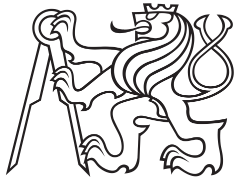
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Experimental slotcar-based platform for distributed control of vehicular platoons

Bc. Martin Lád

Supervisor: doc. Ing. Zdeněk Hurák Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Systems and Control
May 2017

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Control Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Lád Martin**

Study programme: Cybernetics and Robotics
Specialisation: Systems and Control

Title of Diploma Thesis: **Experimental slotcar-based platform for distributed control of vehicular platoons**

Guidelines:

Your goal is to develop an experimental slotcar-based platform for distributed control of a platoon of vehicles. Slotcars will be equipped with their own onboard computer processing the data measured by onboard sensors such as the distances to the predecessor and the follower in the platoon, the current through the motor winding, the velocity of the slotcar, the acceleration and the angular rate. The onboard computer will also be able to communicate wirelessly with other slotcars and the operator's computer. Base your project on the existing realization by Dan Martinec [1], but modify and extend the realization towards greater openness, modularity and reliability. Additional instructions:
1. Give specifications for the hardware part and coordinate its development. Prefer standard, open and modular solutions so that other developers can join in.
2. Design and implement the software both for the onboard computer(s) and for the operator's PC.
3. Make the project documentation and source code public through some popular developer's webs.
The functionality of the platform containing some 10 vehicles should be demonstrated by implementing a distributed LQG controller [2], a CACC [3] and possibly a distributed MPC [4].

Bibliography/Sources:

[1] D. Martinec. Distributed control of platoons of racing slot cars. Diploma thesis, ČVUT in Prague, 2012.
[2] Alam, A., et al. Experimental evaluation of decentralized cooperative cruise control for heavy-duty vehicle platooning. Control Eng. Practice, 38, pp. 11-25. 2015.
[3] Milanes, V., et al. Cooperative adaptive cruise control in real traffic situations. Intel. Trans. Sys., IEEE Trans. on, 15(1), pp. 296-305. 2014.
[4] Dunbar, W. B., Caveney, D. S. Distributed Receding Horizon Control of Vehicle Platoons: Stability and String Stability. IEEE Trans. on Aut. Control, 57(3), 620-633. 2012.

Diploma Thesis Supervisor: doc. Ing. Zdeněk Hurák, Ph.D.

Valid until the summer semester 2017/2018

L.S

prof. Ing. Michael Šebek, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 30, 2017

Acknowledgements

Many thanks to my supervisor Doc. Ing. Zdeněk Hurák Ph.D and co-supervisor Ing. Ivo Herman for guidance and support within this project. And finally, thanks to my family who endured this long process with me, always offering support and love.

Declaration

I hereby confirm that I wrote this diploma thesis on my own and that I listed all the used materials in the references.

In Prague, 26. May 2017

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 26. Května 2017

....

Martin Lád

Abstract

This thesis aims to create a platform for testing of distributed control algorithms for a platoon of vehicles. The platform is built on the basis of slot cars equipped with electronics needed to control their motors, to measure distance to their neighbors and other quantities such as velocity, acceleration, etc. The main computation unit is the Raspberry Pi Compute Module, which is complemented by a Wi-Fi card used for communication between vehicles. The thesis describes the used electronics, principles of measurements, design of software and GUI. In particular implementation of controllers from a perspective of a user, and the use of a library for the MATLAB Simulink. The library allows displaying data and changing reference signals in real-time, as well as simulation of the platoon. The results are presented by comparing simulations and real experiments. The experiments are bi-directional control using PI, PD controller and cooperative adaptive cruise control.

Keywords: Platform, Platooning, vehicles, Distributed control, Cooperative adaptive cruise control

Supervisor: doc. Ing. Zdeněk Hurák Ph.D.

Abstrakt

Tato práce má za cíl vytvořit platformu pro testování distribuovaných řídících algoritmů kolony vozidel. Platforma je postavena na základě autodráhových aut doplněných o elektroniku potřebnou k řízení motoru, měření vzdálenosti ke svým sousedům a dalších veličin, jako je rychlost, zrychlení apod. Hlavní výpočetní jednotkou je Raspberry Pi Compute Module, který je doplněn o Wi-Fi kartu sloužící ke komunikaci mezi vozidly. V práci je popsána použitá elektronika, principy měření veličin, návrh softwaru a GUI. Zejména pak, z uživatelského hlediska, implementace kontrolérů a použití knihovny pro MATLAB Simulink. Knihovna dovoluje zobrazování dat, změnu referenčních signálů v reálném čase a simulaci kolony. Výsledky jsou prezentovány porovnáním simulace a reálného experimentu. Provedené experimenty se týkají obousměrného řízení s použitím PI, PD regulátorů a kooperativního adaptivního tempomatu.

Klíčová slova: Platforma, Kolona, Vozidla, Distribuované řízení, Kooperativní adaptivní tempomat

Překlad názvu: Experimentální autodráhová platforma pro distribuované řízení kolon vozidel

Contents

1 Introduction 1
1.1 The thesis in context of history of the project . . . 3
1.2 Content . . . 4

2 Hardware 5
2.1 Motor driver and back-EMF measurement . . . 5
2.2 Velocity measurement . . . 8
2.2.1 IRC . . . 9
2.2.2 Velocity from Back-EMF . . . 10
2.3 Distance measurement . . . 11
2.4 Super capacitor . . . 12
2.5 Accelerometer, gyroscope and magnetometer . . . 13

3 Software 15
3.1 Slotcar File System . . . 15
3.2 The STM Firmware . . . 16
3.2.1 Behaviour description . . . 16
3.2.2 I2c sensors . . . 18
3.2.3 Communication SPI interface 18
3.3 Why JAVA? . . . 19
3.4 Raspberry Pi Slotcar Application 19
3.4.1 Program description . . . 20
3.4.2 Slotcar states . . . 21
3.5 Graphical User Interface running on PC . . . 22
3.5.1 Basic overview . . . 23
3.6 Simulink Model . . . 24
3.7 Slotcar Controllers . . . 25

4 Beginning with the platform 29
4.1 Clean installation . . . 29
4.2 Connecting to WiFi . . . 30
4.3 Connecting to CM . . . 31
4.4 Eclipse IDE . . . 31
4.5 Flashing STM . . . 32
4.5.1 On Windows by a STM32 DISCOVERY BOARD . . . 32
4.5.2 From CM . . . 32
4.6 Linux command line commands 33

5 Model 35
5.1 Linearization . . . 36

6 Control 41
6.1 Velocity control . . . 41
6.2 Distance control . . . 41
6.2.1 Bidirectional control . . . 42
6.2.2 Cooperative adaptive cruise control . . . 44

7 Conclusion 49

Bibliography 51

A The contents of the enclosed CD 53

Chapter 1

Introduction

Distributed control of platoons of vehicles has been an active research topic for a few decades. Recently, the topic got more popular even in public thanks to driver-less cars. The evolution can be track from cruise control (CC) through adaptive cruise control (ACC) to cooperative ACC (CACC). The cooperation is that a car shares some information with others using wireless communication. An implementation and an experimental evaluation of a CACC is one part of this thesis. CACC is a proved concept and it was tested experimentally, for example, in [8] or in [9]. Other examples also tested withing this work are *predecessor following* [10], *symmetric bidirectional control* [1].

The motivations for creating the platform are following. Having a platoon of real vehicles can be expensive, and even more expensive when an accident happens. It also requires a lot of space. Our platform is small, affordable and collisions are just moments for laughter. The platform (see an example in Fig. 1.1) has been presented at school "open doors" events, at the Gaudeamus, and at the Long Night of Museums. From our experience, it is very attractive; we had many interested visitors, and children.

The platform, as seen in Fig. 1.2 and Fig. 1.3, is based on racing slot cars 1:32 produced by *Carrera* used commonly as toys by children and adults. Originally, these slot cars had no onboard controller. They are also free of any sensor or communication interface. Their velocity is controlled remotely by a human player varying the voltage on the conducting strips on the track. The slot cars have been upgraded significantly by equipping them with custom-made electronics including powerful microprocessors, sensors, and a communication interface. The main computing unit is the popular the Raspberry Pi Compute Module, which runs Debian Linux. All these parts still fit within the small car, and the appearance is (almost) unchanged. A part of the platform is a user PC running GUI and MATLAB Simulink.

The main features of the platform.

- Small size: each car is 13 cm in length and 5.5 cm in height. The simplest required track for experiments is a circle with diameter of 1 m; it easily fits on an office desk.

Figure 1.1: An example of a fraction of our platoon of vehicles.

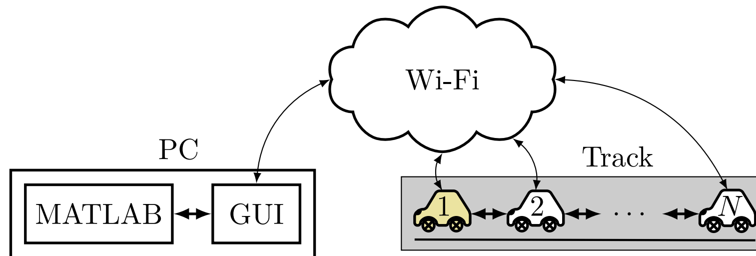


Figure 1.2: A visualization of the platform.

