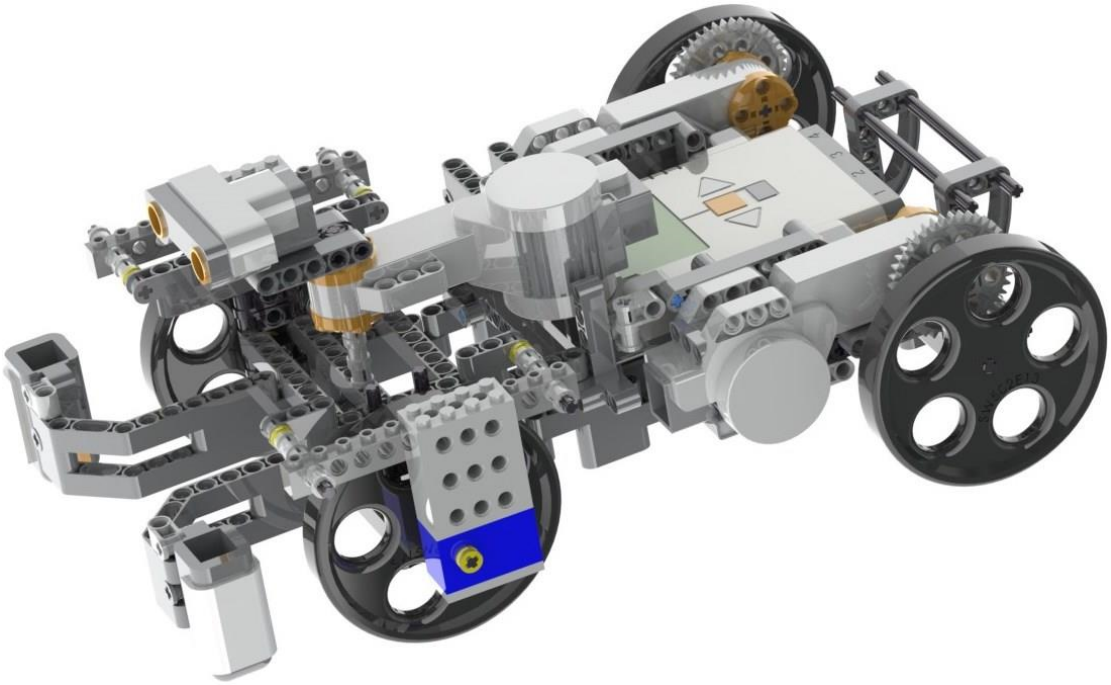
SW502E13

Vehicle Safety and Operational Assistance Systems

Embedded Systems



AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Software 5th semester
Address: Selma Lagerlöfs Vej 300
9220 Aalborg Øst
Phone no.: 99 40 99 40
Fax no.: 99 40 97 98
Homepage: http://www.cs.aau.dk

Project title:
Vehicle Safety and Operational Assistance Systems

Subject:
Embedded Systems

Project periode:
Autumn 2013

Group name:
SW502E13

Supervisor:
Jiri Srba

Group members:
Alexander Drægert
Anders Christian Kaysen
Christoffer Ndũrũ
Dan Skøtt Petersen
Frederik Bruhn Mikkelsen
Jesper Byrdal Kjær

Copies: 8

Pages: 108

Appendices: 1 & 1 CD-ROM

Finished: December 20th, 2013

Abstract:

The purpose of this report was to document the development of a real-time embedded system. The report demonstrates how the vehicle safety and operational assistance systems ABS, TCS, EDC, AEB, EAEB, LDES, and ACC can be implemented on a LEGO Mindstorms robot. The report also contains an analysis of the hardware of the LEGO Mindstorms platform, the system architecture of the real-time operating system used, and the analysis, and implementation of each of the safety systems. Additionally the report contains documentation on how a real-time analysis was performed for the embedded system. The completed system was scheduable, despite minor issues with some features. All features performed acceptably.

Preface

This report was finished in the winter of 2013, as the 5th semester project by software engineering students from the Department of Computer Science at Aalborg University, Aalborg, Denmark. The report documents the complete development and test of a real-time embedded system which provides vehicle safety features such as ABS and TCS using the LEGO Mindstorms platform.

References to sources are done using brackets, e.g. [23], with a number inside. The number is a reference to an entry in the bibliography located on page 107.

A CD is attached to the report, containing the source code, the TAPAAL model file, and this report in electronic form. Additionally the content of the CD can be downloaded as a ZIP file at:

`http://bit.ly/sw502`

The group would like to thank the supervisor, Jiri Srba, for guidance and feedback during the project.

Alexander Drægert

Anders Christian Kaysen

Christoffer Ndũrũ

Dan Skøtt Petersen

Frederik Bruhn Mikkelsen

Jesper Byrdal Kjær

Contents

CHAPTER 1

Introduction

Annually about 1.24 million people die, and between 20 and 50 million people are injured, because of traffic accidents around the world. Unfortunately this number is estimated to grow to 1.9 million by 2020 if no action is taken [20]. There may be many different approaches to mitigate this problem, but one of them is definitely to improve existing or develop new vehicle safety systems.

For several decades, vehicle safety systems have been mechanical in nature. Seatbelts, airbags, crumble zones, laminated windshields, and safety cell passenger compartments are all examples physical additions to a vehicle, that can make accidents significantly less dangerous [26].

But with the introduction of microcontrollers, it not only became possible to increase relative performance and ease of use by making smart engines and automatic gearboxes, but also to implement automatic crash avoidance systems. Suddenly the car was able to help avoid the potentially deadly mistakes, that a driver would typically make in the highly stressful situation of an impending crash.

Examples of these electronics-based safety systems are:

- Anti-lock braking is a system which ensures the vehicle does not lock any of the braking wheels, to prevent skidding.
- Autonomous emergency braking is a system that brakes the vehicle if an obstacle suddenly appears in front of the vehicle. It brakes without the driver touching the brake.
- Traction control is a system that prevents the wheels from spinning during acceleration.
- Electronic differential control is a system that helps the vehicle turn. This is done by decreasing the velocity on the inner driving wheel, making the outer driving wheel push the car around.

This report is about implementing a number of these systems. Some are very traditional, e.g. anti-lock braking and traction control, others are more unconventional, e.g. lane departure emergency system, a system that makes sure the driver does not inadvertently leave his/her lane.

CHAPTER 2

Problem Statement

As described in Chapter 1, a safety system could include autonomous emergency braking, traction control, electronic differential, anti-lock braking, automated parking system and rear view camera. These are all assists which help the driver handle the vehicle in different situations.

It is expensive, time consuming, and difficult to test these features on a real size vehicle. Therefore we would like to see how one could implement a chosen set of safety systems in a real-time system on a LEGO NXT supplied with sensors and parts from the LEGO Mindstorms series. We would also like to determine whether such a system can be scheduled in a real-time operating system. LEGO Mindstorms has some limitations in regards to simulating production vehicles, for example the best propulsion motor immediately compatible with LEGO Mindstorms is a primitive electrically powered servomotor, and LEGO is somewhat limited with regards to advanced mechanical constructions, e.g. brakes. However, the focus of this project is the software implementation of these systems, and the LEGO Mindstorms platform is well suited for creating a simple test bed for such systems.

> *How can an automatic system to improve traffic safety be implemented on a vehicle using the LEGO Mindstorms platform?*
>
> *Furthermore, is the task set of such a system schedulable, and if so, how can it be scheduled in a real time operating system?*

The report only focuses on the set of assists outlined below. These assists are combined into one system.

- Anti-lock braking system
- Traction control system
- Electronic differential control
- Autonomous emergency braking
- Extended autonomous emergency braking
- Lane departure emergency system
- Adaptive cruise control

CHAPTER 3

Hardware Analysis

In this chapter the specification of each piece of hardware used in the project, as well as its software interface, is described briefly in Section 4.3. All specifications are found in the official NXT User Guide [11]. In Section 3.4 the design of the vehicle is described. Even though all the specifications for each sensor are available, it is necessary to test each sensor individually, as described in Section 3.5.

3.1 NXT Brick

The NXT brick, as shown in Figure F3-1, is the brain of the LEGO Mindstorms system. The NXT brick has seven ports; three for servo motors (see Section 3.2) and four for sensors (see Section 3.3). The ports for motors are named A, B, and C, and the ports for sensors are numbered 1-4. The ports for motors cannot be used for sensors and conversely the ports for sensors cannot be used for motors.

Figure F3-1: LEGO Mindstorms NXT Brick.

3.2 Interactive Servomotor

The three servomotors, shown on Figure F3-2, allows the robot to move around. Each motor provides the robot with rotational feedback which allows the robot to control the motor with an accuracy of one degree. It is also possible to use it as a traditional motor, by making it turn continuously.

Figure F3-2: LEGO Mindstorms Servomotor.

Interface

To communicate with the motors and sensors the ECRobot API is used. Chikamasa [1] explains all the available functions in detail, while only the ones essential for this project are mentioned here. To interface with the servomotors two functions are used. `nxt_motor_get_count()` takes a port id as a parameter, and returns the number of degrees turned since system startup. If the motors are set to reverse, this number will decrement and eventually become negative. `nxt_motor_set_speed()` needs three parameters: A port id, the desired speed in percent, -100 to 100, where negative numbers mean

reverse, and lastly a mode, 1 for brake or 0 for float. In brake mode, the motor will block when the speed is set to 0, and in float mode it is still allowed to turn. This function has no return value.

3.3 Sensors

The NXT Mindstorms sensors allow the NXT brick to sense and act on the environment around it. In this section the sensors used in this project are described.

3.3.1 Light Sensor

The light sensor enables a LEGO robot to distinguish between light and dark, meaning it is able to measure the light intensity either in a room, or on coloured surfaces. Figure F3-3 shows the sensor.

Figure F3-3: LEGO Mindstorms Light Sensor.

The light sensor module has a built in LED in addition to the light sensor itself, allowing the sensor to emit red light, to be used when the sensor is in reflected light mode. In this mode the sensor measures how much light differently coloured surfaces reflect. If the sensor is in ambient light mode the LED is turned off, and the sensor measures the amount of ambient light in front of it.

Interface

For interfacing with the light sensor one needs three different functions. `ecrobot_set_light_sensor_active()` takes a port id as a parameter. It should be called when the device is initialised, and it has no return value. Similarly `ecrobot_set_light_sensor_inactive()` takes a port id as a parameter, and should be called when the device is terminated. `ecrobot_get_light_sensor()` takes a port id as a parameter and returns a value between 0 and 1023 depending on the light level. A higher value means it is darker (or there is less reflection).

3.3.2 Ultrasonic Sensor

The ultrasonic sensor, Figure F3-4, gives the NXT the ability measure distances. The sensor works by sending out an ultrasonic sound, and calculating the time for the sound to return. Using the speed of sound, it is possible to calculate the distance to an object.

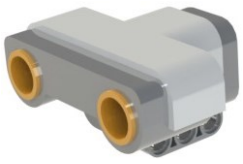


Figure F3-4: LEGO Mindstorms Ultrasonic Sensor.

According to the specifications, the sensor measures the distance in centimetres and it is able to measure a distance from 0 to 250 cm with a precision of 3 cm. Large objects with hard surfaces yield the best results, while smaller objects with curved or soft surfaces, such as fabric, are harder to detect.

Interface

Similar to the light sensor, the ultrasonic sensor has a function to initialise it, `ecrobot_init_sonar_sensor()`, and a function to terminate it `ecrobot_term_sonar_sensor()`. These take a port id as a parameter, have no return value, and should be called when the device is initialised and terminated, respectively. To obtain the distance read by the sensor, `ecrobot_get_sonar_sensor()` is called with a port id as a parameter. It returns -1 if it is not ready for measurement, or a value between 0 and 250 representing a distance in cm. If the distance is greater than 250 cm, it returns 255.

3.3.3 RCX Rotation Angle Sensor

This sensor is not described in the NXT User Guide, but instead in the help files included in the zip file containing support for legacy sensors [12]. The rotation angle sensor makes it possible to count how many times a wheel or similar part has revolved. It was made for a previous version of Mindstorms called RCX, and therefore requires a special conversion cable for communication with the NXT. Furthermore, it returns raw data, that needs to be processed before use. The sensor counts 16 ticks per revolution, and returns an integer representing the current tick. The sensor is only able to return four different values, thus each value is repeated four times in a single revolution as seen in Figure F3-6.

Figure F3-5: LEGO Mindstorms RCX Rotation Angle Sensor.

Interface

The sensor has to be initialised with `ecrobot_set_RCX_power_source()` with a port id, and terminated with `ecrobot_term_RCX_power_source()` with a port id. To get the raw data, `ecrobot_get_RCX_sensor()` is called with a port id. It returns an integer corresponding to a tick from 1 to 4. To count the number of revolutions it has turned the number of ticks must be counted. Section 5.3 explains how the ticks can be counted, and used to implement a speedometer for the vehicle.

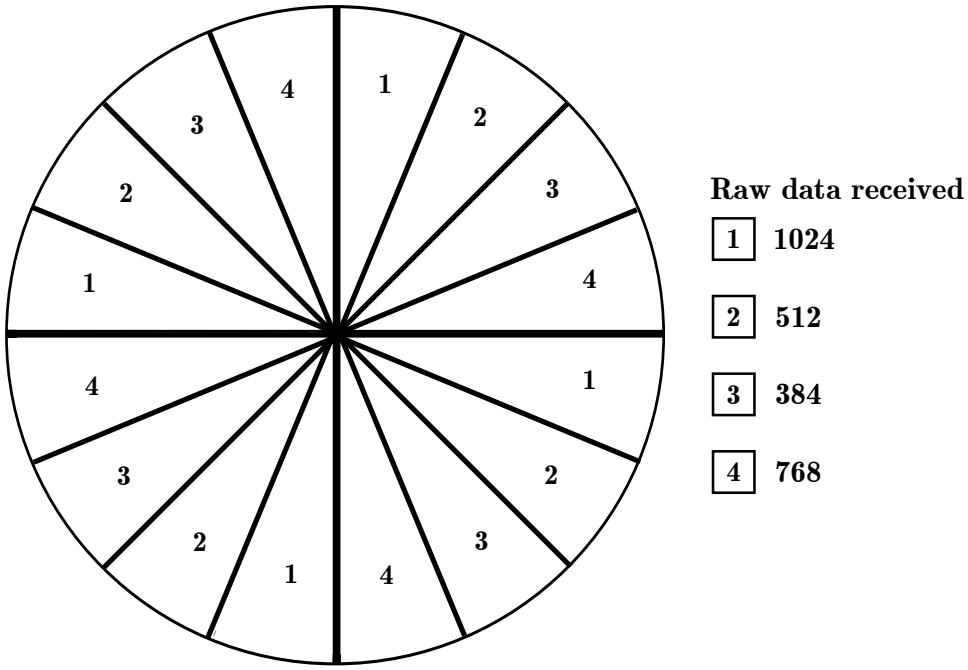


Figure F3-6: Illustration of return values from rotation angle sensor.

3.4 LEGO Vehicle

We based the design of the vehicle on the NXT GT model [2]. We also followed their example of constructing customized wheels, since the LEGO wheels with their rubber tyres have too much friction. The excess friction results in the vehicle not being able to drift, which is essential to prove the effectiveness of several of the implemented systems. A model of the vehicle is shown in Figure F3-7. Three servo motors are used on the vehicle. Two of them are each attached to a rear wheel, and the last motor is attached to the front wheels for the steering. A few modifications are made to the original NXT GT design. The motors attached to the rear wheels are turned upside down to allow replacing one of the gears with a smaller one to give the vehicle an increased top speed. Additionally the two light sensors have been moved to the sides in front of each wheel to detect the road markings of a traffic lane, as explained in Chapter 11. We have added the rotation sensor on the left front wheel to measure the actual velocity of the vehicle, as explained in Section 5.3. The ultrasonic sensor is used to detect objects in front of the vehicle, which is used in Chapters 9, 10 and 12.

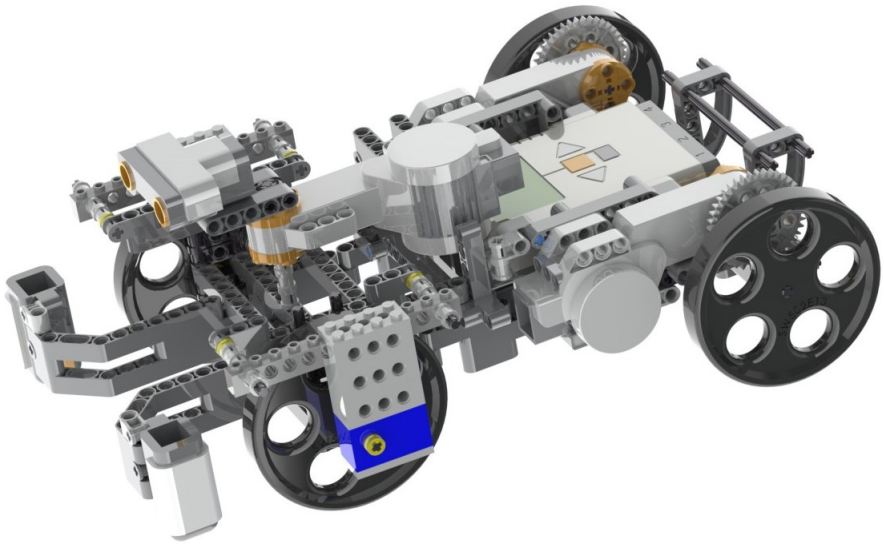


Figure F3-7: 3D Model of our LEGO vehicle.

3.5 Sensor Tests

To evaluate the performance of the different sensors used on our vehicle, we designed a number of tests. The main purpose of the tests were to evaluate the precision and reliability of the sensors. While one could perform the tests on all the individual sensors, to determine their relative precision, e.g. if all the servomotors are equally precise, we decided that this would be unnecessarily time consuming.

Standard Deviation

When evaluating the tests, we used the concept of *standard deviation*. Standard deviation is calculated from a set of data, and indicates the spread of the data, relative to the average. A large standard deviation indicates a wide spread of data, and a small standard deviation indicates a small spread of data. In each test we made the sensors perform a large number of measurements for each test case, and then calculate the mean and standard deviation of the results. The standard deviation can thus give an estimate of the precision and reliability of the sensor under test.

3.5.1 The Servomotors

We identified two properties of the servomotors that were interesting to test: How precise the velocity of the motors are, relative to each other, and how precise they measure their own movement. The second property we tested along with the rotation sensor, as described in Section 3.5.2. The first property is interesting, as the vehicle's ability to drive in a straight line is highly dependent of the motors being able to rotate with the exact same speed. From this point of view, any deviation at all between the two driving motors is unacceptable, as this would make the vehicle drift to either side.

In the test, we started the two servomotors at the same time. When motor A reached exactly 1,000 ticks, both motors were stopped. The number of ticks reached by motor B would then be the result of the test. This process,

repeated ten times, constituted one test. The test was repeated with different power settings, to determine whether the precision is better at lower speeds.

Power	Average	Difference	SD
100 %	970.0	3.0 %	1.49
90 %	979.7	2.0 %	1.63
80 %	978.2	2.2 %	1.81
70 %	978.6	2.1 %	2.59

Table T3-1: Results of testing the servomotors. SD refers to the standard deviation.

The results are described in Section 3.5.1. It is clear that motor B runs slower than motor A. In less than three complete revolutions, motor B is already 2-3 % behind motor A. However the standard deviation shows that the results at least are somewhat consistent, which should make it easier to rectify the problem, e.g. using different power ratios for the two motors. It should also be noted that motor B was significantly slower at 100 %, compared to the other power settings, indicating that motor B also might have a top speed lower than motor A.

3.5.2 The Rotation Sensor

The main question concerning the rotation sensor, is its precision. This is tested by turning it a specific number of revolutions, and check how many revolutions the rotation sensor reports it has turned. The problem is how to precisely turn it a large number of revolutions. We decided to mount it on the servomotor. While the servomotor itself might have precision issues, we decided that it was unlikely that they would both be equally imprecise, as they are different components. It seems reasonable to assume that if they both provide the same result, they are both reasonably precise.

As in the servomotor test, we make the motor turn 100 revolutions, and record the number of revolutions counted by the rotation sensor. This is repeated ten times, after which we can calculate the mean and the standard deviation of the result set.

Average	Difference	SD
99.0	1.0 %	0.75

Table T3-2: Results of testing the rotation sensor. SD refers to the standard deviation.

The results are described in Section 3.5.2. While the rotation sensor is a little off in relation to the servomotor, it is quite consistent. We intend to use the rotation sensor for measuring the vehicle's velocity. For this purpose, the granularity of the results (16 ticks per revolution) should be a greater source of precision loss, than a 1 % offset in revolution count in relation to the servomotors.

3.5.3 The Light Sensor

Before starting formal testing of the sensors, preliminary use of the light sensor showed that it was difficult to use. Its results are highly sensitive to

change in nuances of the ground colour, and to ambient lighting.

We chose to test how different the results would be, depending on the colour of the ground (white, grey or black) and lighting (indoor and outdoor lighting, denoted dark and light respectively). Furthermore we repeated the set of tests with a paper "shield" around the sensor, to lessen the influence of ambient light on the result. Finally we drove it a set distance over a floor of different grey nuances, to determine how much movement would influence the results. For all tests we used the built-in light to lessen the influence of ambient light. Each measurement was repeated 10,000 times, and the mean and standard deviation are calculated from these sets of results.

No movement				
Environment	**Without shield**		**With shield**	
	Average	**SD**	**Average**	**SD**
White, dark	536.07	0.44	521.06	0.24
White, light	461.07	0.69	517.03	0.19
Grey, dark	662.96	0.61	641.99	0.19
Grey, light	623.91	0.65	634.79	0.40
Black, dark	714.92	0.59	721.02	0.24
Black, light	724.58	0.58	703.13	0.54
Moving (grey, dark)				
Tests	**Without shield**		**With shield**	
	Average	**SD**	**Average**	**SD**
Run 1	655.67	4.27	641.50	3.86
Run 2	653.32	4.73	625.42	3.90
Run 3	653.53	5.07	627.2	3.92

Table T3-3: Results of testing the light sensor. SD refers to the standard deviation.

The results are described in Section 3.5.3. When measuring a fixed point, the light sensor seems quite precise, judging from the standard deviations. It can clearly distinguish between the three colours, although the nuances resulting from differing ambient light gives all three colours a large span of readings. However, the moving measurements are somewhat more relevant, as we expect the sensor the to measure continually while the car is moving. But also in this case, the standard deviations indicate only small differences in the measurements, although they are significantly larger than the static measurements. In both situations, the shield clearly improves performance, by reducing both the standard deviation, and the significance of ambient light. Overall the light sensor seems to perform satisfactorily for our project, since it can clearly distinguish between significantly different colours - especially with the addition of the shield.

