Design and implementation of an automotive experimental platform for ADAS

VIKTOR KARLQUIST
Design and implementation of an automotive experimental platform for ADAS

Viktor Karlquist

Master of Science Thesis

Embedded Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

3 February 2017

Examiner: Zhonghai Lu
Supervisor: Yuan Yao
Abstract

Road fatalities are decreasing every year. A major reason is the increased presence and the increased complexity of Advanced Driver Assistance Systems (ADAS). ADAS are becoming one of the biggest research areas in the automotive industry. This thesis presents a design and implementation of an automotive experimental platform for ADAS.

The thesis provides an overview of automotive concepts related to ADAS, such as Electronic Control Unit (ECU), commonly used communication protocols in cars and Operating Systems (OS) used for automotive purposes. In addition a general background to ADAS is presented as well as an introduction to the state of the art technology.

The design of the platform is presented with a detailed hardware description as well as a thorough motivation to the design choices. The design consists of multiple ECUs connected through a switched Ethernet network with multiple switches. The design also includes a model car equipped with actuators and sensors, made to mimic a real car.

Several functions have been implemented in the system such as reading sensor data, controlling actuators and a control interface to control the car. As a proof-of-concept platform the resulting system can be exploited and flexibly extended for various ADAS functions and safety engineering.

Uppsatsen ger en överblick av concept relaterade till ADAS, såsom Electronic Control Unit (ECU), vanligt förekommande kommunikationsprotokoll i bilar och Operativ System (OS) använda i bilindustrin. Dessutom presenteras en generell beskrivning av vad ADAS är och vad som är den senaste teknologin inom ADAS.


Ett flertal implementation presenteras, såsom inläsning av sensordata, kontroll av motorer och ett kontrollgränsnitt för att styra modellbilen. Resultatet av uppsatsen är ett första försök till en design och implementation av en platform. Plattformen är ett proof-of-concept som möjliggör för flexibla tillägg och säkerhetstestning.
Acknowledgements

I would like to acknowledge my examiner and advisers support throughout the thesis. I would also like to thank KTH Service Center for assisting with purchases of components.
# Contents

## 1 Introduction

1.1 Problem description ........................................ 2  
1.2 Goals ......................................................... 2  
1.3 Ethics and Sustainability .................................. 3  
1.4 Research method ............................................. 3  
1.5 Development methodology ................................ 4  
1.6 Structure of this thesis .................................... 4  

## 2 Background

2.1 Electronic Control Unit .................................. 5  
  2.1.1 Microcontroller ........................................... 6  
  2.1.2 Microprocessor ........................................... 6  
2.2 Advanced Driver Assistance Systems .................... 7  
  2.2.1 Sensors .................................................. 8  
  2.2.2 Sensor fusion ............................................ 9  
  2.2.3 Cooperating vehicles ................................... 9  
  2.2.4 Vehicle-to-Infrastructure Communication ............. 10  
2.3 Autonomous cars ......................................... 11  
2.4 Communication protocols in cars ......................... 12  
  2.4.1 LIN ..................................................... 12  
  2.4.2 CAN ...................................................... 13  
  2.4.3 Flexray .................................................. 13  
  2.4.4 Ethernet ................................................... 14  
    2.4.4.1 CSMA/CD .......................................... 15  
    2.4.4.2 Loop-prevention ................................... 16  
    2.4.4.3 AVB/TSN .......................................... 17  
    2.4.4.4 802.1Q VLAN ...................................... 20  
2.5 The real-time concept ..................................... 21  
2.6 Operating systems ......................................... 21  
  2.6.1 AUTOSAR .................................................. 22  
  2.6.2 Embedded Linux ........................................ 24
# Contents

2.6.2.1 Linux Real-time Scheduler ........................................ 24
2.6.2.2 PREEMPT_RT Patch ............................................. 24

2.7 Related work ....................................................... 25

3 Hardware and software tools ........................................... 26

3.1 Hardware ..................................................................... 26
3.1.1 BeagleBone Black .................................................... 26
3.1.1.1 Sitara AM3358 chipset ........................................ 27
3.1.2 Peripherals ............................................................. 30
3.1.3 Device Tree ............................................................. 32
3.1.4 Serial to USB interface - SparkFun FTDI Basic Breakout .... 32
3.1.5 Raspberry Pi 3 model B ............................................. 32
3.1.5.1 RPI dedicated camera .......................................... 33
3.1.6 TP-link TL-SG105E Network Switch ................................ 35
3.1.7 Model Car ............................................................... 36
3.1.7.1 Servo motor ....................................................... 38
3.1.7.2 Brushless motor .................................................. 38
3.1.7.3 ESC ................................................................. 39
3.1.8 Android device ........................................................ 39
3.1.9 Sensors ................................................................. 39
3.1.9.1 Ultra Sonic Sensor, HC-SR04 ................................. 39
3.1.9.2 Reflective sensor, OPB715Z OPTO SWITCH ................ 40
3.1.9.3 Motion sensor, MPU-9250 .................................... 40
3.2 Operating System ....................................................... 40
3.2.1 Linux 4.4-ti-rt ......................................................... 41
3.2.2 Linux 4.4-rt ............................................................. 41
3.3 Software tools ............................................................ 42
3.3.1 Wireshark .............................................................. 42
3.3.2 Android studio ....................................................... 42
3.3.3 config-pin ............................................................. 42
3.3.4 Cyclictest ............................................................... 43
3.3.5 TI PRU C compiler .................................................. 43
3.3.6 Device Tree Compiler ............................................... 43

4 Design ......................................................................... 44

4.1 Hardware ..................................................................... 44
4.1.1 ECUs ..................................................................... 44
4.1.2 Network switches .................................................... 45
4.1.3 Sensors ................................................................. 46
4.1.4 Model car, actuators and electronics ......................... 46
4.1.5 Android device ....................................................... 47
## Contents

4.2 Choice of OS .................................................. 47  
4.3 Overview of the system architecture ...................... 48  

5 Implementations ................................................. 52  
5.1 Car network and ECU to ECU communication .............. 52  
5.2 Implementing the OS ......................................... 54  
5.2.1 BeagleBone ECU ....................................... 54  
5.2.2 Raspberry Pi ECU ....................................... 56  
5.3 Sensors ....................................................... 56  
5.3.1 Ultra Sonic sensor ...................................... 56  
5.4 Controlling actuators ....................................... 59  
5.4.1 Steering servo ......................................... 59  
5.4.2 ESC .................................................. 61  
5.5 Network monitoring ........................................ 61  
5.6 Control and monitoring interface .......................... 62  
5.6.1 Android App ........................................... 63  
5.6.2 UDP server ............................................ 64  
5.7 Error detection .............................................. 67  

6 Conclusions ..................................................... 69  
6.1 Summary ..................................................... 69  
6.1.1 Goals .................................................. 70  
6.2 Future work ................................................ 70  
6.2.1 Adding additional sensors ............................... 70  
6.2.2 Sensor fusion ......................................... 71  
6.2.3 Implement full AVB/TSN support ........................ 71  
6.2.4 Error detection ....................................... 71  
6.2.5 Benchmarks ............................................ 72  
6.2.6 Creating test scenarios ................................ 72  
6.2.7 Creating an API for external communication ........ 73  
6.2.8 Improving the control and monitoring interface .... 74  

Bibliography ........................................................ 75  

A Code ............................................................. 85
List of Figures

2.1 Sensors and Electronics in ADAS, from [1] . . . . . . . . . . . . 8
2.2 Flex ray segments, taken from National Instruments [2] . . . . . 14
2.3 Example of a broadcast storm, from [3, Chapter 8] . . . . . . . . 16
2.4 Credit-based shaper algorithm, from [4] . . . . . . . . . . . . . . 19
2.5 An overview of AUTOSAR ECU Software Architecture, take from [5] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

3.1 The BeagleBone Black board, taken from [6] . . . . . . . . . . . 27
3.2 Programming the PRU, image taken from [7] . . . . . . . . . . . 29
3.3 PRU-ARM message passing, taken from [8] . . . . . . . . . . . . 31
3.4 The Raspberry Pi 3 Model B board, taken from [9] . . . . . . . . 33
3.5 The Raspberry Pi Camera Module v2, taken from [10] . . . . . . 34
3.6 The TP-link TL-SG105E network switch, taken from [11] . . . 35
3.7 Turnigy SCT 2WD 1/10 Brushless Short Course Truck (KIT) upgraded version, taken from [12] . . . . . . . . . . . . . . . . . . . . 36
3.8 Controlling a servo motor . . . . . . . . . . . . . . . . . . . . . 38
3.9 The HC-SR04 is a distance sensor, taken from [13] . . . . . . . 40
3.10 pfifo_fast queuing policy, taken from [14]. . . . . . . . . . . 41

4.1 Network topology . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.2 Example of broken link . . . . . . . . . . . . . . . . . . . . . . . 49
4.3 Overview of the system architecture . . . . . . . . . . . . . . . . 50

5.1 Ultrasonic sensor algorithm . . . . . . . . . . . . . . . . . . . . . 57
5.2 Voltage dividing circuit . . . . . . . . . . . . . . . . . . . . . . . 59
5.3 Servo control algorithm . . . . . . . . . . . . . . . . . . . . . . . 60
5.4 Network monitoring program . . . . . . . . . . . . . . . . . . . . 62
5.5 Android application layout . . . . . . . . . . . . . . . . . . . . . . 63
5.6 Package structure of packet sent from Android Application . . 64
5.7 Server receiving commands from the Android Application . . . 65
5.8 Package structure of the data sent containing status . . . . . . 66
5.9 Mutex example . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.10 Summary of the Control and Monitoring interface . . . . . . . . 67
6.1 External communication possibilities . . . . . . . . . . . . . . . 73
List of Tables

2.1 VLAN tag ........................................... 20
3.1 BeagleBone Black hardware specification .................. 31
3.2 Raspberry Pi 3 Model B hardware specification .............. 33
3.3 Priorities according to traffic type ........................ 36
3.4 Components used to build the model car .................... 37
List of Listings

3.1 Config-pin example ............................................ 42
5.1 Vlan setup script .............................................. 53
5.2 Config-pin example ............................................ 55
A.1 Example code controlling multiple PWM signals ............ 85
# List of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAS</td>
<td>Advanced Driver Assistance System</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ARM</td>
<td>Advanced RISC Machine</td>
</tr>
<tr>
<td>AVB</td>
<td>Audio Video Bridging</td>
</tr>
<tr>
<td>BBB</td>
<td>BeagleBone Black</td>
</tr>
<tr>
<td>CAN</td>
<td>Control Area Network</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSMA/CA</td>
<td>Carrier Sense Multiple Access/Collision Avoidance</td>
</tr>
<tr>
<td>CSMA/CD</td>
<td>Carrier Sense Multiple Access/Collision Detection</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>eMMC</td>
<td>embedded MultiMediaCard</td>
</tr>
<tr>
<td>EU</td>
<td>European Union</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Array</td>
</tr>
<tr>
<td>GPOS</td>
<td>General Purpose Operating System</td>
</tr>
<tr>
<td>IMU</td>
<td>Inertial Measurement Unit</td>
</tr>
<tr>
<td>LIN</td>
<td>Local Interconnect Network</td>
</tr>
<tr>
<td>OEM</td>
<td>Original Equipment Manufacturer</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>PRU</td>
<td>Programmable Real-time Unit</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse Width Modulation</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RPI</td>
<td>Raspberry Pi</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>SoC</td>
<td>System on a Chip</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TDMA</td>
<td>Time division multiple access</td>
</tr>
<tr>
<td>TSN</td>
<td>Time Sensitive Networking</td>
</tr>
<tr>
<td>VLAN</td>
<td>Virtual Local Area Network</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In 2015 the number of road related fatalities was over 26000, half of the number of fatalities which occurred in 2001. It is a drastic decrease and the EU Commission road safety program aims to get these numbers even lower by the year 2020 [15]. Traffic researcher Johann Gwehenberger predicted in 2010 that more than half of the serious accidents in Europe could be prevented if ADAS were present [16]. Advanced Drivers Assistance Systems are the systems used to aid drivers while driving [17] and it is one of the most rapidly growing research areas in the automotive industry. Even though it is a huge research area for the automotive companies the annual revenues from ADAS are still very modest compared to other vehicle systems. One of the reasons is believed to be that many of the implementations are still being refined and have not yet hit the market [18, p. 13].

Many of the available research platforms and solutions available are proprietary and there is a need for an open source based platform which is available for everyone; including students as well as researchers. There are a lot of different types of ADAS on all different levels of complexity. There are very complex systems that can map the environment with help of Radars and Lidars and there are systems that can do depth image analysis with the help of stereo vision. In addition to the very complex systems there are also, somewhat, simpler systems such as parking assistance that can detect objects with range sensors [19].

This thesis focuses on designing and implementing a platform that allows for experimenting with different ADAS in practice. The platform is a proof-of-concept and it provides the set of tools and functionality to implement different functionalities of ADAS.

This chapter gives a general introduction to the research area, formulates a problem description and discusses the goals of the thesis. It also presents the research and development methodologies as well as providing the structure of the
1.1 Problem description

Due to the rapid increase and the growing interest for ADAS there is a need for an experimental platform to allow students and other users to test and implement existing ADAS and to allow for innovation in the subject. This thesis will investigate how to design and implement such an experimental platform for ADAS. The scope of the thesis includes both choosing the design of the platform, choosing parts, ordering parts and assembling them. It also includes implementing the platform, by implementing systems on the assembled platform, to enable users of the platform to implement their own ADAS.

1.2 Goals

This thesis aims to develop an experimental platform for ADAS. The platform should provide necessary functionality to test implementations of different functionalities of ADAS. The goal is not to provide a final solution or product, but to provide some functionality and a starting point for future development of the platform, as a proof-of-concept design.

The platform should consist of multiple ECUs and provide the underlying communication technology between the ECUs. The underlying communication technology should be implemented by researching different communication protocols and to implement what is used today in the state-of-the-art technology. Another goal of the thesis is to investigate whether it is feasible to use Embedded Linux as the operating system run on the ECUs. Below is a composed list of the goals relevant to this thesis.

- Design of an experimental platform
  - The platform should consist of multiple ECUs
  - The platform should fill the need for testing ADAS
  - The platform should contain sensors to test ADAS

- Implementation of an experimental platform
  - The platform should implement a modern underlying communication technology
There should be a way of controlling the platform externally
There should exist functionality to get data from sensors

The result of the thesis should provide a physical platform on which the user of the platform can experiment with different types of ADAS by user software that communicates with the implemented software.

1.3 Ethics and Sustainability

The main ethical benefit of this platform is to create an open source platform available to anyone. This is also an issue, to make sure that the platform is fully open source by identifying the software and hardware used and their licenses. It is also important to document the process to enable anyone to build and set up the platform. Another issue, to keep the platform available to as many as possible, is to try and keep the cost of the platform down.

