Thomas Niedermayr, BSc

Enabling Wireless Automotive SW Updates for the Infineon AURIX ECU

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dott. Dott. mag. Dr.techn. MSc Carlo Alberto Boano

Institute for Technical Informatics

Graz, August 2017

AFFIDAVIT

Kurzfassung

Aufgrund der steigenden Komplexität und dem wachsenden Funktionsumfang moderner Fahrzeuge ist die Anzahl an integrierten Steuergeräten stetig steigend. Sowohl die steigende Zahl der Steuergeräte, als auch die zunehmende Komplexität der Software erfordern einen einfachen, zuverlässigen und schnellen Weg, diese auf den Steuergeräten zu aktualisieren. Der Vorteil einer drahtlosen Übertragung von Softwareupdates zum Fahrzeug liegt darin, die Performance zu steigern und Fehler zu beheben, während auf den bisher dafür notwendigen und kostenintensiven Rückruf der Fahrzeuge verzichtet werden kann. Des weiteren würde es Erstausrüstern (OEMs) erlauben neue Features und Upgrades aus der Ferne freizuschalten und dem Kunden einen noch persönlicheren Service zu bieten.

In dieser Arbeit designte und entwickelte ich ein flexibles und effizientes drahtloses Updatesystem für Fahrzeuge. Dieses besteht aus einem Diagnosetester, einem drahtlosen Fahrzeuginterface und dem AURIX TC277c Steuergerät. Der Diagnosetester (z.B. Smartphone, Laptop oder Tablet) ist drahtlos mit dem Fahrzeuginterface verbunden, welches widerum direkt an das Fahrzeugbussystem angeschlossen ist. Um Vertrauen vom Diagnosetester bis hin zur Fahrzeug-ECU herzustellen, wird eine "Seed und Key" Challenge-Response-Authentifizierung verwendet. Ein 32-Bit-CRC schützt die Integrität auf dem gesamten Datenpfad. Um die Effizienz des Datentransfers am Fahrzeugbus zu steigern, verwendet das entwickelte System eine modifizierte Version des "UDSonCAN" Protokolls. Das vorgestellte System beherrscht zudem parallele Updates, bei denen Daten gleichzeitig an mehrere ECUs gesendet werden. Diese ECUs sind entweder am selben Fahrzeugbus angeschlossen, oder in unterschiedlichen Fahrzeugen verteilt. Um die Geschwindigkeit des Updatesystems weiter zu verbessern, bietet das System partielle Updates, bei denen versucht wird, nur die Veränderungen zwischen zwei SW-Versionen zu übertragen. Sollte während des Updateprozesses ein Fehler auftreten oder eine neue SW-Version fehlerhaft sein, bietet das System die Möglichkeit eine zuvor gesicherte SW-Version wiederherzustellen.

Das System wurde experimentell getestet und mit einem drahtlosen Updatesystem für die Volvo-FlexECU verglichen. Die Änderungen am "UDSonCAN" Protokoll beschleunigen den Datentransfer am Fahrzeugbus um 38%. Ein einfaches drahtloses Update benötigt lediglich 43% der Zeit verglichen mit der Volvo-FlexECU. Bei Verwendung von parallelen Updates konnte mit der getesteten Update-Dateigröße ein um 38% schnelleres Update ausgeführt werden als bei einem sequentiellen Update. Partielle Updates erlaubten im getesteten Szenario eine Verkürzung der Updatezeit auf nur 17%, verglichen mit einem einfachem drahtlosen Update.

Abstract

Due to the rising complexity and the increasing functionality of modern vehicles, the number of electronic control units (ECU) embedded in a vehicle is continuously growing. This, together with the increasing code complexity of ECUs, demands for an easy, reliable, and fast way to update the ECU software. Updating the software of ECUs over the air would indeed enable performance improvement and bug fixes without the need of expensive vehicle recalls, and would further allow Original Equipment Manufacturers (OEMs) to upgrade or enable new features remotely, as well as to offer even more personalized services.

In this thesis, I designed and developed a flexible and efficient concept for wirelessly updating automotive software. The system consists of a diagnostic test device, a wireless vehicle interface, and an Aurix TC277c ECU. The diagnostic test device (that can be a smart-phone, laptop, or tablet) is wirelessly connected to the wireless vehicle interface, which is directly wired to the vehicle bus system. To establish trust between the diagnostic test device all the way to the ECU, a seed and key challenge response system is used, whereas a 32-bit CRC ensures integrity on the complete data path. To increase the efficiency of the data transfer on the vehicle bus, the implemented system uses a modified "UDSonCAN" protocol. The update system is also capable of performing parallel updates, i.e., simultaneously sending data to multiple ECUs connected to the same vehicle bus or distributed to multiple vehicles. To further speed up the update process, the system also implements a partial update functionality, which identifies the changes between different software versions and only transfers the portion of code that has actually changed. If any error occurs during the update process, or the uploaded software is faulty, the system allows to restore a previously saved firmware version.

The developed system was evaluated experimentally and compared to a wireless update system for the Volvo FlexECU. The modifications to the "UDSonCAN" protocol increased the data transfer speed by 38%, and performing a standard wireless update resulted to be 57% faster compared to the reference system. Parallel updates allowed a performance improvement of up to 38% compared to a sequential update, whereas a partial update enables a speed up of 83% in the tested scenario.

Credits

This thesis was written in 2017 at the Institute for Technical Informatics (ITI) at Graz University of Technology and at the the Virtual Vehicle Research Center, both located in Graz, Austria.

I want to express my gratitude towards Marco Steger from the Virtual Vehicle Research Center who assisted in the development of the implemented system and also provided a lot of expertise. He was also very helpful during our many talks, where he provided moral support during the many problems when implementing the system. I also want to thank Dr. Carlo Alberto Boano for the very good assistance during the design of the system and the writing process of the thesis. He was always available for questions and I could always rely on a fast answer. A big thanks goes to Dr. Allan Tengg for the hour long debug sessions and the persistence in finding the errors. Also thanks to the Virtual Vehicle Research Center for the opportunity to perform this thesis and the provided hardware.

A special thanks to my girlfriend Sarah Stummer, you were always very understanding during the long hours and late evenings. You also provided the best moral support when I was doubting that the system would even work at all. Thanks for the countless dinners you cooked alone, because I was working on this project. Thanks, for the mostly welcome interruptions that often gave me the ideas I was looking for for hours.

Graz, August 2017 Thomas Niedermayr

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increasing complexity of automotive software (SW) and the rising number of new services that vehicles have to offer is drastically increasing the code size in modern electronic control units (ECUs). More lines of code also introduce more bugs. In the past, such bugs were either ignored, if not critical for the proper operation of the vehicle and the safety of its passengers, or had to be fixed in a workshop through a wired connection. Recalling vehicles to perform a "wired" SW update is indeed quite expensive for Original Equipment Manufacturers (OEMs). Therefore, if a non-critical bug was present in an ECU (for example, causing the entertainment system to crash every twentieth time), an OEM would leave the customer unsatisfied and fix the bug in the next vehicle model, rather than calling back thousands of vehicles to perform a wired SW update. In recent years TESLA introduced the world to the possibilities of modern technology in vehicles [Gab16]. They were the first to implement a large scale wireless update system to keep their fleet up to date and add new features. One advantage they have, compared to other vehicle manufacturers, is that they only offer pure battery electric vehicles that are much simpler than combustion engine vehicles. This reduces the load on the engine control unit and simplifies the development of the engine control SW. Nevertheless, the possibility to update the software of ECUs over the air (OTA), and hence to enable performance improvement and bug fixes without the need of expensive vehicle recalls, is very attractive to any OEM. The latter can use wireless SW updates to upgrade or enable new features remotely, as well as to offer even more personalized services to customers. Furthermore, the use of OTA SW updates is not only limited to the remote download of up-to-date software directly by the car owners, but can also be exploited in several other stages of a vehicles lifetime: from the vehicle development and the manufacturing stage on the assembly line, to the diagnosis of problems in a workshop or service center.

To enable those functionalities, I developed and evaluated a flexible and efficient concept for wirelessly updating automotive SW. The system was evaluated in a testbed consisting of one diagnostic tester, two wireless vehicle interfaces (Section 3.5) and two AURIX ECUs. Such a system, sketched in Figure 1, can be used to enable the wireless remote installation of new features and to drastically improve the efficiency of maintenance of modern vehicles. The diagnostic test device can be a laptop, tablet, or PDA and is used to provide the user interface and the control of the update system. It is connected via 802.11s to a wireless vehicle interface (WVI), which is directly wired to the vehicle bus system through which all ECUs are interconnected. In this work I used the Controller

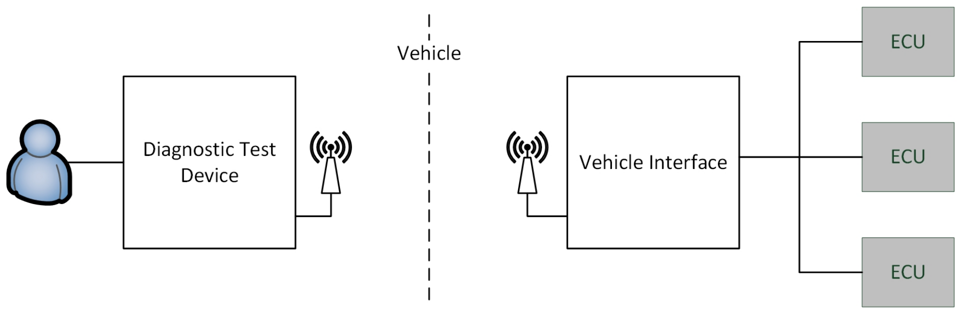


Figure 1.1: Abstract system architecture of the wireless SW update system developed in this thesis.

Area Network bus (CAN) for the connection between the WVI, and the Infineon AURIX as electronic control unit.

1.1 Goals

This thesis aims to practically implement a concept for OTA automotive SW updates and evaluate the latter on a real ECU. The update concept should address four important top level goals:

- Security
- Integrity
- Efficiency
- Safety

Efficiency: Some situations, like the VW emission scandal [New15] require that a lot of vehicles need to be updated in a cost and time efficient way. So the goal of the introduced system is to provide an update mechanism that is as efficient as possible. To improve efficiency of OTA updates for automotive systems, the update sytem to be designed needs to support *parallel* and *partial* updates. Updating multiple ECUs at the same time with the same update is called a *parallel* update and can be done in two ways. Either two or more ECUs are connected to the same vehicle interface through the vehicle bus, or two or more WVIs are connected to the diagnostic test device. For both approaches the same update is done on multiple ECUs at the same time.

A parallel update can be used for the so called “workshop scenario”, where multiple vehicles in a workshop need to be updated. A mechanic can use the wireless update system to simultaneously update the SW in multiple vehicles. A *partial* update hence means that only the portions of the SW that have changed are wirelessly transferred, speeding up the whole update process. This method is very suitable for version updates and bugfixes, as in those cases often only small portions of the SW need to be changed. In a perfect scenario where only one byte changes, only this single byte has to be transferred. This has the potential to drastically decrease the transferred data.

Security: In recent years there were some big scandals with vehicles being hacked and controlled remotely to shut off the engine or control the entertainment system [And15]. This had the consequence that security concerns needed to be addressed in vehicle design. Introducing a system that is capable of changing the SW that controls the car is indeed a major concern for a lot of people. Because of that, it has to be guaranteed that the update is only done in a secure environment. To minimize the overhead and maximize efficiency, an authentication with keys is used to verify the authenticity of the diagnostic test device to the WVI.

Integrity: As shown in Figure 3.1, there are multiple data transmissions involved and the data for the update is read and stored multiple times. In such a system it can happen that some pieces of data get corrupted by an error when reading or storing the data. It is also possible that the data gets modified during transfer.

Safety: In automotive systems every safety relevant system has to have a fail safe mechanism, also called "safe state". To give an example, imagine an electronic steering column lock that uses a bolt to lock the steering wheel. If any part of the complete system fails during operation of the vehicle, it has to be guaranteed that the steering must not be locked. In this case, the "safe state" would be that the bolt stays open indefinitely. For the software update system this concept is used without that many restrictions, the biggest one being that the vehicle is not moving during the update, and another being that the system does not include any moving mechanical components. This radically reduces the safety efforts that have to be implemented in this work and the costs for such measures.

1.2 Challenges

To reach the goals defined above, there were some specific challenges to conquer. Some of those challenges are specific to the system architecture and some are general challenges when designing such a system.

The first challenge deals with security. As I build on top of the SecUp framework [SBK$^+$16], this thesis focuses mostly on the verification of the diagnostic test device, to ensure that the diagnostic test dvice is authorized to perform an update. Another set of challenges comes from the validation of the update process, to ensure that not even a single bit has changed. First, I want to ensure the integrity of the transferred data from the diagnostic test device until it is stored at the ECU. The next challenge is to ensure that the updated SW is executable and that it initializes correctly. The problem in this regard is to find a trade-off between keeping the SW independent of the update system and the error handling capabilities of the system. Which means, to keep the changes in the SW to be updated to a minimum, but ensure that at least the SW is running after an update.

The next challenges concern efficiency. To improve the overall time for the update process, I identified the data transmission between the vehicle interface and the ECUs as the bottleneck. On the one hand, the data throughput has to be maximized over the connection between the interface and the ECU, which means, a redesign of the used protocol. The ECU must still be able to identify missed or out of order packages. On the other hand, the data to be transferred has to be minimized, which can be done by utilizing parallel and partial updates. Parallel updates include multiple ECUs and it has to be managed, that every ECU knows if it is part of the current update or not. Executing partial updates, there is the trade off between complexity and bigger deltas. This means, the bigger the smallest part in a partial update, the easier it is to manage. When the data transfer is optimized, the ECU has to keep up with the increased transfer speed by using architecture specific commands to optimize performance when storing data.

The last challenge arises from safety concerns. The system must be able to recognize errors during the update process and retain a functional state. The detection of errors is already covered by other challenges, which leaves the handling of those errors to be the last challenge for the system. The handling of those errors has to be done in a way that keeps the system in a fully-functional state.

During the implementation of the update concept, one of the biggest challenges was the instability of the debugging environment. Most of the times the debugger was connected to the ECU that received the update, but apparently the behavior of the microcontroller changes when connected to the debugging utility. This really complicated the development procedure. Another problem was the poorly documented ECU, especially in the sections describing how to write to flash memory and how to handle its write protection. This, combined with the limited debugging capabilities, drastically slowed down the implementation and the debugging process.

1.3 Contributions

The key contribution of this thesis is the introduction of an efficient and feature rich wireless update system that utilizes architecture-specific functions of the used ECU. The system uses a redesigned Unified Diagnostic Service (UDS) protocol that maximizes the data throughput during the transfer, but still keeps the functionality of the protocol. As it is standard in automotive industry, the diagnostic test device is authenticated by the ECU with a seed and key algorithm. To ensure data integrity, a cyclic redundancy check (CRC) is used, so that integrity can be ensured along the complete data path. Once the data is transferred and the updated SW is running, a diagnose method is used to check if the initialization was done correctly. Furthermore, the system introduces an approach to handle multiple executable programs on one ECU that can be selected during the boot procedure.

To reduce the amount of data, the system is capable of parallel updates (parallel upload of the data to the ECU). The benefit of a parallel update increases with the number of connected ECUs, as the data is only sent once to all connected devices that need to be updated. The implemented system is also capable of basic partial updates. A basic partial update means that the overhead to minimize the transferred data is kept to a minimum. This implicates that if only one byte changes, the whole memory sector containing that byte has to be transferred. In a more advanced system, it can be possible to transmit only the changed byte and the rest of the management is done on the ECU. Both of the two previously mentioned update mechanisms are implemented in a way that does not affect the code size of the bootloader or the updated SW. The necessary calculations are done on the diagnostic test device to reduce the code overhead on the ECUs. Because there is always a possibility of failure, I also implemented a fail safe mechanism. The implemented system offers the possibility to back up a running SW and, in case of an error during the update, the system is able to perform a rollback to the previously backed up SW. Figure 1.2 shows the connection between the goals, challenges and the contributions.

Scientific publications: The implemented system is combined with the SecUp framework [SBK$^+$16] and will be released as a revised version of the SecUp paper later this year.

It utilizes the presented systems capabilities (parallel and partial update) and system optimizations and focuses on the security and safety aspects of an automotive wireless update system.

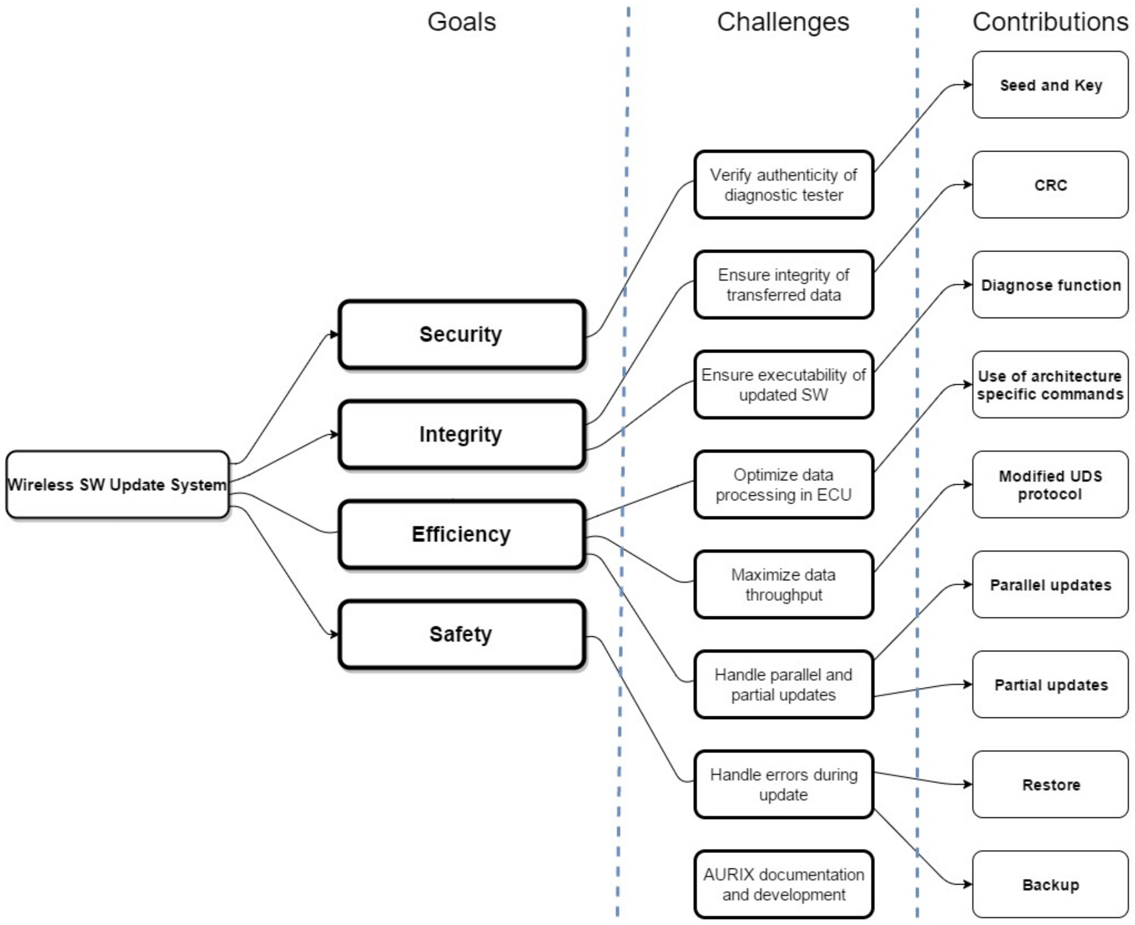

Goals
Challenges
Contributions
Wireless SW Update System
Security
Integrity
Efficiency
Safety
Verify authenticity of diagnostic tester
Ensure integrity of transferred data
Ensure executability of updated SW
Optimize data processing in ECU
Maximize data throughput
Handle parallel and partial updates
Handle errors during update
AURIX documentation and development
Seed and Key
CRC
Diagnose function
Use of architecture specific commands
Modified UDS protocol
Parallel updates
Partial updates
Restore
Backup

Figure 1.2: Connection between goals, challenges and contributions.

1.4 Structure

The remaining part of this work is structured as follows. In Chapter 2, I give an introduction into the necessary background starting with the automotive bus transmission protocols followed by the wireless protocols for in-vehicle communication. After that I will describe the used hardware which includes the Infineon AURIX application kit and the platform that is used for the vehicle interface as well as the diagnostic test device. The chapter is ended with the description of the used CRC algorithm and a short introduction to GNU linker files as well as the Intel Hex Format. Chapter 3 describes the system architecture, before Chapter 4 illustrates the design choices and describes the implementation details. In Chapter 5 I cover the evaluation of the different contributions before Chapter 6 discusses past work related to this thesis. Finally Chapter 7 concludes the thesis and presents future work to further improve the presented system.

Chapter 2

Background

This chapter introduces the necessary background and clarifies some terms that will be used throughout this work. For all used abbreviations see the list of abbreviations at the end of this thesis. Section 2.1 illustrates the automotive transmission protocols used to communicate between the WVI and the ECU. It describes the CAN protocol and its corresponding top level protocol "UDSonCAN" (ISO 14229-3). Following is the description of the hardware that is used for the testbed, which includes two different platforms (Section 2.3). The chapter continues with the description of the used CRC algorithm (Section 2.4) and a very short introduction into linker files (Section 2.5) which are very important for memory management during development. Finally, Section 2.6 introduces the basics of the Intel Hex File format, which is the file format of the SW to be uploaded and is generated by the toolchain used to develop the SW.

2.1 Automotive Protocols

The implemented system has two main data transmission paths with very different physical properties. This requires the use of different transmission protocols and transceivers (transmitter and receiver). The first transmission is a wireless connection between the diagnostic test device and the WVI. The second one is the connection between the WVI and the AURIX. As the system should nicely integrate into the existing vehicle infrastructure, it uses the existing data transmission buses installed in vehicles. The most popular is the CAN bus and is therefore used in this work. On top of the CAN protocol, the "UDSonCAN" protocol described in Section 2.1.2 manages higher level diagnostic functions. In future vehicles, there will probably be more advanced and faster transmission systems which will require some adaptions to be used with the implemented system, or the new transmission systems already provides mechanisms that had to be added to the UDSonCAN protocol in this work.

2.1.1 Controller Area Network (CAN)

CAN 2.0 was introduced in 1991 by Robert Bosch GmbH. It is a vehicle bus standard that allows different devices inside a vehicle to communicate with each other. It is a message based protocol, which means there is no receiver or sender address. Instead, every message has an identification that represents the contents of that frame. It was originally designed for multiplex electrical systems in vehicles, and since then it is also used in many other contexts.

