Bachelor’s/Master's thesis

Information and communication technology

2018

Kimmo Korpelin

BOOT LOADER AND FIRMWARE UPDATE PROTOCOL FOR EMBEDDED DEVICES

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and communication technology

May 2018 | 43 pages, 1 appendix page

KIMMO KORPELIN

BOOT LOADER AND FIRMWARE UPDATE PROTOCOL FOR EMBEDDED DEVICES

This thesis details the design and implementation of a firmware update process developed for an embedded device network. A boot loader program was developed for a group of peripheral devices used in marine vessels. These peripherals were connected to a master device through a CAN bus network. This master device, a Linux-based tablet PC, displayed an interface for and issued commands to the peripheral devices. The boot loader was developed as necessary to enable firmware updating on the peripheral devices. The thesis goes over the boot loader embedded platform and design decisions made on the boot loader program. The CAN bus protocol for communication between the master and boot loader is defined. The functionality of the updater program developed for the tablet PC is described. As a result, the complete update process of all three systems working together is demonstrated. Finally, the update process was released as a feature in the group of products it was developed for.

KEYWORDS:

Boot loader, Embedded systems, firmware update

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka

Toukokuu 2018 | 43 sivua, 1 liitesivu

Kimmo Korpelin

SULAUTETTUJEN ALUSTOJEN KÄYNNISTYKSENLATAAJA LAITEOHJELMISTON PÄIVITYKSELLÄ

Opinnäytetyö käsittelee päivitysjärjestelmän kehitystä pienlaiteverkolle. Laiteverkolle kehitetty käynnistyksenlataaja tekee mahdolliseksi laiteohjelmiston päivityksen päätelaitteissa, jotka ovat kytkettynä hallintalaitteeseen CAN-sarjaverkon yli. Laiteverkossa on kytkettynä veneily- ja pienmeriliikenteen kulkuneuvojen laitteita. Hallintalaite on Linux-pohjainen tablettitietokone, jolla verkkoon kytkettyjä laitteita ohjataan ja niiden tilaa seurataan. Käynnistyksenlataaja mahdollistaa näiden laitteiden ohjelmistopäivityksen sarjaverkon yli. Tämän käynnistyksenlataajan kehitystä ja suunnittelupäätöksiä päätelaitteiden laitteistoalustan rajoitteissa havainnollistetaan. CAN-sarjaverkolle kehitetyn protokollan viestintä määritellään ja sen käyttöä päivitysprosessissa visualisoidaan. Hallintalaitteelle kehitetyn päivitysohjelman ominaisuudet ja toiminta käydään läpi. Lopulta näiden kolmen osan toimintaa yhtenäisenä päivitysjärjestelmänä demonstroidaan. Lopputuloksena päivitysjärjestelmä otettiin käyttöön toimeksiantajan tuotteissa.

ASIASANAT:

Sulautetut järjestelmät, ohjelmistopäivitys, käynnistyksenlataaja

CONTENT

CONTENT 4

FIGURES 6

TABLES 6

1 INTRODUCTION 8

2 THE DEVICE NETWORK 9

2.1 Overview 9

2.2 Boot loader hardware platform 9

3 CAN BUS 11

3.1 Overview 11

3.2 Bus configuration 11

3.3 Data transmission 12

4 SYSTEM REQUIREMENTS 14

4.1 Peripheral device requirements 14

4.2 Updater program requirements 15

4.3 CAN update protocol requirements 15

5 THE BOOT LOADER PROGRAM 16

5.1 Program flow overview 17

5.2 Transition to boot loader code 19

5.3 Permitting critical commands 20

5.4 User application header 20

5.5 Platform identification 21

5.6 Running the user application 22

5.7 Modifying the user application 24

6 THE UPDATER PROGRAM 26

6.1 Polling node type and version 26

6.2 Updating a node 27

6.3 Empty boot loader nodes 30

7 UPDATE PROTOCOL **31**
7.1 Message type 31
7.2 Message addressing 32
7.3 Message validation 32
7.4 Device type identification 33
7.5 Common messages 33
7.5.1 ELEVATE 33
7.5.2 BL_AVAILABLE 34
7.5.3 REQUEST 34
7.5.4 RESPONSE 35
7.5.5 PRODUCT_ID 35
7.5.6 SW_VERSION 36
7.5.7 HW_ID 36
7.6 Boot loader messages 37
7.6.1 RELEASE 37
7.6.2 APP_VERIFY 37
7.6.3 APP_BOOT 38
7.6.4 APP_ERASE 39
7.6.5 HEADER_ADDRESS 39
7.6.6 APP_SPACE 39
7.6.7 APP_PRODUCT_ID 40
7.6.8 CACHE_CLEAR 40
7.6.9 CACHE_WRITE 40
7.6.10 CACHE_CALC_CRC 41
7.6.11 CACHE_COMMIT 41
7.6.12 CACHE_COMMIT_ALL 41
7.6.13 FLASH_PAGE_ERASE 42
7.6.14 FLASH_CALC_CRC 42

7 CLOSING CHAPTER **43**

APPENDIX 1. BOOT LOADER OPTION REGISTER **1**

FIGURES

Figure 1. Device network example. 9
Figure 2. Presentation of an extended DATA frame. 12
Figure 3. Flash memory locations of the boot loader and user application. 16
Figure 4. Shared memory access to option register between the boot loader and user application. 17
Figure 5. Program flow of the boot loader. 18
Figure 6. Program flow of a user application transitioning to boot loader code. 19
Figure 7. The required steps for the transfer from boot loader to user application code. 24
Figure 8. Illustration of CPU time divided between CAN processing and cache writing. 25
Figure 9. Example of a bus scan procedure with 2 bus peripherals. 27
Figure 10. Elevation and boot loader code transition of a node. 28
Figure 11. Node Elevation when running boot loader code. 28
Figure 12. Writing a new user application binary. 29
Figure 13. Peripheral node transitioning to user application code. 30
Figure 14. Writing a user application to an empty boot loader device. 30
Figure 15. Example of a Message ID for peripheral Command #3. 31
Figure 16 Location of Message ID, Target and Source values in the identifier bits. 32

TABLES

Table 1. Computational resources of the microcontroller. 10
Table 2. The parameter fields contained in the user application header. 21

1 INTRODUCTION

This thesis details the design and development of a firmware update scheme for a network of embedded peripherals connected along with a master device. The thesis describes the design of a boot loader for the peripheral devices, an updater program for the master device and an update protocol for communication between the master and the peripheral devices. Basic understanding of software development and embedded concepts on the part of the reader is assumed.

The topic of this thesis was requested by my employer. The request was to implement firmware updating over CAN bus which was used for communicating configurations and commands to peripheral devices. Due to their hardware platform, this required the development of a boot loader for the peripheral devices. In addition an updater program which serves the firmware to the peripherals had to be developed, and the CAN bus protocol had to be expanded no enable the update process.

My employer Nextfour Group Oy specializes in developing embedded devices and their software as part of customer contracts. They also have their own line of products, for one of which this firmware updating system was developed for. These products are computer systems and devices for marine vessel manufacturers for interfacing with various marine vessel utilities and peripherals, such as navigation, sonar and marine vessel electronical device control.

The overall design and development work was done by me with oversight and input from my colleagues. This involved the development of the boot loader program, the updater program, the design of the protocol, production, validation and testing tools and integration with existing systems. The main body of work for these was done full-time from January 2018 to early April 2018.

2 THE DEVICE NETWORK

2.1 Overview

The system consists of a single master device, a Linux tablet PC, connected to peripheral devices via CAN bus. This tablet PC is used to control and display an interface for various peripherals in a marine vessel. Examples of peripherals connected to the bus are sonar, boat heater controller and boat electronics controller. The exact peripherals and interfaces depend on the vessel specific configuration. This thesis does not detail the design and nature of these peripheral devices and their applications. Peripheral firmware is from now on referred as “user application”.