By making the platform highly extendable and available the platform can adapt to future technologies and enable for future development, creating a reusable and sustainable system.

1.4 Research method

This section will start by giving an introduction to research methods and work methods, followed by a motivation to what methods are used in this thesis. A research method can be classified as either quantitative or qualitative. A quantitative method determines the credibility of a hypothesis by collecting a large set of data to analyze it with statistics and other methods. A qualitative research is to collect data and assess it subjectively [20].

A research can also be divided into two different research approaches: deductive and inductive methods. An inductive research approach is when facts and data are collected and a theory or hypothesis is developed from the gathered data. Inductive methods tend to be qualitative. A deductive work method is when a theory is first presented and data is collected to verify and validate the theory [20].

The main research methodology used in this thesis is qualitative, since the results are evaluated subjectively by a few persons. The research approach is mainly inductive, since there is no hypothesis to be proven at the beginning of the thesis. Data is collected in order to design the platform, as there is no known design prior to the work.
1.5 Development methodology

This section will describe the development methodology used and how the work during the thesis is conducted. The development methodology used in this thesis is similar to the Waterfall model. The waterfall model is a development methodology that divides the development process in distinct steps. Each step is to be completed before continuing to the next. The process normally starts by collecting all requirements, followed by the design, the implementation, the verification and maintenance [21].

This thesis started by collecting the requirements in one distinct step. The requirements were part of the thesis description but also part of a literature study to break down the requirements in more detail. Theoretical studies were conducted in order to establish a design that could fit the requirements collected in the previous step. After the design of the platform was decided and the parts required to implement the design was ordered, the implementation of the platform started. The implementation and verification process was done in a more iterative way than described in the waterfall model. Parts of the system were implemented and tested separately, before continuing with other implementations, in order to iteratively improve the implementations. According to the classical waterfall model, the implementation and verification is done in separate steps: first the implementation is done completely and then the verification of the implementation. The maintenance part of the waterfall model is irrelevant, as maintenance is out of the scope of this thesis.

1.6 Structure of this thesis

Chapter 1 gives an introduction to the thesis and states the problem, as well as defining the goal of the thesis. Chapter 2 provides the background to understand the problem and to understand the concepts discussed throughout the thesis. The third chapter, Chapter 3, describes the hardware and software tools related to the thesis. The design of the platform is described in Chapter 4. The implementations of the platform is presented in Chapter 5. The last chapter, Chapter 6, discusses the conclusions of the thesis, with respect to the goals and future work.
Chapter 2

Background

This chapter will provide the necessary background of the thesis. The chapter will start by introducing what an Electronic Control Unit is and provide several examples of such controllers. The next thing to be described is Advanced Driver Assistance Systems, it will provide several examples of implementations and a general description of what it is. This will be followed by a brief introduction to autonomous cars, in particular a case study of Google’s self driving car. Furthermore the most common communication protocols used in cars will be presented together with a new modern approach using Ethernet. Next, the concept of real-time will be briefly introduced, to provide a necessary background to real-time theory. This is followed by a look into Operating Systems in embedded systems and in particular operating systems used in the automotive industry. The last part of this chapter will discuss some related work to this thesis.

2.1 Electronic Control Unit

An Electronic Control Unit (ECU) is the term for any embedded system that controls an electrical system in a vehicle [22]. The number of ECUs in cars is drastically increasing. The amount of average ECUs in a car is expected to be above 40 units by year 2019. This amount is twice as much as it was in 2010, where the number of average ECUs in a car was merely 20 [23, p. 29]. An ECU can be, as previously stated, any embedded system that controls an electrical system. It can be anything from a simple circuit to a computer. This section will focus on Microcontrollers and Microprocessors used in automotive industry, but it is important to understand that other types of control units in cars exists, for example DSPs and FPGAs. Before diving into the two different types of embedded systems it is important to clarify the difference between the two.
A Microcontroller typically has on-chip flash memory, which means it can load the program quickly and be up and running in short time. By having an on-chip memory a microcontroller is also limited to a finite amount of memory and typically has a maximum memory of 2 Megabytes. A microprocessor on the other hand does not have this constraint as it has a separate memory for storing programs and data. This means that typically a microcontroller can be up and running faster but have access to less memory [24].

2.1.1 Microcontroller

There are several different types of microcontrollers used in today’s cars. One major supplier of microcontrollers for the automotive industry is Atmel. They offer a lot of different solutions based on the AVR 8-bit and 32-bit RISC designs. The controllers have applications ranging from controlling the electrical mirrors to infotainment applications [25]. Another supplier is Microchip with their well known PIC microcontrollers. There are a lot of different PIC microcontrollers, but they are all based on the Harvard architecture with RISC instruction set design [26]. The two families of microcontrollers share that they both use RISC design, which is a trend that will continue for the automotive industry [27].

2.1.2 Microprocessor

As algorithms are becoming more and more complex there is a need for low cost high performance CPUs [27]. One major supplier of Microprocessors in the automotive industry is Texas Instruments. They have more than 150 million solutions integrated in vehicles today, in more than 35 different original equipment manufacturers (OEM). They provide solutions for both ADAS, which will be described in detail in Chapter 2.2, and Infotainment [28]. Common for the solutions they provide is that the main processing unit used in the systems are different version of ARM processors based on the ARM architecture [29]. Another major supplier is NXP that offers a lot of different solutions for the automotive industry [30]. Most of their solutions uses microprocessors are implemented using different processors based on the ARM architecture and another popular architecture named POWER architecture [31].

ARM is currently the leading architecture for microprocessors in the automotive industry according to IHF [30] next to the POWER architecture.
2.2 Advanced Driver Assistance Systems

ADAS are the systems that aid the driver while driving, the main goal of these systems is to enable safer and better driving. There are many different types of ADAS aimed to assist with different tasks and difficulties [17].

The major task of ADAS is to provide the driver with different kinds of warnings in case of potential hazards. On top of providing warnings ADAS can also detect hazards and try to actively avoid them. Additionally different types of ADAS can work together to provide even better safety features [32, p. 14-15].

ADAS can be divided into three categories. These categories are: Longitudinal support systems, Lateral support systems and Driver vigilance systems. Longitudinal support systems have been around for a long time. The first longitudinal support system was introduced as early as 1995. A set of different systems can be included in the longitudinal support system category such as:

- Adaptive cruise control - A type of cruise control system that automatically sets the speed to keep a safe distance between the vehicles [33].
- Collision avoidance systems - A system to warn the driver if there is a potential risk of collision [32, p. 12].
- Precrash systems - Also known as collision avoidance system. The system is made to reduce the seriousness of a possible collision [34].
- Pedestrian Protection systems - A system made to protect the pedestrian from a possible collision with the car [32, p. 13].

Lateral support systems first came in the early 21st century. Lateral Support Systems can also be divided in different systems:

- Lane departure warning - A system developed to warn the driver when the car is leaving the lane it is currently driving in [35].
- Lane change assistance - To warn the driver that the intended lane change is not safe.
- Blind spot detection - This system warns the driver if there is a car in the blind spot of the car, i.e. the driver’s side or rear [36].

The last category is the Driver vigilance systems. These systems are developed to warn the driver when they become less alert. Also this category can be divided into different systems:
2.2. Advanced Driver Assistance Systems

- Attention or Drowsiness Assistance - Tries to capture the driver's alertness and warn the driver if it is diminishing [32, p. 12].

- Alcohol lock - To prevent drivers influenced by alcohol from starting the car [32, p. 12].

2.2.1 Sensors

There are a lot of different sensors and different types of sensors required to create the various functions of ADAS described above. Wired Magazine [19] published, in April 2015, an overview of the electronics and sensors used in ADAS. This overview can be seen in Figure 2.1.

The figure also shows a description of what each sensor is used for. The GPS is a sensor that provides a geolocation and it has been around since 1973, when it was launched in the United States [37]. Ultrasonic sensors are devices that both transmit and receive ultrasonic pulses and acquire the distance by converting the ultrasonic pulse to an electrical pulse [38]. As can be seen from the figure the sensors are used for close proximity measures and they are relatively cheap sensors. Another type of sensors, used to provide additional information together with the GPS are Odometry sensors. The sensors tries to estimate the changes of the position over time [39]. LiDaR is, as can be seen in the figure, the most expensive sensor and is used to monitor the surroundings of the car.
The distance to objects are measured by using laser. Initially the LiDaR was developed for a distance warning system back in 1996. More modern approaches using LiDar are multilayered and rotating, being able to map environments in 3D [40, p. 6]. Other important sensors are video cameras, these are also used to monitor the cars surroundings. They provide vital detailed information about surrounding environment of the vehicle. The cameras can also be used in stereo vision applications providing depth information from multiple cameras combined. Lastly there are the radar sensors also providing information about the cars surroundings. RADAR maps the environment using radio waves [41].

2.2.2 Sensor fusion

As can be derived from the discussion about the ADAS sensors above, a lot of the sensors can provide similar functionality. Using multiple sensors provide additional accuracy and certainty in the data. Using data from multiple sensors and combining them is called sensor fusion [42]. Sensor fusion in cars is currently not the norm and only exists in the high end vehicular models. Different sensors in the car can complement each other to give better accuracy. In addition to increasing the accuracy they can also create redundancy when certain types of sensors do not work well or even if they fail [43].

One example of sensors fusion, proposed by EETimes Europe Automotive, is the use of a rear view camera together with an ultrasonic distance sensor. The two systems could provide a good solution on advanced parking assistance, which would not be possible by just using one of the sensors. The camera can be used to detect objects and map the environment while the ultrasonic sensor can provide accurate distance measurement of nearby objects [43].

2.2.3 Cooperating vehicles

Cooperating vehicles is another function of ADAS to increase driver safety. Cooperating vehicle systems are refereed to as Vehicle-to-Vehicle (V2V) safety systems. The systems try to avoid collisions and minimize risks by passing information between nearby vehicles. There are cases when sensors can not detect risks and when V2V communication can provide extra information to warn the driver [32, Chapter 11].

In a collaboration between the project U.S Department of Transportation and Vehicle Safety Communications 2(VSC2) the project Vehicle Safety Communications – Applications (VSC-A) was created. The project listed the following V2V implementations:[44, Chapter 2.3]
• Emergency Electronic Brake Lights (EEBL): The system broadcasts messages that the car is currently breaking hard. Other vehicles receive the warning and warn the driver of that vehicle.

• Forward Collision Warning (FCW): Warns the driver of the vehicle about the risk of collision with another vehicle traveling in the same direction and lane.

• Lane Change Warning (LCW) and Blind Spot Warning (BSW): This system warns the driver when another vehicle is moving towards the vehicle's blind spot zone.

• Do Not Pass Warning (DNPW): A system to warn a driver currently trying to pass another car and it is not safe.

• Intersection Movement Assist (IMA): A system to warn the driver when it is unsafe to enter an intersection.

• Control Loss Warning (CLW): The system sends a broadcast message about loss of control. The receiving cars decide the relevance of the message and act upon the decision.

2.2.4 Vehicle-to-Infrastructure Communication

Vehicle crashes in intersections are the reason for a lot of fatal injuries with over 1.72 million crashes and 9000 deaths every year in the United States [32, Chapter 10.1]. It is hard for normal sensors to prevent such situations. This is one of the cases where the presence of a Vehicle-to-Infrastructure (V2I) solution can assist.

To allow the cars to communicate with the infrastructure, the communication has to be implemented using a wireless communication technology. There are a lot of different applications for V2I, not only for safety reasons but also to assist with tasks to increase efficiency and handle payments and information. Some examples of possible safety applications for V2I are listed below [45, Chapter 9].

• Intersection safety

• Rail crossing operations

• Priority assignment for emergency vehicles

• Warning for hazardous situations
As mentioned above V2I applications does not strictly have to limit to only safety applications. The list below lists some applications that could assist with traffic efficiency [45, Chapter 9].

- Traffic jam notification
- Dynamic traffic control
- Connected navigation

On top of assisting with road safety and increasing traffic efficiency V2I applications could also assist with, as mentioned earlier, handle payments and providing drivers with traffic information.

### 2.3 Autonomous cars

As ADAS become more and more advanced, cars move towards becoming completely autonomous. According to a forecast done by IHF, 21 Million autonomous cars will be sold globally in 2035. And according to IHF Automotive forecast [46]:

“The U.S. market is expected to see the earliest deployment of autonomous vehicles as it works through challenges posed by regulation, liability and consumer acceptance. Deployment in the U.S. will begin with several thousand autonomous vehicles sold in 2020, which will grow to nearly 4.5 million vehicles sold in 2035,”

This means autonomous vehicles are in the near future.

One of the most well-known and leading companies in autonomous driving is Google. They started their development in 2009 and has reached several milestones in their development towards a fully autonomous car. To get a grasp on how far Google and the autonomous driving community has come in the development, the milestones they present are listed below [47].

- 2009: The project started
- 2012: More than 300,000 miles self-driven
- 2012: Moved to complex city streets
- 2014: Designed a new prototype vehicle
- 2015: Our prototypes hit public roads
- 2015: World’s first fully self-driving ride on public roads
2.4. Communication protocols in cars

The amount of ECUs increases as cars become more advanced with more demands on safety as well as comfort. The amount of average ECUs in a car is expected to be above 40 by year 2019. It is twice as much as it was in 2010, where the number of average ECUs in a car was merely 20 [23, p. 29].

The increasing complexity also means new communication technologies have to be developed and implemented to be able to cope with all the additional data. This section will describe different protocols used in cars. The section will not discuss all protocols ever used in cars, but the most common and important ones.

2.4.1 LIN

Not all communication in a car needs to be complex and handle big data rates. Some systems in a car, such as electronic mirrors or central locking system, have low requirements and can be implemented by using simpler and less expensive technology. The primary goal of LIN is to be cost effective and is a good complement to the more advanced and more expensive protocols. LIN is a serial master-slave bus protocol. All messages sent on the bus are initiated by the bus master and the data rate is limited to 19.2 kbps [23, p. 36-37].

The first implementation using LIN was released in late 2002 and it is still used in modern cars, but as mentioned earlier, as a complement to the more advanced protocols. The protocol support up to 16 slaves, a limit set in order to guarantee deterministic timing. There is no need for any additional arbitration in the protocol as it is a master slave protocol. Another feature of LIN is that it can detect faulty nodes. The data frames are provided with a checksum to check the data as well as provide error detection. The protocol is based on standard UART/SCI hardware which is one of the reasons why it is cheap and easy to implement [48].
2.4.2 CAN

CAN was one of the first car networking technologies developed. It has been used since the early 1990s. It was originally developed by BOSCH and their first CAN controller was introduced 1987 [23, p. 31]. CAN is a message based protocol and it can handle different priority messages and arbitration between these messages is done automatically. All priorities has to be defined prior to deploying the system. The media access protocol used in CAN is Carrier Sensor Multiple Access/Collision Avoidance (CSMA/CA). CSMA/CA avoids collisions by assigning each message on the bus a priority. The priority defines the arbitration of the messages. There are dominant and recessive bits; 0 is dominant and 1 is recessive. The message winning the arbitration is the message with the lowest number, since the 0 bit is dominant over the 1 bit [49, Chapter 1].