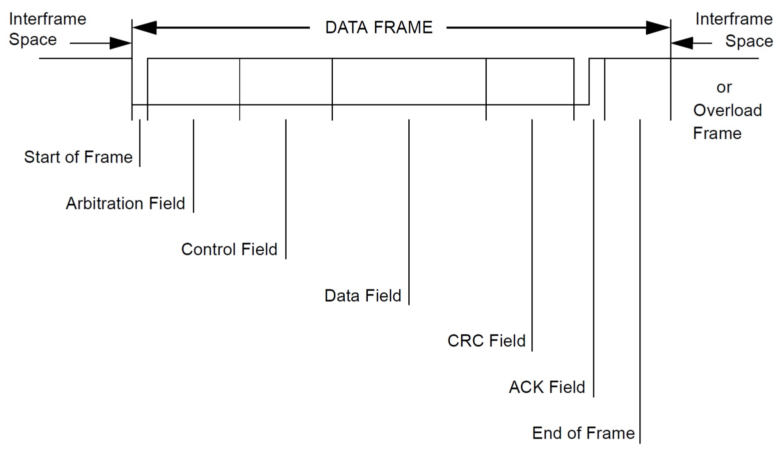


Figure 2.1: CAN Frame Format 2.0.

The CAN bus has some very important properties:

- prioritization of messages;
- guarantee of latency times;
- system-wide data consistency;
- error detection and signaling;
- automatic re-transmission of corrupted messages as soon as the bus is idle again.

Information is sent in fixed format messages of different but limited length. There are two bus levels, 0 as the dominant level and 1 as the recessive level. Transmission is event-triggered, which means the nodes decide individually when to send and there is no master/slave or scheduling required. Therefore, the CAN node checks the bus and, if free, it starts transmitting. At the beginning of the message (see Figure 2.1) there is the Arbitration Field which contains the ID for the message. If two nodes start transmitting at the same time, a collision occurs. This is handled through the following technique. Every node that is sending also monitors the bus and, if a recessive bit is sent, but there is a dominant level during the arbitration field, the node that detects the inconsistency stops transmitting without signaling an error and starts again after the bus is free. To recognize bit errors inside the frame, a 15 bit CRC checksum is included in every message. The transceivers count the number of errors and, if there are too many, they shut down.

2.1.2 UDS - “UDSonCAN” (ISO 14229-3)

Unified Diagnostic Services (UDS) is a diagnostic communication protocol specifically designed for the automotive environment. ISO 14229 tries to generalize the principles of KWP 2000 diagnostics (ISO 14230-3) to make them independent of the underlying bus protocols. ISO 14229-3 describes the mechanism to implement UDS on the CAN bus. As it is a higher level protocol, it only defines the contents of the data frame from the CAN protocol. The message structure of those data frames can be seen in Figure 2.2. A request always starts with a service identifier (ID) specifying the type of service that is requested. For more complex services and some communication control, there is also one byte for the sub function level. The rest of the message is the parameter for the defined service request. The response, if positive, contains the same service ID ored with 0x40 and the same sub function level. For errors there is a list of error IDs and error response codes.

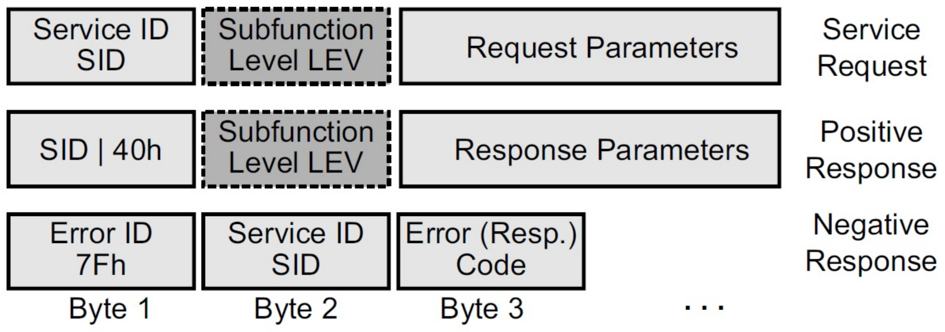


Figure 2.2: Structure of UDS frames (CAN data frame).

The specification of UDS still leaves vehicle manufactures room for specific commands. All ECUs in modern vehicles support this diagnostic service, and one specific function of it is to update the firmware. UDS also defines specific sequences that have to be executed in order to execute a firmware update. It also defines different sessions:

- **Default diagnostic session:** standard session to read and delete diagnostic trouble codes and session control;
- **Programming session:** used to update the firmware;
- **Extended diagnostic session:** unlocks additional diagnostic commands (e.g., sensor adjustment);
- **Safety system diagnostic session:** to test safety relevant functions (e.g., airbag).

The standard also reserves some session identifiers that can be used by vehicle manufacturers and suppliers for specific sessions. For security purposes the standard also defines a “Seed and Key” procedure for security critical services. A “Seed” is sent from the ECU to the diagnostic test device. The device computes a corresponding “Key” and sends it back to the ECU to unlock the requested services.

Table 2.1 lists a few example commands.

Service	CAN data frame	Description
Diagnostic session control	0x1001	Default Session
	0x1002	Programming Session
Security Access	0x2701	Request Seed
	0x2702	Send Key
ECU Reset	0x1101	Hard Reset
	0x1102	Soft Reset
	0x1140 - 0x117E	Manufacturer specific commands

Table 2.1: Sample UDS commands.

2.2 Wireless protocols for in-vehicle communications

In todays vehicles, wireless in-vehicle entertainment and communication is on the rise. Vehicle manufacturers like Opel with their “On-Star” [Ope17] on-board WIFI solution and VW with their “Car-Net” [Vol17] solution, are bringing wireless communication into the vehicles. In the context of this thesis, we have quite a lot requirements to the wireless vehicle communication, that are not all present in the aforementioned solutions. In the following the most relevant requirements that must be considered designing a wireless update system are listed.

- **Reliability:** the automotive environment is quite harsh in terms of interference due to the firing of the spark plugs and the seclusion of the vehicle to the outside world.
- **Security:** as security is a very hot topic for vehicle manufacturers, adding a WVI to the vehicle that is capable of updating the SW of an ECU is a very critical task.
- **Interconnection:** the WVI should be able to connect to existing networks (e.g., a workshop WLAN) to enable remote updates.
- **Throughput:** due to SW sizes of infotainment systems exceeding 100 Mb, the wireless connection needs to offer high throughput for a fast data transfer.
- **Extendability:** the system should be easily extendable to cover larger distances without the need to reconfigure the whole network.
- **Multicast:** the system should be able to send updates to multiple vehicles at once, enabling parallel updates, where several vehicles can be updated at the same time with the same SW version.
- **Functional safety:** in the automotive environment, functional safety is very critical topic. As updating a vehicle ECU is for sure a safety relevant task, functional safety requirements have to be considered selecting the wireless protocol.

With the listed requirements, a suitable communication technology for the wireless update system can be selected. There are a lot of very power-efficient protocols like IEEE 802.15.4, which was designed specifically for power efficiency and simplicity, and Bluetooth Low Energy (BLE), which are not suitable for the desired system due to the low data rate. IEEE 802.11p [80210] was developed specificaly for the automotive environment and is intended for vehicle-to-vehicle communication. A problem is the availability and the expensive price of the hardware. On top of that, the connection to other IEEE 802 networks would be hard, because IEEE 802.11p uses an automotive specific stack and is not IP based. Other IEEE 802.11 networks like IEEE 802.11b/g/n offer the desired data rates and range, and are easily connectable to most of the existing infrastructure as they are the most commonly used wireless protocols. More and more of those wireless chips also support IEEE 802.11s, which is an amendment for mesh networking. It allows additional devices to be used as relay nodes (e.g., a parked vehicle or a placed relay node) between two end nodes to extend the range of the wireless update system without any configuration of the network or the nodes.

2.2.1 IEEE 802.11n

IEEE 802.11n, published in 2009, is the latest broadly used amendment to the IEEE 802.11 protocol. Due to its maximum data rates from 54 Mbit/s up to 600 Mbit/s it is the fastest broadly available and supported standard to date. It adds support for multiple-input multiple-output (MIMO) and 40 MHz channels. MIMO is a technology utilizing multiple antennas to resolve more information than using one antenna. Additionally, the 40 MHz channels double the data rate compared to the previously used 20 MHz channels. This leads to a transfer rate superior to IEEE 802.11a and IEEE 802.11g. IEEE 802.11n is also backwards compatible and enables to connect to devices supporting previous standards (IEEE 802.11b/g). The benefit of the MIMO technology strongly depends on the number of antennas used. For every antenna at the sender and the receiver, an additional data stream can be transferred. To reduce the overhead of the multiple protocol layers, IEEE 802.11n utilizes frame aggregation, which is the process of sending multiple data frames in one transmission. Therefore, multiple Ethernet frames are collected and wrapped in a single IEEE 802.11 header. Because the Ethernet headers are much shorter compared to IEEE 802.11, this drastically reduces overhead and increases the throughput.

2.2.2 IEEE 802.11s

IEEE 802.11s extends the IEEE 802.11 Media Access Control standard by defining a protocol, that supports broadcasts, multicasts and unicasts. To achieve this, it uses radio-aware metrics over self-configuring multi-hop technologies. It defines the Hybrid Wireless Mesh Protocol [80206] as a default mandatory routing protocol which uses on-demand ad hoc routing and tree based routing. Wireless mesh network devices (mesh stations) form mesh links with one another, which forms so called mesh paths that can be used for multi hop communication. IEEE 802.11s also includes mechanisms for deterministic network access, congestion control and power save.

As IEEE 802.11s only changes the MAC layer, it depends on IEEE 802.11 a/b/g/n to carry the actual data. In the implemented wireless update system, IEEE 802.11s is used with IEEE 802.11n carrying the actual traffic.

2.3 Hardware

The testbed requires two different hardware components. First, a Beagle Bone Black is used as diagnostic test device and as WVI. The two Beagle Bone Black boards are equipped with a TP-Link WLAN-USB adapter to enable an 802.11s connection. The WVI is also equipped with a custom made CAN-Cape, illustrated in Figure 2.5, that enables a CAN connection between the WVI and the AURIX platform, which is the second hardware component.

2.3.1 Beagle Bone Black

Beagle Bone Black (BBB), as seen in Figure 2.3, is a low-cost, community-supported linux development platform for developers and hobbyists. It has an AM335x 1GHz ARM Cortex-A8 CPU and 512MB RAM. For further extensions it also hosts a USB connector and two 46 pin headers. It is optimized to boot linux in under 10 seconds and can easily be programmed through the USB port.

Diagnostic Test Device (DT)

The diagnostic test device, seen in Figure 2.3, is based on the BBB, which is equipped with a TP-LINK TL-WN722N WLAN-USB-Adapter to enable an 802.11s connection to the vehicle interface.

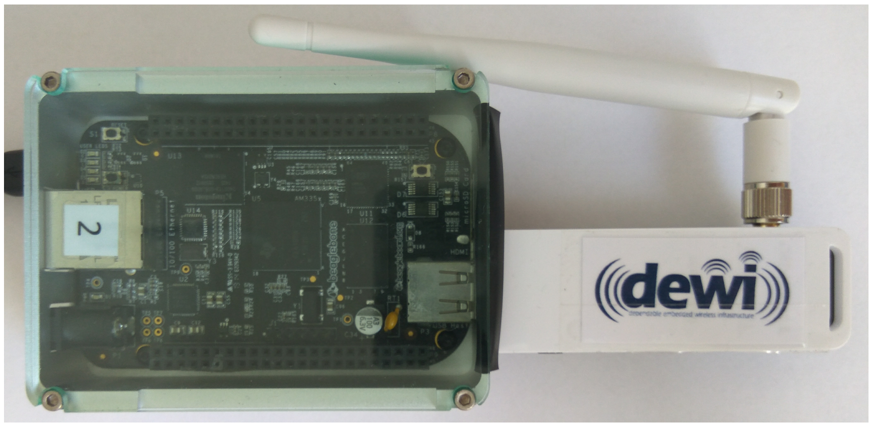


Figure 2.3: Diagnostic Test Device (BBB with WLAN-USB-Adapter).

Wireless Vehicle Interface (WVI)

The wireless vehicle interface, seen in Figure 2.4, is also based on a BBB and is placed inside the vehicle. It is equipped with the same WLAN-USB-Adapter a the DT, which enables a wireless connection outside of the vehicle. Furthermore the BBB is equipped with a CAN cape (see Section 2.3.1) which connects the WVI to the vehicle bus system, and enables it to communicate with the ECUs inside the vehicle.

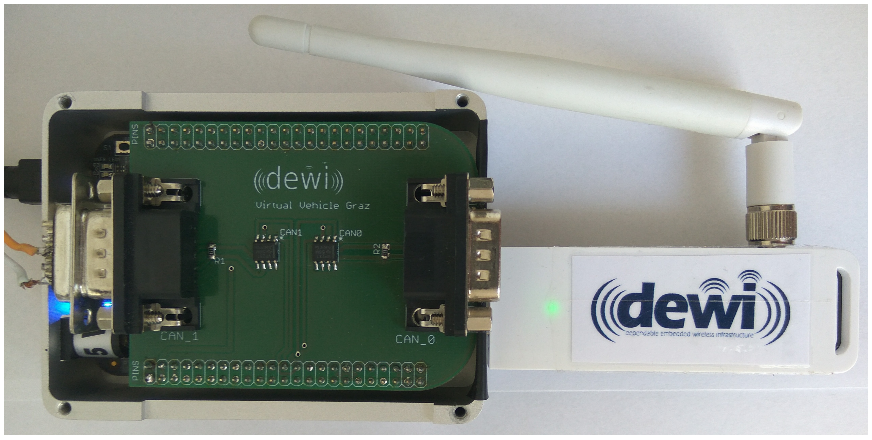


Figure 2.4: Wireless Vehicle Interface (BBB with CAN Cape and WLAN-USB-Adapter).

CAN Cape

Figure 2.5 shows the cape used to enable a CAN connection between the ECU and the WVI. It is equipped with two Texas Instruments CAN transceivers (SN65HVD231), each connected to a 9 pin D-Sub connector. The cape is plugged into the BBB with two 46 pin headers.

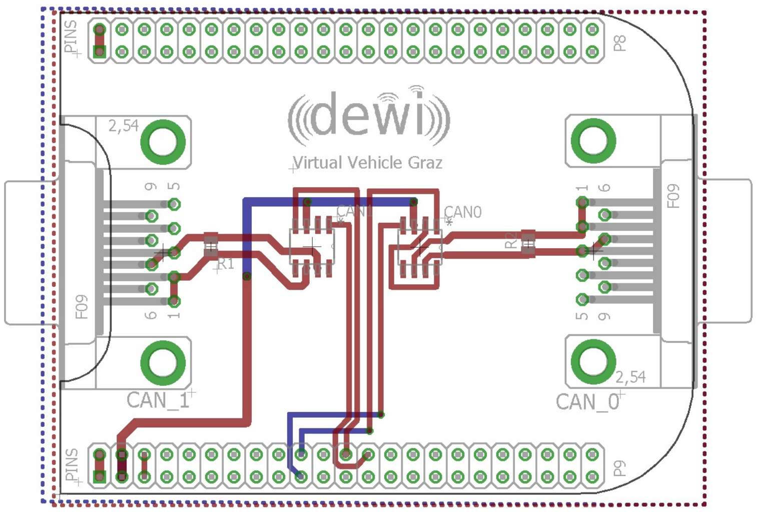


Figure 2.5: Schematics of CAN Cape for Beagle Bone Black.

2.3.2 AURIX ECU

I use the AURIX Application Kit TC277 TFT, illustrated in Figure 2.6 as electronic control unit. It is a high performance platform compliant to support safety requirements up to ASIL-D[1]. Among others, it has the following features:

[1]Automotive Safety Integrity Level (ASIL) is a risk classification scheme with four levels, A to D with D being the highest.

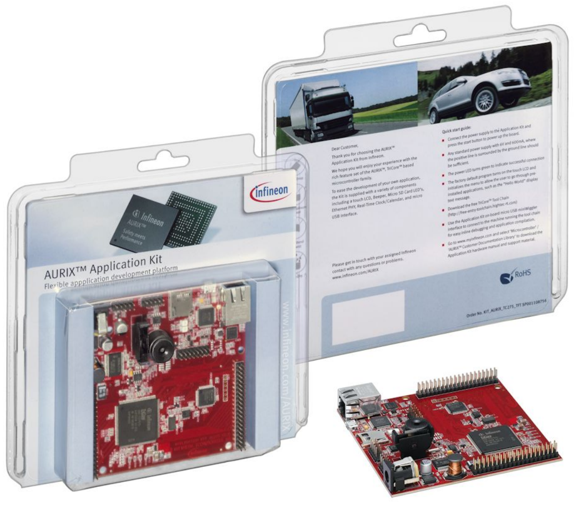


Figure 2.6: TC277 TFT application kit. (Source: Infineon, 2017)

- 3 x 32-bit scalar TriCore^{TM} CPU running at 200 MHz in the full automotive temperature range;
- Up to 4MB Flash and 472KB RAM;
- Ethernet 100 Mbit/sec;
- Support for FlexRay, CAN, CAN FD, LIN, SPI;
- Multi Voltage Safety Micro Processor Supply TLF35584;
- Dedicated closely coupled memory areas per core.

Figure 2.7 shows the block diagram of the TC27x. Relevant for this thesis is the MultiCAN+ module on the left, which is connected to the system peripheral bus (SPB), and the flash modules, in the middle of the upper half that are connected to the CPUs via the Shared Resource Interconnection (SRI) high speed system bus.

AURIX Program memory unit (PMU)

The Program Memory Unit (PMU) controls the flash memory and the BootROM. For the description of the flash memory and its operations I need a specific terminology.

- **Flash module:** A PMU contains one flash module with its own operation control logic.
- **Bank:** A flash module contains separate banks. In the PFlash there are one or more PFx banks and in the DFlash two DFx banks. Banks support concurrent operations with some limitations due to common logic.

- **Page:** In the PFlash a page is an aligned group of 32 bytes and in DFlash of 8 bytes. It is the smallest unit that can be programmed.
- **Program burst:** A burst is the maximum amount of data that can be programmed with one command. The programming throughput is higher than for programming single pages. A burst in PFlash consists of 8 pages (256 bytes) and in DFlash 4 pages (32 bytes).
- **Assembly Buffer:** A buffer for the data before it is stored to the memory. A "Load Page" command fills the buffer and a "Write Page" or "Write Burst" command flush the buffer.

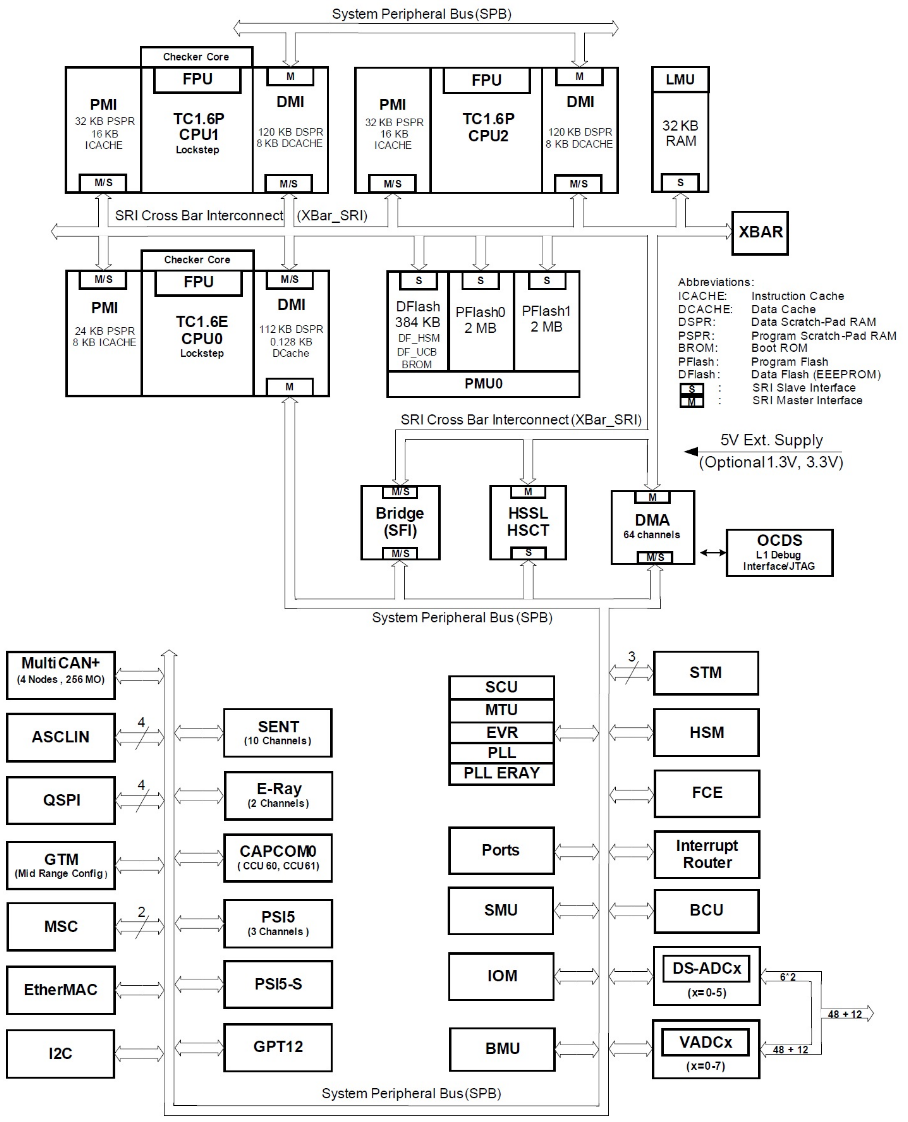


Figure 2.7: TC27x Block Diagram. (Source: Infineon TC27x Target Specification V3.0 2011-12, 2017)

The used application kit with the AURIX TC277 has the following flash structure:

- PF0 and PF1: 2 MByte each;
- DF0 consisting of:

  DF_EEPROM: 384 KByte (48 logical sectors EEPROM0 EEPROM47);

  DF_UCB: Flash area for protection data (16 logical sectors UCB0 UCB15 with 1 KByte each).
- DF1 consisting of:

  DF_HSM: 64 KByte.

To program and erase the flash memory, there are certain command sequences that have to be used. Also, there are some additional requirements when writing to program flash. In Table 2.2 the most important command sequences are summarized.

Table 2.2: Command Sequences for Flash Control.

Command sequence		1.cycle	2.cycle	3.cycle	4.cycle
Enter Page Mode	Address	.5554			
	Data	..xx5y			
Load Page	Address	.55F0			
	Data	WD			
Write Page	Address	.AA50	.AA58	.AAA8	.AAA8
	Data	PA	..xx00	..xxA0	..xxAA
Write Burst	Address	.AA50	.AA58	.AAA8	.AAA8
	Data	PA	..xx00	..xxA0	..xx7A
Clear Status	Address	.5554			
	Data	..xxFA			

The parameter *data* of the command sequences can be one of the following:

- **PA:** Absolute start address of the Flash page. Must be aligned to burst size for Write Burst or to the page size for Write Page.
- **WD:** 64-bit or 32-bit write data to be loaded into the page assembly buffer.
- **xxYY:** 8-bit write data as part of a command cycle. Only the byte YY is used for command interpretation. The higher order bytes xx are ignored.

  xx5y: Specific case for YY. The y can be 0x0 for selecting the PFlash or 0xD to select the DFlash.