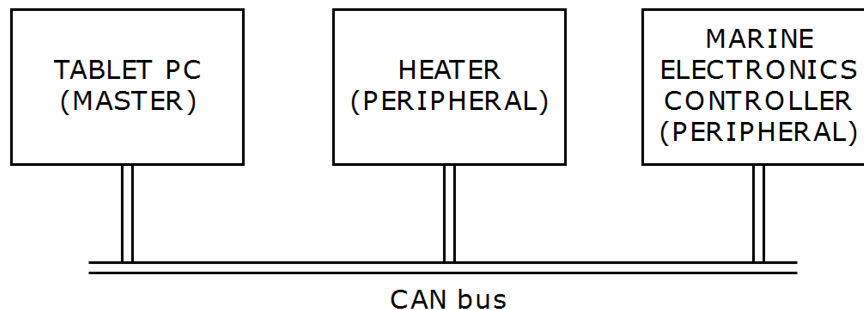

Figure 1. Device network example.

As mentioned, the tablet PC issues commands and configurations to the peripheral devices. It also receives regular over-the-air software updates by a mobile internet connection. With the peripheral boot loader, the firmware of the peripheral devices may also be updated on demand. This facilitates updateability of the whole system.

2.2 Boot loader hardware platform

It was decided that the update feature was first to be implemented in the most commonly used hardware platform in the peripheral devices. These peripheral devices were based on STMicroelectronics STM32F091xC-microcontroller. The nature of this microcontroller made the development of a boot loader for it necessary. The microcontroller’s limited

computational resources, such as low available flash memory for applications, also introduced tight resource constraints for the boot loader.

Table 1. Computational resources of the microcontroller (STM32F091xC, 2017, p. 11).

CPU	ARM Cortex-M0
Max. CPU clock	48 MHz
SRAM	32 KB
Flash memory	128 or 256 KB

The more affordable 128 KB Flash version was exclusively used in the peripheral devices. As the application binaries tended to use up to 100 KB of flash memory, the practical maximum size for the boot loader firmware was set at 16 KB. This also indicates that the microcontroller can hold only one version of the application binary in memory at a time.[1]

The updateability has been developed for this platform. However, the update protocol is not limited for this platform only, and more hardware platforms may be added in the future.

[1] This creates a limitation for the boot loader: if an update process is ended prematurely, only the boot loader program will be left in Flash memory. To mitigate this, the boot loader will always verify the existence and integrity of the user application and the updater program has to actively seek and update empty boot loader devices.

3 CAN BUS

This chapter contains an overview of the CAN bus which is used for the communication between the boot loader and the Master device.

3.1 Overview

Common Area Network (CAN) is a serial bus widely used in embedded devices. Its origins are in the automotive industry where it is used to facilitate communication between embedded control systems in vehicles (Natale, 2008, p. 5). Devices connected to the CAN bus are called nodes. Each of these nodes can transmit and receive data, with the bus featuring a deterministic collision resolution (Natale, 2008, p. 20). Transmitted data is multi-casted to every node in the network and it is up to the nodes to filter the incoming data.

The physical layer consists of a two-wire differential bus. The signal is digital with Non Return to Zero bit encoding. The two bit states are referred as “dominant” (typically 0) and “recessive”. This is used in the collision resolution when two transmitters start sending at approximately the same time: the first to send a recessive bit over another’s dominant will yield the effort to the dominant sender.

As there is no separate clock signal, bit stuffing is used to help keeping the nodes synchronized. This is done by the transmitting controller by inserting a complemented bit to the stream if five consecutive bits of same state are sent.

Most of the CAN protocol management, such as frame priority arbitration, error detection and retransmission handling, is implemented by the CAN controller peripheral included in the microcontroller (RM0091, 2015, p. 836).

3.2 Bus configuration

The boot loader protocol is developed exclusively for the CAN 2.0b standard. CAN 2.0b is used over CAN 2.0a as the former allows for an extended CAN frame format. Fixed bus speed of 250 kilobits/s is used, as it had already been chosen for the network.

Generally, 120 Ohm resistors are used to terminate the signal wires on each end of the network.

3.3 Data transmission

In the CAN bus, data is transferred in units called frames. There are four types of frames: DATA, REMOTE, ERROR and OVERLOAD frames (Natale, 2008, p. 17). ERROR and OVERLOAD are used internally for error handling and flow control by the CAN controller. The boot loader only uses DATA frames explicitly.

Each DATA frame may transfer up to a maximum of 8 data bytes (64 bits), indicated by the Data Length Content (DLC) value. In addition to the data bytes, the extended identifier contains 29 bits for use. Each protocol message utilizes these data slots according to its specification. Incorrect and out-of-specification utilization of these will render a message invalid and cause the DATA frame to be discarded.

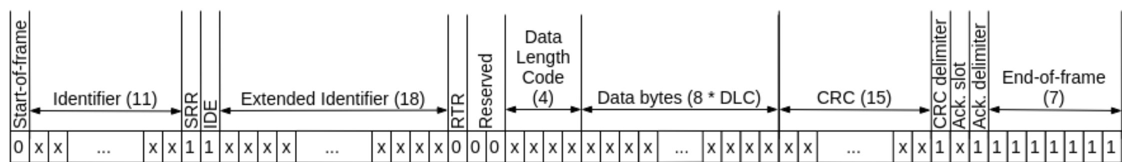

Figure 2. Presentation of an extended DATA frame.

As per how the collision resolution works, recessive sender will yield to dominant senders. This means that a DATA frame with lower identifier number will win over a higher number identifier frame.

Efficient utilization of frame data slots is required to minimize the bus traffic, especially with high-frequency messages. As firmware binaries are transferred over the bus, relatively high amount of data is transferred during a firmware update. With 8 data bytes per DATA frame, the overhead per frame is half of the frame (64 bits out of total 128 bits). With 250 kilobits/s data rate, at best, this results in around 15 kilobytes per second transmit rate. In the real implementation, a 100 kilobyte binary update is completed in around 9.5 seconds compared to the theoretical 6.66 seconds. Additionally, any additional bus traffic that takes priority over the boot loader messages will slow this down.

How DATA frames are utilized by the protocol messages is further detailed in chapter 7.

4 SYSTEM REQUIREMENTS

The firmware update system can be divided into three separate parts: boot loader program, updater program and the CAN update protocol. The CAN protocol acts as an interface between the master device and the peripherals. The updater program is run in the master device. The boot loader program is run on the peripheral devices alongside their own user applications.

This chapter goes over the requirements set on these three parts of the system. Implementation and design of these parts is detailed in chapters 5, 3 and 6.

4.1 Peripheral device requirements

Boot loader program was given the following requirements:

- Running the user application on start up
- Determining device product type
- Determining user application version
- Authentication to access the firmware modifying operations
- Peripheral user application (firmware) modification:
    - Erasing the old user application
    - Writing a new user application
    - Verifying integrity of the new user application
- Transitioning to the user application code
- Operation status and error reporting

The user application had the following additional requirements:

- Determining device product type
- Determining user application version
- Transitioning to the boot loader code

As some of these features were not present in the user applications, additions fulfilling these requirements had to be developed for each user application.

4.2 Updater program requirements

The updater program has the following requirements:

- Search for binaries of peripheral devices from the local storage
- Scan the CAN bus for peripheral devices
- Update peripheral devices if a newer binary is available
- Program empty devices with their latest binary

Other features, such as over-the-air delivery of new binaries, are allocated to other programs on the master device.

4.3 CAN update protocol requirements

A protocol for sending configurations over the CAN bus had already been developed for the peripheral devices. It was decided that the update protocol had to keep compatibility with this existing protocol. This protocol consisted of predefined messages transmitted and received over the CAN bus by the master and the peripheral devices. The existing protocol was expanded upon with update specific commands in order to implement peripheral firmware updating.