CAN can be divided into; High-Speed CAN and Low-Speed CAN. High-speed CAN can support data rates up to 1 Mbps while Low-speed CAN only manages speeds up to 125 kbps.

CAN uses a bus to send all information. Which means that all ECUs are connected to the same bus wires. CAN has two wires: CAN High and CAN Low [49, Chapter 4].

Besides the automatic arbitration, one good feature of the CAN protocol is that it is possible to get the Worst Case Response Timing (WCRT), i.e. making the protocol deterministic. At the early stages of CAN it was not known that it was possible to get the WCRT of lower priority messages, but Tindell and Burns (York University, 1994) showed how research into fixed priority pre-emptive scheduling for single processor systems could be adapted and applied to the scheduling of CAN-messages. Due to of these facts CAN became a popular choice of protocol for the automotive industry.

2.4.3 Flexray

Flexray is a protocol developed by the Flexray consortuim. In 2006 BMW was the first automotive company to implement flexray [50]. The protocol was developed as a result of a study made by a group of companies evaluating the existing protocols. The study was done to evaluate whether or not it was possible to achieve a set of technical specifications. The study showed that it was not possible and this led to the development of a new protocol, namely Flexray [49, Chapter 6.5]

Flexray is not intended to replace protocols such as LIN or CAN it is made to meet the requirements for applications demanding high data speeds. The protocol supports speeds up to 10 MBit/s. It is a time-triggered protocol using Time
Division Multiple Access (TDMA) to allocate the media in time slots, typically between 1-5 milliseconds. The time slots are divided into segments. The picture in Figure 2.2 describes the different segments.

![Flex ray segments](image)

Figure 2.2: Flex ray segments, taken from National Instruments [2].

The static segment, the blue colored segment, is used for deterministic data. Data arriving at a fixed period in time. The dynamic segment, the yellow colored segment, is similar to CAN it handles event-triggered data that does not occur at a regular time interval. The Symbol Window is mainly used for maintenance. The segment called network Idle Time is a fixed time to let the ECUs handle any adjustments needed to be done from the previous cycle [51].

2.4.4 Ethernet

Ethernet is the latest and currently the most modern underlying communication technology used in the automotive industry. The most important motivation for moving toward Ethernet is the need for a communication technology providing high bandwidth to cope with the increasing demand of data. Ethernet is a mature communication technology providing high bandwidth. Due to Ethernet being a mature protocol it is also cheap to implement [52]. An implementation of automotive Ethernet is AVB/TSN. Before going in to the rather complex protocols related to AVB/TSN it is important to know some basics about traditional Ethernet, in order to understand how it can be implemented for automotive purposes.

Ethernet has been around for a long time. It was standardized in the early 1980s. The latest and highest data rates recorded using Ethernet reaches up to 100 gigabits per second [53], which is 100 000 times faster than CAN-bus.
2.4. COMMUNICATION PROTOCOLS IN CARS

Early implementations of Ethernet used CSMA/CD to access the shared media. It introduced an uncertainty in the protocols with respect to timing. However, newer packet switched networks do not need CSMA/CD.

2.4.4.1 CSMA/CD

CSMA/CD is the media access protocol used in early Ethernet. In contrary to many other media access protocols CSMA/CD allows collisions to happen on the shared media. Instead of preventing the collision, CSMA/CD has the ability to detect if a collision has occurred. An overview of how it handles collisions is discussed in this section. The following steps, presented below, are taken in order to handle collisions in CSMA/CD [54, Chapter 3.3]:

1. Check if the media is available
2. If it is not, wait for the media to become available
3. If it is available, start transmission
4. If there is a collision during transmission, handle the collision, otherwise continue transmission until it is finished
5. Handle collision
   (a) Try to transmit during a jamming period, count the number of tries
   (b) If the number of tries is significant abort the transmission
   (c) If the number of tries is not significant, back off according to an exponential backoff algorithm

During the jamming period the CSMA/CD counts the number of tries it took to send the data, if the number is significant (over 16) the transmission is aborted. The BEB is an algorithm calculating the back off time (the estimated waiting time before starting re-transmission of the data) [54, Chapter 3.3].

As mentioned earlier the CSMA/CD protocol is not needed anymore. This is because in a packet switched network the only contention on the media that can happen is when two nodes in the network try to send to each other. However, when operating in full duplex there can not be any contention, because when full duplex is active there are separate transmit and receive channels [55, Chapter 4.1].
2.4.4.2 Loop-prevention

Loops in switched networks can occur when there are more than one path between two endpoints, or when there is a connection on the same switch to another port. The loops create so called broadcast storms. A broadcast storm works the following way; each time a broadcast message is received on one port it is rebroadcast on all its other ports. Figure 2.3 gives an example of a broadcast storm causing a loop.

![Figure 2.3: Example of a broadcast storm, from [3, Chapter 8]](image)

A broadcast message is sent to Switch A. When Switch A receives the broadcast message it broadcasts the message on all its ports, sending it to both Switch B and Switch C. Switch B and Switch C will send the message on all other ports, sending the message to one another. The behavior continues and causes a broadcast storm.

The loops will continue forever and consume the entire bandwidth [56]. However there are ways to handle loops in networks. The previously conducted way is to use the Spanning Tree Protocol (STP) and the newer, more modern approach, is to use Shortest Path Bridging Protocol (SPB).

The STP protocol sends data on the network to try and find out the organization of the switches. Once the organization of the topology is known the protocol blocks ports in order to create a loop free topology.

The SPB Protocol is aimed to replace STP. It is defined under IEEE 802.1aq [57]. The protocol provides logical Ethernet networks on top of native Ethernet networks. There are two different flavors of SPB: Shortest Path Bridging-VID (SPBV) and Shortest Path Bridging-MAC (SPBM) [58]. SPBV is backwards compatible with STP technology. The “VID” in Shortest Path Bridging-VID stands for VLAN identifier. Each individual VLAN following
SPBV uses SPT. A shortest Path VLAN Identifier (SPVID) is assigned to each individual SPT set. This identifier is appended to frames crossing VLANS [58]. The other type of SPB, SPBM, supports up to a 1000 bridges and it is part of the 802.1aq standard. The difference from STP is that this allows to use all physical connectivity. By introducing a control plane which has a global view of the network topology [59]. The protocol appends the information about the global view in the Ethernet header to avoid loops.

2.4.4.3 AVB/TSN

Audio Video Bridging (AVB) is an improvement of standard Ethernet to add support for real-time video and control. It is a set of standards developed by the Audio Video Bridging Task Group. Time Sensitive Networking is extending the functionality of AVB and the name of the new task group in charge of the development is TSN [60]. AVB in the automotive industry have several use cases. It can provide perfect synchronization in various multimedia devices used in cars. Delivering sound and video with perfect synchronization on separate channels. It can synchronize multiple cameras or other sensors requiring high bandwidth, suitable for different applications of ADAS. It can also provide control interfaces with redundancy [61][62]. AVB delivers a set of standards:

- **802.1AS - Timing and Synchronization**: Specifications to ensure timing synchronization for time-sensitive applications [63].

- **802.1Qat - Stream Reservation Protocol**: Specifies network resources to be reserved for specific traffic streams. Allows for end-to-end management of QoS guaranteed streams [64].

- **802.1Qav - Forwarding and Queuing Enhancements for Time-Sensitive Streams**: This standard allows bridges to provide guarantees for time-sensitive (i.e. bounded latency and delivery variation), loss-sensitive real-time audio video (AV) data transmission (AV traffic). It specifies per priority ingress metering, priority regeneration, and timing-aware queue draining algorithms. This standard uses the timing derived from IEEE 802.1AS. Virtual Local Area Network (VLAN) tag encoded priority values are allocated, in aggregate, to segregate frames among controlled and non-controlled queues, allowing simultaneous support of both AV traffic and other bridged traffic over and between wired and wireless Local Area Networks (LANs) [65].
2.4. COMMUNICATION PROTOCOLS IN CARS

- **802.1BA - Audio Video Bridging (AVB) Systems**: This standard specifies information that LAN equipment manufacturers can use to develop AVB-compatible LAN components [66].

- **IEEE 1722 “AVB Transport Protocol (AVBTP)”**: This standard specifies the protocol, data encapsulations, and presentation time procedures used to ensure interoperability between audio- and video-based end stations that use standard networking services provided by all IEEE 802 networks meeting quality-of-service requirements for time-sensitive applications [67].

- **IEEE 1722.1 “Audio Video Discovery, Enumeration, Connection Management and Control (AVDECC)”**: This standard specifies the protocol, device discovery, connection management, and device control procedures used to facilitate interoperability between audio and video based End Stations [68].

Not all standards are relevant for automotive purposes. The following part will discuss the relevance, with regard to automotive use, of the automotive related protocols presented above.

**802.1AS** is used to synchronize nodes in an AVB network with a common reference time. One node in the network will be selected as the grandmaster, it is the node whose clock all other nodes in the network will follow. The grandmaster can be preselected or autoselected with the help of an algorithm implemented in the protocol. In order to achieve synchronization in the network all the nodes need to know the propagation delay to its AVB neighbours. This is done using packets called “pDelay” measurements.

In automotive use cases the Grandmaster is preselected and since most in-car networks have a static topology and layout the pDelay timings are almost static, a fact that can shorten the start up period of the protocol [23, p. 145-147].

**802.1Qat** allows allocating bandwidth for individual traffic streams within the AVB network. The idea is that talkers and listeners in the network announce the streaming data they require to send/receive. Due to this the switches in the network can evaluate the availability of the needed bandwidth. If the bandwidth is available the bandwidth is reserved and if it is not the request is denied. By default 75 percent of the total bandwidth can be reserved, the rest is allocated for non AVB traffic. There are two AVB traffic classes defined in 802.1Qat: The highest priority “Class A” and the lower priority traffic “Class B”. These traffic classes are used together with the non AVB traffic running simultaneously. In car networks a denial of a reservation requests is unacceptable. To avoid this it is important that all transmission rates are known before system deployment. In
addition, due to the known transmission rates the start up time can be reduced [23, p. 147-149].

**802.1Qav** is used to improve the quality of all AV transmissions in the AVB network. This is done by ensuring that all packets are evenly distributed over time. It is important that the bandwidth is not used all at once and then idling, instead of evenly distribute the load. This is a trivial task for the talkers; the different traffic classes are set to send packets at different frequencies. Class A sends every 125 microseconds and Class B sends every 250 microseconds. However, to achieve this in the switches is a complex task. This is achieved using a credit-based shaper inside the AVB switches. An overview of the algorithm can be seen in Figure 2.4.

![Credit-based shaper algorithm](image)

Figure 2.4: Credit-based shaper algorithm, from [4]

If the port is busy credit is collected at the rate of idleSlope which is equal to the reserved bandwidth of the traffic class the data belongs to. As soon as the port is available and the credit is equal or larger than 0 the priority packet is transmitted. Credit is reduced at the rate of sendSlope, which is equal to the rate of the link minus the idle rate. If the credit is still above or equal to 0, the next packet in the queue is sent. If the queue is empty the credit is set back to 0.

The main concern for 801.2Qav is the need to be able to receive safety critical data immediately. There is no simple solution to this and it is ongoing development trying to find a good solution. There are a number of suggestions trying to address this problem. One example is to introduce Time Aware Shaping, which tries to introduce a TDM channel in the network. The shaper blocks other traffic during
specific timing windows to guarantee that control data can be sent over the link [23, p. 149-153].

2.4.4.4 802.1Q VLAN

802.1Q VLAN is a protocol that enables Virtual Lan (VLAN) on Ethernet. Several VLANs can be set up on a single physical LAN. There are several reasons why to use VLANs in vehicles. Some of the major reasons will be discussed briefly.

**Performance:** By minimizing broadcast domains unnecessary broadcast data can be limited to a VLAN, instead of being broadcasted on the entire network [69]. In addition to this VLANs also provide QoS by letting users assign priorities to the VLAN header (described later) [70].

**Security:** VLANs provide security by preventing devices from listening to other VLANs. Prevention is done by assigning devices to VLANs. This is done in two common ways: One way is port-based, where a switch port is manually configured to be a member of a VLAN. Another way is MAC-based, where membership of the VLAN is determined by the MAC address of the device [71].

**Manageability:** VLANs provide an easy way to manage large networks. It provides an easy way to configure logical groups in changing networks [72].

VLAN inserts a VLAN tag into an Ethernet frame. The tag is a 32-bit field. The field is composed of several fields that can be seen in Table 2.1. The table is composed from information found at [73].

<table>
<thead>
<tr>
<th>16 bits</th>
<th>3 bits</th>
<th>1 Bit</th>
<th>12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPID</td>
<td>PRIORITY</td>
<td>CFI</td>
<td>VID</td>
</tr>
</tbody>
</table>

The TPID-field, Tag Protocol Identifier, is used to identify the frame as an IEEE 802.1Q-tagged frame. The PRIORITY-field is used to set the priority frame to be used to implement QoS as discussed above. The 3-bit PRIORITY field can accommodate 8 different priorities (0 - 7). The CFI-field, Canonical Format Indicator, determines whether the frame is represented in canonical format or not. The VID-field, VLAN Identifier, is used to identify which VLAN the frame belongs to.

VLANs can also be used in automotive networks to isolate the broadcast traffic to specific domains [23, p. 153-154].
2.5 The real-time concept

This section will briefly discuss the concept of real-time. Real-time can mean different things depending on the context. When talking about real-time in computer science one refers to meeting deadlines and making the system behave in a predictable manner. Real-time systems tasks can be divided into three categories. Tasks that can tolerate some missed deadlines without causing serious damage are called Soft Real-time tasks. A task that if it misses its deadline and does not cause serious damage but makes the results useless, is called a firm task. A task is called hard if missing the deadline will cause catastrophic consequences [74].

There are a lot of concepts that is important to Real-time theory, to evaluate real-time performance. The list below introduces some terms related to real-time theory [75, p. 257-259].

- **Deadline**: The deadline a time that a task has to finish before.
- **Latency**: The time it takes between when something should happen and when it actually does happen.
- **Jitter**: The variance of the latency.
- **Predictability**: Knowing in advance how long time an operation will take
- **Worst case**: The worst possible execution time
- **Priority Inversion**: When a lower priority task is blocking a higher priority task from running.
- **Periodic task**: A task that runs at regular intervals.

