing received bytes or data to be sent. The direction of the communication is automatically assumed from the address byte. [21]

The I²C driver API includes [24]:

- Initialization of the bus - setting up the environment, including the pins
- Enabling and disabling of the bus
- Starting an operation - start condition and sending address frame + setting data counter (number of data that will be sent/receive in this operation)
- Receiving a byte - get byte from FIFO or try to wait until timeout if it is empty
- Send a byte - put byte to FIFO or try to wait until timeout if it is full
- Wait for a stop condition

1.2.2.5 Interrupts

The mtvec register in RISC-V stores the base address of the interrupt vector table. The possibility of changing its value depends on implementation. [25]

The target processor enables to change the address contained in mtvec and it can even point to SRAM. This means that the whole interrupt handling can be moved to SRAM. This can be needed if the processor needs to be able to continue working while a flash memory erase or write operation is performed. [21]

Chapter 2

Design

In this chapter, I go through the design choices that I needed to make.

Section 2.1 describes the design of the bootloader. First, I describe how the bootloader will work in general. Then I show and describe the memory layout. Finally, I introduce the operation workflow of the bootloader, except for the DFU part that I will describe in section 2.3.

Section 2.2 includes design of the communication stack. In there, I describe which mode will be used for NTAG5 link board and how the communication will work over this channel.

Section 2.3 introduces the design of the DFU operation scheme. It also describes how the communication parameters affect this design.

Section 2.4 introduces the messaging system that will be used between the bootloader and the mobile phone. It includes a list of commands and sequence diagrams of the communication.

2.1 Bootloader

The design of the bootloader in this thesis is strictly dependent on the target processor and its properties. The most limiting factor is small memory size. The processor has only 128 kB of flash memory and 8 kB of SRAM. This affects the functionalities that the bootloader can have (the goal is to have a small bootloader to leave space for user application) and also the process of how the bootloader needs to operate. The second crucial thing is the request for the bootloader to be robust. This means the bootloader (or the user application) must not include operations that could potentially corrupt the bootloader (for example, if the operations failed or were interrupted).

2.1.1 DFU principle

The bootloader created in this thesis will be able to update the user application only. It will not be possible to update the bootloader on-the-fly, because it

is not possible to do it safely without using a multilayer bootloader (the first code that is executed after reset needs to be always fixed).

The new application cannot be pre-downloaded as a whole due to lack of memory space (external flash is not available at all). Because of this, the update of the software is done only in the bootloader. If the pre-downloaded could be done, it could be done while the old app was running. Because it is not possible, the DFU in bootloader will be done in two steps:

1. Clear the memory (delete the old application)

2. Receive new application part by part and write it to flash on the fly

2.1.2 Program flow

The bootloader could theoretically be built on a scheduler. This may seem like a good idea at first, that it would be easier to communicate and operate the flash at the same time, but it is really not necessary. It is even counterproductive because the bootloader should be as simple as possible to minimize its memory footprint and to reduce the space for error (as said, the bootloader will not be updatable on the fly). So the bootloader will work sequentially, just with the use of interrupts.

2.1.3 Flash memory layout and interrupt vector tables

The flash memory layout that will be used in this thesis is displayed in figure 2.1. The bootloader will be placed at the beginning of the user flash memory. It is already defined like this in the specification of the processor. There are two reasons why this was already specified there.

The first one is that the flash controller has a mass erase function with "protection" of the first N memory sectors (4 kB). This also means that the bootloader needs to be aligned to sector size and the user application cannot be placed closer than in the next sector.

The second reason is connected to interrupt vector tables. The bootloader and the user application will each have their own IVT. This enables the user to compile the codes separately. It is possible to have multiple IVTs, because the processor enables the user to specify the address of IVT in mtvec register at run time. Upon reset of the processor, the value of mtvec is always set to the beginning of flash memory (0x1000000), so the bootloader IVT has to be placed there. Once the bootloader is in the stage when the user application should be started, it sets the mtvec to address of the application IVT and it jumps to the application's reset handler. Only these two operations with mtvec (using value after reset and set upon app start) are sufficient for the bootloader to work. Analysis, why it not necessary to also have an IVT in SRAM, is done in section 2.3.

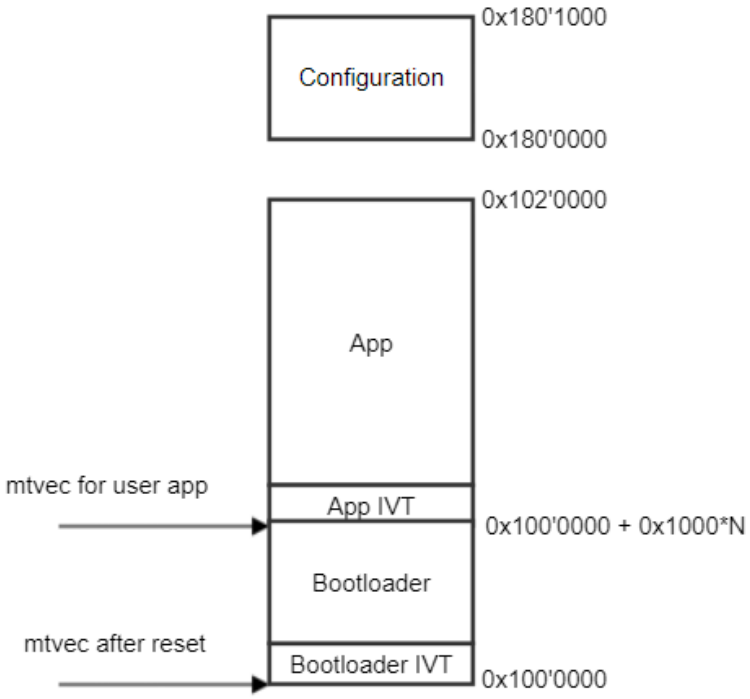

Figure 2.1: Flash memory layout

2.1.4 Bootloader operation scheme

In this section, I will go through bootloader operation design. The first two paragraphs include a brief overview and the following text includes detailed description.

The bootloader is always run on reset of the processor. It checks if the DFU needs to be performed (there was an external request for DFU, or the CRC of application in memory does not match the control CRC stored in configuration memory) and if so, the bootloader performs it. After a successful DFU, the memory includes a valid app and its CRC is stored for future checking to the configuration memory. Once the memory includes a valid app, the bootloader can run it.

A standard request for a DFU is made through the user application. If the user application knows the DFU needs to be performed (e.g. it received a command to do so), it forwards the information to the bootloader by invalidating the CRC in configuration memory, and it resets the processor. After the reset, the bootloader is started and it thinks the application in memory is not valid, so it performs the DFU.

Figure 2.2 displays scheme how bootloader is operating. The bootloader is

started after each reset of the processor. First, it initializes the environment ("Start bootloader"). Then it checks if there is a valid application in memory. This will be done by computing its CRC32 and comparing it to value that has been stored after the last DFU to the configuration flash.

If the CRCs do not match, the bootloader knows that the DFU has to be performed and it switches to it ("Perform DFU"). The second option how to trigger the DFU is a DFU request from outside. This is definitely needed in case when there is a valid app in memory, but it is not able to invalidate the CRC for any reason (e.g. a bug causing reset before it can do it). However, this option is project specific and it will need to be implemented based on the needs of the particular project. It can be, for example, done by sniffing on a pin or pins for a given amount of time or by checking the NFC interface, if there is not a message requesting DFU.

If the CRCs match and there was not a request for DFU, the bootloader starts the application ("Start app"). It is necessary to deinitialize everything before the application is started because the application performs its own initialization. Then the bootloader changes the value in mtvec so that it points to the application interrupt vector table (interrupts should be disabled at the time). And finally, the bootloader jumps to the reset handler of the application.

Operation of user application from bootloader point of view is displayed in figure 2.3. It starts when the bootloader jumps to the application's reset handler ("Start user application"). With this, the application becomes autonomous. Its startup code initializes everything just as if there was no bootloader. Apart from normal functionality, the app is expecting that in future, it will receive a request for DFU. This can be, for example, a request from mobile via the NFC interface. Once the request is received, the application assesses if there are no obstacles, why the DFU cannot be performed at this moment. If everything is ok, the application can prepare for shutdown by saving data and such. Once it is ready, it invalidates the CRC value stored in configuration flash to let the bootloader know DFU needs to be performed, and it restarts the processor to start the bootloader ("Switch to bootloader").

2.2 Communication stack

In this section, I will describe the design choices I made while creating the communication stack between mobile and target processor over NFC, with the usage of NTAG5 link NFC tag. The communication stack will be used by the bootloader but also by the user application to unify the communication. In the bootloader, the main goal of the communication is to transfer data of the new application (up to 128 kB) to the processor. This will require a large number of long messages, but also some short control messages. On the other hand, the application will probably mostly use shorter commands to get status

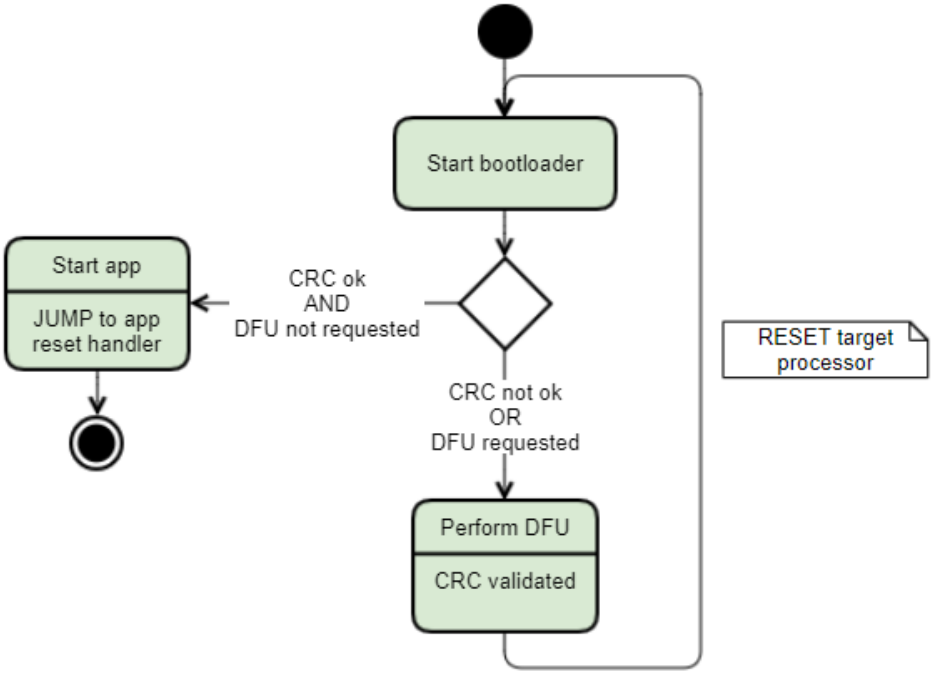

Figure 2.2: Bootloader operation scheme

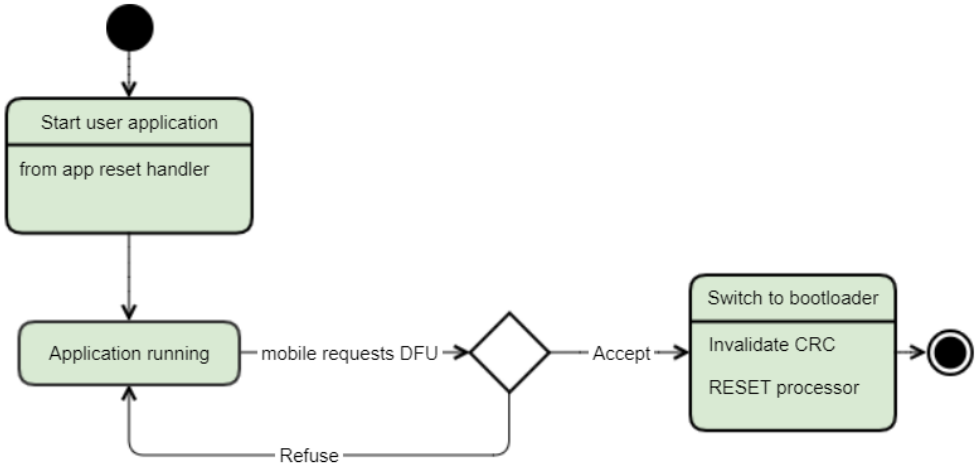

Figure 2.3: User application operation scheme from bootloader point of view

or set some value, but sometimes it may also need to transfer a larger amount of data sometimes (e.g. log data). Therefore the communication stack will need to be flexible to support sending short and long messages both ways.

2.2.1 Communication channel

The communication channel between mobile and target processor consists of two parts.

- NFC part - between mobile and NTAG5 link
- I^2C part - between NTAG5 link and target processor

In between these parts, the data will be stored in SRAM of NTAG5 link. NTAG5 link also supports EEPROM for this job, but it is not suitable for the purpose of sending a large amount of data. The scheme of the data paths is displayed in figure 2.4.