- Low cost: the total cost of each autonomous car is approximately 300 €, including the bare slot car itself. A basic track costs about 70 €.
- Durability: since all the electronics is located inside the plastic cover of the slot car, it can easily withstand almost any crash between cars.
- Ease of controllers design: a controller is just a plugin for the main application in the car. Hence, it can be easily developed, deployed and run. Moreover, source codes for the most commonly used controllers (PI, PD, ...) are already provided.
- Graphical application for the platoon setup: a Java application is provided for the platoon management. It allows a user to upload controllers to the cars, select a controller, and set its parameters, as well as update the firmware of cars remotely.
- Matlab interface: the data measured by each car are communicated among other cars, as well as to GUI. Matlab can be used not only to

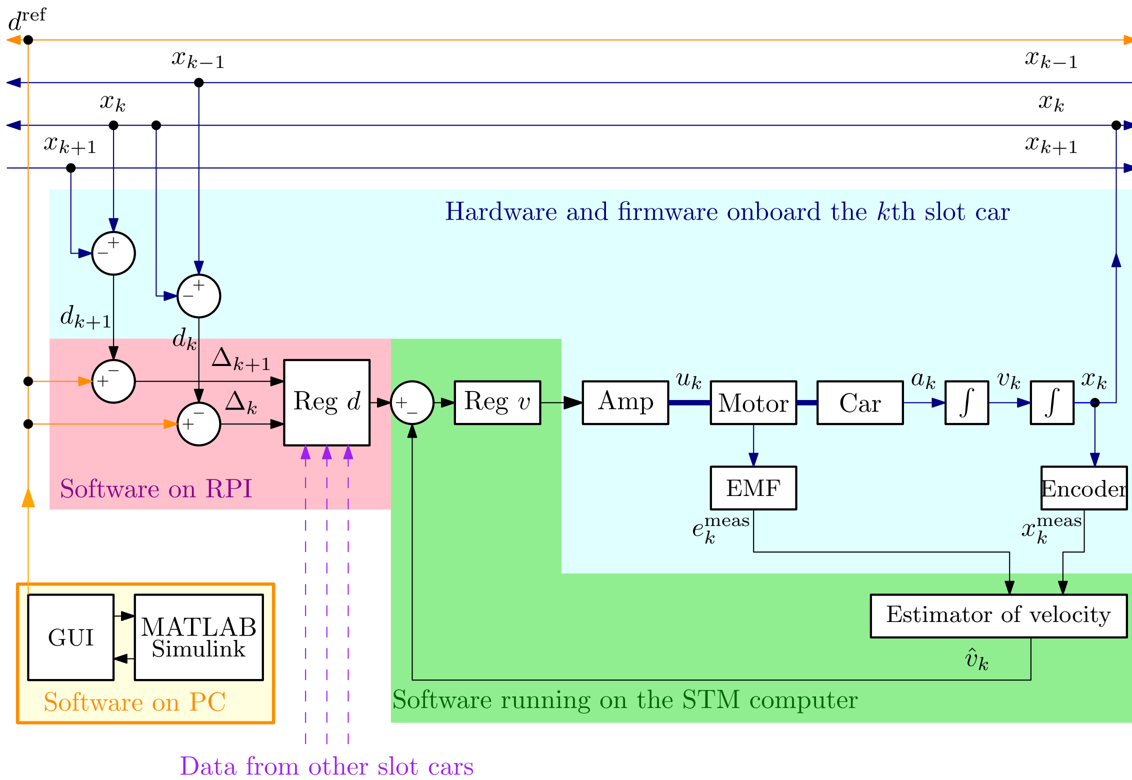


Figure 1.3: A block diagram showing the k-th car in the context of the platform.

visualize and analyze the data, it can also be used to change reference values of the platoon, such as the target velocity of leading vehicle.

The choice of Raspberry Pi Compute Module. From the hardware point of view, the Raspberry Pi Compute Module (CM) was chosen because of its small size, performance and huge community of developers. It became sort-of a standard in embedded technology.

CM runs a operating system(OS)—a Linux distribution called *Raspbian* (based on the popular Debian distribution, as the name reflects). Having an operating system allows a user to use many convenient services such as SSH and SCP for easy maintenance, debugging, and file management. While having already implemented drivers for Wi-Fi, SPI, and other standard peripherals, a developer can only focus on a application itself.

Although we do not use a real-time version of Linux, the OS is capable of running the control tasks with a sampling period of 30 ms.

1.1 The thesis in context of history of the project

A previous version of the experimental platform has already been reported in [7]. An overview of the current version of the platform has also been published [5]. This thesis is a logical follower, where the platform is described in more detail and should serve as a manual for future contributors. It is

also a summary of what the platform is capable of. The platform is still in development; however, it is already able to serve its purpose. Some examples of control principles were successfully tested; see Chapter 6.

The platform is being developed within numerous student projects following the spirit of *open source* and *open hardware*. The code and the schematics are publicly available through a git repository `https://gitlab.fel.cvut.cz/SlotcarPlatooning`. Noticeable contributors or previous work [1] are Jan Moravec, who designed distance sensors and proposed principle of measuring the distance; a revision of the code has been done recently. A group, consisting of me, Filip Svoboda and Filip Richter, designed electronics of the cars. The principle of speed measurement was presented in [4]. Other contributors are listed in our paper. It is not a short-term project for one person for one semester. The recent work has been focused on bug-fixing, cleaning of the code, and experiments. The project is also published at [6]. This is a popularization website allowing you to create a blog about a project. The website is focused on technical, mainly home-made, projects. The community consists of engineers and engineering enthusiasts.

Git repositories description:

slotcar-sw Raspberry Pi application and GUI, both written in Java.

slotcar-fw Firmware for microcontroller STM32F401, written in C.

slotcar-fs Raspberry Pi file system and OS related things.

slotcar-hw Hardware schematics, drawn in Eagle.

slotcar-ctrls Distance controllers, written in Java.

slotcar-doc Papers, documentation, and datasheets.

1.2 Content

This thesis is structured as follows.
Chapter 2 Describes hardware of a single car and the principles of measurements.
Chapter 3 Describes the software running on Raspberry Pi, firmware running on a microcomputer, GUI, and Slotcar MATLAB Simulink library. The microcomputer is responsible for measurements and control of velocity of the car.
Chapter 4 Shows how to work with the platform from a developer's perspective, and how to maintain the platform.
Chapter 5 Describes the mathematical model of the car and show some experiments, that prove correctness.
Chapter 6 Presents some control algorithms; the simulation and the real experiment, and compares it.
Chapter 7 Concludes the results of the work.

[1] Before the git repository was created.

Chapter 2
Hardware

In this chapter, we describe hardware solution and electronics. See Fig. 2.2.

Mechanics. The drivetrain is very simple: a brushed DC motor with a permanent magnet drives the rear axle through a gear train (no differential) with 3:1 ratio. The car drives on the track in a guiding slot, which allows the car to move only forward or backward. The car takes electricity from the pair of powered metal strips placed along the guiding slot. Since the side motion is severely restricted, the platform is only suitable for experiments related to longitudinal dynamics and control.

Electronics. It consists of two boards: *top* and *bottom*. Additionally, there are two distance sensors and one infrared encoder (IRC) used for speed measurement. All schematics of printed circuit board (PCB) are held in the slotcar-hw repository. For design, we used the software named Eagle which is widely used and very user-friendly.

Top board. On the *top* board, there is a Raspberry Pi Compute Module (RPI), which is the main computation unit. Next, there is a USB connector in which a Wi-Fi dongle is plugged in. A Wi-Fi network is used as the main communication interface among all the cars and GUI. We also equipped the cars with the Zig-Bee communication interface [1].

Bottom board containing low-level electronics. Namely power supply chain (rail voltage U_r; 5 V; 3.3 V; 1.8 V), a motor driver, the microcontroller STM32F401 (STM) [12], a sensor for acceleration and rotation, a magnetometer and a super capacitor. IRC and the distance sensors are connected to the *bottom* board by cables.

2.1 Motor driver and back-EMF measurement

The motor is powered from the rail voltage and is driven by STM through the H-bridge motor driver DRV8816 [14]. The whole situation is depicted in Fig. 2.3.

[1] It is not currently used. We prefer Wi-Fi because it is more common communication interface.

Figure 2.1: The hardware of the car; the bottom board, the top board, and the car.

H-Bridge. DRV8816 is not an actual motor driver, but an extended full H-bridge. Among standard functions of a standard H-bridge, it provides a dead-time check and a coast mode. In the coast mode, output pins are set to a high-impedance state, which is essential for back-EMF measurements.

Lock Anti-Phase mode. The motor is driven by two signals A and B. Each signal drives one-half of the H-bridge (the high level = the top transistor is on, the bottom is off and vice versa). There are several ways of driving a motor. Some are explained in [13]. We chose the *lock anti-phase* because, among others, its behavior is linear. In other words, the shape of acceleration of the motor is symmetrical to the shape of braking.

