TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Keijo Lass 153021IASM

FIRMWARE UPGRADE SOLUTION FOR TALTECH SELF-DRIVING CAR CONTROLLERS

Master's Thesis

Supervisor: Mairo Leier
PhD

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Keijo Lass 153021IASM

TALTECHI ISESÕITVA AUTO KONTROLLERITE TARKVARA UUENDAMINE

Magistritöö

Juhendaja: Mairo Leier
PhD

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Keijo Lass

02.01.2019

Abstract

This thesis presents the current trends and state of the art in automotive firmware upgrading. As a case study, a solution for upgrading the controllers on the Tallinn University of Technology self-driving vehicle was proposed. The goal is to provide a foundation upon which the fully remote firmware upgrading solution can be built. From the software design perspective, the developed code is to be modular and portable between the controllers. As a result, a script for preparing the final binary image was devised, along with memory partitioning schemes and portable bootloaders for the controllers and host side scripts to carry out the upgrading process. The developed transport protocol for uploading the firmware to the controllers is independent of communication mediums and offers reliable transfers.

This thesis is written in english and is 47 pages long, including 6 chapters, 26 figures, and 4 tables.

Annotatsioon
TALTECHI ISESÕITVA AUTO KONTROLLERITE TARKVARA UUENDAMINE

Seoses pidevalt suureneva mikrokontrollerite kasutusega tänapäeva autotööstuses tekib vajadus tõhustada tarkvara uuendamise protsesse. Töös on uuritud autotööstuse püsivara uuendamise praegust tehnilist taset ning trende.

Lõputöö rakenduslik osa on teostatud Tallinna Tehnikaülikooli isesõitva auto projekti raames, ning põhieesmärgiks on välja töötada lahendus sõidukil olevate mikrokontrollerite püsivara uuendamiseks. Tarkvara arithektuuri vaatenurgast on loodavas lahenduses oluline keskenduda modulaarsusele ja koodi taaskasutatavusele, et vajadusel oleks võimalikult lihtne kontrollereid süsteemi lisada, ära võtta või muuta. Uuenduste pealelaadimiseks on vaja välja arendada rakenduslik kiht, mis ei sõltuks füüsilisest ja transpordi kihist. Arendatav protokoll peab olema veakindel, ning olema suuteline toime tulema paketikadude ning andmeside katkestuste korral.

Lõputöö tulemusena valminud tarkvara on alustalaks tulevikus kogu süsteemi täielikult kaug-uuendatavaks muutmisel. Esmase sammuna loodi kompileeritud koodi järeltöötlus skript, mis valmistab ette lõpliku mälutõmmise. Töö ühe osana vaadeldakse alglaaduri eripärasi sardsüsteemides, ning implementeeritakse üks võimalikest variantidest. Ühtlasi pakutakse välja meetod kontrolleri mälu sektsioneerimiseks. Samuti valmis tarkvara uue püsivara pealelaadimiseks kontrolleri ja ka peaarvuti (*host*) jaoks. Välja pakutud protokoll uuenduste transpordiks osutus töökindlaks ning kiiruselt võrreldavaks eelnevalt kasutusel olnud viisile kontrollerit uuendada programmaatoriga.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 47 leheküljel, 6 peatükki, 26 joonist, 4 tabelit.

List of Abbreviations

CAN	Controller Area Network
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
ECU	Electronic Control Unit
EEPROM	Electrically Erasable Programmable Read-Only Memory
FOTA	Firmware Over the Air
FSM	Finite State Machine
HAL	Hardware Abstraction Layer
IAP	In-Application Programming
ICE	In-Circuit Emulator
IDE	Integrated Development Environment
JTAG	Joint Test Action Group
MSP	Main Stack Pointer
OS	Operating System
OTP	One-Time Programmable Memory
PC	Program Counter
RAM	Random Access Memory
ROM	Read-Only Memory
ROS	Robot Operating System
RTOS	Real-time Operating System
RWW	Read-While-Write
SHA-1	Secure Hash Algorithm 1

SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol
VTOR	Vector Table Offset Register

Table of Contents

List of Figures

List of Tables

1 Introduction

Starting from the late 1990s, the car manufacturers have started to integrate more and more controllers into the vehicles, thus rapidly increasing the amount of software components used in the automotive industry each year. Around the same time, the mechanics started to use diagnostic testing tools to install software updates for the Electronic Control Units (ECU) [1]. Growing size of the software poses challenges for delivering the updates, often requiring multiple decentralized ECUs to be updated at once, demanding more coordination, time and bandwidth than before.

Modern trend for automotive manufacturers is to perform the diagnostics and firmware updates remotely and over the air, minimizing the costs and customer inconvenience. According to estimates, 75% of the cars shipped worldwide by the year 2020, will have some form of wireless connectivity [2]. As such, the connected vehicle becomes part of the ever growing Internet of Things, and the manufacturers are adapting the paradigm shifts and methods to deal with quality issues, frequently changing requirements and the vast amount of data pouring in from the deployed vehicles.

1.1 Motivation

This thesis topic was proposed by the Iseauto [3] research group in order to accelerate the deployment of new firmware on the vehicle's self-developed controllers. Currently, the update process involves physically attaching the Joint Test Action Group (JTAG) cables to individual controllers and uploading the binary via the controller manufacturer's provided tools. The update is carried out by the member of the software team, which means traveling to the vehicles location. The In-Application Programming (IAP) of the low-level controllers is the first foundation for the further automation of the firmware updating.

The goal of the remote updating is to speed up the process of developing the newest software for the controllers, while giving the option to roll back to previous stable releases at any time. Given that the project is still in its development stages, the design has aimed for modularity and reusability. Having the modular design perspective in mind for updating protocol and software, allows adding, removing and changing the underlying hardware in the future, with minimal efforts required from the software team.

1.2 Objectives

The goal of this thesis is to develop a solution for firmware upgrading the vehicle's low-level controllers via the host computer. In order to achive this, the topic is divided into subtasks as following:

- Research state of the art in automotive firmware upgrading
- Develop necessary tools and scripts to prepare the compiled binary for upgrading
- Establish a memory partitioning scheme suitable for fail-proof updating process
- Create a modular bootloaders for controllers used in this project
- Develop a transfer protocol for upgrading, independent of the communication links or controllers currently used
- Test and verify the reliability and the performance of the developed solution

1.3 Thesis organization

This thesis is divided into six chapters, and the sections are divided as follows:

Chapter 2 presents background information on some of the challenges and current state of the art of firmware upgrading in the automotive industry. Also the overview of the Iseauto project, hardware and software tools used in this thesis is given.

Chapter 3 lays out the necessary steps needed in order to prepare the final binary before uploading to the controllers.

Chapter 4 describes the bootloading and memory partitioning techniques used in this thesis.

Chapter 5 presents the updating protocol and the software modules inside the controllers created to handle the said protocol. It also discusses the designed transfer protocol which is described from the perspectives of performance and reliability. This chapter also presents the performance measurements and reliability experiment.

Finally, Chapter 6 summarizes the thesis.

2 Background

Up until recently, the automobile manufacturers rarely updated the software on already sold cars and only did so in the most urgent of cases. This usually means, that a manufacturer issues a recall and provides the dealers with a new firmware release from the appropriate ECU supplier. The firmware distribution to the dealers is slow and costly process and the download to the vehicle is a manual process, which cannot be scaled due to the rather expensive specialized equipment needed. The update itself is rather slow, depending on the module size and serial protocol used – dealers are usually charging 1-2 hours of labor for the process. In 2016, the manufacturers and suppliers paid almost \$11.8 billion in claims and recorded \$10.3 billion in warranty accruals in the U.S. alone. That in total is an estimated 26% increase over the previous year. The number of recalls related to electronic and electrical systems have risen nearly 30% per year since 2013. [4]

To mitigate the recall costs, the upcoming trend is to provide Firmware Over the Air (FOTA) as a service, which optimizes the cost and enhances the user experience. The update could be performed at the dealership or at the customer location. The updates at the dealership could offer a more controlled environment and faster problem solution in case of any errors and thus, address the reliability and liability concerns of the manufacturers. The possible FOTA advantages include:

- Central firmware package management by the manufacturer – no need to distribute the updates to the dealership, which reduces time to market for new updates
- Parallel updating – since the process is independent of dealerships, the updates can be executed on many cars at once
- Shorter development cycles – since the updates could be deployed more times, it reduces the time the vehicle is operated under faulty conditions
- Forced updates – in case of safety related issues, the owners cooperation is not paramount
- Better user experience – even if the FOTA is still performed at the dealership because of the controlled environment, the update time is minimal and cost effective. [5]

The wireless updating does come with its own set of problems, one of them being security concern. Multiple methods for the FOTA have been proposed over the past decade. One of the earliest works, [6] proposes a generic mechanism for wireless update for the telematics and infotainment unit. In [7] Dennis et al. presented safety-security ECU classification, which assists in designing security for remote diagnostics and FOTA. In their follow up [8], a lightweight protocol was introduced, providing data integrity, data authentication, and data confidentiality. Security requirements were analyzed in [9] and it was shown, that the existing methods for FOTA do not address many aspects of security and a new framework was proposed, consisting of trusted portal issuing updates with verification code and a vehicle with virtualized control and functional systems.

The cost of the wireless update has also been under investigation. Up until now, manufacturers have used over the air updates over cellular network, which become expensive as the number of cars produced increases [10]. This is also true for the solution offered by Tesla, currently the only manufacturer providing remote updates, using mainly 3G and Wi-Fi [11]. Since the update procedures usually involve flashing the entire firmware over an existing firmware in an ECU, the updates can be slow and expensive. To solve that, [12] proposed method called delta flashing, which only flashes the changes between the old and the new firmware. The results showed that the time of re-flashing of the unit can decrease up to 90%. In order to deal with large number of ECUs, [13] introduced a system based on the IEEE 802.11s mesh standard, which showed that such a system could be used during the vehicle development or during the maintenance work.

In order to face these existing challenges, AUTOSAR [14], the global standardization consortium for automotive software architectures has launched an AUTOSAR Adaptive Platform. The goal is to provide developers a stable interface for Operating System (OS) and communication middleware for data exchange with local and remote applications. The core of the platform is the OS based on the subset of POSIX, which provides all the expected functionalities of the operating system, while preserving typical features originated in deeply embedded systems like safety, determinism and real-time capabilities. Currently AUTOSAR has both pros and cons. Overhead on run-time and memory due to standardized architecture causes increasing cost. OS, runtime environment and some other services are mandatory, even if a small system does not need all of their functionalities. The specifications of timing and test are still under development [15].

2.1 Iseauto project

Iseauto [3] is the first Estonian self-driving vehicle, developed in Tallinn University of Technology in cooperation with Silberauto AS and ABB AS. The vehicle is designed to take the role of last-mile transportation, operating mainly on the university's campus. The work on the project began on the January 2017 and officially ended in October 2018, and was divided into four stages: preliminary research, software development, body assembly and system tuning/testing of the self-driving car [16].

Building a self-driving vehicle is a multidisciplinary collaboration, involving mechanical and electrical engineering, mechatronics, electronics, computer systems and software engineering. The aim of the project for the university, was to increase competence in the self-driving vehicles and to get students involved in these fields, by assigning research topics and hands-on practical engineering tasks. This helps to structure the know-how about autonomous vehicles and self-driving algorithms at the university, while providing valuable practical experience for future engineers in Estonia. Also, the collected datasets provide a valuable asset for the evaluation of various sensors in local environmental settings [17]. Integration of the project work into the studies took place in two main ways: final theses and project courses. As a result, at the time of of writing this thesis, ten final theses (4 B.Sc and 6 M.Sc theses) [16], where the students examine various aspects of the Iseauto self-driving car have been published.