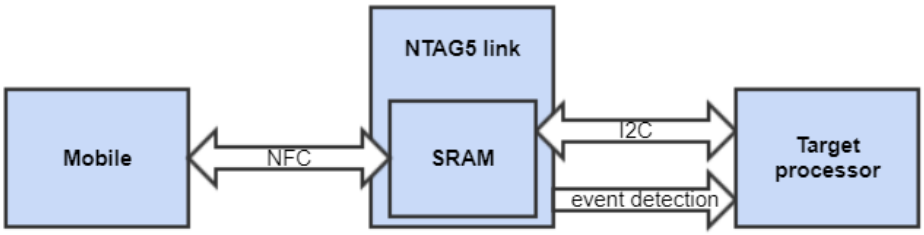

Figure 2.4: Scheme of communication channel between mobile and target processor

2.2.1.1 NTAG5 mode selection

NTAG5 link offers two modes that could possibly be used with SRAM in the middle: the SRAM mirrored mode and pass-through mode (description of both of them was done in 1.2.1.5 section). If we only needed to send short messages, ideally with a fixed length, the mirrored mode would be fine. For a larger amount of data, it is not ideal because it does not support command for writing multiple blocks of SRAM memory at once, which greatly reduces transfer time for this use case.

This command is only supported in Pass-through mode, which is the main reason why this mode will be used. Another specification of this mode is that developer needs to control the direction of communication. This setting needs to be done from I^2C side. Once the direction is set, the board automatically switches interface locks, so only one interface can manipulate the data in memory. The lock is switched on every interaction with the last block of SRAM (write or read of this block). This strict arbitration can be used as an

advantage here. Both sides can check the state of the communication anytime, and they do not need to do any special synchronization.

2.2.1.2 NTAG5 SRAM usage

NTAG5 link has 256 B of SRAM memory divided into 64 blocks (one block has 4 bytes).

From the NFC side, it is only readable and writable by blocks. Write SRAM (respectively Read SRAM) command contains a specification of how many blocks should be read. It is defined by one byte, so it could theoretically write (respectively read) the whole SRAM. However, the NFC interface of each device limits the maximal length of a message. For the NTAG5 link, it is 253 B. The Write SRAM command in selected mode has a 5 B header, so it leaves 248 B (62 blocks) for data transfer. The Read SRAM command can return up to 252 B (63 blocks) of data at once.

From the I^2C side, it is limited by I^2C driver in the target processor, which can handle only 256 frames per operation. With taking command structure into account, 255 B can be read and 252 B written at one operation.

It was necessary to choose an address where the mobile and target processor should expect the beginning of the message. There were two options after putting together length limits of read and write commands from both interfaces:

- start from block 0x02
  - 248 B available
  - big data writable (readable) with one command (suitable for long messages)
  - messages will not start at the beginning of memory
- start from block 0x00
  - 256 B available
  - even long messages need 2 commands to be written (read)
  - messages will start at the beginning of memory

If we wanted to use the whole SRAM, we would need to always use two read and write commands from each interface. Unfortunately, the NFC interface has a quite large overhead for each command. Times measured in initial NFC speed testing I have done are in table 2.1. The times were measured on the event detection pin of NTAG5 link. [1]

[1] Because in the final implementation, the exchange between using only 248 B or the whole 256 B can be done very easily, I was able to measure the difference this choice made in the end. Results of the measurement are included in chapter "Testing".

Table 2.1: Initial NFC write SRAM speed testing with Samsung A21S

Data length	Transfer time	Transfer speed
4 B	23.6 ms	0.17 B/ms
248 B	137.2 ms	1.81 B/ms
256 B (2*128 B)	155.1 ms	1.65 B/ms

In the implementation, I will use the option with using only 248 B of NTAG5 link's SRAM. It enables faster transfer of data and the fact that the data does not start at the beginning of the memory is not crucial even for debugging because the memory is not readable directly.

The operations on the last block of SRAM are monitored and they automatically switch the lock between the interfaces. To be able to control the flow of the data properly, even for long messages, the last memory block will not be used for transferring data but only for manually controlling the switch. This will not be used on the NFC side because it would only prolong the time of the transfer, but from the I^2C side, it is needed to be able to switch the direction of communication safely. Without this feature, the message would be written till the end of the memory. Right after the processor would read the whole message, the interface would automatically switch. It would be recognized by the mobile, which would immediately start to write the next batch of data without giving the processor opportunity to switch the direction and respond.

Figure 2.5 shows the diagram of usage of SRAM memory blocks of NTAG5 link.

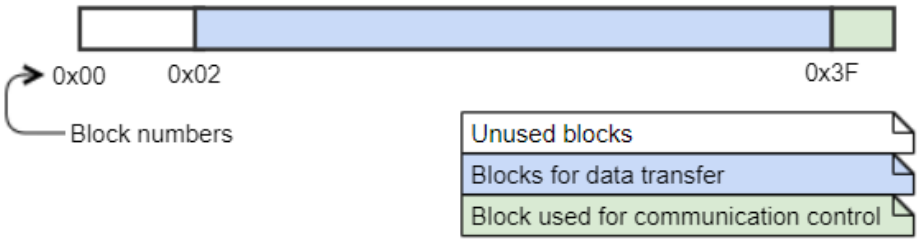

Figure 2.5: Diagram of usage of NTAG5 link SRAM memory

2.2.2 Messaging system

The task requires to have the possibility to send messages with a variable length in both directions. The setting of direction needs to be handled by the target processor. To be able to send messages with a variable length, there needs to be information about their length in the messages. This can be done

by using the TLV (type-length-value) encoding scheme. The message with this encoding consists of 3 fields: type, length and value. Type and length fields have fixed sizes, which need to be set according to the needs and possibilities of the communication stack. The size of the value field is variable and it is specified in the length field. Type field holds information about what data is sent in the value field. The type field can also be thought of as an opcode if the messages are principally commands.

Given the communication channel, which only supports messages with up to 244 B, it is sufficient to have the length field of 1 B. After evaluation of the potential use cases, it will be sufficient to use 1 B for the type field. The bootloader will consume around 10 opcodes. If there was a use case in the application where a larger number of opcodes would be needed, it would be possible to use nested NFC. For the bootloader, it is not necessary and it would be inefficient due to bigger footprint of needed parser.

The structure of the TLV message is displayed in figure 2.6. The position of type and length fields at the beginning of the message can be used for more time efficient communication. It is possible to just read and parse the first two bytes, from which we have the information about the length of the rest of the message and then we read just the necessary bytes. This is much better than reading the whole memory at all times. If an interface has a big overhead for reading operation, it can be beneficial to read more bytes in the first message to cover the majority of small messages to avoid the need for a second read. This is not the case for I^2C, but optimization for NFC in the future mobile application could make a big difference.

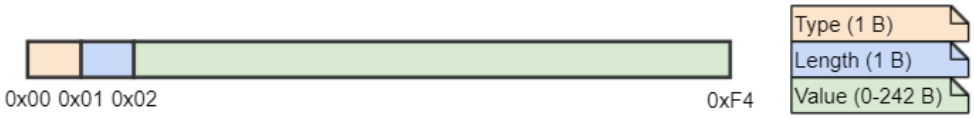

Figure 2.6: Structure of TLV messages

For creating the messaging architecture, it is necessary to establish which device will play what role. In this case, the mobile will be master and the target processor will behave like a slave. The mobile phone is the initiator of actions because it is operated by the user (or some other external force). Also, it will help reduce the power consumption of the processor and the footprint of the code. It will have a set of commands for reading (writing) information from (to) the processor and controlling it. The command set will need to be designed and implemented, so the communication won't break if it is interrupted.

2.3 DFU operation scheme

This section includes the design choices for the actual software update over NFC. It corresponds to "Perform DFU" part from bootloader operation scheme displayed in 2.2 figure. In section 2.1.1 I've already established the general idea for DFU. That is that for this processor, it is needed to erase the old application first to make room for the new one. After that, the new application can be received and written to flash.

DFU operation scheme is in 2.7 figure. Apart from the scheme, how it works, it also includes information, what parts of code will need to be in SRAM and what parts can stay in flash memory. This decision is for implementation, where the application data will be sent via NFC, is based on the time analysis included below. Like this, only parts that directly operate flash memory (write and erase operations) need to be in SRAM. If another potentially faster interface was used, this would need to be reevaluated.

The DFU starts with waiting for the initial message that will start the DFU. Right after that, erase of the whole user flash memory except for bootloader sectors will be done (for flash memory, it is always needed to erase the memory before writing to it). After that, the bootloader will be receiving data and once it has enough for one flash row, it will write it. This will be done until the whole application has been received and written. Once it is done, it is needed to check the CRC of the application, if the data stored in flash memory correspond to data that has been sent. If the CRC is correct, it will be stored to configuration flash along with the length of the application, so in the future, it can be used by the bootloader to verify if there is a valid application in memory. After a successful DFU, the bootloader will reset the processor, and the bootloader will be run from the beginning. Using restart has the advantage of also deinitializing automatically everything that needed to be initialized for the DFU but needs to be deinitialized before the user app is started.

The code, that needs to be run, while erase or write operation is happening in flash memory, needs to be put to SRAM, for the processor not to get stalled uncontrollably. Any other part can stay in flash memory, which is the preferred version to save SRAM, even though the user application does not share SRAM with the bootloader. Times relevant to planning the code memory layout are summarized in table 2.2. NFC transfer time was measured with Samsung A21S.

Each of the three parts "Erase memory", "Write data" and "Write CRC" needs be in SRAM. They will be built on two part strategy: initiating flash controller operation and waiting until the operation ends. Both "Erase memory" and "Write CRC" are only one time operations, so there is no need to stress over trying to do something else while the operation is running. On the other hand, for "Write data" it could potentially be beneficial to also be able to execute other code to speed up the process. The following discussion is

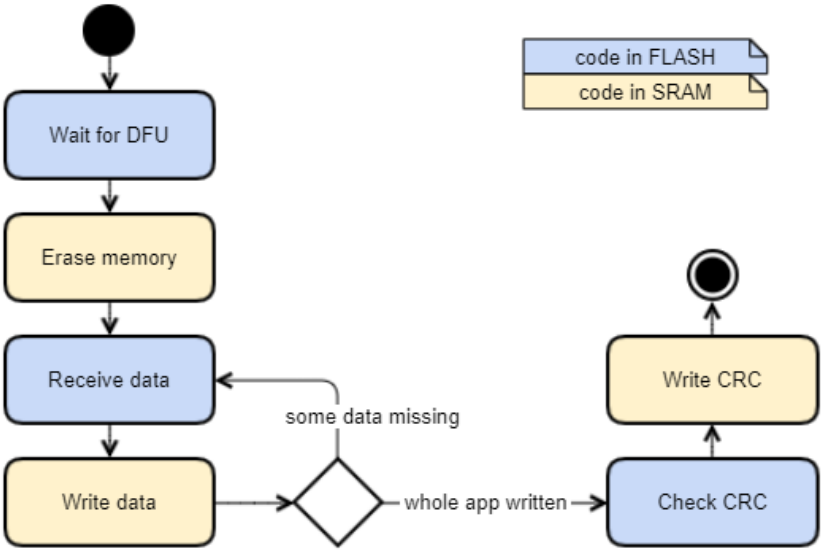

Figure 2.7: DFU operation scheme

Table 2.2: Times of operation needed for time analysis

Operation	**Time**
Memory erase	4-10 ms
Line write	3.5 ms
Transfer of one data batch via NFC (242 B)	135 ms
Transfer of one data batch via I^2C (242 B)	8 ms

graphically illustrated in figure 2.8. With knowledge of the communication interface, I know I will be able to send 242 B of data at once. The flash memory row has 128 B. This means most of the time the "Write data" part will consist of two write row operations. Once in a while, only one will be ready. Once the data is received from NTAG5 board it can immediately start receiving data from the phone without any intervention from the processor. Because of this, the write of the data to flash and transfer of next data over NFC can be done simultaneously. The NFC transmission takes a much longer time in oppose to write times of flash memory. Because of this, there does not need to be any discussion, whether it would be worth performing the I^2C communication from SRAM to be able to do it in parallel with the flash writing. This also eases the interrupt handling because interrupts can be disabled in SRAM, so we do not need new IVT and separate handlers in SRAM.