3.5.4 The Ultrasonic Sensor

The testing of the ultrasonic sensor mainly concerned the precision of the measurements. It was conducted by placing the sensor a known distance from a wall, make it perform 500 measurements, and calculate the mean and standard deviation from the results. Each measurement was performed

25 milliseconds apart, as preliminary experiments had shown that a shorter period between each measurement would result in erroneous readings. This is presumably due to the ultrasonic waves not being able to return the results in time.

Distance	**Average**	**Difference**	**SD**
5 cm	6.98	39.6 %	0.31
10 cm	11.97	19.7 %	0.53
20 cm	20.95	4.75 %	0.93
50 cm	49.90	0.20 %	2.23
70 cm	69.86	0.20 %	3.13
100 cm	100.51	0.51 %	4.52
150 cm	149.70	0.20 %	6.70
200 cm	200.97	0.49 %	9.63
250 cm	249.92	0.03 %	11.23

Table T3-4: Results of testing the ultrasonic sensor. SD refers to the standard deviation. Only a subset of the tests are shown, as all the tests displayed the same tendencies.

The results are described in Section 3.5.4. First of all, the tests show that the sensor is significantly less precise when measuring distances less than 20 cm. Furthermore, we see that while the average result is quite precise above 20 cm, the standard deviation grows with the distance, meaning that as the distance grows, the reliability of the individual measurement declines. Although not shown here, we also observed that at distances around 2 m and above, there is a chance that the sensor might not detect the obstacle at all. Instead it returns its maximum value, 255.

Overall, the performance of the ultrasonic sensor is not impressive, but we do not expect to use outside the range of 0.5-2 m, where it seems to perform most reliably. However, the measurements will probably need to be evaluated by the software, so the most differentiating results are discarded.

CHAPTER 4

System Architecture

In this chapter, the software and operating system used in the project, is described.

First Section 4.1 describes real-time system terminology. Afterwards, a real-time operating system is described, including which options were available for the project.

Lastly the choice of operating system is described, along with the features it provides.

4.1 Real-time Systems Terminology

In this section the general terminology regarding real-time systems is explained.

4.1.1 Tasks

In real-time operating systems there is no distinction between processes and threads. Where general purpose operating systems typically host a large number of highly diverse tasks with varying degrees of interdependency, the real-time operating system usually only solves a few well-defined and closely related tasks, making the distinction between processes and threads less useful. Instead the term "task" is used in the system.

There are three types of tasks: periodic, aperiodic and sporadic.

Periodic tasks are released every x milliseconds, whereas aperiodic are released irregularly. The release of an aperiodic task is typically caused by an event, such as the system reaching a certain state, or the system receiving input from the user. Sporadic tasks are a subtype of aperiodic tasks, characterized by having a minimum interarrival time, i.e. multiple instances of the task cannot be released at the exact same time.

Priorities

Priorities are used to specify in which order the tasks in a system must be executed. The priorities can be static or dynamic, depending on the type of scheduler. Static priorities are assigned one time, and cannot be changed during runtime. Dynamic priorities may be changed multiple times during program execution, for example in Earliest Deadline First scheduling described in Section 4.1.2.

In this project 1 is the lowest priority, and higher numbers denote a higher priority.

Periods

Every task has a period that specifies the frequency with which a periodic task is released. If a task A has a period of 50 ms, it is released every 50 ms.

Deadlines

Deadlines specify the amount of time a task has, starting from release time, until it has to be finished executing. It is important to distinguish between deadlines and periods, because deadlines can be shorter, equal to or greater than periods. If a task has a deadline greater than the period, the system may have a growing queue of released tasks, waiting to be executed.

4.1.2 Scheduling

Scheduling refers to the job of deciding which task should be allowed to use the processor. Scheduling is an ongoing job, as old tasks terminate and new tasks become ready for processing. Scheduling is implemented as an algorithm (called the scheduler), which is executed in response to certain events. For example a task may become ready for execution following a hardware interrupt, such as a concluded IO operation. As this new task may have a priority higher than the currently running task, the tasks need to be rescheduled.

Different types of operating systems have different approaches to scheduling, but in real time systems scheduling is usually as time-efficient as possible in order to minimise the overhead of calling the scheduler. Aside from that, the all-important concern is that all tasks are terminated within their deadline.

Cyclic Executive

The cyclic executive scheduling method uses a predefined set of tasks. The idea is to create a static schedule that satisfies all task deadlines. Executing this schedule will guarantee that all tasks meet their deadline every time.

If it is possible to create a cyclic executive scheduler for the tasks in a system, it proves that the task set is schedulable.

In the cyclic executive scheduler, minor and major cycles are often used. A minor cycle can be determined by calculating the greatest common divisor of the periods of all tasks. Also, minor cycles are used as synchronization points, which ensures that the schedule is synchronized with the system time. A major cycle can be calculated by finding the least common multiple of all task periods. However, it might not always be practical to determine the major and minor cycles with these methods.

Earliest Deadline First

Earliest deadline first (EDF) is a scheduler that uses dynamic priorities. In EDF, every time a new task is released, new priorities are assigned to tasks in the ready state. The task with the earliest deadline gets the highest priority, and the task with the latest deadline get the lowest priority.

Rate Monotonic

Rate monotonic scheduling uses static priorities, that are assigned before runtime. The priorities are based on the period of the tasks. The task with the shortest period has the highest priority, and the task with the longest period has the lowest priority. Rate monotonic does not take execution time into account.

Process	Period	Priority
A	30	4
B	60	2
C	50	3
D	15	5
E	140	1

Table T4-1: An example showing the priorities using rate monotonic scheduling.

In this project rate monotonic scheduling is used, because it is the standard scheduling algorithm in nxtOSEK [4].

4.1.3 Preemption

Preemption is a mechanism that makes it possible to reassign the processor to a task with a higher priority, by interrupting the running task and performing a context switch. This ensures that it is always the task with the highest priority that is executing.

Example 4.1 shows a simple example of preemption.

Example 4.1 This example is based on tasks A and B from Table T4-1 and is illustrated in Figure F4-1.

Task B is released, and as it is the only task in the ready state, it executes. Before B finishes executing, A is released, and because of its higher priority, B is preempted and the processor is assigned to task A. When A finishes or releases the processor, B can continue to execute.

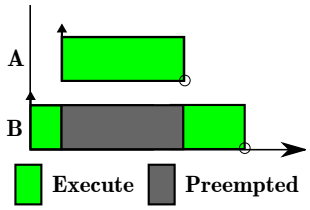


Figure F4-1: Illustation of preemption, described in Example 4.1.

4.2 RTOS and GPOS

The main difference between a general purpose operating system (GPOS) and a real-time operating system (RTOS) is, besides their purpose, their philosophy of scheduling.

In the GPOS, the exact order in which the tasks are scheduled, as well as when and for how long they are executed, is less important. Factors that matter in a GPOS are that all ready tasks get a chance to execute within

a reasonable amount of time, and that the front end feels responsive to the user.

In a RTOS however, timing is all-important. As a real-time system is supposed to sense and respond to the real world, it is unacceptable that the task responsible for actuating a time-critical response "hangs" because the OS is using the processor for less important tasks. Therefore, a RTOS scheduler focuses entirely on the priority and timing of the tasks it handles. This has the effect of RTOS schedulers being much more predictable. This is opposed to the scheduler used in a GPOS, whose complex rules and enormous amount of tasks with hugely different behaviours often lead to it being described as nondeterministic. If one knows the worst case execution time and implementation information of all tasks in the system, one can model the scheduling, and evaluate whether the tasks are scheduled satisfactorily. If so, one has obtained a guarantee that the system will always execute the tasks in a timely manner.

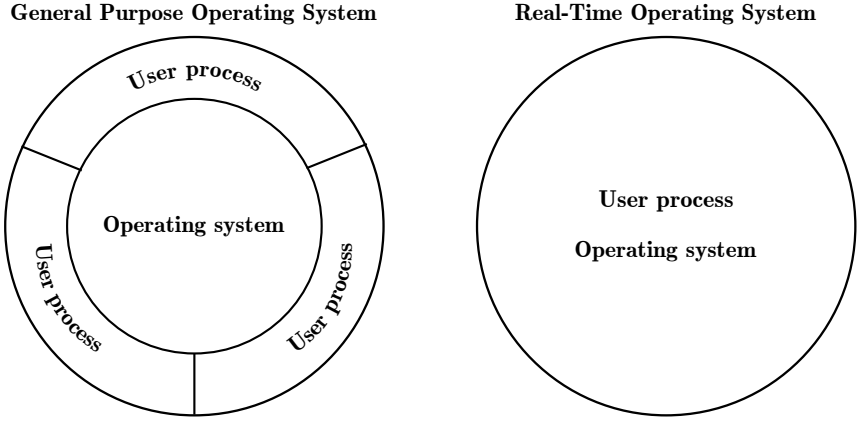


Figure F4-2: This figure illustrates the difference between a GPOS and a RTOS.

Another difference is the way the operating system and the user processes are structured in the system.

As seen in Figure F4-2 the user processes in a GPOS are handled in a software layer separate from the operating system processes, where in a RTOS the user processes are mixed with the operating system processes.

4.2.1 Choice of Operating System

There is a large number of languages available when developing for the NXT platform [27]. The main requirement for this project was that the programming language, and its underlying implementation, should be able to execute in real-time. This rules out interpreted languages ported for the NXT, such as pbLua (Lua) [23].

Furthermore, members of the group already had programming experience in C and Java, which is why it was appropriate to choose an alternative based on one of these, instead of the original NXT programming language.

If it was decided to develop using Java for the NXT, lejosNXJ [13] would be an obvious choice. lejosNXJ is a Java-like language which uses custom firmware. The problem with lejosNXJ for real-time purposes, is that Java is not run natively, but in software (a Java Virtual Machine). Therefore it

cannot provide a timer with real-time guarantees [14]. This obviously makes it unsuitable for any real-time purposes.

The programming language Ada is also an option when programming a real-time project for the NXT. None of the group members have any experience with Ada, and it was deemed too time consuming to learn a new language *and* develop the system, due to the time limit of the semester.

Using nxtOSEK, it is possible to program the NXT in C with real-time performance. Additionally, nxtOSEK is an open source implementation of the OSEK standard (a more detailed description of OSEK is found in [22]), which is a standard developed by the German car industry. Therefore it is an ideal OS, since the project concerns vehicle safety systems.

nxtOSEK was chosen for developing the project because of its adherence to the OSEK standard, its real-time guarantees, and its use of C as implementation language.

4.3 nxtOSEK

We decided to use nxtOSEK as the operating system and development platform. nxtOSEK is an open source implementation of the OSEK standard for the LEGO Mindstorms NXT platform [3]. While there are several programming frameworks and operating systems available for the NXT platform, we found nxtOSEK to be the most appropriate for our project, due to its adherence to the OSEK standard.

4.3.1 OSEK

The following description is based on [22].

Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen (abbr. OSEK, translates to "Open Systems and their Interfaces for the Electronics in Motor Vehicles") is a standard for software architectures in automotive control systems. OSEK was first created in 1993 by a number of corporations in the German automotive industry. In 1994 the standard was renamed OSEK/VDX, as a few corporations in the French automotive industry contributed with a new approach called VDX (For ease of reading, we will simply call it OSEK in this report). At the time of OSEK's creation, the embedding of control software into vehicles was a major expense, as different systems were developed by different manufacturers, with no common interface. Making these systems work together in the same vehicle became a large expense in itself. Therefore, they created OSEK as a standardized interface for real-time systems, making development and integration of new systems easier.

4.4 Tasks

A task is a framework for the execution of functions. By using tasks instead of functions it is possible to divide and execute instructions based on their real-time requirements, e.g. needed measurements from sensors.

OSEK supports two types of tasks, basic and extended tasks. We use extended tasks, which are modelled as shown in Figure F4-3. An extended task can be in one of four states:

- *Running* is when the CPU is executing instructions from the task.
- *Ready* is when the task is ready for the CPU to execute its instructions.
- *Waiting* is when a task needs further information before it can continue executing.
- *Suspended* is when a task is passive and can be activated.

When a task is created, it is in the *suspended* state. When the task is activated it changes state to *ready*. When the scheduler gives processor time to the task, it changes state to *running*. When a task is running, one of two things can happen:

- The task can be in need of more information e.g. measurements from sensors, whereafter the task would change state to *waiting*, so other tasks can be executed. When the task has received the needed information, the state changes from *waiting* to *ready*.
- When all the instructions of the task have been executed, it terminates and changes state to *suspended*. It is crucial that all tasks explicitly terminate themselves, otherwise it will cause unpredictable behaviour.

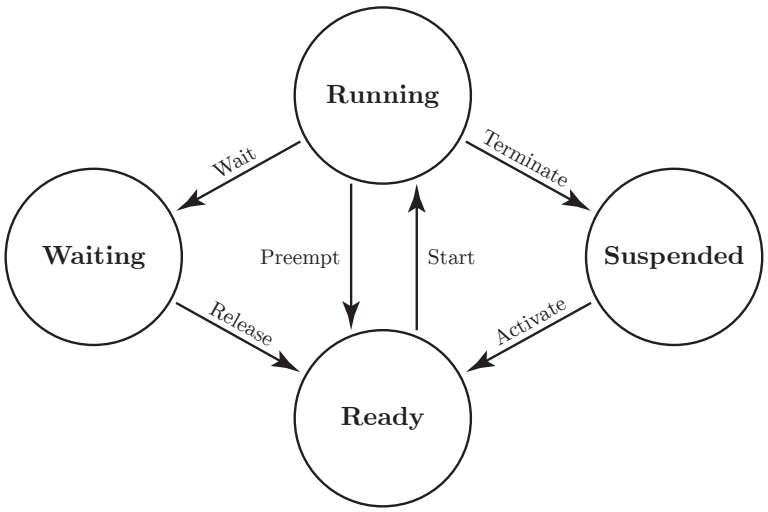


Figure F4-3: Model of states possible for an extended task.

The decision of using extended tasks was made, because the extended tasks are able to be in a waiting state. This comes in handy when a task needs further information to continue running, e.g. measurements from a sensor. Meanwhile the processor is allowed to use its time to execute instructions from other tasks.

4.5 Scheduling in OSEK

This section explains the mechanics of task scheduling, described in the OSEK specification [21]. We start by describing how OSEK uses task priorities, followed by the scheduling policy itself.

4.5.1 Task Priorities

In OSEK, tasks are assigned fixed priorities by the developer. This implies that one cannot use dynamic priority scheduling algorithms in OSEK systems. The OSEK consortium made this choice to ensure low overhead in the scheduler implementations. However, as the priorities are manually assigned by the developers, it is possible to use the range of fixed-priority scheduling algorithms, including rate monotonic and deadline monotonic scheduling. OSEK allows multiple tasks to have the same priority.

The scheduler organises the tasks into FIFO queues, a queue for each priority. The queues only contain tasks in the ready state. Therefore, the task in the front of a queue, is the task of that priority which has been in the ready state the longest. When the processor becomes available to a new task, the scheduler selects the forward-most task in the highest priority non-empty queue. Only when that queue, and all queues of higher priority, are empty, the scheduler allows tasks in the next lower priority queue access to the processor.

This is shown in Example 4.2.

Example 4.2 This example is visualised in Figure F4-4.

1) Task T_1, as the first and only task in the priority 2 queue, is taken to the processor for execution.

2) When T_1 leaves the running state, the priority 2 queue is empty, and the scheduler starts emptying the priority 1 queue. Therefore T_3 is moved to the processor.

3) T_1 becomes ready again, and the associated interrupt causes a rescheduling.

4) T_3 is preempted and moved back to the priority 1 queue, as T_1 is again moved to the processor.

5) T_1 terminates, and T_2, as the oldest task in the priority 1 queue, is moved to the processor.

6) T_2 ends execution, and the next task to be executed is T_3. When T_3 is done executing, T_4 will be executed, assuming no other tasks of priority 2 or 1 become ready in the meantime.

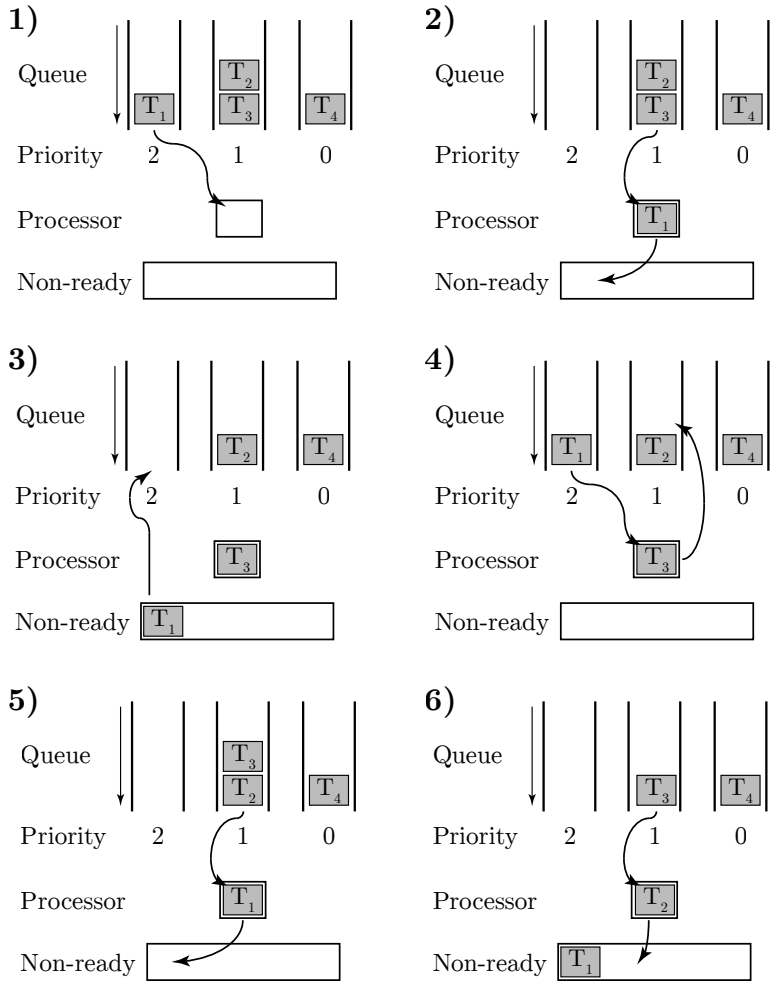


Figure F4-4: Example of scheduling in OSEK. The non-ready state contains all tasks which are not ready to execute, i.e. tasks that are suspended or waiting.

In OSEK, the lowest priority is 0, and higher numbers indicate higher priority. In nxtOSEK however, the lowest allowed priority is 1.

4.5.2 Scheduling

OSEK has several options for influencing scheduling, both by defining scheduling rules for individual tasks, and executing commands inside tasks. Commands may affect scheduling of the task itself, and other tasks in the system. Most of these possibilities are not used in this project, and thus not described in this report.

First of all, OSEK requires to define a scheduling policy for each task, either full preemptive or non preemptive. With full preemptive scheduling, the task may be preempted at any point during its execution, if a higher priority task enters the ready state. This is of course opposed to a non preemptive task, that can only leave the processor when it enters the waiting state, or terminates itself. In this project however, we have no reason to use non preemptive tasks.

So in a fully preemptive system, rescheduling is performed in several situations, of which the most important in this project are:

- The currently processed task terminates itself.

- An interrupt has been handled, and the system is to resume task execution. Interrupts include alarms that activate new tasks, or hardware interrupts that may take tasks from the waiting state to the ready state.

4.6 Resource Management

This section is based on [21].

Resource management (RM) is the part of OSEK used to coordinate access to shared resources. RM ensures that a resource can only be accessed by one task at a time, and precludes deadlocks and priority inversion.

It also makes it possible to use the scheduler as a resource. To avoid a task being preempted, the task can lock the scheduler resource, which means no other task can acquire the scheduler resource, hence avoiding preemption. A restriction that RM introduces is, if a task requires more than one resource simultaneously, then the resources have to be acquired and released following the LIFO principle.

Acquiring and releasing resources may cause both deadlocks and priority inversions. These concepts are described below.

Deadlock

Deadlocks occur when a system is in a state from where it waits infinitely, i.e. it will never continue from its current state. This is handled by the OSEK Priority Ceiling Protocol.

Example 4.3 and Figure F4-5 illustrates a deadlock scenario.

Example 4.3 We have resources R_1 and R_2 and tasks T_1 and T_2, with T_2 as the highest priority task. T_1 is released and acquires resource R_1. Then T_2 is released, and because it has the highest priority it preempts T_1, and T_2 is now running. T_2 acquires resource R_2 and also requires resource R_1. Because task T_1 already has acquired resource R_1, task T_2 cannot acquire R_1 and must therefore wait. Now T_1 continues from where it was preempted. But now task T_1 requires resource R_2, which is acquired by task T_2. This means both T_1 and T_2 ends in a wait state, and will never continue. This is called a deadlock.

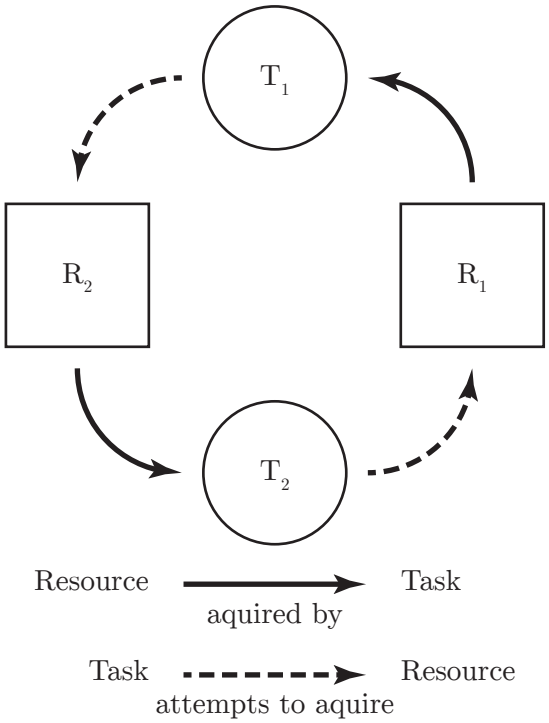


Figure F4-5: Illustration of Example 4.3.