2.1.1 Technical overview

The final design of the bus relied on the Mitsubishi i-MiEV chassis (depicted in the Figure 1). All the molds and panels are specifically designed for the project by Silberauto. For sensing the environment two lidars are placed on both sides on top of the bus, third lidar in the middle of the car one meter above the ground, and cameras near the front and rear bumper. The outputs of the two are merged together to help map the environment around the bus and detect objects. For assistance with detecting objects up close, multiple ultrasonic sensors are placed around, and short distance radar, under the bus. All the sensor data is gathered in the main computer (host) which runs Robot Operating System (ROS), an open source middleware, which contains libraries and tools, used to create robot applications [18].

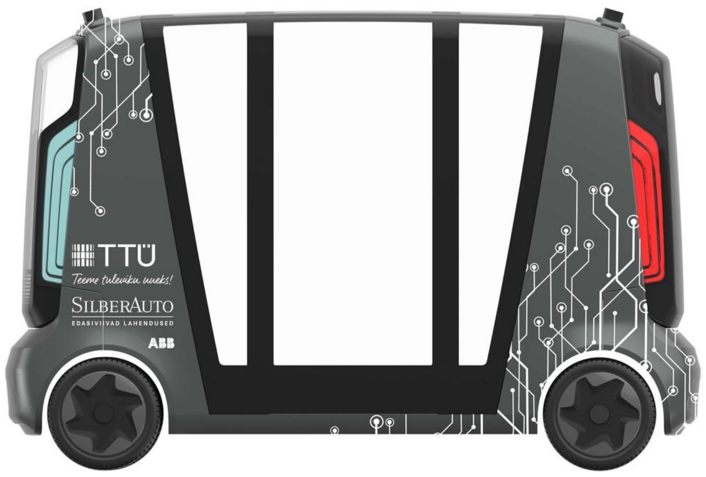

Figure 1. Final design of the Iseauto bus [16].

For controlling the bus itself and returning feedback to the host, the set of low-level controllers are used: a master controller, and the drive controller. The master controller is handling the communication with ROS and the Controller Area Network (CAN) network with other controllers and the i-Miev ECU. The drive controller is responsible for controlling breaks, steering, gearbox and throttle control. Set of body controllers handle car lights, climate control, ultrasonic sensors etc. The safety controller monitors the activity on the CAN bus and has the emergency break capabilities. The full architecture of the system is shown in Figure 2. As of writing this thesis, only master and the drive controller have been developed and deployed.

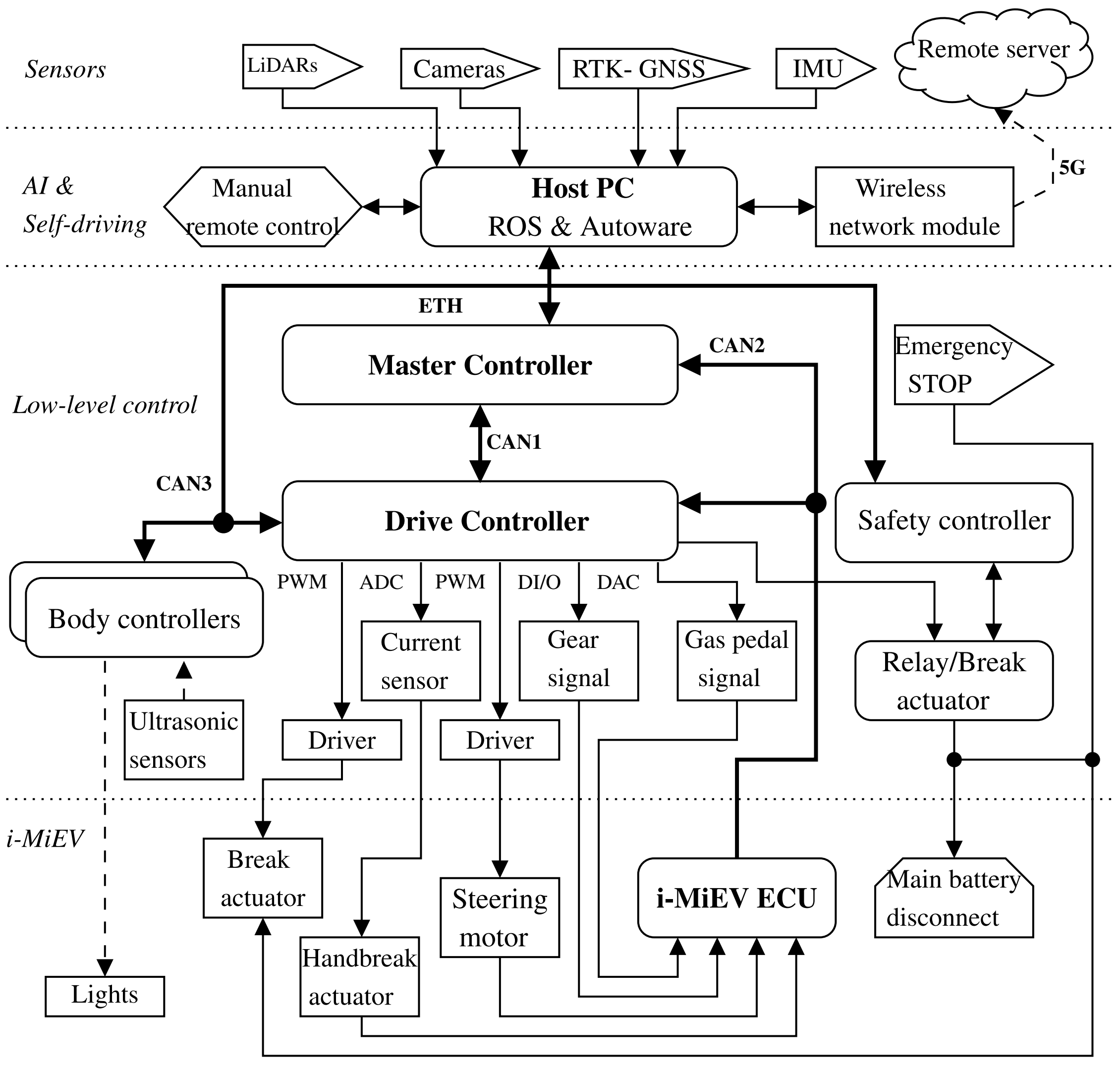

Figure 2. Architecture of the vehicle control system [17].

2.1.2 Low level controllers

The low level (master and the drive) controllers are both 32-bit microcontrollers based on the Arm® Cortex®-M processors, manufactured by STMicroelectronics. The master controller is STM32F767ZI [19], with Cortex-M7 core and 2MB of non-volatile flash memory and 512KB of Random Access Memory (RAM). The drive controller is STM32F407VG [20], with Cortex-M4 core and 1MB of flash memory and 192KB of RAM. Both of the controllers are more than capable of servicing CAN bus peripherals and performing any additional data processing before transferring data between the different nodes. The main features of the low level controllers are presented in Table 1.

Table 1. Key features of low level controllers.

Feature	**STM32F767 (Master)**	**STM32F407 (Drive)**
CPU Core	ARM Cortex-M7	ARM Cortex-M4
Flash memory	2 MB	1 MB
Non-volatile memory	512 KB SRAM	192 KB SRAM
Max clock frequency	216 MHz	168 MHz
Communication peripherals	4 x I^2C 4 x UART 6 x SPI 3 x CAN USB 2.0 Full speed 10/100 Ethernet	3 x I^2C 4 x UART 3 x SPI 2 x CAN USB 2.0 Full speed 10/100 Ethernet
Packaging	LQFP144	LQFP100

Neither of the microcontrollers are designed for automotive industry purposes. The chosen chips offer more general purpose peripherals than a special purpose automotive chip would, they do not offer additional safety features such as ECC memory, built in hardware self-tests, additional supply voltage and temperature sensors, and are not meant to run in extreme temperature ranges. The reason for picking these controllers were much more pragmatic – both of the controllers are sold as part of the development boards and were already available for the developers. Also the software team had previous experience with the said controllers and development tools. Since the solution proposed for ISEAUTO project did not have to scale to mass production, spending time on choosing the perfect controller, and then learning new tools and environments would have delayed the actual development too much.

The development environments provided by STMicroelectronics are free of charge: System Workbench [21] toolchain is an Integrated Development Environment (IDE) based on the Eclipse [22] which supports full range of STM32 microcontrollers and allows to compile, program and visually debug the program using the ST-Link in-circuit debugger [23], which is included on the development board. In conjunction with the IDE, the STM32CubeMX [24] is a graphical software configuration tool, is used to generate the controllers clock tree, peripherals and middleware stacks initialization code. The generator significantly reduces the development time, which otherwise would be spent on the peripheral drivers setup.

3 Firmware preparations

After compiling the firmware code, numerous actions can be carried out with the resulting binary file. This is usually called the post-build processing. The tasks can include, for example, encrypting the firmware, performing checksums and adding additional versioning and build information (metadata). This can also include running automatic tests, automatic version control operations and much more. The post-build program is commonly a shell or batch script and is usually designed to execute after every time, the developer builds the project.

3.1 Metadata placement

In order to store, and later retrieve the post-build information from the binary, the memory allocated for this information needs to be linked to a known address. One of the ways to achieve this, is to use compiler specific function attributes, to create a user defined section. This section is later referred to in a linker script, instructing the linker, how to place this particular section in the memory. The full example of this method is presented in Figure 4.

```c
/* metadata.c */
typedef struct {
  uint32_t crc;
  uint32_t hash;
} metadata_t;

metadata_t metadata
__attribute__((section
   (".metadata")))
__attribute__((used))={
  .crc  = 0xFFFFFFFF,
  .hash = 0xFFFFFFFF
};
```

```
/* linker.ld */
.isr_vector :
{
  /*Vector table*/
  . = ALIGN(4);
  KEEP(*(.isr_vector))
  . = ALIGN(4);

      ...

  /*Our metadata*/
  *(.metadata)
} >FLASH
```

Figure 4. Example: (a) using GCC attributes, (b) placing the section in linker file.

The metadata structure is assigned to appear in special section (".metadata") in the source file (metadata.c) with the help of compiler keyword *__attribute__(section)*. The other attribute *"used"* informs the compiler that the variable is to be retained in the object file, even if it is unreferenced, in order to avoid optimizing it out. The section can then be used in the linker file (linker.ld) to map it in the desirable location in the memory. The main advantage of this method, is that the location of the data can be arbitrary, for example some controllers have internal an Electrically Erasable Programmable Read-Only Memory (EEPROM) memory sections, where the settings and metadata could be stored.