This mode is periodically switching between a *forward* and *reverse* state with a high frequency (tens of kHz); see Fig. 2.4. In average, the proportion

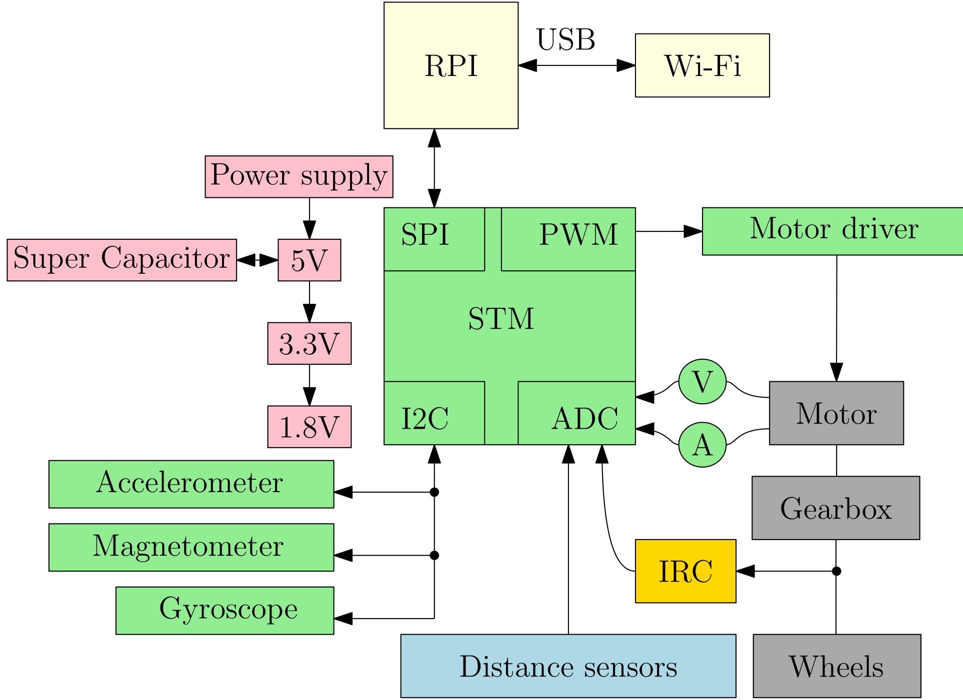


Figure 2.2: Schematic diagram of the hardware.

of staying in one of these states determines whether the motor is rotating forward or backward and how fast. When the proportion is 1:1, the motor stops.

PWM. We control the motor by generating a complementary Pulse Width Modulation (PWM) signal, which means, that one side of the H-bridge is controlled by one PWM signal and the other side is controlled by its inversion. The duty cycle of PWM

$$d_\mathrm{c} = \frac{t_\mathrm{on}}{t_\mathrm{on} + t_\mathrm{off}},$$

where t_on resp. t_off is time of high resp. low state of the signal.

There is a convention problem. In the *lock anti-phase* mode, when $d_\mathrm{c} = 50\%$, the motor stops. The average input voltage on the motor is defined as

$$u_\mathrm{avg} = U_\mathrm{r}\frac{t_\mathrm{on} - t_\mathrm{off}}{t_\mathrm{on} + t_\mathrm{off}} = U_\mathrm{r}D_\mathrm{c} = U_\mathrm{r}(2d_\mathrm{c} - 1),$$

where U_r is the rail voltage and D_c is the duty cycle of our linear model. The conversion formula is

$$d_\mathrm{c} = \frac{D_\mathrm{c} + 1}{2}.$$

A timer which generates PWM is set to a center-align mode (Fig. 2.5). In this mode, the timer counts repeatedly up to a maximum value and then down to zero. It toggles a output pin on a comparison with a preset level. The preset level determines the duty cycle and the maximum value determines the frequency of PWM. The timer generates an interrupt once it reaches zero

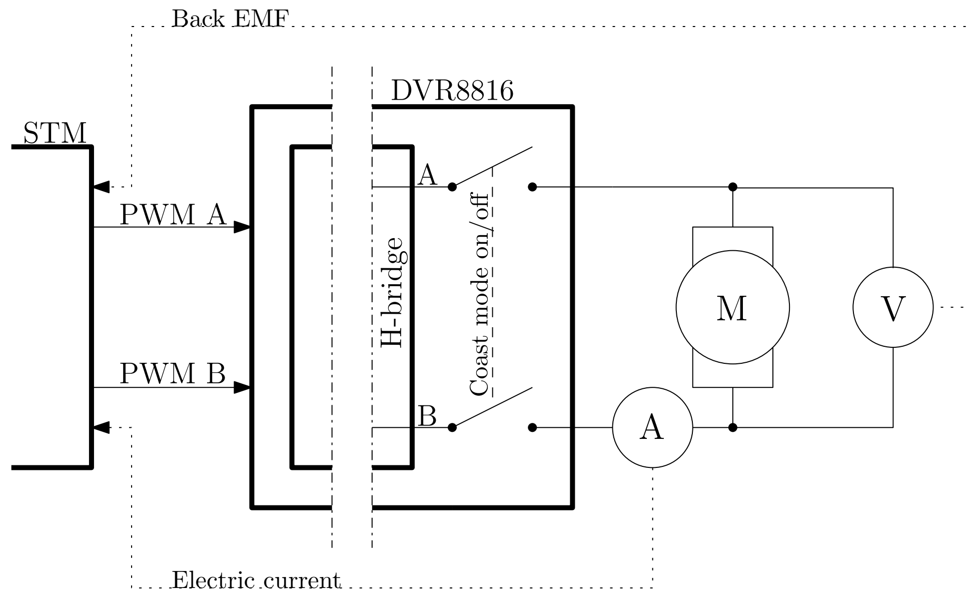


Figure 2.3: Schematic diagram of the interconnection of STM, the driver and the motor.

or the maximum value. Therefore, interrupts appear twice in one period of PWM.

Back-EMF (Back-induced ElectroMotive Force) measurement. We switch the motor driver to the coast mode, and the motor is floating unconnected to the driver. When the electric current decays in the motor winding, the motor generates only the back-EMF, and then we measure it; see Fig. 2.6. The time window, in which the feasible range of electric current is able to decay, is determined experimentally to 300 µs. After that sequence, we switch the motor back to the normal operation. The voltage on the motor is measured by using a differential amplifier and by using the integrated analog-to-digital converter (ADC) on STM.

Electric current measurement. We also measure the electric current flowing through the motor by using an another differential amplifier which measures a voltage drop on a resistor 0.2 Ω. The measurement is synchronized with PWM, and we measure it only once per period at the longer state of PWM, because we do not want to measure any noise factors connected with PWM level changes. Mainly, it is important when the duty cycle is very low or very high.

2.2 Velocity measurement

The translation velocity is measured by using two types of sensors. The considered velocity range is between $-2\,\mathrm{m\,s^{-1}}$ and $2\,\mathrm{m\,s^{-1}}$. At higher velocities, cars can get off the track in turns. The first sensor is a standard (albeit

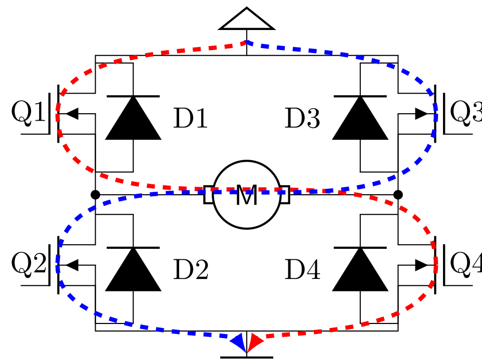


Figure 2.4: In the picture we can see an H-bridge and flows of electric current. In the *forward* resp. *reverse* state, the transistors (Q1, Q4) resp. (Q2, Q3) are open, and (Q2, Q3) resp. (Q1, Q4) are closed. The red resp. blue line shows the flow of the electrical current through the motor.

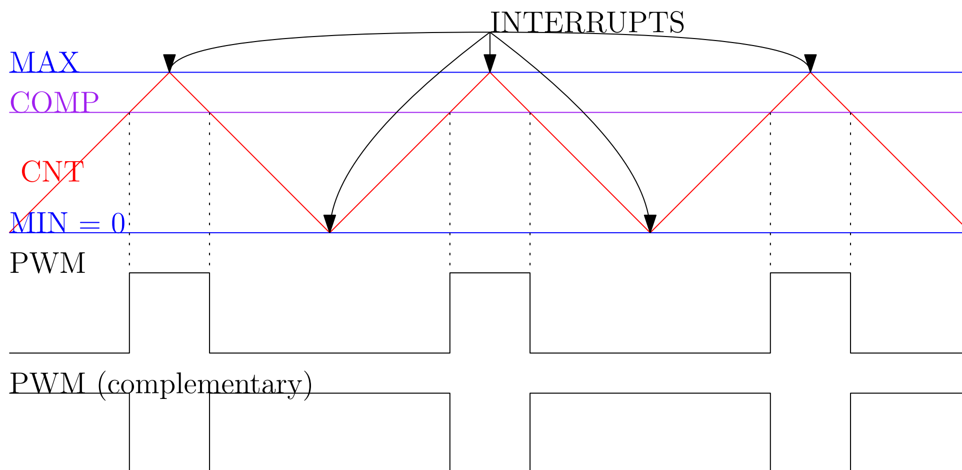


Figure 2.5: PWM center-align mode. The timer is counting (CNT) up/down from MIN to MAX, toggles PWM on equal comparison with COMP value.

home-made) incremental rotary encoder (IRC). The other one is based on measuring the back-EMF. Since we have two measurements, each with a different precision and accuracy, they are be fused using an estimator, namely Kalman filter (see [4]). This filter also provides an estimate of the friction force affecting movement of cars.

2.2.1 IRC

A binary (black and white) disk is attached to the rear axle. The infra-red emitter/receiver QRE1113 (see [2]) is pointed to the disk, and by measuring the intensity of reflected light, is determined the color in front of the sensor. There are two standard ways how to determine the speed of rotation. First, by measuring the frequency of changes of the color in front of the sensor. Second, by measuring the time between these changes.