2.6 Operating systems

This section will present commonly used operating systems in automotive vehicles. Additionally this section will describe a choice becoming more popular in the embedded world, well suited for prototyping - Embedded Linux.

There are different subsystems in cars, and therefore different types of operating systems are used. One system used in cars is infotainment systems, that consists of tasks with soft real-time constraints. There are three major operating systems when it comes to infotainment [76]:

- QNX
2.6. OPERATING SYSTEMS

• Microsoft Embedded Windows

• Different versions of Linux

QNX is said to have over 50% of the market share, Windows roughly 20%. Linux is rapidly growing and is to grow continuously [76].

QNX is a UNIX-based RTOS intended for the embedded systems market. The product development started back in the 1980s and the company was bought by BlackBerry in 2010 [77].

Microsoft Embedded windows has different families, targeting different embedded devices. Windows Embedded Automotive is one embedded operating system targeted towards embedded systems, in particular the automotive industry [78][79].

An example of an infotainment Linux operating system is Automotive grade Linux [80].

Patrick Shelly from Mentor Graphics Corporation describes in one of his presentations the current trends of OS in cars. The presentation presents a number of different operating systems used for cars, and he mentions the use of Linux when prototyping for ADAS. He states that the trend for under hood electronics and ADAS is moving towards systems following the Autosar standard.

2.6.1 AUTOSAR

AUTomotive Open System ARchitecture (AUTOSAR) is a standard developed by a number of partners and the organization was founded in 2003. Among the partners are several of the leading automotive companies including BMW, Volkswagen, Ford, Toyota, General Motors and more [81].

AUTOSAR provides a number of concepts and architectural specifications. One thing included is the ECU Software Architecture. The architecture is divided in three different layers [82]:

• AUTOSAR Software

• AUTOSAR Runtime Environment

• AUTOSAR Basic software

AUTOSAR provides a good overview of the layers that can be seen in Figure 2.5.

The AUTOSAR software layer consists of AUTOSAR Software Components. The software components are self contained units of software [83].

The AUTOSAR Runtime Environment is middleware that provides a communication abstraction to the AUTOSAR Software Components. The middleware allows the Software Components to communicate to other ECUs as well as services provided
2.6. OPERATING SYSTEMS

Figure 2.5: An overview of AUTOSAR ECU Software Architecture, take from [5]

by the lower abstraction layer. All this is abstracted by the middleware and it provides a generic interface for communication, no matter if the communication is to another ECU or to services provided by the same ECU [84].

AUTOSAR Basic Software provides the lowest abstraction layer. Interfacing against the ECU hardware and providing necessary functions to run the functional parts of of the software. This part is responsible for the Intercommunication, Services, Operating System and Micro Controller abstraction. AUTOSAR provides requirements on the Operating System and has the following stated on their website [85]:

“AUTOSAR allows the inclusion of proprietary OSs in Basic Software components. To make the interfaces of these components AUTOSAR compliant, the proprietary
2.6. OPERATING SYSTEMS

OS must be abstracted to an AUTOSAR OS. The standard OSEK OS (ISO 17356-3) is used as the basis for the AUTOSAR OS. ”

2.6.2 Embedded Linux

There is no specific OS called Embedded Linux, when talking about Embedded Linux it is referred as mainline Linux applied to an Embedded system [86, Chapter 3]. Linux is originally developed as a GPOS but it has grown into becoming much more than that, due to the large community and its demands. An important feature that has been developed during the past years is the Linux Scheduler. The default scheduler used in a desktop system is the Completely Fair Scheduler, it divides the processing time evenly among the processes. Another scheduler available, besides the default Linux scheduler, is the Linux Real-time Scheduler. To understand what the real-time scheduler is and why it is needed it is important to understand and define what real-time is, this was explained in Chapter 2.5.

2.6.2.1 Linux Real-time Scheduler

Running the default scheduler is not optimal in terms of real-time, it is not made to handle hard or even soft real-time tasks. However, the Linux real-time scheduler enables higher priority tasks to take over execution from lower priority tasks. The scheduler has two different policies: FIFO and RR. FIFO is a simple scheduler, when a task with higher priority becomes available it will take over execution from the lower priority tasks. RR is a similar scheduling policy with one difference: tasks always runs for a specific period of time before being put at the back of the line. Even the Linux real-time scheduler is not made to handle hard real-time tasks, as it is impossible to guarantee deadlines and worst case response times [75, Chapter 12].

2.6.2.2 PREEMPT_RT Patch

CONFIG_PREEMPT_RT is the current project developing Linux into a real-time system. One part in turning the Linux system suitable for real-time is making it more preemptible. The PREEMPT_RT patch turns Linux into a fully preemptible kernel, allowing even the kernel threads to be preempted [87]. The patch also provides the kernel with a real-time scheduler. It balances tasks across CPUs while still taking real-time tasks in consideration - in contrary to the standard Linux scheduler. The real-time scheduler uses a push-pull strategy to handle scheduling tasks with different priority level [88].
2.7 Related work

The MOPED platform is a platform developed as a research and educational platform for Cyber-Physical Systems. The platform targets automotive systems and provides an architecture containing both software and hardware. The platform is running on a scale 1:10 model car equipped with actuators and sensors. The platform has multiple ECUs connected through an Ethernet hub. Parts of this paper is reused for this thesis. Similar parts of the model car has been used in the development of the platform presented in this thesis [89].
Chapter 3

Hardware and software tools

There are several hardware related components used in the realization of this thesis. In addition to the hardware multiple software tools have been used throughout the work process. This chapter will start by introducing the hardware that has been used. Followed by the Operating Systems used on the ECUs. The last part of this chapter describes the software tools.

3.1 Hardware

This section will present the hardware used in this thesis. Each hardware component will be discussed in detail. Why it was chosen and how it is used in the architecture of the platform will be discussed in the next chapter, Chapter 4.

3.1.1 BeagleBone Black

The BeagleBone Black is one type of ECU used in the platform. It is a credit card sized one card computer and development platform. An image of the BeagleBone Black can be seen in Figure 3.1.

The board is developed by the BeagleBoard.org foundation and is a non-profit organization settled in the US, which provides open source hardware as well as open source software. The foundation hosts a community driven support forum, where users can discuss and provide solutions to problems related to the platforms.

All the hardware on the BeagleBone Black is open source as well as all the software available to the users. The SoC of the platform is a Sitara AM3358 chipset from Texas Instrument [90].
3.1.1.1 Sitara AM3358 chipset

The main processing unit on the chip is a single core ARM Cortex-A8 microprocessor. The microprocessor is a superscalar processor with a RISC architecture. It can issue two instructions per clock cycle with a deep pipeline. The clock is rated at 1 GHz but can be scaled up to 1.35 GHz and scaled down to 300 MHz to save power. The microprocessor has a 32KB L1 instruction cache and 32KB data cache. It has a 256 KB L2 cache with error correcting code. It also has 176KB of On-Chip Boot ROM and 64 KB of On-Chip dedicated RAM. There is a JTAG interface on the chip, which makes it possible to interact with and debug the core at run time. The chipset also comes with an SIMD co-processor. The co-processor is an Arm NEON general purpose SIMD processor to help with signal processing algorithms to enhance the encoding and decoding of multimedia.

The main memory is a DDR3 memory with a total of 1GB address space and it is clocked at 800 MHz. The memory has a 16 bit wide data bus.

To help with real time tasks the chipset has a Real-Time clock with an internal 32 KHz oscillator. The oscillator can be used to generate internal interrupts.

There are multiple peripherals on the chip. It comes with USB 2.0 ports, Industrial Gigabit Ethernet MACs, CAN ports, Multichannel serial audio ports, UART interfaces, SPI serial interfaces, IX2C interfaces, Interface for 8-bit MMC and SD memory, access to general purpose I/O pins, multiple general purpose timer and a watchdog timer, an LCD controller, ECAP modules to help generate PWM outputs.

Additionally the AM3358 chipset has two on-chip micro controllers. The two micro controllers are inside a subsystem called the Programmable Real-Time
3.1. HARDWARE

Unit Subsystem and Industrial Communication SubSystem (PRU-ICSS), and each micro controller is called Programmable Real-Time Unit (PRU) [91].

3.1.1.1.1 Programmable Real Time Unit

The two PRUs on the chip are clocked at 200 MHz and they are two identical microcontrollers. The microcontrollers have a RISC architecture with no pipeline, IE it can only issue and execute one instruction per clock cycle. Each PRU has 8KB of instruction RAM and 8KB of data RAM each memory with Single-error detection. The two PRUs also has a 12 KB shared RAM memory. The PRU has access to a local interconnect bus to connect to the peripherals inside the PRU-ICSS. The peripherals inside the PRU-ICSS is one UART port, one eCAP module, two Ethernet ports that support Industrial Ethernet and one Management Data Input/Output (MDIO) port.

To grant the PRUs access the rest of the system they have an interrupt controller to handle system events. It handles interrupts such as interrupts from the ARM core, interrupts from peripherals or interrupts from the other PRU.

The PRUs can access and change the state of an output pin in one clock cycle. To load the code onto the the PRU the code has to be transferred onto the PRU from the ARM environment. This is done through Linux device drivers [91].

3.1.1.1.2 Programming the Programmable Real Time Unit

To program the PRU a set of steps need to be taken. The code to be run on the PRU has to be loaded from the ARM environment. This process includes a set of Linux drivers and a specific compiler targeting the PRU architecture. The compiler used to compile the source files into the binaries will be described in the Software tools Chapter 3.3.5.

The Linux driver that handles the PRU to ARM communication is the Linux Remoteproc driver. This driver is a Linux core driver run in kernel space and it provides an API for the specific PRU kernel modules. The kernel PRU modules are called pruss module and pru_rproc module, which are loaded from user space as any Linux module, using the Linux tool “insmod”.

The specific software binaries to be run on the PRU are placed in /lib/firmware with the specific names; am335x-pru0-fw and am335x-pru1-fw. The module pru_rproc looks in the folder for the two specific files. In each binary file the module looks for sections describing the resources and allocates the resources.
The module also creates VRINGS in the DDR memory [92]. VRINGS will be described later in 3.1.1.1.3.

The next step is for the pru_rproc driver to load the binaries on to the PRU program memory as well as giving it instructions about the resources. The last step for the pru_rproc is to give the PRUs instructions to begin execution. Texas Instrument provides a good overview picture describing this. It can be seen in Figure 3.2.

Figure 3.2: Programming the PRU, image taken from [7]

3.1.1.1.3 Communication between ARM and PRU

The PRU is a separate system on the same chip as the ARM core and it is necessary for the two to be able to communicate. The communication between the two systems is implemented using shared memory. There are several functions and pieces of software connected to passing messages between the systems.

The mechanism used to pass messages is called RPMsg. RPMsg communicates with the shared memory that is in the DDR memory of the ARM core. There are four virtual memory areas to allow communication between the systems. These areas are specific memory structures called VRINGS and are divided in two parts: used buffers and available buffers. The four VRINGS are divided as follows: two for each PRU, each PRU has one VRING for host and one for the slave (PRU)
If the ARM sends a message to the PRU it takes a used buffer, fills it with data and puts it in the available buffers. The PRU then reads the data in the available buffers and moves the emptied buffer in to the used buffers again [92].

In the case where the PRU sends the ARM a message it takes a buffer from the available buffer, fills it with data and then places it in the used buffers. The ARM takes the buffer from the used buffers, reads the data and places it in the available buffers [92].

In order for the systems to know that new messages are available in the VRINGS there is a mechanism implemented called mailboxes. The mailboxes works as interrupts between the systems. If one system wants to send the other system new data it notifies the mailbox, which in its turn will ”kick” the other system, to let it know that there is an incoming message [92].

There is a module used in kernel space that communicates with the RPMsg mechanism that allows for user space communication. The module is called rpmsg_pru and it communicates to user space using a character device. If the user wants to send a message to the PRU it writes to the character device. The character device file can be read to get messages from the PRU [92]. Again, Texas Instrument provides an overview of how the message passing works, this can be seen in Figure 3.3.

3.1.2 Peripherals
There are 65 GPIO pins available on the BeagleBone Black. The board is supplied with 5 Volts but all the GPIO pins operate on 3.3 Volts. Seven of the pins can be used as an ADC with a voltage limit of 1.6 Volts. It has access to an SPI controller as well as an I2C controller. The board has a CAN controller that can be accessed through an expansion connector. It is possible also to access four timers and four serial ports through the I/O pins. The Beaglebone Black comes with an SD/MMC Connector that supports microSD cards. It is possible to connect a display through the miniHDMI port. To connect external peripherals there is two USB ports and one miniUSB port to power and communicate to the board. The Table below shows the BeagleBone Black specification [93].
3.1. HARDWARE

Figure 3.3: PRU-ARM message passing, taken from [8]

Table 3.1: BeagleBone Black hardware specification

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>AM335x 1GHz ARM® Cortex-A8</td>
</tr>
<tr>
<td>GPU</td>
<td>SGX530 3D, 20M Polygons/S</td>
</tr>
<tr>
<td>DRAM</td>
<td>512MB DDR3 RAM</td>
</tr>
<tr>
<td>Onboard Flash</td>
<td>4GB 8-bit eMMC</td>
</tr>
<tr>
<td>SD/MMC Connector</td>
<td>MicroSD 3.3V</td>
</tr>
<tr>
<td>Video Out</td>
<td>HDMI</td>
</tr>
<tr>
<td>Audio</td>
<td>Via HDMI, Stereo</td>
</tr>
<tr>
<td>Ethernet</td>
<td>10/100, RJ45</td>
</tr>
<tr>
<td>Expansion connectors</td>
<td>McASP0, SPII, I2C, GPIO (69 max), LCD, GPMC, MMC1, MMC2, 7 AIN, 4 timers, 4 serial ports, CAN0, EHRPWM(0,2), XDMA Interrupt, Power Button, Expansion Board ID</td>
</tr>
<tr>
<td>PRU</td>
<td>2x PRU 32-bit microcontrollers @ 200 MHz</td>
</tr>
</tbody>
</table>
3.1. HARDWARE

3.1.3 Device Tree

Embedded Linux on the BeagleBone does not use a bios to boot and configure the device. Instead it uses files on the eMMC or an inserted SD card to set up the system. Previous distributions of Embedded Linux on the BeagleBone had specific code inside the Linux kernel to handle the specific device, but this meant having lots of extra code added to mainline Linux. Instead of using source code files to setup the devices newer ARM boards and kernels use Flattened Device Trees.

Flattened Device Trees is a data structure to describe the hardware of the system. To change the hardware set up a device source tree file has to be compiled and put it in a specific directory. In order for the changes to take effect the device has to be rebooted.