Alternatively, if the address of this data is not required to be somewhere specific, as in some other memory area (EEPROM, or any external memory chip), the top of the vector table could be used instead. This provides a known location – somewhere at the beginning of the binary, depending on the size of the vector table. In short, the metadata is placed after the last interrupt vector in the vector table structure (shown in Figure 5). This technique is used, for example, in Texas Instruments Tiva (formerly known as Stellaris) series controllers, for storing Cyclic Redundancy Check (CRC) information. The first two words are set markers 0xFF01FF02 and 0xFF03FF04. This way, external parser tools can locate the start address of the data. After that, the fields of the metadata follow. In this project, three 32-bit words are used for information storage, but technically, the memory size is the only limitation on the amount of data stored there. The word 0xFFFFFFFF is a placeholder value, this information is filled in later by the binary parser tools.

```
/* startup.s */
g_pfnVectors:
  .word  _estack    /* Stack pointer */
  .word  Reset_Handler
  .word  NMI_Handler
  ...
  .word 0xFF01FF02 /* Prefix 0 */
  .word 0xFF03FF04 /* Prefix 1 */
  .word 0xFFFFFFFF /* Image length */
  .word 0xFFFFFFFF /* CRC32 */
  .word 0xFFFFFFFF /* Git hash */
  .word 0xFFFFFFFF /* Reserved */
  .word 0xFFFFFFFF /* Reserved */
  .word 0xFFFFFFFF /* Reserved */
```

Figure 5. Packing metadata into the vector table.

3.2 CRC-32

CRC is widely used error-checking code in data transmission systems based on polynomial manipulations using modulo arithmetic. The motivation to use CRC in firmware binary, is to check for either transmission errors, flash memory write errors and integrity of the flash memory over time. The algorithm and polynomial used has to be identical when first calculating the value after building and on the device later calculating the image again.

CRC result is calculated over the whole binary image, except for the 32-bit CRC32 word, stored in the vector table. This field is skipped, because when the first calculation is done by the *binpack*, the field contains placeholder value 0xFFFFFFFF. After the calculation, the field is replaced with the correct CRC result for the firmware binary. All the subsequent CRC's that are calculated inside the controllers, ignore this particular word as well. This word is later used to compare the result with the calculated check result.

For calculating the first CRC value and storing it into the binary, the *Binpack* tool is used. *Binpack* is simple, open source command line tool, supplied by Texas Instruments, as a part of the Tiva Firmware Development Package. *Binpack* takes a binary file input which contains structure described in Figure 5, calculates the firmware image checksum using CRC-32 algorithm and finally overwrites the placeholder value in the binary with the CRC result. Image length is also stored alongside with the CRC result.

3.3 Commit hash

Git is a distributed version control system used primarily for source code management. Since its release in 2005, it has become the most used source code management tool [25]. Since it is easy to get started with Git on all operating systems and because the software team had previous experience with it, the natural choice was to use it on ISEAUTO project as well.

With Git, each commit has a digital signature associated with it, calculated using the Secure Hash Algorithm 1 (SHA-1) hashing function. Embedding the commit hash into the binary gives developers simple way to track firmware version on the device, without going through the trouble of version labeling each iteration. This proves to be very convenient, especially in the early development phase, when the code changes are very frequent. The commit message acts as a minuscule release note, making it even faster to look up changes made to the devices firmware on the field.

In order to embed the commit hash into the binary, a small modification to the *Binpack* tool, described in Section 3.2 was made. Since the parser already took a binary and found the right location for the CRC and image length field, adding the extra hash field to replace the original reserved field was trivial. The parser now takes extra input flag and hash value. It should be noted, that the short SHA-1 hash is used, because of the 32-bit word allocation in metadata field. In most projects, using the short hash should be sufficient to avoid collisions [26, p. 217].

3.4 Appending the bootloader

For the release binaries, which are flashed onto the empty chips with In-Circuit Emulator (ICE), it is necessary to program at least two images at the right locations – bootloader and the application. This process could be done in two steps: load the bootloader onto the chip at the correct address and then do the same for the application. This is cumbersome and potentially more time consuming, when mass producing the devices. In order to simplify this process, one of the last steps of the post build can be concatenating the application and the bootloader images. For this to work correctly, the location of the correct bootloader binary has to be passed into the script as a parameter. Also the correct amount of padding has to be inserted between the bootloader and the application, in order to place application start at the right memory address. For this, the application start address is also required by the script to calculate the padding. The padding bytes used are of value 0xFF. The figure 6 illustrates the release binary layout.

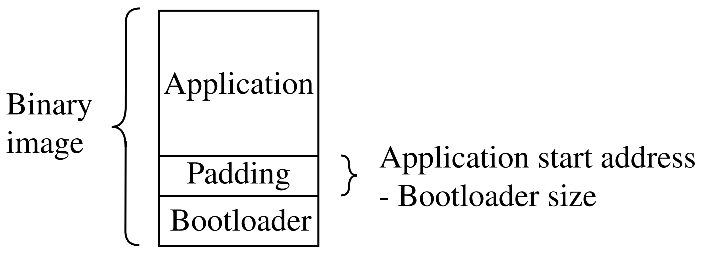

Figure 6. Release binary file layout.

4 Bootloader

When the microcontroller is powered on, before the main application of that device is run, small application is executed before it. This application is called the bootloader. For larger systems, the task of the bootloader is to initialize the low level peripherials and setup system for the OS kernel. In embedded systems, the bootloader typically has two main functions: provide an updating mechanism via any communication peripheral available, and handle application integrity errors, with CRC for example.

Bootloaders usually come with two specific constraint: size and location. In embedded firmware, the program size is always a consideration. Since for mass production, it is usually prefered to save a few cents per unit, and flash size corresponds directly with the cost of the microcontroller. This means that bootloader should be kept as small as possible to reserve space for the main application. They also commonly run in Read-Only Memory (ROM), One-Time Programmable Memory (OTP), EEPROM or in flash memory-protected area, which means that the bootloader itself can not be overwritten. Therefore great care must be taken, when implementing the bootloader, because any bugs present in code will remain in there for the whole product life-cycle.

4.1 Memory partitioning

Every microcontroller has some kind of on-chip non-volatile memory, flash technology being the most prevalent. Although the memory is often presented to the programmer as a one big chunk, it is wiser to divide the memory into distinct sections. When mapping out the device memory, it is important to know, how the memory is organized internally. Typically the memories are divided into blocks (sectors) consisting of multiple pages. Memory organizations for both chips used in this project are shown in Appendix 1.

Each device also specifies the memory write and erase granularity. While both controllers used in this project allow byte, half-word and word write, the minimum flash erase width is the memory sector. This is because erasing the cell requires higher voltages and to save the die area, this erasing logic is kept to minimum. In NAND and NOR architecture flash memory, the erased (default) cell is usually at logic level '1'. This means that when writing into the flash memory, bits can be only set to logic '0'. This means, that before overwriting any flash area with new data, it must be erased first. Therefore, because erasing can be done per sector, the partitions for bootloader and application area are bound with the sector sizes.

After setting the partition boundaries, it is time to decide, how to actually divide the memory into different areas. The most simple way would be to cut it into two areas for bootloader and application (shown in Figure 7a). This approach has the benefits of having a separate bootloader for the application image verification before running the main application. It also leaves most of the memory space to the application image. However, in this arrangement, it is the bootloaders task to communicate with the outside world, in order to get the new firmware. Also because there is no dedicated buffer for the application, before fetching the new firmware, old application has to be erased. This may leave the system without any firmware, in case of any flash erase or transmission failure. To counter these failures, the application memory space is split into two parts: main area, where the current firmware runs from and buffer for the new firmware (Figure 7b). This buffer can also act as a backup version. The main idea in this setup, is to have a working firmware present in somewhere in the memory at all times. The downside is that memory area for the application, which could be scarce resource in cost-driven design, is cut in half. The upside is that it is possible to designate firmware fetching to the application.

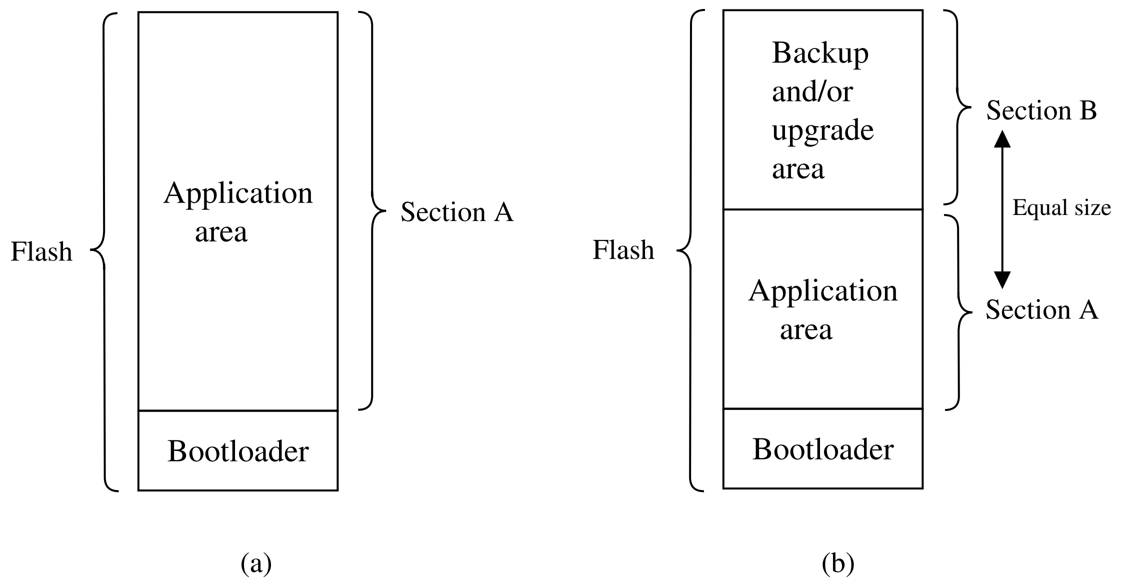

Figure 7. Typical memory partitioning schemas: (a) Single application, (b) Two applications.

Dividing the memory into two application areas, provides the update module software the opportunity to verify the new firmware before erasing the old one from the memory. When the image verification is successful, the bootloader can then erase the old image area and then copy the new firmware to a working memory area. The entire time flow of an updating process in this configuration can be seen in Figure 8.

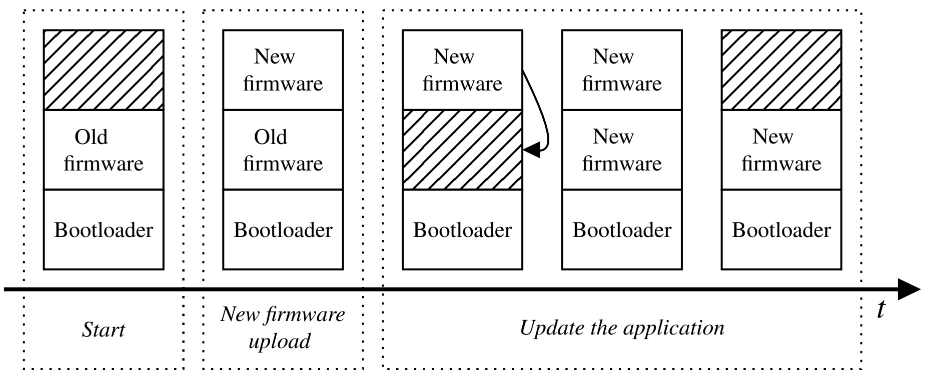

Figure 8. Two partition schema update time flow.

The memory could be divided up even more. Doing so cuts into the maximum individual image size yet again. Although the two application areas described previously, solve most of the reliability issues when upgrading the firmware, three partitions for application image introduces more possibilities. For example it is possible to keep one area for 'stable' firmware and other for more experimental one, which could prove to be useful, especially in the development phase. Other uses might include storing a factory default firmware image, or keeping two separate 'modes' of firmware, which can be chosen from during the booting phase.