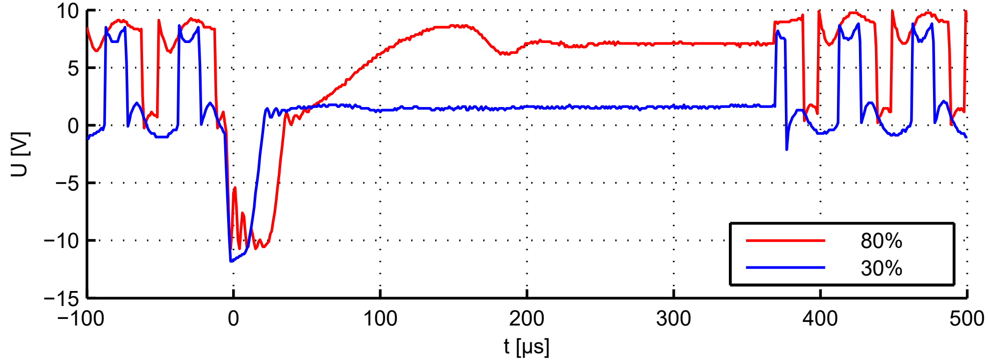


Figure 2.6: Two measurements of back-EMF for two different duty cycles.

Measurement of the time between changes. The reason why we use this method is that we have only six (three black and three white) stripes which means six changes per one turn. Moreover, the sensor is situated after the gear train from the perspective of the motor. Therefore the angular speed is divided and becomes too slow for measuring just the frequency, even at higher velocities.

The speed equation in a time of the color change k is following

$$v(k) = \frac{4\pi r}{A}\frac{1}{\tau(k)} \ , \tag{2.1}$$

where r is the radius of the wheel, A is the number of color stripes and τ is the time between the recent and the last color change. In our case, we measure the time by a timer

$$\tau(k) = N(k)\frac{1}{f_{\mathrm{vf}}} \tag{2.2}$$

where f_{vf} is frequency of the timer, and N is number of ticks of the timer, which is reset after each change of the color. In fact, the sensor can measure only the speed of rotation and not its direction. The information about direction is borrowed from the back-EMF measurement.

2.2.2 Velocity from Back-EMF

The back-EMF, denoted u_{bemf}, is proportional to the motor angular speed as

$$u_{\mathrm{bemf}}(t) = k\omega(t), \tag{2.3}$$

where k is the so-called back-EMF constant (in SI units identical to the motor torque constant). The equation for obtaining the speed is following

$$v_{\mathrm{e}}(t) = \frac{rn}{k}\,u_{\mathrm{bemf}}(t) \ , \tag{2.4}$$

where n is gear ratio and r is the radius of wheel of the car. Unlike IRC, the back-EMF measurement gives the direction of rotation. The con of this method is that measuring back-emf is very complicated. However, once you have the value, it is directly proportional to the velocity.

2.3 Distance measurement

We could not find any commercially available sensor which would satisfy the following requirements:

- small size, such that it fits into the car,
- wide field of view (important when going through turnings),
- no interference between the front and rear distance sensor.

It was possible to satisfy the first two requirements; however, the third one seemed like a big obstacle. That is why we designed own sensor. It is based on an infrared (IR) LED and a phototransistor. The diode emits square pulses with a given frequency, and the phototransistor receives the signal. To get rid of disturbances such as the sunshine or the room lighting, the demodulation of the received signal exploits the correlation (synchronous detection)—the frequency of the transmitted signal is known. There are distinct frequencies for both front and the rear distance measurements: for the front it is $f_\mathrm{f} = 1111\,\mathrm{Hz}$ and for rear it is $f_\mathrm{r} = 1666\,\mathrm{Hz}$. Both are chosen to be sufficiently well separated and also not being the multiplier (higher harmonics) of $100\,\mathrm{Hz}$, which is a frequency of the fluorescent tubes used in the lab.

We do not use reflections. Reflection-based sensors are limited by the angle of reflections. In our case, each car receives the emitted signal by its neighbors (front and rear). Since we are using wide angle IR LEDs and phototransistors, it gives a good precision even in turnings. The sketch of the distance measurement is in Fig. 2.7.

Measurement principle. The car with index $i-1$ emits the signal from its rear LED with the frequency f_f. The car i detects this signal at its front transistor and demodulates it. The distance d is calculated from the strength s of the demodulated signal by

$$d(t) = \sqrt{\frac{c}{s(t)}}, \tag{2.5}$$

where c is a constant derived from calibration. The range of the sensor is approximately from 5cm to 50cm. Similarly, the car $i-1$ calculates its distance to the car i. Compared to the distance measurement schemes based on reflections (some commercially available hobby-grade distance sensors rely on it), the proposed scheme achieves four times stronger signal (the distance for the light to travel is halved).

As any other sensors, these also have some disadvantages.

1. Sensors require calibration procedure. Currently, we calibrate the constant c by telling the sensor what is the real distance and by comparing it to measurement and c is readjusted.

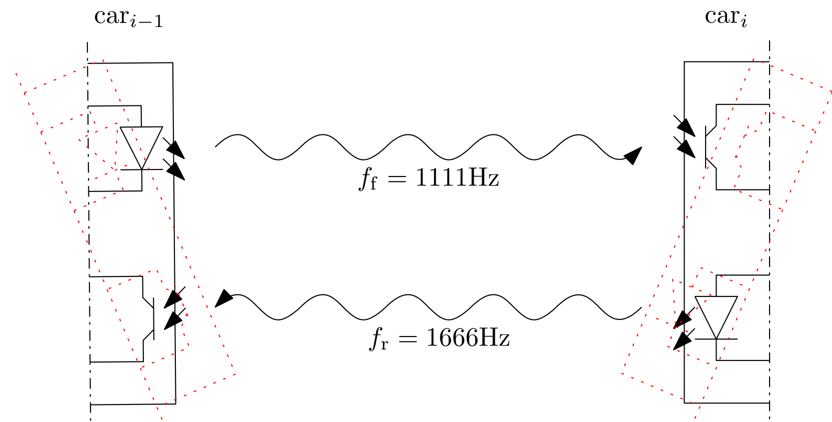


Figure 2.7: Schematic of distance measurement between two cars. Two neighbors car$_{i-1}$ and car$_i$ exchange IR signals (with different frequencies). From intensities of the signals, the cars calculate the distance. In turnings (red color), the sensors are not in parallel position; each signal has slightly different time of fly. Therefore, the intensities differ and also the measured distances are different.

2. It is necessary to have a car ahead, transmitting the modulated signal. However, this does not cause any problems in our setting, since there is always a platoon leader. The leader drives independently, following the desired speed profile. Hence, the leader itself does not need any front distance measurement.

3. In turnings, bumpers of two neighbors are not parallel (see Fig. 2.7); therefore, the distances that measure car i and $i-1$ are different. It cause problems for bidirectional control. We avoided this by using the communication between two neighbors. By sharing the information, the predecessor takes the front distance of the successor car as its rear distance. In future work, the car should combine the information of its measurement and of measurement of its successor.

2.4 Super capacitor

The main problem with the RPI is its power consumption. It is necessary to have a backup power source because the car, for example during crashes, may lose the connection with the power line. The loss of power leads to shutdown of RPI, which cause a loss of data on the one hand and an additional time needed for reboot on the other hand. The backup is something that saves time, since the boot time is about a minute long. Of course, repairing harmed OS image requires much more.

Therefore, we added a supercapacitor (1 F; 5 V) which serves as a power backup for ≤ 5 V branches. It is able to backup the circuit for approximately 5 s, which seems to be enough for emergency situations, such as crashes. Note that since the car has a significant power consumption, a regular capacitor is not sufficient for this task. Adding a battery is also not sufficient, because it

needs a regular maintenance.

2.5 Accelerometer, gyroscope and magnetometer

We call this group of sensors *I^2C sensors*, because they are all connected to the common I^2C bus shared between both RPI and STM.

The accelerometer and gyroscope are one-package sensor named LSM330DLC (see [11]). The magnetometer is MAG3110 (see [3]). All of them are 3D.

The supported ranges are: for accelerometer ±2 g, ±4 g, ±8 g, ±16 g and for gyroscope it is ±250 deg/s, ±500 deg/$s and$±2000 deg/s. The magnetometer has fixed range and it is ±1.55 µT.

Chapter 3
Software

The software is divided into three parts, corresponding to three processors: the *Firmware* running on STM, the *Car Application* running on RPI, and *Graphical User Interface* (GUI) running on PC. In this section, we describe the environment of RPI, the Slotcar MATLAB library and the architecture of the three software parts.

3.1 Slotcar File System

What we call the Slotcar File System are just files (configurations and scripts) categorized by their position in the Linux directory tree. These files edit the Raspbian on top of its standard configuration. We will not describe all the files in detail. Instead, we denote some main features provided by this configuration.

Read-only file system. We divided the flash memory of RPI into two partitions the root (*/*) and */slotcar*. The root is for OS and is mounted as read-only, and */slotcar* is for the car's application which is mounted as read-write. Some directories of the file system are mounted to RAM, for example: */tmp*, */var/log*. This precaution is a standard safety mechanism dealing with some unpredictable or unwanted events like loss of power while writing to the flash memory and others.

Time synchronization. We use the Network Time Protocol (ntp) which is a Linux service providing a precise time synchronization againts a server. The server is a car designated by a user.

Networking. We are using a Wi-Fi network in Ad-hoc mode, which means that there is no hot-spot and cars are communicating directly with each other. We are using IP range *192.168.1.0/24*, and each car has a fixed IP address which is determined based on its number (a logic address). Cars are numbered from 1 to N and therefore their address is *192.168.1.X*, where $X = 100 + n$. For example car 1 has IP *192.168.1.1*. Default logical address for a car is 0.

Slotcar Linux service. Using services in Linux is an easy way how to run an process in background of the OS. It provides functions like stop, start, restart the process, for example on error and more functions. We created such a service for our application, which is very convenient and standardized way how run a long term application in Linux.

Slotcar script for maintenance. It allows us to start/stop the car application, to show logs, to set the logical address of the car, to set the NTP server, to load program to STM, etc. As well it can make the upgrade of the whole file system, by downloading it from Git *slotcar-fs*. More information can be found in Chapter 4.