A big drawback with this setup is the need to reboot after each change which makes run time configuration difficult [86, Chapter 6].

Fortunately this drawback has been solved by using something called Device Tree Overlays along with a tool called CapeManager. A utility that does this automatically will be described later, in Chapter 3.3.3.

3.1.4 Serial to USB interface - SparkFun FTDI Basic Breakout

A serial to USB interface is used to communicate to the BeagleBone Black’s ARM core on the serial pins. The device converts the serial data from the pins to serial data on the USB port.

It is sometimes needed to access these pins when compiling and testing different kernels of the BBB. The serial pins allow the user to get the output of the kernel without having access to SSH or other means of communicating with the kernel. The converter used is a SparkFun FTDI Basic Breakout, it uses 6 pins and provides a miniUSB interface to hook it up to a computer [94].

3.1.5 Raspberry Pi 3 model B

One of the ECUs used in the platform is the Raspberry Pi 3 Model B and is the third generation of Raspberry Pi. Figure 3.4 provides an image of the Raspberry Pi 3 Model B board.

Its performance has increased drastically over the years, while keeping its small size and low price. The newest generation has a quad core 1.2 GHz 64-bit ARMv8 CPU and it has built in WiFi. It also has support for Bluetooth 4.1 and BLE. On top of the powerful ARM-core it also has a dedicated 3D graphics
3.1. HARDWARE

Figure 3.4: The Raspberry Pi 3 Model B board, taken from [9]

core. There is 40 available GPIO pins as well as dedicated ports for a camera and display. The table below lists the Raspberry Pi 3 Model B specification [95].

Table 3.2: Raspberry Pi 3 Model B hardware specification

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>4× ARM Cortex-A53, 1.2GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>Broadcom VideoCore IV</td>
</tr>
<tr>
<td>DRAM</td>
<td>1GB LPDDR2 (900 MHz)</td>
</tr>
<tr>
<td>Storage</td>
<td>microSD</td>
</tr>
<tr>
<td>Video Out</td>
<td>HDMI</td>
</tr>
<tr>
<td>Audio</td>
<td>3.5mm analogue audio-video jack</td>
</tr>
<tr>
<td>Networking</td>
<td>10/100 Ethernet, 2.4GHz 802.11n wireless</td>
</tr>
<tr>
<td>Ports</td>
<td>4× USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI)</td>
</tr>
</tbody>
</table>

3.1.5.1 RPI dedicated camera

The Raspberry Pi 3 Model B has a built in connector for a dedicated camera. The camera used is The Raspberry Pi Camera Module v2, seen in Figure 3.5.
The camera can capture video 1080p in 30 FPS or 720p in 60 FPS. The camera sensor is a Sony IMX219 8-megapixel sensor [96].
3.1.6 TP-link TL-SG105E Network Switch

To allow the ECUs to communicate with each other a network switch is used. This section will describe the network switch used in this project. The TP-link TL-SG105E is a five port network switch. An image of the actual switch can be seen in Figure 3.6.

![TP-link TL-SG105E network switch](image)

Each port is an RJ45 port support 10/100/1000 Mbps. The switch supports network monitoring, traffic prioritization, QoS and VLANs. The switch is managed through a web-based user interface. It supports Ethernet standards such as 801.2p and 801.2q. The switch also has features to prevent loops in the network. The switch has multiple options to enable QoS. One way is to port-based QoS where each port can be assigned four different priorities and the switch will move all packets of the port to the according priority queue. Another way of implementing QoS is the 801.2P based solution. The switch with a set up VLAN will look at each 801.2Q frame and assign a priority to the packet, based on the priority found in the frame. Below is the priority assignments described [97]:

- Priority 1 and 2 are assigned to the 1 (Lowest) priority queue.
- Priority 0 and 3 are assigned to the 2 (Normal) priority queue.
- Priority 4 and 5 are assigned to the 3 (Medium) priority queue.
- Priority 6 and 7 are assigned to the 4 (Highest) priority queue.

It is not stated by TP-link why the priorities are not assigned continuous. Below, in Table 3.3, is a table which explains each priority level defined in the 802.1D standard [98].

Another important feature, as mentioned earlier, of the TP link switch is the function that enables loop prevention. The loop protection blocks the port if a loop is detected [97]. It is not explicitly stated which protocol it uses to prevent loops. Most likely it uses SPT, described in Chapter 2.4.4.2, or SBTV, described in the same chapter, since it states that it blocks the port.
Table 3.3: Priorities according to traffic type

<table>
<thead>
<tr>
<th>Value</th>
<th>Traffic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Best Effort</td>
</tr>
<tr>
<td>1</td>
<td>Background</td>
</tr>
<tr>
<td>2</td>
<td>Spare</td>
</tr>
<tr>
<td>3</td>
<td>Excellent Effort</td>
</tr>
<tr>
<td>4</td>
<td>Controlled load</td>
</tr>
<tr>
<td>5</td>
<td>Video</td>
</tr>
<tr>
<td>6</td>
<td>Voice</td>
</tr>
<tr>
<td>7</td>
<td>Network control</td>
</tr>
</tbody>
</table>

3.1.7 Model Car

The model car is built from several different components put together to mimic a real car. The chassis of the car is a Turnigy SCT 2WD 1/10 Brushless Short Course Truck (KIT) upgraded version, is a 1:10 scale car with a length of 530 mm and a width (between the wheels) of 330 mm. The chassis of the model car can be seen in Figure 3.7.

Figure 3.7: Turnigy SCT 2WD 1/10 Brushless Short Course Truck (KIT) upgraded version, taken from [12]

The steering servo, to turn the wheels, is a HobbyKing™ High Torque Analog Servo Waterproof 4.5kg / 0.13sec / 40g. The torque of the servo is 4 kg.cm and its turn rate is 0.16sec/60deg. How a servo is controlled is described in 3.1.7.1. The motor in the car is a Turnigy TrackStar 17.5T Sensored Brushless Motor 2270KV. It is a sensored brushless motor rated at 2270kv. To run the brushless motor a specific controller is needed, called Electronic Speed Controller (ESC). The ESC used in the model car is a HobbyKing® ™ X-Car 45A Brushless Car ESC (sensored/sensorless), can supply a coninous current of 45A and is compatible
3.1. HARDWARE

with sensored brushless motors. How a brushless motor and an ESC work is described further in Chapter 3.1.7.3.

To supply the car with power a ZIPPY 4000mAh 2S1P 30C Hardcase Pack battery is used. It is a 2-cell LiPo battery with a high discharge rate. To supply the on board electronics with low volt power the car needs a voltage regulator. The voltage regulator used is a TURNIGY 8-15A UBEC for Lipoly it can supply both five and six volts channels each with a maximum draw of eight amperes on each channel. The car also needs special devices for charging the LiPo battery and for tuning the ESC. To charge the battery a Turnigy Accucell-6 50W 6A Balancer/Charger LiHV Capable is used and for tuning and configuring the ESC a HobbyKing® ™ X-Car Beast series LCD Program Card is used.

The table below, Table 3.4, gives a summary of the components used to build the model car.

Table 3.4: Components used to build the model car

<table>
<thead>
<tr>
<th>Description</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car chassis</td>
<td>Turnigy SCT 2WD 1/10 Brushless Short Course Truck (KIT) upgraded version</td>
</tr>
<tr>
<td>Steering servo</td>
<td>HobbyKing™ High Torque Analog Servo Waterproof 4.5kg / 0.13sec / 40g</td>
</tr>
<tr>
<td>Motor</td>
<td>Turnigy TrackStar 17.5T Sensored Brushless Motor 2270KV</td>
</tr>
<tr>
<td>ESC</td>
<td>HobbyKing® ™ X-Car 45A Brushless Car ESC (sensored/sensorless)</td>
</tr>
<tr>
<td>Battery</td>
<td>ZIPPY 4000mAh 2S1P 30C Hardcase Pack</td>
</tr>
<tr>
<td>Voltage regulator</td>
<td>TURNIGY 8-15A UBEC for Lipoly</td>
</tr>
<tr>
<td>Battery Charger</td>
<td>Turnigy Accucell-6 50W 6A Balancer/Charger LiHV Capable</td>
</tr>
<tr>
<td>ESC programmer</td>
<td>HobbyKing® ™ X-Car Beast series LCD Program Card</td>
</tr>
</tbody>
</table>
3.1. HARDWARE

3.1.7.1 Servo motor

A servo motor is a motor that when given a PWM pulse will move to a specific angle based on the shape of the pulse. The angle usually goes from 0 to 180 degrees. A servo is in most cases controlled using a PWM signal. There are two important factors when talking about PWM signals: the pulse width and the duty cycle. The pulse width is the width of the pulse and the duty cycle is the time cycle from the start of one pulse to the start of the next pulse.

Normally the duty cycle, when talking about servo control, is set to 20 milliseconds. The width of the pulse defines the angle, the common case is that a width of 1.5 milliseconds sets the motor at 90 degrees, 1 millisecond at 0 degrees and 2 milliseconds sets it at 180 degrees. Figure 3.8 shows an image describing how the PWM pulse works and how it controls a servo.

![Image showing PWM pulse and servo motor](image)

**Figure 3.8: Controlling a servo motor**

3.1.7.2 Brushless motor

To understand how a brushless motor works it is easiest to compare it to a normal brushed DC motor. In a normal brushed DC motor the coil is surrounded by magnets and the coil changes the direction of the current to rotate. A brushless motor on the other hand have multiple coils and rotates by changing which of the coils have current running through them. This is controlled with a specific controller or in specific software [99].
3.1. HARDWARE

3.1.7.3 ESC
Since the brushless motor can not be powered by applying DC-current a specific controller is needed. One way of controlling a brushless motor is using an Electronic Speed Controller (ESC). An ESC is essentially a driver circuit to drive the motor. ESC driving brushless motors normally takes a servo pulse as the input, and outputs power to the motor over a three-wire interface [100].

3.1.8 Android device
An android device is used to control and steer the car. It is connected through WiFi to the Raspberry Pi. The Application running on the device will be described in detail in the next chapter. Android is a mobile operating system that is developed by Google. The operating system is based on a Linux kernel and it is mainly designed to target mobile devices such as phones and tablets [101]. The android devices used in this thesis is both emulated devices supplied by Android studio (discussed in Chapter 3.3.2) and a One Plus One mobile phone, running android 6.0.

3.1.9 Sensors
The model car is equipped with multiple sensors, in addition to the camera already described.

3.1.9.1 Ultra Sonic Sensor, HC-SR04
The HC-SR04 is a distance sensor using ultra sonic waves to detect objects. It supports detecting objects in a range from 2 cm up to 400 cm, with a resolution of 0.3 cm. Figure 3.9 shows an image of the sensor used.

It is recommended to send pulses to the sensor in a frequency lower than 40 Hertz, in order for the echo pulse to settle. The sensors automatically handles undetected pulses. If a pulse is too long or never comes the sensor will end the pulse equal four meters. The sound is spread in a cone with an angle of 30 degrees. The user has to send a pulse to the sensor that is 10 microseconds long. The sensors reacts to the 10 microsecond pulse and sends an 8 cycle burst at 40 kHz. An incoming pulse equal to the time the pulse traveled will be output from the sensor. The user has to interpret the pulse and measure it in order to get the distance.

The sensor has four pins, one pin to trigger the pulse called TRIG, one pin to read the echo called ECHO, one pin to supply Vdd and one ground pin. The
supply voltage is five volts, which is the same voltage level as the logic of the sensor. The HC-SR04 is a transceiver, it can both receive and send pulses. The module also has a built in controller and the distance is interpreted as a pulse. The length of the pulse determines the distance [102].

3.1.9.2 Reflective sensor, OPB715Z OPTO SWITCH

OPB715Z OPTO SWITCH, is used to determine the speed of the car. The sensor measures IR light reflections. It constantly emits IR light and checks for an incoming reflection from the light. As the sensor receives the light it sends out a high pulse that can be captured using a controller. The sensor can be run at five volts and will then generate a five volt output [103].

3.1.9.3 Motion sensor, MPU-9250

The MPU-9250 is a nine-axis motion sensor. The sensor has a three-axis gyroscope, a three-axis accelerometer and a three-axis magnometer. The sensor is interfaced using I²C protocol and data can be fetched as digital data. The gyroscope detects differences in angles; the accelerometer measures the relative position in XYZ to earth’s gravity and the magnometer can act as a compass [104].

3.2 Operating System

This section will describe the Operating System run on the ECUs. There are multiple OS available for cars. Some of the most popular ones have already been discussed in Chapter 2.6. This platform uses non of these, but a distribution of
3.2. Operating System

Linux with a real time kernel configured. The reasoning why this OS was the choice will be further discussed in Chapter 4.

3.2.1 Linux 4.4-ti-rt

The operating system that is run on the BeagleBone Black is Linux. The kernel is a 4.4 Linux kernel with the PREEMPT_RT patch applied. More specifically it is the 4.4.19-ti-rt-r41. The kernel is a kernel modified by Texas Instrument, the manufacturer of the SoC of the BeagleBone Black. The reason why Linux was chosen as the Operating System is discussed in Chapter 4.2. It is important to know what networking scheduling policy the Kernel is using. The 4.4.19-ti-rt-r41 kernel uses pfifo_fast scheduling policy, as can be seen in Figure 3.10. pfifo_fast is the default network queueing policy. The policy provides three different FIFO queues to separate the traffic. Packets with highest priority are always placed in FIFO 0 and is always served first. FIFO 1 is always served before FIFO 2 [105].

![pfifo_fast queuing discipline](image)

Figure 3.10: pfifo_fast queuing policy, taken from [14].

3.2.2 Linux 4.4-rt

The operating system run on the Raspberry Pi 3 Model B is a linux based operating system with the PREEMPT_RT patch applied. The kernel is, similar to the one
3.3. Software tools

This section will describe the software tools used throughout the project. There are multiple tools used, each tool will be described and a description of why it is needed will be provided.

3.3.1 Wireshark

Wireshark is a network protocol analyzer. Wireshark lets the user both capture packages live and create recordings. It is possible to create filters to only inspect certain packets or protocols [106].

The tool is used to inspect and debug the traffic sent over the internal network of the system. It is possible to inspect packets at different abstraction level, which makes it possible to see the VLAN tags and VLAN priority fields. It is also used to inspect the traffic being sent over the WiFi from the RPI ECU.

3.3.2 Android studio

Android studio is the most widely used tool to develop Android applications. It is a GUI based tool, with a powerful debugger. It is also possible to run an Android emulator though Android studio, emulating a wide selection of different target devices.

3.3.3 config-pin

The config-pin utility helps to configure the output pins by using the devicetree overlays and capemanager. It is possible through a simple command, with the pin ID as the parameter, to configure a pin for a specific task. Listing 5.2 shows an example use of the config-pin utility.