Since in this project, the flash memory is abundant – 2 MB total memory in master controller and 1 MB on drive controller, the usual limitation of the memory size is mostly irrelevant. At the time of writing, the master controller main application is 79KB (ca 4% of total memory available) and drive control main application 32KB (ca 3%). Considering all this, it was decided, to divide the controllers flash memory into four areas: bootloader, application, backup and upgrade image area. Separate upgrade image area allows to implement IAP while keeping the backup area intact. This ensures that if any failure during memory erasing or writing occurs, backup version of the firmware can be restored. This also leaves developers an option to restore previous version quickly when testing. All memory areas are the same size, except for the bootloader. This limits the program size to 512KB on master controller, and 256KB on drive controller. The final memory is layout presented in Figure 9.

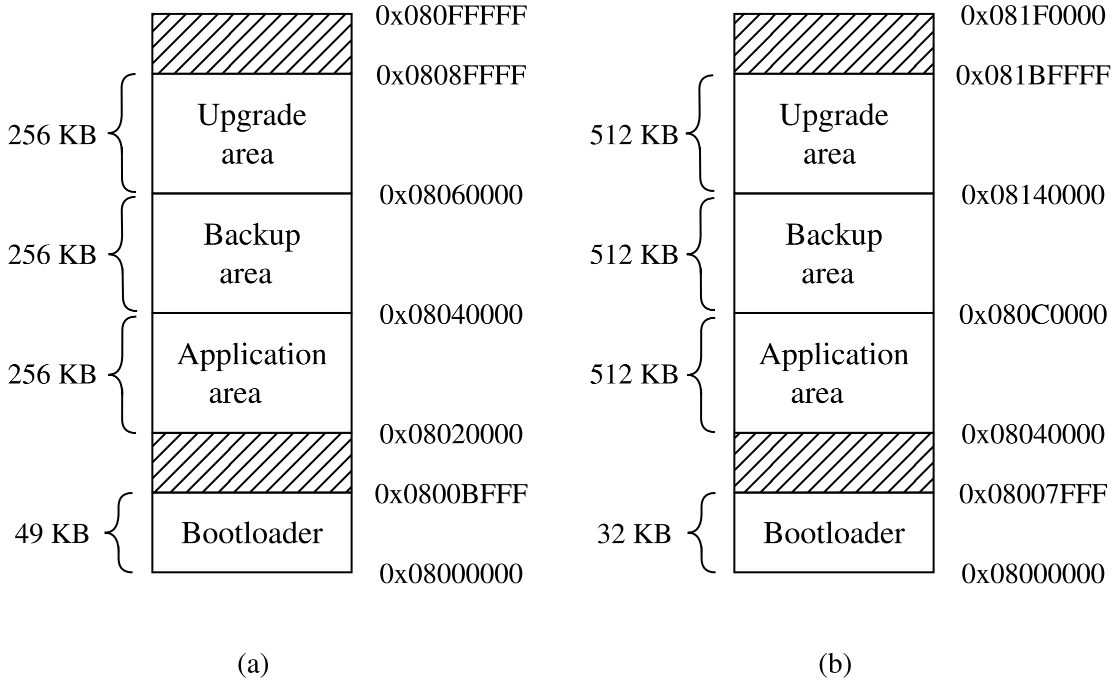

Figure 9. Controllers flash memory layout: (a) Drive controller, (b) Master controller.

Using three partitions means separate buffer for the new firmware, so the backup is not destroyed while fetching the update. When the new firmware is verified, the old application is moved to backup area. Complete time flow of the three partition update is shown in Figure 10.

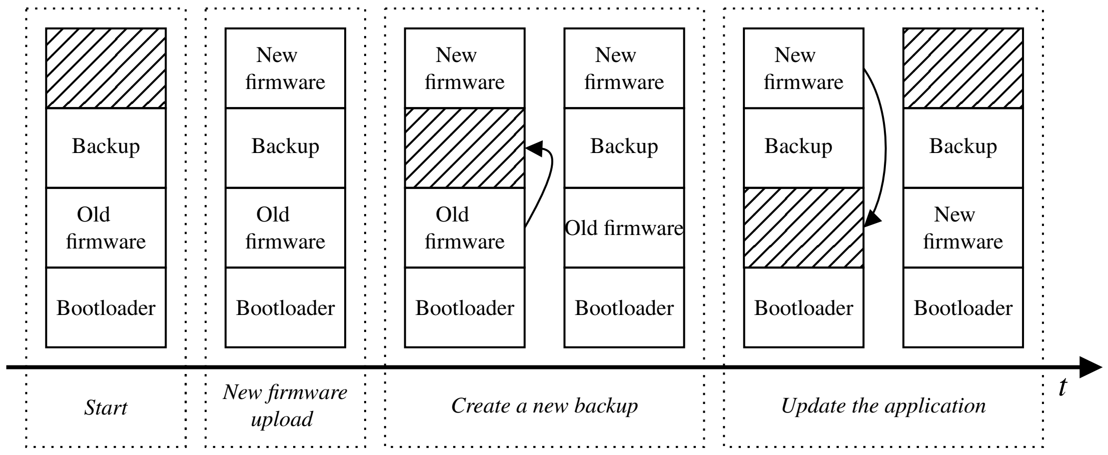

Figure 10. Three partition updating time flow.

4.1.1 Dual banking

Previously, a single bank of memory was partitioned into distinct areas by the programmer, but many controllers offer a dual bank structuring on the hardware level. Choosing this mode alters the sectioning of the memory, basically doubling the number of sectors, while reducing the total sector size by half. Dual bank mode also supports the dual boot mode, which is used to select the bank to boot from, by mapping different physical banks to boot program area. Both of the STM32 devices used in this project support dual banking and booting.

Using the dual banks mode introduces a Read-While-Write (RWW) capability, that is, to read one bank, for example, the main application itself, while the other bank is being written to (for example buffer for the new firmware). However, it is not possible to execute an erase or program operation on one bank, while erasing or programming the other bank (except for a mass erase that erases both banks at the same time). In a single bank mode, while the flash memory is being written or erased, the Central Processing Unit (CPU) execution is stalled. For example, when the interrupt arrives during the write/erase operation, this interrupt is not handled until the flash operation is completed. However, if the dual banking mode is used, the CPU can execute reading from the other bank without stalling [27]. RWW differences in single and dual bank mode are demonstrated in the Figure 11.

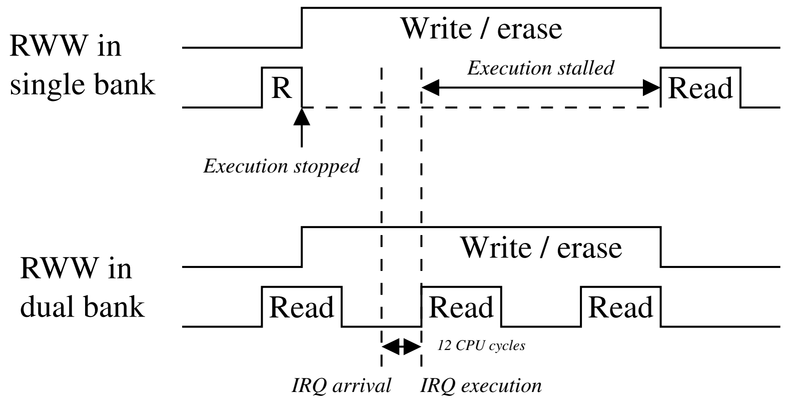

Figure 11. RWW in Flash memory single bank versus dual bank [27].

4.2 Embedded bootloader

Both STM32F4 and STM32F7 have their own built in bootloaders located in the system memory. These bootloaders have multiple interfaces available for uploading new firmware. To access the embedded bootloader during the startup of the system, the physical BOOT pins must be set accordingly. Each of the peripheral has its own firmware upload protocol. The advantage of using the embedded bootloader is that its already implemented and takes no extra memory space when used. STM also provides PC side software to upload the new firmware to the device for each peripheral available. The main disadvantage is that there are no safety features such as CRC, encryption or, as with the case of STM32F7, no ethernet peripherial available inside the bootloader. Using the onboard bootloader also means introducing a different mode of operation for updating the device – the application code cannot run during the upload.

4.3 Custom bootloader

Building custom bootloader gives developer a free reign over the bootloader design.

One of the first considerations, when writing a bootloader, is size. Since in this project, the IAP route was chosen, no communication peripheral for the software upload has to be initialized and used, compressing the bootloader size significantly. Only Universal Asynchronous Receiver-Transmitter (UART) is used for debugging purposes. This leaves most of the bootloader code dealing with flash memory manipulation and CRC calculations. Because the two devices are very similar, the goal was to share as much as code between them, as possible. This is achived mostly by using the same control flow and common utilities – such as CRC algorithm module. For flash operations, the STM Hardware Abstraction Layer (HAL) is used, which provides identical interface for both of the devices. This means that only hardware initialization and memory map of the device has to be ported from one device to another.

The main task of the bootloader implemented in this project is to perform CRC checking of the images and move firmware images around the memory. In order to achive the former, the decision has to be made between using hardware peripheral and software implementation. Hardware CRC unit is potentially faster and might use less code. With software CRC, the same code described in Section 3.2 can be reused, which guarantees the integrity of the calculation result. Because STM32F4 used for drive control, lacked the necessary hardware unit, and to reuse preexisting code as much as possible across platforms, in this project, the software approach was chosen. Every image area is checked for image length CRC value and commit hash, placed there by the post-build process described in Section 3. The calculations are performed again in bootloader and compared against the ones in binary for every image area. With this information, the correct flash manipulations can be performed. The full program flow of the implemented bootloader is presented in Figure 12.

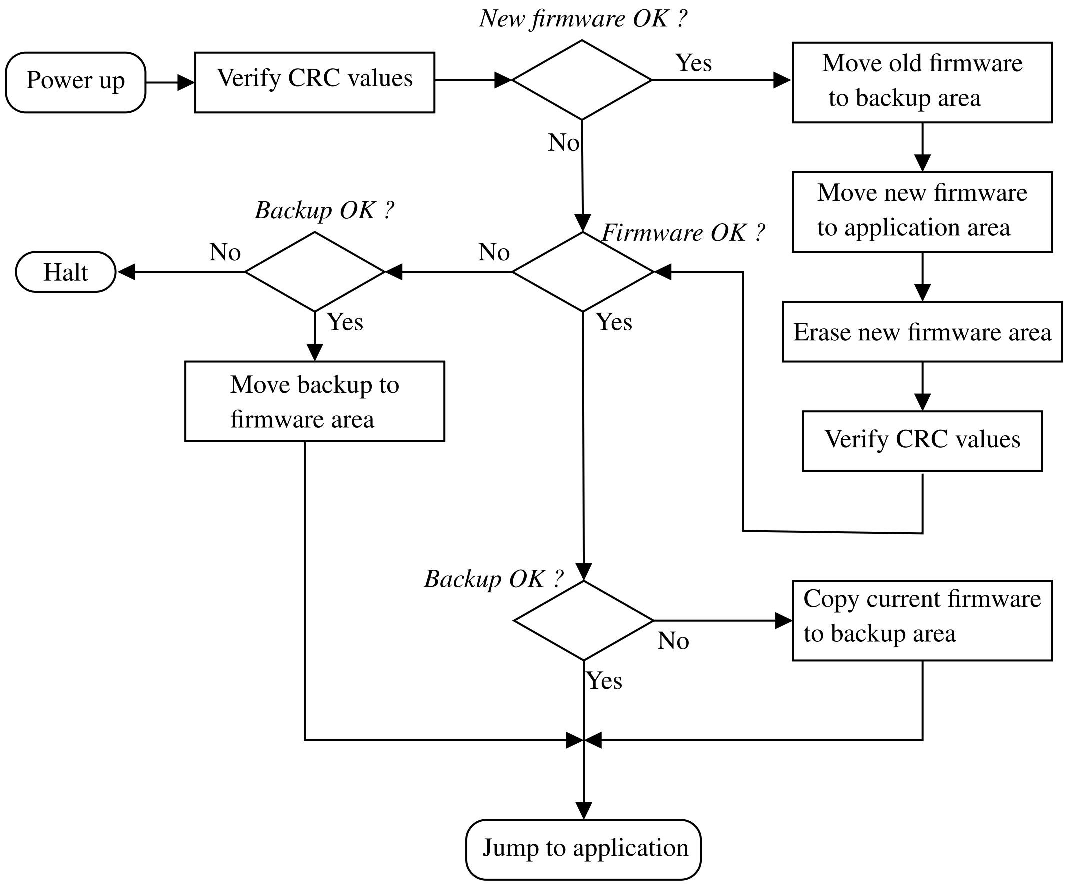

Figure 12. Bootloader sequence flowchart.