Priority Inversion

Priority inversion is when a high priority task is released, but blocked by one or more lower priority tasks, because they need the same resources, and the lower priority tasks has acquired the resources first.

Example 4.4 describes a priority inversion.

Example 4.4 For this example we define two tasks T_1 and T_2 and a resource `Q`. Priority 1 is the lowest.

Process	Priority	Execution Sequence
T_1	1	`QQQ`
T_2	2	`EQ`

This example is illustrated in Figure F4-6.

T_1 is released first, acquires resource `Q` and executes for one unit. Then T_2 is released which results in preemption of T_1 because it has higher priority than T_1. T_2 then executes for one unit of time before it tries to acquire `Q`. This fails because T_1 has not released `Q` and T_2 is blocked. The scheduler then lets the next ready task with the highest priority execute. In this case it is T_1. T_1 then finishes execution in two units, and releases `Q`. T_2 can then acquire `Q` and finishes executing.

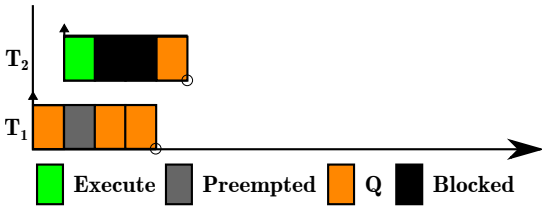


Figure F4-6: This figure illustrates the priority inversion Example 4.4.

OSEK Priority Ceiling Protocol

The OSEK Priority Ceiling Protocol prevents priority inversion and deadlocks by assigning a ceiling priority for each resource. This value is determined for resource R, by taking the priority from the highest priority task trying to access R, and assign a ceiling priority for resource R higher than this. Also, the ceiling priority of R has to be lower than the priority of all tasks that do not access R, and have a priority higher than the highest priority task that access resource R.

When a task acquires a resource, the task's priority is raised to the ceiling priority of that resource. When the task releases the resource, the task's priority is lowered to the originally assigned value of that task.

Time delays can occur using priority ceiling, because a high priority task can be blocked by a low priority task if the two tasks require the same resource. The time delay is limited by the maximum running time of the low priority task, as when the low priority task releases the resource, the high priority task can acquire it.

Example 4.5 describes the use of priority ceiling.

Example 4.5 In this example the genereric resources `Q` and `V` are used. Also, generic tasks $T_1..T_4$ are used. Each task has a priority corresponding to its number so T_3 has the priority 3. Since both resource `Q` and `V` are used by task T_4 they both have a ceiling priority of 5 or higher.

This example is illustrated in Figure F4-7.

T_1 is released and executes for one unit, then acquires `Q` which gives T_1 a temporary priority of 5. Before T_1 releases `Q`, tasks T_2, T_3, and T_4 are released, but are blocked by task T_1. When T_1 releases `Q`, its original priority is restored. Now T_4 has the highest priority and preempts all other tasks. The tasks are now run according to their priority.

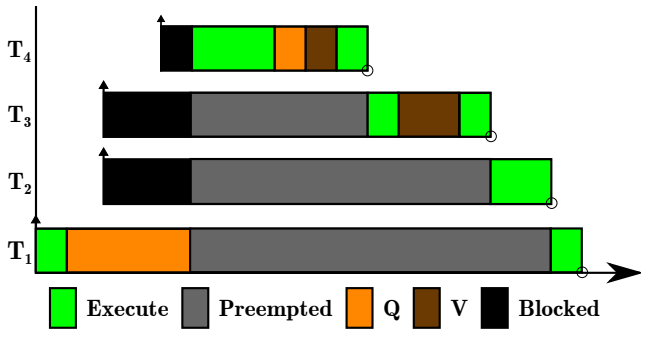


Figure F4-7: This figure illustrates priority ceiling.

4.7 Alarms

In the OSEK operating system alarms are provided as services for taking action on recurring events. Such alarms could for example be timers throwing interrupts with certain intervals.

To throw the interrupts with certain intervals, these events are registered in the OSEK operating system, by implementation specific counters. With these counters, the operating system gives the developer the possibility to attach alarm mechanisms to the application.

In the OSEK operating system a counter is represented by a value, which is incremented by each system tick, and some constants which are specific to the counter. The source of a counter could be the system clock. On the NXT a system tick is one millisecond, meaning that a counter with the system clock as its source, would increment every millisecond. Although for each counter you declare you are able to set the base of each tick in that counter. This means that you can make a counter where each tick is five milliseconds.

The operating system is responsible for incrementing the counter and it also manages the actions of the alarm.

Management of alarms

Alarms are managed by the OSEK operating system, which also provides services to:

- Set events.
- Activate tasks.
- Call an alarm-callback function.

Since we in this project only use alarms to activate tasks, the alarm-callback function will not be described further.

An alarm is triggered when the attached counter reaches a predefined value. It is possible to define this value as either relative to the actual counter value or as an absolute value. When the predefined value is relative to the actual counter value it results in in a relative alarm. However, if the predefined value is absolute the alarm is an absolute alarm. Alarms can be defined as either single alarms or cyclic alarms. In this project cyclic alarms are used. Besides defining alarms, the operating system provides functionality to stop an alarm, and get the current state of an alarm.

A counter can have multiple alarms attached to it. Upon system generation, alarms are statically assigned to a statically defined counter and a statically defined task. Not only are alarms statically assigned to counters, the actions to be performed at the alarm trigger, are also statically defined.

4.8 Testing

Unit testing was relevant to the project for several different reasons:

- We chose a test-driven development model, in which test cases were developed prior developing the actual production code.

- A real-time system is particularly difficult to test with the "write-compile-run-observe" cycle, that one might find sufficient for small desktop programs.
- The behaviour of real-time systems is often highly dependent of outside factors, that can be difficult simulate consistently.
- The development team may only have access to a limited number of the hardware platforms the system is supposed to work on, denying the team the ability to let every developer test continuously on the target platform.
- Unit testing guarantees that the program logic continues to be valid during code refactoring (depending on the correctness and coverage of the test cases themselves).

Unit testing provides a method for continually verifying that individual units of the system (in the case of C, a unit is typically a function) live up to their requirements, ideally independently of outside dependencies, i.e. hardware and other software components.

4.8.1 Test Framework

For this project we used the Unity test framework [10]. While there are several test frameworks for C/C++, Unity is one of the few designed specifically for C, and is thus quite minimalist, compared to frameworks that also cover C++. Furthermore, we found Unity quite easy set up and use, as much of the code for organizing and running the tests is automatically generated.

4.8.2 Mocking

An important aspect of unit testing is the ability to isolate the code under test (abbr. CUT) from outside dependencies. In our case, this especially included the ECRobot interface, used for calls to the hardware. If this dependency was not removed, the tests would, besides not being able to run on the development computers, be highly dependent of the behaviour of the sensors and actuators.

To resolve this problem, we *mock* the ECRobot interface, that is, create an implementation that only serves to allow the CUT to compile and run. However, there are even greater benefits of using mocks. As we now control the behaviour of what the CUT believes to be the hardware interface, we can both control what values the CUT receives from the hardware, and retrieve what values the CUT sends to hardware. This allows us nearly total control of the environment in which the CUT runs, and ensure that it acts appropriately in all situations (again limited by the correctness and coverage of the test cases).

In practice, the ECRobot interface proved to be a very complex system of mutually dependent header- and source code-files, and therefore very difficult to mock directly without making the build procedures for each test suite hugely complex. Instead, we created `nxt_interface`, which works as a layer between the ECRobot interface and our own system modules. All functions in the ECRobot interface used by our system, are mirrored in `nxt_interface`. When a function in nxt_interface is called, the production

implementation, `nxt_interface.c`, calls the corresponding ECRobot function. But in the test suites, the header file `nxt_interface.h` is instead linked to a `mock_nxt_interface.c`, that allows the test suite to read with what parameters the CUT called a certain function, or control what value should be returned to the CUT.

CHAPTER 5

Core Structures and Functions

This chapter describes the core structures and functions of our implementation. These structures and functions are used widely throughout the system.

5.1 Data Types

The nxtOSEK implementation provides some user-defined types, which we adopted. They are shown in Table T5-1. The letters `U` and `S` are short for unsigned and signed, respectively. The numbers correspond to the size in bits. The purpose of having these types is presumably to force the developers to reflect upon which type would be the most efficient for that context.

Original type	Defined type
`unsigned char`	`U8`
`signed char`	`S8`
`unsigned short`	`U16`
`signed short`	`S16`
`unsigned long`	`U32`
`signed long`	`S32`

Table T5-1: Definition of the nxtOSEK defined types.

5.2 Cyclic Array for Filtering

Sensors may occasionally return erroneous data, and especially the ultrasonic sensor tends to be very unreliable. In Section 3.5 the sensors are tested, to estimate their reliability.

It is possible to filter the results and only save the useful result, at the cost of requiring a larger result set. Only the ultrasonic sensor is so unreliable that it is necessary to filter the results, and to do this a cyclic array is used. Two different approaches are used; one for filtering the data from the ultrasonic sensor, and another to filter the input to the `runAEB()` function (described in Section 9.3).

To filter the data from the ultrasonic sensor, three readings are filled into the array. The average and standard deviation are calculated, and the result that differs from the average by more than the standard deviation is discarded, and the remaining values are moved to a new array. Finally the lowest value from the new array is saved as the current reading. This

calculation is constantly executed in a dedicated task, and the result is saved in a global variable, so the newest filtered reading is always available to other tasks.
When the data is used in `runAEB()` it is filtered again. The idea is to prevent the car from stopping after an erroneous reading caused by environmental changes, such as small objects passing the sensor, but not actually blocking the vehicle's path. The filtered readings from before are saved in a cyclic array, and every time the function is executed, the entire array is checked, and only if two or more readings are within a critical distance, the AEB system is activated. To prevent the vehicle from waiting too long before it stops, and risk hitting an object, the array is small and will only prevent very brief interferences.

5.3 Speedometer

The speedometer calculates the current velocity of the vehicle, and the velocity of each motor powering the wheels. The velocity of the vehicle is measured by using the rotation sensor, described in Section 3.3.3, and the velocity of the rear wheels is calculated from data retrieved from the motors directly.

5.3.1 Essential Structures

The speedometer is built upon the `VehicleStruct` structure, presented in Listing 5.1. This structure is also referenced several times throughout the implementation. `VehicleStruct` has two instances of the `MotorStruct`, which holds information about each rear wheel.

```
1    typedef struct MotorStruct
2    {
3        U8 id;
4        U8 lockState; // Tells if the wheel is locked
5        float velocity; // applied velocity
6        // total amount of ticks rotated when the wheel
             was last locked by the brake
7        S32 brakeLockedTick;
8    } MotorStruct;
9
10   typedef struct VehicleStruct
11   {
12       float vFront; // real velocity
13       float vBack; // applied average velocity
14       float vFrontRounds;
15       MotorStruct leftMotor;
16       MotorStruct rightMotor;
17   } VehicleStruct;
```

Listing 5.1: Structure containing information about the vehicle.

5.3.2 Calculating the Velocity

After a measurement from the rotation sensor, the raw data result is used to calculate the velocity of the vehicle. Initially, the four possible raw data values (1024, 512, 384 and 768) are translated into ticks (1, 2, 3 and 4). The distance driven is then added to an array, as seen in Listing 5.2.

```
1  void updateSpeedometerData()
2  {
3      S16 rawData = getRotationSensor();
4      S16 currentTick = convertRawDataToTicks(rawData);
5
6      if (currentTick == -1)
7      {
8          currentTick = rotationSensor.lastTick;
9      }
10
11     if((currentTick+2) % 4 == rotationSensor.
           secondToLastTick)
12     {
13         if ((currentTick+1) % 4 == rotationSensor.
               lastTick)
14         {
15             rotationSensor.ticks++;
16             rotationSensor.secondToLastTick =
                   rotationSensor.lastTick;
17         }
18         else if ((currentTick+3) % 4 == rotationSensor.
               lastTick)
19         {
20             rotationSensor.ticks--;
21             rotationSensor.secondToLastTick =
                   rotationSensor.lastTick;
22         }
23     }
24     rotationSensor.lastTick = currentTick;
25
26     addToArrayFront(rotationSensor.ticks * (360 / 16.0));
27     addToArrayBack(-motorGetCount(LEFT_MOTOR), -
           motorGetCount(RIGHT_MOTOR));
28 }
```

Listing 5.2: Converts the raw data to ticks and adds the driven distance to an array.

Lines 6-9: If a measuring error occurs, the last viable measurement is used.

Lines 11: This statement assures the vehicle moves before calculating how much. This is to prevent erroneous readings: When the vehicle is stationary, the rotation sensor may continue alternating between two values, presumably because it has stopped right on the "border" between them. If this is the case, one of these two values would have been saved in `rotationSensor.secondToLastTick`. Therefore, both values will fail this test, and `rotationSensor-.ticks` will not be incremented.

Lines 13-17: If `currentTick` is viable and forward moving, `rotationSensor.ticks` increases, and `rotationSensor.secondToLastTick` is updated with the value of `rotationSensor.lastTick`.

Lines 18-22: If the `currentTick` is viable and backward moving, `rotationSensor.ticks` is decremented and `rotationSensor.secondToLastTick` is updated with the value of `rotationSensor.lastTick`.

Lines 24: `rotationSensor.lastTick` is updated with the value of `currentTick`.

Lines 26-27: Adds the distance driven to arrays used to calculate the velocity.

The velocity is calculated by calculating the difference in distance driven and multiplying it with the circumference of the wheel. For the motor velocity, the velocity is multiplied by the gear ratio to get the real velocity of the rear wheels.

```
1  float calculateVelocityBack(U8 id)
2  {
3      S32 degreeDiff;
4      if(id == LEFT_MOTOR){
5          degreeDiff = tickCounter.backDegreesArrayLeft[
               tickCounter.newIndexBack]
6                               - tickCounter.
                                    backDegreesArrayLeft[
                                    tickCounter.oldIndexBack];
7      } else
8      {
9          degreeDiff = tickCounter.backDegreesArrayRight[
               tickCounter.newIndexBack]
10                              - tickCounter.
                                    backDegreesArrayRight[
                                    tickCounter.oldIndexBack];
11     }
12     float degreePerMs = degreeDiff / 250.0;
13     float roundsPerSec = (degreePerMs * 1000) / 360.0;
14     float metersPerSecond = roundsPerSec * WHEEL_CIRC *
           GEAR_RATIO;
15
16     return metersPerSecond;
17 }
18
19 void calculateVelocity()
20 {
21     /* BackWheel speedometer */
22     vehicle.rightMotor.velocity = calculateVelocityBack(
           vehicle.rightMotor.id);
23     vehicle.leftMotor.velocity = calculateVelocityBack(
           vehicle.leftMotor.id);
24
25     vehicle.vBack = (vehicle.leftMotor.velocity + vehicle
           .rightMotor.velocity) / 2;
26
27     /* FrontWheel speedometer */
28     float degreeDiff = tickCounter.frontDegreesArray[
           tickCounter.newIndexFront]
29                         - tickCounter.frontDegreesArray[
                               tickCounter.oldIndexFront];
30     float degreePerMs = degreeDiff / 250.0;
31     float roundsPerSec = (degreePerMs * 1000) / 360.0;
32     vehicle.vFront = roundsPerSec * WHEEL_CIRC;
33 }
```

Listing 5.3: Calculates the velocity of the vehicle (front wheel) and the velocity of the motors (rear wheel).

The function `calculateVelocityBack()` calculates the velocity of a rear wheel. This function has to be called twice, once for each motor.

Lines 4-11: The amount of ticks driven since last execution is calculated, depending on which motor id has been passed as parameter.

Lines 12-16: The ticks are converted to meters per second and returned to the caller.

Lines 22-23: The velocity for the rear wheels is stored.

Lines 25: The average velocity of the rear wheels is stored in `vehicle.vBack`.

Lines 28-32: The same procedure as above however with the data for the front wheel, which is stored in `vehicle.vFront`.

5.4 Braking Distance

The theoretical braking distance can be calculated with knowledge of the velocity of the vehicle, v, and the coefficient of friction, μ. The calculation of braking distance d is shown in Equation (5.1), where g is the gravitational acceleration.

$$d = \frac{v^2}{2 \cdot \mu \cdot g} \tag{5.1}$$

The coefficient of friction is found by placing the vehicle with its brakes engaged on a surface similar to the one it will be driving on, and tilting the surface. When the vehicle starts sliding, the amount of degrees between the ground and the surface is measured. This is translated to μ as shown in Equation (5.2) where α is the measured inclination in degrees.

$$\mu = \tan(\alpha) \tag{5.2}$$

This test was performed, and it was found that the vehicle would slide at 9 degrees, yielding:

$$\mu = \tan(9) \approx 0.158.$$

The value used for the gravitational acceleration is the standard gravity on earth, $9.80665\ \mathrm{m/s^2}$ [19].

5.5 Turning a Set Amount of Degrees

`turnDegreesLaneControl()` is used in lane control to turn the front wheels to a given angle. Driver-control of the steering should be disabled before calling `turnDegreesLaneControl()`. This is done by setting `turnDegreesActivated` in the `steer` structure to 1. Driver-control can be reinstated by setting it to 0. When the angle is reached, the wheels are locked in that position, until a new angle is specified, or driver-control of the vehicle is re-enabled.

```
1 void turnDegreesLaneControl(S32 degrees)
2 {
3     float deg = degrees;
```

```
4
5       if(deg > MAX_TURNING_ANGLE)
6       {
7           deg = MAX_TURNING_ANGLE;
8       }
9       else if(deg < -MAX_TURNING_ANGLE)
10      {
11          deg = -MAX_TURNING_ANGLE;
12      }
13
14      steer.turnDegrees = (S32)(deg * (100 /
            MAX_TURNING_ANGLE));
15  }
```

Listing 5.4: Implementation of `turnDegreesLaneControl()`.

Line 5-8: It is checked whether the applied turning angle is greater than `MAX_TURNING_ANGLE`. If so, the turning angle is set to `MAX_TURNING_ANGLE`.

Line 9-12: Here it is checked whether the applied turning angle is less than `-MAX_TURNING_ANGLE`. If so, the turning angle is set to `-MAX_TURNING_ANGLE`.

Line 14: Turning `MAX_TURNING_ANGLE` is equivalent to 100%, and `-MAX_TURNING_ANGLE` is equivalent to -100%. This formula calculates how many percent the wheels are turned, and saves it in `turnDegrees` in the `steer` structure.

CHAPTER 6

Anti-lock Braking System

According to [18], a problem in cars built before the introduction of anti-lock braking systems (ABS) was the risk of wheels "locking up" during hard braking, for example in case of an emergency. Brakes are powerful enough to completely stop one or more wheels from rotating, but the car may still have enough momentum to continue moving forward, and will therefore continue skidding. Skidding results in worsened braking lengths and possible loss of vehicle control. The risks are even greater on surfaces with bad traction, such as wet or icy roads.

ABS is made to prevent this problem by adjusting the brake pressure on the wheels, providing the maximum possible braking power, without locking the wheels. This provides the vehicle a much improved braking distance, and reduces the risk of losing control, also on surfaces with decreased traction.

6.1 Operation

The operation of ABS may vary slightly, depending on the system setup. The following is therefore based on Nice's explanation [16]. Common for all setups are the components: Speed sensors for determining the speed of individual or sets of wheels, valves on the brakes to reduce brake pressure, a pump for increasing brake pressure, and a microcontroller that connects and manages all the components. Setups vary in the number of sensors and valves.

If the driver of the vehicle applies the brakes too hard for the conditions, one or more wheels may start to lock. When this happens, the controller, through the speed sensors, detects an unusually high rate of deceleration on the affected wheels. To determine whether the deceleration is unusual, the speed of the wheel is compared to the reference speed. The reference speed is an approximation of the vehicle's actual speed, calculated from previous readings of the speed of the individual wheels. In case of locking, the wheels will decelerate much faster than the vehicle would ever be able to, due to its momentum. The controller then opens the valves to prevent further pressure from being applied to the brake. When the wheel again starts to accelerate, the controller applies pressure to the brake until the wheel decelerates again. These actions are performed several times each second, and the brakes are therefore constantly held at pressure just below where the wheels lock up, thus achieving the most effective braking. The cycle continues until the risk of locking is gone, either when the car has decelerated enough or the driver has stopped applying the brake.

6.2 Theory

This section explains the theory behind ABS, and also describes the calculations necessary for ABS to work.

To calculate if a wheel is locking up, the speed of the vehicle and the speed of the wheel are needed. The ratio between the speed of the vehicle and the speed of the wheel is called the slip ratio.

The formula for calculating the slip ratio is, as described by WABCO [25]:

$$\lambda = \frac{\mathbf{V_{vehicle}} - \mathbf{V_{wheel}}}{\mathbf{V_{vehicle}}} \cdot \mathbf{100}\,\% \tag{6.1}$$

λ is the slip ratio, $V_{vehicle}$ is true speed of the vehicle itself, and V_{wheel} is the speed of the wheels. $V_{vehicle}$ is estimated from the speed of all four wheels, as there is no practical mechanism to measure vehicle speed accurately except for the sensors on the wheels, which of course do not take slip ratio into account.

If $\lambda = 0$, the speed of the wheel is equal to the speed of the vehicle, and if $\lambda = 100$, the wheel is blocked. For optimal braking, the slip ratio has to be between 10 % and 30 %. This is also important when turning, as a completely locked wheel has almost no cornering force. Therefore ABS controls the brake force, to keep the slip ratio between 10 % and 30 %, and to keep the vehicle steerable.

When the driver brakes the vehicle, ABS monitors the wheel speed, and when the slip ratio exceeds 30 % it quickly and shortly reduces the brake pressure. This reduces the slip ratio, and brake pressure is applied again. This cycle is performed until the driver removes the brake pressure.

6.3 Design

We considered a few different designs for our implementation of ABS, first of all a traditional ABS (as described in Chapter 6), but, as is described in the following section, this proved not to be a viable option, and we instead considered how we could achieve an effect similar to ABS on our vehicle.

Traditional ABS

The concept for anti-lock braking systems in traditional vehicles is described in Chapter 6.

Implementing a traditional ABS on our test platform entails some difficulties. As our system transfers power directly from the servomotors to the back wheels, with no dynamic transmission, taking power from the motors will result in an instantaneous and equivalent loss of power to the wheels. The servo motors has the option to either block or float, when the applied power is 0 %. If one chooses to block, the vehicle will almost certainly skid (depending on speed, friction of the surface and gradient), as its momentum forces it to continue, despite the locked back wheels. If one turns off the motors at the vehicle's maximum speed, with floating mode selected, the vehicle will simply roll to a halt.