The following were set up as requirements for the update protocol:

- Backwards compatibility with the configuration protocol
- Message specifications for the following:
  - Gaining authentication to update messages
  - Messages for sending the firmware binary
  - Messages for verifying the firmware binary
  - Messages for transitioning to user application
- Translation of the messages into CAN frames and vice versa
- Device update within a reasonable time (10s of seconds max.)

5 THE BOOT LOADER PROGRAM

On small scale embedded devices, such as cheap microprocessors and SoCs, the main application interfaces directly with the hardware and operates without a separate operating system. One of the ways to perform a software update in this environment is by adding a boot loader into the device. The boot loader is the initial program run on the microprocessor. It decides whether to run the managed user application or if signaled by some external trigger to update the user application from an external source. This external trigger may be a specific pin state, message to a communication bus, value in memory, etc. The external source may be a file transmitted over-the-air, a serial connection or some another bus. The boot loader essentially acts as a limited file system manager, managing the user application.

The boot loader developed for our system is an embedded program for the STM32F091xC microcontroller. It is designed to be programmed into the microcontroller alongside the managed user application. It can completely remove and rewrite the contained user application, and also stay operating without a user application.

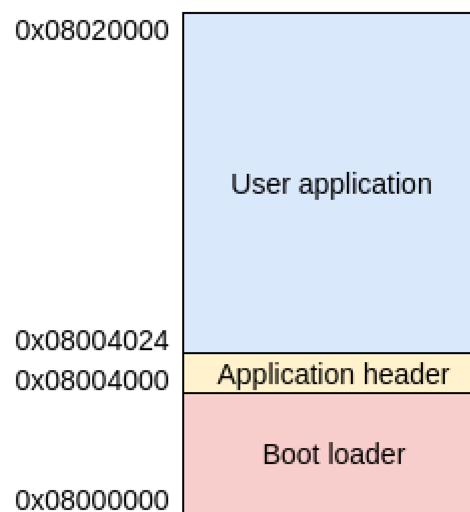

Figure 3. Flash memory locations of the boot loader and user application.

The program is mainly written in C++11, with minor portions in C and assembly. It is cross-compiled for the microcontroller using the GNU Embedded Toolchain for Arm compiler (ARM Ltd, 2018) on a modern Linux-desktop development environment. The

build and development of the boot loader was integrated with the build process of the peripheral devices it was developed for. The boot loader and the peripheral program were programmed and debugged on the target device using ST-LINK/V2 in-circuit programmer (ST-LINK/V2, 2016).

Every peripheral program compiles its own version of the boot loader with platform specific hardware configurations and compiled-in platform identification values. As the boot loader may be left empty without a user application, these identification values are used to decipher the type of firmware that is programmable to the device. More on the identification values in chapter 5.5.

5.1 Program flow overview

The boot loader runs as the initial program after a System reset[2]. First action taken upon start up is to check if any boot loader option register values have been set. These option registers are located in a predefined location in SRAM, accessible by both the boot loader and the user application. Setting the option register values may trigger alternative actions at launch, such as if running the user application should be initially skipped. Presuming that any alternative actions are not configured and the application verification passes, the user application is then run immediately. When the boot loader option registers are unset, this is the default program path taken on a System reset.

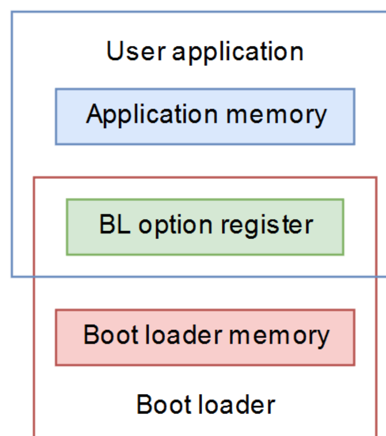

Figure 4. Shared memory access to option register between the boot loader and user application.

[2] System reset sets all registers to their reset values. For details, see (RM0091, 2015, p. 99).

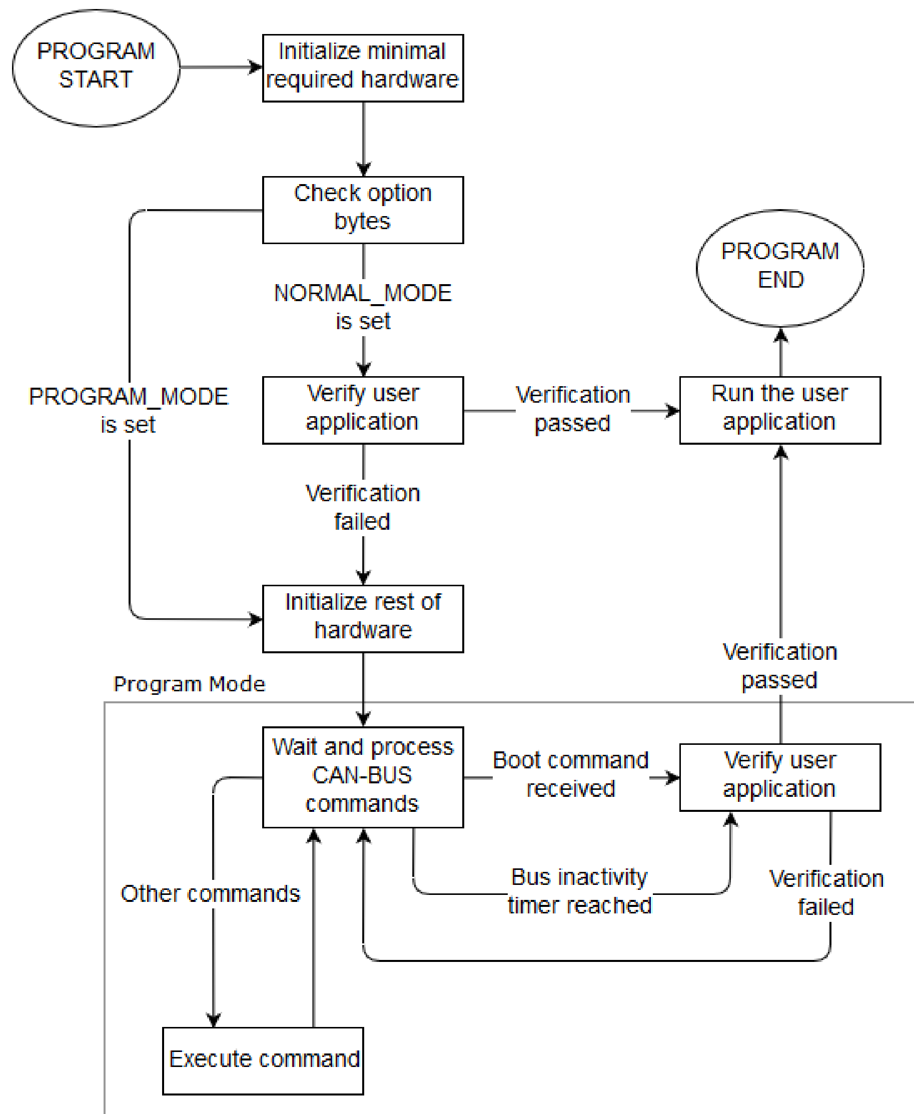

Figure 5. Program flow of the boot loader.

In case the user application is not booted, Program Mode is entered. The boot loader initializes the required hardware to listen on the CAN bus. Once in Program Mode, the boot loader waits for and executes commands received over the CAN bus. A timer monitors the time of inactivity since last command. Every received valid command refreshes the timer to 0. If the timer reaches a high enough value (by default 10 seconds), the user application will be resumed if it is still intact. This is done in case the master device was disconnected after the transfer to boot loader code, yet no changes to user application were completed.