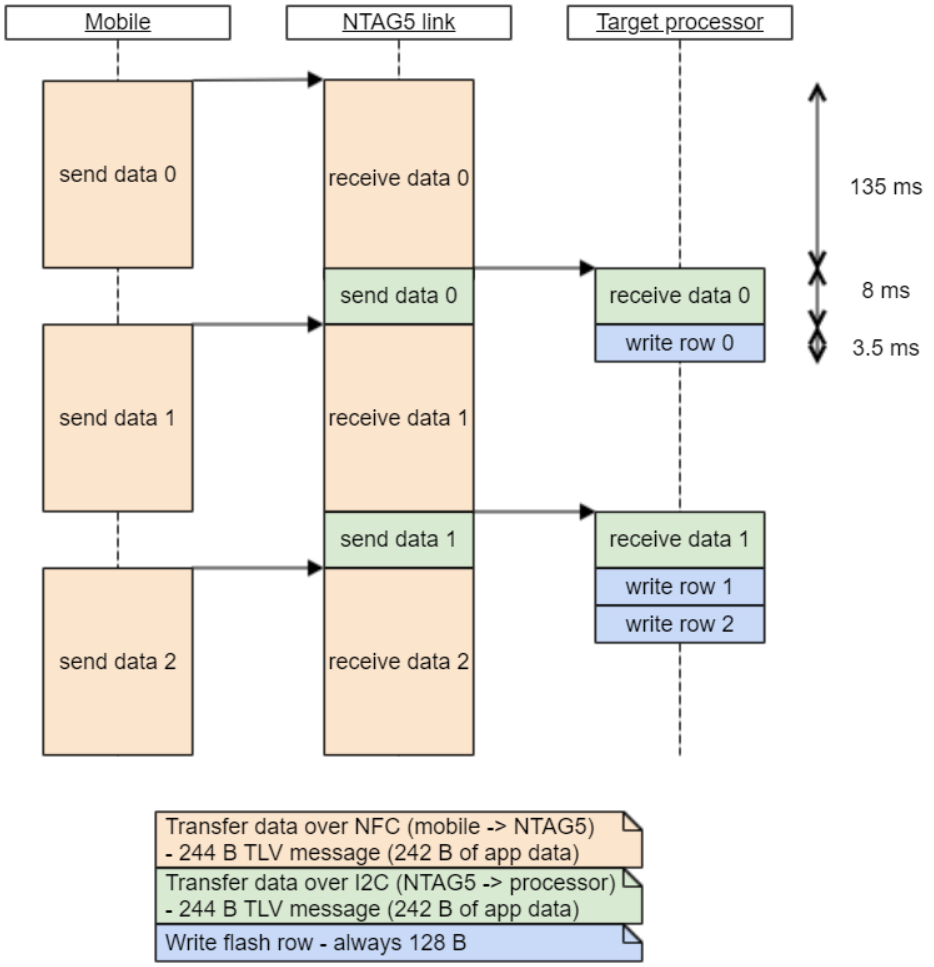

Figure 2.8: Data manipulation diagram for time analysis

2.4 Bootloader communication

I already described how the communication channel will look, so the only thing missing is the list of commands that will be used and how the communication will look like.

Table 2.3 includes a list of commands that will be exchanged between the bootloader and the mobile phone. For each command, there is its name, its description and indication, who will send the message. Even though implementation of user application will not be included in this work, I added commands for communication regarding firmware update even for application because it is part of the designed solution.

Any time the two devices come together (NFC connects), the mobile phone needs to ask the processor for its state. This is needed because the mobile

Table 2.3: Command set for bootloader communication

Command	**Sender**	**Description**
GET_STATUS	M	ask processor for status
SEND_STATUS	P	send current status (response to GET_STATUS)
START_DFU	M	tell bootloader to start DFU (tell application to switch to bootloader)
DFU_ACK	P	response to START_DFU - accept or refuse
SEND_DFU_DATA	M	send part of new application
SEND_CRC	M	send CRC
CRC_ACK	P	response to SEND_CRC
START_APP	M	tell bootloader to start application after successful DFU
UNKNOWN_COMMAND	MP	last command was not recognized
INVALID_COMMAND	MP	last command was corrupted
CANCEL_DFU	M	tell bootloader to cancel current DFU
GET_VERSION	M	ask application for its version
SEND_VERSION	P	send version of app (response to GET_VERSION)

phone cannot know if the processor did not restart or change its state from the last communication. The processor needs to be able to respond to the "GET_STATUS" command with "SEND_STATUS" the whole time the NFC communication is available. This will arrange that the devices can always synchronize. The content of the value field of the "SEND_STATUS" command may change for each status because sometimes the processor will need to specify the state in more detail.

The sequence diagram in figure 2.9 shows the communication, regarding firmware update, between the mobile phone and the applications running on the target processor. The diagram starts when the mobile get close to the NTAG5 and the NFC communication starts. A user application is running at the target processor. The first thing the mobile does, it to ask the processor for its status ("GET_STATUS"). The processor responds with "GET_STATUS" with value indication there is a user application running. The mobile asks the processor what version of the user application is running ("GET_VERSION") and the processor responds with "SEND_VERSION" command. The mobile

shows the info to the user, which can choose a new application and click the "Start DFU" button, after which the mobile sends "START_DFU" command to the processor. In a real case, the mobile application would probably receive info, that there is a new application available from the server, it would check the currently present version and offer the user to install the new version (the user cannot select the application). The user application evaluates if it can end and it accepts or declines the request ("DFU_ACK"). If it can, it finishes its work, invalidates CRC and resets the processor to switch to the bootloader. The bootloader recognizes the need for DFU and performs it (the DFU communication is in a separate diagram). After the DFU is successfully done, the bootloader starts the new user application.

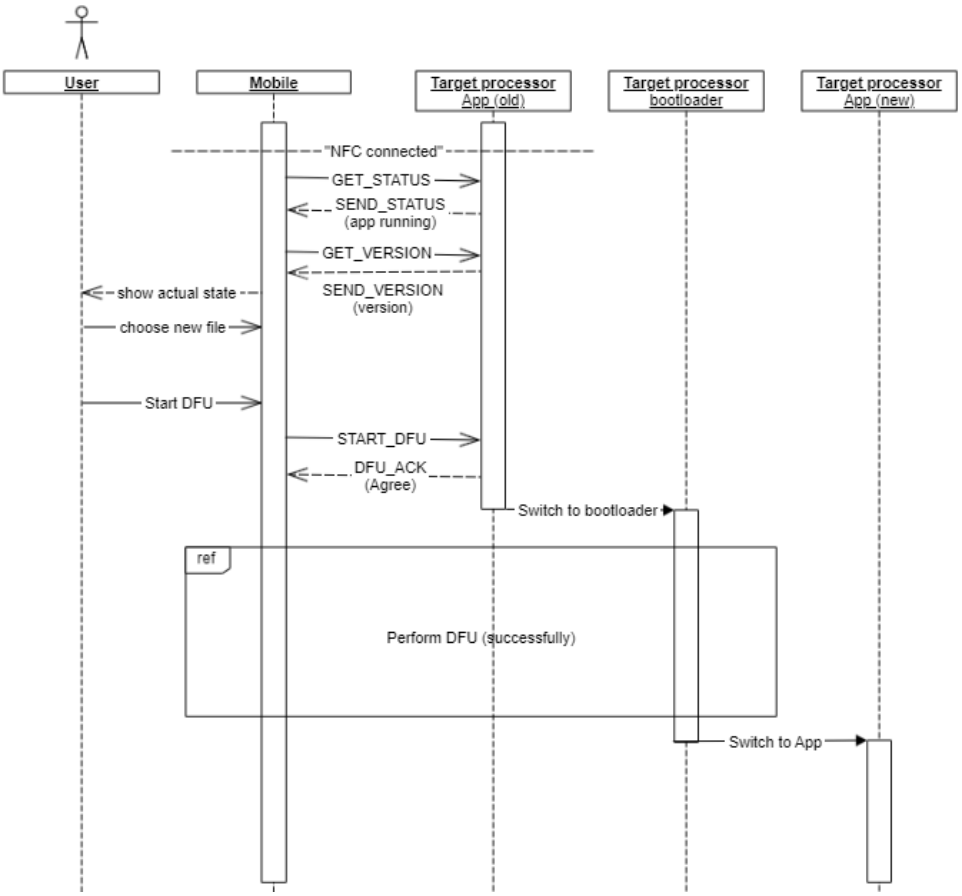

Figure 2.9: Sequence diagram of communication between mobile and target processor for bootloader purposes

The DFU communication is displayed in figure 2.10. The communication starts with the check of the processor's status, so the mobile can confirm that the switch between the user application and the bootloader was performed.

After that, the mobile initiates the DFU with "START_DFU" command, which includes the length of the new application. If the length is valid, the bootloader accepts the DFU and starts waiting for the messages with the new application. The mobile sends chunks of the new application until the whole application is sent. If the connection is lost and reestablshed again, the mobile checks the status of the processor. In the middle of the DFU the response from the processor includes the index of data it expects next. Like this, the devices do not get desynchronized. After every data chunk is received, the processor can notify the mobile, if something went wrong (e.g. the battery is low, so it is not possible to write the flash). Once all the data is sent, the mobile checks that the processor received all the data, by checking its status. This is just a failsafe because the devices should always be synchronized. After that, the mobile sends CRC32 of the application so that the processor can check, the application in memory is correct. If it is, the processor responds with "CRC_ACK" command, with "CRC_OK" value, after which the mobile sends "START_APP", to which the processor reacts by starting the new application.

The NFC communication includes a CRC16 check in every message, but if it wouldn't be enough, it could be a problem, because the error would not be detected until the end of the long DFU and it would be needed to start from scratch. Because of this, I originally wanted to add a continuous CRC checking every few iterations of sending the data. However, I tested the version without it and no problems were occurring (see chapter 4), so the CRC is sent only at the end.

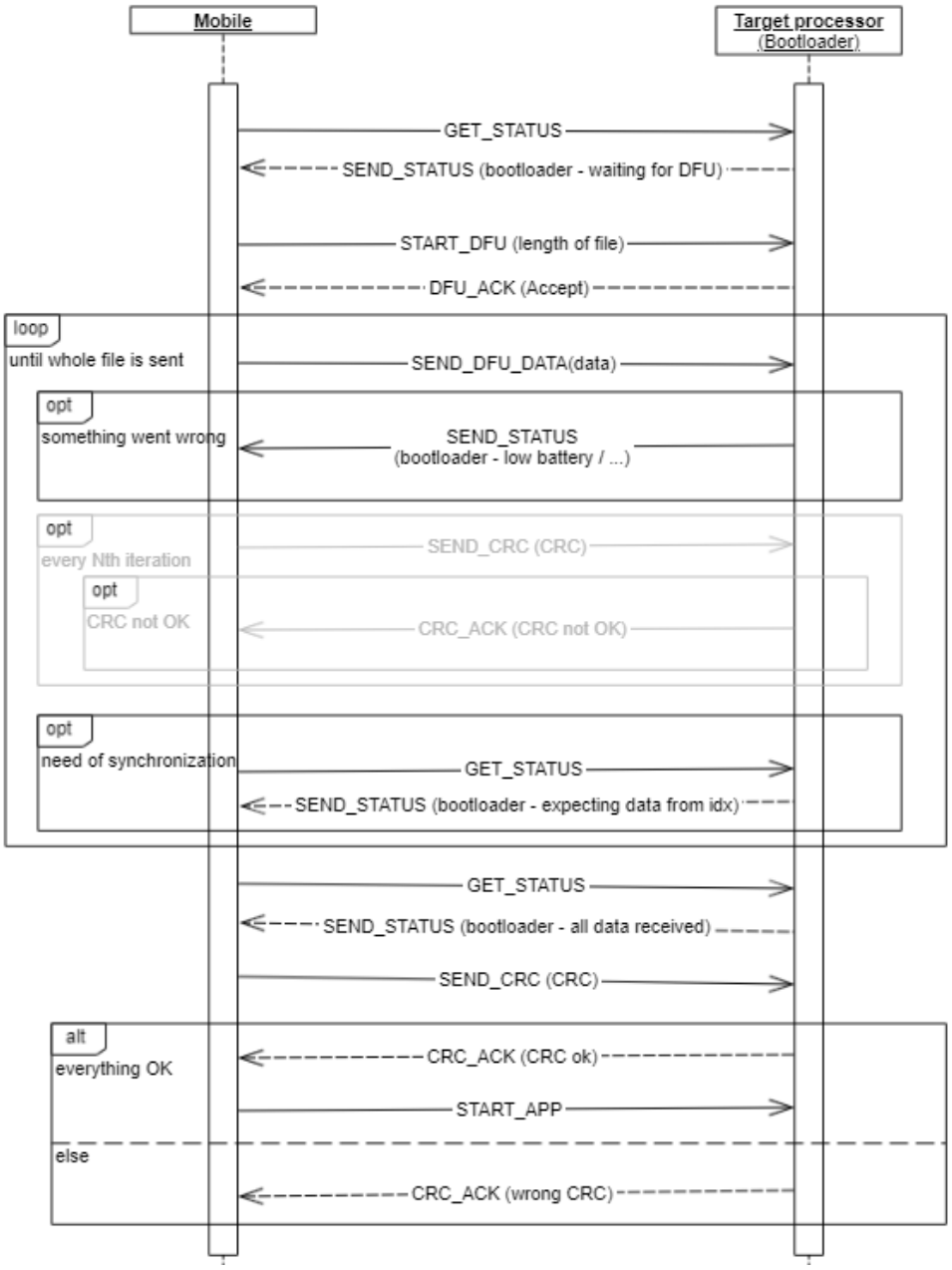

Figure 2.10: Sequence diagram of DFU communication between mobile and target processor

Chapter 3

Implementation

The assignment of this thesis comes from a company. The company requested for the bootloader to be integrated into a bigger project written in C. Because of this, some issues were handled differently than if it was a standalone code. The implementation uses some already implemented modules such as the I^2C driver. Some parts created originally for the bootloader will also be used elsewhere in the future. The module structure may seem too complex in terms of the bootloader, but it was necessary to enable reusability and potential easy switch between external boards or even interfaces for data sending. The code needs to be well documented and readable. The code was written to meets all project guidelines.