Furthermore, our test platform has no traditional analogue brakes, as these are mainly a mechanical construction, and difficult to implement in LEGO. It would also require at least one more output port from the NXT for the actuator. As we do not have an unused output port, we would need to install an additional NXT on the vehicle. We would then have to allow the two NXTs to communicate, in order to share data they both need, and consider the implications of scheduling the two separate systems. The increased complexity in both construction and software is clear.

ABS with Servomotors

As outlined in the section above another approach of implementing ABS needs to be designed. It is not possible to design the brakes with an increasing braking power and it is not possible to brake on all four wheels. Therefore our ABS design is somewhat simpler and consists of two cases.

1. The wheel is locked, release the brake by making it float.
2. The wheel is spinning, lock the wheel by activating the brake.

To test whether the wheel is locked, the slip ratio is calculated for each wheel. Instead of estimating the vehicle velocity, we can use the real velocity, calculated using the rotation sensor.

6.4 Control Flow

This section presents the control flow of the ABS implementation, shown in Figure F6-1. This flow is traversed continually as long as the brakes are applied.

Initially the current slip ratio is calculated. If it is above 20 %, the brake on the wheel is released, as this means that the wheel is completely or nearly locked. The wheel's lock state is also set to 1. If the slip ratio is below 10 % the braking "pressure" is increased and lock state set to 0. If the slip ratio is $10 - 20$ %, the brakes are handled as if the slip ratio is below 10 or above 20 % by looking at the lock state.

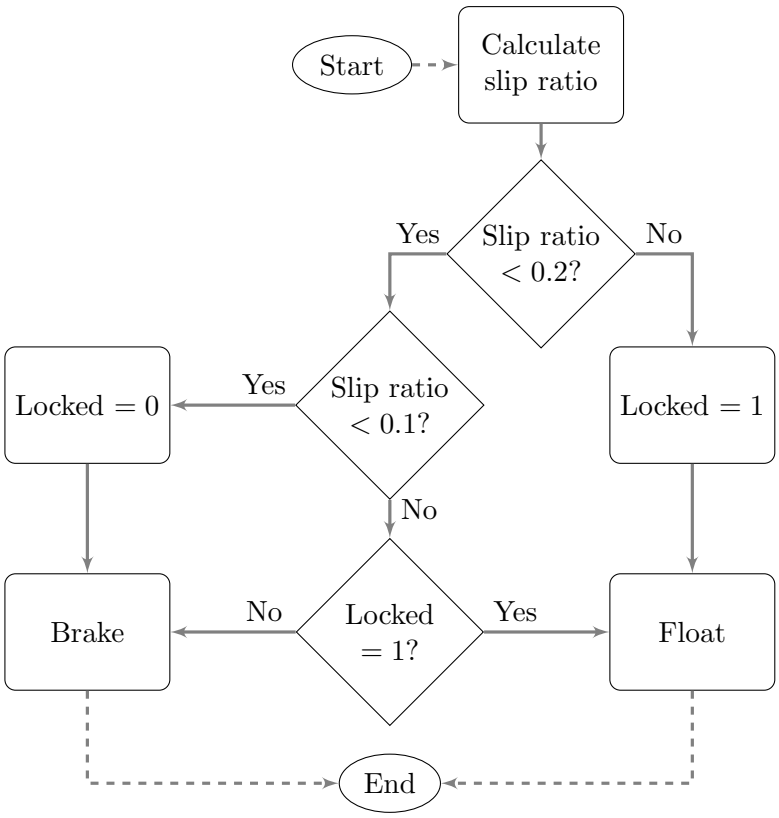


Figure F6-1: Illustration of the control flow in ABS.

6.5 Implementation

The implementation of ABS is based on the theory described in Section 6.2, and some implementation inspiration is retrieved from the student report by SW502E12 [24]. Although the theory states that the slip ratio should be between 10-30%, we found that with this interval the vehicle would start spinning out of control. Therefore we tried narrowing the interval, and found that with 10-20% we achieved a straight braking path.

When the driver or an assist on the vehicle wants to brake the vehicle, the function `brake()` is called. This function is responsible for adjusting the braking power on each rear wheel according to the requirements of ABS.

To adjust the braking power `brakeMotor()` is called once for each motor. By adjusting the braking power for each motor we achieve an ABS-like effect, meaning that the braking is controlled and the wheels do not lock up.

For `brakeMotor()` to adjust the power of the motor, it needs to know if that motor is locked due to braking. This is achieved by calling `updateLockState()`, presented in Listing 6.1.

```
1 void updateLockState(MotorStruct* motor) {
2     if (fabs(vehicle.vFront) > 0.0) {
3         float slipRatio = (fabs(vehicle.vFront) - fabs(
              motor->velocity)) / fabs(vehicle.vFront);
4         if (fabs(slipRatio) > 0.2){
5             motor->lockState = 1;
6         }
7         else if (fabs(slipRatio) < 0.1)
8         {
```

```
9                 motor->lockState = 0;
10            }
11        }
12    }
```

Listing 6.1: This function determines whether a motor is locked due to braking.

Line 2 Ensures that the function is only computed when the vehicle is moving. This avoids a devision by zero in line 3.

Line 3: Calculate the slip ratio.

Line 4-10: Determine whether the motor is locked and update accordingly.

6.6 Evaluation

For evaluating the ABS, several tests were conducted. The ABS is tested by accelerating to 1.9 m/s after which it would brake. We expect the braking distance to be shorter, the wheels not to lock and the braking to be more controlled. The optimal results would be a graph where the motor's velocities decrease to a certain point, after which it increases very little, and then starts decreasing again. The actual results are shown in Figure F6-2, Figure F6-3 and Figure F6-4.

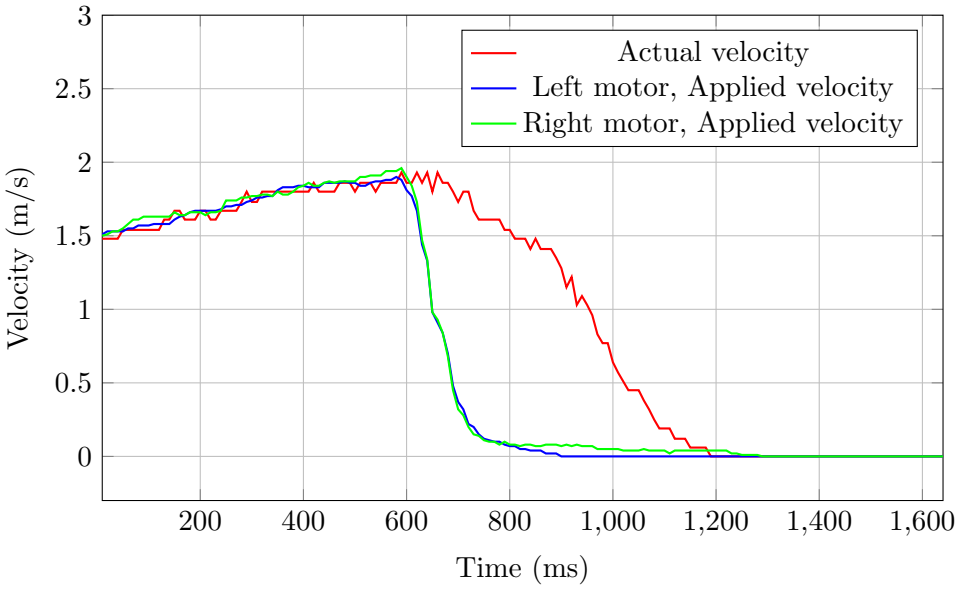


Figure F6-2: Graph of velocities with ABS disabled.

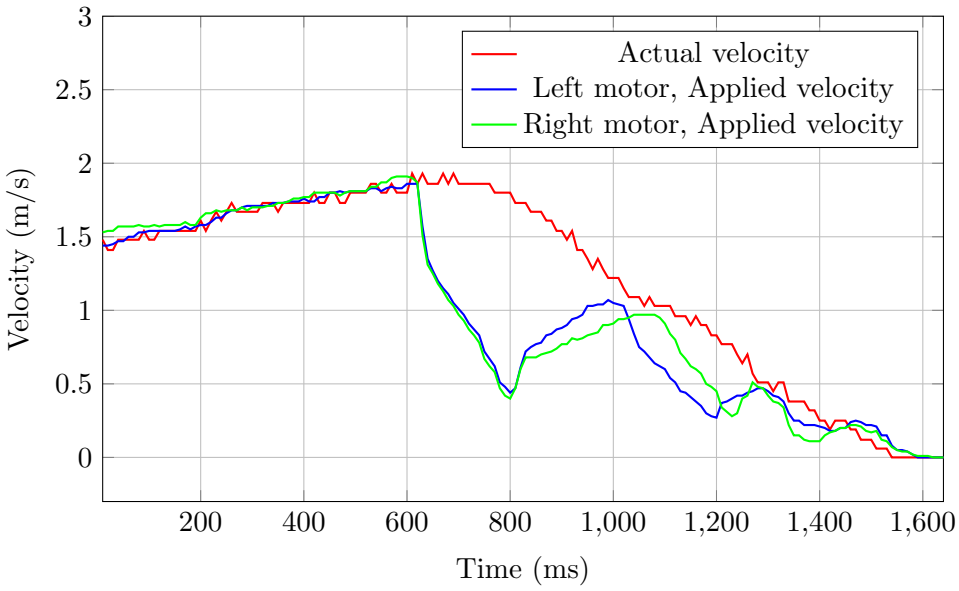


Figure F6-3: Graph of velocities with ABS enabled.

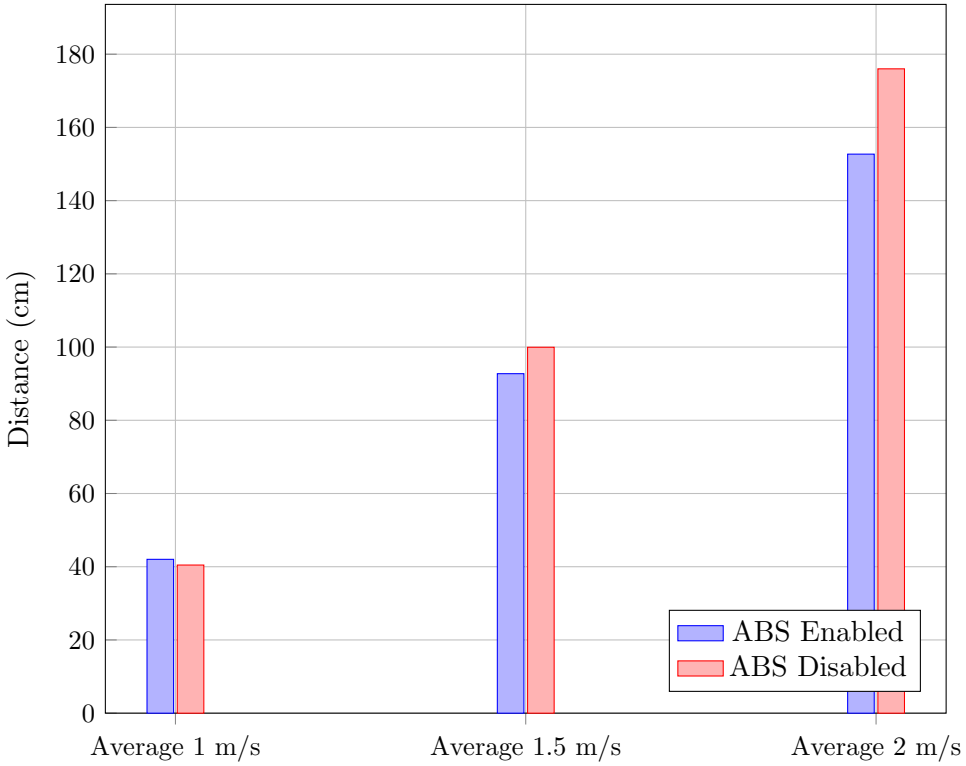


Figure F6-4: Graph of brake distances.

Based on the data shown in Figure F6-2 and Figure F6-3 it can be concluded that the brakes do no longer lock, which leads to a much more controlled braking. The brake distance seen in Figure F6-4 has been shortened, as expected. Tests where the vehicle braked ten times without ABS, and ten times with ABS, showed that the brake distance had shortened from 176 cm with ABS deactivated to 152.7 cm with ABS activated when driving 2 m/s. As illustrated in Figure F6-4 the difference in brake distance increases as velocity increases. This development suggests the ABS increases safety when braking at speeds above 1.5 m/s, as it not only ensures a controlled braking

but also restricts the brake distance. Even at velocities under 1.5 m/s the vehicle was easier to control while braking with ABS enabled. In the light of this, we consider the requirements and expectations to have been met. As Figure F6-3 shows, the system only unlocks the wheels 2-3 times before the vehicle has fully stopped. This may be caused by a slow response from the motors, or because the feature monitoring the wheels when the vehicle is braking, is not executed often enough. The latter problem can likely be solved by giving this feature a dedicated task, thus allowing it to run more frequently. Doing this would likely result in a lower braking distance. Other options to achieve this could be to use more responsive motors, or attaching brakes to the front wheels.

CHAPTER 7

Traction Control System

This section is based on the description by the Editors of Publications International, Ltd. [7].

The idea of a traction control system (TCS) is to minimize the risk of a vehicle losing traction on one or more wheels during acceleration. The problem typically arises when accelerating a stationary vehicle, or on roads with reduced traction, for example wet, icy or gravel roads.

If the driver depresses the throttle too quickly, the wheel is unable to increase the vehicle's acceleration, due to the weight of the vehicle. Instead, the wheel loses grip on the road and starts spinning without moving the vehicle. The risk of this increases as the traction of the road decreases.

TCS prevents this by applying brakes to the affected wheels, or decreasing the throttle.

7.1 Operation

The operational principles of TCS are quite similar to ABS, described in Chapter 6, and they are often implemented as a combined system. When a wheel loses traction, the speed sensor registers an unrealistic increase in acceleration on that wheel. The controller reacts to this by applying the brake to the affected wheel, until the wheel's speed matches the speed of the other wheels. Some traction control systems may decrease the throttle, if multiple wheels start to spin.

7.2 Theory

As with ABS, achieving an optimal slip ratio is key to the functionality of TCS. Slip ratio is described in Section 6.2. To determine the slip ratio for TCS, one uses the formula shown in Equation (7.1).

$$\lambda = \frac{\mathbf{V_{wheel}} - \mathbf{V_{vehicle}}}{\mathbf{V_{wheel}}} \cdot \mathbf{100}\,\% \tag{7.1}$$

The optimal slip ratio, called the target slip ratio, depends on the type of wheels and the character of the surface the vehicle drives on.

7.3 Implementation

The idea behind our TCS implementation, is to calculate how much power to apply the rear wheels individually.

This means that the driver is applying an amount of power, which is then adjusted for each rear wheel by TCS.

If the rear wheel is rotating more than 20% faster than the front wheels, the rear wheel's applied power is reduced. The power is reduced to match the front wheel's rotation speed and then increased by 20% to allow the vehicle to accelerate faster, since we found the vehicle accelerates very slowly without this increase.

7.4 Evaluation

For the evaluation of TCS, a test was devised. The vehicle will accelerate with full power until it reaches the maximum velocity. We expect the measurements to show an applied velocity closer to the actual velocity. The optimal result would be the applied velocity being almost equal to the actual result. The data from the test is presented in Figure F7-1 and Figure F7-2

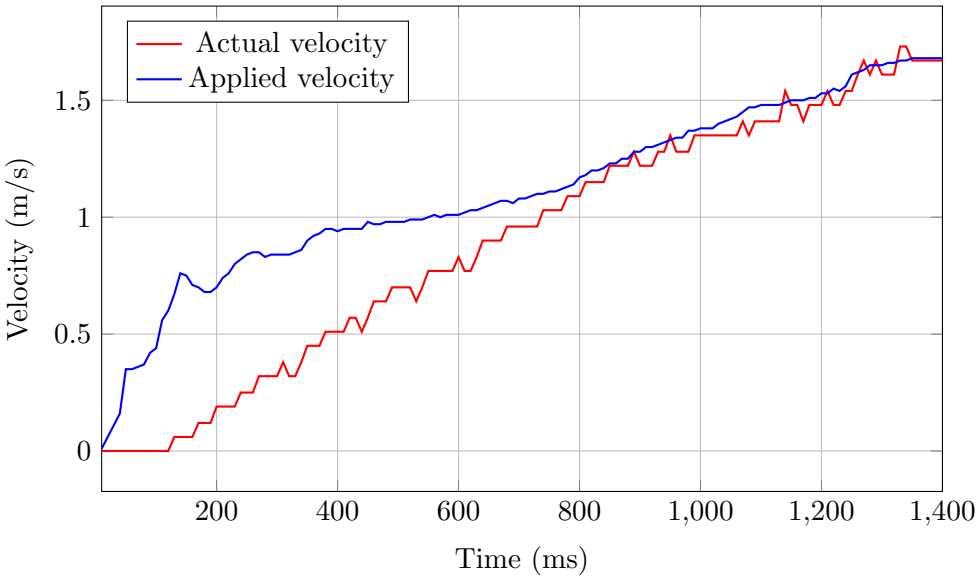


Figure F7-1: Graph for evaluation with TCS disabled.

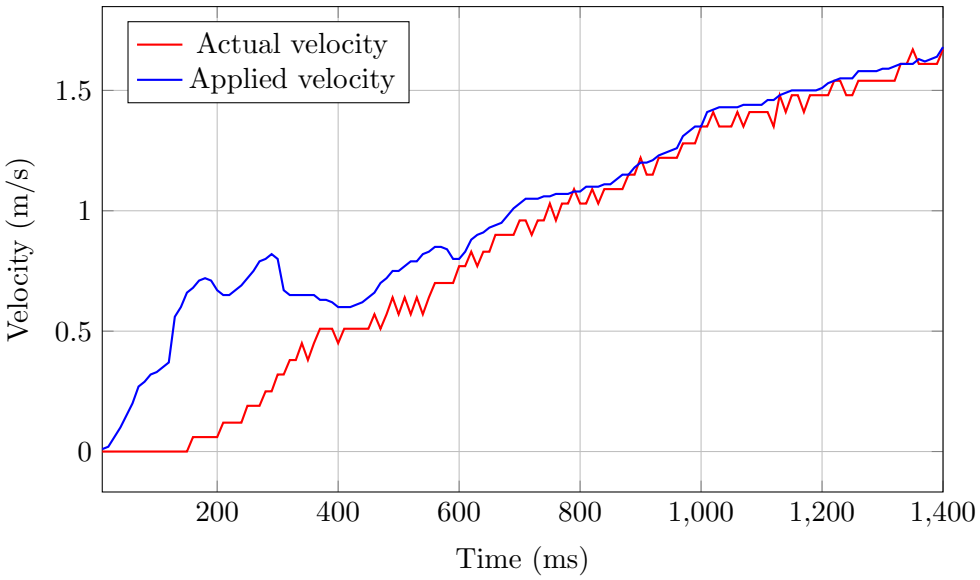


Figure F7-2: Graph for evaluation with TCS enabled.

The result illustrates how the traction control shortens the amount of time it takes for the applied velocity to match the actual velocity. Figure F7-1 shows how it takes about 0.8 m/s to match the velocities with TCS disabled, while Figure F7-2 shows that this is about 0.7 m/s with TCS enabled. It can therefore be concluded that the TCS meets the requirements. For future use of TCS, it is recommended to explore different target slip ratios. This in turn, could result in an even faster match between the applied velocity and the actual velocity.

CHAPTER 8

Electronic Differential Control

A differential is a mechanism that makes it possible to have different rotation speed and torque distribution on the powered wheels. This can be achieved using a mechanical differential or an electronic differential (EDC). This section describes electronic differential.

On regular vehicles, the differential is necessary when cornering. This is due to the vehicle only having one engine delivering the same speed to all powered wheels. Without the differential, the inner and outer wheels would rotate with the same speed, either making the inner wheel slip, or the outer wheel drag. By using a differential, the wheels can rotate with different speeds, making turning possible without loss of traction.

8.1 Operation

The vehicle used in this project powers the wheels with two electric motors, one for each wheel. Therefore the differential has two tasks.

One is to synchronise the speed of the wheels when driving in a straight line. This is to avoid unintended turning of the vehicle.

The other task is to reduce the applied power to the inner wheel when turning. Because the turn radius of the inner wheel is smaller than the turn radius of the outer, the inner wheel has to turn slower than the outer wheel. To calculate the speed of the inner wheel, the equations in the following section were derived.

8.2 Theory

This section describes the theory behind EDC. In this section two special measurements are used, namely wheelbase and track, Figure F8-1 illustrates these measurements.

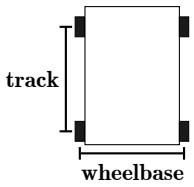


Figure F8-1: This illustration presents the measurements wheelbase and track.

For EDC, three equations are used. These are described below.

Diameter of the turning circle

This formula approximates the turning circle diameter, D, based on the wheelbase length, $L_{\text{wheelbase}}$, and steering angle, α.

$$D = \frac{2 \cdot L_{\text{wheelbase}}}{\sin(\alpha)} \tag{8.1}$$

Ratio between inner and outer turning circle

The EDC ratio, R, is the percentage of which the inner motor speed has to be reduced. R is found using the track, L_{track}, and the turning circle diameter.

$$R = 1 - \frac{2 \cdot L_{\text{track}}}{D} \tag{8.2}$$

Inner motor speed

The resulting applied power of the inner motor, $P_{\text{resulting}}$, is calculated using the current power to be applied to the motor, P_{current}, and multiplying it with the ratio, which represents how much the speed of the inner motor has to be reduced. It is multiplied by 1.5 to make it better in practice for our vehicle.

$$P_{\text{resulting}} = P_{\text{current}} \cdot R \cdot 1.5 \tag{8.3}$$

8.3 Implementation

The implementation of EDC is straight forward. In addition to computing Equations (8.1) to (8.3), we need to check if the vehicle is turning and in which direction.

As Equation (8.3) states, the result is multiplied by 1.5, since we found that it slowed the vehicle down to much, which made the vehicle spin around. We suspect the low traction between the wheels and the floor to be the reason. Another special case is that if the resulting power gets below 65 %. The motors are not able to rotate with a power less than 65 % as a cause of the friction and the gearing. Therefore we need to take this into account to prevent the vehicle from spinning. So if the resulting power is less than 65 %, we set it to 65 %.