5.2 Transition to boot loader code

To modify the user application when the user application is already running, a transition back to boot loader code first is required. The boot loader option BOOT_MODE has to be set to the value PROGRAM_MODE to have the boot loader skip the application verification and enter Program Mode (see appendix A for boot loader option registers). After setting the option, the transition to boot loader code may be done by performing a software reset. If BOOT_MODE is left to its default value, the boot loader will default to verifying the application first and thus transfer right back to application code.

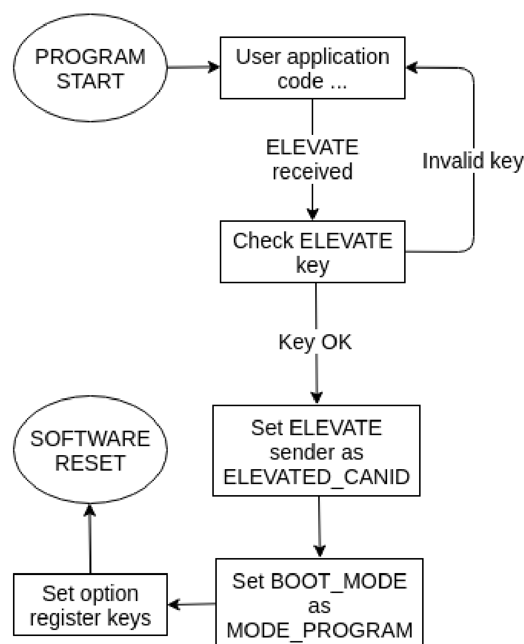

Figure 6. Program flow of a user application transitioning to boot loader code.

As the ability to transition into boot loader code by a CAN bus message has to be available, some additional protocol handling code is required from the user application. The ELEVATE-command, in addition to its other function[3], is used to request the transition into boot loader code. By setting the ELEVATED_CANID, the node which triggered the transition is marked as Elevated upon boot loader start-up. Fortunately, the required additions are relatively small and applicable as is for most of the peripheral device firmware (see Figure 2.). However, if a user application does not comply with

[3] ELEVATE is also used to grant full permissions to critical commands (see chapters 3.3 and 5.5.2 for details).

these requirements, the updateability of the device over the CAN bus will be broken. This can only be mitigated with testing the compliance with this protocol with each release.

5.3 Permitting critical commands

In Program Mode, certain boot loader commands are not available by default. Permission to execute critical commands, such as those that modify the user application, are granted only to nodes[4] that have provided a correctly generated key sent with the ELEVATE-command. This key is generated from the boot loader's unique device ID (UID), which is fetched from the boot loader. Nodes that are granted this permission are called Elevated and have full access to the boot loader's features. If an Elevated node already exists, new node Elevation is not available until the last Elevation has been released or after no communication has been received from the last Elevated node for a certain time period (by default, 5 seconds).

This protection scheme is not very secure as simply observing the bus traffic and re-sending the key frame at a later time is enough to gain access to these commands. The process is mainly in place to reduce the chance of incompatible or unrelated messages modifying the contained user application. The Elevation scheme also ensures that only one device controls the update functions of the boot loader at a time.

5.4 User application header

Every valid user application has to define an application header. This header contains information about the user application, such as application version, hardware platform type and unique product ID the application is for. The user application header is located right at the start of the application binary.

The header is for the most part generated by the compiler from a C-struct placed at the start of the Flash memory. The fields APP_SIZE and APP_CHECKSUM have to be calculated from the compiled binary and modified into the binary post-build. This procedure can be automatized with a relatively simple tool script.

---

[4] For introduction to CAN nodes see chapter 4.1.

Table 2. The parameter fields contained in the user application header.

Field	Description
PLATFORM	Hardware platform enumeration value.
HEADER_VERSION	Version number of this user application header.
HEADER_ADDRESS	Location of this header. Used to check the absolute address the application was built for.
APP_OFFSET	Application offset relative to HEADER_ADDRESS.
APP_SIZE	Size of the application (excluding the header).
APP_CHECKSUM	CRC value for the application (excluding the header).
VERSION_MAJOR	Most significant number of the application version.
VERSION_MINOR	Middle number of the application version.
VERSION_BUILD	Least significant number of the application version.
PRODUCT_TYPE	Enumeration value indicating type of the device.
PRODUCT_MANUFACTURER	Enumeration value indicating the manufacturer of the device.
PRODUCT_MODEL	Model number of the device. Indicates hardware revision.

5.5 Platform identification

The boot loader has to be identifiable as a boot loader for a specific peripheral device. For this reason, every boot loader program is compiled with matching user application identification values. If these don’t match the user application header’s values, the user application is rejected and not run.

The identification values for the user application are found in the header. The following identification values are compiled in the boot loader program:

- PLATFORM
- HEADER_ADDRESS
- PRODUCT_TYPE
- PRODUCT_MANUFACTURER
- PRODUCT_MODEL

5.6 Running the user application

To verify the user application contained in Flash memory, the following four checks are performed on the application binary:

1. **Application checksum value comparison**. The header field APP_CHECKSUM must match the application binary. The checksum value for the user application is calculated[5] from the area indicated by fields APP_OFFSET and APP_SIZE. If the calculated checksum matches the one defined in the application header, the application is interpreted as intact.

2. **Hardware platform check**. The header field PLATFORM must match with the compiled-in value of the boot loader. This is done to verify that the application binary is intended for the current hardware platform.

3. **Build address check**: The header field HEADER_ADDRESS must match with the compiled-in value of the boot loader. If this differs, all position-dependent code[6] in the application would fail and cause undefined behavior.

4. **Product ID check**: The fields PRODUCT_TYPE, PRODUCT_MANUFACTURER and PRODUCT_MODEL are compared against compiled-in values. Product ID check is done to further verify that the user application was compiled for this exact device type and model.

---

[5] The checksum algorithm used is one defined by ITU-T V.42 (ITU-T, 2002).
[6] For example, vector table contains are position-dependent addresses for interrupt routines.

If the application verification passes, the boot loader transitions to the application code. Due to platform's restrictions, straightforward jump to the entry point of the application is not sufficient. The prerequisite procedures to make this transfer will be detailed in this chapter.

In STM32F0xx-devices, interrupt vector addresses are fetched from a continuous array of addresses called vector table. Interrupt vectors are designated handlers for various asynchronous events, such as hardware peripheral interrupts. The vector table is located in boot memory space, which starts from address 0x0000 0000 (RM0091, 2015, p. 217). Boot memory space can be aliased to either Flash memory, SRAM or to System memory[7] (RM0091, 2015, p. 52). By default, the boot memory is aliased to Flash memory and as the boot loader occupies the start of the Flash memory it's vector table is accessed. On STM32F091xC the boot memory can only be mapped to the start of the Flash memory[8]. As each user application implements its own vector table, the boot memory has to be set to point to the application's vector table before it can be run. To achieve this, an array the length of the vector table is reserved from the start of SRAM in both the boot loader and the user application. The boot loader copies user application's vector table to SRAM and configures boot memory space to use SRAM by changing the value of MEM_MODE in the register SYSCFG_CFGR1 (RM0091, 2015, p. 176).

---

[7] System memory is read-only and it contains the USART In-System-Programmer (STM32F091xC, 2017, p. 13).
[8] Flash memory is mapped to start at 0x0800 0000 (RM0091, 2015, p. 55).