One of the most crucial aspects of implementation was the very limited memory space. The internal flash has 128 kB in total and the bootloader should consume as little as possible to leave space for normal application. The size of the bootloader needs to be aligned to flash sectors (4 kB). After the initial assessment, the target size was set to 8 kB.

In this chapter, I will describe the implementation details. First, I will go through all implemented modules for the target processor. Then I will describe the implemented mobile application for testing. Then I will introduce the example user application that is being flashed to the target processor. Then I discuss the robustness of the solution. And finally, I will go through possible future extensions of the code.

3.1 Software modules

The implementation is split into modules. Module diagram is displayed in figure 3.1. Modules with yellow background were implemented in this thesis. The tlv module was implemented after I designed the communication stack, but it was done by someone else in the team because it was promptly needed for other parts of the project.

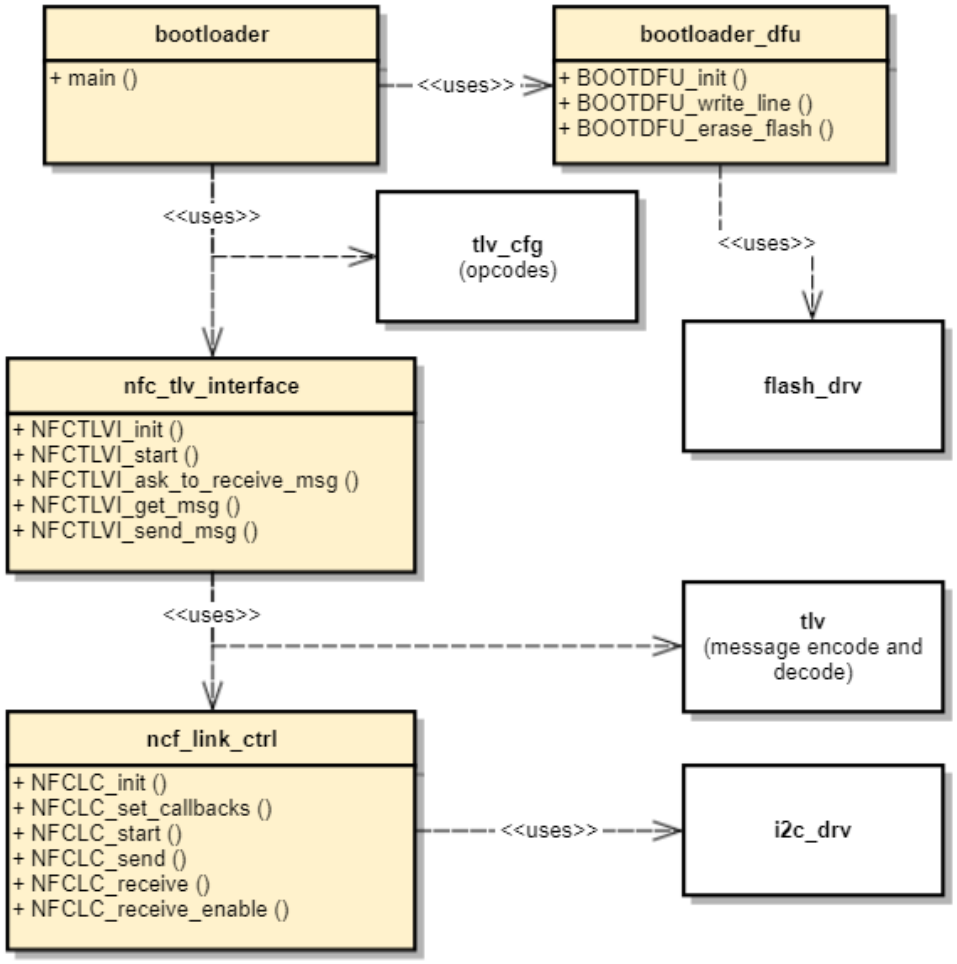

Figure 3.1: Bootloader modules architecture

All of the modules are discussed later in this chapter, but I include a short summary to give an overall understanding to the reader.

The **ncf_link_ctrl** is module for NFC communication over NTAG5 link board. It enables the user to read and write data to the board's SRAM. It uses I^2C interface for the communication, which is why it uses i2C_drv.

The **nfc_tlv_interface** module allows sending and receiving TVL messages over the NFC interface. It uses the tlv module for encoding and decoding the messages, which are transferred using the ncf_link_ctrl module.

The **bootloader_dfu** module handles interactions with flash memory. It includes parts that are run from SRAM and precautions connected to it.

The **bootloader** is the main module. It can start the user application or perform the DFU. It uses nfc_tlv_interface for communication with the mobile phone and bootloader_dfu for operating flash memory.

3.1.1 NTAG5 controller

NTAG5 controller contains functions for using NFC over NTAG5 link board with the usage of I^2C interface. The module is named *nfc_link_ctrl* so that it is possible to easily exchange the NTAG5 controller for a controller of another board.

3.1.1.1 Interface

The interface of the module is adjusted to project recommendations for communication modules. Figure 3.2 shows a class diagram of this module.

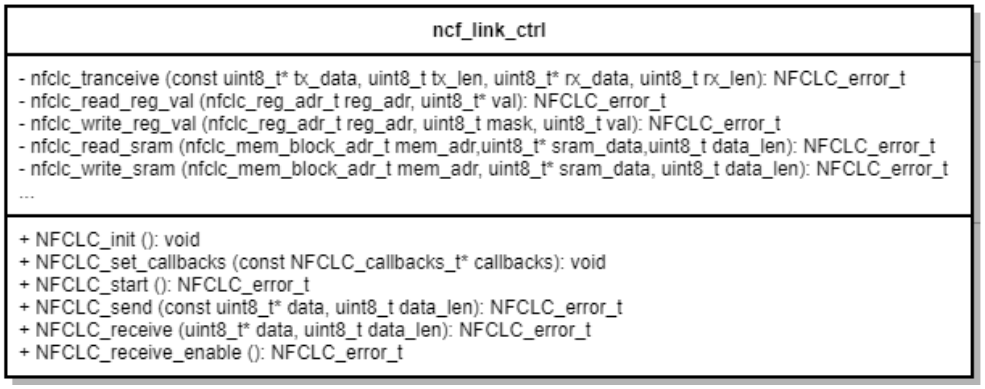

Figure 3.2: Class UML of nfc_link_ctrl module

The interface includes the following functions:

- **NFCLC_init** - initialize environment - set pin configuration
- **NFCLC_set_callbacks** - set callback functions, that will be used to notify upper code layers
- **NFCLC_start** - setup environment, so the communication can begin - configure NTAG5 board and enable interrupts
- **NFCLC_receive_enable** - set NTAG5 board to state, where it can receive data from mobile
- **NFCLC_receive** - read data from NTAG5 board that were previously written by mobile
- **NFCLC_send** - write data to NTAG5 board

The *NFCLC_init* and *NFCLC_set_callbacks* are used only to setup the environment before the actual communication can start. After the initialization,

the *NFCLC_start* function can be called, which contains the first communication with the board. The function checks the setting of NTAG5 board. It is mandatory that the board is set to I^2C slave mode and the SRAM is enabled. These settings can be set only from the mobile phone side. The *NFCLC_start* function also sets the configuration of the board to pass-though mode with direction from I^2C to NFC. This prevents the mobile from immediately start writing the flash and instead, it needs to wait until the higher layers of code decide what direction of communication will be first. Also, it sets event detection (ED) pin to react to NFC field detection.

After everything is set, the actual communication can begin. It is controlled from the upper layer of code and there are two actions (sending and receiving data) that can be done and each consists of multiple stages. Each of the actions needs to be finished before another can start.

Sending data is started by calling the function *NFCLC_send* with a pointer to a data buffer that should be sent. The function transfers the data over I^2C to the board and writes the last data block to let the board know it is all, and it is the mobile turn to read the data. After the last block is written, the function returns. Once the data is read from the board by mobile phone, there is an interrupt on ED pin, which uses the callback, that was set in *NFCLC_set_callbacks* to notify the upper layer of code. With this, the sending data action is finished and the next sending action can be triggered by calling *NFCLC_send* again.

Receiving data from the board consists of three parts. The first one is calling *NFCLC_receive_enable* to change the direction of the communication so that the mobile can write the data to the board. Once the data is written, there is an interrupt on ED pin, which uses the callback, that was set in *NFCLC_set_callbacks* to notify the upper layer of code. Now the upper layer can call *NFCLC_receive* to get the data from the board over I^2C. The read can be done in multiple steps, where the next call of the function returns data from the point where the previous call stopped. In total, it is only possible to read until the end of SRAM, which is used for data transfer. Theoretically, the action ends once the interrupt is handled, and it is possible to start the next action then, but it will not be possible to get to the data once it is done, so it is necessary to read all the wanted data before starting next action.

3.1.1.2 Internal functions

The module includes internal functions handling communication with NTAG5 board on level of I^2C frames. Requirements and principles of this communication were introduced in chapter 1.2.1.2. Functions *nfclc_read_reg_val* and *nfclc_write_reg_val* handle reading and writing bytes to session registers of the board. Functions *nfclc_read_sram* and *nfclc_write_sram* handle reading and writing data to SRAM from a specified address.

All of these functions use a common function *nfclc_tranceive*, which in one call takes care of one whole operation. All operations first send bytes to the board and some operations also include receiving data from the board afterwards. The function takes one input and one output buffer with specification, how long is the data, that should be sent, and how many bytes should be received afterwards.

3.1.1.3 Event detection and callbacks

ED pin is used for receiving signals from the NTAG5 board. It is needed, so the processor does not need to periodically ask for the board's status (e.g. field present, data received). This way, I only configure the NTAG5 board to inform me of the event I am currently interested in, and I configure the processor to react to the changes on the pin with an interrupt.

The NTAG5 board specifications and example applications recommend using "I^2C to NFC pass-through" and "NFC to I^2C pass-through" events for pass-through mode communication. They notify about interaction on the last SRAM block, as is needed for pass-through mode synchronization. However, they are reset when the NFC field is left, which is a problem because it breaks the synchronization. Instead, I use the "Arbiter lock" event, which indicates if the NFC interface is locked. This lock is operated by the board automatically in pass-through mode and it does not reset when the field is left.

I also implemented an option to use the "NFC field present" event, but in the end, it is not used in the bootloader.

The module includes a callback mechanism, which enables the module to notify the upper layers of the code that an event has been detected. It puts together the interrupts on ED pin with the information, which operation is being performed at this time to call the proper callback.

3.1.2 NFC TLV interface

Nfc_tlv_interface module contains functions for using NFC interface for sending and receiving TLV messages.

3.1.2.1 Interface

The interface includes following functions:

- **NFCTLVI_init** - initialize nfc_link_ctrl and setup its callbacks
- **NFCTLVI_start** - start nfc_link_ctrl
- **NFCTLVI_ask_to_receive_message** - use NFCLC_receive_enable to enable receive of message from mobile
- **NFCTLVI_get_msg** - get TLV message from NTAG5 board

- **NFCTLVI_send_msg** - write TLV message to NTAG5 board

Figure 3.3 shows a class diagram of this module.

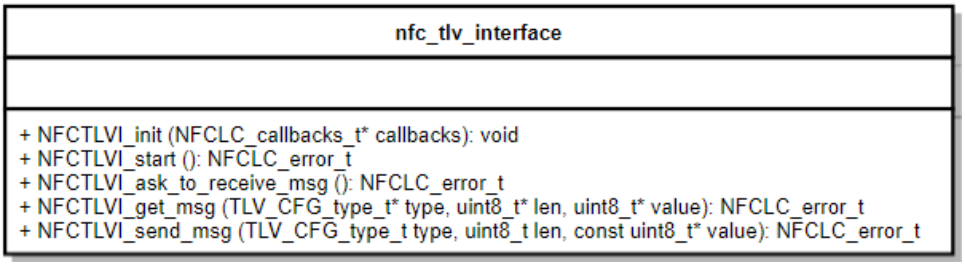

Figure 3.3: Class UML of nfc_tlv_interface module

Function NFCTLVI_get_msg includes two parts. First, the function reads just the first two bytes from the message. They include the type and length fields of the TLV. Based on the preloaded length, the function then loads the rest of the message.

3.1.3 Bootloader

3.1.3.1 Bootloader manager

The *bootloader* module includes the main control functions of the bootloader. Figure 3.4 shows a class diagram of *bootloader* module.

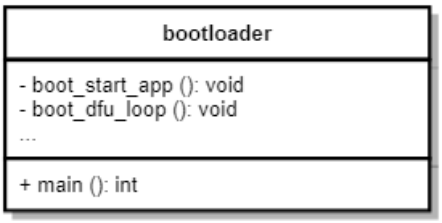

Figure 3.4: Class UML of bootloader module

The reset handler is run first after the reset. It initializes register values to zero, initializes stack, allocates memory for static variables and zeros it, moves code from flash to SRAM (only parts that were defined in linker script), moves initialized data from flash to SRAM. The reset handler than enters the main function. The reset handler was provided by the MCU manufacturer. I just added the part that relocates parts of code to SRAM.