8.4 Evaluation

The evaluation of the EDC is based on a test we devised. In the test the vehicle is set to drive 3 m/s. We then turn the wheels gradually, to ensure

the inner wheel does not just brake, but instead slows steadily. We expect to see an decrease in velocity on the right wheel, but unchanged left wheel. The optimal result would be a step-like graph. The data from the test is presented in Figure F8-2 and Figure F8-3.

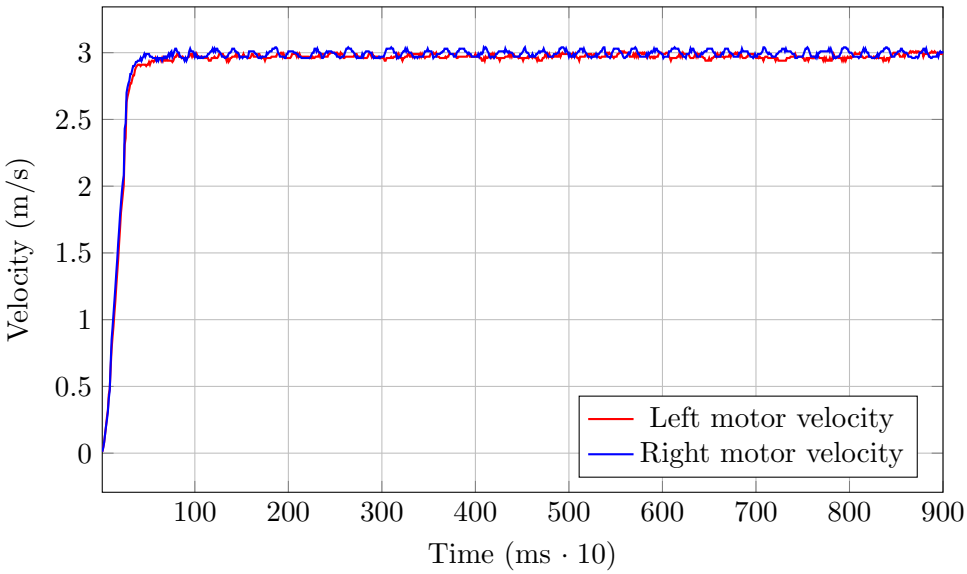


Figure F8-2: Graph for evaluation with EDC disabled.

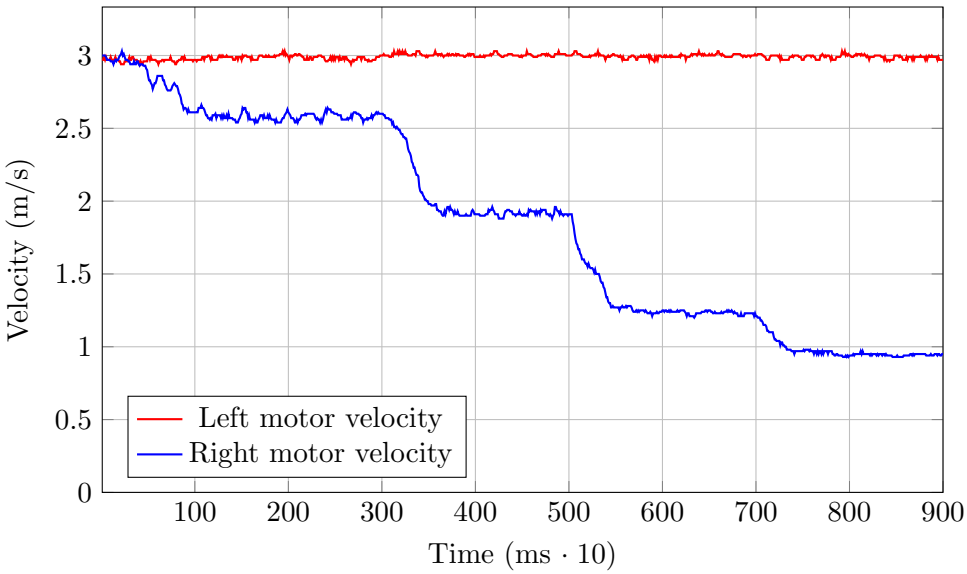


Figure F8-3: Graph for evaluation with EDC enabled.

Figure F8-3 shows that when the vehicle turns right with EDC enabled, the velocity of the inner (right) wheel has decreased from 3 m/s to 1 m/s. This drop in velocity is caused by the lowered power to the motor which is the purpose of the feature. As a result of this the control of the vehicle has been greatly improved. It is now possible to turn and even drive in circles without drifting or skidding. It can therefore be concluded that the EDC meets the requirements. We do not believe it is necessary to improve EDC further in the future.

CHAPTER 9

Autonomous Emergency Braking

The main idea of autonomous emergency braking (AEB), is to prevent or mitigate accidents caused by the driver not reacting quickly enough in a dangerous situation. Slow driver reaction time is typically caused by low visibility or the driver being inattentive. As explained by EURO NCAP [15], AEB systems are built to detect these situations and warn the driver, possibly slowing down the vehicle autonomously if the driver does not react quickly enough. The following description of the operation of AEB is based on the explanation by EURO NCAP.

9.1 Operation

AEB is more of a concept than a technology and implementations vary widely between manufacturers. In most implementations, a range measuring technology (for example radar or lidar) is mounted in the front end of the vehicle and connected to a computer, which also has information about the vehicle's speed and direction.

If the system detects an object with which there is a risk of collision, it attempts to warn the driver. If the driver does not react appropriately to prevent the collision, the system applies brake pressure, either fully or partially. This, however, varies between implementations. Sometimes the system may also ready the vehicle for impact, for example by tightening the seatbelts.

9.2 Theory

To determine the distance from the vehicle to objects that are in its way, an ultrasonic sensor is used. How the sensor works is explained in more detail in Section 3.3. The sensors will detect objects in a cone shape in front of the vehicle, and as a consequence it might detect objects that are not in the path of the vehicle. Figure F9-1 illustrates an example of this. The red path is the path the vehicle is following, the blue square is an object that can be either in or outside the path.

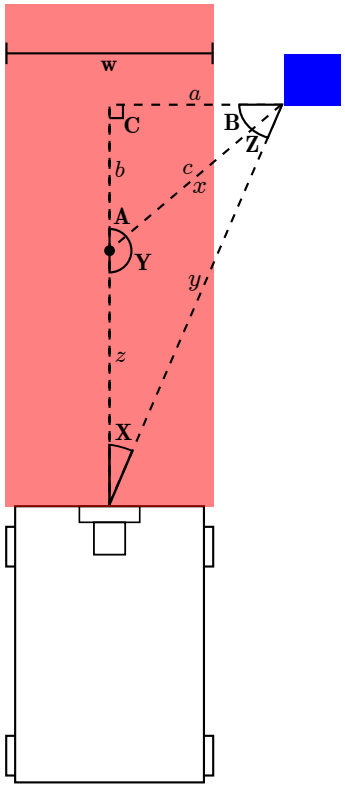


Figure F9-1: Illustration of the vehicle's path shown in red and a blue object.

The right-angled triangle drawn between the vehicle and the object illustrates the information we need to obtain, to determine whether or not the vehicle will hit the object. The hypotenuse, labelled y, is the shortest distance between the sensor and the object. The two legs are drawn so one is following the vehicle's direction, split in two parts labelled z and b, and the other, labelled a, is the distance from the middle of the path to the object. In Section 9.3 the main implementation of AEB is described.

In this section it is described how to calculate the length of a and b. When a and b are known, it can be determined if the object is in the vehicle's path, as well as the distance before impact.

The formulas are based on the fact that only one ultrasonic sensor is available, meaning that it is not possible to determine whether the object is located to the left or the right of the vehicle's path. As a consequence we make the assumption that all objects are on the right side.

9.2.1 Derivation of Formula

As mentioned, we want to find the length of sides a and b in the triangle in Figure F9-1. First it will be shown how to find a and then use this result to find b. To do this, the relation stated in 9.1 is used.

$$a = c \cdot \sin(A) \tag{9.1}$$

The length of c is known to be the same as x (this side has two names to seperate the two smaller triangles). It is now needed to figure out the angle A, which is done by subtracting Y from 180°.

To find Y, the following relation is used:

$$\cos(Y) = \frac{x^2 + z^2 - y^2}{2 \cdot x \cdot z}$$
$$\Updownarrow$$
$$Y = \cos^{-1}\left(\frac{x^2 + z^2 - y^2}{2 \cdot x \cdot z}\right) \tag{9.2}$$

This relation makes it possible to calculate A, as shown in Equation (9.3).

$$A = 180 - Y$$
$$\Updownarrow$$
$$A = 180 - \cos^{-1}\left(\frac{x^2 + z^2 - y^2}{2 \cdot x \cdot z}\right) \tag{9.3}$$

By using the result from Equation (9.3) in Equation (9.1), the formula to calculate side a is obtained, as shown in eq. (9.4).

$$a = c \cdot \sin\left(180 - \left(\cos^{-1}\left(\frac{x^2 + z^2 - y^2}{2 \cdot x \cdot z}\right)\right)\right) \tag{9.4}$$

With this, it can be determined whether the object is in the path of the vehicle. If this is the case it is necessary to know the distance between the vehicle and the object (distance to impact). This corresponds to the length of side b, which can be calculated using the Pythagorean theorem, as shown in Equation (9.5):

$$b = \sqrt{c^2 - a^2} \tag{9.5}$$

9.3 Implementation

To implement AEB, we need knowledge of the distance to the object in the path of the vehicle, explained in Section 9.2, and the vehicle's braking distance, explained in Section 5.4. To simplify the calculations, it is assumed the object blocking the vehicle's path is stationary and, as mentioned before, on the right side of the path.

The distance from the path to an object is calculated in the function `updateDistanceFromPath()` shown in Listing 9.1. The function uses Equation (9.2) and Equation (9.4) to calculate the distance from the path to an object. For the formula to work, two measurements are needed at different points in time. Therefore only one measurement is made for each execution of the function. This measurement is used in conjunction with the measurement from the previous execution of `updateDistanceFromPath()`.

```
1 int updateDistanceFromPath() {
2     U32 currentTime = getSysTime();
3
4     int result = -1;
5
6     if (aeb.lastTimeReadPath == 0)
7     {
```

```
8           aeb.distanceToObj1 = vehicle.sonarReading;
9           aeb.lastTimeReadPath = currentTime;
10          result = -1;
11      }
12      else if (currentTime - aeb.lastTimeReadPath >=
            AEB_TIME_BETWEEN_READINGS)
13      {
14          aeb.distanceToObj2 = vehicle.sonarReading;
15
16          if (aeb.distanceToObj2 < 255)
17          {
18              float distanceTravelled =
                    getDistanceTravelled();
19
20              float angleToObj2 = acos((pow(aeb.
                    distanceToObj2,2) + pow(distanceTravelled
                    ,2) - pow(aeb.distanceToObj1,2)) / (2 *
                    aeb.distanceToObj2 * distanceTravelled));
21              aeb.distanceFromPath = aeb.distanceToObj2 *
                    sin(PI - angleToObj2);
22
23              aeb.distanceToObj1 = aeb.distanceToObj2;
24              aeb.lastTimeReadPath = currentTime;
25
26              if(aeb.distanceFromPath <= VEHICLE_WIDTH)
27              {
28                  result = 1;
29              }
30          }
31      }
32      return result;
33  }
```

Listing 9.1: Function that determines whether an object is within the path of the vehicle.

Line 6-11: `aeb.lastTimeReadPath` is initialised to 0 when the program starts, so the distance from the sensor to the object and the current system time is saved. `result` is set to -1, as there is no useful result yet.

Line 12-31: If the time between two readings, `aeb.lastTimeRead` and `currentTime` is larger or equal to `AEB_TIME_BETWEEN_READINGS` (25ms) the distance to the object is calculated.

Line 14: The newest measurement from the ultrasonic sensor is retrieved and stored in `aeb.distanceToObj2`.

Line 18: The distance travelled since the last reading is calculated, and stored in `distanceTravelled`.

Line 20-21: First the `angleY` is calculated by using Equation (9.2). Then `aeb.distanceFromPath` is calculated according to Equation (9.4).

Line 23-24: The newest measurements are now stored as old measurements.

Line 26-29: If the object is in the path of the vehicle, `result` gets the value of 1.

Line 32: Returns whether or not an object is in the path of the vehicle. If the object is in the path of the vehicle, the return value is 1, otherwise it is -1.

The distance to an object on the path is calculated in `getDistanceToImpact()`. When executed, the function calls `updateDistanceFromPath()` to determine if the detected object is within the path or not. If the object is within the path, the function calculates the distance to the object by using Equation (9.5). The calculation of the distance can be imprecise due to the low precision of the ultrasonic sensor.

```
1  float getDistanceToImpact(){
2      float result;
3
4      if (updateDistanceFromPath() > -1)
5      {
6          result = sqrt(pow(aeb.distanceToObj2,2) - pow(aeb
               .distanceFromPath,2));
7      }
8      else
9      {
10         result = -1;
11     }
12     return result;
13 }
```

Listing 9.2: Function that determines the distance to an object within the path of the vehicle.

Activating the AEB is controlled by the function `runAEB()`. To filter the results and prevent AEB from activating because of a bad reading, a cyclic array with three elements is used, as explained in Section 5.2:

```
1  void runAEB()
2  {
3      if (vehicle.vFront < 0.01 || AEB_flag == OFF)
4      {
5          aeb.activated = 0;
6          return;
7      }
8
9      int i;
10     U8 criticalDistance = 0;
11
12     float brakeDistance = getBrakeDistance() +
           SONAR_ERROR_MARGIN;
13
14     for (i = 0; i < SAMPLE_COUNT; i++)
15     {
16         if ((aeb.distances[i] <= (brakeDistance +
               BRAKE_SAFE_ZONE)) && aeb.distances[i] > -1)
17         {
18             criticalDistance++;
19         }
```

```
20
21      }
22
23      if (criticalDistance >= 2 || aeb.eaebState !=
            STATE_NOT_ACTIVATED)
24      {
25          aeb.activated = 1;
26          if(EAEB_flag == ON)
27          {
28              runEAEB();
29          }
30      }
31      else
32      {
33          aeb.activated = 0;
34      }
35  }
```

Listing 9.3: Function that determines if the AEB should be activated.

Line 3-7: If the vehicle is not moving, or the AEB system is disabled, AEB is deactivated, and the function ends.

Line 14-21: The measurements in the cyclic array are evaluated, and for every measurement less than the allowed distance, `criticalDistance` is incremented.

Line 23-30: If enough measurements were critical, AEB is activated, and if EAEB is enabled, the EAEB algorithm is called. This path is also entered if EAEB is not in its default state, as it may need to update its state.

When the `aeb.activated` flag is set, the controller task disables the throttle and enables the brake.

9.4 Evaluation

To verify the functionality of the implementation, two tests were devised. The first test is a maximum test, which will identify the highest velocity at which the vehicle can drive towards an object, and still be able to use AEB as intended. The vehicle will be driven at increasing velocities towards an object, and monitor if and how it brakes. We expect the AEB to be functional at the vehicle's top speed. The optimal result will be a horizontal graph which illustrates that the vehicle halts at the same distance, regardless of the velocity. The results are shown in Figure F9-2.

We find no reason to test the vehicle with AEB disabled as it would just drive straight into the object ahead.

The purpose of the second test is to verify the ability to drive past an object not in the path of the vehicle without braking, i.e. if it calculates the distance from the path correctly. The vehicle will be driven towards an object, and it is noted whether AEB activates or not. The corner of the object is placed relative to the middle of the vehicle, where the ultrasonic sensor is placed. A negative distance places the vehicle further in front of the

object, and a positive distance places the vehicle further away. The distance from the centre to the edge of the vehicle is 12 cm. A few centimetres is added to this amount, and the vehicle is expected to activate AEB when the calculated distance between the object and the path of the vehicle, is under 25 cm.

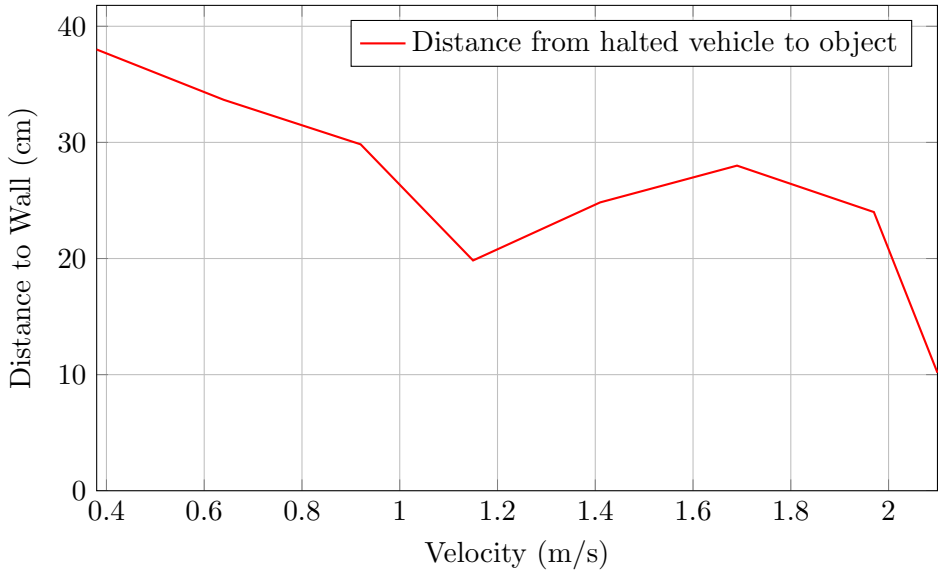


Figure F9-2: Graph for the vehicle driving straight for an object with AEB enabled.

AEB status determined by distance to object					
Distance	-5 cm	0 cm	5 cm	10 cm	15 cm
Status	Activated	Activated	Activated	Activated	Activated
Distance	17 cm	18 cm	19 cm	20 cm	
Status	Activated	Activated	Activated	Not activated	

Table T9-1: The results of the second test.

The result presented in Figure F9-2 shows how the AEB is able to make the vehicle come to a complete halt, before hitting the object, and regardless of the velocity. This was the primary requirement for the AEB feature and can therefore be categorised as successful. The results of the second test in Section 9.4 shows that the vehicle is able to calculate the distance to an object and determine whether or not it is in the path of the vehicle. The results show a 5 cm difference between expected distance and the actual distance at which the AEB deactivates. This can be tolerated because the actual distance is longer than the expected distance and therefore safer than expected. The results are somewhat different from the most optimal results. This is caused by the use of a fault tolerance zone, which we have implemented to ensure the vehicle does not hit the object, if the input is erroneous. For future use, it is recommended to examine this safety measure, and develop a way to ensure the functionality of the AEB without using safety measures.

CHAPTER 10

Extended Autonomous Emergency Braking

In the following chapter, the theory and implementation of the Extended Autonomous Emergency Braking (EAEB) system is described. The idea of EAEB is to prevent a collision between the vehicle and an object, by attempting an automatic evasion manoeuvre. EAEB is an extension of the AEB system, described in Chapter 9.

10.1 Operation

The EAEB is triggered if the theoretical braking distance between the vehicle and an object ahead, is greater than the distance to the object. If this is the case it makes the vehicle turn to the left in order to avoid a collision with the object ahead.

10.2 Theory

When triggered, the main task of the Extended Autonomous Electronic Brake (EAEB) system is to calculate when the vehicle needs turn in order to avoid colliding with an object ahead.

To calculate when the vehicle must turn, in order to avoid colliding with the object ahead, trigonometry is used. The scenario can be imagined as a triangle as the scenario in figure Figure F10-1 illustrates.

The length of side A is how much the object ahead intersects with the vehicle plus a safe zone distance (so the vehicle will not just *barely* avoid the object). Through experimentation, we determined that the vehicle performs optimally, when $A = 25$.
The side B is the distance to the object from the vehicle. The distance to the object is known from the AEB.

It is necessary to determine B, as this is the distance at which, with the known data, the vehicle will be able to make the sharpest possible turn (40°), and still avoid the object.

The side A and the angle φ can then be used to calculate at which distance the vehicle must turn, in order to avoid the object when turning 40°.

The turn angle is illustrated by the angle α in the figure.

The definition $\tan(\alpha) = \frac{A}{B}$ can be used to isolate the side B as done in Equation (10.1).

$$\tan(\alpha) = \frac{A}{B} \Longrightarrow B = \frac{A}{\tan(\alpha)} \tag{10.1}$$

When the side B has been isolated, the known symbols are substituted and B is found, as done in Equation (10.2).

$$B = \frac{25\,cm}{\tan(40°)} = 29.7974\,cm \tag{10.2}$$

B is the distance the vehicle is required to be from the object, to turn 40° and at the same time avoid crashing into it.

In order to make sure the vehicle does not hit the object ahead, B is set to 35 cm in the implementation.

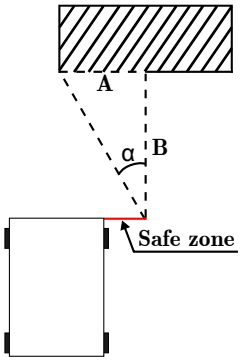


Figure F10-1: An illustration showing which data is used to calculate the amount of degrees to turn in order to avoid a collision with an object ahead.

10.3 Implementation

The EAEB algorithm is implemented as the `runEAEB()` function. The function is called by `runAEB()` if the AEB is activated (see lines 23-30 in Listing 9.3).

EAEB can have four different states, corresponding to the phases of avoiding an obstacle: *not activated*, *braking*, *turning* and *straightening*. *Not activated* is the default state, and does not influence the default behaviour of the vehicle. *Braking* is used both up until the vehicle is forced to turn away from the obstacle, and after the vehicle is clear of the obstacle. While EAEB is in this state, the controller task disables the throttle, and enables the brake. *Turning* forces the vehicle to turn its front wheels, in order to navigate clear of the obstacle. *Straightening* puts the front wheels back in neutral position.

The source code for `runEAEB()` is shown in Listing 10.1. The main functionality is as follows:

Line 5-8: If the vehicle has stopped, EAEB is deactivated, and control is returned to the driver.

Line 10-16: If the EAEB function is called, there must be a need for the EAEB. Thus, if the EAEB is in the *not activated* state it needs to be activated. The state changes to *braking*, and driver control of the front wheels is deactivated.

Line 18-28: While braking, the distance to the potential impact is continually updated. If a collision is inevitable, the EAEB switches to the *turning* state, and a number of variables are initialised in preparation for this.

Line 30-40: The vehicle turns while driving a set distance, to give the wheels time to actually turn, after which the state is changed to *straightening*.