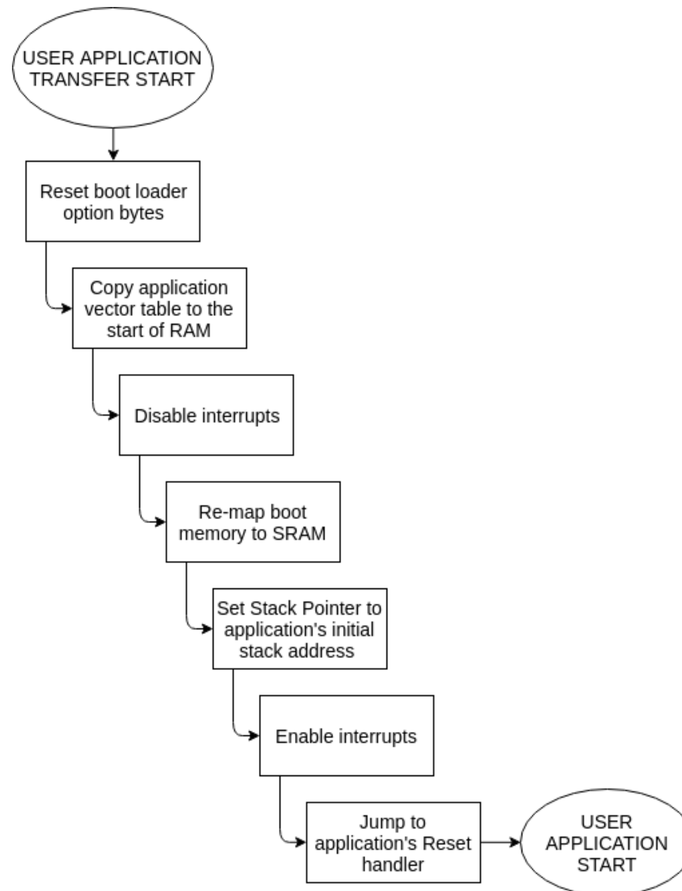

Figure 7. The required steps for the transfer from boot loader to user application code.

The stack pointer is set to the initial stack address[9] of the user application. Finally, by calling the user application's Reset-handler[10], the user application code is entered. On the next System reset, boot memory is aliased back to Flash memory and the boot loader program is run as the initial program again.

5.7 Modifying the user application

Modification of the user application binary involves erasing and writing of Flash memory. Commands that modify the Flash memory are granted only to Elevated nodes. Internally, any operations on Flash memory are limited to the address area of the user application (see Figure 2) to prevent any accidental modification of the boot loader code itself.

---

[8] The initial stack address value is located at 0x0000 0000.
[10] The address for the Reset handler is located at 0x0000 0004.

On STM32F091xC, 128 kilobytes of Flash memory is available, from which 112 kilobytes is available for the user application. For the boot loader protocol, the user application is addressed virtually starting from address 0x0000 0000 and ending at 0x0001 C000. The boot loader internally handles the translation of flash operation addresses to the real user application address space.

Before any Flash write operations can be done, the target area has to be erased first. The erase operation is done in units of Flash pages. Each page is 2048 bytes, with 64 pages in total (RM0091, 2015, p. 54). After erasure, the area of the page may be written to. The Flash write granularity is 16 bits (RM0091, 2015, p. 58).

For optimizing the update speed, Flash writes are first gathered into a cache in SRAM. This cache is then committed to Flash memory in the free time between receiving and processing CAN frames. The cache is guaranteed to be at least the size of a single Flash page (2048 bytes on STM32F091xC). This allows for a simple updater program loop of page erasure, cache writing and a cache commit per page. While only one cache is provided for editing at a time, multiple caches can be queued for writing in the background. If all of the caches are busy being committed, the boot loader will block the update protocol until at least one cache becomes available.

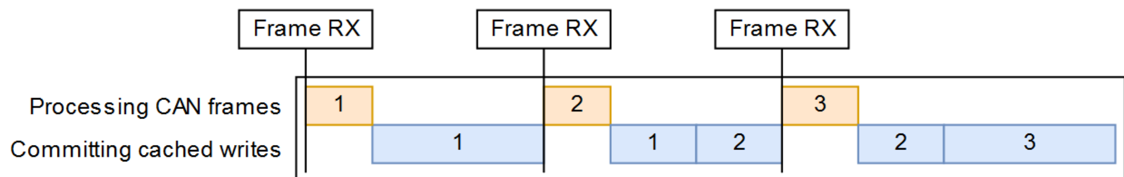

Figure 8. Illustration of CPU time divided between CAN processing and cache writing.

During the update process, some writes may be lost due to a bad bus connection or some other failure. For checking that all writes were successful, the boot loader can be asked to verify the user application. If this fails, the boot loader can be polled for CRC checksums over areas of Flash memory. By checking the Flash memory piece by piece, areas that were not written correctly can be re-written. The exact implementation of this algorithm is up to the updater program on the master device.

6 THE UPDATER PROGRAM

The updater is a program for the tablet PC in the system (for the system overview, see chapter 2.1). It is cross-compiled for the ARM-platform of the Linux tablet PC and runs as a systemd-managed service. Its main objective is to observe the peripheral devices connected to the CAN bus and update them if necessary. On the main storage of the tablet PC it manages a library of firmware binary entries. Each entry contains the version and peripheral device information, which are used to determine the correct target peripheral device. New firmware binaries are downloaded through a mobile internet connection when necessary.

When acting as a CAN bus node, the updater program is referred as the Master node.

This chapter will demonstrate on how the updater program uses the CAN bus protocol to update a peripheral device on the bus. The definitions for the protocol commands referenced in this chapter are found in chapter 6 (protocol commands are in all-caps).

6.1 Polling node type and version

The updater program should keep the devices on the bus updated. Frequent scanning for and polling of info from the nodes on the bus is required achieve this. The frequency and timing of these scans may be configured according to specific requirements of the system.

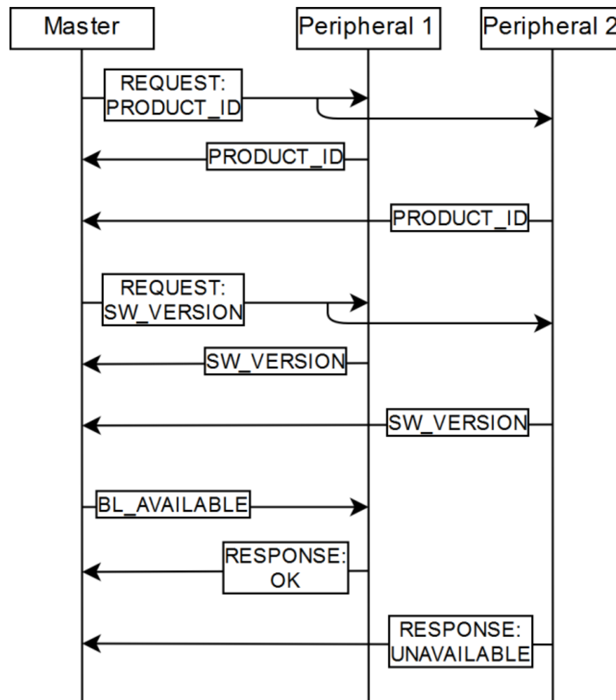

Figure 9. Example of a bus scan procedure with 2 bus peripherals.

Requests for PRODUCT_ID and SW_VERSION are broadcasted. Availability of the boot loader is polled with the command BL_AVAILABLE. If the target node has and can transition to boot loader code, and a binary with matching PRODUCT_ID and newer version is found from the firmware library, an update job may be initiated on the node.

6.2 Updating a node

At the start of an update job, the target node is elevated to gain the access to firmware modification commands. An Elevation key is generated from the unique device ID, which is fetched from the device by requesting HW_ID. This Elevation key is then bundled with the ELEVATE-command sent to the device. Presuming that the key is correct, the sending node is Elevated and Program Mode is entered.