Initialization in main function consists of calibrating the MCU, setting up watchdog and configuring the clock driver.

After the initialization is done, the code checks if DFU should be performed. First, the processor checks if there is a request for DFU from outside.

This is temporarily done by checking state of one pin, just as an example. In the future, this will be modified according to the needs of a specific project. If the pin is set to 1, the processor starts the DFU, if it is 0, the processor continues to checking, if there is a valid application in memory.

This is done by comparing the CRC32 of the application which is currently present in memory to the CRC32 value stored in memory after the last DFU. Properly the CRC and length of application should be stored in the configuration memory. However, there is a problem, that it is not possible to write anything there at this moment. It is caused either by a HW bug or there is a crucial piece of information missing from the MCU specification. Either way, I am waiting for the fix to be done by the MCU provider. To get around this, I temporarily stored the information in the last block of standard flash memory.

Checking validity of the application is done by the following process:

1. Retrieve the supposed length of the application from memory
2. Check if the length is valid (smaller than the maximal length of the application)
    - If not, start DFU
3. Compute CRC32 of memory starting, where the application should start (beginning of new block right after bootloader), and with length retrieved previously
4. Compare the computed CRC with the one store in memory
    - If they match, start the application
    - If they do not match, start the DFU

Starting the user application consist of three parts:

1. **Deinitializing the environment**, because the application initializes everything itself. However, the bootloader does not initialize anything, that needs to be deinitialized, if the start the application. Some things are initialized in DFU, but the processor restarts every time after DFU, so it is not relevant.
2. **Moving IVT** by writing its address of the user application IVT to mvtec register.
3. **Jumping to the restart handler of the user application** - it is at fixed position as is its IVT (which is arranged in the linker script)

The second and the third point is implemented in *boot_start_app* function. The points are implemented by the following code:

```
write_csr(RISCV_MTVEC, APP_IVT_ADDRESS)
asm volatile("jal x1, 0%" ::"i"(APP_RESET_HANDLER_ADDRESS))
```

The DFU process is implemented in *boot_dfu_loop* function, which never returns (the processor resets after the DFU). Its function is described in the following section.

3.1.3.2 Bootloader DFU - controller

3.1.3.2.1 Loop overview

The DFU loop is built on communication and reacting to the communication. The loop is infinite and the only way out is resetting the processor, which is done at the end of a DFU. In every iteration, one complex communication action is done (viz. 3.1.3.2.2) and then the main controller processes a received message (if there was any) and it comes up with the next communication action.

Algorithm 1: DFU loop

Input: Initial communication action, initial state

```
while 1 do
    Perform communication action
    Main controller - procces last communication outcome, set next
     communication action
    if Next action is processor reset then
        Reset processor
    end
end
```

3.1.3.2.2 Communication controller

The communication controller takes care performing complex communication actions requested by the main controller loop. One action can consist of multiple subtasks with rearguard to interactions with NFC. For example, if the main controller wants to send a message, the communication controller needs to first start the send operation and then wait until it is done. The complete list of actions the main controller can ask for is following:

- Send message - Send message and wait, until the mobile reads it
- Ask for message - Enable mobile to send message to NTAG5 board

- Receive message - Receive message that NTAG5 got from mobile (needs to be used right after "Ask for message")
- Ask for message and receive it - Enable mobile to send message to NTAG5 board, wait until it gets it and receive it
- Send message and receive message - "Send message" concatenated with "Ask for message and receive it"
- No action - nothing should be done

Possibilities of communication controller are graphically displayed in figure 3.5. The start points represent actions, which the main controller can request. The endpoints represent the end of one action, after which the main controller is asked for next action request.

Even though the existence of complete receive action ("Ask for message and receive it") and separate receive action ("Ask for message" and "Receive message") may seem redundant. The separate action is needed for receiving long messages (data of the new app) because it enables the processor to do something else in the meantime. And the complete action is for any other case, so the main controller does not need to include dummy states.

Subtasks, which require waiting until the NFC operation is done, are built on until a callback is run from the interrupt in *nfc_link_ctrl* module. The communication controller includes a variable, which is cleared before the communication, then when the controller needs to wait, it first checks if the variable isn't already set. If it is not set, it uses wait_for_interrupt function to minimize the power consumption. The bootloader does not check if the NFC field is currently present. It would have to do it periodically by manually checking the status of NTAG5. The information would not be useful, and the checking would only increase the power consumption.

3.1.3.2.3 DFU state controller

The control of DFU is built on a finite state machine (FSM). Diagram of the FSM is displayed in figure 3.6.

There are two types of states in the FSM:

- **Yellow states** - react to a received message (it is always needed to receive a message, when entering this state); the controller stays in a yellow state, until a specific command is received
- **Violet states** - "fall through" states - they do to react to anything (if a message was received when entering this state, it would be discarded except for "CANCEL_DFU" command); the controller automatically goes to the next state

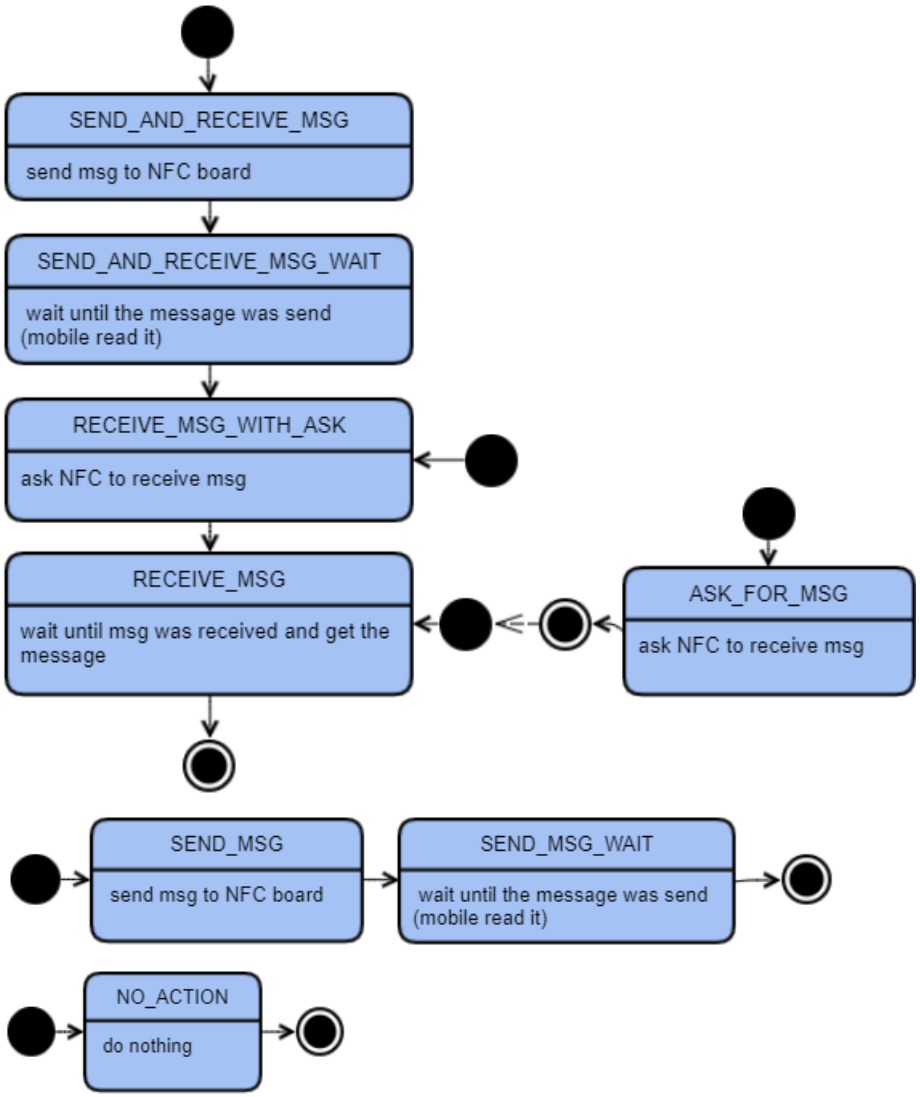

Figure 3.5: Communication controller - actions

The entry point in the figure corresponds to entering the DFU loop. The endpoint corresponds to the reset of the processor. If the controller receives a "CANCEL_DFU" command at any time, it resets to the initial state ("WAITING_FOR_DFU"). The diagram does not include loops on the yellow states for better readability. All the yellow states are able to react to "GET_STATUS" command to fulfill the request stated in the design chapter to be able to synchronize the devices at any time.

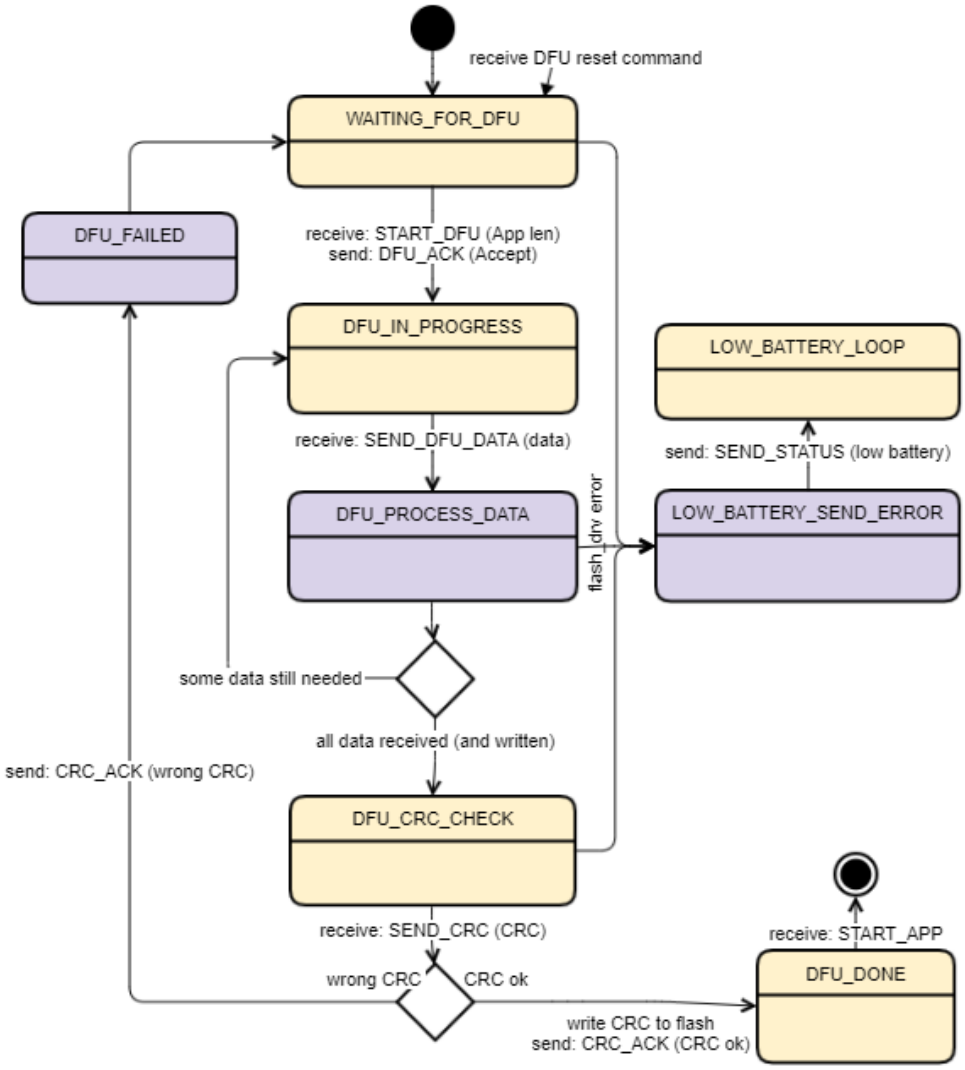

Figure 3.6: DFU controller diagram

3.1.4 Bootloader DFU - flash operations

Module *bootloade_dfu* includes functions, that perform flash memory operations. All function in this module are relocated to SRAM.

3.1.4.1 Interface

Figure 3.7 shows a class diagram of *bootloade_dfu* module.

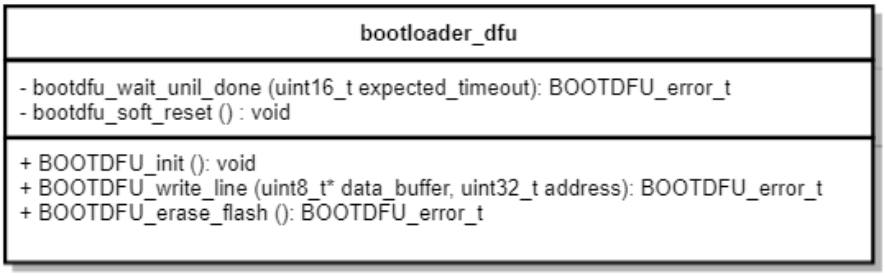

Figure 3.7: Class UML of bootloader_dfu module

The interface includes following functions:

- **BOOTDFU_init** - initialize the flash driver
- **BOOTDFU_erase_memory** - erase memory except for blocks with bootloader
- **BOOTDFU_write_line** - write one line of data to memory

Once the exact cause of the inability to write the configuration flash is established, a new function will be added to this module to handle the writing of the configuration flash according to the new specification.