Before a page can be programmed, it is very important that it is deleted. After that, the containing Flash has to be put in page mode with the command *Enter Page Mode*. Next, the *Load Page* command sequence loads the given data into the page assembly buffer.

When enough data for a page is loaded, the *Write Page* starts the programming process for a single page. To increase performance, there is also the *Write Burst* command, which starts the programming process for an aligned group of pages. Both write commands automatically reset the page mode flag.

Safety mechanisms (Endinit)

The Endinit function of the AURIX platform is a protection mechanism to prevent initialization registers from being written during normal application run. The Endinit feature consists of an ENDINIT bit incorporated in each WDT control register. Registers protected via an Endinit determine whether or not writes are enabled. There are 2 types of Endinit protections. The first is an Endinit bit for each CPU. The second one is the safety Endinit, which is very important for writing to program flash. During the command sequences for the program flash, the safety Endinit has to be disabled otherwise programming fails. Another thing to consider is that as soon an Endinit bit is set, the watchdog starts a time-out. If the bit is not set before the time-out runs out, a malfunction is assumed.

2.4 Cyclic redundancy check (CRC32)

To confirm if data is transmitted and stored correctly, cyclic redundancy checks can be used. CRCs are designed to protect against common communication errors and to provide assurance of data integrity. They are based on the remainder of a polynomial division of the data. The transmitter calculates a check sum that is sent to the receiver, which also calculates a checksum with the received data and the same polynomial. If the two checksums do not match, there was either an error during data transmission or during the transmission of the checksum. CRCs are the remainder of a polynomial division with a fixed polynomial. The selection of the polynomial used, is crucial for the error-detecting capabilities and the collision probability. A CRC is called an n-bit CRC when its result (check value) is n bits long. For an n bit check value an n+1 bit polynomial is needed. The CAN protocol defines a 15-bit CRC that is attached to every message. The generator polynomial that is used for CAN is 1100 0101 1001 1001. To protect the integrity of whole data blocks, a 32-bit CRC is used. It uses the same polynomial that is used for Ethernet, i.e., 1 0000 0100 1100 0001 0001 1101 1011 0111. The mathematical analysis of the error detection capabilities is a very complex subject and mostly focuses on random data. Partridge, Hughes and Stone [SGPH98] evaluated the performance of checksums on real data and found a significant deviation to the mathematical results for random data. For the 32-bit Ethernet CRC it is safe to say, that it detects the following errors:

- Any 1 bit error;
- Any two adjacent 1 bit errors;
- Any odd number of 1 bit errors;
- Any burst of errors with a length of 32 or less.

Figure 2.8 illustrates the calculation of a simple two bit CRC. The quotient of the division is not needed for the CRC.

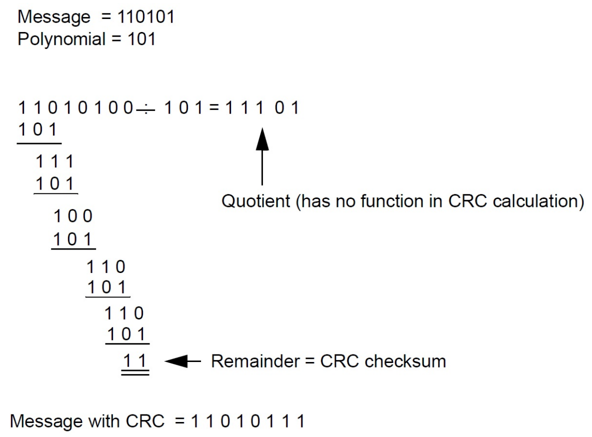


Figure 2.8: Calculation of a simple two bit CRC.

2.5 Linker Files

Linker files are used to determine the memory layout before compilation. The basic principle is to define different named sections that can be alligned as defined in the linker file. The names of those sections can then be used in the C code to define which parts of the code should be placed in which sections. For every section, the absolute start address and the size can be defined. Following are some of the most important operators and commands.

- **"ENTRY":** defines the first executable instruction in an output file;
- **"MEMORY":** describes the location and size of blocks of memory in the target;
- **Global definitions:** integer variables and symbols can be defined global in the linker file;
- **dot".":** always contains the current output location counter. Assigning a value to the . symbol will cause the location counter to be moved (never backwards). This may be used to create holes in the output section;
- **"SECTIONS":** controls exactly where input sections are placed into output sections, their order in the output file, and to which output sections they are allocated.

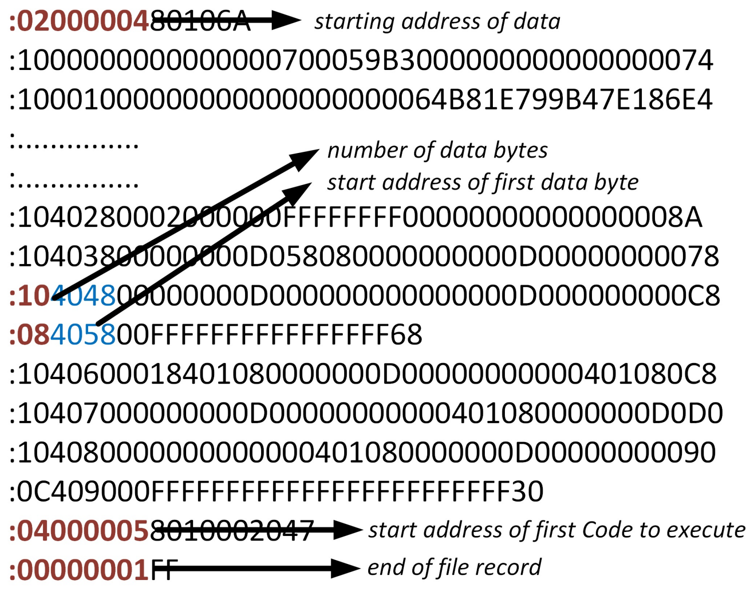
This is by far not a complete list, but should provide a very basic understanding of how linker files work and what they can be used for.

Linker files also offer the possibility to put program code, data and variables into the RAM for faster access. To accomplish that the regarding section has to be put into the copy section, i.e., during the start up procedure that section is copied from the ROM to the RAM memory. An example for a new memory section placed at a specific address can be seen in Figure 4.4 in Section 4.2.3.

2.6 Intel Hex Format

The program data, generated from the TriCore Entry Toolchain used to program the ECU, is presented as an Intel Hex File. This format was introduced in 1973 by Intel and is used ever since. This file is selected at the diagnostic test device to be uploaded to the ECU. It contains the program code of the updated SW and needs to be interpreted by the diagnostic test device. Figure 2.9 shows the most important commands that are relevant for this thesis:

- **:02000004** is followed by the first two bytes of the starting address of the data;
- **:04000005** is followed by the starting address of the first code to execute;
- **:00000001** signals the end of file;
- **:06** to **:10** at the beginning of the lines is the number of data bytes in the line;
  - The number of data bytes, is followed by the lower two bytes of the address.



```
:0200000480106A
:10000000000000000700059B3000000000000000074
:100010000000000000000000064B81E799B47E186E4
:...............
:...............
:10402800020000000FFFFFFFF0000000000000008A
:1040380000000000D058080000000000D00000000078
:104048000000000D000000000000000D000000000C8
:08405800FFFFFFFFFFFFFFFF68
:1040600018401080000000D0000000000000401080C8
:104070000000000D000000000000401080000000D0D0
:104080000000000000000401080000000D00000000090
:0C409000FFFFFFFFFFFFFFFFFFFFFFFF30
:040000058010002047
:00000001FF
```

starting address of data
number of data bytes
start address of first data byte
start address of first Code to execute
end of file record

Figure 2.9: Description of Intel Hex File.

The hex file always starts with the first two bytes of the starting address. The following lines contain the data bytes, starting with the number of data bytes in this line and, the lower two bytes of the start address of the first byte of this line. The following byte is always “0x00” for lines containing data bytes, followed by the actual data and concluded with a one byte checksum. The checksum is the two’s complement of the least significant byte of the sum of all byte values in the line, except the start symbol “:”. If at some point, the last two bytes of the start address of the first byte of a line reaches an overflow, a new line starting with “:02000004” increments the start address. Interpreting a hex file it has to be considered, that consecutive data lines are not necessarily connected in the memory.

Therefore it can be checked if the number of data bytes added to the lower two bytes of the start address result in the next lines start address. At the end of every hex file, there are two lines. Second to last, a line starting with ":04000005" and the address of the first code to be executed. The last line always signals the end of the hex file.

Chapter 3

System Architecture

This section describes the system architecture for the wireless update system. Figure 3.1 shows the different parts of the system, including the diagnostic test device, the WVI and the ECU, as well as the two data transmission paths.

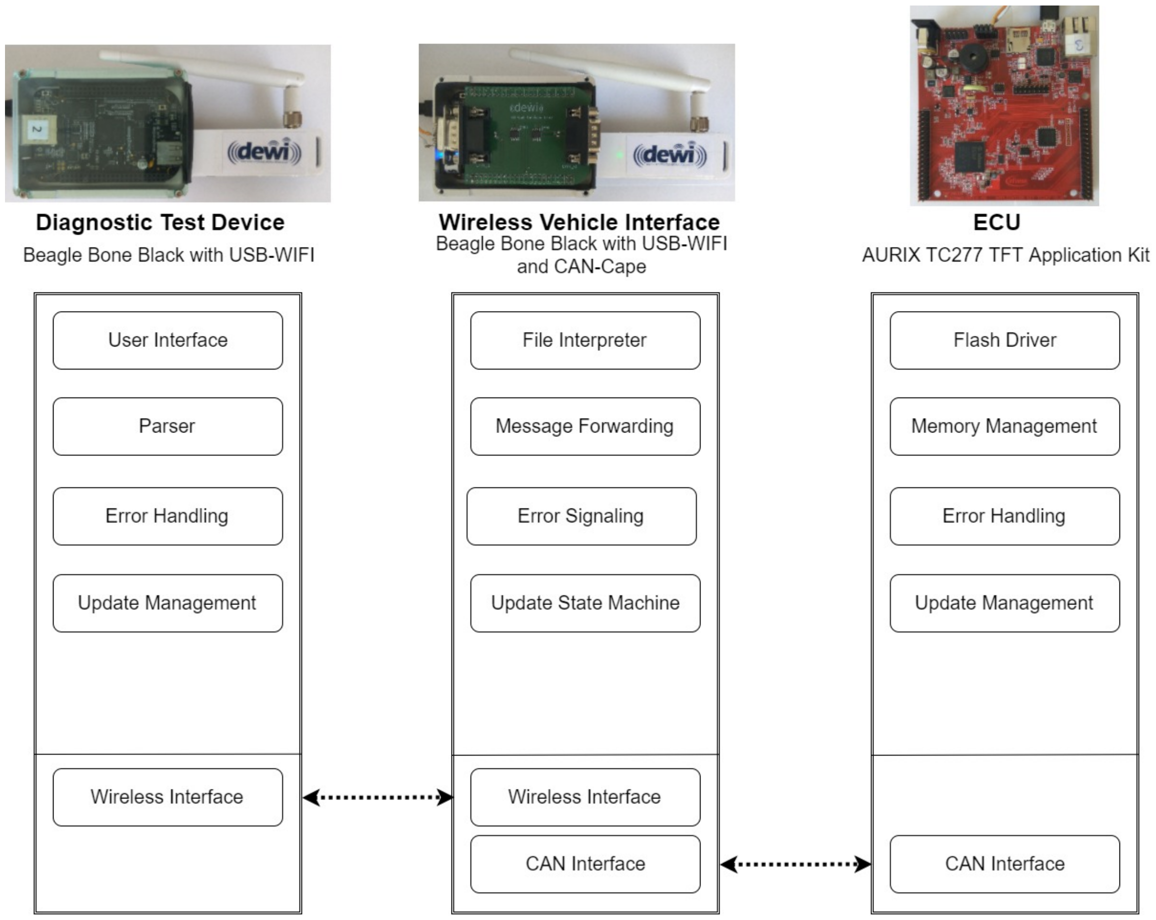


Figure 3.1: System architecture of the wireless update system.

This chapter starts with the description of the workflow for a wireless update (for more details, please refer to Section 3.1). After that it describes the different parts introduced in Figure 3.1, beginning with the two data transmission paths, the wireless interfaces (Sect. 3.2) followed by the CAN interfaces (Sect. 3.3). This chapter further describes the three hardware components and their responsibilities in the wireless update system: the diagnostic test device (Sect. 3.4), the WVI (Sect. 3.5), and the ECU (Sect. 3.6).

3.1 Workflow

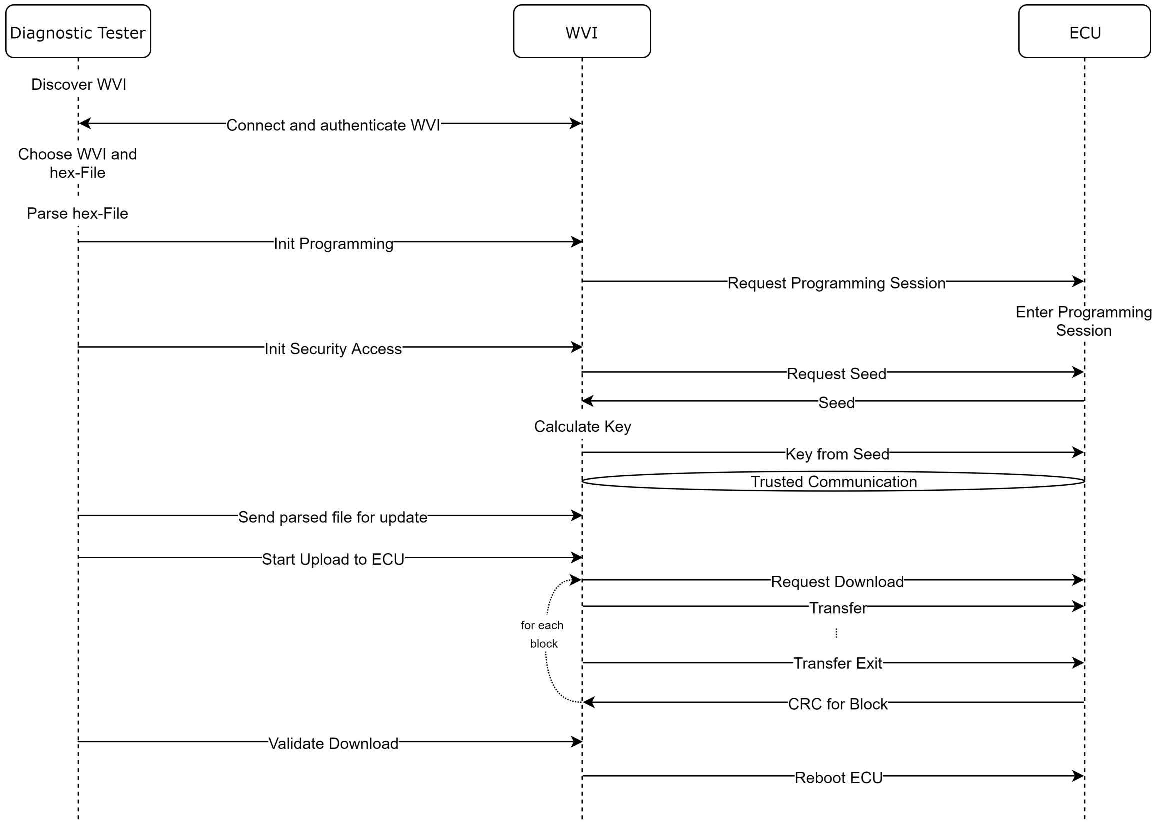


Figure 3.2: Sequence for the Update Process.

Figure 3.2 describes the overall workflow of the complete system. First, the hex file (Section 2.6) is parsed and the necessary information is being extracted. The program data is temporary copied into another file, where it is stored according to the memory sections. For the partial update, the data file is now compared to the data file of the original SW currently running on the ECU. This is done such that only the memory sections that have changed need to be transferred. Now the WVI connects to the ECU using the specific sequence defined by the UDS protocol described in detail in Section 4.6. The next step is the data transfer, whose integrity is ensured by a CRC at the end of every frame, and also at the end of every memory section. The latter is checked against the CRC calculated over the stored memory content. Those redundancy checks ensure the integrity of the transfered SW. After the transfer is complete and the data has been checked, the system has to switch from the bootloader to the newly transferred SW. This is done by storing the start address to the data flash block and rebooting the ECU.

To handle errors in a newly downloaded SW, the system offers the possibility to backup SW on the ECU and restore it, if the new SW is faulty. This is done with two newly introduced commands and described in more detail in Section 4.10.

3.2 Wireless Interface

The wireless interface connects the DT to the WVI. To support the desired functionalities the wireless interface has to meet the following requirements (detailed description in Section 2.2):

- Fast data transfer;
- Mesh networking;
- Multicast.

The fast data transfer is necessary to ensure short update times. As the data transfer is sequential, first the upload from the DT to the WVI on the wireless path, and after , the download from the WVI to the ECU, a performance benefit on the wireless path, directly contributes to a shorter overall update time. To enable a broad range of applications, the wireless path should enable mesh networking. In a future scenario, it should be possible that vehicles parked on the workshop parking slot, are connected to the workshop network. To improve the signal quality and enable connections to vehicles parked further away, a future wireless update system should provide multi-hop connections. For the parallel update, the wireless interface should provide the possibility for multicasts, to send an update to multiple recipients at the same time. Those requirements are fulfilled by the IEEE 802.11s protocol (described in Section 2.2.2). It provides all desired functions and is easy to implement. 802.11s was introduced by a study group back in 2003 and was incorporated in the 2012 release of the 802.11 specification. The amendment adds mesh networking capabilities and allows wireless devices to be connected in a way that broad- and multicasts are possible.

This enables the “workshop-scenario”, where multiple vehicles in a workshop need to be updated at the same time. In such a scenario, a mechanic can use the wireless update system to simultaneously update the SW in multiple vehicles, and the same data packets can be sent to multiple receivers at the same time. This applies when multiple vehicles needing the same update are connected to the same mesh and can therefore be updated at the same time. The time improvement depends on how many vehicles are updated at the same time which means, the more vehicles are updated, the bigger the benefit. The connection is already part of the SecUp framework [SBK$^+$16] and can be used as is.

3.3 CAN Interface

As connection between the WVI and the ECU, only automotive buses can be used. There are multiple possibilities like Controller Area Network (CAN), FlexRay, Lin and MOST.

MOST is used for multimedia systems and optimized for high data rates, but is not used for safety critical systems as it is not a deterministic bus system. LIN is the simplest bus system, which was designed to connect sensors and actors. It is mostly used for comfort systems where the higher performance of CAN is not needed. That leaves only the CAN and FlexRay bus to be used for the implemented system. Due to the reduced complexity and lower cost compared to FlexRay, CAN was selected for the connection between the WVI and the ECU. The Controller Area Network (CAN) is a deterministic and fault tolerant bus system that offers data rates up to 1 MBit on its two data lines “CAN high” and “CAN low” (for a more detailed description, please refer to Section 2.1. The communication between the WVI and the ECU complies with a redesigned “UDSonCAN” protocol. “UDSonCAN” is specified by ISO 14229 and, as a higher level protocol, describes the contents of the CAN data frame. Its redesign focused on maximizing the data transfer capabilities by increasing the number of data bytes that can be transferred with every message from 5 to 7 . Those changes required a redesign of the complete protocol, which is described in Section 4.6, to keep the functionality provided by the original protocol. The messages have to follow a defined sequence illustrated in Figure 3.2. If a message violates the sequence, a defined error message is sent.

3.4 Diagnostic Test Device

The diagnostic test device is a Beagle Bone Black (BBB) running linux equipped with a USB Wi-Fi stick. The SW running on the BBB is responsible for parsing the hex file, generating the intermediary file, containing the data split into the different blocks and, sending the data wirelessly to the WVI using the 802.11s wireless protocol described in Section 3.2. The diagnostic tester is also responsible to calculate the delta of the data files for the partial update, as described in Section 4.8. The 802.11s protocol enables the diagnostic test device to send data to multiple WVIs at the same time, which is used for the parallel update. The diagnostic test device is placed outside the vehicle, for example a tablet or a smartphone in a workshop.

3.4.1 Parser

The parser is part of the diagnostic test device SW and responsible for interpreting the Intel Hex File. The Hex File illustrated in Figure 2.9 contains a lot of added data. To reduce the transferred data, this file needs to be interpreted and the necessary information stored in a new file containing the different blocks of data from the Hex File. An illustration of the desired functionality can be seen in Figure 4.2.

3.4.2 Error Handling

The DT needs to detect and handle errors during the update procedure. The code overhead and the complexity of the error handling should be minimal. The DT should be able to detect if the sequence of commands complies with the defined sequence of the UDS protocol. It should also handle CRC errors and errors establishing the connection between the DT and the WVI. Most of the errors are detected by the ECU and forwarded to the

DT through the WVI. The handling of most of them is the responsibility of the DT. The details are listed in Section 4.2.4.

3.4.3 Update Management

The update management is mainly done by the DT. It is responsible for the following tasks:

- **Connection to the WVI**: Establish a secure connection to the WVI;
- **Upload of SW to the WVI**: Upload the parsed hex file to the WVI;
- **Choose download mode**: Signal to WVI if normal, partial or parallel update is used;
- **Initiate download to ECU**: The DT commands the WVI to start programming with the selected mode;
- **Validate update**: Check responses from ECU for CRC checks;
- **Reboot ECU**: Signal the WVI to send reboot command after update is done.

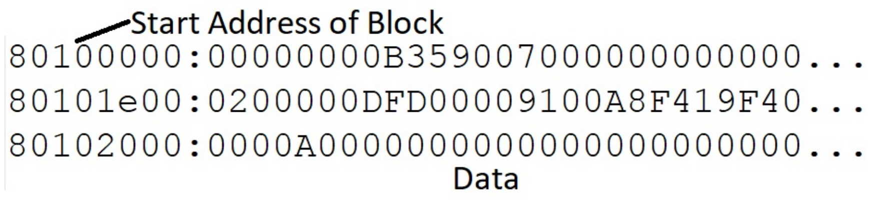
A description of a complete wireless update process, including the aforementioned tasks, can be found in Section 4.2.4.

3.5 WVI

The WVI is the gateway from the outside into the vehicle. It is connected to the CAN bus and also features a USB Wi-Fi stick capable of the 802.11s protocol. To enable CAN on the BBB it is equipped with a cape that is connected to the pin headers. The cape, as seen in Figure 2.5 was custom made to fit the BBB. The responsibility of the WVI is to receive the SW from the diagnostic tester and send it to the ECU that needs to be updated. The WVI is used as a wireless gateway to the vehicle bus. The DT manages the update by sending commands to the WVI that trigger specific update sequences. Those sequences have to be triggered in a defined order to successfully execute an update. It has to interpret the file received from the DT and send the data to the ECU. If errors occur, the ECU sends a message to the WVI that has to be forwarded to the DT. To receive status updates, the WVI also has to inform the DT on the status of the update.