Line 42-54: The vehicle drives a set distance while setting the front wheels back to neutral position. Afterwards the state is again changed to *braking*, ensuring that the vehicle stops completely, and control of the front wheels is returned to the driver.

```
1  void runEAEB()
2  {
3      float distanceToImpact = 0;
4
5      if (vehicle.vFront < 0.1)
6      {
7          aeb.eaebState = STATE_NOT_ACTIVATED;
8          steer.turnDegreesActivated = 0;
9
10     } else if(aeb.eaebState == STATE_NOT_ACTIVATED)
11     {
12         aeb.distance = 0.0;
13         aeb.eaebState = STATE_BRAKING;
14
15         // prevents user from steering while the EAEB is
               running
16         steer.turnDegreesActivated = 1;
17
18     } else if(aeb.eaebState == STATE_BRAKING)
19     {
20         distanceToImpact = getDistanceToImpact();
21
22         if(distanceToImpact < BRAKE_WAIT_DIST)
23         {
24             aeb.eaebState = STATE_TURNING;
25             aeb.distance = 0.0;
26             aeb.lastReadTick = tickCounter.
                   frontDegreesArray[tickCounter.
                   newIndexFront];
27             turnDegrees(MAX_LEFT_TURN);
28         }
29
30     } else if (aeb.eaebState == STATE_TURNING)
31     {
32         aeb.distance += getDistanceTravelled();
```

```
33
34          if (aeb.distance > TURN_DISTANCE)
35          {
36              aeb.eaebState = STATE_STRAIGHTENING;
37              aeb.distance = 0.0;
38              aeb.lastReadTick = tickCounter.
                    frontDegreesArray[tickCounter.
                    newIndexFront];
39              turnDegrees(0);
40          }
41
42      } else if (aeb.eaebState == STATE_STRAIGHTENING)
43      {
44          aeb.distance += getDistanceTravelled();
45
46          if (aeb.distance > STRAIGHTENING_WAIT_DIST)
47          {
48              aeb.distance = 0.0;
49              aeb.eaebState = STATE_BRAKING;
50
51              // allow the user to steer again
52              steer.turnDegreesActivated = 0;
53          }
54      }
55  }
```

Listing 10.1: Function that determines if the AEB or EAEB should be activated.

10.4 Evaluation

In order to evaluate the EAEB feature, a test was devised to determine if EAEB will activate in time, at all velocities. The vehicle drove straight towards an object with a set power between 65 and 100, and increments by 5 for each test. For each test the vehicle would accelerate until the highest possible velocity for that power was reached. The distance from the object to where EAEB started turning the vehicle, was measured. We expect the vehicle to start turning, before the distance to the object is less than 35 cm. Ideally, the vehicle would initially brake and start turning when the distance to the object was 35 cm, regardless of the velocity of the vehicle.

The result are shown in Section 10.4.

Distance from turning point to object								
Power	65	70	75	80	85	90	95	100
Distance (cm)	40	37	35	55	65	80	90	90

Table T10-1: Results from the EAEB test.

In all test cases, the vehicle starts turning before the distance to the object becomes shorter than 35 cm, but it is also clear, that the distances vary a lot depending on the vehicle's velocity, even though it should be constant. The fact that EAEB is activated so far from the object, means that the basic AEB will never be activated.

In conclusion, EAEB functions as a safety system, despite the overestimated distance to the object, at which the system activates. The overestimation is likely to be caused by imprecise sensor readings and lack of calibration. We would recommend using better sensors, and calibrating these sensors before each use. If this is done, the increased precision would make it possible to run EAEB and AEB at the same time, and only activate EAEB if needed.

CHAPTER 11

Lane Departure Emergency System

In this chapter, a description of the Lane Departure Emergency System (LDES) is presented. LDES helps the driver keep the vehicle in the lane. In Section 11.2, different approaches to implement LDES are described. In our project, we are using light sensors to detect markings on the road. In reality a camera combined with image analysis would be a better choice for LDES, according to Toyota Motor Corporation [5].

11.1 Operation

LDES is a system that should stop the vehicle, in case of an unintended lane departure. It is not considered an assistance system in the same sense, as for example cruise control, but merely a safety system that should only activate if the vehicle crosses a road marking without the driver giving signal, or steering the vehicle across the road marking. If such a situation arises and LDES is activated, it should attempt to adjust the vehicle's direction and activate the brake, until the driver takes control again, as demonstrated in Example 11.1.

Example 11.1 The vehicle drives in a lane, and slowly approaches a road marking on the right side of the vehicle. When the vehicle detects the road marking, it awaits the driver's reaction. When the driver doesn't react, and the vehicle crosses the road marking, LDES is activated, adjusting the direction of the vehicle and starts braking. The driver takes over again, and keeps going as before.

11.2 Design

When designing how the LDES should operate, different setups were considered. The setups varied in the amount of sensors and the design of the road markings.

The chosen LDES setup is a realistic one, consisting of two road markings, one on each side of the lane. The vehicle has two sensors mounted on the front; one on the left side and one on the right side. Each sensor has to detect the road marking on their respective side of the vehicle. The design of the vehicle and the road markings is shown in Figure F11-1.

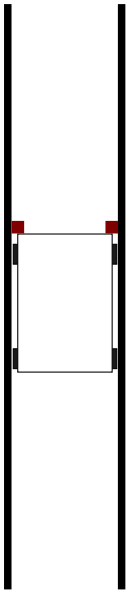


Figure F11-1: Illustration of LDES setup, showing the placement of the sensors and the road markings.

11.3 Control Flow

This section presents the control flow, shown in Figure F11-2, used in the implementation of LDES. First it is checked whether there is a road marking under one of the sensors. If a road marking is detected, the vehicle continues until the same sensor recognises the road again. Then it is checked whether it is the left or right sensor that crossed the road marking, after which the vehicle's direction is adjusted before it brakes.

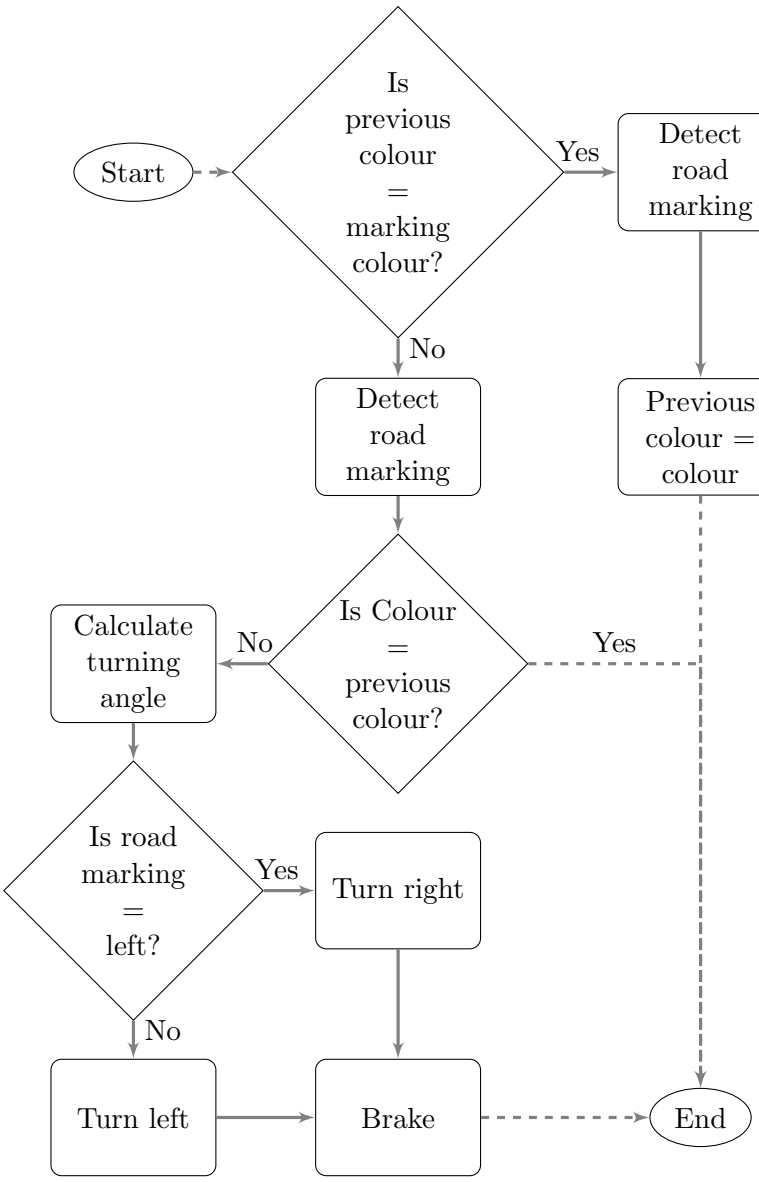


Figure F11-2: Illustration of the control flow in LDES.

11.4 Implementation

This section describes key functions used in LDES.

To detect road markings, the function `detectRoadMarkings()` is used. This function is shown in Listing 11.1. If a road marking is detected, `adjustVehicle()` is invoked. This function starts by calculating the angle by which the vehicle should be adjusted. The adjusting angle is found by calling `calculateTurnAngle()`. `adjustVehicle()` and `calculateTurnAngle()` is presented in Listing 11.2 and Listing 11.3 respectively.

```
1 U8 detectRoadMarking()
2 {
3     if (getLightSensorLeft() >= (roadMarkingColourLeft -
          COLOUR_SAFETY) && getLightSensorLeft() <= (
          roadMarkingColourLeft + COLOUR_SAFETY))
```

```
4     {
5         return LEFT_ROADMARKING_VALUE;
6     }
7     else if (getLightSensorRight() >= (
          roadMarkingColourRight - COLOUR_SAFETY) &&
          getLightSensorRight() <= (roadMarkingColourRight +
           COLOUR_SAFETY))
8     {
9         return RIGHT_ROADMARKING_VALUE;
10    }
11    else
12    {
13        return 0;
14    }
15 }
```

Listing 11.1: This code is used to check whether the left and right light sensors, recognise a road marking.

Line 3: It is checked whether the left sensor reads a value that matches a road marking, plus or minus the colour safety. The colour safety ensures that a road marking is recognised, even with a little difference between the sensor's value and the calibrated value.

Line 5: `LEFT_ROADMARKING_VALUE` is returned, if the left sensor recognises a road marking.

Line 7: The same as in line 3 is performed, but using the right sensor.

Line 9: `RIGHT_ROADMARKING_VALUE` is returned, if the right sensor recognises a road marking.

Line 13: 0 is returned, if no road marking is recognised by either sensor.

```
1  void adjustVehicle(U8 roadMarking)
2  {
3      S32 turnAngle = calculateTurnAngle(roadMarking);
4
5      if(turnAngle != 0)
6      {
7          steer.turnDegreesActivated = 1;
8
9          turnDegrees(turnAngle);
10
11         turnTimes++;
12         controlSteering();
13
14         currentTime = getSysTime();
15         while(getSysTime() - currentTime <
               BUSY_WAIT_TIME_LANECONTROL)
16         {
17             // busy wait
18         }
19
20         currentTime = getSysTime();
```

```
21          while(getSysTime() - currentTime <
                BUSY_WAIT_TIME_LANECONTROL)
22          {
23              controlMotors(ADJUSTING_SPEED_FORWARD,
                    BRAKE_OFF); //Drives forward.
24          }
25
26          aeb.activated = 1;
27
28          emptyArray(btReceiveBuf);
29
30          steer.turnDegreesActivated = 0;
31      }
32  }
```

Listing 11.2: `adjustVehicle()` is used to adjust the vehicle if a road marking is detected.

Line 3: `turnAngle` holds the angle which the vehicle should turn. The angle is calculated using `calculateTurnAngle()`, shown in Listing 11.3.

Line 7: `turnDegreesActivated` disables driver control of the front wheels, to ensure the vehicle turns as expected.

Line 9-12: `turnDegrees()` stores the turn angle, which is then used to turn the motor with the function `controlSteering()`. When turning, the turn counter `turnTimes` is incremented.

Line 14-18: `currentTime` is updated with the system time, after which the system waits, allowing the motor enough time to turn before continuing.

Line 20-24: `currentTime` is updated with the system time, after which the system waits, allowing the motors enough time to drive back on track before continuing.

Line 26: By setting the AEB flag, the vehicle brakes are activated.

Line 28: The array containing instructions from the controller, is cleared to ensure the vehicle does not act unexpectedly.

Line 30: `turnDegreesActivated` enables driver control once again, now that the vehicle is back on track.

```
1  S32 calculateTurnAngle(U8 roadMarking)
2  {
3      S32 turnAngle = 0;
4
5      /* If the car is driving fast, it will turn less than
           if it is driving slow this is to prevent crashing
           */
6      if(turnTimes >= TURNS_BEFORE_ANGLE_INCREASE)
7      {
8          turnAngle = MAX_TURNING_ANGLE;
9          turnTimes = 0; //Resets turnTimes.
```

```
10      }
11      else
12      {
13          turnAngle = MIN_TURNING_ANGLE; //Standard turn.
14      }
15
16      /* if the car has a lane on the left side, turn right
17         else if the car has a lane on the right turn left
              */
18      if (roadMarking == LEFT_ROADMARKING_VALUE)
19      {
20          return turnAngle; // turn right
21      }
22      else if (roadMarking == RIGHT_ROADMARKING_VALUE)
23      {
24          return -turnAngle; // turn left
25      }
26      else
27      {
28          return 0; // does not turn if already on the road
29      }
30  }
```

Listing 11.3: `calculateTurnAngle()` is used to calculate the angle for the vehicle to turn.

Line 6-14: The turn angle is chosen based on the number of turns, and stored in `turnAngle`.

Line 18-29: Based on whether or not a road marking is detected, and in which side, the appropriate turn angle is returned.

11.5 Evaluation

The evaluation of LDES is based on a functionality test. The test was conducted by forcefully driving the vehicle off the road, and noting the vehicle's reactions. We expected the vehicle to correct itself and stop.

In the test, the vehicle acted as expected. When the velocity was sufficiently high, the vehicle corrected itself before it braked. At lower velocities the vehicle only braked, as it would take too long to turn in that scenario. In some cases, the vehicle missed the road marking altogether.

The problem with the vehicle not always detecting the road marking, can be traced back to the way we read from the sensor. Currently, the LDES task requests data from the sensor directly, whereas a separate task to handle these requests, could potentially solve the problem. Another problem, is that the correction of the vehicle is only made, if the velocity is high enough. This may be caused by the busy wait used to turn, after the vehicle has crossed the road marking. For future use we recommend solving the problems described. We further suggest expanding the system to do one of the following.

1. When detecting a road marking, the vehicle should adjust and drive on.

2. Keeping the vehicle in the middle of the road.

Both of the suggestions require better sensors (perhaps cameras), to detect and measure the distance to the road markings.

CHAPTER 12

Adaptive Cruise Control

Adaptive cruise control (ACC) is based on the standard cruise control. The idea of the adaptive cruise control is to relieve the driver of any unpleasant sitting positions, as well as increase safety by keeping a distance to the leading vehicle, or braking if the distance becomes to small.

12.1 Operation

The standard cruise control is, as explained by Nice [17], a system that controls the velocity of a vehicle. The system is activated by the driver, and deactivated when the brake or clutch is engaged, or the driver turns it off. After the driver has set a desired velocity, the cruise control activates and steadily keeps the set velocity until deactivated. For the system to work, some data is needed about former velocity, current velocity and driven distance.

Howard [8] describes how the adaptive cruise control has the ability to detect vehicles in its path, and adjusting the velocity to match theirs. The driver then has the option of setting a desired distance between the vehicles. In order to determine the distance to the lead vehicle an ultrasonic sensor or similar is need.

12.2 Theory

When an adaptive cruise control receives a distance measurement from the ultrasonic sensor, the first task is to calculate the time it would take to drive that distance with the current velocity. The adaptive cruise control measures distance in time. By doing this the system can account for the braking time of the vehicle at any given moment, as well as the reaction time of the driver.

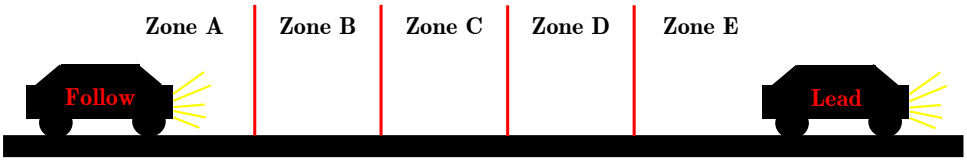


Figure F12-1: The zones of an adaptive cruise control.

When the distance between the vehicles is known, the system determines which of the zones the vehicle is in, based on the driver's settings for desired distance. The vehicle is in the zone when the prerequisites shown in Table T12-1 are fulfilled.

Zone	Prerequisites
Zone A	No lead vehicle is detected, or the distance to the lead vehicle is longer than what is configured as maximum distance for enabling adaptive cruise control.
Zone B	A lead vehicle is detected, and the distance to said vehicle is closer than the configured maximum distance. The distance to the lead vehicle is longer than the desired distance plus tolerated deviation.
Zone C	A lead vehicle is detected, and the distance to said vehicle is within the desired distance plus tolerated deviation.
Zone D	A lead vehicle is detected, and the distance to said vehicle is longer than the configured minimum distance for enabling adaptive cruise control. The distance to the lead vehicle is shorter than the desired distance plus tolerated deviation.
Zone E	A lead vehicle is detected and the distance to said vehicle is shorter than the configured minimum distance for enabling adaptive cruise control.

Table T12-1: Prerequisites that must be fulfilled in order for the vehicle to be in the zone.

When the zone is determined the appropriate action is taken. The action taken based on each zone is shown in Table T12-2.

Zone	Action
Zone A	The vehicle keeps driving with the set velocity, similar to a standard cruise control.
Zone B	The vehicle accelerates and increases velocity to shorten the distance between the vehicles.
Zone C	The vehicle will maintain the velocity or only adjust it slightly.
Zone D	The vehicle decelerates and decreases velocity to increase the distance between the vehicles.
Zone E	The vehicle brakes and rapidly decreases velocity.

Table T12-2: Actions taken when the vehicle is in each zone.

12.3 Implementation

When the adaptive cruise control is activated, the measurements from the sensor are stored in a variable and converted to time. The velocity of the lead vehicle is calculated based on the measurements, allowing the zone can be determined. The determination of the zone is shown in Listing 12.1.

```
1  void ACC(float actualTimeBetweenVehicles, S32
       distanceToVehicle, float leadVehicleVelocity)
2  {
3      if(actualTimeBetweenVehicles >
           MAXIMUM_TIME_BETWEEN_VEHICLES) //(Zone A)
4      {
5          /*Drive accordingly to standard Cruise Control*/
6          controlMotors(INITIAL_MOTOR_POWER, BRAKE_OFF);
7      }
8      else if(actualTimeBetweenVehicles <
           MINIMUM_TIME_BETWEEN_VEHICLES) //(Zone E)
9      {
10         brake();
11     }
12     else
13     {
14         if(leadVehicleVelocity > vehicle.vFront)
15         {
16             if((DESIRED_TIME_BETWEEN_VEHICLES -
                   MAXIMUM_DEVIATION) >
                   actualTimeBetweenVehicles &&
17                 MINIMUM_TIME_BETWEEN_VEHICLES <
                       actualTimeBetweenVehicles) //(Zone D)
18             {
19                 slowDown(MOTOR_POWER_ADJUST);
20             }
21             else //(Zone C)
22             {
23                 maintainDistance();
24             }
25         }
26         else
27         {
28             if((DESIRED_TIME_BETWEEN_VEHICLES +
                   MAXIMUM_DEVIATION) >=
                   actualTimeBetweenVehicles &&
29                 (DESIRED_TIME_BETWEEN_VEHICLES -
                      MAXIMUM_DEVIATION) <=
                      actualTimeBetweenVehicles) //(Zone C)
30             {
31                 maintainDistance();
32             }
33             else //(Zone B)
34             {
35                 speedUp(MOTOR_POWER_ADJUST);
36             }
37         }
38         acc.firstDistanceMeasurement = acc.
               secondDistanceMeasurement;
39     }
40 }
```

Listing 12.1: The function that determines which zone the vehicle is in.

Line 3-7: The vehicle is in zone A, and will therefore function similar to a standard cruise control.

Line 8-11: The vehicle is in zone E, and will therefore immediately brake.

Line 12: If the vehicle is neither too far away from or too close to a lead vehicle, one must assume it is within sight of a lead vehicle. Given this fact, the following assumptions are possible.

Line 14: If the distance to the lead vehicle increases as result of the velocity of the lead vehicle being fastest, one can safely assume the desired distance to the lead vehicle is longer than the current distance. Otherwise, the vehicle would have increased its velocity, so the lead vehicle would have been the slowest.

Line 16-24: If line 14 is true, the vehicle must be in either zone C or zone D. If the distance to the lead vehicle is within the desired distance and acceptable deviation, the vehicle is in zone C and therefore maintains the distance. If not, the vehicle is in zone D and the vehicle slows down.

Line 28-36: If line 14 is false, the vehicle must be in either zone B or zone C. If the distance to the lead vehicle is within the desired distance and acceptable deviation, the vehicle is in zone C and therefore maintains the distance. If not, the vehicle is in zone B and the vehicle speeds up.

12.4 Evaluation

The evaluation of adaptive cruise control is based on a test conducted using two vehicles. We have a "lead" vehicle driving in front of a "follow" vehicle. The follow vehicle has ACC enabled. We expect the velocity for the follow vehicle to first fall just below the velocity for the lead vehicle, and then rise to match the velocity of the lead vehicle, with a small offset. It would be acceptable if the graph for the follow vehicle briefly exceeds the graph of the lead vehicle, if it immediately decreases to match the graph of the lead vehicle. The optimal result would be if the graph for the follow vehicle matches the graph for the lead vehicle at all times.

The actual results are shown in Figure F12-2.