<table>
<thead>
<tr>
<th>Listing 3.1: Config-pin example</th>
</tr>
</thead>
<tbody>
<tr>
<td>config-pin -a P8.45 pruout</td>
</tr>
<tr>
<td>config-pin -q P8.45</td>
</tr>
</tbody>
</table>
The example above sets pin 45 on header P8 to be an output connected to the PRU subsystem and pin 46 on header P8 to be an input to the PRU.

### 3.3.4 Cyclictest

Cyclictest is a benchmarking software used to test real-time performance of an operating system. Essentially it measures the latency of a response to a stimulus. The system can be tested running the operating under different loads [107].

### 3.3.5 TI PRU C compiler

To program the PRU the TI PRU C compiler is used. It is necessary to use a specific compiler for the PRU that is targeting the specific architecture of the PRU. There are other compilers available, but this is the one supported by Texas Instruments, the manufacturer of the BeagleBone Black SoC. The compiler is also included in CCS, a development environment from Texas Instruments, but it is possible to run it outside the environment [108].

At the early stages of the PRU development there was no C compiler, instead assembly was the only way to program the PRU. At present time, there is C library code to access most of the registers and functionality exposed to the PRU, however the documentation is lacking in detail. It is necessary to look through the library code in order to fully understand how to use all available C functions.

### 3.3.6 Device Tree Compiler

When working with the BeagleBone black device trees and device tree overlays are needed to configure and set up the hardware and peripherals. The device tree compiler takes device tree source file and compiles it in to device tree binaries. What device trees are and what they are used for was previously discussed in Chapter 3.1.3.
Chapter 4

Design

This chapter will describe the design of the platform. The design is a major part of the results of this thesis and one of the core goals. The chapter starts by discussing the hardware chosen for the project in terms of why the specific hardware was chosen. The next part discusses the operating systems used. The last part of this chapter provides an overview of the system architecture.

4.1 Hardware

There are a lot of hardware components described in the previous chapter. All of the hardware has been chosen for specific reasons. This section will discuss the design of the platform and the reasoning to why the specific design was chosen. More specifically this section will describe the hardware related parts of the design.

The first hardware to discuss is the ECUs that are part of the platform. The next step to discuss is the choice of network switches. Furthermore the choice of sensors are discussed, followed by a discussion regarding the choice of model car, actuators and electronics. The last piece of hardware to be discussed is the android device.

4.1.1 ECUs

The platform is equipped with two different types of ECUs, the BeagleBone Black and the Raspberry Pi 3 Model B. The BeagleBone Black was chosen due to its ability to handle real-time tasks with the help of the on-chip PRUs. Without having these real-time units the use of sensors that require exact timing on the output pins would become a difficult task. The PRUs also enable for further implementation of specific protocols by programming the PRUs to act as controllers. Additionally
4.1. HARDWARE

the BeagleBone Black has the capability of running as an AVB device. This is, however, not possible yet, as it would require additional work before it can be available. This will be discussed further in Chapter 6.2. Another reason why to use the BeagleBone is that the hardware is Open Source.

Initially the plan was to only use BeagleBone Blacks as ECUs but there was a need for a wireless interface. This could be done by adding a USB wireless network interface to one of the BeagleBone Blacks, but it would cost almost as much as adding another ECU that already has WiFi capabilities. Additionally, the BeagleBone Blacks are not very good at handling heavy parallel tasks such as image processing. This would limit the ability of the platform because one of the important sensors, being part of ADAS, are cameras. The RPI 3 Model B has both WiFi and, due to its relatively sophisticated GPU, the ability to do image processing. Also, there is a lot of information related to image processing, such as object detection and image recognition, available by the Raspberry Pi community.

Both of the two ECUs are running on an ARM main processing unit, which is a common choice for automotive ECUs, as was discussed in Chapter 2.1. The BeagleBone also has access to an on-chip microcontroller (the PRU), running a RISC core, which is also a popular choice in the automotive industry.

4.1.2 Network switches

The network switches used are the TP-link TL-SG105E Network Switches, described in detail in Chapter 3.1.6. They are cheap and they support the 802.1Q VLAN standard. It is important for the switches to support real-time Ethernet features. VLANs can also be used for automotive purposes to limit broadcast traffic in automotive Ethernet networks, see Chapter 2.4.4.4. There are switches that have full AVB/TSN support and can provide much better performance in terms of real-time. The downside to these switches are that they are very expensive and complicated to use. Many of these switches that are available on the market only support proprietary software to implement the AVB standard, and it is not possible to integrate them in a project like this. Another significant reason why not to use such switches are that in order to follow the AVB/TSN standard there is a need for specific Ethernet drivers, hardware support in the end nodes and a software stack. As of today none of this is available for the BeagleBone Black nor the Raspberry Pi 3 Model B.
4.1.3 Sensors

The sensors used in the project include an ultrasonic sensor that can measure the distance to objects in front of it, a reflective sensor, a nine-axis IMU and a camera. The ultrasonic sensor is used in a lot of projects available on the internet and there is a lot of documentation to it. The sensor is also well suited as it can be used to demonstrate some simple scenarios, as parking assistance, automated breaking. As mentioned in Chapter 2.2 the ultrasonic sensor is commonly used in ADAS, and has been for quite a while.

The reflective sensor is used to detect the wheel speed. It is fitted on the wheel axis of the car to point to the inside of the wheel. Inside the wheel there is reflective tape to generate pulses. By measuring these pulses it is possible to get the speed of the wheel and the car. This is a similar technique to what the MOPED project used to acquire the speed of the vehicle.

The nine-axis IMU is used in cars to measure dynamically changing movements of cars. The camera is frequently used in ADAS and is a cheap option to implement complex functions of ADAS. In addition the RPI Camera module is a well documented camera and the Raspberry Pi community provides a lot of support for various implementations using the camera.

4.1.4 Model car, actuators and electronics

A lot of the parts related to the model car was chosen because they were previously used in the MOPED project, described in Chapter 2.7. The author of this thesis has previously worked with the MOPED platform and has insights to what was working for the MOPED platform. Some parts of the car were not used in the MOPED project, because these parts were replaced by newer versions.

The chassis of the car is not the same as the one suggested by MOPED, but a similar one that comes in pieces. The reason why this chassis was chosen instead of what was used for MOPED is that it is cheaper and it provides more modularity. The motor is the same one as the MOPED project used, it is a high quality motor and it makes it possible to run the car at low speeds. It is important to be able to run the car at low speeds when testing different scenarios. A lot of other brushless motors are better suited to run at higher speeds.

To steer the car a steering servo is needed. The one MOPED used was included in the chassis, but the chassis used in this thesis came without one. The servo chosen is a high torque servo, which is necessary to be able to turn the wheels. This was one of the cheaper high torque servos that was in the correct size to fit the chassis.

The ESC is the same one as the one in the MOPED project, and it can be tuned to run the motor at lower speeds using a programming card. The ESC is rated at 45
amperes and can supply enough power to the motor. The battery powering the car is also the same one as the one used in MOPED. It is important to have a high discharge rate battery to be able to handle the large amount of current drawn by the motor. LiPo batteries have high discharge rates and are well suited to power Radio Controlled cars and projects like this.

Another important part is the voltage regulator, described in Chapter 3.1.7. It is compatible with the battery as it can take the same voltage. The regulator can output enough current to run all the electronics running at 5 volts, including all ECUs, the servo and the sensors.

4.1.5 Android device

The choice of android device is not important as almost any android device can be used for the purpose of controlling the car. What is more important is the reasoning behind the choice of an Android platform rather than what specific Android device to use. The choice to go with an android device is because it is easy to use and it can be hand held. The development environment is easy to use and the author has experience in Java programming and Android development.

4.2 Choice of OS

The operating system chosen is presented in Chapter 3.2. It is a Linux operating system with a real-time patch applied. Linux is, as already stated not initially intended to be an RTOS, but a GPOS. However with the real-time patch applied it provides better performance in terms of real-time. There are other alternatives of Operating Systems that will perform better and are more suited for real-time and reliability tasks, such as OS implementing the AUTOSAR standard. However, this platform is not intended to be a product that lives will depend on. The point of the platform is to create a prototype as a proof-of-concept to test different types and concepts of ADAS. As stated in Chapter 2.6.2 Linux is used for prototyping systems when it comes to ADAS.

The advantage of Linux is that it is widely used, open source and it has a huge community providing support. Linux is also the OS supported by BeagleBone and RPI. Furthermore, a big advantage for Linux in this specific product is that a lot of students is familiar with Linux. Below is a list compiled with the benefits and drawbacks of using Linux for this platform.

- + Easy to use

- + Large community, support by the BeagleBone and RPI community
• + Provides real-time features
• + OS known to students
• + Good for prototyping
• - Not hard real-time

Moreover, the PRUs on the BeagleBone board give the user possibilities to do hard real-time tasks.

The other alternative, to use an AUTOSAR derivative OS would require a lot more work, in terms of developing new software, learning the OS and compiling it for the BeagleBone Black board. It would also increase the amount of time spent for anyone trying to use the platform. In addition, implementations following the AUTOSAR standard are not open source. Using an AUTOSAR implementation would limit the availability and the extendlility of the platform.

4.3 Overview of the system architecture

This section will describe how the hardware is assembled to create the architecture. The platform uses three TP-link TL-SG105E network switches to create the communication network of the platform. Each network switch is connected to the other with a Cat 6 network cable. Each node in the network is connected through a VLAN with a 801.2Q tag, defining the priority of each data stream. The network topology can be seen in Figure 4.1
4.3. Overview of the System Architecture

The topology used on the platform is three switches all connected to each other and the ECUs. The reason why this topology was chosen is because this topology emulates a real topology that could be used in a real system. It provides redundancy without requiring a large amount of switches. The topology allows for link faults, while still providing a link between the ECUs. An example is shown in Figure 4.2.

Figure 4.2: Example of broken link

If Link2 is broken and SW1 wants to send a packet to SW3, the packet can
travel SW1 - SW2 - SW3 instead.
The system consists of four ECUs, three BeagleBone Blacks and one Raspberry Pi. Two of the BeagleBone Blacks read sensor data and propagate it to the third BeagleBone Black that controls the actuators. The forth ECU, the Raspberry Pi 3 Model B, reads the camera data and handles the external WiFi communication and monitors the network status. There is also a fifth node connected to the network: an android device. Exactly how the communication works, how the actuators are controlled, how the sensor data is read, how the android Application and communication works is described in detail in Chapter 5, where it is described how the implementations work. Figure 4.3 gives an overview of the system architecture.

![Figure 4.3: Overview of the system architecture](image)

The nodes BBB1, BBB2, BBB3 are the BeagleBone Black nodes. The BBB1 and BBB2 ECUs are connected to the sensors and collect the sensor data. The BBB3 is connected to the actuators of the platform. The RPI with WiFi
block is the Raspberry Pi 3 Model B node. The RPI ECU handles the external communication and the camera sensor. All ECUs are connected together over the internal network, where SW1, SW2 and SW3 are the Ethernet network switches.

The architecture tries to mimic a real system inside a car, containing multiple ECUs, an in car network with multiple network switches, sensors and actuators. The car also provides a wireless connection to the outside world, with the help of the WiFi interface on the RPI 3 ECU. Due to the multiple ECUs built on top of the network architecture advanced features of ADAS can be tested, even features requiring sensor fusion. The flexible underlying architecture also allows for extensibility of the platform, where additional ECUs can be added on top of the underlying communication architecture. In addition additional sensors can be added either to the already existing ECUs, or by adding additional ECUs. The wireless interface of the architecture allows the platform to communicate with external interfaces. This can be used for an external control interface to control the platform. It can also be used to monitor the car from any external interface that has an Ethernet interface. In addition a wireless interface allows the platform to perform highly advanced functions of ADAS, such as V2I and V2V communication. These features will be further discussed in the next two chapters.
Chapter 5

Implementations

This chapter will discuss the implementations done for the thesis. Each implementation will be presented individually. The chapter will start by discussing the implementation that handles the communication between the ECUs. The next thing that will be presented is the kernel implementations on the ECUs. Furthermore the implementation that reads sensor data will be presented, followed by a description of the implementation to control the actuators. The next thing that is presented is an implementation to monitor the car network. Moreover the implementation to control and monitor the car is presented. A way of logging errors on the platform is briefly discussed.

5.1 Car network and ECU to ECU communication

An important task of this thesis is to create the underlying communication between the ECUs. This section will describe how the network is built and how the ECUs communicate.

The communication network follows the 802.11Q Ethernet protocol. The protocol is set up using VLANs, described in Chapter 2.4.4.4. The VLAN makes it possible to set priorities to traffic flows. The priority can be statically configured from the VLAN script or set for specific data flows. The priority can be set from C-code using the setsockopt()-function. The function sets the priority field in the VLAN tag for a specific socket. The priority is read in the network switches and is assigned to the internal priority queues in the switch, as described in Chapter 2.4.4.4. Additionally, changing the priority of the socket will also change the priority of the network scheduler in Linux. Increasing the priority will send packets with higher priority according to the policy described in Chapter 3.2.1.
The VLAN is automatically configured when the ECU boots. This is done using a shell script. The shell script used to set up the VLAN on one of the BeagleBone Blacks is shown in Listing 5.1.

Listing 5.1: Vlan setup script

```
#!/bin/sh
# located in /etc/init.d/vlan_setup.sh
# setup vlan
sudo vconfig add eth0 34
sudo ifconfig eth0.34 192.168.1.2 netmask 255.255.255.0
sudo vconfig set_egress_map eth0.34 0 0
sudo vconfig set_egress_map eth0.34 1 1
sudo vconfig set_egress_map eth0.34 2 2
sudo vconfig set_egress_map eth0.34 3 3
sudo vconfig set_egress_map eth0.34 4 4
sudo vconfig set_egress_map eth0.34 5 5
sudo vconfig set_egress_map eth0.34 6 6
sudo vconfig set_egress_map eth0.34 7 7
exit 0
```

The script configures the VLAN to be using the eth0 Ethernet interface - the Ethernet interface used on the BeagleBone Black. Next the script tells what network the VLAN is to be set up on, in this case 192.168.1.2. Lastly the egress map has to be set up. The egress map tells how the mappings of the priorities of the outgoing packets should be arranged. In this case the priorities are mapped to the priority they are set from the context. The map can be used to map preset priorities from other software to either a lower or a higher priority.

Another important thing to take into account is that, due to the design of the network topology, switching loops can occur. This phenomenon is described in Chapter 2.4.4.2. To avoid the problem the switches are configured to avoid loops. This setting is reached from the configuration GUI of the network switch.