3.5.1 File Interpreter

The WVI has to interpret the file received from the DT and illustrated in Figure 3.3. Every line represents a data block of the SW, starting with the start address for the following data. For the standard update, the first block contains the start up section of the uploaded SW (see Section 4.2.4 for details).



```
80100000:00000000B35900700000000000000...
80101e00:0200000DFD00009100A8F419F40...
80102000:0000A0000000000000000000000...
```

Figure 3.3: Layout of parsed file from DT.

3.5.2 Update State Machine

Contrary to the DT that actively manages the update process, the WVI is controlled by the DT to execute specific command sequences. The WVI is triggered to execute the update process once the parsed file is transferred from the DT. The sequence is the same for every block and repeated until all blocks are transferred.

If an error occurs during the update process, the ECU sends an error message to the WVI. This message has to be forwarded to the DT to signal the error. The responsibilities of the WVI during the update process are described in Section 4.2.4.

3.6 ECU

The ECU needs to handle the communication to the WVI. Therefore it has to interpret the modified UDS protocol as described in Section 4.6. It is also responsible to perform the switch to the new SW and to run the bootloader which coordinates the whole update process on the ECU.

3.6.1 Flash Driver

To store data on the ECU, a flash driver (Section 4.5) is needed to handle the communication with the memory controller. Before storing data, the flash page has to be set to *Page Mode*, which is reset after every write command. The flash memory offers two different techniques for storing data. First, the standard *Write Page* command that writes only one page (32 bytes) to the flash memory and second, the *Write Burst* command, which writes up to 256 bytes at once. The write address has to be aligned with the page borders (every 32 bytes), to enable the *Write Page* command. The *Write Burst* command is only enabled if the address is aligned to the burst size, which is every 256 bytes. The driver should automatically use the fastest method to store the data and simplify the complexity of the memory controller.

3.6.2 Memory Management

The ECU has to handle the data for the update and store all relevant information for the update process (Section 4.2.2). Due to the limited amount of flash memory and RAM, the placement of data has to be well thought. This starts by minimizing the code size and therefore memory usage of the bootloader. Furthermore the additional information that has to be stored can be divided into two types, persistent and non persistent data. The non persistent data, is information that is necessary for the update process itself, but not the uploaded SW afterwards. The main part is, storing the status of the memory sections to prevent deleting already stored data. The rest are some flags and temporary variables. Non persistent data is stored in RAM for fast and easy access. The persistent data, is information that is important for the ECU at anytime and needs to be valid even after a reset and after the update system switched to the newly uploaded SW. It consists of the following information:

- ECU identification number: uniquely identifies an ECU inside a vehicle;
- Vehicle Identification Number (VIN): uniquely identifies a vehicle;
- SW Version: the version number of the currently installed SW that is running on the ECU;
- Program Address: the address where the currently installed SW starts;
- Layout of SW: the start and end addresses of the different blocks of the currently installed SW;
- Layout of Backup SW: the start and end addresses of the different blocks of the previously backed up SW.

This information is valid not only for the bootloader, but for the uploaded SW as well. That means during development of the SW to be uploaded, it has to be prevented, that this information is overwritten by the SW. As the memory overhead should be minimized, this information has to be managed to use minimal memory resources. The persistent data can be stored in the program or the data flash. For larger data sets, the AURIX ECU also offers an SD card interface, which again needs a separate driver.

3.6.3 Error Handling

If an error occurs during the update, the ECU has to be able to detect it and handle the consequences. The following errors have to be considered:

- Single and multiple bit errors during transmission;
- Single and multiple bit errors storing the data;
- Unauthorized access to update procedure;
- Invalid commands;
- Invalid command sequences;
- Memory violations.

Single and multiple bit errors are mostly a result of so called “bit flips”, where a single or multiple bits unintentionally change their state, altering the data. Such errors can happen during data transmission or when storing the data, but have to be detected by the ECU (Section 4.4). As updating the ECUs SW is a very sensitive process, it has to be assured that only authorized systems (Section 4.3) are allowed to perform an update.

If during the initialization of the update or the update process itself, an invalid command is received, the ECU has to react accordingly, sending a predefined error message. The whole update process follows a defined sequence of commands as seen in Figure 3.2 and described in Section 4.2.4 that, if not correctly followed, leads to the termination of the update. During the update, the ECU also has to check for memory violations from the uploaded SW. It has to be guaranteed that the uploaded SW only uses memory sections that it is allowed to. In any error case, the ECU has to send defined error messages to inform the WVI, which forwards the messages to the DT.

3.6.4 Update Management

The ECU is the main part of the wireless update system and has to manage the update process. It has to implement all valid commands (see Table 4.1) and perform the corresponding tasks. It has to support parallel and partial updates, described in Sections 4.7 and 4.8, and therefore offer some identification to distinguish between multiple ECUs connected to the same WVI. Figure 3.4 illustrates the sequence of commands, that has to be sent to perform an update. Every command is acknowledged and if a command violates the defined sequence, the ECU has to decide if the update is aborted or the command, violating the sequence, is ignored.

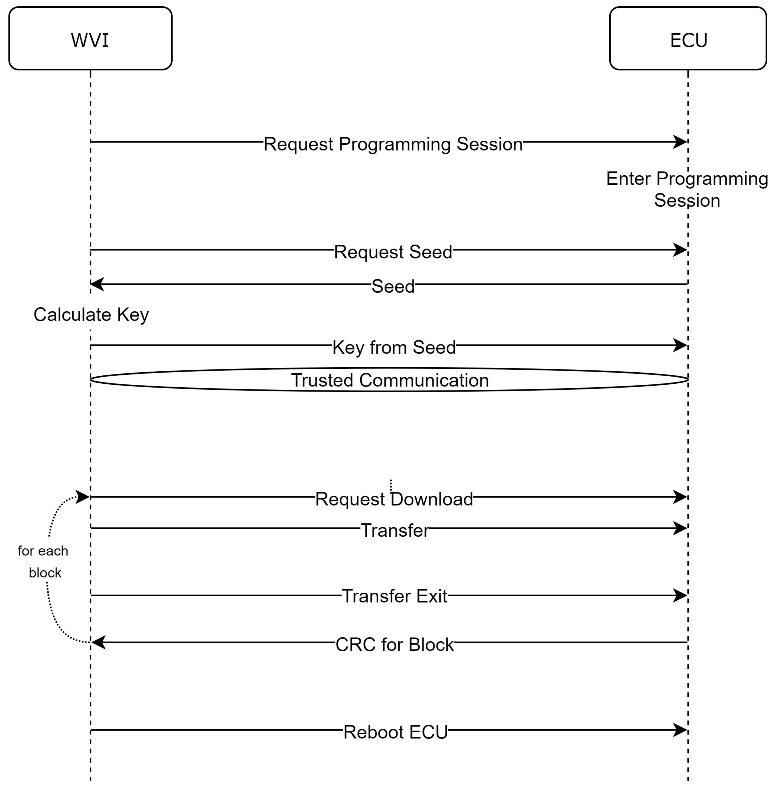


Figure 3.4: Update sequence for the ECU bootloader.

The overhead for the parallel and partial update should be minimized, so the sequence of commands is the same as for a standard wireless update described in Section 4.2.4.

Chapter 4

Design and Implementation

The following section describes the implemented parts in more detail. It begins with some general remarks on the implementation in Section 4.1 and a description of the wireless update process in Section 4.2. After that, I will describe the different contributions from Figure 1.2, starting with the implementation of the seed and key algorithm in Section 4.3 followed by the CRC implementation in Section 4.4. To maximize the performance of the wireless update system, Section 4.5 describes the AURIX architecture specific optimizations and Section 4.6 describes the changes to the UDS protocol made to maximize the data throughput during the update. The details for the parallel and partial update are described in Sections 4.7 and 4.8, respectively. Thereafter, Section 4.9 details the implementation and purpose of the diagnose function and Section (4.10) illustrates the implementation details of the backup and restore feature.

4.1 General Remarks

Before describing the details of the different parts of the system, I want to shed light on some important details that have affected the design of the system as well as the implementation process.

4.1.1 AURIX Documentation

Before starting the development of the system, extensive research into the AURIX platform was necessary. The documentation of the platform is spread through multiple files and application notes, where in some cases the files contradict each other. One of the bigger issues was the flawed documentation of the watchdog mechanism. The AURIX ECU has a safety watchdog mechanism that needs to be deactivated during programming. To do so a specific command sequence has to be transferred. In the original data sheet, this command sequence is wrong and leads to the ECU being stuck in a reset loop and shutting down after a few tries. The correct command sequence was only published in a separate application note that appeared once the system was already partially developed.

4.1.2 Memory

One of the biggest challenges during development was the instability of the flash drivers. The program flash has some specific addresses that do not behave as expected. In particular, trying to write to those addresses results in undefined memory contents. As of now, the identified memory locations are:

- 0x80110000 to 0x80110040;
- 0x80120000 to 0x80120040;
- 0x80130000 to 0x80130040.

The datasheet does not describe the mentioned memory sections as special, but that behavior is still under investigation. To avoid this complications, the SW to be updated should be situated in other memory areas than the ones mentioned above.

4.1.3 Reset Behavior

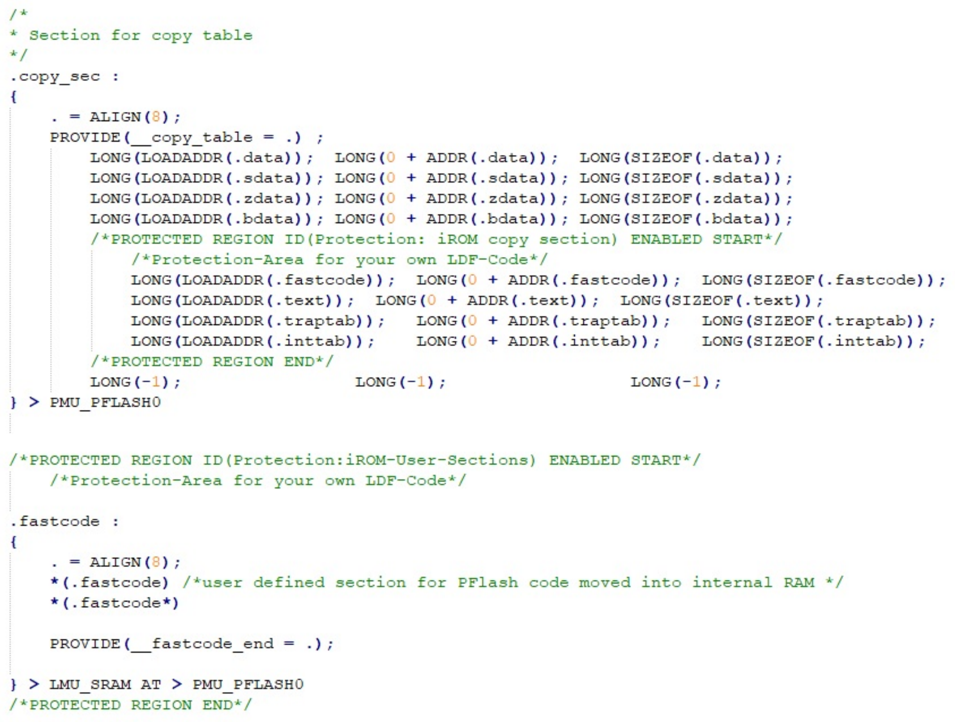
The next problem I have been facing during the design and implementation of the system is the behavior of the reset. There are two different reset mechanisms that can be triggered from SW. The first is an application reset, that leads to a defined state of the application system by re-initializing all peripherals, the CPU and parts of the system control unit (SCU). The second is the system reset, which affects the complete system without a reset of the power subsystem and with no effect on the reset configuration.

After an update, the ECU receives a message triggering a SW reset. After the reset, the start up code looks for the start address of the uploaded SW and jumps directly to it. This leads to the immediate execution of the uploaded SW after the reset. Now there are two options to return to the bootloader. The first is to send a command to the ECU that clears the starting address and another command to execute the reset. The second is the use of a pin that is checked during the start up section of the bootloader before the start address of the SW is checked. If that pin is set after a reset, the bootloader ignores the uploaded SW and executes the bootloader.

A problem with that procedure is that the switch from the uploaded SW back to the bootloader did not work as desired. After the reset, the system was stuck in a reset loop and had to be reset manually again to resume its normal operation.

4.1.4 Writing to the Program Flash (PFlash)

During the execution of a write command, the flash memory in question must not be accessed. This includes the execution of code from the program flash while it is written to. Therefore the flash driver has to be placed in the RAM. Figure 4.1 shows the changes to the linker file to add a new memory section called .fastcode that is placed in the RAM and is therefore able to host the flash driver for the program flash. That means all of the code that is used to program the memory, has to be put into this section. . This section is placed at "> LMU_SRAM AT > PMU_PFLASH0", which means that the original code is in the program flash but is copied to the RAM. After that the used section also has to be placed in the so called "copy table". This results in the PFlash driver being copied to the RAM during the start up procedure, before the actual bootloader code starts.For simplification the complete flash driver is put in the RAM. For platforms with very little RAM, only the critical program flash driver parts can be placed in the RAM.



```
/*
* Section for copy table
*/
.copy_sec :
{
    . = ALIGN(8);
    PROVIDE(__copy_table = .) ;
        LONG(LOADADDR(.data));  LONG(0 + ADDR(.data));  LONG(SIZEOF(.data));
        LONG(LOADADDR(.sdata)); LONG(0 + ADDR(.sdata)); LONG(SIZEOF(.sdata));
        LONG(LOADADDR(.zdata)); LONG(0 + ADDR(.zdata)); LONG(SIZEOF(.zdata));
        LONG(LOADADDR(.bdata)); LONG(0 + ADDR(.bdata)); LONG(SIZEOF(.bdata));
        /*PROTECTED REGION ID(Protection: iROM copy section) ENABLED START*/
            /*Protection-Area for your own LDF-Code*/
            LONG(LOADADDR(.fastcode));  LONG(0 + ADDR(.fastcode));  LONG(SIZEOF(.fastcode));
            LONG(LOADADDR(.text));  LONG(0 + ADDR(.text));  LONG(SIZEOF(.text));
            LONG(LOADADDR(.traptab));   LONG(0 + ADDR(.traptab));   LONG(SIZEOF(.traptab));
            LONG(LOADADDR(.inttab));    LONG(0 + ADDR(.inttab));    LONG(SIZEOF(.inttab));
        /*PROTECTED REGION END*/
        LONG(-1);                LONG(-1);                  LONG(-1);
} > PMU_PFLASH0

/*PROTECTED REGION ID(Protection:iROM-User-Sections) ENABLED START*/
    /*Protection-Area for your own LDF-Code*/

.fastcode :
{
    . = ALIGN(8);
    *(.fastcode) /*user defined section for PFlash code moved into internal RAM */
    *(.fastcode*)

    PROVIDE(__fastcode_end = .);

} > LMU_SRAM AT > PMU_PFLASH0
/*PROTECTED REGION END*/
```

Figure 4.1: Linker file changes for memory section in RAM.

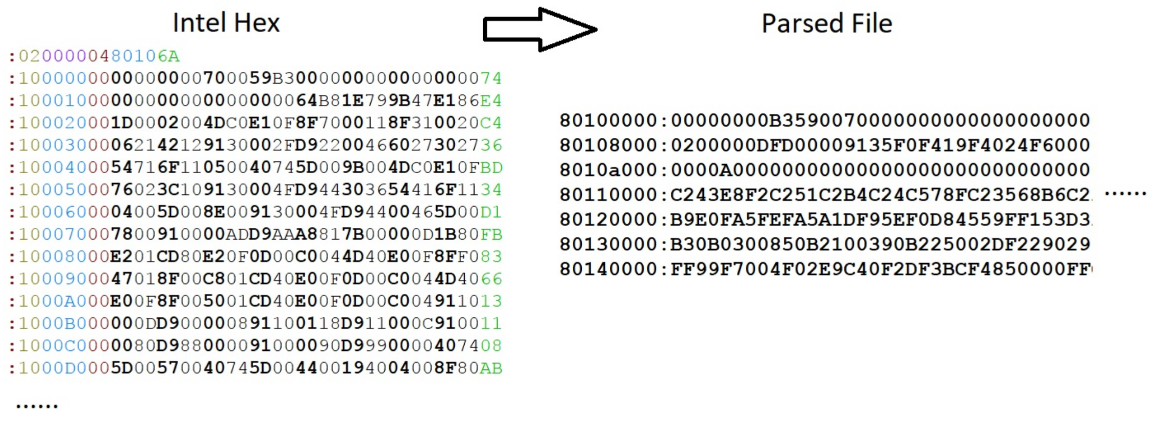
4.2 Standard Wireless Update

The basic principle for the wireless update as seen in Figure 3.2 consists of multiple SW parts in the different hardware components (DT, WVI and ECU). This section describes the different parts of the wireless update system that where not already mentioned, and tries to describe how all of them work together to perform an efficient, secure, and reliable update.

Before the wireless update process is described in Section 4.2.4, I introduce the parser for the hex file in Section 4.2.1 which is responsible to extract the data necessary for the update. To understand what information is necessary on the ECU, Section 4.2.2 describes the implemented data structures for the persistent and non-persistent data storage, about the update process itself as well as the updated SW. Section 4.2.3 illustrates the necessary changes in the linker files of the bootloader itself, as well as the SW to be uploaded. After the update process is completed, a switch to the uploaded SW is only possible thanks to a modification of the bootloader's boot code, described in Section 4.2.5.

4.2.1 Parser

The Intel Hex File described in Section 2.6 contains a lot of extra data that is not needed for the update. Therefore, to minimize the transferred data, the parser should extract only the necessary data and store it into a new file as seen in Figure 4.2. The hex file is parsed line by line. If the line starts with ":02000004", the parser stores the following two bytes as the first two bytes of the address of the data. For data lines after that, beginning with ":06" to ":10", the lower two bytes of the start address are combined with the higher two bytes stored previously. If the first two bytes of the start address are changing in the middle of the hex file, another line starting with ":02000004" indicates that. The parser constantly checks for jumps in the addresses. This can happen if small memory sections are not completely filled, resulting in small jumps in the start address. This is not indicated inside the hex file and needs to be recognized by the parser. The resulting file is built in a way that every line starts with the address of the first byte in the row, and is followed by the continuous data block starting at that address. The parser also considers the memory structure of the ECU, such that if a continuous data block reaches over multiple memory sections, it is split up to fit the memory sections. The reason for that, is described in Section 4.2.2. At the end of the hex file, the line starting with ":04000005" contains the 4 byte address for the entry point of the SW. This is the address the bootloader has to jump to, in order to execute the SW. This address is usually 0x20 added to one of the starting addresses of the blocks in the parsed file. The parser identifies the line that contains the start up procedure and moves it to the top.



```
:0200000480106A
:10000000000000000700059B30000000000000000074
:10001000000000000000000064B81E799B47E186E4
:100020001D0002004DC0E10F8F7000118F310020C4
:10003000062142129130002FD92200466027302736
:1000400054716F11050040745D009B004DC0E10FBD
:1000500076023C109130004FD944303654416F1134
:1000600004005D008E009130004FD94400465D00D1
:100070007800910000ADD9AAA8817B00000D1B80FB
:10008000E201CD80E20F0D00C0044D40E00F8FF083
:1000900047018F00C801CD40E00F0D00C0044D4066
:1000A000E00F8F005001CD40E00F0D00C004911013
:1000B000000DD900000891100118D911000C910011
:1000C0000080D988000091000090D9990000407408
:1000D0005D00570040745D004400194004008F80AB
......
```

```
80100000:00000000B359007000000000000000000
80108000:0200000DFD00009135F0F419F4024F6000
8010a000:0000A00000000000000000000000000000
80110000:C243E8F2C251C2B4C24C578FC23568B6C2 ......
80120000:B9E0FA5FEFA5A1DF95EF0D84559FF153D3
80130000:B30B0300850B2100390B225002DF229029
80140000:FF99F7004F02E9C40F2DF3BCF4850000FF
```

Figure 4.2: Intel Hex File and parsed data File with reduced size.

4.2.2 Memory Management

There are two different types of information that needs to be stored. The first is the data that needs to be available during the update and can be discarded afterwards. Such information can be stored in a non persistent memory like the RAM and is lost after a reset of the system, which is done immediately after the update is finished. The second type is information that is necessary for the next update and needs to be available after a reset. The only available persistent memories are the program and the data flash.

DFlash - persistent data

To store all necessary variables, a section in the DFlash is reserved. The structure of this segment is shown in Figure 4.3. It contains all information necessary to uniquely identify the ECU and the SW running on it. When a back up is executed, that data has to be copied so that it is not overwritten by the next update. The backed up block is using an offset of 0x1000, so that the address for the data block of the backup is 0xaf05f000.

DFlash Address	Byte 0	Byte 1	Byte 2	Byte 3
0xaf05e000	ECU ID	Number of Blocks	VIN	
0xaf05e010	SW Version		Backup Prog. Address	Prog. Address
0xaf05e020	Start Address Block 1	End Address Block 1	Start Address Block 2	End Address Block 2
0xaf05e030	Start Address Block 3	End Address Block 3	...	

Figure 4.3: Structure of the DFlash segment.