STM program loader. We did not find any suitable solution for flashing the STM memory from a Linux via SPI. Therefore, we wrote our own C application, using WiringPi library for programming the STM processor. It uses the processor's SPI built-in bootloader. However, it does not cover all functionality yet, for example (un)locking sections. Therefore, for the first programming it is still needed to used a official software to unlock the sections.

3.2 The STM Firmware

The code is written in C language, and it has two purposes:

1. Being a sensor. It measures quantities with ADC, reads out data from other sensors and filters them.
2. Being a controller. It controls the velocity of the car.

It measures the motor back-EMF, rail voltage, super capacitor voltage, motor electric current, voltage from distance sensors and pulses from IRC. Then it processes (demodulation, calculations, filtration) the data to get real values or other quantities derived from them. It reads out 3D acceleration, 3D rotations and 3D magnetic field from external sensors. Based on velocity measurements, it controls the duty cycle of PWM to track a reference velocity.

We use two interrupts and we try to do in main routine as much work as possible. However, it is in balance with keeping the code as simple as possible.

3.2.1 Behaviour description

The main function starts with initialization of all components and then jumping into a infinite loop. The infinite loop has two parts, synchronous and asynchronous. Jobs that need continuous execution are run in the asynchronous part. In our case, it is polling of sensors. Jobs that need to be called periodically are called in the synchronous part, for example, speed control. Other jobs, that need precise high-frequency periods, are done in interrupts, generated from ADC and SPI. See 3.1.

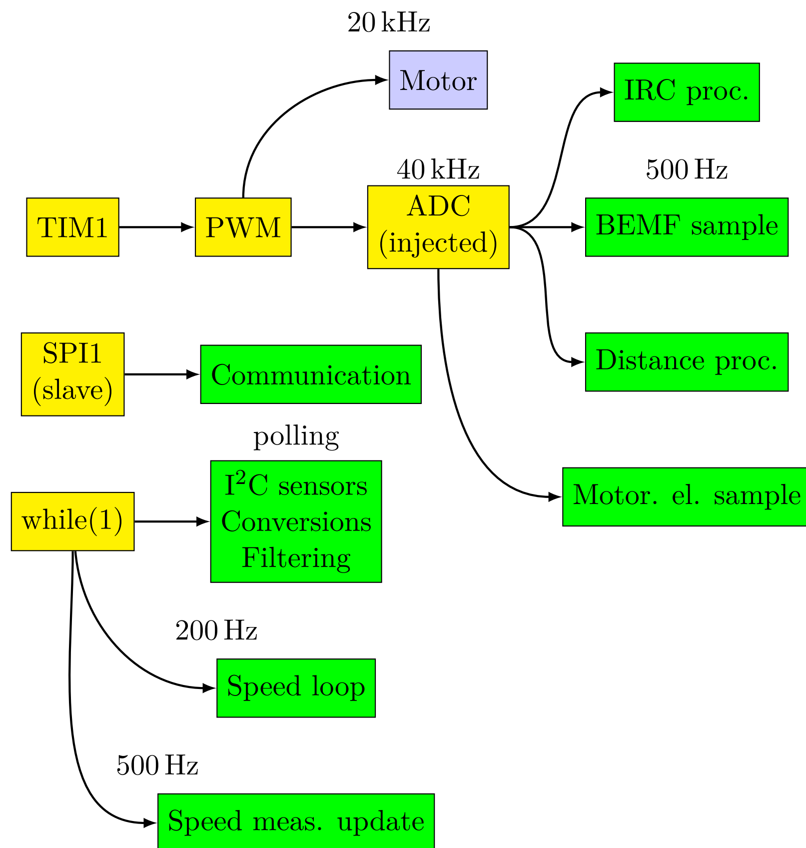


Figure 3.1: STM interrupts and processing measurements

The asynchronous part of the infinite loop. It is running continuously and we use it for:

1. Polling I2C sensors and measurements conversions.
2. Back EMF sample conversion and filtering.
3. Motor electrical current sample conversion and filtering.
4. IRC time conversion

Note that the samples are equivalent voltages given by measurement circuits and measured by ADC. Therefore, they need to be converted to real values. Conversions and filtering are always done only when a new sample is available.

The synchronized part of the infinite loop. It runs on 1 ms period, from which all slower periodical events are derived such as speed control loop (200 Hz) and speed measurement update.

The SPI interrupt. It come from SPI when a byte is received. Since STM is a slave on SPI, its response needs to be fast and flawless. Therefore, it has the highest priority.

The ADC interrupt. It comes from ADC, when injected [1] measurements are done. It is periodical with frequency 40kHz. In this interrupt, we measure motor electric current and back-EMF, we also do the distance signal demodulation and the IRC measurement procedure. Other ADC measurements are done in regular mode, which means that they are continuously read out.

IRC. We are measuring the time τ in the ADC interrupt by counting pulses. In the interrupt, we also check the color. The counter is reset when it changes otherwise it is incremented. Additionally, we add together the time of crossing black, and white together to increase precision. When a new change of color happens we replace the latest crossing time with the new one. The number of ticks per sequence (B-W-B or W-B-W) is in hundreds of ticks.

3.2.2 I2c sensors

Sensors are connected to STM via I^2C (100 kHz) bus. Polling method is used for initialization of a data transfer, which is then done automatically via DMA. On initialization of the data transfer, the status of I^2C is checked and when it is busy no action is taken, otherwise the data transfer for a next sensor is initiated and a data conversion for the previous sensor is called. Sensors return RAW data, therefore they must be converted by a conversion constant.

3.2.3 Communication SPI interface

The communication interface is using SPI bus, where STM acts as slave. STM provides access to its virtual memory of variable (the memory is defined in code as *MyVariables*) by using addresses. In order to read resp. write the data, the read (see Tab. 3.2) resp. write (see Tab. 3.1) frame is used. Follows the description of each data blocks in the frames.

Byte	0	1	2	3	4 ... N	5 + N	6 + N
Data	SYNC	N	ADDR	WRITE+AINC	DATA	CRC	RESP

Table 3.1: SPI communication, write frame

Byte	0	1	2	3	4	5...N	6 + N
Data	SYNC	N	ADDR	READ+AINC	RESP	DATA	CRC

Table 3.2: SPI communication, read frame

SYNC Start of the frame (0x16). Resets the communication state.

N The length of the bytes to be written/read.

ADDR The address of the cursor register.

[1] ADC has regular (continues) and injected (periodical) measurement.

WRITE/READ Write (0x01) / Read (0x02) frame

AINC Auto increment (0x04). If N is bigger than the size of one register, remaining data are stored in the following register, which means with every successful write/read the address is incremented.

DATA The data to be written(read) to(from) register. The meaning of the data depends on the register type.

CRC Longitudinal redundancy check. It is calculated from the frame. The first byte used in the calculation is the SYNC; the last is the last byte in the payload. The Slave calculates the CRC as well and stores the data only when both the CRCs match. The calculation is a simple XOR operation over all bytes.

RESP The response byte of the slave. If everything is OK, it returns RES_OK(0x01), otherwise, it returns RESP_ERR (0xFF)

SPI is set as duplex 8bit slave. No CRC control and software chip select (CS), where the physical CS used rather as an indication of a new communication. Each falling edge on CS is considered as a new data request. The logic behind is a state machine which is reset with CS.

3.3 Why JAVA?

Following sections are dedicated to GUI and RPI software. Both are written in Java. Now we put down several reasons why Java is an ideal programming language for our case.

1. Java Virtual Machine (JVM). It is a big advantage because it makes Java a multi-platform language. In our case, the car's SW is running on a Linux OS and we expect that GUI will usually run on a personal PC with the Windows. However, JVM makes both almost the same from the perspective of the code. Therefore, the car's SW and GUI are developed simultaneously in one project because they share a lot of code. The only difference is in their main functions. Nevertheless, JVM gives an advantage of remote debugging, and it takes care of memory management (memory leaks).

2. Native support of graphics. It makes creating graphical user interface easy because it does not need any third-party framework.

3. Java is a high-level language with a huge community and a lot of libraries.

3.4 Raspberry Pi Slotcar Application

The main purpose of this software is to run a *distance controller* and communicate with other cars and GUI. Generally, the *distance controller* receives

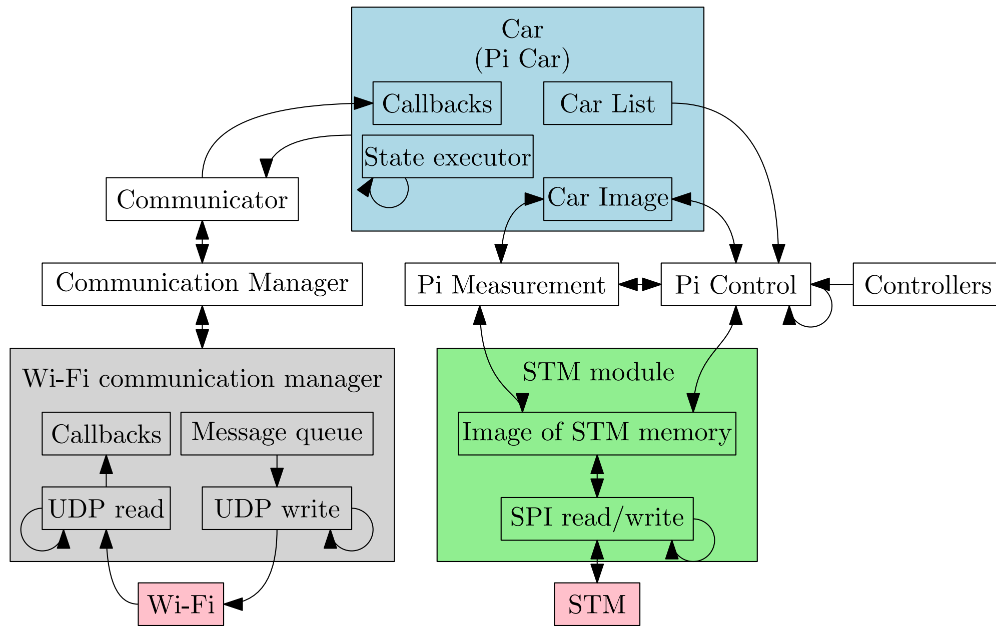


Figure 3.2: Simplified structure of the slotcar application.

the full state of the local and all other vehicles (obtained by the wireless communication). Based on these data, it calculates the reference (desired) speed for the speed controller in STM. Alternatively, it can directly set the duty cycle of PWM applied to the motor.