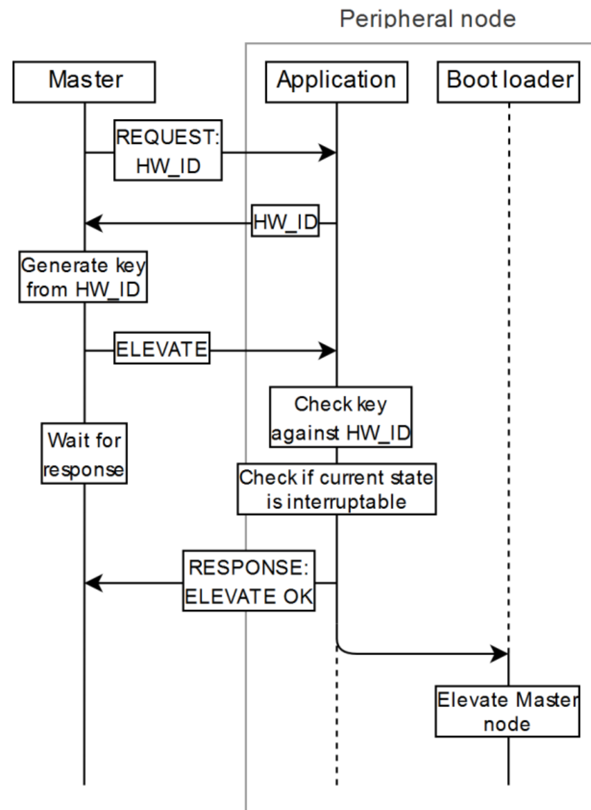

Figure 10. Elevation and boot loader code transition of a node.

From the Master's point of view, the protocol for Elevation is the same whether the target device is in boot loader code or in user application code; see figures 10 and 11.

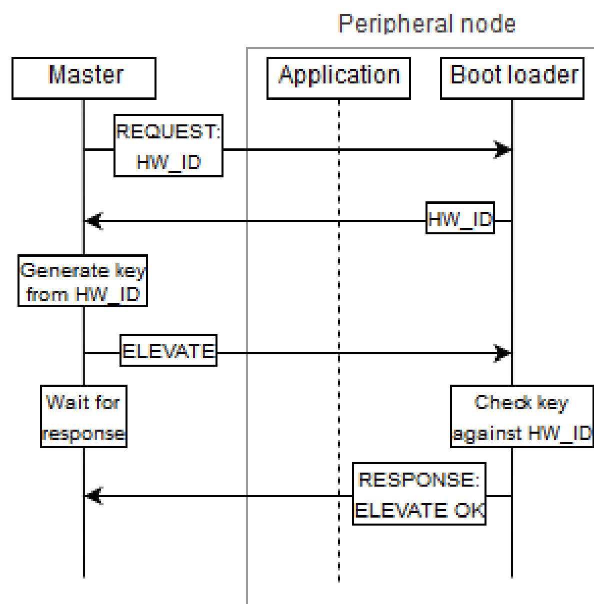

Figure 11. Node Elevation when running boot loader code.

After a successful elevation, the new user application binary is programmed into the peripheral. Erasing and writing the binary page by page is the simplest implementation for this. Each written page is checked is by comparing the CRC checksums calculated with CACHE_CALC_CRC against the updater's local binary. If the whole page check fails, checksums are calculated over smaller slices of the page. Any erroneus slices are then re-transmitted and the verification is attempted again.

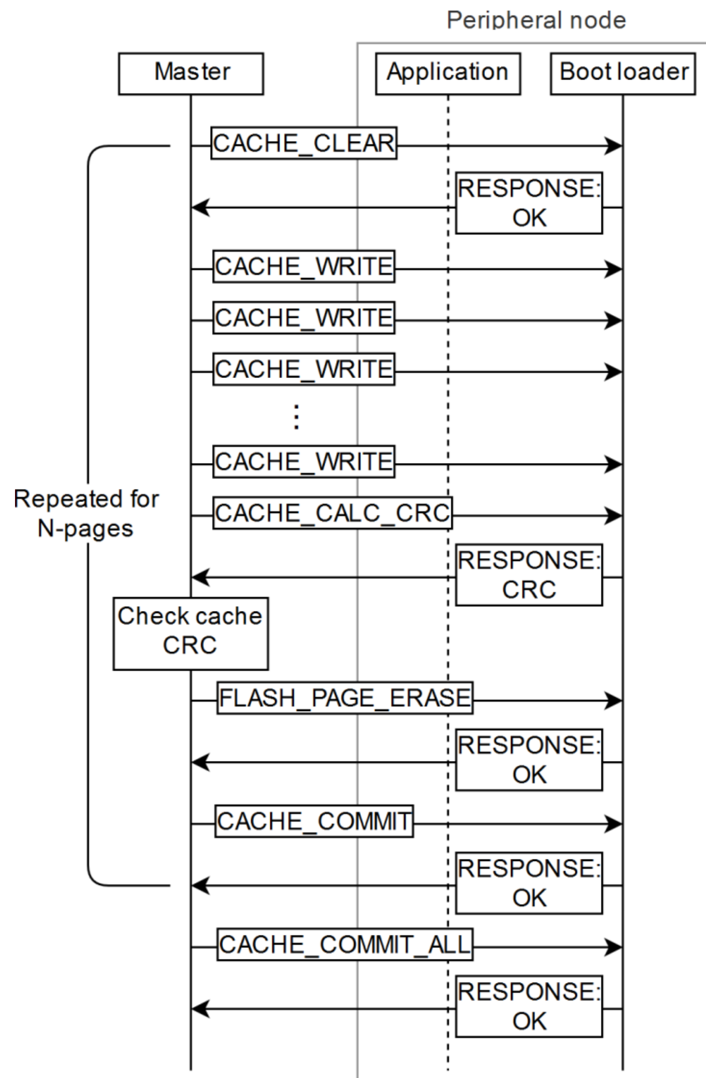

Figure 12. Writing a new user application binary.

Finally, the transition to the user application is requested with the command APP_BOOT. Verification failure in this point may be handled either by restarting the update process or finding the corrupt Flash memory area with FLASH_CALC_CRC. After transitioning

the user application advertises itself on the bus by broadcasting its PRODUCT_ID. For the updater program, this marks the update process as successful.

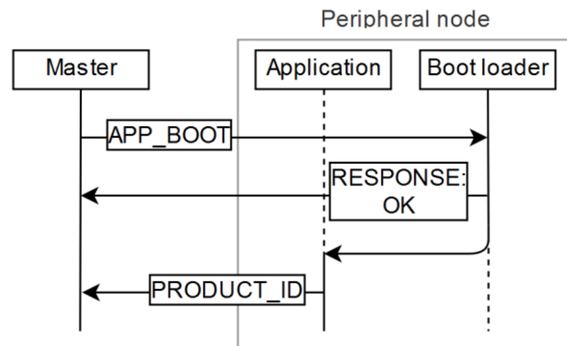

Figure 13. Peripheral node successfully transitioning to user application code.

6.3 Empty boot loader nodes

If the updater encounters an idle device running boot loader code on the bus, a boot to back to application code is attempted. If this fails (due to a missing or broken user application), the latest binary from the firmware library for that device is programmed into the device. The type of the peripheral is resolved by polling the APP_PRODUCT_ID from the boot loader.

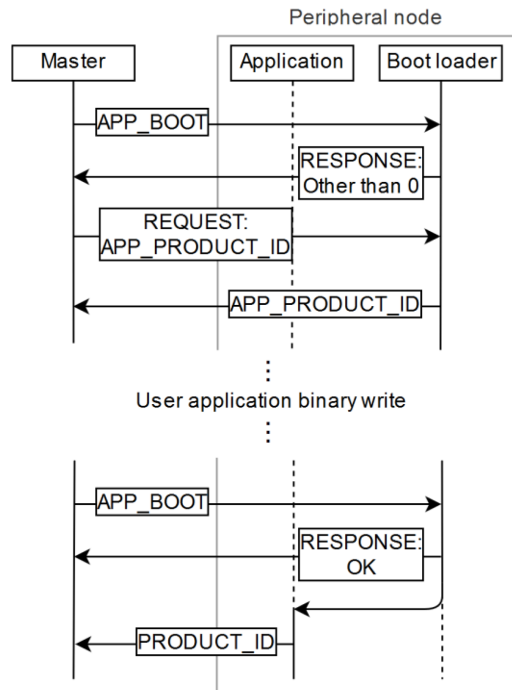

Figure 14. Writing a user application to an empty boot loader device.