In case the upgrade image is available and the CRC is verified, the upgrading branch is executed. First the old application is moved into the backup area. This ensures that if anything happens during the flash writing of the new firmware, we have a working program available for a version roll back. Second step is to move the new firmware into the working application area. As a third step, the upgrade area is erased, to prepare the area for future firmware upgrade. It should be noted, that the every application move operation includes deletion of the destination area. As a final step of upgrading, the verification of the images is preformed again. Although flash erases and writes are verified at the byte level by HAL, this form of redundancy can be allowed, because of the CRC calculation speed. From there on, program continues to execute from where it left of prior to upgrading. The subsequent checks are for current firmware and backup. If firmware is corrupt, the backup will be loaded into the firmware area and vice versa. The goal is to have atleast working version intact while the other area is being erased or written to.

4.4 Vector table

Vector table contains stack pointer and all the exception handler vectors. Minimum alignment for ARM Cortex M-4 and M-7 family is 32 words, enough for up to 16 interrupts [28][29]. Every vendor configures the top range value depending on the number of peripherals used. When the processor is reset, the Main Stack Pointer (MSP) is loaded from the first word of the vector table and the Program Counter (PC) is loaded with second word, which directs execution to the reset handler. Vector table is fixed at the address start of the address space (0x00000000), however it could be relocated, using the Vector Table Offset Register (VTOR).

Since the bootloader is just like any other application, it has its own vector table, which is linked to address 0x00000000. When device powers up, it continues to execute bootloaders reset handler. This also means, when the bootloader uses any peripheral, for example UART, the interrupt handler address has to be located in the bootloaders vector table. At the end of the bootloaders execution, before the control is transfered to the application, the vector table address must be relocated, using VTOR.

5 Upgrading

Firmware upgrading consists of two main parts: uploading the new firmware onto the device, and then physically writing the new firmware into the non-volatile memory. In many cases, the uploading part is done by the bootloader. This way, the main program is not executed while the device is being upgraded, and communication peripherals and flash writing is taken care of inside the bootloader. Another approach is to perform these operations in application program, while the main application is also being executed – this is called IAP. In this project a kind of hybrid of the two is implemented, as the uploading is done by the application and the final relocations of the images by the bootloader.

In the application, the parts responsible for updating the firmware are separated in layers. The highest layer is Real-time Operating System (RTOS) task, which delegates the data to the update module. Tasks such as writing to the memory, keeping track of the packets and relaying the packets to the next node are taken care of in the update module. Updating software hierarchy in the application is illustrated in the Figure 13.

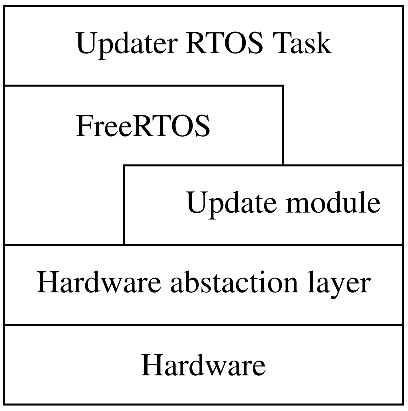

Figure 13. Updating software hierarchy.

5.1 RTOS Task

In this project, the FreeRTOS is used as the RTOS. Benefits of the FreeRTOS include being free (distributed under MIT license), is portable to many platforms, has small memory footprint (6K to 12K) and active and large community support [30]. Therefore, the updater has its own task, that is scheduled to execute by the RTOS when the correct data arrives to the communication port. The RTOS task is also responsible for tracking the time between the protocol frames, to issue a timeout error if necessary.

The updater task has to be designed independent of the communication peripheral used to carry out the updating routine. This way, changing or adding the communication channel is relatively simple and standalone task. To achieve this, the bridge between the task and the transmission is a queue data structure, where the communication peripheral produces and pushes the incoming data into the queue and the updater task consumes it. As such, the RTOS can schedule the tasks separately, depending on the workload of the other tasks and priorities. The Figure 14 illustrates the interaction between the communication link and the updater task.

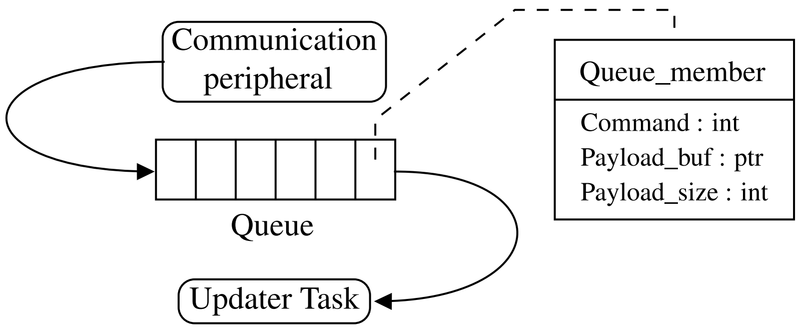

Figure 14. Decoupling of the task and communication system.

5.2 Updater module

Updater module is controller specific component whose purpose is to:

- Report the versions of the present firmware
- Negotiate the parameters of the updating process
- Store the incoming data in non-volatile memory in correct order
- Transfer the data to the next node, if necessary
- Acknowledge the successful or unsuccessful data transfer
- End the transaction and clean up

To carry out these tasks, the updater has to have some internal states to keep track of the updating process. Figure 15 illustrates the Finite State Machine (FSM) implemented.

After the device is powered on, the module waits for either version query or the initialization frame. If the valid initialization parameters are passed to the module, it is set up and ready to receive new firmware packets. The received packets are written in the RAM cache (if allocated), and written into the non-volatile memory when the cache is full or flushed manually. The packets are received and processed until the final packet is arrived, at which point the remaining cache is flushed into the main memory, and the module returns to inactive state after performing a cleanup routine.

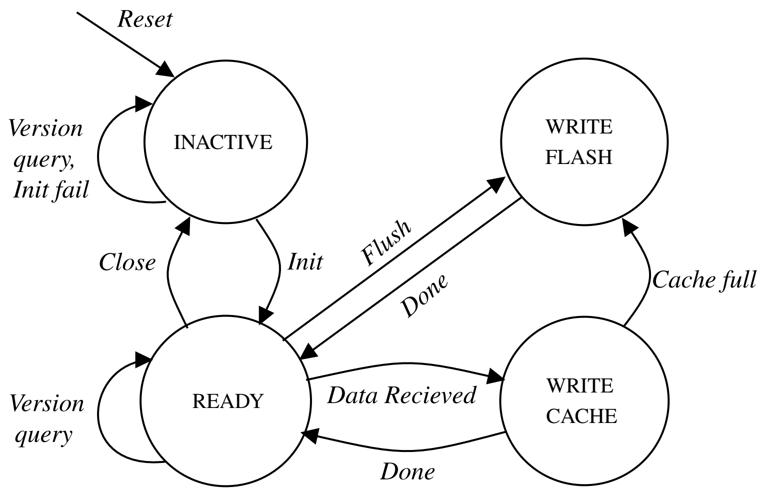

Figure 15. Updater module state machine diagram.

5.3 Transfer protocol

The updating process has to follow an exact steps in order to successfully deliver the new firmware image. The internal state machines are designed to first initialize the update, receive the image frame by frame and finally perform cleanup. These steps are also reflected in the packet transfer protocol, which is divided into three stages accordingly. The overall protocol has to be designed such that the data transfer is reliable, fast and is applicable to any underlying physical communication layer on which the transfer takes place.

When updating the devices, all the communication is going to or through the master controller. Master controller has the information about the other devices ID's and their communication peripherals. For updating the drive controller, the master prepares the arrived frame as a CAN bus message and sends it immediately to the drive controller. The data frames are further explained in the chapter 5.3.3. A successful transfer between the host and the drive controller is illustrated in the Figure 16.

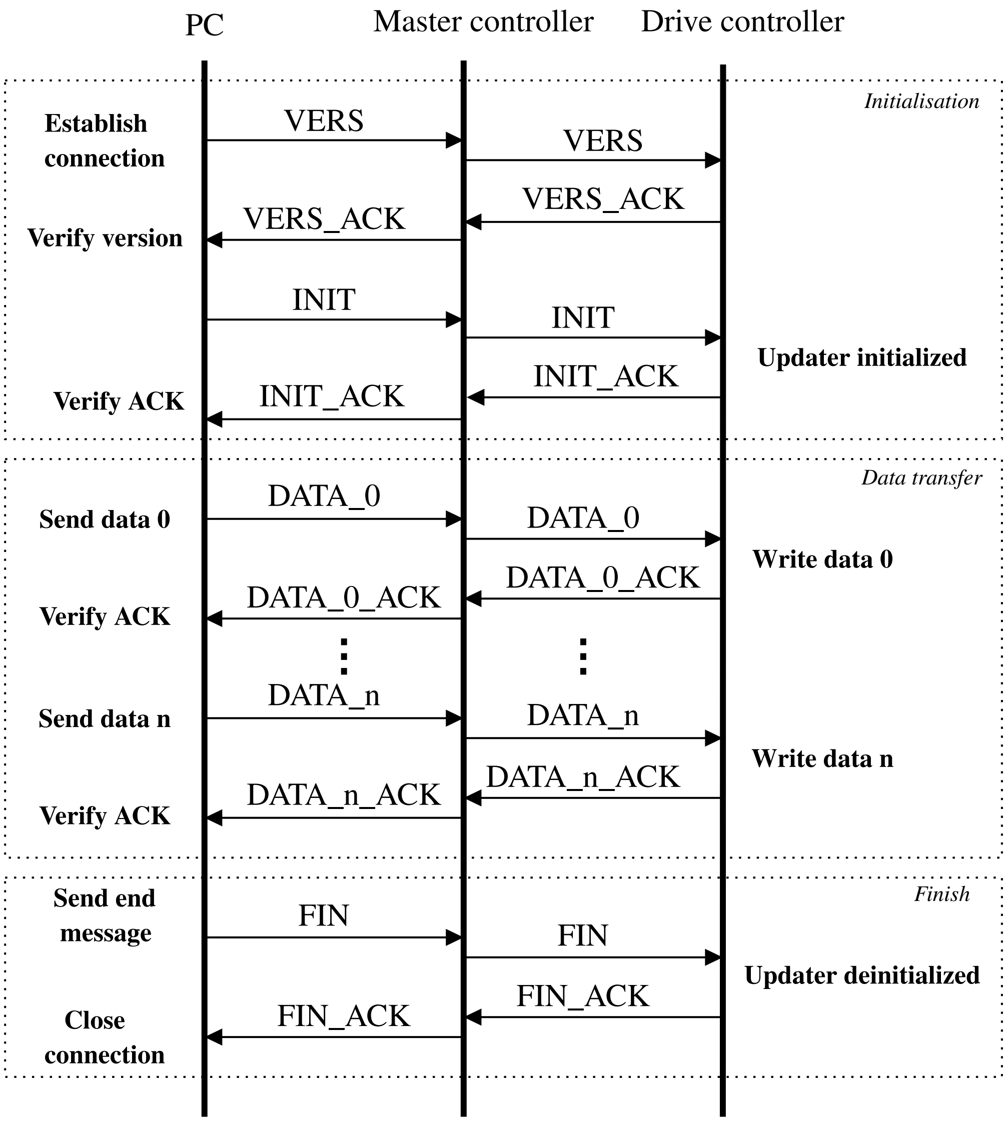

Figure 16. Drive controller successful update sequence.