3.4.1 Program description

The application is multi-threaded. The structure of the program is depicted in Fig. 3.2.

Thread: Communication with STM. The STM output resp. input memory are periodically (5 ms) read out resp. written in all at once.

Thread: Wi-Fi communication. The communicator manager reads a UDP socket. It calls all registered callbacks once it receives a message. It is also responsible for sending messages. Each message to be send is stored in queue from which the manager reads them and sends them.

Thread: Control. It runs a *distance controller* selected via GUI. The controllers are loaded dynamically on every change of the selected controller (more in Section 3.7).

Thread: State executor. The main program behaves like a state-machine. It has several states, where each state has it initial, running and end function. These states also tells the user what the car is doing at the moment, for example state *ready* says that car is doing nothing; waiting for commands. On the other hand *positioning* says that car is running a positioning procedure.

3.4.2 Slotcar states

Positioning This procedure determines the position (index) of the car in the platoon. We take advantage of having distance sensor depended on the car on the other side. When the car in front of other car turns its rear sensor off, the other car cannot measure the distance. The procedure is as follows: the car who sees no one in front of it sends an echo to GUI and turns off its rear sensor. This step repeats until all cars respond to GUI. Then GUI sends the list of positions to each car. The procedure takes about ones of second.

Experiment This procedure starts communication between cars in which they send its state. We have implemented two principles. (1) The First is so called Token ring, where each car is waiting for its predecessor to communicate (the first waits for the last). They also have a backup timeout strategy when a message got lost. (2) The second principle is using default random access policy, and each car sends the messages with a constant period.

Following test showed that the second method is faster. Instead of dealing with communication on user space level, it is easier and faster to pass it to the driver.

```
RANDOM ACCESS
Period:
20.00 ± 1.02 (5.00, 35.00) ms
time of sending:
0.37  ± 1.85 (0.00, 21.00) ms
# Regular:  0/4745
# Timeout:  4745/4745
Cars communication time periods:
car2 = 23.2 ± 13.6 (2.0, 151.0) ms
car3 = 22.6 ± 13.3 (8.0, 56.0)  ms
car4 = 22.6 ± 12.8 (4.0, 50.0)  ms
car5 = 21.8 ± 12.6 (3.0, 110.0) ms

TOKEN RING
Period:
36.7 ± 18.7 (1.0 , 114.0) ms
time of sending:
1.94 ± 1.97 (0.00, 16.00) ms
# Regular:  3216/3392
# Timeout:  171/3392
Cars communication time periods:
car2 = 39.1 ± 22.3 (23.0, 59.0)  ms
car3 = 38.4 ± 23.9 (23.0, 96.0)  ms
car4 = 37.4 ± 20.8 (3.0 , 76.0)  ms
car5 = 39.5 ± 22.7 (3.0 , 127.0) ms
```

Control Starts/stops executing the selected controller in each car.

Calibration Automatic calibration of distance sensors on each car. It means that a car readjust the calibration constant c. There are two ways how to do it. First, a user rearranges the cars the way that a certain distance is between all of them. Then GUI sends the distance as a reference to each car, so it can re-adjust its measurement. Second option is to use the communication. The calibration is done only on every second car, which takes the reference distance from its neighbors measurements.

It is important that two neighbors measure same distances, rear and front, between them, for example for bidirectional control.

Logging Start/stops periodical car state logging. The logs are stored in CSV format.

3.5 Graphical User Interface running on PC

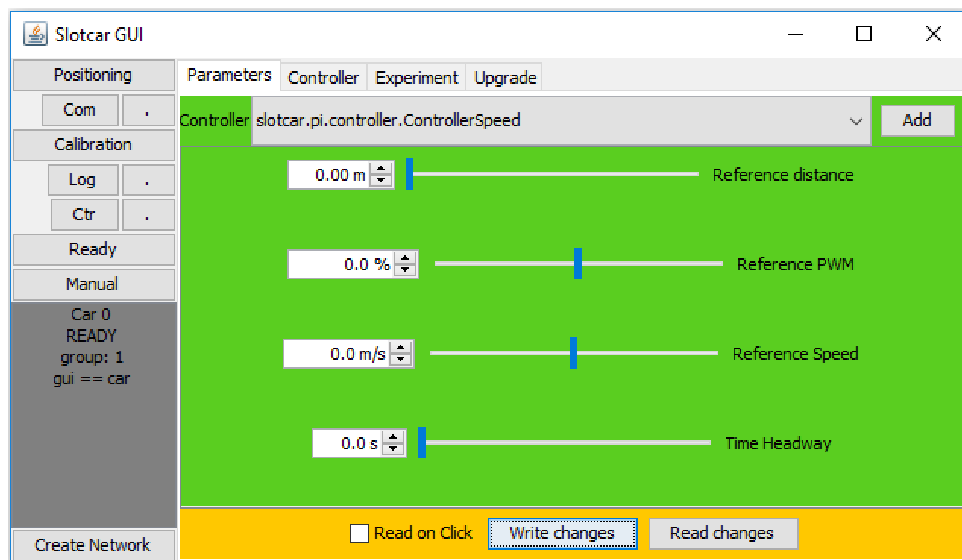


Figure 3.3: Screenshot of the GUI. On the left there is a list of connected cars.

The Graphical User Interface (GUI), as seen in 3.3, serves as a tool for maintaining the platoon. It allows us to select a controller, to set its parameters, to set the reference values for each car, to upload controllers to cars, as well as to start and to stop the experiment. On the top of that, the application allows us to upload the cars' firmware (using SCP), to reset and flash STM and a few more system operations. GUI is specified only for purposes of a user. For a developer, it is more convenient to use SSH access (more in chapter 3.4).

GUI acts as a server, to where the clients (cars) connect. The car connects to the server after the boot sequence or on a request from the server. After the car connects, the GUI updates the list of all connected cars and shares this

list with all cars. Then it asks the car for its settings–the control parameters (type of controller, desired distance, controller parameters). The user can modify the parameters and upload them back to a car.

3.5.1 Basic overview

GUI is divided into parts: (1) Control Buttons (left-top corner), under these (2) the List of Cars follows. (3) Operation Tabs.

Control Buttons. With these buttons, we can change the state of application, and initiate several actions. Start the *positioning* procedure on the platoon, start the global inter-car periodical *communication*, do a *calibration* of distance sensors, by manually setting cars to equal distances from each other and then telling them what the distance is. As well as start car's state logging, start control (the distance controller is executed), reset the state to *ready* (do nothing) and manual control (moving the car forward or backward by pressing keyboard's arrows up or down).

List of Cars. Provides not only information about the list of connected cars. The list is sorted by position of cars in the platoon (the *possitioning* must be executed in order to achieve correct positions) as well as the current state of the application in the car, group [2], and status of synchronization. Before you change settings to certain cars, you need to select them in the list.

Tabs. We have several tabs, which are assigned a group of jobs. (1) Parameters tab, in which we have can set five basic parameters of the car and the choice of the controller. The list of parameters is easily extensible by new parameters, see Listing 3.1. Controllers are loaded the same way the car loads it. The user generates a JAR file with controllers and adds it by clicking the *add* button [3]. In order to read/write changes you need to click read/write button. If there is an inconsistency in settings between dummy and shadow car, the parameters background is red.

(2) Controller tab, where you can change settings (constants) of controllers.

(3) Experiment tab servers for very basic plotting state data of connected cars. However, we stopped developing the functionality since MATLAB provide much better plotting tools.

(4) Upgrade Tab. You go there typically when you need to make some maintenance on the car. There are the most common actions that were needed while development. If the action is not there you can always open SSH tunnel. Provided functions are nothing else than just wrapped console commands.

```
CarParameter p = new CarParameter(
        "Reference Speed",  // Name of the parameter.
        0.0f,               // Default value.
        -1.5f,              // Min value.
        1.5f,               // Max value.
        0.1f,               // Minimal step between two values.
        "0.0 m/s"           // Displayed format.
        )
{
  @Override
  public void setter(CarImage car, float value) {
    car.getCarControl().setReferenceSpeed(value);
  }

  @Override
  public float getter(CarImage car) {
    return car.getCarControl().getReferenceSpeed();
  }
};
```

Listing 3.1: Defining new parameter in GUI just by defining getter and setter.

3.6 Simulink Model

Along with GUI we also provide the Slotcar MATLAB Simulink library (see Fig. 3.4). This library servers for plotting real-time cars' states as well as change reference signals, and for simulation of the platoon. The library is just an extension of GUI, from which it takes data (obtained by communication). In this section, we describe individual blocks of the library.

Platoon It is a thin client for GUI. The connection between GUI and the library is duplex (uses UDP on the local network). It sets references and reads states for each car. However, it cannot set the controller directly, that is what the user must do in GUI.