7 UPDATE PROTOCOL

This chapter details how the devices use the CAN bus and contains definitions for the update protocol messages.

7.1 Message type

There are two main types of protocol messages: Notifications and Commands. Notifications are messages intended to communicate values or status codes and are most of the time sent after an explicit REQUEST-command. Commands are requests for specific actions to take place, and a response with a specific RESPONSE-notification is expected for each Command message.

Most of the communication between the boot loader and the Master device consists of the Master device sending Commands to the boot loader and the boot loader responding using RESPONSE or Notification-messages. Overall, the boot loader is a passive device, simply carrying out Commands directed to it.

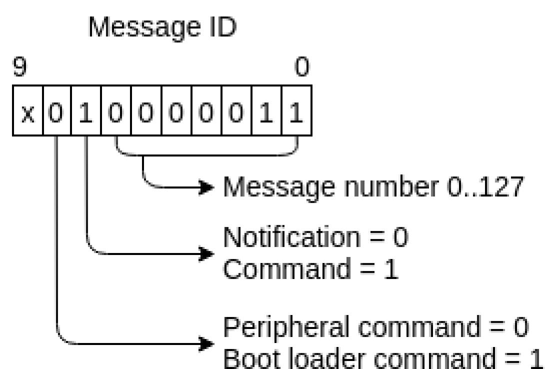

Figure 15. Example of a Message ID for peripheral Command #3.

Message ID (MID) is a 10-bit value which deciphers the type of the message contained in a CAN DATA frame. The first 7 bits of MID are reserved for the identification number of the message. The 8th bit of MID deciphers whether the message is Notification or Command. As the boot loader protocol was built upon an existing one, the boot loader specific commands were additionally distinguished from others by the 9th bit, which signals that the message is specifically a boot loader message. The boot loader

messages also maintain the Notification and Command dichotomy but due to the 9th bit share a different number space with respect to peripheral device commands.

7.2 Message addressing

Each node in the bus has been assigned with a unique 8-bit static address called Local Address. There may be only one device per Local Address on the bus. A node should only process and respond to messages targeted to its Local Address or messages sent to a specific Broadcast Address (value 255).

The following values are placed in the identifier bits as shown in Figure 8:

- Message ID
- Target
- Source

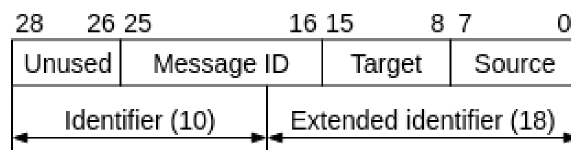

Figure 16 Location of Message ID, Target and Source values in the identifier bits.

Source is the Local Address of the sender node, and correspondingly, Target is the Local Address of the receiving node. The Broadcast Address may not be a Source address.

7.3 Message validation

The CAN bus controller verifies the integrity of each CAN frame with the built-in CRC error check. Frames with unknown MIDs are discarded. The value of DLC is checked against the valid range specific for each message. Standard-length frames are ignored as the message format intrinsically requires the extended CAN frame format. In addition, each message may have its unique validation requirements.

7.4 Device type identification

Each device on the bus advertises its type with the Notification PRODUCT_ID. The master device regularly requests this Notification from the devices on the bus to identify the peripherals. The type of the device is indicated by the parameter Type from the PRODUCT_ID-message. Note that whenever a peripheral transitions to boot loader code, its device type changes to “boot loader”. The master device decipher for which device the boot loader is for by requesting APP_PRODUCT_ID from the boot loader.

7.5 Common messages

Common messages are messages that every CAN device has to implement to be compatible with the CAN protocol. In the boot loader device, common messages are fully supported without the need for Elevation. This chapter will contain definitions for the common messages.

In addition to Common messages, each CAN bus device may have their own message definitions. However, these may not use the MIDs defined for Common messages.

Note this is not a complete list of available protocol messages. Messages required only for the functionality CAN bus protocol and the update process are listed here.

The parameter types displayed in the message definitions follow the integer type definitions from the standard header <stdint.h> from POSIX.1-2017 (POSIX.1-2017, 2017).

7.5.1 ELEVATE

Name:	ELEVATE
Type:	Command
Critical command:	No
MID:	0x180
DLC:	8
Parameters:	Key (uint64_t)
Responses:	0: OK 1: INVALID_KEY 2: UNAVAILABLE 3: NO_BOOTLOADER

ELEVATE grants permission to critical commands to the sending node (node Elevation). A key calculated from the target node's UID is sent with the message (parameter Key) and checked by the boot loader. When successful, the boot loader sets the sending node as the Elevated node on the boot loader.

ELEVATE is also used to trigger a transfer from the user application to the boot loader code. The sender node is then Elevated after the transfer. The reason for the dual use is convenience, as the protocol for gaining access to update functionality is the same regardless if the device is running the boot loader or the user application.

Possible responses:

- OK: Node Elevation was successful.
- INVALID_KEY: The given key did not match.
- UNAVAILABLE: Transfer to boot loader mode is currently unavailable (either another node is already Elevated or the user application is not interruptible).
- NO_BOOTLOADER: Sent by devices with no boot loader.

7.5.2 BL_AVAILABLE

Name:	BL_AVAILABLE
Type:	Command
Critical command:	No
MID:	0x181
DLC:	0
Responses:	0: OK 2: UNAVAILABLE 3: NO_BOOTLOADER

BL_AVAILABLE asks from the device if the transfer to Elevated boot loader mode is currently available. It may be sent to either boot loader or the user application.

Responses are the same as for ELEVATE, except INVALID_KEY is omitted.

7.5.3 REQUEST

Name:	REQUEST

Type:	Command
Critical command:	No
MID:	0x88
DLC:	5
Parameters:	MessageID (uint16_t) DataID (uint16_t) Scope (uint8_t)
Responses:	Notification defined by parameters, or nothing in case of an erroneous request.

REQUEST is used to request a Notification from a device. If the request is valid, the device will respond with the Notification of the type defined in parameter MessageID. Values for parameters DataID and Scope are contextual for each different Notification.

7.5.4 RESPONSE

Name:	RESPONSE
Type:	Notification
MID:	0x08
DLC:	6
Parameters:	MessageID (uint16_t) ResponseCode (uint32_t)

RESPONSE is used to return response values or error codes for Commands. The parameter MessageID tells which Command is being responded to. The parameter ResponseCode contains the response code, specified separately for each Command.

RESPONSE is always sent by a device a Command was targeted to. The target device will send RESPONSE back to the device which issued the Command.

7.5.5 PRODUCT_ID

Name:	PRODUCT_ID
Type:	Notification
MID:	0x00
DLC:	8
Parameters:	Type (uint16_t) Manufacturer (uint16_t) Model (uint16_t) ProtocolVersion (uint16_t)

PRODUCT_ID is used to identify the type of the device connected to the CAN bus. Common practice for polling the devices connected to the CAN bus is to send a REQUEST for PRODUCT_ID to the Broadcast Address, which is then responded by every device with their PRODUCT_ID. Each device also advertises its PRODUCT_ID on the moment it connects to the CAN bus.

Parameter Type is an enumeration value deciphering the type of the device. For example, a marine electronics controller may be 0x01, heater may be 0x02 and so on. When in boot loader mode, a device answers its type as a boot loader.

Parameter Manufacturer is an enumeration value identifying the manufacturer of the device.

Parameter Model is used to identify the hardware model of a device. Usually this value is 0, as most devices have only one model.

Parameter ProtocolVersion is used to identify the version of the CAN protocol the device uses. This value is ignored in the boot loader protocol.