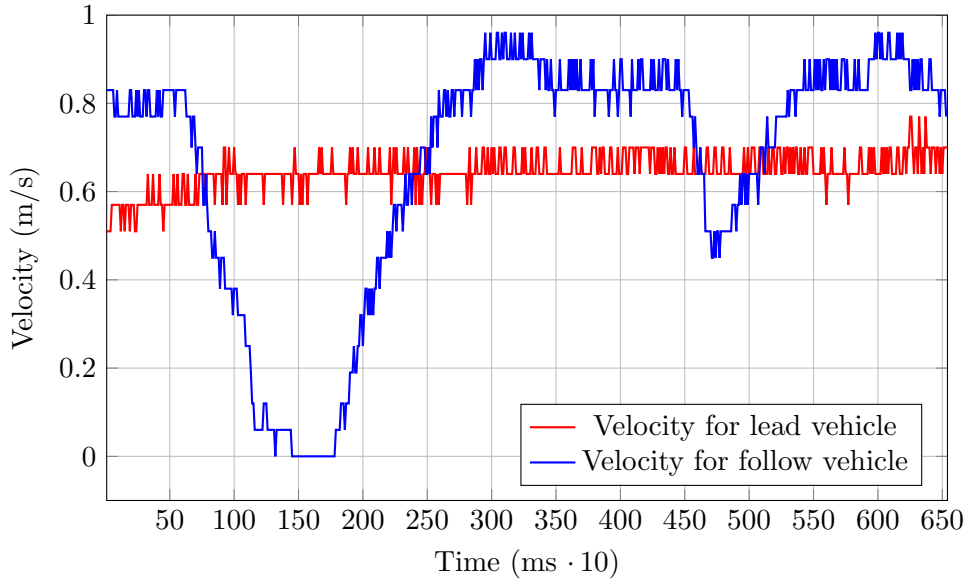


Figure F12-2: Graph for comparison of lead vehicle and follow vehicle.

The results in Figure F12-2 show how the ACC does not quite live up to the expectations. The greatest shortcomings are:

- The issue seen around $150 - 175$ ms in Figure F12-2 is most likely caused by the tolerated deviation being too short. This leads to the follow vehicle quickly finding itself in zone E, and reacts by braking.
- The issue seen around $250 - 300$ ms in Figure F12-2 is caused by the follow vehicle's miscalculation of the lead vehicle's velocity. Believing the lead vehicle has a higher velocity than it actually has, the follow vehicle accelerates too much and moves into zone D, where it decelerates.
- If the set velocity of the follow vehicle is much higher than the velocity of the lead vehicle, the acceleration will be so fast that the follow vehicle will accelerate and decelerate continuously, making its movement irregular. This is caused by the follow vehicle reaching the lead vehicle too fast and then decelerate or brake until the lead vehicle once again has gained enough distance to the follow vehicle. This issue, in turn, is most likely caused by the tolerated deviation being too short, and therefore, the follow vehicle not being able to slowly accelerate or decelerate in the respective zones. If an adjustment of the tolerated deviation does not correct the issue, it would help to make the acceleration increase slowly instead of setting the desired power.

Although ACC has some shortcomings, it works decently given optimal surroundings. Before using this feature in the future, it is recommended to correct the described issues.

CHAPTER 13

Real-Time Analysis of the System

Analysis of a real-time system is important, because it is essential to ensure that the timing, priorities and scheduling of tasks, allow the system to function correctly at all times.

The analysis of the tasks used for this project focused on the best case and worst case running times of each task. Even though it may seem superfluous to measure a best case running time for each task, this is not the case. The best case running time measurements are used to model the system with Petri nets using a software tool called TAPAAL [6].

13.1 WCET Analysis

In order to measure worst case execution time (WCET), the following approach was taken.

The "worst" path through each of the tasks' code must be taken. The worst path through the code is defined as the path that requires the most time to execute. When the worst path through a task's code has been determined, the time required to execute this path an arbitrarily large number of times, is measured. After this, the average execution time of all the test runs, is calculated. This is the average worst execution time. The reason the average execution time is used as WCET, is that the clock on the NXT does not have enough granularity to measure the task execution times. The granularity is 1 ms. Therefore an average must be taken in order to be able to measure task execution times below 1 ms.

The measurements of the best case execution times (measured because it is needed for modelling in TAPAAL) of each task were done in a way similar to the measurements of the WCET execution times.

A table of WCET for each task can be seen in Table T13-1.

Task	Worst (ms)	Best average (ms)	Worst average (ms)	No. of tests
ReadRawDataTask	1	0.0202	0.0266	100,000
CalculateVehicleSpeedTask	1	0.0603	0.0603	100,000
ControllerTask	1	0.0748	0.3067	100,000
DataLoggingTask	1	0.0230	0.0313	100,000
LDESTask	4	0.0053	4	100,000
AEBTask	1	0.0075	0.3437	100,000
AdaptiveCruiseControlTask	1	0.0289	0.0374	100,000
SonarTask	1	0.1173	0.1852	100,000

Table T13-1: Table with the systems worst, best and worst average runtimes.

When performing a utilisation analysis on a system, it is necessary to know the *computational cost* of each task. For this analysis, we will use the average WCET of each task, as the computational cost.

13.2 Utilisation-Based Schedulability Analysis

This section describes how to calculate the utilisation factor of a set of tasks. Also, it explains the calculation of the least upper bound utilisation, which is used in rate monotonic scheduling when determining if the tasks are schedulable.

13.2.1 Calculation of Utilisation Factor

This section is based on chapter 13 in [9].

In order to check whether a set of tasks, Γ, is schedulable with a scheduling algorithm, A, a utilisation analysis can be performed. The utilisation analysis is performed to guarantee that a set of tasks can be scheduled with a specific scheduling algorithm. The utilisation factor U represents how much processor time is used to execute a given task set. If U is below a certain threshold, U_{lub}, which each scheduling algorithm has, Γ is schedulable with A.

The utilisation factor, U, given a set, Γ, of N periodic tasks, $\Gamma = \{\tau_1..\tau_N\}$, is given by the sum of each task's computational cost divided by its period:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i}$$

Where C_i is the computational cost of task τ_i (in our case the task's average WCET), and T_i is the period of the task.

If the utilisation factor $U > 1$, the task set is not schedulable with *any* algorithm. Conversely, the task set is schedulable by *some* algorithm, if $U \leq 1$. The utilisation factor for the set of tasks in this project can be seen in table Table T13-2.

Task	**T** (ms)	**C** (ms)	**U**
ReadRawDataTask	5	0.0266	0.53%
CalculateVehicleSpeedTask	5	0.0603	1.21%
ControllerTask	10	0.3067	3.07%
Datalogging	10	0.0313	0.31%
LDESTask	20	4	20.00%
AEBTask	25	0.3437	1.37%
AdaptiveCruiseControlTask	25	0.0374	0.15%
SonarTask	30	0.1852	0.62%
Utilisation factor:			27.26%

Table T13-2: Table showing the utilisation factor, U, for each task based on the period, T, and cost, C of each task.

13.3 Response-Time Based Schedulability Analysis

This section is based on chapter 14 in [9].

Response time analysis (RTA) is an exact (necessary and sufficient) schedulability test for any fixed-priority assignment scheme on single-processor systems. With it, it is possible to predict the worst case response time (WCRT) of each task in a task set.

The WCRT of a task depends on the interference it can possibly experience. The sources of the possible interferences, are tasks with higher priorities than the task itself, as they have the ability to preempt it.

When the WCRT analysis is completed, the results can be compared to the deadlines of the individual tasks in the task set. This is useful, because it tells us whether all tasks in a task set, are guaranteed to meet their respective deadlines.

The WCRT, R, of a task, τ_i, is defined as the recursive relation seen in Equation (13.1)

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil \cdot C_j \tag{13.1}$$

C_i is the cost of τ_i, $hp(i)$ is the set of tasks with higher priorities than τ_i and T_j is the period of task $\tau_j \in hp(i)$.

When $R_i^{n+1} = R_i^n$, an equilibrium has been reached and R_i^{n+1} is the WCRT of the task τ_i.

Table T13-3 shows the response times and deadlines for each task in this project's task set. A task, τ_i, meets its deadline, D_i, if and only if $R_i \leq D_i$. As can be seen in Table T13-3, none of the tasks' response times exceed their deadlines. Therefore it can be concluded, that no task in the task set

exceeds its deadline.

The calculations for the task set can be found in Appendix A

Task	**Deadline** D_i (ms)	**Response time** R_i (ms)
ReadRawData	5	0.0266
CalculateSpeed	5	0.0603
Controller	10	0.3936
Datalogging	10	0.1182
LDES	20	4.4249
AdaptiveCC	25	4.4623
AEB	25	4.7420
Sonar	30	4.9912

Table T13-3: Table showing the task set and their respective response times.

In rate monotonic scheduling, the least upper bound utilisation factor, U_{lub}, depends on the number of tasks. If $U \leq U_{lub}$, it means the task set Γ is schedulable using rate monotonic scheduling. The formula used for calculating U_{lub} is shown in Equation (13.2), where N is the number of tasks in the system.

$$U_{lub} = N(2^{\frac{1}{N}} - 1) \tag{13.2}$$

In our system there are 8 tasks, and U_{lub} is calculated in Equation (13.3).

$$\begin{aligned} U_{lub} &= 8(2^{\frac{1}{8}} - 1) \\ &\Downarrow \\ U_{lub} &= 0.724 \\ &\Downarrow \\ U_{lub} &= 72.4\% \end{aligned} \tag{13.3}$$

As shown in Table T13-2, the utilisation factor of our task set is 27.26 %, which means it is schedulable using rate monotonic scheduling.

Response-Time Calculation Example

In this example, calculation of the WCRT of *ControllerTask* from Appendix A is explained.

The non-expanded formula for calculating the WCRT is listed in Equation (13.1). Since the formula calculates the WCRT by adding the cost of the current task with the sum of the WCRT's of tasks with higher priorities than itself, it is needed to check which tasks have higher priorities. As can be seen in Table T13-4, the tasks with higher priorities than *ControllerTask* are *ReadRawDataTask* and *CalculateVehicleSpeedTask*.

Task τ	**Priority**	**Period** (ms) T	**WCET** (ms) C
ReadRawDataTask	5	5	0.0266
CalculateVehicleSpeedTask	5	5	0.0603
ControllerTask	4	10	0.3067
DataLoggingTask	4	10	0.0313
LDESTask	3	20	4.0000
AEBTask	2	25	0.3437
AdaptiveCruiseControlTask	2	25	0.0374
SonarTask	1	30	0.1852

Table T13-4: Table with task data for RTA.

In order to calculate the WCRT of a task it is helpful to expand the formula before filling in actual task values. Since it is now known which tasks can interfere with *ControllerTask*, the formula can be expanded, as shown in Equation (13.4):

$$\begin{aligned} R^{n+1}_{ControllerTask} &= C_{ControllerTask} \\ &+ \left\lceil \frac{R^{n}_{ControllerTask}}{T_{ReadRawDataTask}} \right\rceil \cdot C_{ReadRawDataTask} \\ &+ \left\lceil \frac{R^{n}_{ControllerTask}}{T_{CalculateVehicleSpeedTask}} \right\rceil \cdot C_{CalculateVehicleSpeedTask} \end{aligned} \tag{13.4}$$

First, $R^{0}_{ControllerTask}$ must be calculated using information from Table T13-4. This is shown in Equation (13.5).

$$\begin{aligned} R^{0}_{ControllerTask} &= 0.3067 + \left\lceil \frac{0.3067}{5} \right\rceil \cdot 0.0603 + \left\lceil \frac{0.3067}{5} \right\rceil \cdot 0.0266 \\ &= 0.3936 \end{aligned} \tag{13.5}$$

We keep doing this until an equilibrium is found where the next calculation equals the previous, so

$$R^{n+1}_{i} = R^{n}_{i}$$

. This is the WCRT of the task. Therefore the next iteration, $R^{1}_{ControllerTask}$, of the formula in Equation (13.4) is done. This time $R^{n}_{ControllerTask}$ is substituted with the result just found, 0.3936. This calculation is shown in Equation (13.6).

$$\begin{aligned} R^{1}_{ControllerTask} &= 0.3067 + \left\lceil \frac{0.3936}{5} \right\rceil \cdot 0.0603 + \left\lceil \frac{0.3936}{5} \right\rceil \cdot 0.0266 \\ &= \mathbf{0.3936} = \mathbf{R^{0}_{ControllerTask}} \end{aligned} \tag{13.6}$$

An equilibrium was found, and therefore the WCRT of the task *ControllerTask* is 0.3936 ms.

13.4 TAPAAL Verification

To confirm that the schedulability of the system, we decided to recreate parts of it in a model-checking tool. We decided to use TAPAAL [6], an application for verifying timed-arc Petri nets (abbr. TAPN).

13.4.1 Timed-Arc Petri Nets

TAPN's differentiate themselves from traditional Petri nets by allowing the modelling of time-dependent systems. How this changes the traditional Petri net concepts, is summed up in the following:

Tokens: Tokens contain a real-time age, that synchronously increments through *time delays*. In TAPAAL, tokens are shown as a number representing their age.

Places: Places can have a time invariant, that limits how old a token in that place can become. If a token reaches the maximum age allowed by the invariant, and there is no enabled transition anywhere, the model is in a deadlock. In TAPAAL, places are visualised as black circles with white fill.

Transitions: In TAPN, there is a subtype of transition, called an *urgent transition*. When an urgent transition is enabled, it is not possible to do a time delay. In TAPAAL transitions are visualised as black squares. An urgent transition has a white dot in the middle.

Arcs: Arcs can have a time guard, that defines an upper and/or lower limit for the age of tokens travelling through the arc. Tokens do not retain their age when travelling through normal arcs, i.e. their age in the destination place is 0. In TAPAAL arcs are visualised as arrows. There exists to subtypes of arcs:

- **Transport arcs:** These arcs allow tokens to retain their age in the destination place. In TAPAAL, transport arcs have a diamond in place of an arrowhead.
- **Inhibitor arcs:** These arcs always start at places and end at transitions. They do not allow tokens to travel, but instead allow tokens to control each others behaviour. When the source place of an inhibitor arc contains a token, the destination transition is disabled. In TAPAAL, inhibitor arcs have a black circle with white fill, in place of an arrowhead.

13.4.2 TAPAAL Components

In TAPAAL a full model can be built from a number of components, each consisting of a separate TAPN. The components are tied together by shared transitions and places. There is only one instance of a shared transition or place, but it can appear across all the components. This allows one to split a large TAPN into several smaller units, making the model easier to both create and read. Shared transitions and places are marked with a dotted line around them.

13.4.3 Modelling the System

The basic idea of our model was to recreate the task life cycle. We made a separate component for each task, with each component modelling the life cycle of that specific task. Figure F13-1 shows the model of the *LaneControlTask*. All other tasks are modelled in a similar manner.

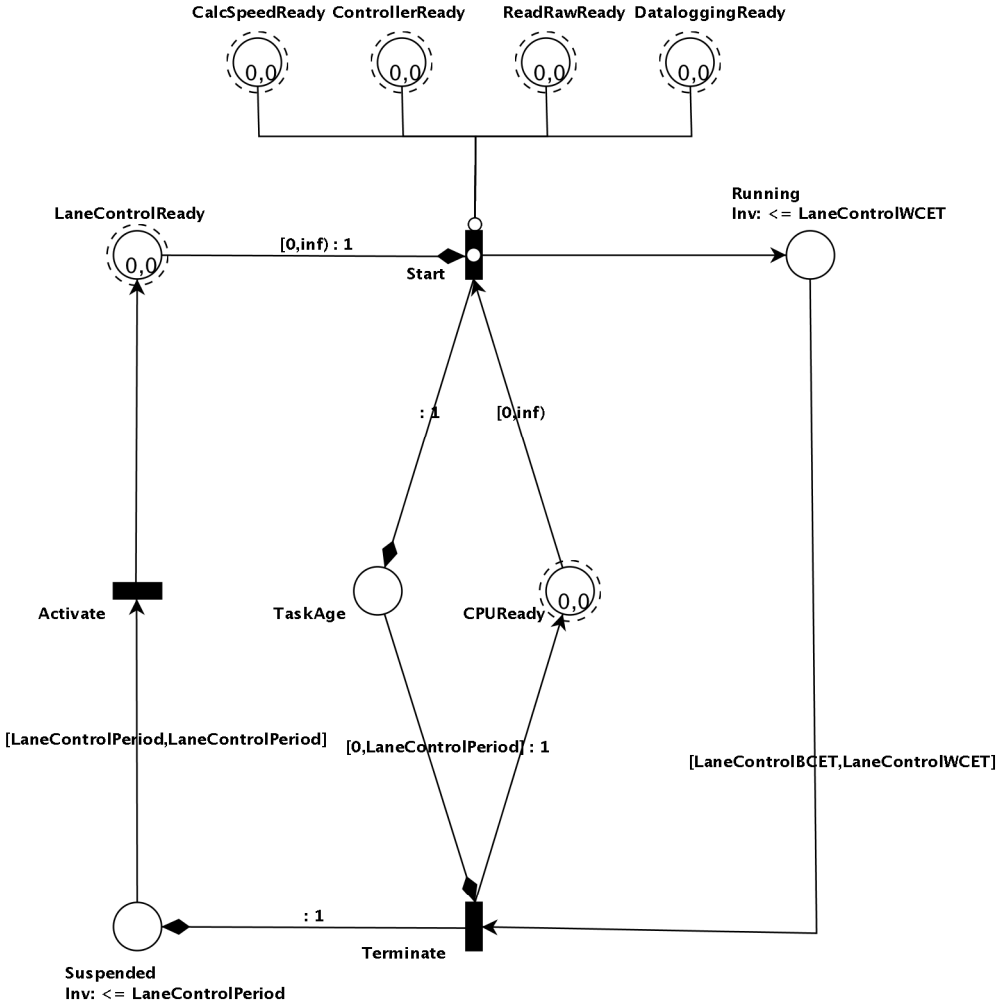


Figure F13-1: An example of a task modelled in TAPAAL.

The main shared place is *CPUReady*. *CPUReady* illustrates the availability of the processor, through the presence of a token. If a token is present, the CPU can be assigned to a ready task. This is modelled by having an arc from *CPUReady* to the *Start* transition of each task. Furthermore, all the task's *Ready* places are shared, to allow the simulation of task priorities. Higher priority tasks have inhibitor arcs from their *Ready* place to the *Start* transitions of lower priority tasks. Thus, if a high priority task is in its *Ready* place, low priority tasks are unable fire their *Start* transitions.

All tasks start with a token in their ready state. They can fire the *Start*-transition if the CPU is ready, and if no higher-priority tasks are ready. After firing the *Start*-transition, a token in *Running* keeps track of the task's execution time, while a token in *TaskAge* keeps track the time elapsed since the task was released. When the token in *Running* has an age between the task's best case and worst case execution time, it is allowed to fire the *Terminate*-transition, and move to *Suspended*, while the CPU is released. In

Suspended, the task continues to monitor how much time has elapsed since its last release. When the token's age equals the task period, it is forced to fire the *Activate*-transition, and again enter its *Ready*-place.

Preemption and waiting is not modelled, as TAPN's are unable to recreate the general case of these situations. It could be done for the specific cases of our tasks, buy creating a *Running*-place for each discrete time unit of the task's execution time. However, we chose not to spend time on this, specially as the case of *LaneControlTask* would require 40 additional places and transitions.

After the model had been created, we asked TAPAAL to verify that there was no way for the model to go into deadlock, i.e. go into a state where it is impossible to continue due to timing restraints. TAPAAL confirmed that it was indeed not possible to go into deadlock (as shown in Figure F13-2), thus the schedulability was also verified by software modelling.

Figure F13-2: The result of the TAPAAL verification.

CHAPTER 14

Conclusion

This project was aimed towards implementing an automatic system to improve traffic safety on a vehicle, using the LEGO Mindstorms platform. It was also a goal to determine if this system would be schedulable in a real-time operating system.

The system is built on the LEGO Mindstorms NXT platform, which provides developers with a variety of sensors. The sensors allow the system to act on the environment around it, but some of the sensors have limited reliability. This is especially true in the case of the ultrasonic sensor, as described in Section 3.5. It is of course a limitation of the project format, but a more complex platform would definitely benefit the overall quality of the project.

Because readings from the ultrasonic sensor were very unreliable, it was necessary to filter them before use, to assure the calculations were precise enough. We do this, as described in Section 5.2, by discarding the readings that differ from the average of three readings by more than the standard deviation. Furthermore, after sending the filtered readings to the AEB feature, they are filtered again. This second filtering is not only redundant, but also increases the response time of the feature. Despite this, the feature still functions properly, but removing the second filtering would be an easy improvement to make. In addition to that, the cyclic arrays used, only has three elements, which might prove problematic in some cases, as the sensor might give two erroneous readings, making it impossible to determine which is the right result. In addition to this, the calculation used to filter it is not useful, unless the three elements in the array are close to each other. This might not be the case when the sensor returns an erroneous reading. One filtering phase, with more elements in the array, and based on a comparison with the median instead of the average of the readings, would result more reliable readings, as a few wrong results will not influence the median in the same way as an average value.

The system is developed for the nxtOSEK operating system, which is described in Chapter 4. nxtOSEK provides functions to communicate with the LEGO NXT sensors. The nxtOSEK operating system has been relatively easy to use for developing a real-time system, and most of our problems were caused by the lack of knowledge about vehicle mechanics. Debugging on the nxtOSEK operating system was, however, very difficult, since we were not able to attach a debugger.

The purpose of the ABS is to prevent the wheels from locking up while braking, and thereby achieve better braking distance and controllable brak-

ing. The ABS was developed with some limitations, since a real vehicle brakes on all its wheels, and we were only able to brake on the rear wheels, since these were the ones attached to the motors. Also, we did not have analogue brakes, and were only able to have the brakes fully activated or not activated at all. In spite of these limitations, we were able to improve the brake distance and make braking more controlled. This is described in Section 6.6.

TCS assists the driver while accelerating, by preventing the driving wheels from spinning around, and thus giving more traction to the vehicle. A problem when developing this system was that the wheels had too much traction, resulting in the car not being able to spin the wheels while accelerating. To solve this, we came up with some custom plastic wheels to achieve less traction. With the implementation, improvements were achieved both when accelerating and turning. The evaluation is described in Section 7.4.

The purpose of EDC is to prevent the wheels from losing traction while turning. This is achieved by decreasing the velocity of the inner driving wheel, thus pushing the car around it with a higher outer velocity. An issue with our implementation was caused by the fact that only four sensors can be connected to the NXT brick. This means that we are only able to have one rotation sensor attached to our system. This causes EDC to function improperly when turning to the left. Because the rotation sensor is mounted on the left front wheel, the velocity of the left rear wheel will be lowered too much. To properly adjust the rear wheels when turning, it should be lowered according to the outer front wheel. Despite the issue we got good results while turning both left and right. Section 8.4 describes the tests.