GUI is passively listening to the cars' communication, from which it takes the states of cars. The data are passed from GUI to Simulink on request. Also, Simulink is sending reference signals to GUI which passes them to cars. It does not start the communication of the cars automatically.

CarModel Car Model, which represents the model of a single car. Please take a look to Chapter 5 which describes the model.

Vel.loop Velocity loop, which wraps the car model to a speed loop with a speed controller.

CarDist The car distance block is a wrapper for the velocity loop block. It allows us to make a platoon by interconnecting more of these blocks together.

[2] All cars with the same setting belong to the same group

[3] It does not upload controllers to cars, you must do by the *upgrade* tab or by IDE

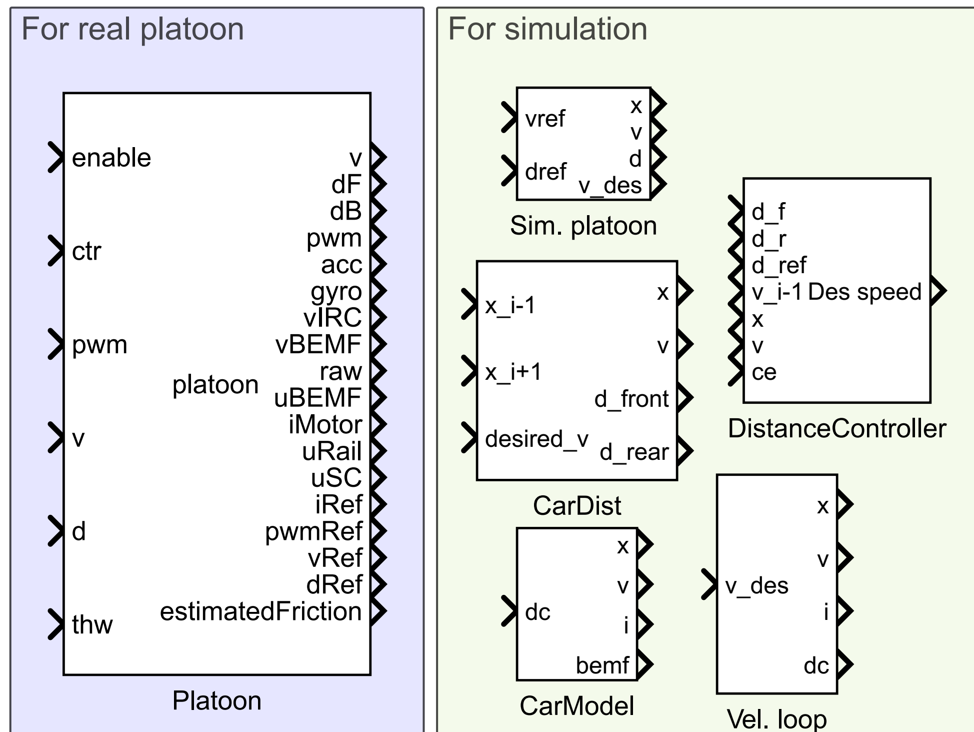


Figure 3.4: The Slotcar MATLAB Simulink library.

Sim. Platoon The block which simulates the platoon. The inside of the platoon is automatically generated by script *createPlatoon* at compile time. The block gets as inputs number of cars and the name of the controller, which will be added to each car. The controller needs to be implemented as a block with certain inputs/outputs. For more details, we provide an example, from which everything will be clear.

DistanceController An interface of a distance controller used in Sim. Platoon. An example is of a controller, as is used in this work, is implemented inside; however, it can be re-implemented by breaking the link to the library.

3.7 Slotcar Controllers

Our goal was to design the interface for making new controllers as easy as possible. Therefore, we created a Java abstract class *Controller* from which all new controllers inherit. This inheritance prescribe several methods, which are self-describing. Hence, controllers are dynamically loaded to the slotcar application in the run-time as plugins. Therefore, we do not need to compile the whole application. In other words controllers can be developed independently.

Being decoupled from the development of the main application, the control designer is freed from the need to understand the complicated infrastructure.

In fact, the code for the controller looks similar to a Matlab S-function, with which many control engineers and students are familiar. A code snippet for a simple controller is given in listing 3.2. All this gives an important advantage: with the structure of basic controller fairly simple, even a person with little programming experience can quickly implement his or her own controller.

Constructor. The constructor of Controller will ask you for period at which the controller runs and the type of output. The types of output are following

STOP Just for internal purpose. The output is ignored and the car stays still.

PWM Duty cycle of PWM, which is directly applied to motor.

SPEED Desired speed of controlled car, which is used as a reference for the in-build STM speed controller, that controls the PWM duty cycle.

Step method. Once a controller is loaded and started, the step method is periodically called. The step method need to be implemented carefully for proper functionality. It has two arguments; last known state of local and of all other cars (in case that communication is running).

Running precision. Since we are not using real-time OS, we also provide some statistics of period stability for a simple PD controller.

```
Controller statistics: src.ControllerPD
Period: 30.00 ± 0.86 (27.00, 33.00) ms
Ref period: 30 ms
Step time: 3.06 ± 0.61 (2.00, 6.00) ms
```

Implementation tips. Catch Java exceptions and minimize usage of loops, especially never use infinite loop. Careful implementation is necessary; the application has no tools how to check your code.

```
@ClassInfo(label = "P controller with FF")
public class ControllerP_FF extends Controller {


  @FieldInfo(label="Controller P constant")
  public float kp = 5f;

  private final static long period = 30l;

  public ControllerP_FF() {
    super(OutputType.SPEED, period);
  }

  @Override
  public float step(CarImageInterface me, CarListInterface cars) {
    float distRef = me.getReferenceDistance();
    float distMeas = me.getDistanceFront();
    float ctrEffort = kp*(distMeas - distRef);
    float ldrSpeed = cars.getPos(0).getSpeed();
    float totalEffort = ctrEffort + ldrSpeed;
    float desVel = Nonlinearities.sat(totalEffort, -1f, 1f);
    return desVel;
  }

  @Override
  public void init() {
    // In this case, nothing needs to be initialized.
  }

  @Override
  public void afterWritingConstants() {
    // In this case, nothing needs to be renewed.
  }
}
```

Listing 3.2: Example code of a simple proportional controller augmented with a feedforward of the leader's velocity.

Chapter 4

Beginning with the platform

This chapter should serve as a basic introduction for new people getting familiar with the project. You are working with the Raspberry Pi Compute Module (CM) and all information about the it could be found at the `www.raspberrypi.org`[1]. The most important item is the datasheet, where you can find a table with I/O [2] and other information. The website `stackoverflow.com` is your best friend while solving a problem and you even do not need to search for CM, just standard Raspberry Pi will do.

4.1 Clean installation

1. Donwload Raspbian Jessie Lite [3]
2. Flash CM [4] via CM development board, but do NOT boot just yet! It would cause resize of file system, which we do not want.
3. Before first boot you need to edit /boot/cmdline.txt and remove "init=...".
4. Then boot the RPI and connect to it via Serial COM or just with keyboard and monitor
5. log as root.

   ```
   sudo -s
   ```

6. Follow the script *slotcar-fs.git/car-fs/boot/slotcar-install.sh*. Please run the script line-by-line it is not yet automatized. Running it all at once may cause errors.
7. Change logical address of the car (it will change both the logical and the IP address.)

---

[1] `https://www.raspberrypi.org/documentation/hardware/computemodule/`

[2] Please note, that GPIO corresponds with BCM as used here `https://pinout.xyz/`, where alternative functions are also shown. All pins are the same as in the standard version of Raspberry Pi, but CM have a lot of extra pins.

[3] `https://www.raspberrypi.org/downloads/raspbian/`

[4] `https://www.raspberrypi.org/documentation/hardware/computemodule/cm-emmc-flashing.md`

```
addr {address}
```

8. Reboot.

Final tip: Do not repeat this for every car. Just prepare one image and the same way as you write into the memory you can read-out the image from the memory and write it to other cars. This guarantees that all cars will be the same.

4.2 Connecting to WiFi

Connect to the Wi-Fi in ad-hoc mode might be a problem since not every card supports it. Therefore you must be sure that you have the right card in your PC.

Linux. Connection to ad-hoc network on Linux is quite straight forward, while using a network manager. You will not notice a difference.

Windows. Windows might make the situation complicated, because for some reason they do not like them. This guide assumes that you use windows 7 or newer.

First , you need to find out whether your card supports the ad-hoc mode or not. You can do that running following command in command line.

```
netsh wlan show wirelesscapabilities
```

If your card has the support, the result includes following statement.

```
IBSS : Supported
```

Then you can search for the car-control network.

```
netsh wlan show networks
```

If the network exists you must add the network profile by following these steps:

1. Go to "Network and Sharing Center"
2. Click "Set up a new connection or network"
3. Double click "Manually connect to a wireless network"
4. Enter "car-control" (ssid) into the "Network name" field
5. Configure security settings as none.
6. Uncheck "Start this connection automatically" (important)
7. Click "Next", then "Close"

Then we need to set that the network is ad-hoc.
For Windows 10 we must do

```
netsh wlan set profileparameter car-control ^
connectiontype=ibss ^
connectionmode=manual
```

Otherwise

```
netsh wlan set profileparameter car-control connectiontype=ibss
```

Now we will connect to the network by

```
netsh wlan connect car-control
```

If you have more interfaces in your PC, then you must also specify the interface in the commands.

4.3 Connecting to CM

If you use CM development board, you can connect with a keyboard and a monitor. Or you can use serial COM, which is present on *GPIO14* and *GPIO15*. You need to use a UART-USB converter. These pins are also present on our slotcar TOP board.