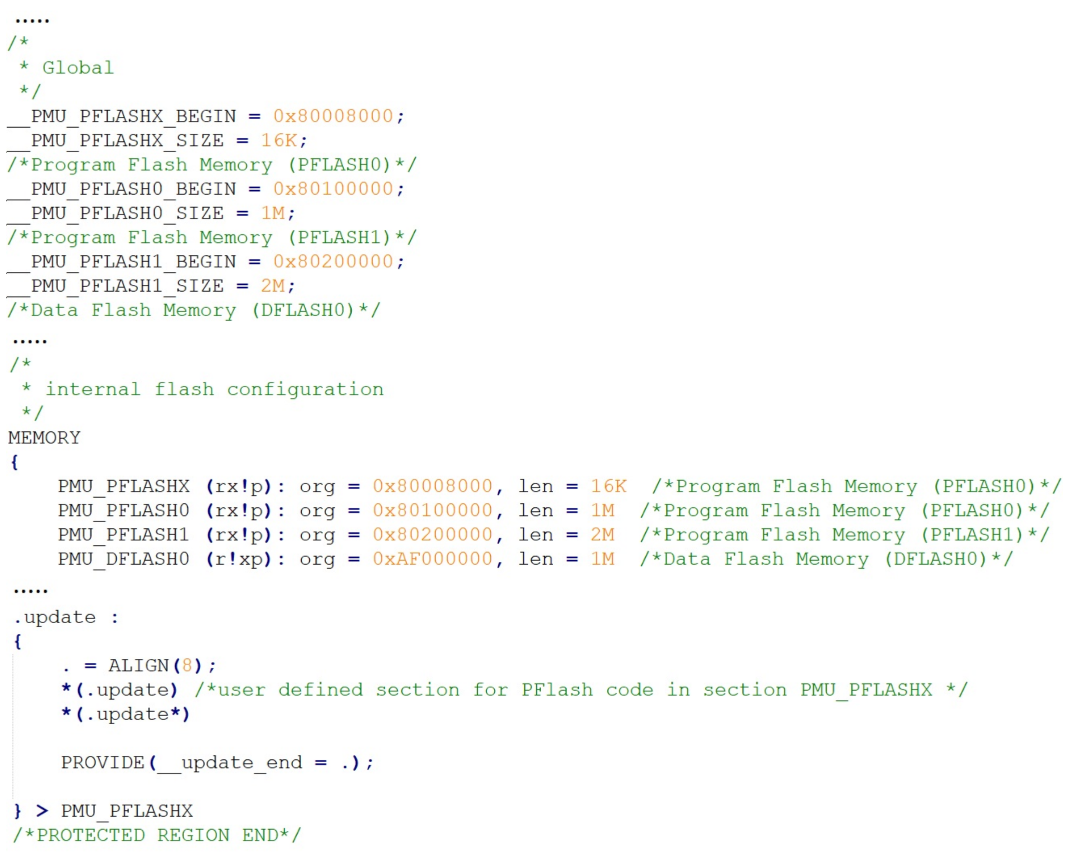
RAM - non persistent data

The only non persistent data that is needed (apart from some flags), is a vector that stores which memory sectors have been deleted. It is a vector with 27 entries that, for every sector contains "0", if the sector has not been deleted or "1", if it has been. This is needed, as it is possible that multiple blocks of the SW to be uploaded are situated in the same memory sector. If a new block is transferred, the ECU always checks if the corresponding memory sector has been deleted and if not, deletes it. This is necessary, because a memory sector can only be written to, if it has been deleted before. A problem is, that it is not possible to delete just parts of a sector but only the complete sector. For the case mentioned above, the system keeps track of the deleted sectors and does not delete them again, possibly deleting previously transferred data, if the next block is in the same memory sector as the previous one. This check is done for every "Request Download" command (see Section 4.6), which means if a block stretches over multiple memory sectors and is not split up in multiple blocks in the parsed data file, the following memory sector is not being deleted and therefore the write command fails.

4.2.3 Linker File

The linker file is used for two reasons. The first is to put the PFlash driver into the Random Access Memory (RAM) as described in Section 4.1.4. This is necessary, because during the programming of the PFlash, the latter must not be accessed.

The second use of the linker file is necessary to prevent the uploaded SW from using the same memory sections as the bootloader. Therefore, a linker file is used to reduce the available memory. To optimize the system for partial updates, the SW to be uploaded can be split into two parts. One part contains all the SW parts that are likely to be updated. The other one contains all data that will most likely stay the same. It is also possible to have multiple sections likely to be updated in different sectors of the PFlash. With the partial update enabled, if one of those SW parts changes, only the concerning section needs to be updated. An example for such a section can be seen in Figure 4.4 where a new memory region, called "PMU_PFLASHX", is defined. It has its start address at the beginning of a smaller memory sector and its size is limited to that memory sector. Now, all data that is likely to be updated can be put in a new section ".update", which is placed in the previously defined memory region. The rest of the code is placed in the scaled-down "PMU_PFLASH0". It should be considered that, if multiple smaller sections are used, those sections should not adjoin each other, or span over multiple memory sectors. If this would be the case, a change in one of the smaller sections could potentially result in a shift in the following sections. This would mean, that all shifted sections require an update, even if the data itself has not changed, eradicating the effect of dedicated update sections.



```
.....
/*
 * Global
 */
__PMU_PFLASHX_BEGIN = 0x80008000;
__PMU_PFLASHX_SIZE = 16K;
/*Program Flash Memory (PFLASH0)*/
__PMU_PFLASH0_BEGIN = 0x80100000;
__PMU_PFLASH0_SIZE = 1M;
/*Program Flash Memory (PFLASH1)*/
__PMU_PFLASH1_BEGIN = 0x80200000;
__PMU_PFLASH1_SIZE = 2M;
/*Data Flash Memory (DFLASH0)*/
.....
/*
 * internal flash configuration
 */
MEMORY
{
    PMU_PFLASHX (rx!p): org = 0x80008000, len = 16K  /*Program Flash Memory (PFLASH0)*/
    PMU_PFLASH0 (rx!p): org = 0x80100000, len = 1M  /*Program Flash Memory (PFLASH0)*/
    PMU_PFLASH1 (rx!p): org = 0x80200000, len = 2M  /*Program Flash Memory (PFLASH1)*/
    PMU_DFLASH0 (r!xp): org = 0xAF000000, len = 1M  /*Data Flash Memory (DFLASH0)*/
.....
.update :
{
    . = ALIGN(8);
    *(.update) /*user defined section for PFlash code in section PMU_PFLASHX */
    *(.update*)

    PROVIDE(__update_end = .);

} > PMU_PFLASHX
/*PROTECTED REGION END*/
```

Figure 4.4: Changes to linker file of SW to be uploaded.

4.2.4 Update Process

Figure 4.5 illustrates the update process on the ECU for a wireless update. The first step in the update process is the authentication of the DT to the WVI, which is part of the SecUp framework [SBK$^+$16]. After the DT is authenticated and connected to the WVI, the DT parses (Section 4.2.1) the selected hex-file containing the program code for the update, producing the parsed data file as seen in Figure 4.2. Now the DT initiates the WVI to send the "Request Programming Session" command to the ECU (processing of the command in the Appendix, Listing B.1) which resets the "MemoryDeletionMap", a

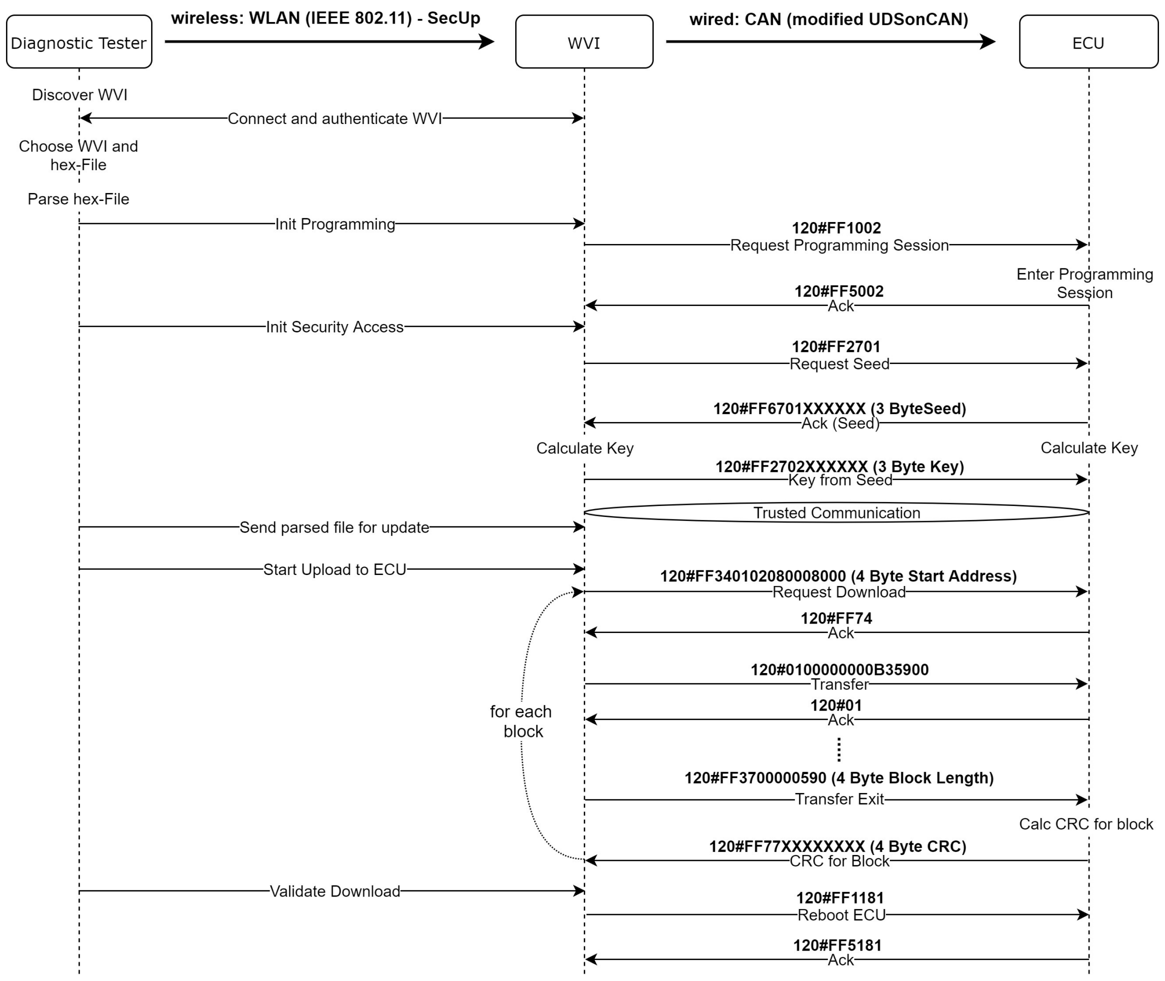


Figure 4.5: Example Update Process for a standard wireless update with first code block containing startup sequence at address 0x80008000.

vector storing the already deleted memory sectors, and sets the ECUs internal state to “Programming Session”. Now, before an update can be started, the DT has to request the seed and calculate the correct key to gain permission. Therefore the DT sends the “Request Seed” command which is acknowledged by the ECU with the seed. Now the DT as well as the ECU calculate the key with the secret algorithm. When finished, the DT sends the key to the ECU with the “Key from Seed” command. The ECU now checks if the received key matches the one calculated by the ECU and if that is the case, it sets an internal flag (“TRUSTED”) that signals that the programming session is initiated by a trusted device. Now that the “Trusted Communication” link is established, the parsed file is transferred from the DT to the WVI. After the transfer is finished, the DT initiates the download from the WVI to the ECU.Only if all previous steps encountered no errors, an update is possible. The WVI now starts to interpret the file received from the DT and sends the following commands for every block (every line in the parsed file):

- **Request Download:** Setting the start address for the following data;
  - **Start Block Byte:** Signaling the type of the following data block. “0x01” if the block contains the start up sequence, “0x02” otherwise;
  - **Offset Byte:** *Only relevant for block containing the start up sequence.* Is added to the “Start Address” to get the starting point for the uploaded SW (typically 0x20);
  - **Start Address:** Four byte start address of the following data block;
- **Transfer:** Sending 7 bytes of data with a one byte sequence number;
- **Transfer Exit:** Signaling the end of the current block;
  - **Block Length:** Four byte block length.

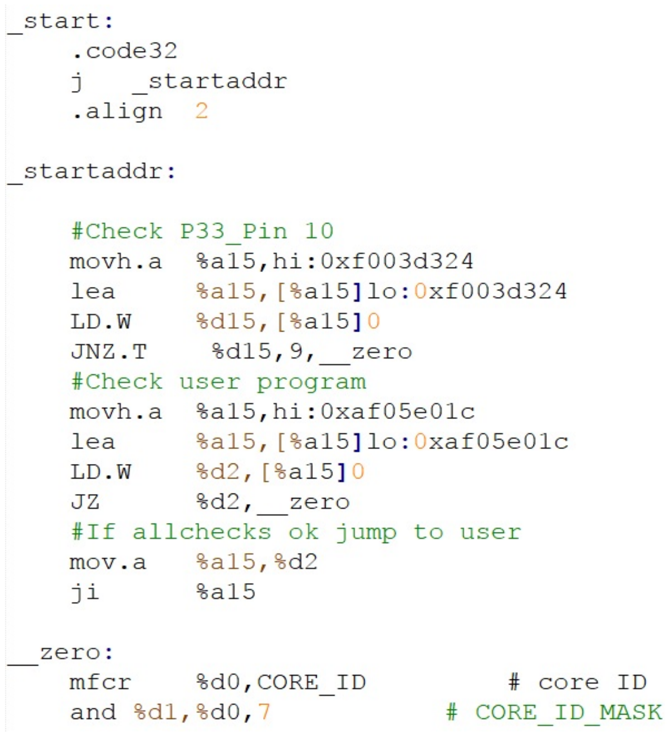
The “Request Download” command also contains the type of the following block in the “Start Block Byte”. If the block contains the start up procedure for the uploaded SW indicated by “0x01” as the third data byte, as in the example above, the ECU stores the address and the offset as the entry point for the uploaded SW. This entry point is stored to the “DFlash Block” (Section 4.2.2) before the reboot and is the address of the first command that is executed after the ECU restarts. If the next block does not contain the start up procedure, indicated by “0x02” as the third data byte, the entry point does not change. Independent of the type of the block, the ECU sets the address in the flash driver for the next data block. Now that the start address for the next data block is set, the WVI sends all the data of the block using the “Transfer” command, capable of transporting 7 data bytes. The “Transfer” command includes a one byte sequence number that ensures the correct sequence of the data packets. The sequence number starts with 0x01 for every new block. 0xFE is the highest valid sequence number, as 0xFF is reserved for commands, and 0x00 for future extensions. The code segment in the Appendix, Listing B.3, illustrates the handling of the sequence numbers of the data packages. The ECU does not return an error if it receives the same packet multiple times. If one or more package is lost, the ECU calculates the number of missed packages and cancels the update procedure. As long as all packages arive in order, the ECU always waits for 8 bytes before it uses the “Load Page” command to write the data into the assembly buffer. As described in Section 4.5, the ECU picks the best time to flush the buffer and store the data, to maximize performance (example of the “Load Page” function illustrated in the Appendix, Listing B.5). The “Load Page” function is called directly out of the CAN handler and returns “1”, if the data in the assembly buffer should be flushed. If that is the case, the ECU disables the interrupts and flushes the buffer using the “Write Buffer” function, as illustrated in the Appendix, Listing B.6. The function automatically uses the fastest way to store the data. When the block is transferred completely, the “Transfer Exit” command is sent by the WVI to signal the end of a data block. The command also contains the size of the previously transferred data block. If any data is still in the assembly buffer, the ECU flushes the buffer and stores all remaining data before calculating the CRC for the transferred block. The calculated CRC is transferred to the WVI in the acknowledgment message for the “Transfer Exit” command. Now the WVI interprets the next line in the parsed file and transfers the next data block. This cycle is repeated until all blocks are transferred. If all blocks were transferred correctly, the DT validates the download and the WVI sends the “Reboot ECU” command, which initiates the ECU to store the entry point for the SW in the DFlash block (Section 4.2.2) and reboot. After the reboot, due to the modified start up code (Section 4.2.5), the bootloader automatically executes the uploaded SW.

If an error occurs during the update process, the procedure is simply repeated or the data block re transmitted and after multiple failed attempts, the update process is aborted.

4.2.5 Modified Boot Code

As described earlier, to handle multiple SW parts in one memory, the system needs to be able to execute any of them depending on the current state of the system. To achieve that, I modified the start up code of the bootloader. It has to be the start up code of the bootloader, as it is placed at the beginning of the Program Flash and is automatically executed after reset.

When a SW is uploaded, the start address is stored to the Data Flash at address 0xAF05E01C. After a reset the start up code checks that address. If the address contains zero, the ECU starts the bootloader SW. If the address contains the start address of a previously uploaded SW, the ECU executes this program. Based on a problem with the reset, I introduced P33 Pin 10 to be used to force the bootloader to start even if a program was uploaded to the ECU. Figure 4.6 shows the changes implemented in the start up code of the bootloader.



```
_start:
    .code32
    j    _startaddr
    .align  2

_startaddr:

    #Check P33_Pin 10
    movh.a   %a15,hi:0xf003d324
    lea      %a15,[%a15]lo:0xf003d324
    LD.W     %d15,[%a15]0
    JNZ.T     %d15,9,__zero
    #Check user program
    movh.a   %a15,hi:0xaf05e01c
    lea      %a15,[%a15]lo:0xaf05e01c
    LD.W     %d2,[%a15]0
    JZ       %d2,__zero
    #If allchecks ok jump to user
    mov.a    %a15,%d2
    ji       %a15

__zero:
    mfcr     %d0,CORE_ID          # core ID
    and %d1,%d0,7               # CORE_ID_MASK
```

Figure 4.6: Changes to the start up code of the bootloader.

4.3 Seed and Key

To comply with the UDS procedure and to strengthen security, I implemented a "Seed and Key" algorithm. The algorithm uses bit masks and bit shifting to calculate the key from the seed. An existing implementation from an automotive OEM was translated from Java to C for the AURIX ECU. To protect the wireless update system, the algorithm to calculate the key from the seed should be kept a secret as it allows a system to access the update system. The details of the algorithm can not be disclosed, as the automotive OEM does not allow distributing the algorithm. Basically any typical seed and key algorithm can be applied.

The basic sequence is that the diagnostic tester sends the "Request Seed" command to the ECU which returns the Seed, a three byte random number. The ECU and the diagnostic tester are calculating the key based on the seed with a defined algorithm. First the seed is expanded to eight byte by extending it with a five byte security constant. The resulting eight byte value is shifted and combined with various constants multiple times. Now that the key is calculated, the diagnostic tester sends the key to the ECU using the "Key from Seed" command. If the received key matches the calculated one, the ECU knows it is communicating with a trusted diagnostic tester. This is necessary, because only a trusted device can be allowed to perform an update of the ECUs SW.

4.4 CRC

The system uses two different CRC algorithms. A 15 bit CRC for every CAN frame as described in the CAN specification [NXP13]. It is included in every CAN frame and ensures the integrity of the data transmission between the WVI and the ECU. Using a standard CAN transceiver, the CRC for the CAN messages is calculated and transmitted by the transceiver. If the CRC does not match the transmitted data, the sender automatically re-transmits the data, until it is either received correctly or the transmission is aborted due to an error on the bus or the transceivers.

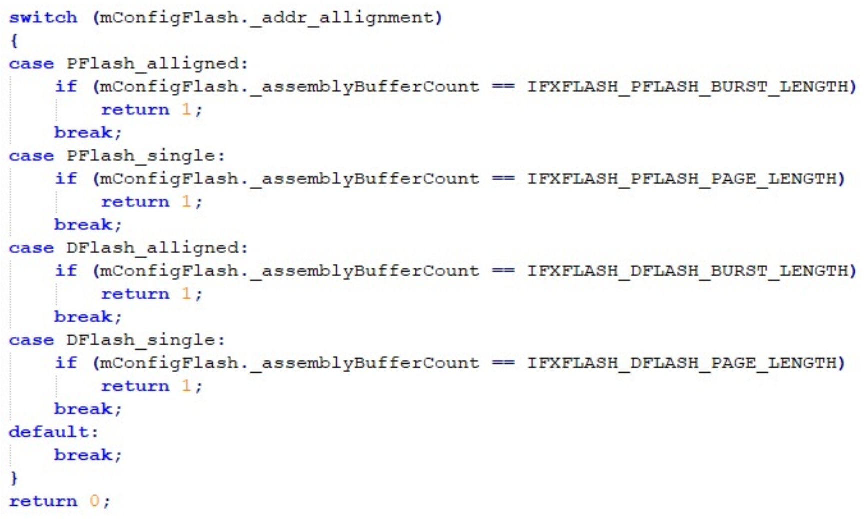
When the data is received on the ECU, it is analyzed and if it contains data of the uploaded SW, it is stored in the program flash. During this process the data is transmitted on ECU internal buses to the program flash. To ensure the integrity of the transferred SW, I use a standard Ethernet 32 bit CRC library as described in Section 2.4. This CRC is calculated after the "Transfer Exit" command is received. The command contains the length of the last block that was received. The ECU calculates the CRC32 over the complete data block and sends the result to the WVI in the acknowledgment. The CRC is calculated over the previously transferred block, where the start address is stored from the "Request Download" command, and the end address is the current flash address after the last write command, which should be the same as the start address added with the block length. If the DT detects that the CRC does not match, the whole block is transferred again.

4.5 AURIX Architecture specific Optimization

To maximize the performance and the usability for the programmer, as well as the user of the wireless update system, the bootloader SW and the developed flash driver should automatically utilize the architecture specific functions of the AURIX ECU. In the remainder of this section, we describe the implemented flash driver and the optimizations for the AURIX ECU.

Writing to the flash memory of the ECU can be done in two ways. First, a standard "Write Page" command, which stores 32 bytes of data from the assembly buffer and can be used if the current flash address is aligned with the page borders (every 32 bytes). Second, a "Write Burst" command, which stores 256 bytes at once, but it can only be used if the current flash address is aligned with the flash burst size (every 256 bytes). The flash driver should automatically choose the write mode to optimize the performance.

To achieve that, the "Load Page" command returns if the data from the write buffer is ready to be written to the memory. Depending on the alignment of the current flash address, the return value indicates if data should be written. Figure 4.7 shows part of the "Load Page" function that automatically checks the alignment of the current flash address and the amount of data in the assembly buffer and returns if it is necessary to flush the buffer and store the data. During the write commands, all interrupts are disabled, such that nothing except the flash driver has access to the program flash.



```
switch (mConfigFlash._addr_allignment)
{
case PFlash_alligned:
    if (mConfigFlash._assemblyBufferCount == IFXFLASH_PFLASH_BURST_LENGTH)
        return 1;
    break;
case PFlash_single:
    if (mConfigFlash._assemblyBufferCount == IFXFLASH_PFLASH_PAGE_LENGTH)
        return 1;
    break;
case DFlash_alligned:
    if (mConfigFlash._assemblyBufferCount == IFXFLASH_DFLASH_BURST_LENGTH)
        return 1;
    break;
case DFlash_single:
    if (mConfigFlash._assemblyBufferCount == IFXFLASH_DFLASH_PAGE_LENGTH)
        return 1;
    break;
default:
    break;
}
return 0;
```

Figure 4.7: Flash driver automatically chooses best way to write data.

To reduce the complexity of the driver, the two write commands where combined, such that the only remaining write function (WriteBuffer()) automatically chooses either to use the "Write Page" or the "Write Burst" command. The function is illustrated in the Appendix, Listing B.6 .Before data can be written to a flash page, the page has to be put in "Page Mode" with the "Enter Page Mode" command. This command is automatically transmitted before the write command if the page in question is not already in "Page Mode". After the write command, the flash driver waits until the storage process is completed and returns afterwards.

4.6 Modified UDS Protocol

For the data transfer I have one assumption, that proved correct during the tests, that nearly all of the messages are data messages of the transferred SW. This assumption lead to one main goal for redesigning the protocol. Optimize the amount of data that can be transferred in one message.

Before the implementation I analyzed the UDSonCAN specification and its message structure, especially the data transfer message. In the original protocol the data transfer message is structured as seen in Figure 2.2, which means that without any added data, only 7 bytes are available for the data transfer. Another byte is lost due to the fact that some kind of sequential ID is needed to keep the messages in order and, to be able to recognize missing packets. It is also very common to use 16 bit sequential IDs which need two bytes. That means that only 5 byte of data can be sent with every message. After carefull consideration I completely redesigned the protocol to increase the data transfer performance.