5.3.1 Reliability

Transferring the data from one node to another requires techniques to prevent errors along the path. Although the whole image is always accompanied with the CRC, as discussed previously in Section 3.2, and retrying the whole upgrade process is an option, this is far from satisfactory. Most of communication standards offer reliable transfers, which is usually obtained using error detection/correction code, numbering and acknowledging the packets [31].

The purpose of numbering the packets is to assure the correct ordering of the data. Each data packet gets a unique sequence number, which is transferred alongside the payload. This way, when losing a packet or having two packets arrive in the wrong order, the device which is updated can ignore the frames and not corrupting the new image. This is illustrated in the figure 17.

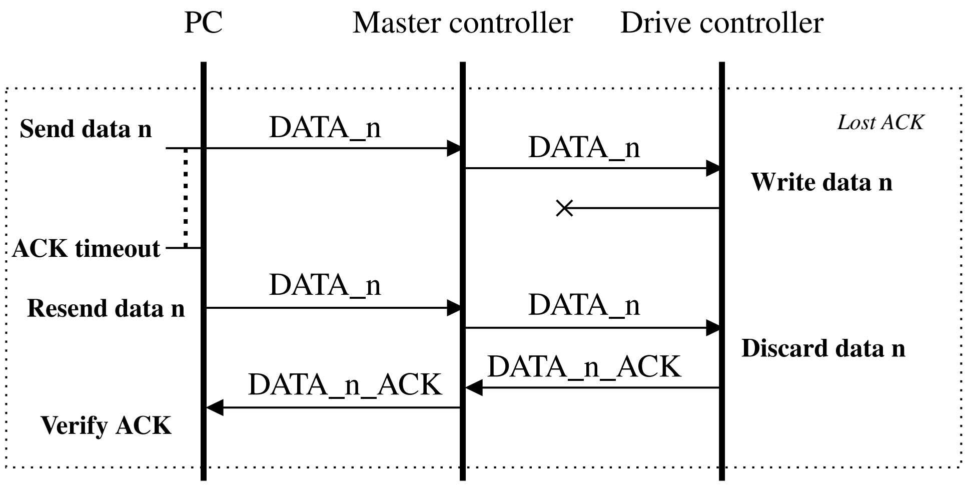

Figure 17. Error control: lost acknowledgement.

Sequencing the packets coincides with acknowledging the packets by the receiver. For every frame that is sent out, the sender waits for the receiver to acknowledge it. This allows to keep track of the time of flight and after a set time, retransmit the frame. Losing data along the way is mitigated, as illustrated in Figure 18. The downside is considerable overhead in data frames and lowered transmission rates, because of the acknowledgement traffic. In case of the CAN bus or Transmission Control Protocol (TCP), the acknowledgements are implemented in the transport protocol, but for example, when using plain UART, Serial Peripheral Interface (SPI) or even User Datagram Protocol (UDP) protocol, the acknowledgements have to be implemented in the application layer.

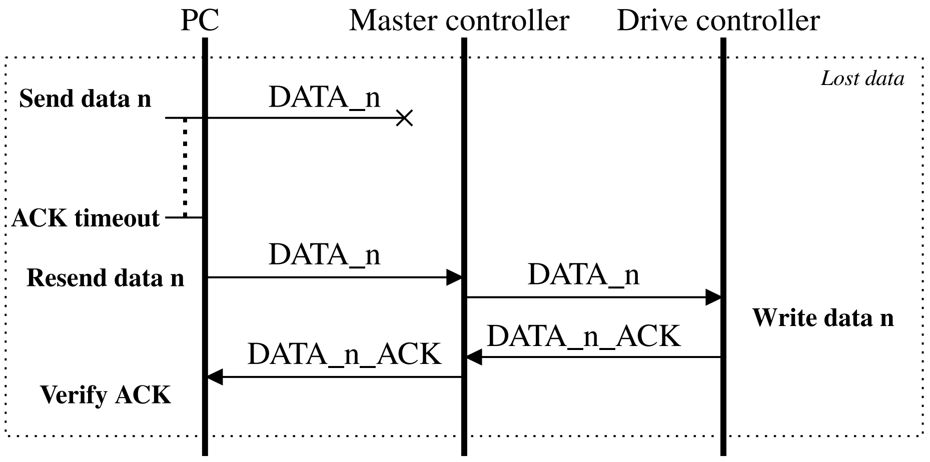

Figure 18. Error control: lost data frame.

Setting timeouts between the packets allows diagnosing a dead connection during the exchange. Every device has its own timeout counters and the host also sets the limit on number of packet retransmissions. This way, the updater state machines are not left hanging in some intermediate state, and the whole updating process can be restarted once the connection is reestablished, without having the need to power cycle the devices, or worse, the whole system. One of the possible situations is depicted below, in the Figure 19.

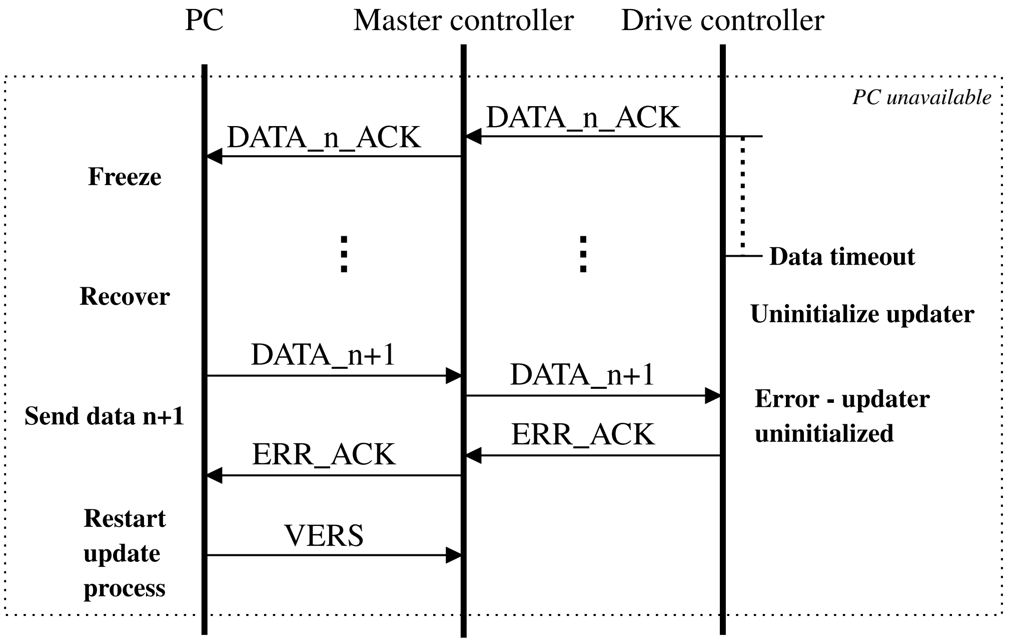

Figure 19. Error control: host unavailable.

5.3.2 Physical layers

The underlying physical layers have a huge impact on the final performance of the updating protocol. Since the goal of the proposed updating protocol was to be independent of the physical layer, the reliability had to be guaranteed in the upper layer protocol. As a result, some of the strengths and weaknesses of the used physical protocols are amplified. The unique frame sequence number, for example, is not a problem for protocols with larger payloads or dedicated sequence fields, but with small payload protocols such as CAN bus. The CAN bus is used for communication between the master and the drive controller, and potentially between any node added to the system in the future developments. The CAN bus was designed not send large blocks of data, but for large number of nodes to broadcast many short messages to the entire network [32]. Since CAN protocol does not include sequence number in their header (depicted in the Figure 20), it has to be embedded into the payload, which may contain maximum of 8 bytes. To avoid cutting into the useful data portion too much, the sequence number is allocated one byte, and has to overflow in every 256 frames. On the upside, the updating command and the device identification number can be encoded inside the 11 bit identifier field.

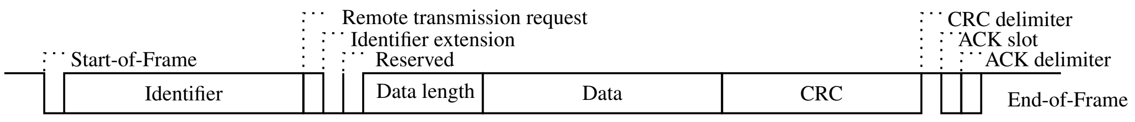

Figure 20. CAN base frame format.

As an alternative link between the master and the drive controller, the UART connection was set up. UART is easy to add to the system, as it requires only two (or possibly only one) wires to connect between the controllers and both of the controllers have many UART peripherals available. UART offers low frame overhead, and fast transfer rates, and is suitable for updating purposes as our transfer protocol will guarantee the data reliability on the higher layer. One possible downside to using UART is scalability. In case there will be many controllers added to the system in the future, separate peripherals may run out. Then some type of master/slave type communication such as SPI could be used instead. The performance difference between the CAN and UART is further discussed in the section 5.4.

The connection between the host and the master controller is using Ethernet as its link layer and, at the time of developing and writing this thesis, the UDP protocol is used for the transport layer. The UDP is a connectionless and unreliable datagram packet service. Although it provides CRC for data integrity, it has no handshaking mechanism, data packets are not guaranteed to be in order or to reach their destination at all. The UDP protocol implementation was in place as a result [33], where the TCP connection was deemed too slow for ROS messages. In the case of data reliability being important, using UDP is less than optimal and if the Ethernet as a underlying layer is used at all, the switch to the TCP stack should be made. Implementing the TCP stack however, was out of the scope of this thesis. But by using the UDP stack, the data delivery should be faster, due to the low overhead of the datagram frame and lack of acknowledgements for each packet. Also, it serves as a good demonstration, that the solution and protocol developed in this project can and will work with unreliable transport layer, and if necessary, could be used with any communication stack.

5.3.3 Protocol frames

The updating protocol is initiated by the host machine. Each frame is prefixed with 8-bit command sequence and 16-bit unique device identification. This allows to extend the protocol to more than just the 2 devices currently used and also to add more commands if needed. The 8-bit command and 16-bit identification field's encodings are shown in the Table 2.

Table 2. Protocol frame field encodings.