To assist the driver in avoiding collisions, AEB was implemented. AEB works by automatically stopping the vehicle before an impact. This happens if the system detects an object within the vehicle's path. During development of the AEB we encountered a problem with the ultrasonic sensor delivering unreliable results. This resulted in erroneous activations of the AEB. This, however, was partly solved by filtering the data received from the sensor, but error margins were still an issue, and therefore we had to use a constant fault tolerance zone. The filtering is described in Section 5.2. Our resulting implementation yields overall good results, as shown in Section 9.4.

EAEB extends AEB by allowing the vehicle to evade, if an object gets so close that the vehicle is unable to brake before impact. For EAEB the same problem with sensor reliability, encountered in AEB, is present. Furthermore EAEB has the limitation that we assume that the object to evade is on the right side of the vehicle's path, since the ultrasonic sensor cannot determine on which side of the vehicle, the detected object is present. The evaluation of EAEB is presented in Section 10.4.

To prevent unintended lane departure the driver is assisted by LDES. To provide this assistance, LDES adjusts the vehicle's direction and velocity after crossing a road marking. The problem with the LDES feature is that we are unable to guarantee that the vehicle detects the road marking in time. As mentioned in Section 11.5, a solution to this problem could be giving

the sensors their own task, which is then executed more often. A different solution could be upgrading the sensors, to allow more precise adjustments of the vehicle.

ACC assists the driver in keeping a steady distance to the vehicle ahead, by adjusting its own velocity accordingly. If there is no vehicle ahead it maintains a velocity set by the driver. Similarly to AEB and EAEB the ultrasonic sensor causes problems. As before, the problem is solved by filtering, as described in Section 5.2. When a vehicle follows another vehicle, the distance between the vehicles is not constant, which makes the pursuing vehicle's velocity oscillate slightly. With this feature we achieved an acceptable result as the velocity oscillation is a minor issue. The evaluation of ACC can be found in Section 12.4.

In order to determine the schedulability of the task set in our real-time system, a utilisation analysis of the task set was carried out. The analysis showed that the task set is schedulable on a single core system with a scheduler that supports preemption, with any fixed-priority scheduling algorithm and that all tasks are guaranteed to meet their respective deadlines.

We also used a model checking tool, TAPAAL, to create and verify a model of the system. While the tool confirmed the schedulability of the system, the model did not take waiting and preemption into account. The confirmation indicates, that even if preemption was disabled, the system should still work. Disabling preemption has its own benefits, e.g. eliminating the risk of race conditions.

While the utilisation analysis verified schedulability, while taking preemption and waiting into account, the amount of calculations was substantial. Small changes to the system might invalidate the results, and the calculations would likely have to be redone entirely. The model on the other hand, is easy to change as the project evolves. This however, has some limitations in regards to the more complex aspects of scheduling, such as interference from other tasks. As discussed in Section 13.4, we could model these aspects for the specific case of each task, but that would make the model far less flexible. One could solve this, by creating a tool that generated the model automatically, with the task parameters as input. In a larger project, modelling would make sense, as the utilisation analysis quickly becomes unwieldy. This however, relies on the system model being correct.

APPENDIX A

Response Time Analysis Calculations

Task i	**Priority**	**Period** (ms) T_j	**WCET** (ms) C_i
ReadRawDataTask	5	5	0.0266
CalculateVehicleSpeedTask	5	5	0.0603
ControllerTask	4	10	0.3067
DataLoggingTask	4	10	0.0313
LDESTask	3	20	4.0000
AEBTask	2	25	0.3437
AdaptiveCruiseControlTask	2	25	0.0374
SonarTask	1	30	0.1852

Table T1-1: Table with task data for RTA.

ReadRawDataTask

$$R^{n+1}_{ReadRawDataTask} = C_{ReadRawDataTask} + \sum_{j=hp(ReadRawDataTask)} \left\lceil \frac{R^n_{ReadRawDataTask}}{T_j} \right\rceil C_j =$$

$$R^{n+1}_{ReadRawDataTask} = C_{ReadRawDataTask} + 0 \cdot 0$$

$$R^0_{ReadRawDataTask} = 0.0266$$

$$R^1_{ReadRawDataTask} = \mathbf{0.0266} = \mathbf{R^0_{ReadRawDataTask}}$$

CalculateVehicleSpeedTask

$$R^{n+1}_{CalculateVehicleSpeedTask} = C_{CalculateVehicleSpeedTask} + \sum_{j=hp(CalculateVehicleSpeedTask)} \left\lceil \frac{R^n_{CalculateVehicleSpeedTask}}{T_j} \right\rceil C_j =$$

$$R^{n+1}_{CalculateVehicleSpeedTask} = C_{CalculateVehicleSpeedTask} + 0 \cdot 0$$

$$R^0_{CalculateVehicleSpeedTask} = 0.0603$$

$$R^1_{CalculateVehicleSpeedTask} = \mathbf{0.0603} = \mathbf{R^0_{CalculateVehicleSpeedTask}}$$

ControllerTask

$$R^{n+1}_{ControllerTask} = C_{ControllerTask} + \sum_{j=hp(ControllerTask)} \left\lceil \frac{R^n_{ControllerTask}}{T_j} \right\rceil C_j =$$

$$\begin{aligned} R^{n+1}_{ControllerTask} &= C_{ControllerTask} \\ &+ \left\lceil \frac{R^n_{ControllerTask}}{T_{ReadRawDataTask}} \right\rceil \cdot C_{ReadRawDataTask} \\ &+ \left\lceil \frac{R^n_{ControllerTask}}{T_{CalculateVehicleSpeedTask}} \right\rceil \cdot C_{CalculateVehicleSpeedTask} \end{aligned}$$

$$\begin{aligned} R^0_{ControllerTask} &= 0.3067 + \left\lceil \frac{0.3067}{5} \right\rceil \cdot 0.0603 + \left\lceil \frac{0.3067}{5} \right\rceil \cdot 0.0266 \\ &= 0.3936 \\ R^1_{ControllerTask} &= 0.3067 + \left\lceil \frac{0.3936}{5} \right\rceil \cdot 0.0603 + \left\lceil \frac{0.3936}{5} \right\rceil \cdot 0.0266 \\ &= \mathbf{0.3936} = \mathbf{R^0_{ControllerTask}} \end{aligned}$$

DataLoggingTask

$$R^{n+1}_{DataLoggingTask} = C_{DataLoggingTask} + \sum_{j=hp(DataLoggingTask)} \left\lceil \frac{R^n_{DataLoggingTask}}{T_j} \right\rceil C_j =$$

$$\begin{aligned} R^{n+1}_{DataLoggingTask} &= C_{DataLoggingTask} \\ &+ \left\lceil \frac{R^n_{DataLoggingTask}}{T_{ReadRawDataTask}} \right\rceil \cdot C_{ReadRawDataTask} \\ &+ \left\lceil \frac{R^n_{DataLoggingTask}}{T_{CalculateVehicleSpeedTask}} \right\rceil \cdot C_{CalculateVehicleSpeedTask} \end{aligned}$$

$$\begin{aligned} R^0_{DataLoggingTask} &= 0.0313 + \left\lceil \frac{0.0313}{5} \right\rceil \cdot 0.0603 + \left\lceil \frac{0.0313}{5} \right\rceil \cdot 0.0266 \\ &= 0.1182 \\ R^1_{DataLoggingTask} &= 0.0313 + \left\lceil \frac{0.1182}{5} \right\rceil \cdot 0.0603 + \left\lceil \frac{0.1182}{5} \right\rceil \cdot 0.0266 \\ &= \mathbf{0.1182} = \mathbf{R^0_{DataLoggingTask}} \end{aligned}$$

LDESTask

$$R^{n+1}_{LDESTask} = C_{LDESTask} + \sum_{j=hp(LDESTask)} \left\lceil \frac{R^n_{LDESTask}}{T_j} \right\rceil C_j =$$

$$\begin{aligned} R^{n+1}_{LDESTask} &= C_{LDESTask} \\ &+ \left\lceil \frac{R^n_{LDESTask}}{T_{DataLoggingTask}} \right\rceil \cdot C_{DataLoggingTask} \\ &+ \left\lceil \frac{R^n_{LDESTask}}{T_{ControllerTask}} \right\rceil \cdot C_{ControllerTask} \\ &+ \left\lceil \frac{R^n_{LDESTask}}{T_{ReadRawDataTask}} \right\rceil \cdot C_{ReadRawDataTask} \\ &+ \left\lceil \frac{R^n_{LDESTask}}{T_{CalculateVehicleSpeedTask}} \right\rceil \cdot C_{CalculateVehicleSpeedTask} \end{aligned}$$

$$\begin{aligned} R^0_{LDESTask} &= 0.4249 \\ &+ \left\lceil \frac{0.4249}{10} \right\rceil \cdot 0.0313 \\ &+ \left\lceil \frac{0.4249}{10} \right\rceil \cdot 0.3067 \\ &+ \left\lceil \frac{0.4249}{5} \right\rceil \cdot 0.0266 \\ &+ \left\lceil \frac{0.4249}{5} \right\rceil \cdot 0.0603 \\ &= 4.4249 \end{aligned}$$

$$\begin{aligned} R^1_{LDESTask} &= 0.4249 \\ &+ \left\lceil \frac{4.4249}{10} \right\rceil \cdot 0.0313 \\ &+ \left\lceil \frac{4.4249}{10} \right\rceil \cdot 0.3067 \\ &+ \left\lceil \frac{4.4249}{5} \right\rceil \cdot 0.0266 \\ &+ \left\lceil \frac{4.4249}{5} \right\rceil \cdot 0.0603 \\ &= \mathbf{4.4249} = \mathbf{R^0_{LDESTask}} \end{aligned}$$

AEBTask

$$R^{n+1}_{AEBTask} = C_{AEBTask} + \sum_{j=hp(AEBTaskS)} \left\lceil \frac{R^n_{AEBTask}}{T_j} \right\rceil C_j =$$

$$\begin{aligned} R^{n+1}_{AEBTask} = {} & C_{AEBTask} \\ & + \left\lceil \frac{R^n_{AEBTask}}{T_{LDESTask}} \right\rceil \cdot C_{LDESTask} \\ & + \left\lceil \frac{R^n_{AEBTask}}{T_{DataLoggingTask}} \right\rceil \cdot C_{DataLoggingTask} \\ & + \left\lceil \frac{R^n_{AEBTask}}{T_{ControllerTask}} \right\rceil \cdot C_{ControllerTask} \\ & + \left\lceil \frac{R^n_{AEBTask}}{T_{ReadRawDataTask}} \right\rceil \cdot C_{ReadRawDataTask} \\ & + \left\lceil \frac{R^n_{AEBTask}}{T_{CalculateVehicleSpeedTask}} \right\rceil \cdot C_{CalculateVehicleSpeedTask} \end{aligned}$$

$$\begin{aligned} R^0_{AEBTask} = {} & 0.3437 \\ & + \left\lceil \frac{0.3437}{20} \right\rceil \cdot 4.0000 \\ & + \left\lceil \frac{0.3437}{10} \right\rceil \cdot 0.0313 \\ & + \left\lceil \frac{0.3437}{10} \right\rceil \cdot 0.3067 \\ & + \left\lceil \frac{0.3437}{5} \right\rceil \cdot 0.0266 \\ & + \left\lceil \frac{0.3437}{5} \right\rceil \cdot 0.0603 \\ = {} & 4.7420 \end{aligned}$$

$$\begin{aligned} R^1_{AEBTask} = {} & 0.3437 \\ & + \left\lceil \frac{4.7420}{20} \right\rceil \cdot 4.0000 \\ & + \left\lceil \frac{4.7420}{10} \right\rceil \cdot 0.0313 \\ & + \left\lceil \frac{4.7420}{10} \right\rceil \cdot 0.3067 \\ & + \left\lceil \frac{4.7420}{5} \right\rceil \cdot 0.0266 \\ & + \left\lceil \frac{4.7420}{5} \right\rceil \cdot 0.0603 \\ = {} & \mathbf{4.7420 = R^0_{AEBTask}} \end{aligned}$$

AdaptiveCruiseControlTask

$$R^{n+1}_{AdaptiveCruiseControlTask} = C_{AdaptiveCruiseControlTask} + \sum_{j=hp(AdaptiveCruiseControlTask)} \left\lceil \frac{R^{n}_{AdaptiveCruiseControlTask}}{T_j} \right\rceil C_j =$$

$$\begin{aligned} R^{n+1}_{AdaptiveCruiseControlTask} = {} & C_{AdaptiveCruiseControlTask} \\ & + \left\lceil \frac{R^{n}_{AdaptiveCruiseControlTask}}{T_{LDESTask}} \right\rceil \cdot C_{LDESTask} \\ & + \left\lceil \frac{R^{n}_{AdaptiveCruiseControlTask}}{T_{DataLoggingTask}} \right\rceil \cdot C_{DataLoggingTask} \\ & + \left\lceil \frac{R^{n}_{AdaptiveCruiseControlTask}}{T_{ControllerTask}} \right\rceil \cdot C_{ControllerTask} \\ & + \left\lceil \frac{R^{n}_{AdaptiveCruiseControlTask}}{T_{ReadRawDataTask}} \right\rceil \cdot C_{ReadRawDataTask} \\ & + \left\lceil \frac{R^{n}_{AdaptiveCruiseControlTask}}{T_{CalculateVehicleSpeedTask}} \right\rceil \cdot C_{CalculateVehicleSpeedTask} \end{aligned}$$

$$\begin{aligned} R^{0}_{AdaptiveCruiseControlTask} = {} & 0.0374 \\ & + \left\lceil \frac{0.0374}{20} \right\rceil \cdot 4.0000 \\ & + \left\lceil \frac{0.0374}{10} \right\rceil \cdot 0.0313 \\ & + \left\lceil \frac{0.0374}{10} \right\rceil \cdot 0.3067 \\ & + \left\lceil \frac{0.0374}{5} \right\rceil \cdot 0.0266 \\ & + \left\lceil \frac{0.0374}{5} \right\rceil \cdot 0.0603 \\ & = 4.4623 \end{aligned}$$

$$\begin{aligned} R^{1}_{AdaptiveCruiseControlTask} = {} & 0.0374 \\ & + \left\lceil \frac{4.4623}{20} \right\rceil \cdot 4.0000 \\ & + \left\lceil \frac{4.4623}{10} \right\rceil \cdot 0.0313 \\ & + \left\lceil \frac{4.4623}{10} \right\rceil \cdot 0.3067 \\ & + \left\lceil \frac{4.4623}{5} \right\rceil \cdot 0.0266 \\ & + \left\lceil \frac{4.4623}{5} \right\rceil \cdot 0.0603 \\ & = \mathbf{4.4623} = \mathbf{R^{0}_{AdaptiveCruiseControlTask}} \end{aligned}$$

SonarTask

$$R^{n+1}_{SonarTask} = C_{SonarTask} + \sum_{j=hp(SonarTask)} \left\lceil \frac{R^n_{SonarTask}}{T_j} \right\rceil C_j =$$

$$\begin{aligned}
R^{n+1}_{SonarTask} = {} & C_{SonarTask} \\
& + \left\lceil \frac{R^n_{SonarTask}}{T_{AEBTask}} \right\rceil \cdot C_{AEBTask} \\
& + \left\lceil \frac{R^n_{SonarTask}}{T_{AdaptiveCruiseControlTask}} \right\rceil \cdot C_{AdaptiveCruiseControlTask} \\
& + \left\lceil \frac{R^n_{SonarTask}}{T_{LDESTask}} \right\rceil \cdot C_{LDESTask} \\
& + \left\lceil \frac{R^n_{SonarTask}}{T_{DataLoggingTask}} \right\rceil \cdot C_{DataLoggingTask} \\
& + \left\lceil \frac{R^n_{SonarTask}}{T_{ControllerTask}} \right\rceil \cdot C_{ControllerTask} \\
& + \left\lceil \frac{R^n_{SonarTask}}{T_{ReadRawDataTask}} \right\rceil \cdot C_{ReadRawDataTask} \\
& + \left\lceil \frac{R^n_{SonarTask}}{T_{CalculateVehicleSpeedTask}} \right\rceil \cdot C_{CalculateVehicleSpeedTask}
\end{aligned}$$

$$
\begin{aligned}
R^0_{SonarTask} &= 0.1852 \\
&+ \left\lceil \frac{0.1852}{25} \right\rceil \cdot 0.3437 \\
&+ \left\lceil \frac{0.1852}{25} \right\rceil \cdot 0.0374 \\
&+ \left\lceil \frac{0.1852}{20} \right\rceil \cdot 4.0000 \\
&+ \left\lceil \frac{0.1852}{10} \right\rceil \cdot 0.0313 \\
&+ \left\lceil \frac{0.1852}{10} \right\rceil \cdot 0.3067 \\
&+ \left\lceil \frac{0.1852}{5} \right\rceil \cdot 0.0266 \\
&+ \left\lceil \frac{0.1852}{5} \right\rceil \cdot 0.0603 \\
&= 4.9912
\end{aligned}
$$

$$
\begin{aligned}
R^1_{SonarTask} &= 0.1852 \\
&+ \left\lceil \frac{4.9912}{25} \right\rceil \cdot 0.3437 \\
&+ \left\lceil \frac{4.9912}{25} \right\rceil \cdot 0.0374 \\
&+ \left\lceil \frac{4.9912}{20} \right\rceil \cdot 4.0000 \\
&+ \left\lceil \frac{4.9912}{10} \right\rceil \cdot 0.0313 \\
&+ \left\lceil \frac{4.9912}{10} \right\rceil \cdot 0.3067 \\
&+ \left\lceil \frac{4.9912}{5} \right\rceil \cdot 0.0266 \\
&+ \left\lceil \frac{4.9912}{5} \right\rceil \cdot 0.0603 \\
&= \mathbf{4.9912} = \mathbf{R^0_{SonarTask}}
\end{aligned}
$$

Bibliography

[1] Takashi Chikamasa. *ECRobot API*. http://lejos-osek.sourceforge.net/ecrobot_c_api.htm. Visited 28/11/2013. 15

[2] Takashi Chikamasa. *NXT GT*. http://lejos-osek.sourceforge.net/nxtgt.htm. Visited 24/9/2013. 18

[3] Takashi Chikamasa. *nxtOSEK*. http://lejos-osek.sourceforge.net/. Visited 29/11/2013. 27

[4] Takashi Chikamasa. *Rate Monotonic Scheduling*. http://lejos-osek.sourceforge.net/rms.htm. Visited 24/11/2013. 25

[5] Toyota Motor Corporation. *Lane Keeping Assist*. http://www.toyota-global.com/innovation/safety_technology/safety_technology/technology_file/active/lka.html. Visited 11/12/2013. 73

[6] Alexandre David, Lasse Jacobsen, Morten Jacobsen, Kenneth Yrke Jørgensen, Mikael H. Møller, and Jiří Srba. *TAPAAL 2.0: Integrated Development Environment for Timed-Arc Petri Nets*, 2012. 87, 92

[7] Editors of Publications International, Ltd. *Traction Control Explained*. http://auto.howstuffworks.com/28000-traction-control-explained.htm. Visited 11/9/2013. 51

[8] Bill Howard. *What is adaptive cruise control, and how does it work?* http://www.extremetech.com/extreme/157172-what-is-adaptive-cruise-control-and-how-does-it-work. ExtremeTech. Visited 8/10/2013. 81

[9] Gabriele Manduchi Ivan Cibrario Bertolotti. *Real-Time Embedded Systems*. CRC Press, 2012. ISBN: 978-1-4398-4161-7. 88, 89

[10] Mike Karlesky, Mark VanderVoord, and Greg Williams. *Unity Intro*. http://throwtheswitch.org/white-papers/unity-intro.html, 2012. 35

[11] LEGO. *NXT User Guide*. http://lego.com/r/education/-/media/lego%20education/home/downloads/user%20guides/global/mindstorms/9797_lme_userguide_us_low.pdf. Visited 23/9/2013. 15

[12] LEGO. *RCX Legacy Sensor Blocks for NXT Software Version 2.0*. http://cache.lego.com/r/education/-/media/lego%20education/home/downloads/software/rcx%20legacy%20sensor%20blocks/macen.zip. Visited 28/11/2013. 17

[13] leJOS developer team. *leJOS, Java for Lego Mindstorms*. http://www.lejos.org. Visited 13/12/2013. 26

[14] leJOS developer team. *leJOS NXJ API documentation.* http://www.lejos.org/nxt/nxj/api/java/util/Timer.html. Visited 13/12/2013. 27

[15] EURO NCAP. *Autonomous Emergency Braking.* http://www.euroncap.com/rewards/technologies/brake.aspx. Visited 17/9/2013. 59

[16] Karim Nice. *How Anti-Lock Brakes Work.* http://auto.howstuffworks.com/auto-parts/brakes/brake-types/anti-lock-brake1.htm. HowStuffWorks. Visited 11/9/2013. 43

[17] Karim Nice. *How Cruise Control Systems Work.* http://auto.howstuffworks.com/cruise-control.htm. HowStuffWorks. Visited 17/9/2013. 81

[18] Royal Automobile Club of Victoria. *Effictiveness of ABS and Vehicle Stability Control Systems.* http://www.monash.edu.au/miri/research/reports/other/racv-abs-braking-system-effectiveness.pdf. Visited 11/9/2013. 43

[19] International Bureau of Weights and Measures. *The International System of Units (SI).* http://www.bipm.org/utils/common/pdf/si_brochure_8_en.pdf, 2006. Visited 28/11/2013. 41

[20] World Health Organization. *Road traffic injuries.* http://www.who.int/mediacentre/factsheets/fs358/en/. Visited 19/9/2013. 11

[21] OSEK/VDX. *Operating System.* http://portal.osek-vdx.org/files/pdf/specs/os223.pdf. Visited 18/12/2013. 28, 31

[22] OSEK/VDX. *What is OSEK/VDX?* http://portal.osek-vdx.org/index.php?option=com_content&task=view&id=4&Itemid=5. Visited 29/11/2013. 27

[23] pbLua development team. *LEGO Is Just a Hobby, Right?).* http://www.lejos.org. Visited 13/12/2013. 26

[24] SW502E12. *Vehicle safety systems implemented on a LEGO car*, 2012. 46

[25] WABCO. *Anti-Lock Braking System (ABS) and Anti-Slip Regulation (ASR).* http://inform.wabco-auto.com/intl/pdf/815/01/94/8150101943.pdf, 2011. Visited 4/11/2013. 44

[26] Wikipedia. *Automobile Safety.* http://en.wikipedia.org/wiki/Automobile_safety. Visited 11/9/2013. 11

[27] Wikipedia. *Lego Mindstorms NXT.* http://en.wikipedia.org/wiki/Lego_Mindstorms_NXT#nxtOSEK. Visited 13/12/2013. 26