To achieve this goal I used the first byte of the data transfer message as a one byte sequential ID with values between 0x01 and 0xFE. This improves the data transfer capabilities to 7 bytes per message, which is an increase of 40%. This improvement made it necessary to change the structure of all remaining services. The services now all start with 0xFF as the first byte, moving the service identifier to the second and the sub function level to the third byte. The number of possible parameters is reduced to five. To provide the same functionality as defined in UDSonCAN, some requests need to be split into two messages. Figure 4.8 shows the modified structure of the UDS commands optimized for data transfer performance.

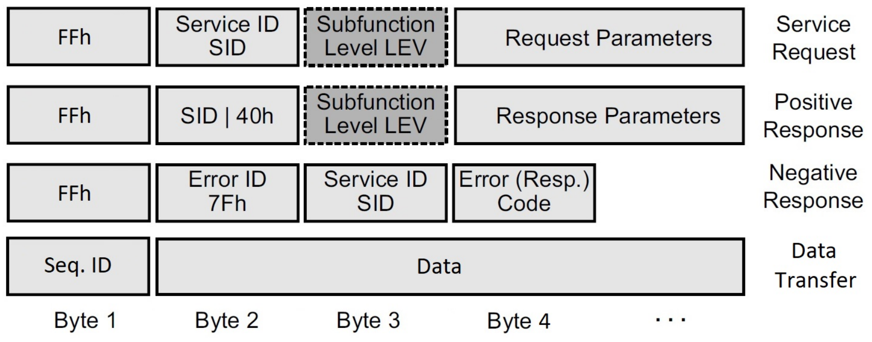


Figure 4.8: Structure of modified UDS frames (CAN data frame).

I also added some commands, including the OBD command to get the VIN, to better fit the automotive environment, where the system will be used. Table 4.1 shows all valid commands for the bootloader. The ID of the CAN commands is always 0x120 with the configured ECU ID added to it. The only exception is the OBD command to get the VIN where the CAN ID is defined by the OBD protocol, and has to be 0x7DF. The “Get Vehicle Information Number” command is inherited from the OBD protocol. It is used to get the Vehicle Information Number (VIN) which is a unique number identifying the exact make and model of the car including a country code, the type of the vehicle, a model year, a plant code and a unique production number. To determine which update is needed I need to get this unique identifier. The mentioned command provides a mapping from the ECU ID to the VIN. The following six commands are used to configure and read the ECU ID, the VIN as well as the currently installed SW version. "Request Programming Session" is used to change the state of the ECU to the UDS programming session. Without this command programming of the ECU is disabled. To prevent that anyone can access this session, the following two commands are used. "Request Seed" is sent from the diagnostic test device to the ECU and returns the seed, which is used to calculate a key. The key is sent to the ECU with the "Key from Seed" command.

For the case of a parallel update, where multiple ECUs are connected to one WVI, the WVI has to select the relevant ECUs by sending the "Activate parallel Programming" command, which sets an internal flag and adds a CAN listener to messages with a specific ID only used for parallel updates. The ECU still acknowledges the messages with their own ECU ID. The "Deactivate parallel Programming" command resets the flag and removes the CAN listener for messages with the "parallel ID". "Request Download" initiates the actual update process by sending the four byte start address of the following block. The offset byte is used to determine the starting point for the execution. The start block byte identifies the type of the following block. 0x01 is used for the block where the execution should start. 0x02 is used for all other blocks.

Now that the address is set, the "Transfer" command uses a one byte sequence number that starts with 0x01 ranging to 0xFE. The remaining seven bytes are used for the program data. When the data block is transferred completely, the "Transfer Exit" command ends the block and sends the length of the transferred data. The ECU then calculates the CRC value over the transferred block and sends it in the response. After the last block is transferred, the "Reboot ECU" command triggers a reset and after that, the ECU automatically starts the previously uploaded SW.

The "Backup SW" and "Restore SW" command are described in detail in the corresponding section 4.10.

Table 4.1: List of commands for the bootloader.

Command Description	Byte 0	Byte 1	Byte 2	Byte 3	...
Get Vehicle Identification Nr. (OBD)	0x7A	0x60	0x18	0xC2	
Set ECU ID	0xFF	0xFF	0x01	ECU ID	
Get ECU ID	0xFF	0xFF	0x21		
Set VIN	0xFF	0xFF	0x02	VIN (4 Byte)	
Get VIN	0xFF	0xFF	0x22		
Set SW Version	0xFF	0xFF	0x03	SW Ver (4 Byte)	
Get SW Version	0xFF	0xFF	0x23		
Request Programming Session	0xFF	0x10	0x02		
Request Seed	0xFF	0x27	0x01		
Key from Seed	0xFF	0x27	0x02	Key (3 Byte)	
Activate parallel Programming	0xFF	0xFF	0x10		
Deactivate parallel Programming	0xFF	0xFF	0x1F		
Request Download	0xFF	0x34	Start Block Byte	Offset Byte	Start Addr (4 Byte)
Transfer	Seq. Nr	Data (7 Byte)			
Transfer Exit	0xFF	0x37	Block Length (4 Byte)		
Reboot ECU	0xFF	0x11	0x81		
Backup SW	0xFF	0xFF	0x0A		
Restore SW	0xFF	0xFF	0x0B		

4.7 Parallel Update

The parallel update has two different types as seen in Figure 4.9. The first covers the scenario of one vehicle with two or more ECUs, that have the same SW version and need to have the same update. This scenario is applied if for example, the ECU responsible to control the window movement is only in charge of one the driver side, therefore the same ECU is present on the passenger side. Both ECUs most likely run the same SW version and can therefore be updated like illustrated in scenario one. In this case the WVI connects to the two ECUs individually, initiates a programming session, and performs the seed and key authentication. After that, the WVI sends the "Activate parallel Programming" to the two ECUs, activating that they listen to messages with the ID specifically for parallel

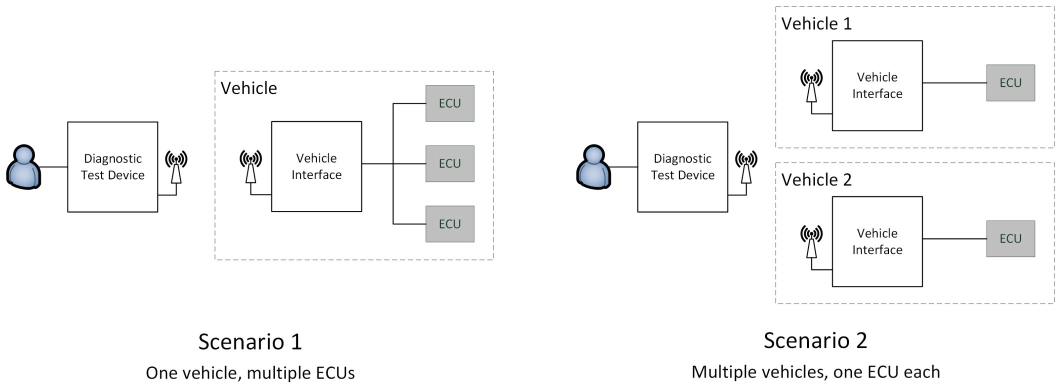


Figure 4.9: Types of parallel updates.

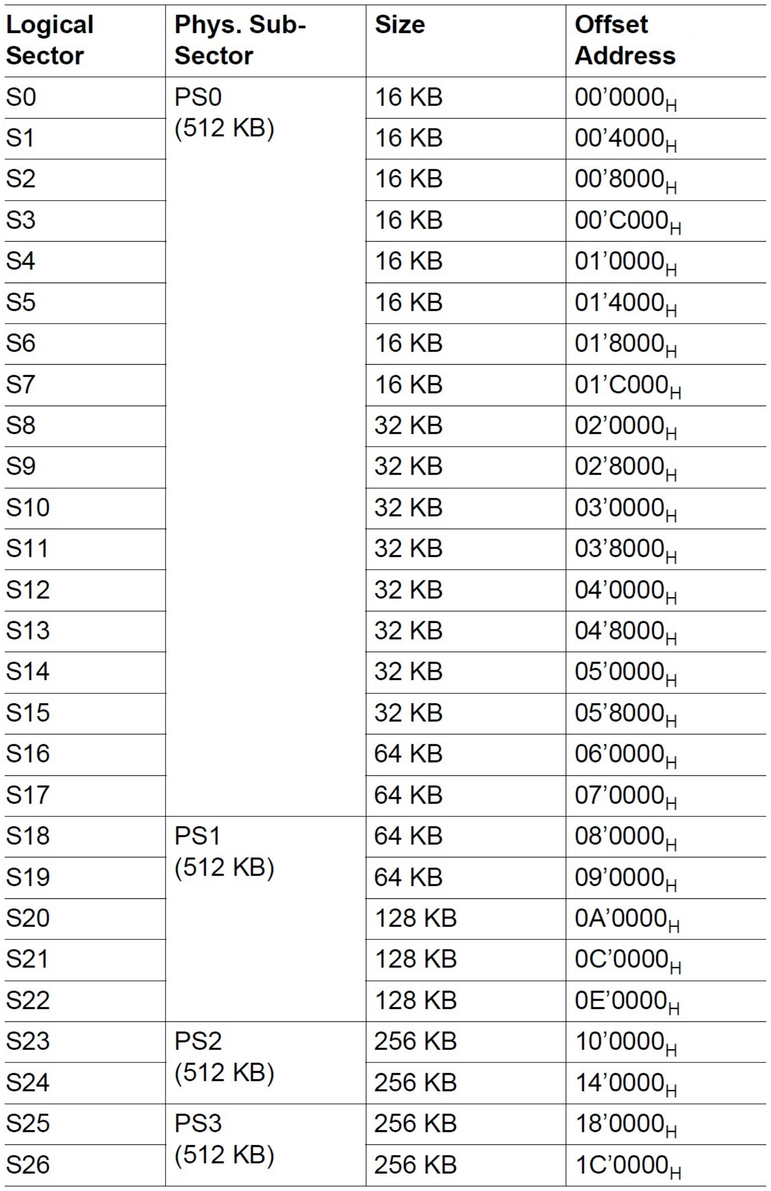
updates. Now both ECUs can be updated as they listen to the same ID, only the WVI has to wait for as many acknowledgments as ECUs are being updated. The ECU itself does not need to know about the other ECUs, or that a parallel update is performed (Except the differnt ID of the messages). To still handle transmission errors, the only one that needs to know how many ECUs are updated, is the WVI. It counts the number of acknowledgment messages for every packet before transmitting the next. As the ECUs acknowledge the messages with their own ID, the WVI is also able to detect if and which one of the ECUs suddenly gets unresponsive. This reduces the speed of the update as it is slowed down to the speed of the slowest ECU and increases the number of acknowledgment messages on the bus, but the data is only sent once.

The second type covers the "workshop scenario" mentioned in Section 1.1 where multiple vehicles with the same ECU running the same SW version need to be updated. After the VW emission scandal with around 11 million affected vehicles, a great portion of those vehicles only required a SW update. With standard update procedures today, every vehicle can only be updated one after the other. With the wireless parallel update, the cars only need to be parked outside the workshop and all identical cars, that require the same SW version, can be updated at once. In this case the management of the parallel update is done by the diagnostic test device. The DT connects to the selected WVIs one after the other. When all relevant WVIs are connected, the DT starts to initiate the seed and key authentication for every connected WVI. When the authentication between the WVIs and the ECUs is completed, the DT sequentially downloads the SW to the connected WVIs over the fast wireless connection. When that step is completed, the DT directs all connected WVIs to upload the SW to the concerning ECUs. This means, that the upload of the SW to the ECU is the only parallel step during a parallel update with multiple WVIs. As the CAN connection is the slowest part of the update system, it has the biggest impact on the overall update time. The WVI as well as the ECU to be updated do not need to know that a parallel update is performed, as for them nothing changes compared to a standard wireless update described in Section 4.2.4.

The code overhead for the parallel update is kept to a minimum with just seven lines of code specifically for the parallel update. This code is only needed for a parallel update as illustrated in Scenario 1 in Figure 4.9. If only parallel updates like in Scenario 2 are needed, those lines can be deleted.

4.8 Partial Update

To handle partial updates, I had to consider the memory structure of the ECU. When a block is going to be transferred, it starts with the “Request Download” command which also includes the Start Block Byte that identifies the type of the following block. 0x01 is used for the block that contains the startup procedure. When such a block is transferred, the value of the Offset Byte is added to the start address and stored to the data flash block (Section 4.2.2) containing all the information about the uploaded SW. If the Start Block Byte is 0x02, the start address in the data flash block is unaffected. The part of the code handling the partial update on the ECU is illustrated in the Appendix, Listing B.2.



Logical Sector	**Phys. Sub-Sector**	**Size**	**Offset Address**
S0	PS0 (512 KB)	16 KB	00’0000$_H$
S1		16 KB	00’4000$_H$
S2		16 KB	00’8000$_H$
S3		16 KB	00’C000$_H$
S4		16 KB	01’0000$_H$
S5		16 KB	01’4000$_H$
S6		16 KB	01’8000$_H$
S7		16 KB	01’C000$_H$
S8		32 KB	02’0000$_H$
S9		32 KB	02’8000$_H$
S10		32 KB	03’0000$_H$
S11		32 KB	03’8000$_H$
S12		32 KB	04’0000$_H$
S13		32 KB	04’8000$_H$
S14		32 KB	05’0000$_H$
S15		32 KB	05’8000$_H$
S16		64 KB	06’0000$_H$
S17		64 KB	07’0000$_H$
S18	PS1 (512 KB)	64 KB	08’0000$_H$
S19		64 KB	09’0000$_H$
S20		128 KB	0A’0000$_H$
S21		128 KB	0C’0000$_H$
S22		128 KB	0E’0000$_H$
S23	PS2 (512 KB)	256 KB	10’0000$_H$
S24		256 KB	14’0000$_H$
S25	PS3 (512 KB)	256 KB	18’0000$_H$
S26		256 KB	1C’0000$_H$

Figure 4.10: Sector structure of Program Flash.

For partial updates the diagnostic tester has to know the memory structure, and the borders of the different memory sections of the ECU as seen in Figure 4.10. This is necessary, because it is possible that multiple blocks are located in the same sector. In that case, even if only one of the blocks has changed, all blocks in the same section still would need to be transferred. To prevent this, it would be necessary to save the whole sector before the update and copy everything back, except the updated block, when the update is done. This would result in an undesired overhead to manage the update on the ECU.

To optimize the system to prevent the influence of big memory sections, I split up the SW in two sections according to the layout and different size of the sections. Basically the SW can be divided into two different parts. First is the part of the SW that most likely does not need any updates. It does not need to be split up further and can be placed in one of the larger sections of the memory. The second is the part of the SW that most likely needs to be updated and should be placed in the smaller memory sections. It is also possible to further split up this part into multiple sections, which further reduces the data that needs to be transferred and improves update time. In Figure 4.10 the "Offset Address" is added to the base address of the regarding program flash bank, 0x80000000 for Bank 0 and 0x80200000 for Bank 1. Looking at the different sizes of the sections, it can be seen that the size is increasing from the beginning to the end. So ideally the smaller sections at the beginning should be used for blocks that need frequent updates. It has to be considered that the blocks should not be filled completely, so that a change in one of the blocks, does not affect the following block.

4.9 Diagnose Function

To identify problems during initialization of the uploaded SW, I introduced a diagnose functionality to the uploaded SW. It is done by adding a new method to the uploaded SW that is called after the initialization sequence of the uploaded SW and sends a status to the WVI. The content of this method depends on the functionality of the uploaded SW and should check if the used ECU modules are initialized correctly and maybe check their functionality if possible. If the WVI does not receive any status or an error status, the WVI is able to restore the previously backed up SW version. This provides assurance, to some extent, that the SW works correctly and automatically starts a restore procedure if the SW does not initialize correctly and a backup is available. In the test setup, the diagnose function read the temperature value from a previously initialized sensor on the TC277 application kit. If there is an error in the start up procedure, the initialization phase, or the diagnose function itself, the temperature sensor would return an error, or the diagnose function would not return an acknowledgment at all.

4.10 Backup and Restore

The last major contribution is the introduction of a backup and restore functionality. To provide a fallback, a running SW can be backed up before an update. It is done by copying the SW from the program flash bank 0 (0x80000000 - 0x803FFFFF) to bank 1 (0x80400000 - 0x807FFFFF). If more than the first program flash bank is needed, the backup could also be stored on the SD card, providing more space for data in the program flash. For that the bootloader would also have to include a driver for the SD card interface. To be able to restore the SW it is also necessary that the block in the data flash, described in section 4.2.2, is saved. During the backup, the data flash block is copied from its original address at 0xAF05E000 to the backup address at 0xAF05F000. If the SW and the data flash block are stored, the backup is completed and signaled by writing the start address to the “Backup Program Address” field in the data flash block. The backup starts by reading the start address of the first block and the corresponding end address. Now the data of the first block is stored to the backup address. In the implemented system, the backup is stored in the second program flash bank “PFlash1”. The process is repeated until all data blocks are copied. Now the data flash block is copied to its backup address.

To restore a previously backed up SW, the backup data flash block is analyzed and the SW in program flash bank 1 is copied back to bank 0 block by block. The bootloader deletes the used memory sections before restoring the backup blocks and keeps track of the deleted sectors, so no sector is deleted after data has been restored to it. When all blocks are restored, the backup data flash block is restored to its original address. After the next reset, the restored SW automatically starts.

Chapter 5

Evaluation

In this chapter, the evaluation of the implemented wireless system is presented.

The following sections describe the tests and results for the different contributions separated into the four overall goals presented in Section 1.1. Section 5.2 shows the benefit of the modifications to the UDS protocol as well as the speed up from the optimizations of the flash driver. It further describes the efficiency evaluation of the wireless update system by comparing the time needed for a standard, a parallel, and a partial update compared to a reference wireless update system. Section 5.3 evaluates the implemented seed and key algorithm, before Section 5.4 evaluates the CRC algorithm by injecting faulty data. Finaly, Section 5.5 analyzes the time needed for the different steps of the backup and restore mechanism.

5.1 Evaluation Setup

Figure 5.1 shows the testbed that was used for the evaluation of the different parts.

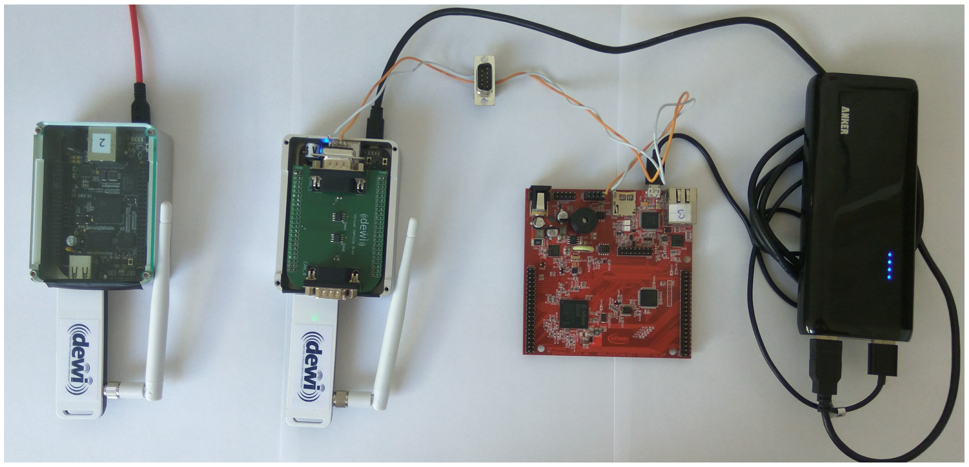


Figure 5.1: HW testbed used for evaluation.

It consists of the diagnostic test device on the left, which is directly connected to the PC through a USB connection. The diagnostic test device is a Beagle Bone Black, running the SW described in [SBK+16]. It communicates with the WVI in the middle using the TP-LINK TL-WN722N WLAN-USB-Adapter. The WVI is also a Beagle Bone Black equipped with a custom made CAN Cape (Section 2.3.1). The CAN Cape enables a CAN connection from the WVI to the ECU. It hosts two CAN transceivers that are connected to a 9-pin DSub connector each. Right to the WVI is the AURIX ECU, connected through the CAN bus (orange/white twisted wires). On the right side is a power pack that provides the power for the WVI as well as the ECU. All evaluations of the standard, parallel, and partial update were performed on the illustrated testbed. The evaluation of the flash driver and the Seed and Key algorithm, which were executed directly on the AURIX ECU.

5.2 Efficiency

This section evaluates the different contributions to improve the efficiency of the update system. It includes the evaluation of the modified UDS protocol and the optimized flash driver in Sections 5.2.1 and 5.2.2. Afterwards it evaluates the implemented wireless update system by comparing it to a Volvo FlexECU provided by DEWI project partner Volvo Trucks. Section 5.2.3 illustrates the improvements of the AURIX update system compared to the Volvo ECU. Sections 5.2.4 and 5.2.5 illustrate the benefit of the parallel and partial update.

5.2.1 UDS Evaluation

To evaluate the benefit of the modified UDS protocol, described in Section 4.6, I performed a standard wireless update with three different file sizes, 45, 386 and 445 kB. The update was performed 20 times for each file size, and the UDS protocol was used in the modified version, as well as the original version (only for the transfer command) resulting in 120 update runs. The used metric is the time needed for the transmission of the data to the ECU, which should be significantly lower when using the modified protocol. There should be no effect on the other parts of the update process, as the number of commands necessary for the update, is unchanged compared to the original protocol. Table 5.1 shows the results of the evaluation, highlighting that, for all file sizes the download time decreased. The resulting benefit of the modified UDS protocol is between 38.7% and 38.9% for file sizes of more than 45kB. As the modified protocol only benefits the download to the ECU, the benefit for the overall wireless update process depends on the file size of the update. For an update with 1 Mb, the time improvement can be as high as 38.9%, because the other steps of the update process take an insignificant amount of time. Contrary, with an update size of a few bytes, where the benefit for the overall update can be up to 0%, because the data download takes just an insignificant amount of time compared to the other parts of the update process (as seen in Table 5.3).

Table 5.1: Modified UDS protocol is 38% faster than standard (ISO 14229-3) UDS protocol.

Protocol	45 kB	386 kB	445 kB
Standard UDS protocol	1793 ms ± 22	16296 ms ± 243	18905 ms ± 271
Modified UDS protocol	1292 ms ± 20	11728 ms ± 215	13626 ms ± 223
Speedup	38.7%	38.9%	38.8%

5.2.2 Flash driver evaluation

To evaluate the performance of the optimized flash driver, I copied 250 KByte from the data flash to the program flash. The first run was performed with the standard flash driver, only using the “Write Page” command. The second one used the optimized flash driver, that automatically decides which method it uses for storing the data to maximize the performance. Table 5.2 shows the benefit of the optimized driver, by decreasing the time needed for the copy process by 61.1%, so the optimized flash driver takes only 38.9% of the time to write the same amount of data than the standard driver with the “Write Page” command, to write the same amount of data. The tests were performed using 5V programming and all used sectors were deleted before each test to eliminate the influence of the time needed for deleting.