Frame section	**UDP/UART (Hex)**	**CAN (Hex)**
VERS	0xF0	0x100
VERS ACK	0xF1	0x104
INIT	0xF2	0x101
INIT ACK	0xF3	0x105
DATA	0xF4	0x102
DATA_ACK	0xF5	0x106
FIN	0xF6	0x103
FIN_ACK	0xF7	0x107
Master controller ID	0x0001	0x000
Drive controller ID	0x0002	0x008

The CAN encodings are different to UDP and UART encodings, because the update command and the controllers ID are placed inside the 11-bit identifier field (Figure 21). As of writing this thesis, the communication between the master and the drive controller takes place on dedicated CAN bus (CAN1, shown in Figure 2), which means it is not necessary to encode the device ID inside the field. As the bus is dedicated, the acknowledgement messages do not really need separate encodings either, because the master always knows, that the messages on CAN1 are from the drive controller. In the current implementation, all of the CAN messages are divided into message groups, where the lower group number means higher priority. This solution ables to connect up to 31 slave controllers for each of the master controller's CAN peripheral.

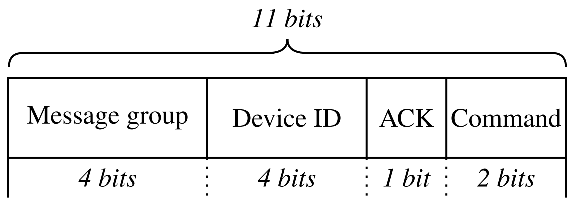

Figure 21. CAN frame identification field for the update messages.

First it sends out an version query to gather information about the firmware in device under update. The current implementation has the device responding with CRC hashes of the main application and the backup, but also could be instead returning version numbers or git hashes as well. The host side compares the versions (possible implementations include version-hash lookup table) and decides whether to start the upgrading or not. The version query and acknowledgement frames are illustrated in the Figure 22.

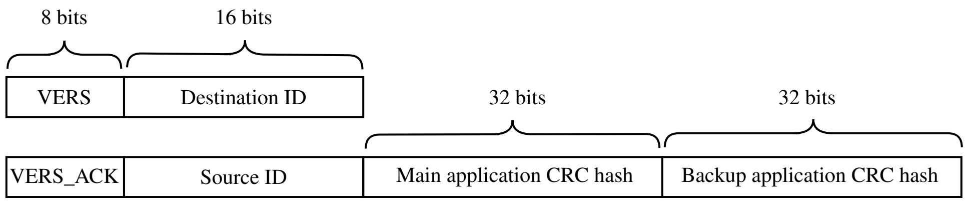

Figure 22. Version query and acknowledgement frames.

Next up is the initialization packet. The host sets a cache size and the total image that is going to be sent. The device responds with the desired payload size of the data packet and an error code, if there are any complications with the initialization packet, otherwise set to zero. The effects of the cache and data packet size are discussed further in the Section 5.4. Host sent initialization and the receiver acknowledgement frames are illustrated in the Figure 23. After setting up the receiver, host starts sending the data frames.

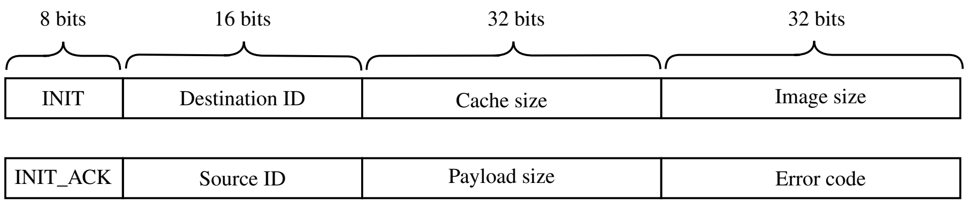

Figure 23. Initialization and acknowledgement frames.

The actual data frames are fairly straight forward. The frame is prefixed with the header, followed by sequence number of the data packet and then followed by the data. Length of the data is determined in the previous exchange by the device. Each data frame is acknowledged with the same sequence number as the received packet and an error code, set to zero if everything went well. Total number of data frames comes out as total image size divided by the desired packet length. Figure 24 illustrates the data and the corresponding response frame.

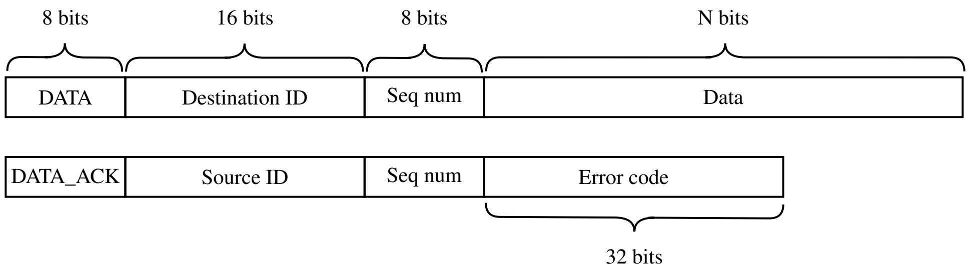

Figure 24. Data and acknowledgement frames.

The last frame sent to the receiver indicates, that all of the data has been sent. Updater then confirms it with total bytes written and error code. After this, the cleanup routine is executed, freeing memory and terminating the RTOS task. Final frames are illustrated in the Figure 25.

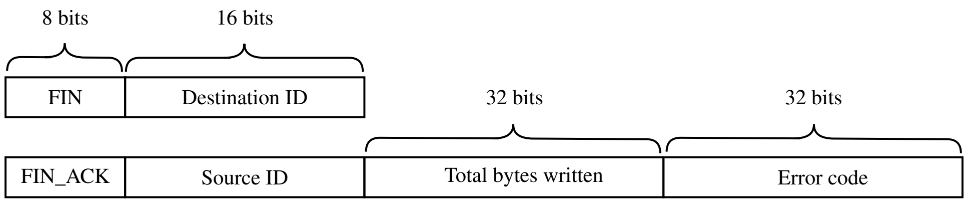

Figure 25. Finish and acknowledgement frames.

5.4 Results

The most important parameters of the updating protocol are the reliability and the speed of the transfer. The speed is mostly determined by the protocol used. As discussed in section 5.3.2, some physical layers are better suited to handle large amounts of data than others. The bandwidth of the physical layer is reflected in the firmware updating measurements. Measurements were carried out on three very different communication mediums: UDP, which is used between the master controller and the host, CAN, currently the only communication link between the master controller and drive controller, and UART, an alternative asynchronous interface between the master and the drive controllers.

The measurement results in the Table 3 are average values across at least 20 successful transfers, where no errors occurred and no data frames were resent. In order to demonstrate the importance of the payload size, all available channels were additionally tested on the CAN bus maximum payload of 7 bytes. The data rate is also measured in ideal settings: no other traffic, except for the upgrading data is passing through the channels. It should be noted that the UDP transfers were only made between the host and the master controller – the data was not passed on to the drive controller, as was the case with CAN and UART. This means, that the data rate is higher, because the processing only takes place on the master controller.

Table 3. Communication medium data rate comparison.

Data packet payload	**Data rate (Bytes per second)**			
Bytes	***CAN***	***UART (Baud 115200)***	***UART (Baud 406800)***	***UDP***
7	667	669	634	1181
128	-	4880	6997	17897
256	-	5482	9072	22783
512	-	5810	10175	27645
1024	-	5863	9597	30814
2048	-	-	-	32729

As it currently stands, with the best configuration possible, assuming the maximum image size, the master controller update takes 16.6 seconds (512KB) and 25.2 seconds for the drive controller (256KB). When using CAN bus, uploading the same size image to the drive controller would consume about 6 and a half minutes.

There are two important parameters determining the speed of the transfer: the baud rate, at which the data is sent and the payload size – bigger the payload, the less overhead there is, up to a certain point. The bigger payload assumes bigger data buffer, so the trade-off between the transfer speed and memory consumption has to be made. With UART, the optimal packet size is 512 bytes, while with the UDP, it is either 512 or 1024 bytes.

In order to try speed up the transfer for low packet size protocols such as CAN, an additional volatile memory cache was proposed. This cache stores multiple data frames and when full, the contents of the cache are written into the flash in one go. In theory, this eliminates the call overhead for the each flash write routine. The measurements (shown in Figure 26) were carried out via the CAN bus, using eight different cache sizes and the control measurements with no cache applied. Each size step was measured at least fifteen times. As the results indicate, in practice, the speed up is nonexistent or insignificant and does not justify the complexity and the memory overhead it adds to the overall updating system. It should be noted, that the measurements were carried out independent to the rest of the system – meaning that the cache might have a bigger impact, when the main function of the controllers are active, e.g. when the car is driving. The measurements were also carried out when using UART and UDP, where the results were similar, demonstrating almost no impact on the overall performance. As a result of this, the cache size parameter in the INIT frame (Figure 23) is left as a reserved field for future use.

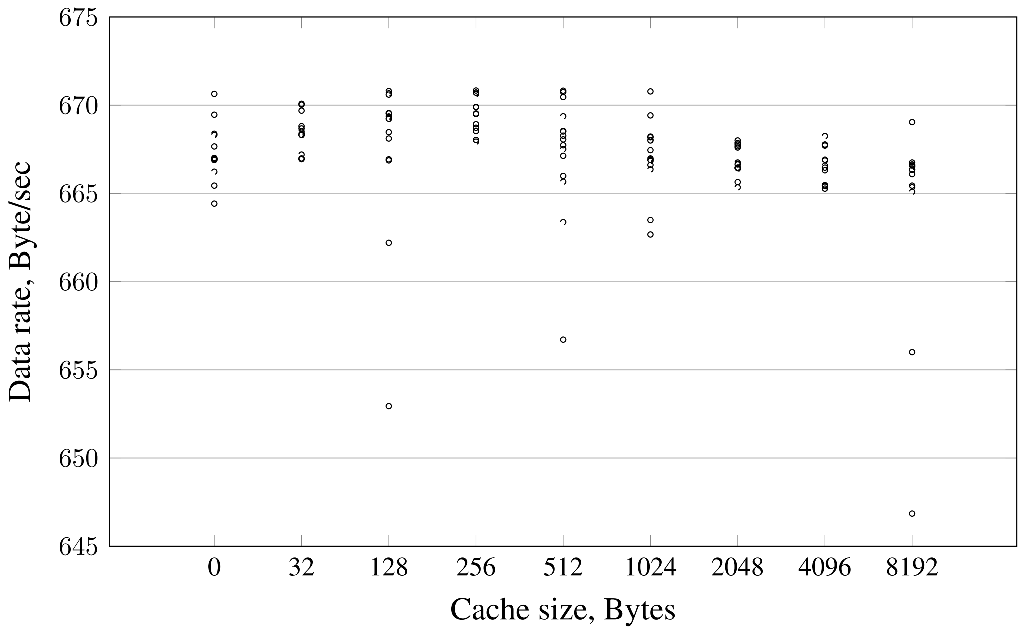

Figure 26. Data rate via CAN bus across different cache sizes.

Reliability is as important as performance when it comes to upgrading the system. The worst case would be running faulty code or crashing due to memory corruption. During the update, the device is in its most vulnerable state and it is important to identify various scenarios, which might lead to corrupt image transfer or worse, malfunctioning device. The devised test cases are presented in the Table 4.

Table 4. Test cases.