3.1.4.2 Internal functionality

Functions *BOOTDFU_erase_memory* and *BOOTDFU_write_line* operate on similar principle. They include a call of flash controller operation and waiting until the operation is done. Because of this they and every function that is called from them while the operation is running needs to be put into SRAM and there can be no reads from flash memory. The interrupts need to be disabled because the IVT and handlers are in flash memory, but it is sufficient to disable them inside the function, even if it is in SRAM, because only the part of code, which is executed between the start and end of the flash memory operation. The procedure of the functions can be put into following points:

1. Preparatory calculations
2. Interrupt disable
3. Flash memory operation start
4. Wait until the operation is done

5. Interrupt enable

6. Return

Waiting until operation is done is implemented in *bootdfu_wait_until_done* function. It uses *FLASH_wait_status* function, which busy waits, until the bit, indicating that operation is done, is set or until the timeout is reached. The used timeout differs for each operation. I set the timeouts based on the maximal times for operations from specification, with an additional reserve. I checked the real times, but these will most probably change when moving from the emulator, so it will need to be verified again on real HW. If the operation is not done until the timeout is reached, the code assumes the flash controller got stuck and resets the processor. This can sound controversial because one should normally avoid resetting the processor at all costs while the operation is running, but if the operation does not end, there is nothing else to do.

3.2 Mobile app

To be able to test and benchmark the bootloader, I needed to create a mobile application. I chose the Android platform because it is easier to develop for and I lack the devices needed for iOS application development. The application is just an example for testing and it will not be used in production. The application is based on the example application for NTAG5 link from NXP, which is under Apache license version 2.0.

Most of the reused functionality is present in BaseActivity and MainActivity, which include basic NFC and UI support. BaseActivity includes functions for:

- checking if NFC is enabled on the mobile phone
- detecting tag and initializing the communication
- sending commands to the tag

MainActivity adds functions for:

- checking if the tag is configured correctly
- configuring the tag
- logging - not usable

I have created DFUActivity with inspiration from PassThroughActivity, which was used for demonstrating the pass-through mode of the tag. The DFUActivity is a child of MainActivity. It handles UI events and also reacts to newly detected tags. Internally it includes an asynchronous task, which covers DFU communication and control.

I also needed to add the support of the communication stack I designed in this thesis. The support for TVL encoding and decoding is included in the class TLV, that I created. Functions "receiveMessage" and "sendMessage", that are located in DFUActivity, are used for sending and receiving TVL messages from the NTAG5's SRAM.

The asynchronous task is started once the user selects the application and click on the "START DFU" button. Behavior in the Asynchronous task is controlled by a FSM, which is very similar to the one that controls the DFU in the bootloader. The task is based on an infinite loop, where in every iteration, it finds out the status of NTAG5 board (the currently set direction and lock of communication). From this, the task finds out if there is a message to be read or the memory is opened for writing a new message. Based on this and the state it is in, it performs the next action. The created communication corresponds to the one presented in the design chapter. If the tag gets out of range of the mobile's field, the task is notified because the next NFC operation fails. The task just tries to contact the tag periodically until it is successful and resynchronizes with the processor. This is possible because it is just a testing app. For a real application, a better solution would be needed.

3.2.1 UI and functionality behind it

The UI of the application consists of three screens. The menu and DFU screens are displayed in figure 3.8. There is also a screen with user guide how to operate the application.

In the menu screen, there are links to the two other screens. There is also a button, which on push executes the long-term setting of the NTAG5 link board. This setting needs to be done only once.

The DFU screen consists of three parts:

- New firmware: setting the parameters of the DFU (selection of the new user app binary file, debug option to intentionally send wrong CRC)
- DFU progress: displaying current status of the DFU (progress bar, text evaluation of progress, indication, if the NFC is connected, total DFU time, after it finishes)
- Button: for starting and stopping the DFU

3.2.2 Logging and testing

To test the DFU, I needed to add a possibility to run the DFU multiple times automatically and log the results. Because logging of the communication is time-consuming, I added this possibility with the use of static final variables, which behave similarly as conditional compilation in C. Therefore, if I wanted to test the time consumption of the DFU, it would not be affected by the logging.

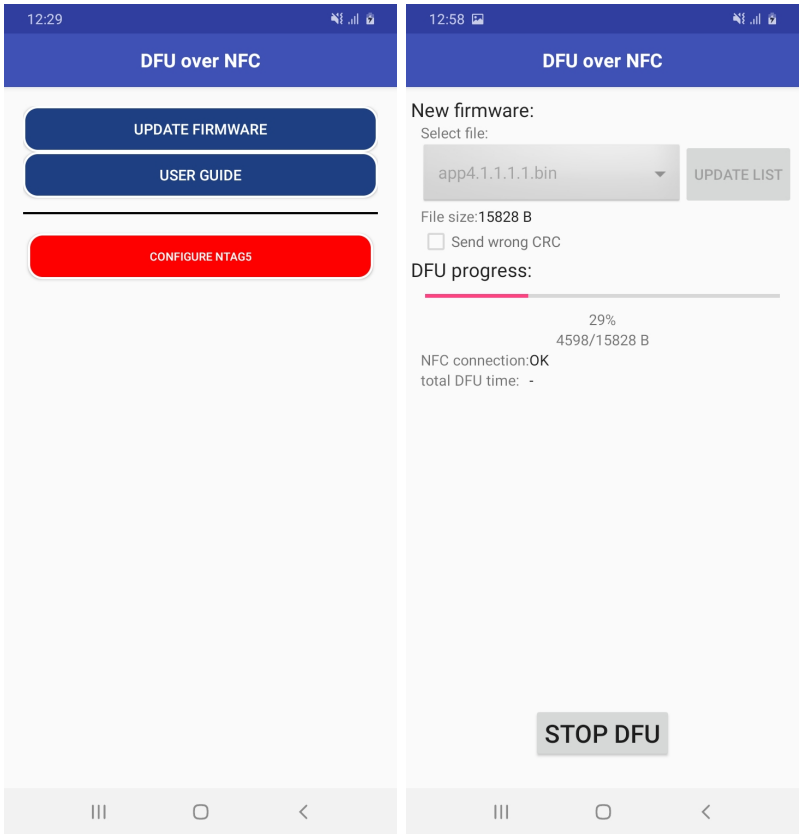

Figure 3.8: Mobile application UI: menu screen (left), DFU screen (right)

The application, to be loaded with the DFU, is selected at runtime from the mobile application. There is an option to randomize data and in that case, only the length of the selected application is taken into account. The number of test iterations can also be specified. Both options, the randomization and number of iterations, need to be selected in the source code and it cannot be set directly in the app.

If testing is enabled, the final log is stored in the "/DFU_TEST_LOGS" mobile folder. Figure 3.9 shows how the test results may look in the log.

```
05-05 13:12:20.180 22522 22522 D DFU loop: TEST DONE
05-05 13:12:20.180 22522 22522 D DFU loop: file size: 122752 B (119 kB)
05-05 13:12:20.180 22522 22522 D DFU loop: number of iterations: 50
05-05 13:12:20.181 22522 22522 D DFU loop: total time: 3944,55 s
05-05 13:12:20.181 22522 22522 D DFU loop: avg. dfu time: 78,89 s
05-05 13:12:20.182 22522 22522 D DFU loop: FAILED: 0/50
05-05 13:12:20.182 22522 22522 D DFU loop: SUCCESSFUL: 50/50
```

Figure 3.9: Example of testing output in log

3.3 Example user application

The example user application that is being loaded by the bootloader is just for demonstration purposes. It does not have any special functionality. It only flashes a LED to be able to see that it is running. Because the integration of the NFC communication to the user app is out of scope of this thesis, the invalidation of CRC and reset of the processor is performed on a push of a button. Discussion of how the NFC could be integrated into the app is done in section 3.5.1. When compiling the user application, it is needed to link it for the correct place in memory.

3.4 Robustness discussion

The goal of this thesis was to create a robust bootloader. In this section, I will go through the identified dangers, and I will describe how they are handled in the final solution.

3.4.1 Corruption of the bootloader

The first risk is that the bootloader code could get corrupted and the device would get bricked. This could happen if the bootloader's code was modified at runtime, and the processor would reset in the middle of the operation. This is not a problem in my solution because the bootloader's code is never modified, and therefore the processor is always in a stable state for reset in terms of the bootloader. Also, the bootloader is protected from the mass erase function.

3.4.2 Buggy user application

The second risk is that the user application that has been flashed in the memory is not working properly, and it cannot trigger the DFU the standard way. For example, it always resets before it can invalidate the CRC in the configuration memory. This is taken care of by including the option to trigger the DFU in the bootloader by a signal from an external source. Without this option, it would be relatively easy to get the processor to a state when it cannot update the user application anymore. The form of the external signal will be dependent on the specific project the MCU will be used in.

3.4.3 Desynchronization of devices during DFU

The third risk is that the mobile phone and the target processor will get desynchronized during the DFU. There are multiple events, which could cause this, but the solution must be done from the mobile application side because it is the master of the communication, and the bootloader does not even know about the problem.

However, the bootloader needs to meet some requirements, so the mobile phone can solve the situation. The first requirement is that the bootloader's DFU controller is implemented so that in every state, it is able to respond to the "GET_STATUS" command, and the responses must be informational enough. The second requirement is that it has to possible to cancel (reset) the DFU at any time when the "CANCEL_DFU" command is received. If the bootloader implements those two requirements (which it does), the mobile phone is able to evaluate the situation and handle it properly.

Note that if the target processor restarts, it loses track of the current progress of the DFU, and it expects to start from scratch. The same applies to the mobile application.

List of possible causes of desynchronization and situation resolution:

- **The target processor restarts** - the NFC connection drops, once reconnected, the mobile application gets the target processor status, finds out, that the DFU is not in progress there and the mobile starts the DFU from scratch
- **The mobile phone application restarts** - the NFC connection drops, once reconnected, the mobile application gets the target processor status, finds out, that the target processor is in middle of a DFU, so it sends the "CANCEL_DFU" command and that it starts the DFU from scratch
- **The mobile phone stops DFU** - the mobile phone does not notify the target processor immediately and so this situation is transformed into the "The mobile phone application restarts", once the user starts new DFU and it is handled the same way
- **NFC gets out of range** - after reconnection the mobile phone gets the status of the target processor, if both devices are still in the middle of a DFU, the mobile phone starts to send data, from point, which the the target processor requested; otherwise the situation is the same as one from above and it is handled the same way

3.4.4 Low battery

The fourth risk is that the voltage gets low and the flash memory cannot perform erase and write operations. For now, this risk is only hypothetical because the processor is being emulated. The bootloader detects the low voltage at first run of a flash operation. Once the bootloader knows about the problem, it informs the mobile phone about the situation, and the DFU controller switches to "LOW_BATTERY_LOOP" state, where it stays, until "CANCEL_DFU" command is received or the processor resets. The user is informed by the mobile phone about the situation and he is expected to solve it by changing the battery or charging the device.

3.5 Future extensions

3.5.1 Adaptation to specific project

The code of the bootloader will need to be adapted to the projects it will be used in.

The first adaptation will need to be done on the implementation of a request for a DFU from an external source. It enables the bootloader to update the user application if there is a valid app in memory, but for some reason, it is not able to invalidate CRC to initiate DFU. The form of the request is dependent on the needs of the specific project. It could be, for example, implemented as sniffing on a pin for a certain amount of time after reset or trying to read a message from NFC.

The second adaptation will be the integration of the bootloader needs (CRC invalidation) and NFC communication to the real user application. The application will need to implement a mechanism for receiving requests to perform DFU and it will have to be able to invalidate the CRC to trigger it after reset. The communication can be done over the NFC interface that was developed in this thesis. The only potential modification needed is the exchange of interrupt-based callbacks for events to be able to better integrate it into a task in the application. Also, the application needs to be linked for the correct place in memory.

The third adaptation will be the modification of opcode values (type field values of TLV messages). This can be done in configuration files. It is needed to synchronize opcodes from the bootloader and the user application with the mobile phone application.

3.5.2 Security suggestions

Communication over NFC is not secure. By default, the devices do not authenticate, the data is not encrypted and anyone can eavesdrop on it from a small distance. It is even a bigger issue for the bootloader because it could flash a forged application, or someone could get to the binary file of the real application. This topic will need to be investigated in the future and the risks taken or adequate contra measures made.

Suggested points for investigation:

- NTAG5 offers AES authentification (not encryption), which could enable authentication of the mobile phone, so the tag would react only to a known device

- Possibility to lock some parts on NTAG5's configuration memory, so it is not possible to write it from NFC

- Research, if there is a cryptographic library with a very small footprint so that it would fit in the very limited memory, and it would leave enough space for the user application

3.5.3 Transfer from emulator

Once the project moves from emulator to the real HW, it will be needed to retest the whole solution, especially the time consumption and flash memory behavior. Also the power consumption of the solution the will need to be measured. There will probably be some modifications needed based on the results.