Table 5.2: Optimized flash driver is 61% faster than standard flash driver.

Driver	250 KByte
Standard flash driver	748 ms ± 5
Optimized flash driver	291 ms ± 2
Benefit	61.1%

When all acknowledgments, sequence numbers and inter frame spaces are taken into account, the theoretical limit of the effective data rate is 33.7% of the data rate of the CAN bus. Equations 5.1 and 5.2 show the calculation of the maximum effective data rate from the flash performance and the effective data rate from the actual data rate of the bus. Due to the slow data transmission, the flash driver has no influence on the update time. If the transmission between the WVI and the ECU would have an effective data rate of more than 2.67 Mbit/s, only using the “Write Page” command would slow down the update process. With the optimized flash driver, effective data rates of up to 6.87 Mbit/s are possible without slowing down the update process. That means, if the only availaible bus is CAN, the driver optimizations have no benefit. Instead, if used with FlexRay or Ethernet, the benefit is a speed up of up to 257%.

$$\frac{8}{StoringTimefor1MByte[s]} = max.EffectiveDataRate[MBit/s] \tag{5.1}$$

$$max.EffectiveDataRate[MBit/s] = DataRate[MBit/s] * 33,7\% \tag{5.2}$$

5.2.3 Volvo FlexECU and AURIX ECU wireless update comparison

To evaluate the implemented wireless update system, the duration of the different wireless update steps was measured performing 20 wireless SW updates for every mode. The same tests were performed with the Volvo FlexECU to gain a reference value. For the evaluation, the update is split into four parts:

- **Connection-related:** Connecting the WVI to the DT (SecUp);
- **ECU-related:** Authenticating the WVI to the ECU (Seed and Key);
- **Upload:** Uploading the parsed hex file to the WVI;
- **Download:** Download the SW to the ECU.

Table 5.3 shows that the standard wireless SW update on the AURIX ECU can be performed more than two times faster. This increased update speed has several reasons. First, the increased CPU power of the AURIX ECU and the simpler protocol allow for 92% faster initialization of the update process. Second, the size of the parsed file is about 30% smaller than the original file, speeding up the upload to the WVI and at last, the update sequence on the Volvo ECU is more complex, needing more time. On the Volvo ECU, the first part of the update is the bootloader, which has to be stored and executed, before the rest of the update is performed. This benefits the AURIX ECU making it 63% faster during the download phase. Overall the complete update process of the AURIX ECU is 57% faster compared to the Volvo FlexECU.

Table 5.3: Standard wireless update for AURIX ECU only takes 43% of the time, compared to the Volvo Flex ECU.

ECU/Mode	Connection-related	ECU-related	Upload	Download	Total
Volvo/std	2369.7 ms	2410.6 ms	6585.4 ms	37315.3 ms	48681.0 ms
AURIX/std	2340.2 ms	205.4 ms	4597.0 ms	13626.3 ms	20768.9 ms
Benefit	2%	92%	31%	64%	57%

5.2.4 Parallel SW update evaluation

To evaluate the performance of the parallel update, the duration of the update is compared to a standard wireless update on the AURIX ECU and I computed the time a sequential update of two ECUs would take to compare it to the parallel update. For the evaluation the testbed was extended, such that one DT was connected to two WVIs, each connected to one AURIX ECU. This reflects the "workshop scenario" and can also be applied at the assembly line for the initial SW upload. Executing a parallel update, the SW is installed on both ECUs simultaneously instead of sequentially. Table 5.4 shows the results of the evaluation and reveals, that the parallel update takes about 25% more time compared to the standard wireless update. The increased update time is, because some of the update steps can not be completely parallelized and therefore increase the update time. If the update protocols are designed specifically for parallel SW updates, they would be able to reduce the overhead for the parallel update. Despite the increased update time and considering that in 25% more time, two ECUs instead of one where updated, the parallel update is way faster compared to a standard wireless update. In the tested scenario, the parallel update was 38% faster than updating two ECUs one after the other. The duration of the connection related part of the update increases by 65% as the DT has to connect to two WVIs. The upload of the SW to the WVI is still sequentially and therefore the upload takes 60% more time than at a standard update of a single ECU. The major improvement is during the download phase from the WVI to the ECU. As this part of the update process is completely parallel, the download to two ECUs only takes 5.5% longer compared to a single download. If the update system would utilize IEEE 802.11s multicast, the connection related, the ECU-related, as well as the upload part of parallel updates could be improved significantly.

Table 5.4: Parallel update only takes 62% of the time compared to sequential update (computed values), and only 25% slower than update of single ECU.

ECU/Mode	Connection-related	ECU-related	Upload	Download	Total
AURIX/std	2340.2 ms	205.4 ms	4597.0 ms	13626.3 ms	20768.9 ms
AURIX/sequential	4680.4 ms	410.8 ms	9194.0 ms	27252.6 ms	41537.8 ms
AURIX/parallel	3868.3 ms	262.9 ms	7378.5 ms	14372.2 ms	25881.9 ms

5.2.5 Partial SW update evaluation

The partial update is also compared to the standard wireless update on the AURIX ECU. Often a SW update only contains minor bug fixes or small changes to parameters that are used by the ECU. Such updates are very similar to the previous SW version, with most of the SW unchanged, and only a few bytes changed. A partial update only transmits the changed parts of the SW, reducing the transferred data. To evaluate this scenario, I developed two applications, both utilizing a 1024 byte parameter field, and both .hex files with 445 kB. The parameter field, which was the only difference between the two applications, was placed in a small section of the AURIX ECU as described in Section 4.2.3. Using the partial update, only the block containing the changed parameter field had to be transferred, drastically reducing the upload time to the WVI by 92.5%, as well as the download time to the ECU by 95%. Table 5.5 illustrates the impact of the partial update, being 83% faster than the standard wireless update. Biggest improvements are the 92.5% time cut transferring the data from the DT to the WVI. The results of this evaluation are only valid for the specific application and vary depending on the difference of two SW versions, the applications memory layout and the memory architecture of the used ECU. The implemented system uses a very basic partial update concept, that if not considered by the application developer can have no benefit at all. In the worst case, the partial update takes as long as a standard update of the whole SW. This is the case when the SW is located in adjoining memory blocks, which means that a simple change of one line of code at the beginning could shift the rest of the data. A shift is not recognized by the implemented system and would therefore result in an update of all data blocks.

Table 5.5: Partial update takes 83% less time than standard wireless update.

ECU/Mode	Connection-related	ECU-related	Upload	Download	Total
AURIX/std	2340.2 ms	205.4 ms	4597.0 ms	13626.3 ms	20768.9 ms
AURIX/partial	2351.7 ms	216.7 ms	347.1 ms	655.2 ms	3570.7 ms
Benefit	-0.5%	-5.5%	92.5%	95%	83%

5.3 Security

The security of the system is provided by the trust between the ECU and the DT, because only a trusted DT is allowed to perform an update. The trust between the DT and the ECU is based on a common knowledge of the seed and key algorithm. If the DT knows the algorithm to calculate the correct key from the seed, the ECU assumes it is communicating with a trusted DT. To evaluate this algorithm, I used randomized keys for brute force attacks to get access to the ECU. As I use a 24 bit key, there are 16.777.215 different solutions. In a real world scenario, the ECU randomly generates the seed for every request and therefore, brute force attacks fail to penetrate the system. In the testbed, a fixed seed was used, which allows an attacker to perform a brute force attack, trying all 16.777.215 combinations. If we would assume, that sending the request and sending the key would take 1 ms, and that the key is found after trying 50% of th combinations, a brute force attack would take at least 8.388.607 ms, which equals about 2 hours and 20 minutes. With a random seed for every request, the chance of guessing the correct key is 5,96 $*10^{-6}$% for every single attempt.

Therefore, the seed and key algorithms security contribution to the wireless update system relies solely on the secrecy of the used algorithm and a random seed to prevent brute force attacks.

Table 5.6: Evaluation of Seed and Key algorithm with fixed and random seed. (Results calculated)

	Time/Probability
Brute forcing a fixed seed 1 ms per attempt, key found after 50% tested	2 hours 20 minutes until key is found
Brute forcing a random seed	5,96 $*10^{-6}$% probability that the key is guessed

5.4 Integrity

To check the integrity of the transferred data, I compare the CRC-32 value calculated at the DT with the one calculated at the ECU. If they differ, at least one bit has been stored wrong. Evaluating a CRC to the bones, would require a separate master thesis, and therefore I just tested the algorithm with just a few cases of manipulated data. I changed respectively 1, 2, 3 and 32 Bit in a row, and every single attempt was recognized by a modified CRC value. After that, I modified 2 and 3 Bit as well as 32 Bit, independent of each other and again the corrupted data was recognized by the CRC mechanism.

Table 5.7: CRC32 succeeded for all tested manipulations.

Manipulation	Result
Flipping 1 Bit	✓
Flipping 2 Bits in a row	✓
Flipping 3 Bits in a row	✓
Flipping 32 Bits in a row	✓
Flipping 2 Bits independent	✓
Flipping 3 Bits independent	✓
Flipping 32 Bits independent	✓

According to [SGPH98] the following errors are detected by a CRC-32:

- All errors that span less than 32 contiguous bits;
- All 2-bit errors less than 2048 bits apart;
- All cases where there are an odd number of errors.

They also write, that the chance of not detecting an error in uniformly distributed data is 1 in 2^{32}. Hence, I believe that a CRC32 is sufficient enough to handle the possible corruption of single bits inside the data. Recovering data after detecting an error was not part of the thesis. If an error is detected, the concerning memory section is retransmitted. Of course there are error cases that are not detected by the CRC32 [SGPH98], but those cases are highly unlikely to happen by accident.

5.4.1 Diagnose function evaluation

To evaluate the diagnose function, I checked its behavior if an error should not be detected by the CRC-32. If undetected errors lead to a corrupt SW, the diagnose function will probably not return any acknowledgment. If the corrupted data is not part of the start up procedure, the initialization phase, or the diagnose function, a corrupt SW could be undetected until faulty behavior signals the error. During the evaluation, I toggled various bits in different locations of the code to corrupt the SW. Table 5.8 illustrates the results of the evaluation. In the first scenario, I corrupted the start up procedure, which lead the SW to being stuck during start up and therefore no response like described in Section 4.9. In a second scenario, I corrupted the initialization of the CAN module, the interrupt system, and the system (base frequency,...) each. With every of the three mentioned modifications, the SW did not give any response. A faulty response could only be accomplished when the initialization of the temperature sensor was corrupted. In that case, when reading the temperature value, the sensor returns an error value. If a variable, constants or data that is not necessary for a correct execution of the CAN module or the contents of the diagnose function, is corrupted, the response of the diagnose function is correct. In such a case, the corruption will probably be recognized through unexpected behavior of the SW or not at all.

Table 5.8: Diagnose function works correctly or not at all.

Modification	No Response	False Response	Response OK
Corrupt Start Up Procedure	✓		
Corrupt Initialization (CAN, Interrupt, System)	✓		
Corrupt Initialization (Temperature Sensor)		✓	
Corrupt Data (Variables, Constants)			✓

5.5 Safety

To evaluate the performance of the backup and restore feature, the same application previously used for evaluating the standard wireless update as well as the partial and parallel update, was used. The original .hex file with a size of 445 kB resulted in an application size of 316 kB. The application was uploaded to the ECU using the standard wireless update. To figure out, if the uploaded application initialized correctly, a diagnose function is included in the application, that can be activated through a predefined CAN message. After the diagnose function of the application returned valid information, the application was backed up, measuring the time between the instruction to back up the application and the acknowledgment. This process was repeated 20 times, and the same was done for the restore procedure. Table 5.9 shows the results of the evaluation.

Table 5.9: Duration of the backup/restore process in ms.

Application	316 KByte
Backup	5685 ms
Restore	5677 ms

As expected the difference between the backup and restore procedure is minimal. A reason for the restore procedure being slightly faster is, that when performing a backup, the information about the backup is also stored in the current data flash block (Section 4.2.2). The slowest part of the whole process is the erase procedure for every used block. This also implicates that the performance of the backup and restore feature strongly depends on the number of used memory sectors and of course the size of the application. The impact of the erase time increases when the used sectors are not filled completely.

Chapter 6

Related Work

In the last decade, several concepts and systems for wireless automotive SW updates have been introduced. Most of the concepts focused on the security aspect of an automotive wireless update system and the introduced security risks.

Security and integrity in automotive update systems Liu et al. [LLF$^+$14] have introduced an authentication protocol to connect electric vehicles to the smart grid, which ensures the authenticity of the connected vehicle. Mahmud et al. [MSH05] introduced an architecture for secure SW updates, that ensures data integrity in a wireless update system by transferring multiple copies of the SW binary. The system ensures data integrity, but does not cover vehicle authentication and key management. Dennis et al. [NL08] proposed a protocol for a secure wireless software upload in intelligent vehicles, which provides data integrity, authentication, confidentiality and freshness. They use trusted sources to authenticate the whole update chain. Steger et al. [SKH$^+$16] propose a generic framework to enable secure and efficient wireless automotive SW updates. They use a dedicated cross-layer security concept that applies strong authentication as well as encryption. This framework was implemented and enhanced in their SecUp security concept [SBK$^+$16], where they tried to cover all security and safety requirements of a wireless automotive SW update system. They use, among others, symmetric and asymmetric keys stored on devices in the network to prove the identity of those nodes and to ensure data integrity and confidentiality. The implemented system in this thesis enables the use of the SecUp framework for the AURIX ECU. With the help of SecUp's authors, the framework was also extended to provide the new mechanisms needed for the partial and parallel update.

IEEE 802.11s - Multicast Steger et al. [SKH$^+$15] also performed measurements on IEEE 802.11s networks to prove their applicability for automotive wireless SW updates. They performed several tests indoors as well as inside two different vehicles and concluded that IEEE 802.11s is able to fulfill the system requirements of a wireless SW update system for automotive. V2X[1] has become an important research topic in recent years. Tuan et al. [TSK12] introduce a priority and admission control mechanism for applications in a vehicular communication network. Their network architecture implements vehicular ad-hoc networks, based on the wireless automotive standard IEEE 802.11p and wireless LAN

[1]General term for "vehicle to vehicle" and "vehicle to infrastructure" communication.

mesh networks based on IEEE 802.11s. Their main goal is to ensure a seamless deployment of infrastructure to vehicle emergency services. To minimize the end-to-end delay as well as the probability of congestion, they transfer emergency messages using both unicast and multicast messages. Chakraborty and Nandi [CN13] propose an efficient channel access mechanism based on IEEE 802.11s. They adapt the standard mesh controlled channel access (MCCA), which provides equal time fair channel access, to fit the automotive environment by providing a proportional fair channel access. Those changes enable the different nodes to get more channel share depending on their traffic load. Utilizing multi cast groups in WSNs without the use of IEEE 802.11s, Nirmala and Manjunath [NM15] introduce SCUMG, a secure code update for multicast groups in WSNs. They divided the network in different multicast groups based on their location and each multicast group receives a different update. Their main focus is the secure and confidential transmission of the update to each group by using a key agreement protocol implemented in TinyOS (an operating system for WSN nodes).

Parallel update systems With respect to parallel updates and the use of multicast, there is also very little related work in the automotive domain. Most of the existing work has been produced by the WSN research community. Yi et al. [YFYC13] propose a testbed for WSNs with parallel reprogramming capabilities. Their testbed is able to reprogram all the nodes simultaneously which significantly reduces the time needed for WSN application update. They also used the CAN bus to connect the different nodes and connected a server that is responsible for the reprogramming. The parallel reprogramming is done by broadcasting the update to all nodes. Only nodes that encounter an error use a dedicated reply period to signal the error; therefore, the reprogramming time does not grow with the number of connected nodes. Wang et al. [WZC06] discussed different approaches for reprogramming WSNs. They looked at the different challenges of node reprogramming in WSNs, mainly scalability, energy efficiency, and hardware limitations. They also discussed several reprogramming approaches including the Firecracker protocol [LC04] and the MNP service [KW05]. The Firecracker protocol proposed by Levis and Culler [LC04], uses a combination of routing and broadcasts to deliver data to every node in a network. The data source sends the data to very distant points in the WSN. Once the data reaches those points it is broadcasted to all neighbors. This protocol is tailored for large homogeneous WSN and provides all nodes with the same data. When that protocol is used for a wireless update, it would only allow to update any connected node with the same SW. The same goal is achieved by Kulkarni and Wang with their MNP service [KW05], a multihop network reprogramming service. It is designed specifically for the Mica-2/XSM motes and uses a sender selection algorithm that tries to find the sender mote in a neighborhood that has the biggest impact broadcasting the data. To avoid collisions, their service guarantees that in a neighborhood there is at most one sender. An example in the automotive sector is provided by Lee et al. [LKHJ15]. They propose a parallel re-programming method to reduce the SW update time of ECUs. Their system is integrated in the vehicle gateways that connect the different vehicular networks. Therefore, they simply send the SW on all desired networks, involving the limitation that only one ECU per network can be updated at a time.

Partial update systems Although there is very little related literature on automotive partial or incremental updates, there is a lot of literature produced by the wireless sensor networks (WSN) research community. The basic idea of wireless updates for WSN is the same as for automotive wireless updates, but the general system requirements are completely different. Jeong and Culler [JC04] present an incremental network programming mechanism for wireless sensor networks that transmits SW updates by transferring the incremental changes. They use the Rsync algorithm to generate the difference of two SW versions. In their evaluation they reach a speed up, compared to a non-incremental update, of up to 9 times faster when changing a constant and up to 2.5 times faster for changing a few lines of the code. Marrón et al. [MGL$^+$06] introduced FlexCup, a flexible and efficient code update mechanism for sensor networks. Their system is involved in the compiling and linking process and generates meta-data that describes the compiled components. This meta-data is later used when performing an update to place the new code inside the running application and relink function calls to the correct locations.

[MSH05], [NL08], and [SBK$^+$16] mainly focus on security without considering parallel and partial updates to improve efficiency. As mentioned, [SBK$^+$16] is already part of this thesis, as it is responsible for the secure wireless communication. As the concept of this thesis can also be applied to WSNs. [YFYC13], [KW05], and [LC04] could be extended to also provide partial update capabilities, further improving their efficiency for updates. [LKHJ15] could be modified with parts of this thesis to enable the parallel programming of multiple ECUs in the same network, drastically reducing the update time for multiple ECUs. As most of the times, power efficiency is the biggest limitation for WSNs, reducing the transferred data is a main goal. As the wireless radio normally uses more power than the CPU, it is more expensive in terms of power efficiency to send data than to run some CPU cycles. Therefore, partial update concepts for WSNs are mostly more complex than the one used in this thesis. Concepts like [JC04] and [MGL$^+$06] could be implemented in the system proposed in this thesis, to further reduce the transferred data for partial updates.

Chapter 7

Conclusion and Future Work

In this master thesis, an efficient and secure wireless update system for the Infineon AURIX ECU is proposed. The system is designed with a revised version of the UDS protocol to maximize the data transfer capabilities which increased the CAN data transfer speed by 38%. Compared to a reference wireless update system (Volvo FlexECU), the implemented system, when performing an update, is 57% faster . I also implemented parallel updates, which proved to be 38% faster than a sequential update of two ECUs. Implementing partial updates enabled a speed up of 83% in the tested scenario. This speed up of course heavily depends on the structure of the SW to be uploaded and the difference between the two SW version. The wireless communication and security framework was already present as part of the SecUp framework [SBK$^+$16] and was only adapted to fit the new protocol and update sequence. To ensure the integrity during the data transmissions, multiple CRCs are used, which proved sufficient during the development of the system. An efficient backup and restore feature was added to provide a fallback solution for failed SW updates.

The evaluation section illustrates the benefits of the implemented wireless update system and the advanced features for partial and parallel updates, as well as the improved data throughput of the modified UDS protocol. It also evaluates the robustness of the used "CRC" and "Seed and Key" algorithm, and shows the improvements of the optimized flash driver. The evaluation clearly shows the improvements of the implemented system and its advanced features.

With more time and resources, I would have implemented a more advanced partial update system to further reduce the transferred data and minimize the dependencies to the applications memory layout. It should also be capable of detecting shifts in the data blocks. To achieve that, a more complex diff-based update (like [JC04]) has to be used, implicating a higher management overhead on the ECU. I would have also liked to utilize 802.11s multicast to distribute the SW to multiple WVIs at the same time, further speeding up the parallel update.

From the beginning, security was not a priority, but could improve the overall system greatly. At the moment, there are many vulnerabilities, that could be exploited especially in the communication between the WVI and the ECU. The implemented system is designed for a CAN connection between the WVI and the ECU: this connection is the bottleneck in the whole update system. Using a faster bus system (FlexRay, Ethernet) would benefit the update speed. Another way to extend the functionality of the update system would be, to connect the DT and the WVI over the Internet. This would enable a remote update infrastructure and save OEMs a lot of time for vehicle maintenance.

Starting the development all over, I would change the data handling at the ECU to be independent of the seven data bytes per message. I would also use an SD memory card for the backups, to provide more storage for the uploaded SW.

Apart from that, I am very satisfied with the implemented system, the parallel and partial update, and the achieved evaluation results. When the system is further improved in terms of security and error robustness, as well as conforming with the Automotive SPICE [1] safety requirements and the European Commission General Safety Regulation (EC) No 661/2009, it would be very well suited to be used at the assembly line and in workshops.

[1]Automotive SPICE is a process assessment model (PAM) and is intended for use when performing conformant assessments of the process capability on the development of embedded automotive systems. It was developed in accordance with the requirements of ISO/IEC 33004.

Appendix A

Definitions

Diagnostic Test Device (DT) Computing platform with wireless interface responsible to provide a link to the vehicle from the outside.

Wireless Vehicle Interface (WVI) Computing platform with wireless interface and a connection to the vehicle bus system that provides a wireless bridge to the vehicle bus.

ECU Electronic control unit used to control vehicle functions. It is connected to the vehicle bus system, and hosts the bootloader SW.

Boot Code - Start up procedure Code that is executed before the programmed application starts.

Uploaded SW SW that was uploaded to the ECU through the wireless update system

SW to be uploaded SW that is going to be uploaded to the ECU through the wireless update system

Bootloader SW running on the ECU, managing the wireless update on the ECU

Parser SW that transforms one file into another file.

Interpreter SW that reads and extracts data from a file.