Besides the underlying communication layer described above the ECUs communicate using the transport layer protocol UDP. Each ECU in the network has a statically configured IP-address and each of the ECUs know the address of all the other ECUs in the network. Packets are sent using the static IP-address and by using UDP sockets. All socket software implemented is written in C.
5.2 Implementing the OS

This section will describe the process to implement the operating systems running on the ECUs. Firstly the section will describe the process of deploying the OS running on the BeagleBone ECU, followed by a description of how to implement the OS on the Raspberry Pi ECU.

5.2.1 BeagleBone ECU

A big time consumer in this thesis, was to figure out how to compile a stable kernel with the PREEMPT RT patch applied and at the same time enable the PRU environment. This proved to be quite difficult as there is no complete guide on how to do this. This section will describe the procedure on how to do this, for anyone trying to reproduce the OS used in this thesis.

The first step to do is to compile a kernel with the PREEMPT RT patch applied. This can be done by following a guide supplied by EEWIKI [109]. The guide gives complete instructions on how to compile a kernel and how to deploy different version of Linux file systems on top of the kernel. The process includes instruction on how to cross-compile the kernel targeting the ARM architecture.

An SD card is needed to complete this guide, and only one card is needed, as the OS can be moved to the eMMC of the BeagleBone Black. The kernel chosen is the ti-rt-4.4 kernel. This kernel has the real-time patch already applied to it. The root file system used is a light version of Debian, namely Debian 8 (small flash). It is important to install the bb-overlays as stated in the guide. There is no need to enable the HDMI, as it will only limit the amount of GPIO pins and there is no need for a graphical interface. The guide also gives instructions on how to flash the SD-card to the eMMC of the BeagleBone Black. There is no need to do this yet as it would only complicate the process. The BeagleBone Black should now be able to boot with the SD-card inserted by holding down the USER/BOOT button. To verify that it is booting from the SD-card, type uname -a in the terminal. The output should list the kernel running.

The next step is to download the dtb-rebuilder to be able to compile the device tree files. The dtb-rebuilder can be found in this git repository [110]. The dtb-rebuilder is a device tree compiler to compile device tree source files into binaries.

Edit the am335x-boneblack-overlay.dts, the device tree overlay to be used, which can be found in the git repository [110]. Uncomment the line saying #include “am33xx-pruss-rproc.dtsi” in the .dts file. This enables the pruss-rproc driver, described in Chapter 3.1.1.1.2. Type make; sudomakeinstall to install the
5.2. Implementing the OS

device tree. The next step is to tell the operating system to use the device tree file, this can be done by adding \texttt{fdtfile = am335x - boneblack - overlay.dtb} to \texttt{/boot/uEnv.txt}. To make sure the pruss-rproc driver is enabled, reboot the BeagleBone board and type \texttt{lsmod}. The driver should be be visible in the list of drivers currently active.

To make it easier to change the hardware pin mapping the config-pin utility, described in Chapter 3.3.3, has to be installed. This is be done by cloning the repository\cite{111} and installing the universal cape overlay. The universal cape overlay allows the user to use most I/O without editing .dts files and recompiling them. However, there are several errors in the namings of the overlay. This can be fixed by changing the names to the correct ones. A script to automate this process was created and is shown in Listing 5.2.

Listing 5.2: Config-pin example

```bash
#! /bin/sh

# change names of gpio
sed −i 's/gpio1/gpio0/g' cape-universala-00A0.dts
sed −i 's/gpio2/gpio1/g' cape-universala-00A0.dts
sed −i 's/gpio3/gpio2/g' cape-universala-00A0.dts
sed −i 's/gpio4/gpio3/g' cape-universala-00A0.dts

# change name of uarts
sed −i 's/uart1/uart0/g' cape-universala-00A0.dts
sed −i 's/uart2/uart1/g' cape-universala-00A0.dts
sed −i 's/uart3/uart2/g' cape-universala-00A0.dts
sed −i 's/uart4/uart3/g' cape-universala-00A0.dts
sed −i 's/uart5/uart4/g' cape-universala-00A0.dts
sed −i 's/uart6/uart5/g' cape-universala-00A0.dts
```

The script changes the name of the GPIO pins and the UART pins, using the \texttt{sed} command. The \texttt{sed} command is used to edit text files.

To install the overlay run \texttt{make;makeinstall}.

The operating system should now work and support PRU development. To test that it is working the following example and guide can be used \cite{112}. 
5.2.2 Raspberry Pi ECU

The kernel used for the Raspberry Pi ECU is a pre-compiled kernel from [113]. It is possible to compile the kernel from scratch and apply the preemptive patch manually, but that would require unnecessary extra work. The simplest way to install the pre-compiled kernel is to first download and install an image from the raspberry pi web page [114]. The image used is the Raspbian Jessie Lite. The web page provides instructions on how to deploy such an image on to an SD-card. After the image is downloaded the kernel can be replaced with the pre-compiled preemptive kernel. This can be done by following the guide on [115]. After the guide is completed the Raspbian Jessie Lite image should be up and running with the PREEMPT_RT patch applied.

5.3 Sensors

The platform is currently only implementing one sensor, an ultrasonic sensor. The wheelspeed sensor, the IMU and the camera is as of current not implemented. This section will describe how the sensor data from the ultrasonic sensor is acquired and interpreted.

5.3.1 Ultra Sonic sensor

This section will describe the process to acquire the data from the sensor. To get sensor data it requires software to run both on the ARM core and on one of the PRUs. Figure 5.1 explains the algorithm to retrieve data from the sensor on the PRU, in a graphical way.

In order to get the data from the HC-SR04 ultrasonic sensor a 10 microsecond pulse is generated. The sensor outputs a pulse with the length equal to the time it takes for the sound to travel to the object and back. The PRU subsystem handles the generation of the pulse as well as interpreting the incoming response from the sensor. To generate the pulse the PRU sets the output pin to zero, waits for 2 microseconds, sets the pin to high, waits for 10 microseconds and then sets it back to zero again. This generates a pulse with the width of 10 microseconds.

The next step is for the PRU to wait for the incoming pulse. To do this the PRU waits for the input pin (the pin connected to the ECHO pin of the sensor) to become high. When the pin becomes high the timer is started and its counter is set to zero. Further the PRU waits until the pin goes back to zero again. When the pin is set to zero the timer is stopped and its value is acquired. The value collected
(the number of cycles that has passed) is then converted in to microseconds by using the following formula:

\[ \text{microseconds} = \text{cycles} \times \frac{10^6}{\text{clock frequency}} \]  \hspace{1cm} (5.1)

Where microseconds is the width of the pulse received, cycles the number of cycles in the timer register, \(10^6\) is to scale it to microseconds and the clock frequency is the clock frequency of the PRU (200 MHz). Next, the microseconds is converted to a distance in millimeters, by using the following formula:
Where distance is the distance to the object in millimeters; microseconds is the width of the pulse; 1000 is to scale it to millimeters, and 5882 is the time it takes in microseconds for a round-trip of one millimeter (the pulse measures the round-trip time). The last step for the PRU is to send the value back to the ARM, by passing the value to the shared memory.

There are some risks related with waiting for the sensor data, as there could be deadlocks if the sensor never returns a pulse, or if the sensor never ends an ongoing pulse. If a deadlock occurred the PRU would hang and be unreachable. This is avoided by starting the timer when the sensor waits for an incoming pulse and if the pulse doesn’t come the timer will set a flag to detect a timeout. The same goes if an ongoing pulse is never ended, the timer starts at the beginning of the pulse and if it is too long it generates a timeout. If the PRU detects this timeout it will return a negative value to the ARM instead of the actual distance. The ARM will interpret the negative number and take measures to handle the error.

The ARM asks the PRU every time it needs a value from the sensor, but it is important that it does not ask for sensor data in a higher rate than 40Hz (the maximum rate to get "safe" data). To fetch the data from the PRU there is a program running that reads and write to a character device file, as described in Chapter 3.1.1.1.3. The character device file is read and written to from a C-program running on the ARM.

The sensors operating voltage is 5 volts but the BeagleBone Black’s logic level is 3.3 volts. To make it compatible a logic level shifter can be used. However the sensor works even when supplied with a 3.3 voltage signal, so it is possible to create a circuit to only limit the volts output by the sensor to the BeagleBone Black. This circuit can be realized using only two resistors, creating a voltage dividing circuit. According to Equation 5.3 and the schematic seen in Figure 5.2

\[
V_{out} = V_{in} \times \frac{r_1}{(r_1 + r_2)}
\]

There are multiple solutions to the equation. The values used are a resistor 2k ohm for \(r_1\) and 1k ohm for \(r_2\).
5.4 Controlling actuators

This section will describe how the actuators are controlled. The platform has two different actuators: one steering servo and one brushless motor that is controlled with the ESC.

5.4.1 Steering servo

As previously stated servo motors are controlled by using PWM signals. To generate PWM signals the PRU is used. The steps to control the servo includes software implemented both in the ARM core and the PRU environment. This section will start by introducing the program running on the PRU.

The servo pulse can be generated in an easy way by setting a pin to high, sleep for some cycles and then set it to zero and repeat. However, this is a naive approach, as it consumes one PRU completely. A more sophisticated approach is to use one of the timers exposed to the PRU and capture the interrupt generated by the timers.

Chapter 3.1.1.1 describes the PRU and the presence of two different timers: the Industrial Ethernet Timer (IEP) and the eCAP timer. The timer this implementation uses is the IEP timer. The timer uses compare registers to handle timer events. A compare register is loaded with a value and, if the comparator is enabled, an interrupt is generated when the value of the timer is equal or above the value of the comparator. Interrupts are generated in a special interrupt register, that can be accessed from the PRUs.
The PRU program to control the steering servo is described in Figure 5.3.

![Servo control algorithm](image)

**Figure 5.3: Servo control algorithm**

Firstly the register that contains the interrupt vector is checked, to identify if any interrupt has occurred. The next step is to identify what caused the interrupt. If the interrupt was generated from the timer, it has to be clarified which of the comparators fired the interrupt. If it was cmp0 (comparator 0) the PWM pin is set to one and the timer register is cleared. The CMP0 is used to notify that the duty cycle is over and the next cycle starts. If it was not cmp0 that fired the interrupt cmp1 (comparator 1) is analyzed. In the case it was cmp1 that fired the interrupt the PWM pin is set to 0, this means that the pulse is over.

If it was neither of the comparators that fired the interrupt it is checked if it was the ARM that was interrupting the PRU. If the ARM interrupted the PRU the messages are check and the new servo pulse value is checked. After the servo value is parsed the cmp1 has to be set, this means that the pulse width of the
5.5. Network monitoring

The platform has, as discussed in previous sections and chapters, multiple ECUs connected over Ethernet. Ethernet has built in error correction, but errors can still occur in ways such as buffer overflows, resulting in dropped packages or increased delay. This section will describe how the platform handles the monitoring of the network.

The monitoring of the network is not only good for monitoring, but it can also be used to adapt the data being sent from other systems running on the platform by prioritizing certain data flows. There are already built in tools in Linux that can check latency and check package drops. The Linux Ping tool can get the latency to nodes on a network. There are four ECUs present in the network. All ECUs have a static IP address from the start. The network monitoring program implemented on the platform is running on the running RPI 3 ECU and can be seen in Figure 5.4.

The program is checking the connection to all other ECUs by running the Linux Ping tool from a C program context. The program starts a separate thread to keep track of the connection to each ECU. The Ping tool output is captured
5.6 Control and monitoring interface

Since the car is a model car and is in size 1:10 there is a need for another control interface. This control interface has to be an external interface to be used remotely, and preferably wireless. Additionally a monitor interface is integrated in the implementation. This section will describe the control and monitoring interface implemented for this platform, both on the server side and on the client side, in form of an Android Application. In addition to this the section will also describe

Figure 5.4: Network monitoring program

in C and parsed to acquire the numerical value of the Ping packet. It gathers information from each one of the ECUs pings and stores it in a data structure. Another separate thread is running to store the data from each ECU in a shared data structure. In this shared structure the average and the total number of dropped packages is stored.

Each thread will send a ping packet five times a second, i.e. once every 200 milliseconds. If the Ping packet does not return within the 200 milliseconds, the packet is considered lost and generates a timeout. Every time a the program catches a timeout a counter is increased and stored in the data structure mentioned above.
the client runs on each of the BeagleBone Boards that sends the status information to the RPI 3.

5.6.1 Android App

The android application implemented is made mainly to provide an interface to control the platform. Additionally, the application provides some possibilities of monitoring the platform. When the app starts, it starts sending packets to the server to get the monitoring data. Packets are sent once every second to get the data. Currently, the data sent is the average ping in the network, the total amount of dropped packages, and the data captured by the ultrasonic sensor, but this can be extended to get even more data. The data is received on a UDP server running in the Android application, and the packets sent are also UDP packets. The interface to control the car is also implemented using a UDP data stream. The user can control the car using sliders, one slider to control the speed and one slider to control the angle of the wheels. The layout of the application can be seen in Figure 5.5. This is an initial layout and there is major improvements to be done on both the monitoring interface as well as the control interface. The improvements will be discussed in Chapter 6.2.8.

![Android application layout](image-url)
The car has to be connected to WiFi to establish the connection. The android app can either be connected to the same local network using local IP to the car or from an outside connection. If the Android device is not on the same local network ports have to be opened in the router in order for the application to get access to the ports of the car.

### 5.6.2 UDP server

The server receiving data is running on the RPI 3, which is a UDP socket server attached to a specific port. The reason why the communication between is implemented using UDP instead of more sophisticated protocols such as TCP is because there is no need for any re-transmission as all the data is real-time. All old data should be discarded and not acted upon. In the case the old data might steer based on old data, and the vehicle would behave faulty. The server is run in its own thread receiving any packet coming in to the port. The packets received are propagated to the BeagleBone Black in charge of controlling the actuators. There is no point in sending old data to the actuator control. In order to know that we do not act on old data, the appended packet number has to be inspected. The structure of the packet can be seen in Figure 5.6.

![UDP data diagram](image)

**Figure 5.6: Package structure of packet sent from Android Application**

If the number received from the sender is lower than the previous packet the new packet can be discarded and considered outdated. The identifier field in the packet tells whether to control the brushless motor or the servo motor, and the value is the value to write to the actuator. Before sending the packet the priority is set to the highest, i.e priority 7. Figure 5.7 describes the program flow of the program.

To send the data about the health and status of the car another separate thread is created. The thread has a UDP socket server receiving packets on another port. This port receives data about the status of the car from the other ECUs. The program takes all the information collected from the other ECUs and stores it in
a data structure. The information sent to the server is currently not decided, but it can take any information relevant to the platform, such as speed, sensor status etc. The basic structure of the UDP data containing the status can be seen in Figure 5.8.

It is important, yet again, to discard the old data and not store it in the data structure. This is done, by appending an incremental counter in the data as described earlier, this field is, again, called Packet number. Since not all packets might contain the same amount of data it is important to set the size of the data, to make it easier to parse the packet. Each "status" appended to the packet by first adding an identifier of the status followed by the value.