Description	**Expected result**	**Actual result**
Power up the device flashed with correct bootloader application firmware and backup	Start the application	PASS
Power up the device with missing or faulty application	Load the application from the backup, start the application	PASS
Power up the device with missing or faulty backup	Create a new backup from the application, start the application	PASS
Power up the device with correct new firmware available	Move the old application code to backup area, copy new firmware to application area, erase new firmware buffer area, start the new application	PASS
Power up the device with faulty new firmware available	Erase the new firmware area buffer, start the old application	PASS
Power up the device with only bootloader	Halt the program execution, flash error LED	PASS
Report wrong image size in INIT message	If the size is bigger than available memory space, do not start upgrading at all. If the size is smaller, and the end message is sent, the CRC check will fail. No END message results in timeout.	PASS
Disconnect ethernet cable during the data transfer	If the cable is connected before set timeout, the host retries to send the data and transfer will continue where it left off. If the timeout occurs, the update has to be started again from the beginning.	PASS
Disconnect the CAN/UART cable during the transfer to drive controller		PASS
Stop transfering data from the host	The transfer will end with timeout and the upload has to be started again.	PASS
Send data frame in wrong order	Receiver sends the previous ACK again with the packet sequence number it wants to receive next.	PASS
Hard reset during bootloader flashing operation execution	Bootloader will CRC check every image and resumes execution.	PASS

6 Conclusion

This thesis presents a solution for firmware updating the controllers for the universities self-driving car project. For distributing the new firmware, the goal was to create a reliable and physical layer agnostic transfer protocol. A requirement for fail-safe upgrading also meant hosting multiple versions at once, which lead to developing bootloaders for the controllers. The overall programming philosophy of the project was to create modular building blocks for updating the controllers, flexible enough to endure infrastructure changes in the future. The work also includes reviwing the current state of upgrading the firmware in automotive industry, where the biggest challenge today is delivering fast and reliable updates over the air.

As a result, various software components related to the upgrading of the firmware were created. First the bootloaders, which are responsible for verifying the controller's memory every time the car is started. Bootloaders also manage the available firmware versions so that the vehcile is guaranteed to recover from nearly all of the faulty memory states. The main effort went to implementing the transfer protocol mechanisms. In order to achive reliable and fault safe upgrading, the solution had to pass numerous tests, where edge cases and erronous situations were introduced. Testing was carried out on three different physical layers, showing that the developed protocol is portable and reliable on any medium. Performance wise, the solution can work faster than currently used JTAG, given the right physical layers. Additionally, post-build scripts to enhance the final image with necessary metadata and host side scripts to upload the binaries were developed. As a welcome byproduct of the thesis, the work can also act as a general guide to memory partitioning and bootloaders in embedded devices.

Future work entails getting the new firmware image onto the host computer remotely, perhaps via 5G or Wi-Fi. Additional security enhancements can be made via encryption and signatures to ensure data privacy and authenticity. Finally, as a development cycle improvement, the binary preparation and upload to the vehicle should be part of the continuous integration pipeline, going through automated testing and delivery process, in order to promote better quality of the developed firmware.

References

[1] J. Vangelov, “Software Updates in Automotive Electronic Control Units,” in *Software Update as a Mechanism for Resilience and Security: Proceedings of a Workshop*, Washington DC, U.S.A: The National Academies Press, 2017.

[2] J. Greenough, *THE CONNECTED CAR REPORT: Forecasts, competing technologies, and leading manufacturers*, Jan. 2016. [Online]. Available: `https://www.businessinsider.com/connected-car-forecasts-top-manufacturers-leading-car-makers-2015-3` (visited on 01/12/2019).

[3] Tallinn University of Technology, Silberauto AS, *Iseauto*. [Online]. Available: `http://iseauto.ttu.ee` (visited on 01/12/2019).

[4] A. M. Michael Held and J. Reaves, *The auto industry’s growing recall problem -— and how to fix it*, Jan. 2018. [Online]. Available: `https://emarketing.alixpartners.com/rs/emsimages/2018/pubs/EI/AP_Auto_Industry_Recall_Problem_Jan_2018.pdf` (visited on 01/12/2019).

[5] R. von Stokar, *Updating Car ECUs Over-The-Air (FOTA) White paper*, Sep. 2011. [Online]. Available: `http://www.eenewsautomotive.com/content/updating-car-ecus-over-air-fota` (visited on 01/12/2019).

[6] G. de Boer, P. Engel, and W. Praefcke, “Generic Remote Software Update for Vehicle ECUs Using a Telematics Device as a Gateway,” in *Advanced Microsystems for Automotive Applications*, Berlin, Heidelberg: Springer, 2005, pp. 371–380.

[7] D. K. Nilsson, P. H. Phung, and U. E. Larson, “Vehicle ECU classification based on safety-security characteristics,” in *IET Road Transport Information and Control*, Manchester, UK, May 2008.

[8] D. K. Nilsson, L. Sun, and T. Nakajima, “A Framework for Self-Verification of Firmware Updates over the Air in Vehicle ECUs,” in *2008 IEEE Globecom Workshops*, New Orleans, LA, USA, Nov. 2008.

[9] M. Steger, C. Boano, M. Karner, J. Hillebrand, W. Rom, and K. Römer, “SecUp: Secure and Efficient Wireless Software Updates for Vehicles,” in *2016 Euromicro Conference on Digital System Design (DSD)*, Limassol, Cyprus, Aug. 2016.

[10] G. Nelson, “Over-the-air updates on varied paths,” *Automotive News*, Jan. 2016. [Online]. Available: `http://www.autonews.com/article/20160125/OEM06/301259980/over-the-air-updates-on-varied-paths` (visited on 01/12/2019).

[11] *Tesla support - Software Updates*. [Online]. Available: `https://www.tesla.com/support/software-updates` (visited on 01/12/2019).

[12] D. Bogdan, R. Bogdan, and M. Popa, “Delta flashing of an ECU in the automotive industry,” in *2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, Timisoara, Romania, May 2016.

[13] M. Steger, M. Karner, J. Hillebrand, W. Rom, E. Armengaud, M. Hansson, C. A. Boano, and K. Römer, “Applicability of IEEE 802.11s for automotive wireless software updates,” in *2015 13th International Conference on Telecommunications (ConTEL)*, Graz, Austria, Jul. 2015.

[14] *Autosar*. [Online]. Available: `https://www.autosar.org/` (visited on 01/12/2019).

[15] D. Kum, G.-M. Park, S. Lee, and W. Jung, “AUTOSAR migration from existing automotive software,” in *2008 International Conference on Control, Automation and Systems*, Seoul, South Korea, Oct. 2008.

[16] A. Rassõlkin, R. Sell, and M. Leier, "Development case study of the first estonian self-driving car, ISEAUTO," *The Journal of Riga Technical University*, vol. 14, 1 Jul. 2018.

[17] R. Sell, M. Leier, A. Rassõlkin, and J.-P. Ernits, "Self-driving car ISEAUTO for research and education," in *Proceeding of the 2018 19th International Conference on Research and Education in Mechatronics*, Delft, The Netherlands, Jun. 2018, pp. 111–116.

[18] Open Source Robotics Foundation, *The Robot Operating System (ROS)*. [Online]. Available: http://www.ros.org/ (visited on 01/12/2019).

[19] STMicroelectronics, *STM32F76xxx and STM32F77xxx advanced Arm®-based 32-bit MCUs*, Reference manual. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/reference_manual/group0/96/8b/0d/ec/16/22/43/71/DM00224583/files/DM00224583.pdf/jcr:content/translations/en.DM00224583.pdf (visited on 01/12/2019).

[20] ——, *STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm®-based 32-bit MCU*, Reference manual. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf (visited on 01/12/2019).

[21] *System Workbench for STM32*. [Online]. Available: http://www.openstm32.org/HomePage (visited on 01/12/2019).

[22] *The Eclipse Foundation*. [Online]. Available: https://www.eclipse.org/ (visited on 01/12/2019).

[23] *ST-LINK/V2 in-circuit debugger/programmer for STM8 and STM32*. [Online]. Available: https://www.st.com/en/development-tools/st-link-v2.html (visited on 01/12/2019).

[24] *STM32Cube initialization code generator*. [Online]. Available: https://www.st.com/en/development-tools/stm32cubemx.html (visited on 01/12/2019).

[25] I. Skerrett, *Eclipse community survey 2014*. [Online]. Available: https://www.slideshare.net/IanSkerrett/eclipse-community-survey-2014 (visited on 01/12/2019).

[26] S. Chacon and B. Straub, *Pro Git*, Second Edition. Apress, 2014. [Online]. Available: http://link.springer.com/10.1007/978-1-4842-0076-6.

[27] STMicroelectronics, *AN4826 Application note: STM32F7 Series Flash memory dual bank mode*. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/application_note/group0/d2/bd/77/e8/0d/3a/43/cf/DM00266999/files/DM00266999.pdf/jcr:content/translations/en.DM00266999.pdf (visited on 01/12/2019).

[28] ARM, *ARM Cortex-M4 Devices Generic User Guide*. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/DUI0553A_cortex_m4_dgug.pdf (visited on 01/12/2019).

[29] ——, *ARM Cortex-M7 Devices Generic User Guide*. [Online]. Available: https://static.docs.arm.com/dui0646/a/DUI0646A_cortex_m7_dgug.pdf (visited on 01/12/2019).

[30] FreeRTOS Project, *The FreeRTOS™ Kernel*. [Online]. Available: https://www.freertos.org/ (visited on 01/12/2019).

[31] Atmel Corporation, *Safe and Secure Bootloader Implementation for SAM3/4*, Application note. [Online]. Available: http://ww1.microchip.com/downloads/en/AppNotes/Atmel-42141-SAM-AT02333-Safe-and-Secure-Bootloader-Implementation-for-SAM3-4_Application-Note.pdf (visited on 01/12/2019).

[32] Texas Instruments, *Introduction to the Controller Area Network*, Application report. [Online]. Available: http://www.ti.com/lit/an/sloa101b/sloa101b.pdf (visited on 01/12/2019).

[33] P. Trink, "Autonomous path following on a vehicle using an open source software autoware," sup. Ernits, Juhan-Peep, Master thesis, Tallinn University of Technology, 2018. [Online]. Available: https://digi.lib.ttu.ee/i/file.php?DLID=10630&t=1 (visited on 01/12/2019).

Appendix 1 – Flash module organizations

Table 1. Flash module organization of STM32F407 [28].

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	...	...	...
	Sector 11	0x080E 0000 - 0x080F FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

Table 2. Flash module organization of STM32F767 [29].

Block	**Name**	**Block base address on AXIM interface**	**Sector**
Main memory block	Sector 0	0x0800 0000 - 0x0800 7FFF	32 KB
	Sector 1	0x0800 8000 - 0x0800 FFFF	32 KB
	Sector 2	0x0801 0000 - 0x0801 7FFF	32 KB
	Sector 3	0x0801 8000 - 0x0801 FFFF	32 KB
	Sector 4	0x0802 0000 - 0x0803 FFFF	128 KB
	Sector 5	0x0804 0000 - 0x0807 FFFF	256 KB
	Sector 6	0x0808 0000 - 0x080B FFFF	256 KB
	...	...	...
	Sector 11	0x081C 0000 - 0x081F FFFF	256 KB
Information block	System memory	0x1FF0 0000 - 0x1FF0 EDBF	60 Kbytes
	OTP	0x1FF0 F000 - 0x1FF0 F41F	1024 bytes
	Option bytes	0x1FFF 0000 - 0x1FFF 001F	32 bytes

Appendix 2 – Source code repositories

Source code is controlled with git and the repositories are available via following URLs:

Master controller: `git@gitlab.pld.ttu.ee:iseauto/master_controller.git`

Drive controller: `git@gitlab.pld.ttu.ee:iseauto/drive_controller.git`

Bootloaders: `git@gitlab.pld.ttu.ee:iseauto/bootloader.git`