Chapter 4

Testing

In this chapter, I go through the testing results of the bootloader. First, I summarize the environment and devices I had available. Then, I describe how I have tested the functionality and the results of the tests. Then, I introduce the time measurements and analysis I have performed. Finally, I show the analysis of the bootloader's memory consumption. The power consumption measurement, which was requested in the assignment, cannot be performed because the processor is emulated on an FPGA.

4.1 Development and testing environment

I used Eclipse IDE for the development and debugging of the bootloader. For developing the mobile application, I used Android studio. For programming the code to the flash I used Jlink together with OpenOCD. The configuration file for OpenOCD was provided by the MCU manufacturer. I used OpenOCD from Eclipse and from the command line to be able to manually modify and check the content of the flash memory. For binary file browsing and editing, I used the HexEditor. For recording digital and analog signals from the pins, I used a Saleae with the Saleae Logic software (beta version).

I had two Android phones at my disposal for testing the bootloader:

- Samsung A21s (2020)
- Samsung Galaxy A10 (2019)

Both of the phones have Android 10.

4.2 Functionality testing

The testing of functionality was done in two stages. The first one was manual and the second one automatic.

The manual testing was done throughout the development. Among other things, it was focused on interrupting the communication selected in specific places, synchronizing the devices (resetting the DFU process on mobile, or resetting the target processor, etc.), sending invalid CRCs and commands, invalidating CRC manually with OpenOCD, etc.

The automatic testing was possible because I added testing and logging feature to the mobile application. This testing included over 900 runs of the DFU. The tests were run on both available mobile phones. The testing included sending fixed binaries (real application included) in size of 16 kB and 120 kB (maximal possible length) and random data with the same length. The tests were made with different distances of the mobile and NTAG. The first part was done with the devices very close to each other. The second part was done on a longer distance. And the third part was done with distance on the edge of the range, so the devices were disconnecting from time to time in the process.

No failed DFU appeared in the whole automatic testing process (I checked that if a wrong CRC was sent, that it would be detected correctly). Based on this, I abandoned the idea that a continuous CRC checking during DFU would be beneficial because it would only prolong the DFU.

4.3 Time analysis

I used a Saleae with the Saleae Logic software (beta version) for measuring time consumption. Figure 4.1 shows a recording of a DFU with a 16 kB user application. We can see both I^2C pins, event detection pin recorded as digital signals, and NFC recording being an analog.

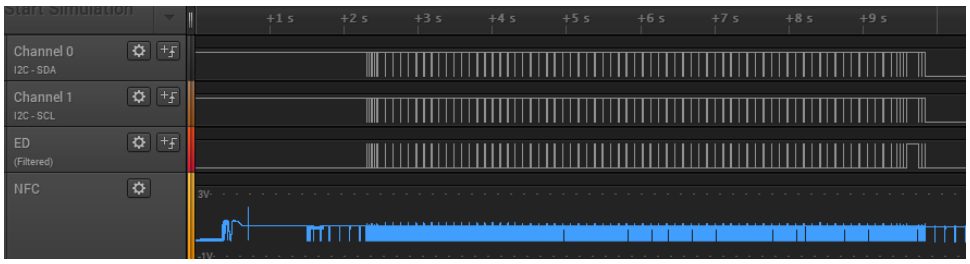

Figure 4.1: DFU recording from a Saleae (16 kB user app)

4.3.1 DFU times

Table 4.1 presents average DFU times of a 120 kB user application. The times are separate for each phone and for the options, how much of NTAG5's SRAM is used. Using only 248 B of SRAM is faster because the mobile phone only needs one command to write the whole data chunk, and because of it,

I selected it for the final solution. Any other data in the Time analysis section is measured when using 248 B of SRAM.

Table 4.1: DFU times for a 120 kB user application with using 256 B or 248 B of NTAG5's SRAM

Mobile	**Time (256 B)**	**Time (248 B)**
Samsung A21s	79.1 s	73.7 s
Samsung A10	61.9 s	56.6 s

4.3.2 Transition time of one data chunk

In figure 4.2 we can see that the bootloader reacts quickly and nearly immediately starts the I^2C transmission after the ED pin is set high. We also see that there are very long pauses between the NFC transitions. These pauses are significantly different for each mobile phone, as we can see in figures 4.3 and 4.4. Table 4.2 summarizes the times of the intervals of NFC communication, that send one data frame.

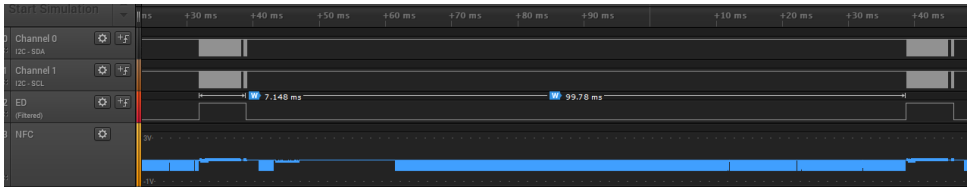

Figure 4.2: Record of transferring 248 B over NFC with Samsung Galaxy A10

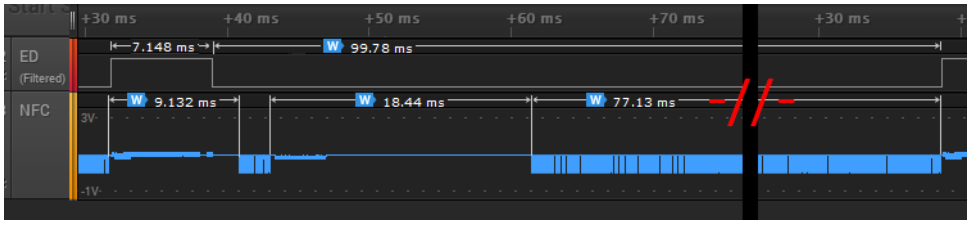

Figure 4.3: Record of transferring 248 B over NFC with Samsung Galaxy A10 - zoom with time measurements

From the table 4.2, we see that the NFC transmissions take exactly the same time, and the only difference is the pause between the NFC transmissions. Also, the pauses get longer throughout the DFU, which may suggest the task gets lower and lower priority. Even though the mobile phones are from the same manufacturer and have the same android version, their behavior is very different. Anyhow the problem, why the communication takes longer

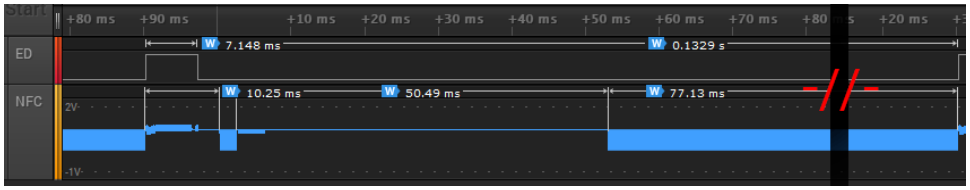

Figure 4.4: Record of transferring 248 B over NFC with Samsung A21s - zoom with time measurements

Table 4.2: Times of intervals in NFC communication

Mobile	Pause	Get status	Pause	Data transition
Samsung A21s	10.3 ms	2.22 ms	50.5 ms	77.13 ms
Samsung A10	9.1 ms	2.22 ms	18.4 ms	77.13 ms

than it could, is on the Android side. It could probably be optimized, but it is out of scope ot this thesis.

4.3.3 Others

Computation of CRC32 of a 120 kB application in flash memory takes 1.3 s.

Table 4.3 presents times of flash memory operations. It includes maximal times taken from the specification (Spec time) and the measured values on the emulator (Real time).

Table 4.3: Times of flash memory operations

Operation	Spec time	Real time
Mass memory erase	10 ms	5 ms
Line write	3.5 ms	3 ms

4.4 Memory analysis

The bootloader is compiled with "-Os" optimization to minimize the footprint. Figure 4.5 includes overview of bootloader consumption of flash (on the left) and SRAM memory (on the right). The chart was created by a script from the project, which parses elf and object files for information. Table 4.4 shows information about footprint size of each module bigger than 200 B (the rest is summed under "others"). If a part of a module is also relocated to SRAM on start-up, the information about how many bytes is relocated is present in brackets e.g. "(X relocated)".

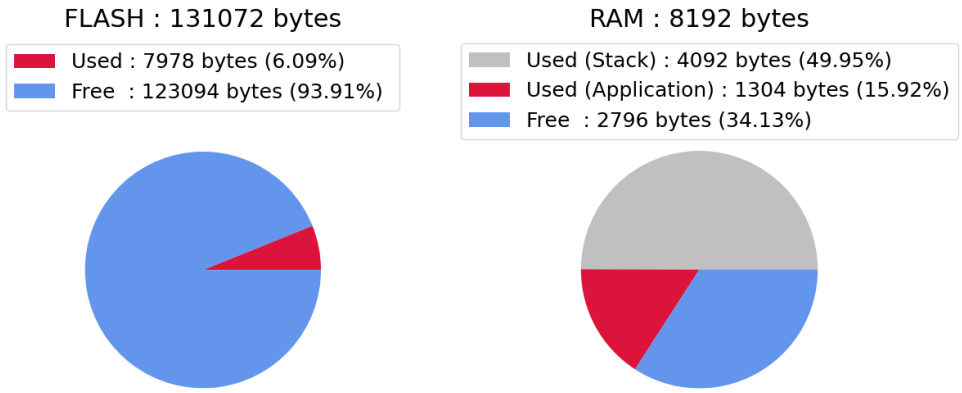

Figure 4.5: Memory consumption of bootloader

Table 4.4: Flash memory footprint of modules in bootloader

Module	**Flash footprint [B]**
bootloader	1850
gpio_drv	1662
nfc_link_ctrl	1296
i2c_drv	828
flash_drv	444 (342 relocated)
clock_drv	432
bootloader_dfu	336 (336 relocated)
nfc_tlv_interface	262
others	766
total	**7978**

The bootloader fits into 8 kB of flash memory (2 memory blocks). There is not much space left at this moment for potential extensions. There are two possible solutions. The first one is increasing the bootloader space to 12 kB. The second one would be trying to reduce the memory consumption of the underlying modules that are used in the bootloader. The modules are implemented for the whole project, so they include code that the bootloader does not need (most branches are not used), but it cannot be optimized out because the concrete function is called. I already reduced the footprint of the underlying modules in this thesis. However, I only focused on the biggest consumers, so there will still be some room for improvement. The biggest consumer was gpio_drv and especially one function with a large switch-case.

By putting most of the cases behind conditional compilation, so it is not compiled for the bootloader, I managed to free nearly 500 B.

The consumed SRAM consists partly of the code that had to be relocated to SRAM and partly of static variables. The relocated code consumes at least 678 B (bootloader_dfu + flash_drv). The biggest consumers from the static variables are the data buffers needed for communication with a total of 512 B. The SRAM memory is not shared between the bootloader and the user application (it is reinitialized in start-up codes), so a bit higher consumption of the bootloader is not a problem.

Conclusion

The goal of this thesis was to create a robust flash memory bootloader for a proprietary RISC-V microcontroller over NFC, with usage of NTAG5 link board. The assignment was created by a company. Because of it, the implementation had to be created inside an existing project, that was still being worked on. I had to familiarize myself with the project and multiple technologies, I have never worked with before.

The assignment requested implementing a flash driver in case it was not already implemented. In the end, the implementation of the flash driver was provided by the microcontroller manufacturer, but I actively participated in its testing and debugging. I also had to modify it a little, so I could integrate it into the project.

As requested, I designed and implemented a communication stack, which uses the NTAG5 link board. This stack enables communication between the mobile phone and the target processor via TLV encoded messages.

With the use of the created communication stack and the flash driver, I designed and implemented the flash bootloader. I focused on the robustness of the solution and I tried to minimize the memory consumption. The final bootloader fits into 8 kB of flash memory.

To test the bootloader, I implemented an example Android application. The application also includes an automatic testing option with logging and a user guide.

I tested the functionality of the implementation, both manually and automatically. I analyzed the memory footprint of the bootloader and I measured the time consumption of the firmware update. Finally, I evaluated and documented all the results. However, I could not measure and document the power consumption of the realization, because the processor was just emulated on an FPGA.

Because the target processor was still in development, when I worked on this thesis, its specification was continuously being updated and some things were not working properly. Even though this made my work more difficult

and sometimes I had to spent more time on seemingly simple things, it was very exiting to work on a real project, which will be used in the future.

Bibliography

[1] NFC for embedded applications [online]. NXP Semiconductors, 2014, [cit. 2021-4-7]. Available from: `https://www.nxp.com/docs/en/brochure/75017587.pdf`

[2] Wang, L.; Chen, X. A Design and Implementation of Bootloader Based on MPC8280. *Applied Mechanics and Materials*, volume 668-669, 10 2014: pp. 592–597, ISSN 1662-7482.

[3] Sha, C.; Lin, Z. Design Optimization and Implementation of Bootloader in Embedded System Development. In *2015 International Conference on Computer Science and Applications (CSA)*, 2015, pp. 151–156, doi:10.1109/CSA.2015.37.