7.5.6 SW_VERSION

Name:	SW_VERSION
Type:	Notification
MID:	0x02
DLC:	8
Parameters:	Major (uint16_t) Minor (uint16_t) Build (uint16_t)

SW_VERSION reports the software version of the peripheral. Software version is composed of three values: Major, Minor and Build. When determining higher version number, Major has preference over Minor and Minor has preference over Build.

7.5.7 HW_ID

Name:	HW_ID
Type:	Notification
MID:	0x03
DLC:	7

Parameters:	Id (uint8_t[6]) Part (uint8_t)

HW_ID returns a device-specific unique identifier. Array parameter Id contains the unique identifier. Parameter Part contains a part number of the string. Upon request, a device may return more than one HW_ID. The parts are then ordered by parameter Part and parameter array Ids are concatenated in order to form the complete unique identifier.

7.6 Boot loader messages

This chapter details the boot loader specific protocol messages. These include the messages used to program and verifying the Flash memory and manage the user application.

7.6.1 RELEASE

Name:	RELEASE
Type:	Command
Critical command:	Yes
MID:	0x186
DLC:	0
Responses:	0: OK

RELEASE relinquishes the Elevation status of the sender node. If the sender was not currently Elevated, the command is ignored. In any case, boot loader will respond with code 0.

7.6.2 APP_VERIFY

Name:	APP_VERIFY
Type:	Command
Critical command:	Yes
MID:	0x1b0
DLC:	0

Responses:	0: User application was successfully verified. 1: CRC checksum failed. 2: Header field APP_SIZE is too large for the Flash memory. 3: Missing user application header (page is erased). 4: Header field PLATFORM differs from the compiled-in value. 5: Header field HEADER_ADDRESS differs from the compiled-in value. 6: Header field APP_OFFSET value is invalid. 7: Header fields for product ID differ from the compiled-in values.

APP_VERIFY will verify the user application currently in Flash memory. RESPONSE to this command will contain the verification status code.

7.6.3 APP_BOOT

Name:	APP_BOOT
Type:	Command
Critical command:	Yes
MID:	0x1b1
DLC:	0
Responses:	0: User application transition triggered successfully. Other responses are the same as in APP_VERIFY.

APP_BOOT will try to trigger the transition to user application code. The response code is the same as for APP_VERIFY. APP_BOOT is usually issued at the end of an update sequence.

7.6.4 APP_ERASE

Name:	APP_ERASE
Type:	Command
Critical command:	Yes
MID:	0x1b2
DLC:	0
Responses:	0: Application erased successfully.

APP_ERASE will erase the whole user application space on the device. However, this command should not be used during normal firmware upgrade, as the user application may utilize part of the free Flash memory area.

7.6.5 HEADER_ADDRESS

Name:	HEADER_ADDRESS
Type:	Notification
Critical command:	Yes
MID:	0x131
DLC:	4
Parameters:	HeaderAddress (uint32_t)

HEADER_ADDRESS contains the compiled-in check value for HEADER_ADDRESS from the boot loader. This value must match the one defined in a user application header.

7.6.6 APP_SPACE

Name:	APP_SPACE
Type:	Notification
Critical command:	Yes
MID:	0x133
DLC:	4
Parameters:	Space (uint16_t)

APP_SPACE contains the available space for user application. Parameter Space contains this value in bytes. On STM32F091xC, this value is at least 112 kilobytes.

7.6.7 APP_PRODUCT_ID

Name:	APP_PRODUCT_ID
Type:	Notification
Critical command:	No
MID:	0x134
DLC:	6
Parameters:	Type (uint16_t) Manufacturer (uint16_t) Model (uint16_t)

APP_PRODUCT_ID contains the compiled-in check values for PRODUCT_TYPE, PRODUCT_MANUFACTURER and PRODUCT_MODEL. This Notification is used to identify for which peripheral device the boot loader is for.

7.6.8 CACHE_CLEAR

Name:	CACHE_CLEAR
Type:	Command
Critical command:	Yes
MID:	0x1a0
DLC:	0
Responses:	0: Cache cleared successfully.

CACHE_CLEAR zeroes the write cache on the boot loader.

7.6.9 CACHE_WRITE

Name:	CACHE_WRITE
Type:	Command
Critical command:	Yes
MID:	0x1a1
DLC:	8
Parameters:	Address (uint16_t) Data (uint8_t[6])
Responses:	None

CACHE_WRITE writes the array parameter Data to the cache address indicated by parameter Address. No response is sent for this command to conserve bus traffic.

7.6.10 CACHE_CALC_CRC

Name:	CACHE_CALC_CRC
Type:	Command
Critical command:	Yes
MID:	0x1a1
DLC:	6
Parameters:	Address (uint32_t) Data (uint8_t[6])
Responses:	None

CACHE_CALC_CRC writes the array parameter Data to the cache address indicated by parameter Address.

7.6.11 CACHE_COMMIT

Name:	CACHE_COMMIT
Type:	Command
Critical command:	Yes
MID:	0x1a3
DLC:	8
Parameters:	Address (uint32_t) Length (uint32_t)
Responses:	0: OK 1: BAD_ADDR 2: BAD_SIZE

CACHE_COMMIT begins writing the cache into the Flash memory. The cache will be written at the address indicated by the parameter Address. The parameter Length indicates the amount of bytes to be written from the cache starting from cache address zero.

Responses BAD_ADDR and BAD_SIZE indicate that the Flash write area indicated by Address and Length is out of bounds.

7.6.12 CACHE_COMMIT_ALL

Name:	CACHE_COMMIT_ALL
Type:	Command
Critical command:	Yes
MID:	0x1a4

DLC:	0
Responses:	0: All cache writes have been committed.

CACHE_COMMIT_ALL blocks the boot loader device and sends a RESPONSE after all active cache writes have been committed into Flash memory. This command is used at the end of an update process to complete the remaining cache writes.

7.6.13 FLASH_PAGE_ERASE

Name:	FLASH_PAGE_ERASE
Type:	Command
Critical command:	Yes
MID:	0x192
DLC:	4
Parameters:	Address (uint32_t)
Responses:	0: OK 1: BAD_ADDR

FLASH_PAGE_ERASE erases a Flash memory page, preparing it for writing. It is used in the update process before committing the cache into Flash. The page where the value of parameter Address points is erased. If Address is out of bounds, BAD_ADDR is returned.

7.6.14 FLASH_CALC_CRC

Name:	FLASH_CALC_CRC
Type:	Command
Critical command:	Yes
MID:	0x193
DLC:	8
Parameters:	Address (uint32_t) Size (uint32_t)
Responses:	CRC checksum over an area of Flash memory.

FLASH_CALC_CRC calculates a CRC checksum over an area of Flash memory. The parameter Address indicates the starting address and Size the length of the CRC calculation. FLASH_CALC_CRC is used to locate a write error in case of user application verification failing during an update process.

7 CLOSING CHAPTER

Overall, the project to develop an update system for the device network was a successful one. The protocol, boot loader and updater were developed at Nextfour Group Oy during a four month period in 2018. As for May 2018, the update system is undergoing final testing. After the final validation is complete, the update system will be released as a new feature in the Q Experience product.

APPENDIX 1. BOOT LOADER OPTION REGISTER

Option	Description
BOOT_MODE	Determines the action taken upon boot loader start up. Possible values: NORMAL_MODE: The application is run if its verification passes. Otherwise, Program Mode is entered. MODE: Program Mode is entered directly.
ENABLE_FLAGS	These bit flags mark which boot loader options have been enabled. Contained flags and value implications: LOCAL_CANID_ENABLED: 1 to enable LOCAL_CANID, 0 otherwise. ELEVATED_CANID_ENABLED: 1 to enable ELEVATED_CANID, 0 otherwise. CLEAR_RESET_FLAGS: If 1, reset flags of the microcontroller will be set to default values before running the user application.
LOCAL_CANID	If enabled, overrides the default CAN Local Address with the contained value.
ELEVATED_CANID	If enabled, automatically elevates the contained CAN Local Address.