The network health about the car, described in Chapter 5.5, stores the data in a data structure on the RPI 3. The data has to be fetched and the appended to the data sent to the Android Application. There is however some problems related to this, since there are multiple threads accessing the same data, both being able to read and write at the same time. This
5.6. CONTROL AND MONITORING INTERFACE

Figure 5.8: Package structure of the data sent containing status

rises the issue of concurrency. The data structure has to be accessed exclusively when either read or written to. The problem at hand can be solved by using a Mutex. A mutex is a form of lock that a thread asks to either acquire or release. The acquire call is blocking, and if one thread currently has the lock the other thread has to wait. This is demonstrated in Figure 5.9.

For example: If thread 1 gets access to the lock it starts to write values to
the data structure and thread 2 wants to read, thread 2 has to wait until thread 1 releases the mutex.

When the the data is acquired safely from the data structure holding the network monitoring data, it is appended to the status message. The data is only sent as the Android app needs it. The requests from the App is received in a separate thread on another port. When the server receives a request it takes the data acquired and sends it to the Android App. However, also here there can be concurrency issues and another mutex is needed to make sure the correct data is being sent.

The client to control the actuators and to read the sensor data is described previously. The client that sends the status data on the BeagleBone Black ECUs sends the sensor data at a constant rate to the server. Currently it is only the ultrasonic sensor that sends status data.

There are a lot of details about the control and monitoring interface described in this section. Figure 5.10 provides a summary of what has been described in this section.

Figure 5.10: Summary of the Control and Monitoring interface

5.7 Error detection

Error detection is an important task of the platform is to identify if an error has occurred. Currently the way the system handles and detects error is by writing to a log file. Each software component implemented logs the error if one is detected. For example: If the software running on the ARM can not communicate with the
software running on the PRU the ARM software will log the error in the log file. Another example of an error that could occur is if one of the sockets fails. This way of handling errors could be improved and extended. The improvements to this will be discussed in Chapter 6.2.4.
Chapter 6

Conclusions

This chapter explains the conclusions obtained throughout the thesis. The chapter will give a summary of the conclusions, followed by a discussion about future work.

6.1 Summary

A lot of time was spent doing the initial research of this thesis, to acquire enough background knowledge on the subject in order to come up with a good design. As the design was finished the components in the design had to be ordered. This ordering proved to be a much more difficult task than predicted. A lot of time was spent on trying to get the correct forms to order the parts and to get them approved by KTH. In addition to this a lot of parts were delayed. As a result of this the development of the implementations were delayed. In retrospect, the ordering could have been done in parallel with the theoretical study.

A big time-consumer during the thesis was to understand how the PRU works. This included both understanding how the drivers work and how the actual microcontroller works. The process to set up the PRU environment and setting up the tool chain to program it took a lot of time and effort. Time could have been saved on this by using an external microcontroller, like an Arduino, and communicate to it using a serial protocol. However, using the PRU this can be done on a single chip saving both money and space on the platform. The thesis provides a guide on how to set up a kernel with the PRU enabled. It also describes how to program and how to communicate with the PRU, making it easier for the next user of the platform, or anyone trying to use the PRU.
6.2. Future work

6.1.1 Goals

This section will discuss the goals of the thesis and evaluate the results of the thesis with respect to the goals. The two main goals of the thesis was to design and implement an automotive experimental platform.

There were some constraints on the design: The platform should consist of multiple ECUs; the platform should fulfill the needs for testing some ADAS and the platform should provide some sensors to test some functionalities of ADAS. The design met all of these goals as it has multiple ECUs. It provided a platform in shape of a model car to be able to test ADAS. It also had multiple sensors that are commonly used for ADAS: an ultrasonic sensor, an IMU, and a camera. The communication technology is using Ethernet which is the latest trend in the automotive industry.

Regarding the implementation goal of the thesis - the initial idea was to provide some implementations to provide basic functionality to set up a simple test scenario. However, this was not possible due to the delayed arrival of the components and the limited time span. The delayed components resulted in a less time to implement the necessary functionality to set up a test scenario. Most of the functionality exists to perform the test scenarios, but connecting all specific implementations and connecting it to the physical model car is yet to be done.

6.2 Future work

Due to the size of the task to both design and implement the platform and the limited amount time at hand, parts were left undone and other parts remain to be improved. This section will discuss the future work of the platform; what can be done to improve it and how the continued work should be conducted.

6.2.1 Adding additional sensors

As mentioned earlier there is one sensor implemented on the platform, an ultrasonic sensor. There are more sensors present on the platform but they are not yet implemented. It is the IMU, the wheelspeed sensor and the camera.

Implementing the camera on the platform enables a lot of more possibilities of experimenting with ADAS, as can be seen in Chapter 2.2. Adding an additional camera, making it two in total, could also allow for stereo vision, enabling depth image analysis. To implement the wheelspeed sensor requires less work than
implementing the IMU and it provides features to read the speed of the car. Implementing support for the nine-axis IMU that is currently in the design of the platform could further enable support for more advanced types of ADAS.

Other sensors could also be added to increase the coverage of ADAS that can be tested. As of present the LiDaR sensors are quite costly, but they might become more available in the near future. Adding a LiDaR sensor to the system would enable it to experiment with some of the most advanced and high end ADAS.

### 6.2.2 Sensor fusion

A modern approach to handling sensor data is to fuse the data of different, or similar, sensors together. This can be done by fusing for example the ultrasonic sensor with the camera feed, to create a parking assistance system. However, if this data is to be fused together using data captured from different ECUs, there has to be some kind synchronization method of time between the ECUs (As AVB suggests).

### 6.2.3 Implement full AVB/TSN support

Currently AVB is not implemented on the platform. The protocol used is 802.11Q which only partly implements some of the functionality of AVB/TSN. However, an idea presented at Google Summer of Code presents that the BeagleBone Black has the necessary silicon support to implement AVB support. To enable the device for the use of AVB the Ethernet drivers have to be rewritten. Also in order to implement AVB a software stack is needed. There is an open-source AVB stack called open-AVB, but in order to make the stack work on the BeagleBone hardware it has to be ported to the specific platform [116].

### 6.2.4 Error detection

An important feature is a way of detecting errors on the platform. Currently the only way the system handles errors is by logging it to a file. This could be extended by implementing a way to broadcast errors from each ECU to notify the system that something is erroneous. Each ECU would have a server listening to the port that the errors are broadcast on. By knowing an error has occurred the system can adapt and enter a kind of ”error mode” to operate safely. This has to be done on different levels, errors could be anything from a sensor failure to some higher level software generating an error.

The network monitoring service also has to notify the ECUs if the network is
6.2. Future work

congested - if it is the ECUs they have to adapt and send less information. This has to be done by selecting the important information and the not so important information, and limit the less important traffic. This could be extended even further by detecting the road condition environment and the data needed for the specific environmental situation. How this could be implemented and applied on the platform is beyond the scope of this thesis.

6.2.5 Benchmarks

The platform requires a lot more testing and benchmarking. The kernels can be tested using cyclic tests, described in Chapter 3.3.4 measures the response time of the kernel and can be run under different amounts of stress on the kernel. These benchmarks could be used to actually measure the benefits of using a patched real-time kernel, to using a default Linux kernel, with the default scheduler.

Another feature that needs to be tested and benchmarked is the communication network of the car. To test this there has to be congestion in the network switches. It is quite difficult to generate enough traffic to congest the gigabit switches of the platform. However it is possible to limit the bandwidth of the switches through the graphical configuration interface of the switches. Whether this is a valid way of testing and creating congestion has to be evaluated.

6.2.6 Creating test scenarios

Initially the plan was to get the platform to a stage where some simple test scenarios could be run. This was however not possible due to delays in the ordering process of the components. With the current sensor implemented some simple test scenarios could be run. An example could be to implement a simple auto brake function using the ultra sonic sensor to determine the distance to the nearest object. The tester could run the car at constant speed towards an object and the car shall stop before it hits the object.

Another simple test scenario using the ultra sonic sensor is to use it as a parking assistance sensor, to determine the distance to the object behind it. Additional test scenarios could be set up if the camera is fully implemented, the camera could be used to detect road signs or other obstacles and take action based on the findings.
6.2.7 Creating an API for external communication

As already discussed the platform is equipped with WiFi through the RPI 3 ECU. The WiFi allows for connection to any other interface using WiFi. Currently it is only set up to communicate with the Android Application. This could be extended to create an API that defines how data is passed between the platform and any other device. Such an API could be extended to support any monitoring and control interface, for example to access it through a PC. This API could be extended even further to allow the platform to communicate to other instances of the platform, to act as a V2V interface. Not only to other physical platforms but it could be used to communicate to emulated platforms running on a PC. Furthermore it could be extended to enable the platform to communicate to an infrastructure, enabling V2I communication. Figure 6.1 provides an overview of the use of an API for external communication.

![Diagram of external communication possibilities](image)
6.2.8 Improving the control and monitoring interface

Currently the car is controlled and monitored through a simple android application with limited capabilities. The monitoring is, as previously discussed, limited to only see the average ping and dropped packages of the internal car network. This could be extended further to monitor sensors, actuators and the ECUs of the car. Furthermore the control interface could be improved to be more complete. The way to control the steering and the speed is currently implemented using two sliders. This could be researched into to see what are the requirements to provide a good interface.
Bibliography


[24] p. m. m. Frédéric Gaillard and A. Andreas Eieland, senior product marketing manager, “Microprocessor (mpu) or microcontroller (mcu)? what factors should you consider when selecting the right processing device for your next design,” Article, 2013. [Online]. Available: http://www.atmel.com/Images/MCU_vs_MPU_Article.pdf


[70]


Appendix A

Code

Listing A.1: Example code controlling multiple PWM signals

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <pru_iep.h>
#include <pru_intc.h>
#include "servo_controller.h"

volatile register uint32_t __R30; /* output register */

int setup_servo(){ /* to init timer */
    CT_IEP.TMR_GLB_CFG.bit.CNT_EN = 0; /* Disable counter */
    CT_IEP.TMR_CNT = 0x0; /* Reset Count register */
    CT_IEP.TMR_GLB_STS.bit.CNT_OVF = 0x1; /* Clear overflow status register */
    CT_IEP.TMR_CMP_STS.bit.CMP_HIT = 0xFF; /* Clear compare status */
    CT_IEP.TMR_COMPPEN.bit.COMPPEN_CNT = 0x0; /* Disable compensation */
    CT_IEP.TMR_CMP_CFG.bit.CMP0_RST_CNT_EN = 0x0;
}
```
/ * Disable CMP0 and reset on event */
CT_IEP.TMR_CMP_CFG_bit.CMP_EN = 0x0; /* disable all comparators */
comparator_en = 0; /* make sure it is 0 */
CT_IEP.TMR_GLB_CFG_bit.DEFAULT_INC = 0x1; /* Configure incr value */
}

int set_global_period(int period_us) { /* use cmp0 - normally 20 ms / 50 hz */
    CT_IEP.TMR_CMP0 = CMP_US * period_us;
    CMP0.EN = 0x1;
    comparator_en = comparator_en | CMP0.EN;
}

int set_servo_0_duty(int duty_us) { /* use cmp1 */
    CT_IEP.TMR_CMP1 = CMP_US * duty_us;
    CMP1.EN = 0x2;
    comparator_en = comparator_en | CMP1.EN;
}

int set_servo_1_duty(int duty_us) { /* use cmp2 */
    CT_IEP.TMR_CMP2 = CMP_US * duty_us;
    CMP2.EN = 0x4;
    comparator_en = comparator_en | CMP2.EN;
}

int set_servo_2_duty(int duty_us) { /* use cmp3 */
    CT_IEP.TMR_CMP3 = CMP_US * duty_us;
    CMP3.EN = 0x8;
    comparator_en = comparator_en | CMP3.EN; /* bit 3 - enable the comparator */
}

int start_servos() {
    /* period setup */
    CT_IEP.TMR_CMP_CFG_bit.CMP_EN = comparator_en; /* enable all comparators needed */
    CT_IEP.TMR_GLB_CFG_bit.CNT_EN = 1; /* enable timer counter */
}

int stop_servos() {
CT_IEP.TMR_CMP_CFG_bit.CMP_EN = 0x0; /* disable all comparators */
__R30 = 0x0; /* set all pins to low */
return 0;

int check_servo_evt(){ /* to check if the timer caused an interrupt – and which of the comparators it was */
    if (CT_INTC.HIPIR1 == PRU_IEP_EVT){ /* if the timer is of highest prio */
        /* Identify which comparator caused the interrupt */
        if ((CT_IEP.TMR_CMP_STS_bit.CMP_HIT & CMP3_EN) == CMP3_EN){ /* if cmp3 fired the interrupt */
            __R30 = __R30 & 0xFFFFFFFFB; /* set bit 1 to 0 */
            CT_IEP.TMR_CMP_STS_bit.CMP_HIT = CT_IEP.TMR_CMP_STS_bit.CMP_HIT | CMP3_EN;
        }
        if ((CT_IEP.TMR_CMP_STS_bit.CMP_HIT & CMP2_EN) == CMP2_EN){ /* if cmp2 fired the interrupt */
            __R30 = __R30 & 0xFFFFFFFFD; /* set bit 1 to 0 */
            CT_IEP.TMR_CMP_STS_bit.CMP_HIT = CT_IEP.TMR_CMP_STS_bit.CMP_HIT | CMP2_EN; /* Clear event */
        }
        if ((CT_IEP.TMR_CMP_STS_bit.CMP_HIT & CMP1_EN) == CMP1_EN){ /* if cmp1 fired the interrupt */
            __R30 = __R30 & 0xFFFFFFFFE; /* set bit 0 to 0 */
            CT_IEP.TMR_CMP_STS_bit.CMP_HIT = CT_IEP.TMR_CMP_STS_bit.CMP_HIT | CMP1_EN; /* Clear event */
        }
        if ((CT_IEP.TMR_CMP_STS_bit.CMP_HIT & CMP0_EN)
== CMP0_EN\{ /* cmp0 — period — fired the interrupt */
    R30 = (CMP1_EN\gg1) | (CMP2_EN\gg1) | (CMP3_EN\gg1); /* set pins of all active pwms to zero */
    CT_IEP.TMR_CMP_STS.bit.CMP_HIT = CT_IEP.TMR_CMP_STS.bit.CMP_HIT | CMP0_EN; /* Clear event */
    CT_IEP.TMR_CNT = 0x0; /* reset counter */
\}

CT_INTC.SICR.bit.STS_CLR_IDX = PRU_IEP.EVT; /* clear timer event */
return 1; /* the timer caused an interrupt */
\}
return 0; /* The timer did not cause the interrupt */
\}