Intel Hex File (Hex File) File created by the development toolchain containing the data of the SW to be uploaded

Parsed File File created by the parser by extracting the data from the Hex File

Appendix B

Code Samples

B.1 Bootloader

```
void my_can_rx_handler(unsigned int id, unsigned char len, unsigned char *
   data, unsigned char mbidx)
{
  if(id != 0x120+ECU_ID && id != 0x7DF)
    return; //should not be reached

  int i = 0;
  // Send message back with ID (oldID | 0x40)
  unsigned int idack=id;
  idack|=0x40;

  if(id == 0x7DF) //OBD command
  {
    //Map from ECU ID to VIN
    uint32 VIN = 0;
    getVIN(&VIN);

    if(*(data)==0x02 && *(data+1)==0x09 && *(data+2)==0x02)
    {
      unsigned char *Ackdata = malloc(6 * sizeof(unsigned char));
      tidx = CAN_AddMessageTX(0, 6, 0x7E7+ECU_ID, 0);
      *(Ackdata)    = 0x05;
      *(Ackdata+1) = 0x49;
      *(Ackdata+2) = (VIN & 0xFF000000) >> 24;
      ...
      *(Ackdata+5) = (VIN & 0x000000FF);
      CAN_UpdateMessage(tidx, Ackdata);
      free(Ackdata);
    }
    return;
  }

  if(id == (unsigned int)(0x120+ECU_ID)) //New UDS commands
  {
    if(*(data)==0xFF && *(data+1)==0xFF)
    {
      if(*(data+2)==0x01) //set ECU ID
      {
        ECU_ID = *(data+2);
        updateECUID = 1;
      }else if(*(data+2)==0x02) //setVIN
      {
        VIN |= (*(data+3) << 24) & 0xFF000000;
        ...
        VIN |= (*(data+6)       ) & 0x000000FF;
        updateVIN = 1;
      }else if(*(data+2)==0x03) //setSWVer
      {
        SW_Ver |= (*(data+3) << 24) & 0xFF000000;
        ...
        SW_Ver |= (*(data+6)     ) & 0x000000FF;
        updateSWVer = 1;
      }
      if(*(data+2)==0x0A) //Backup SW
      {
        backupSWflag = 1;
      }else if(*(data+2)==0x0B) //Restore SW
      {
        restoreSWflag = 1;
      }
      if(*(data+2)==0x01 || *(data+2)==0x02 || *(data+2)==0x03 || *(data+2)
   ==0x0A || *(data+2)==0x0B) //send response with command code |0x40
      {
        tidx = CAN_AddMessageTX(0, 3, idack, 0);
        *(data + 2) = *(data + 2) | 0x40;
        CAN_UpdateMessage(tidx, data);
        return;
      }
    }
    if(*(data)==0xFF && *(data+1)==0x10 && *(data+2)==0x02) //Request
   Programming Session
    {
      curr_session = SESSION_PROGRAMMING;
      tidx = CAN_AddMessageTX(0, 3, idack, 0);
      *(data+1) = *(data+1) | 0x40;
      CAN_UpdateMessage(tidx, data);
      memset(MemoryDeletionMap, 0, sizeof MemoryDeletionMap); //Set all
   Sections to not deleted as a new programming session starts
      return;

    }else if(curr_session == SESSION_PROGRAMMING)
    {
      if(*(data)==0xFF) //We have a command
      {
        if(*(data+1)==0x27 && *(data+2)==0x01) //Request Seed (Calculates
   Key for Seed and sends Seed)
        {
          ...
        }else if(*(data+1)==0x27 && *(data+2)==0x02) //Key from Seed
        {
          ...
```

Listing B.1: Part of CAN rx handler.

```
    if(curr_session == SESSION_PROGRAMMING)
    {
      if(*(data)==0xFF) //We have a command
      {
        ...

        }else if(*(data+1)==0x34) //Request download
        {
          if (*(data+2) == 0x01 || *(data+2) == 0x02)
          {
            START_BLOCK_ADDRESS = 0x0;
            START_BLOCK_ADDRESS |= (*(data+4) << 24);
            START_BLOCK_ADDRESS |= (*(data+5) << 16);
            START_BLOCK_ADDRESS |= (*(data+6) <<  8);
            START_BLOCK_ADDRESS |= (*(data+7)      );

            Flash_Set_Address(START_BLOCK_ADDRESS);
            firstPackage = 1;

            if (*(data+2) == 0x01) //This is the download request for the
    first Block, which means the program address in the DFlash Block has to
    be changed
            {
              START_BLOCK_OFFSET = *(data+3);
              START_OF_PROG = START_BLOCK_OFFSET+START_BLOCK_ADDRESS;
            }

            tidx = CAN_AddMessageTX(0, 2, idack, 0);
            deleteFlash = 1;
            _disable();
            return;
          }
          return;
        }
        ...
```

Listing B.2: Handling for partial update.

```
  if(id == (unsigned int)(0x120+ECU_ID)) //
  {
    unsigned char* seq = (unsigned char*) malloc (1* sizeof(unsigned char));
    memcpy(seq,data,sizeof(char));
    if(*seq == 0x00 || *seq ==0xFF) //should not be reached
      return;

    //We have a data package with seq Nr between 0x01 and 0xFE
    if(firstPackage!=0)
    {
      lastAcked=*seq;
      firstPackage=0;
    }else if(*seq==lastAcked)
    {
      //we already received this package ==> Skip this
      free(seq);
      return;
    }else{
      //check if this is the next package -> seqNr incremented by one
      if(*seq == (lastAcked+1) || ((*seq==1) && lastAcked == 0xFE))
      {
        //This is the next id so we are ok
        lastAcked=*seq;
      }else if(*seq>(lastAcked+1)||(*seq==1) || *seq<(lastAcked+1))
      {
        if(lastAcked == 0xFE && *seq != 0x01)
        {
          missed +=(*seq-1);
        }else if(*seq>(lastAcked+1))
        {
          missed += *seq-lastAcked-1;
        }else if(*seq<(lastAcked+1))
        {
          missed += (0xFF-lastAcked)+ *seq;
        }
        lastAcked=*seq;
        *data = 0x00;
      }else
      {
        free(seq); //should not be reached
        return;
      }
    }
    tidx = CAN_AddMessageTX(0, 1, idack, 0);
    free(seq);
  }
  CAN_UpdateMessage(tidx, data);  // send Acknowledge (Ack is simply the
    sequence number or 0x00 for an error)
```

Listing B.3: Handling of sequence of data packets.

B.2 Flash Driver

```
Flash_Address_Allignment Flash_Set_Address(unsigned int pageAddr) {

  IfxFlash_clearStatus();

  if (mConfigFlash._assemblyBufferCount != 0)
  {
    //Checking Page Mode shouldn't be necessary because if load page was
    called we are in page mode
    //and otherwise assemblyBufferCount should be 0
    Flash_WriteBuffer();
  }

  if ((pageAddr & 0xf0000000) == 0x80000000)
  {
    pageAddr&=0x0fffffff;
    pageAddr|=0xa0000000; //Always write to cached memory
  }

  if ((pageAddr & 0xff000000) == 0xa0000000)    // program flash
  {
    if (pageAddr <= IFXFLASH_PFLASH_END)
    {
      if ((pageAddr - IFXFLASH_PFLASH_START)
          % IFXFLASH_PFLASH_BURST_LENGTH == 0)
      {
        mConfigFlash._addr_allignment = PFlash_alligned;
      } else if ((pageAddr - IFXFLASH_PFLASH_START)
          % IFXFLASH_PFLASH_PAGE_LENGTH == 0)
      {
        mConfigFlash._addr_allignment = PFlash_single;
      } else
        mConfigFlash._addr_allignment = PFlash_unalligned;
    }
  } else if ((pageAddr & 0xff000000) == 0xaf000000)       // data flash
  {
    if (pageAddr <= IFXFLASH_DFLASH_END)
    {
      if ((pageAddr - IFXFLASH_DFLASH_START)
          % IFXFLASH_DFLASH_BURST_LENGTH == 0)
      {
        mConfigFlash._addr_allignment = DFlash_alligned;
      } else if ((pageAddr - IFXFLASH_DFLASH_START)
          % IFXFLASH_DFLASH_PAGE_LENGTH == 0)
      {
        mConfigFlash._addr_allignment = DFlash_single;
      } else
        mConfigFlash._addr_allignment = DFlash_unalligned;
    }
  } else
  {
    mConfigFlash._addr_allignment = Flash_Area_Invalid;
  }

  if (mConfigFlash._addr_allignment != Flash_Area_Invalid) //Only set
   address if address valid
    mConfigFlash._actprogaddr = pageAddr;

  return mConfigFlash._addr_allignment;
}
```

Listing B.4: Flash Set Address automatically detects address alignment.

```
  ...
  mConfigFlash._assemblyBufferCount += 8; //increment assemblyBufferCount
   (64 bit = 8byte)

  //Automatically execute Write Burst if possible otherwise write page until
     aligned with burst
  switch (mConfigFlash._addr_allignment)
  {
  case PFlash_alligned:
    if (mConfigFlash._assemblyBufferCount == IFXFLASH_PFLASH_BURST_LENGTH)
      return 1; //Flash_WriteBuffer();
    break;
  case PFlash_single:
    if (mConfigFlash._assemblyBufferCount == IFXFLASH_PFLASH_PAGE_LENGTH)
      return 1; //Flash_WriteBuffer();
    break;
  case DFlash_alligned:
    if (mConfigFlash._assemblyBufferCount == IFXFLASH_DFLASH_BURST_LENGTH)
      return 1; //Flash_WriteBuffer();
    break;
  case DFlash_single:
    if (mConfigFlash._assemblyBufferCount == IFXFLASH_DFLASH_PAGE_LENGTH)
      return 1; //Flash_WriteBuffer();
    break;
  default:
    break;
  }
  return 0;
}
```

Listing B.5: Part of Flash_Load_Page, that signals when to write the buffer.

```
  ...
  if (mConfigFlash._addr_allignment == PFlash_alligned || mConfigFlash.
    _addr_allignment == PFlash_single)
  {

    if (mConfigFlash._assemblyBufferCount <= (IFXFLASH_PFLASH_BURST_LENGTH)
        && mConfigFlash._assemblyBufferCount >  IFXFLASH_PFLASH_PAGE_LENGTH
   && mConfigFlash._addr_allignment == PFlash_alligned)
    {
      //use writeBurst to PFlash
      IfxFlash_writeBurstPFlash(mConfigFlash._actprogaddr);
      while (FLASH0_FSR.U & 0x1E);
    } else
    {
      //use writePage to PFlash
      int i = 0;
      for(i=0; i<mConfigFlash._assemblyBufferCount; i+=
   IFXFLASH_PFLASH_PAGE_LENGTH)
      {
        if (!FLASH0_FSR.B.PFPAGE)
        {
          IfxFlash_clearStatus();
          IfxFlash_enterPageModePFlash(IFXFLASH_PFLASH_START);
          while (FLASH0_FSR.B.SQER);
        }
        i+=1;
        IfxFlash_writePagePFlash(mConfigFlash._actprogaddr+(i-1)*0x10);
        while (FLASH0_FSR.U & 0x1E);
      }
    }
    ...
```

Listing B.6: Part of Flash_WriteBuffer, that automatically uses best way to write data.

Appendix C

Modifying SW to be used with wireless SW update

The following pages briefly describe the changes that have to be performed at an existing project, so it can be used with the wireless update system. The first part describes the necessary changes so the bootloader of the wireless update system is not overwritten. It also illustrates how to introduce new memory sections at specific addresses which can be beneficial for the partial update. The second part describes the code changes, that allow the wireless update system to interrupt a running SW and start the bootloader.

C.1 Mandatory

There are two changes to be made in the linker File, to move the code so it does not overwrite the bootloader. The lowest address where the SW to be uploaded is allowed to start is 0x80008000. If the SW should start somewhere else, the size has to be adapted accordingly.

The following changes have to be made at the beginning of the linker file (Line numbers refer to an unmodified linker file of the Free Tricore Entry Toolchain for the AURIX Tc277c):

```
20 /*
21  * Global
22  */
23 /*Program Flash Memory (PFLASH0)*/
24 __PMU_PFLASH0_BEGIN = 0x80008000;   /*Change it from 0x80000000*/
25 __PMU_PFLASH0_SIZE = 2016K;       /*Change it from 2M*/
26 /*Program Flash Memory (PFLASH1)*/
27 __PMU_PFLASH1_BEGIN = 0x80200000;
28 __PMU_PFLASH1_SIZE = 2M;
29 /*Data Flash Memory (DFLASH0)*/
30 __PMU_DFLASH0_BEGIN = 0xAF000000;
31 __PMU_DFLASH0_SIZE = 1M;
```

Listing C.1: Change global definition of PMU PFLASH0 in linker file.

```
/*
 * internal flash configuration
 */
MEMORY
{
  PMU_PFLASH0 (rx!p): org = 0x80008000, len = 2016K  /*Program Flash Memory
     (PFLASH0)*/
  PMU_PFLASH1 (rx!p): org = 0x80200000, len = 2M  /*Program Flash Memory (
    PFLASH1)*/
  PMU_DFLASH0 (r!xp): org = 0xAF000000, len = 1M  /*Data Flash Memory (
    DFLASH0)*/
...
```

Listing C.2: Change size of PMU PFLASH0 in memory configuration in linker file.

C.1.1 Optional - for partial update optimization

It is also possible to introduce more sections at different addresses. An example of that can be seen below:

```
/*
 * Global
 */
__PMU_PFLASHX_BEGIN = 0x80008000;
__PMU_PFLASHX_SIZE = 16K;
/*Program Flash Memory (PFLASH0)*/
__PMU_PFLASH0_BEGIN = 0x80100000;
__PMU_PFLASH0_SIZE = 1M;
/*Program Flash Memory (PFLASH1)*/
__PMU_PFLASH1_BEGIN = 0x80200000;
...
...
/*
 * internal flash configuration
 */
MEMORY
{
  PMU_PFLASHX (rx!p): org = 0x80008000, len = 16K  /*Program Flash Memory (
    PFLASH0)*/
  PMU_PFLASH0 (rx!p): org = 0x80100000, len = 1M  /*Program Flash Memory (
    PFLASH0)*/
  PMU_PFLASH1 (rx!p): org = 0x80200000, len = 2M  /*Program Flash Memory (
    PFLASH1)*/
...
```

Listing C.3: Adding global definition for 16K memory PMU PFLASHX and adding it to the memory configuration in the linker file.

At last a new section has to be added which is placed in the new memory section, this can be added at the end of the linker file:

```
  .newSec  :
  {
    PROVIDE(__newSec_start = .);

     /*PROTECTED REGION ID(Protection: iROM . newSec.begin) ENABLED START*/
       /*Protection-Area for your own LDF-Code*/
     /*PROTECTED REGION END*/

     *(. newSec) /*Code section*/
     *(. newSec *)
     *(.gnu.linkonce.t.*)

     /*PROTECTED REGION ID(Protection: iROM . newSec) ENABLED START*/
       /*Protection-Area for your own LDF-Code*/
     /*PROTECTED REGION END*/

    PROVIDE(__newSec _end = .);
    . = ALIGN(8);

  } > PMU_PFLASHX /* PMU_PFLASHX: Program Flash Memory (PFLASHX) */
```

Listing C.4: Introduce a new section .newSec in linker file that is placed in PMU PFLASHX.

In the code itself, functions or variables can be placed in the newly created section as seen below. Everything between the two “pragma” keywords is put in the section. It can also contain the whole function implementation.

```
#pragma section .newSec

int VariableInSectionNewSec;
int Flash_WriteBuffer();

#pragma section
```

Listing C.5: Place a variable and a function in section .newSec.

C.2 Optional

To also provide the possibility to switch back to the bootloader once the program is uploaded, the uploaded SW has to contain a few more functions described in the next three sections.

C.2.1 Add CAN and Flash driver

First the SW has to implement a flash and a CAN driver. To do this the two files (flash.c and can.c) as well as the corresponding header files and their dependencies have to be added to the project. Then the two drivers have to be initialized before the main loop of the SW starts as seen in the following code:

```
  // Setup interrupt handlers
  InterruptInit();

  // Initialize systen timer
  TimerInit(SYSTIME_CLOCK);
  TimerSetHandler(my_timer_handler);

  //Initialize CAN module
  CAN_init(500000, 1);
  CAN_Install_Callbacks(my_can_rx_handler, my_can_tx_handler);
```

```
Flash_Init();
// Enable interrupts globally
_enable();

ECU_ID = getECU_ID();    //function from stats.c providing the stored ECU
  ID
unsigned char ridx = CAN_AddMessageRX(0, 4, 0x120+ECU_ID, 0, 0x7FF); //0
  x100...0x107
...
while (1)
{
...
```

Listing C.6: Initialization of interrupt handlers, system timer, CAN driver, and flash driver.

C.2.2 Add stats.c/.h file

The stats.c/.h file implement all functions to get the ECU ID and to delete the program address stored in the DFlash without losing other data (VIN, SW Ver). To execute the bootloader after a reset, the SW has to delete the program address stored in the DFlash. After that a reset leads to the execution of the bootloader. Among others, stats.c defines the following two functions:

- deleteProgramAddress();
- getECU_ID().

C.2.3 Add delete and reset command to can handler

The last step is to add the CAN commands to trigger the deleteProgramAddress() and the SYSTEM_Reset(). Those commands can also be combined and triggered by one CAN command. Using two separate commands an example can be seen in the following code. A CAN message wit ID 0x120 + ECU_ID and the data “0x0F0F” triggers the delete program address command. 0x120 + ECU_ID and the data “0x0505” triggers the system reset. Those commands can be adapted to fit the developed SW.

```
void my_can_rx_handler(unsigned int id, unsigned char len, unsigned char *
    data, unsigned char mbidx)
{
  unsigned char tidx;
  // Send message back with ID (oldID | 0x40)
  unsigned int idack=id;
  idack|=0x40;

  if(id == (0x120+ECU_ID))
  {
    tidx = CAN_AddMessageTX(0, 8, idack, 0);
    CAN_UpdateMessage(tidx, data);

    if(*(data)==0x0F && *(data+1)==0x0F)
    {
      deleteProgramAddress();
    }
    if(*(data)==0x05 && *(data+1)==0x05)
    {
      SYSTEM_Reset();
    }
  }
  tidx = CAN_AddMessageTX(0, 8, id, 0);
  CAN_UpdateMessage(tidx, data);
  return;
}
```

Listing C.7: Add a handler for the CAN commands to delete the program address and to execute a system reset.

Bibliography

[80206] IEEE Standard Association Working Group IEEE 802.11. HWMP Protocol specification, November 2006.

[80210] IEEE Standard Association Working Group IEEE 802.11. IEEE Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications-Amendment 6: Wireless Access in VehicularEnvironments, June 2010.

[And15] Greenberg Andy. Hackers Remotely Kill a Jeep on the Highway - With Me in It. *WIRED*, Jul 2015.

[CN13] S. Chakraborty and S. Nandi. IEEE 802.11s Mesh Backbone for Vehicular Communication: Fairness and Throughput. *IEEE Transactions on Vehicular Technology*, 62(5):2193–2203, Jun 2013.

[CWK11] Ting-Yun Chi, Wei-Cheng Wang, and Sy-Yen Kuo. *uFlow: Dynamic Software Updating in Wireless Sensor Networks*, pages 405–419. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[Gab16] Nelson Gabe. Over-the-air updates on varied paths. *Automotive News*, Jan 2016.

[JC04] Jaein Jeong and D. Culler. Incremental network programming for wireless sensors. In *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004.*, pages 25–33, Oct 2004.

[KW05] S. S. Kulkarni and Limin Wang. MNP: Multihop Network Reprogramming Service for Sensor Networks. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 7–16, June 2005.

[LC04] Philip Levis and David Culler. The Firecracker Protocol. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, New York, NY, USA, 2004. ACM.

[LKHJ15] Y. S. Lee, J. H. Kim, H. V. Hung, and J. W. Jeon. A parallel re-programming method for in-vehicle gateway to save software update time. In *2015 IEEE International Conference on Information and Automation*, pages 1497–1502, Aug 2015.

[LLF$^+$14] H. Liu, X. Liang, L. Fang, B. Zhang, and J. W. Zhao. A Secure and Efficient Authentication Protocol Based on Identity Based Aggregate Signature for Electric Vehicle. In *2014 International Conference on Wireless Communication and Sensor Network*, pages 353–357, Dec 2014.

[MGL$^+$06] Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. *FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks*, pages 212–227. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[MKKJ09] H. Mukhtar, B. W. Kim, B. S. Kim, and S. S. Joo. An efficient remote code update mechanism for Wireless Sensor Networks. In *MILCOM 2009 - 2009 IEEE Military Communications Conference*, pages 1–7, Oct 2009.

[MSH05] S. M. Mahmud, S. Shanker, and I. Hossain. Secure software upload in an intelligent vehicle via wireless communication links. In *IEEE Proceedings. Intelligent Vehicles Symposium, 2005.*, pages 588–593, June 2005.

[NASZ05] V. Naik, A. Arora, P. Sinha, and Hongwei Zhang. Sprinkler: a reliable and energy efficient data dissemination service for wireless embedded devices. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10 pp.–286, Dec 2005.

[New15] BBC News. Volkswagen says 800,000 cars may have false co2 levels - bbc news. http://www.bbc.com/news/business-34712435, Nov 2015. (Accessed on 08/07/2017).

[NL08] D. K. Nilsson and U. E. Larson. Secure Firmware Updates over the Air in Intelligent Vehicles. In *ICC Workshops - 2008 IEEE International Conference on Communications Workshops*, pages 380–384, May 2008.

[NM15] M. B. Nirmala and A. S. Manjunath. SCUMG: Secure Code Update for Multicast Group in Wireless Sensor Networks. In *2015 12th International Conference on Information Technology - New Generations*, pages 249–254, April 2015.

[NXP13] NXP Semiconductors. *High-speed CAN transceiver*, 4 2013. Rev. 3.

[Ope17] Opel. Opel onstar online- und service-assistent. http://www.opel.at/onstar/onstar.html, Aug 2017. (Accessed on 21/08/2017).

[SBK$^+$16] 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)*, pages 628–636, Aug 2016.

[SGPH98] J. Stone, M. Greenwald, C. Partridge, and J. Hughes. Performance of checksums and CRCs over real data. *IEEE/ACM Transactions on Networking*, 6(5):529–543, Oct 1998.

[SKH$^+$15] 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)*, pages 1–8, July 2015.

[SKH$^+$16] M. Steger, M. Karner, J. Hillebrand, W. Rom, C. Boano, and K. Römer. Generic framework enabling secure and efficient automotive wireless SW updates. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, Sept 2016.

[TSK12] D. T. Tuan, S. Sakata, and N. Komuro. Priority and admission control for assuring quality of I2V emergency services in VANETs integrated with Wireless LAN Mesh Networks. In *2012 Fourth International Conference on Communications and Electronics (ICCE)*, pages 91–96, Aug 2012.

[Vol17] Volkswagen. Volkswagen car-net description. http://volkswagen-carnet.com/int/en/start/app-overview.html, Aug 2017. (Accessed on 21/08/2017).

[WLfL09] T. Wen, Z. Li, and Q. f. Li. An Efficient Code Distribution Protocol for OTAP in WSNs. In *2009 5th International Conference on Wireless Communications, Networking and Mobile Computing*, pages 1–4, Sept 2009.

[WZC06] Qiang Wang, Yaoyao Zhu, and Liang Cheng. Reprogramming wireless sensor networks: challenges and approaches. *IEEE Network*, 20(3):48–55, May 2006.

[YFYC13] Kefu Yi, Renjian Feng, Ning Yu, and Peng Chen. PARED: A testbed with parallel reprogramming and multi-channel debugging for WSNs. In *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 4630–4635, April 2013.