[4] Bogdan, D.; Bogdan, R.; et al. Design and implementation of a bootloader in the context of intelligent vehicle systems. 11 2017, pp. 1–5, doi:10.1109/SusTech.2017.8333509.

[5] Lacamera, D. *Embedded Systems Architecture: Explore architectural concepts, pragmatic design patterns, and best practices to produce robust systems*. Birmingham: Packt Publishing, Limited, 2018.

[6] Fan, X. *Real-Time Embedded Systems: Design Principles and Engineering Practices*. Cambridge: Elsevier Science & Technology, 2015, ISBN 9780128015070.

[7] Kachman, O. Effective Multiplatform Firmware Update Process for Embedded Low-Power Devices. *Information Sciences and Technologies*, volume 11, no. 1, 06 2019: pp. 6–11.

[8] Interrupt Based Bootloader Design For Embedded System Updates. *Electronics for You*, 2021.

[9] Mansor, H.; Markantonakis, K.; et al. Let's Get Mobile: Secure FOTA for Automotive System. In *Network and System Security*, edited by M. Qiu; S. Xu; M. Yung; H. Zhang, Cham: Springer International Publishing, 2015, ISBN 978-3-319-25645-0, pp. 503–510.

[10] Kacem, H.; Ben Halima, I. Efficient Implementation of Updating MCU Using the Firmware on the Air. In *Image and Signal Processing*, Cham: Springer International Publishing, 2018, ISBN 978-3-319-94211-7, pp. 176–185.

[11] Coskun, V.; Ok, K.; et al. *Near Field Communication (NFC): From Theory to Practice*. Hoboken: Wiley, first edition, 2012, ISBN 9781119971092.

[12] Coskun, V.; Ozdenizci, B.; et al. The Survey on Near Field Communication. *Sensors*, volume 15, no. 6, 2015: pp. 13348–13405, ISSN 1424-8220.

[13] TN1216, ST25 NFC guide Rev 2 [online]. STMicroelectronics, 2016, [cit. 2021-4-19]. Available from: `https://www.st.com/resource/en/technical_note/dm00190233-st25-nfc-guide-stmicroelectronics.pdf`

[14] NfcV [online]. [cit. 2021-6-26]. Available from: `https://developer.android.com/reference/android/nfc/tech/NfcV`

[15] UM10204: I2C-bus specification and user manual Rev. 6 [online]. NXP Semiconductors, 2014, [cit. 2021-6-22]. Available from: `https://www.nxp.com/docs/en/user-guide/UM10204.pdf`

[16] NTAG® 5 link: NFC Forum-compliant I2C bridge for IoT on demand [online]. NXP Semiconductors, [cit. 2021-3-5]. Available from: `https://www.nxp.com/products/rfid-nfc/nfc-hf/nfc-tags-for-electronics/ntag-5-link-nfc-forum-compliant-ic-bridge-for-iot-on-demand:NTAG5-LINK`

[17] OM2NTx5332: NTAG® 5 development kits [online]. NXP Semiconductors, [cit. 2021-3-5]. Available from: `https://www.nxp.com/products/rfid-nfc/nfc-hf/nfc-tags-for-electronics/om2ntx5332-ntag-5-development-kits:OM2NTX5332`

[18] NTP53x2: NTAG 5 link - NFC Forum-compliant I2C bridge Rev. 3.3 [online]. NXP Semiconductors, 2020, [cit. 2021-3-15]. Available from: `https://www.nxp.com/docs/en/data-sheet/NTP53x2.pdf`

[19] AN12364: NTAG 5 - Bidirectional data exchange Rev. 1.0 [online]. NXP Semiconductors, 2020, [cit. 2021-3-15]. Available from: `https://www.nxp.com/docs/en/application-note/AN12364.pdf`

[20] RM00221: NTAG 5 - Android application development Rev. 1.1 [online]. NXP Semiconductors, 2020, [cit. 2021-3-15]. Available from: https://www.nxp.com/docs/en/reference-manual/RM00221.pdf

[21] Internal company document. The company wants to stay anonymous, 2021, [cit. 2021-6-25].

[22] Waterman, A.; Asanović, K. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2. RISC-V Foundation, May 2017. Available from: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[23] Internal company document. The company wants to stay anonymous, 2021, [cit. 2021-6-25].

[24] Internal company document. The company wants to stay anonymous, 2021, [cit. 2021-6-25].

[25] Waterman, A.; Asanović, K. "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified. RISC-V Foundation, June 2019.

Appendix A

NTAG5 - Pass-through mode

This appendix includes suggested usage of NTAG5 in Pass-through mode from the manufacturer. Figure A.1 shows communication with direction from I^2C to NFC. Figure A.2 shows communication with direction from NFC to I^2C.

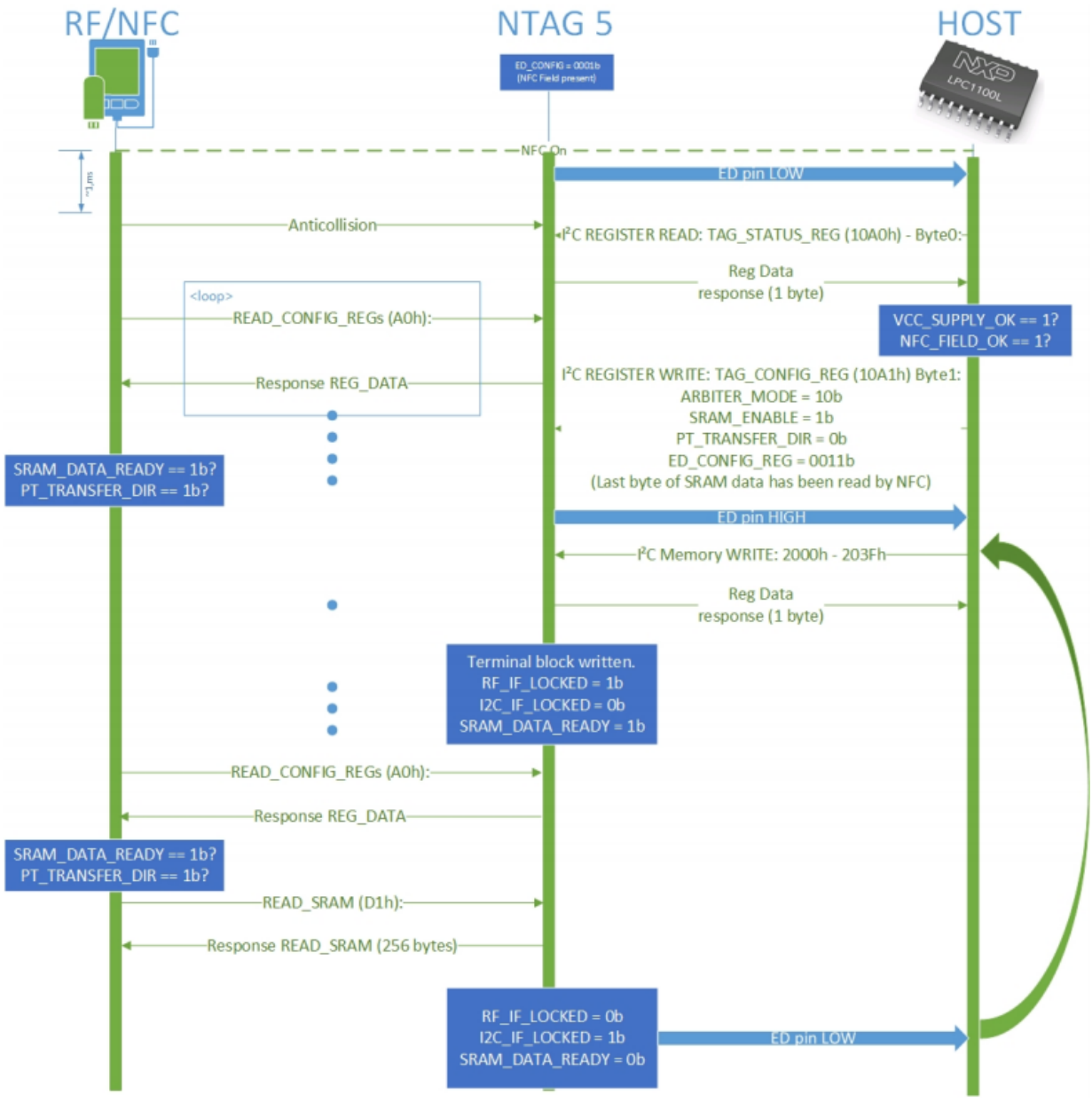

Figure A.1: Example usage of NTAG5's Pass-through mode (I^2C to NFC) [19]

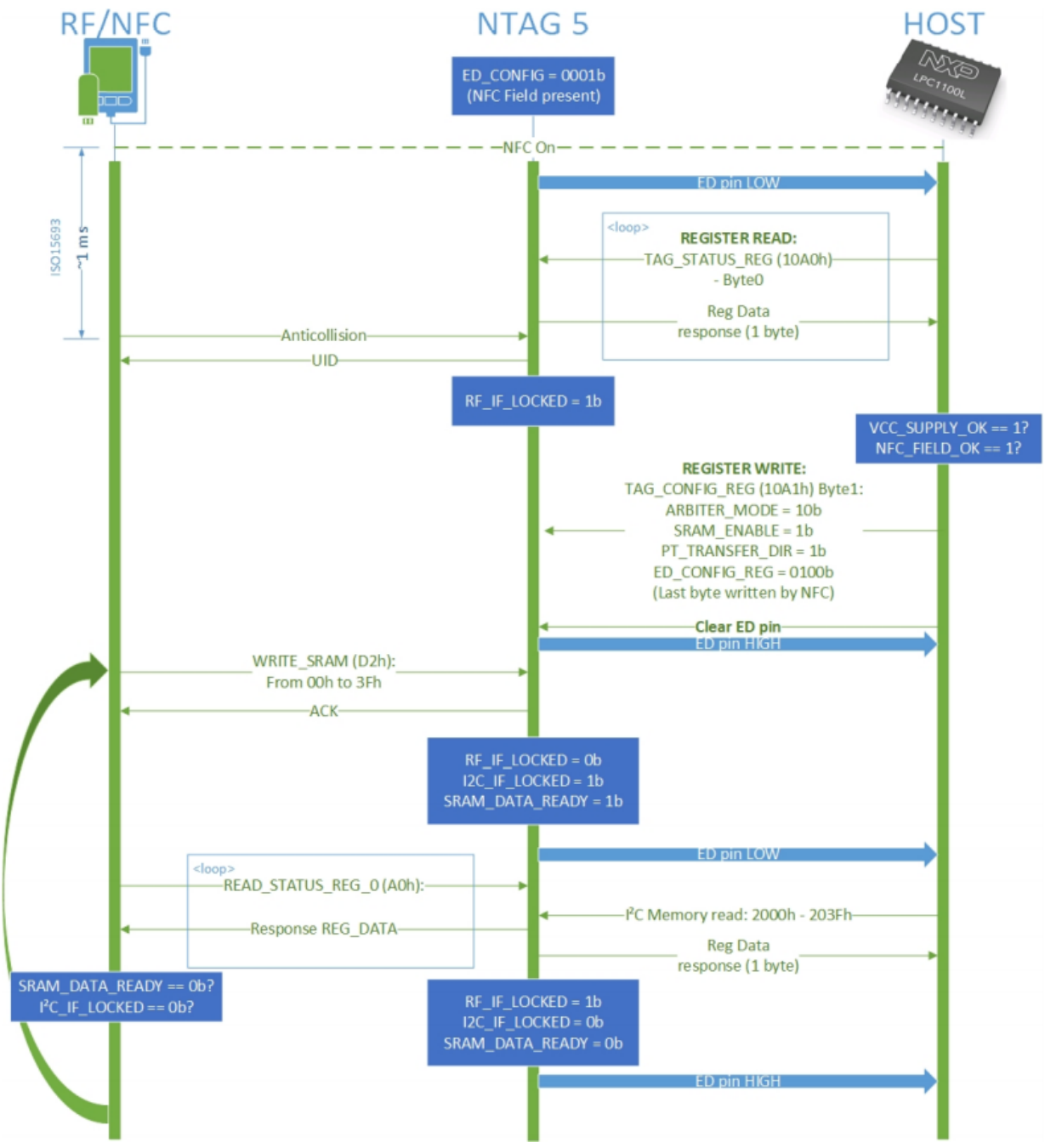

Figure A.2: Example usage of NTAG5's Pass-through mode (NFC to I^2C) [19]

Appendix B

Acronyms

DFU Device firmware update

ED Event detection

GPIO General-purpose input/output

HW Hardware

I²C Inter-Integrated Circuit

IoT Internet-of-things

ISA Instruction set architecture

IVT Interrupt vector table

MCU Microcontroller unit

NFC Near field communication

OTA Over-the-air

OTW Over-the-wire

PWM Pulse-width modulation

RAM Random-access memory

SRAM Static random-access memory

Appendix C

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `bootloader` ..... implementation sources - bootloader
  - `mobile_app` ..... implementation sources - mobile application
  - `thesis` ..... the directory of LaTeX source codes of the thesis
  - `doc` ..... documentation of source code
- `videos` ..... videos of the DFU
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format