Once you have flashed the CM you and you installed slotcar-fs.git. The CM will create or connect to the *car-control* Wi-Fi network. You can connect to it (see section 4.2) and use SSH.

Cars are accessible via SSH; log in by root is allowed with password root [5].

Or you can also use ssh-key for connecting directly as root.

```
private key:  ./slotcar-fs/car-fs/root/.ssh/slotcar_id_rsa
public key:   ./slotcar-fs/car-fs/root/.ssh/slotcar_id_rsa.pub
putty key:    ./slotcar-fs/car-fs/root/.ssh/slotcar_id_rsa.ppk
```

For file exchange, you should use SCP and programs like WinSCP.

4.4 Eclipse IDE

We chose Eclipse IDE as a primary development tool in which we develop software and firmware. The IDE has an advantage of plugins. For example CDT (for C development), Git, Maven, Ant scripts (XML), and GNU ARM Eclipse [6], which we all use.

You also need to install JDK (version 1.8, or newer) and GNU ARM tools embedded [7] for cross compilation, in case you want to program STM.

[5] If you change it, you also have to change it in GUI, and in the Eclipse's Ant scripts. However, we do not need to care about security; the cars are not connected to the Internet most of the time.

[6] `http://gnuarmeclipse.github.io/`

[7] Please follow their installation steps on `https://launchpad.net/gcc-arm-embedded`.

We use Maven for library management and for building the JAR for RPI (We provide eclipse launcher *SlotcarPi.launch*, that automatize the build procedure).

We use Ant script with JSCH[8] for deploying the binaries to RPI: each project has one: (1) *jar2car* uploads JAR to RPI and restarts the application, (2) *ctrl2car* uploads controllers to RPI and (3) *bin2car* which uploads BIN to RPI and flash STM. The list of cars is specified within each script, where you need to edit it. Each script is simple and commented in detail for quick understanding.

Project settings for Eclipse are also distributed via Git. All you need to do is import

```
File -> Import -> General -> Existing Projects into Workspace
```

For *slotcar-ctrls*, it is important to have up-to-date libraries, which are generated from *slotcar-sw* by script *CreateControllerLib*

4.5 Flashing STM

4.5.1 On Windows by a STM32 DISCOVERY BOARD

1. Install STM32 ST-LINK Utility, which also installs the driver for the discovery board.

2. Connect discovery board by SWD connector to the car. You must use prepared cable or make it by yourself there is a specific connector. You must connect STM SWD to discovery SWD connector. See Discovery manual for pins information. Then search slotcar-hw for the same pins and connect them.

3. Jumpers on ST-LINK must be on.

4. Put another jumper on slotcar as in setup to prevent STM falling to boot mode.

5. Connect to STM via ST-LINK.

6. Set Option Bytes, disable flash protection for all sectors.

7. Load .bin file.

4.5.2 From CM

1. Install WiringPi.

[8] To add JSCH to Eclipse, you need to download it from `http://www.jcraft.com/jsch/` and Window -> Preferences -> Ant -> Runtime -> Classpath -> Ant Home Entries -> Add External JARs...

2. Connect STM to CM via SPI when using IO board. On car's board it is connected already.
3. Copy bootloader stm32boot to RPi (You can find it in slotcar-fs.git/car-fs/slotcar/stm32boot).
4. Run "/slotcar/stm32boot -p path_to_bin.bin 0x08000000".

4.6 Linux command line commands

Slotcar. First script is used for maintaining the car. the help of the script is self explaining.

```
/bin/bash /slotcar/slotcar.sh

Usage: /slotcar/slotcar.sh {COMMAND} arg1 arg2 ...

COMMANDS:
        mro     Makes / and /boot read-only.
        mrw     Makes / and /boot read-write.
        start   Starts the slotcar application.
        stop    Stops the slotcar application.
        status  Shows status of the slotcar application.
        log     Shows the log of the slotcar application.
        addr    Returns (and sets if {arg1} is set) the address of the slotcar.
                {arg1} must be a number.
        upgrade Upgrades slotcar's file-system from Git.
        stm     Subscript for maintaining the STM.
                For more information write 'stm help'.
        ntp     Sets ntp server.
                {arg1} is the IP address of the server.
        help    Shows this message.
```

STM. Another script is a subscript of the previous one. This script particularly controls STM only. Again we show its call and help.

```
/bin/bash /slotcar/slotcar.sh stm

Usage: stm {COMMAND}

COMMANDS:
        program, p    Program STM flash.
                      Source file is "/slotcar/slotcar_stm.bin".
        start         Starts STM program.
        stop          Stops STM program.
        restart       Restarts STM program.
        help, h       Display this message.
```

Standalone STM loader. As was mentioned, we created a STM loader which a standalone application.

```
/slotcar/stm32boot -p /slotcar/slotcar_stm.bin 0x08000000
```

Bash aliases. The most used commands has been added into aliases (see listing 4.1).

Listing 4.1: Bash Command line aliases.

```
SLOTCAR_SH='/bin/bash /slotcar/slotcar.sh'
alias l='/bin/ls -al'
alias mro='$SLOTCAR_SH mro'
alias mrw='$SLOTCAR_SH mrw'
alias start='$SLOTCAR_SH start'
alias stop='$SLOTCAR_SH stop'
alias restart='$SLOTCAR_SH restart'
alias status='$SLOTCAR_SH status'
alias log='$SLOTCAR_SH log'
alias addr='$SLOTCAR_SH addr'
alias stm='$SLOTCAR_SH stm'
```

Chapter 5
Model

The vehicle is modeled as a loaded brush-type permanent magnet DC motor connected to the wheels through the gear train. The rotation of the wheels produces a force that pushes the car. The force is proportional to the torque of the wheels and also to the torque of the motor because we neglect slipping of the wheels and we consider the gear train as ideal. The car alone is represented by mass with weight m. The equations can be written down almost immediately using the bond graph in Fig. 5.1

$$\frac{\mathrm{d}i(t)}{\mathrm{d}t} = \frac{u(t)}{L} - \frac{R}{L}i(t) - \frac{k}{rnL}v(t), \tag{5.1}$$

$$\frac{\mathrm{d}v(t)}{\mathrm{d}t} = \frac{k}{mnr}i(t) - \frac{F_f(v)}{m}, \tag{5.2}$$

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = v(t). \tag{5.3}$$

The states are: the electrical current i flowing through the motor winding (in A), the velocity v of the car (in $\mathrm{m\,s^{-1}}$), and the traveled distance x (in m).

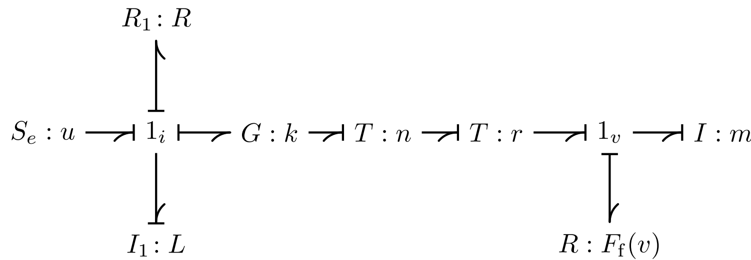


Figure 5.1: Bond graph of the one-dimensional electromechanical dynamics of a slot-car as a loaded permanent magnet DC motor with a permanent magnet.

The control input is the voltage u (in V). In reality, the input voltage comes in the Pulse Width Modulated (PWM) signal of frequency 40 kHz with the duty cycle D_c, where $D_\mathrm{c} \in [-1, 1]$. Hence $u(t) = U_\mathrm{r}\, D_\mathrm{c}(t)$ denotes just

the low-frequency content, where U_r [1] (in V) is the rail voltage lowered by the input diode voltage (0.8V). The model has one significant non-linearity and it is the static friction. The friction force

$$F_f(v(t)) = b_\mathrm{d} v(t) + b_s \mathrm{sign}(v(t))$$

applied on the car has two parts static and dynamic, where coefficient b_d is the dynamical (viscous) friction and b_s is the static friction[2]. The relevant physical parameters are in Table 5.1.

Physical parameter	Symbol	Value	Unit
Resistance of the motor winding	R	8	Ω
Inductance of the motor winding	L	2	mH
Torque constant	k	0.006	$\mathrm{N\,m\,A^{-1}}$
Mass of the slot car	m	0.15	kg
Dynamic friction coefficient	b_d	0.27	$\mathrm{kg\,s^{-1}}$
Static friction coefficient	b_s	0.53	N
Radius of the wheel	r	0.01	m
Gear train ratio	n	1/3	

Table 5.1: Parameters for the vehicle model.

We implemented the full non-linear model in MATLAB Simulink, see Fig 5.4, which is used for simulations. We can see in the figure 5.2 a comparison of the simulation and the reality. The input for this test was the PWM duty cycle from 0.1% to 0.5%. We can see that velocity and of course the back EMF (from which the velocity is calculated) fit. Therefore the model of non-linear friction is accurate. It needs to be noted, that it only works once the wheels are rotating as you can also see from the figure. It turns out that the model of static friction is more complicated when starting from zero velocity, because on the way up 0.1% was not enough to move the wheels and it was enough on the way down; it has a hysteresis.

5.1 Linearization

Note that while the linear model turns out nearly perfect for most aspects here, it fails badly when describing the friction phenomenon. Here the friction comes from three sources—friction induced by the angular motion of the rotor shaft, friction in the slot, and the rolling friction. It is well known that the rolling friction does not depend on the velocity but only depends

[1] The input (rail) voltage is not stabilized, therefore it can vary along the track because of changes in electrical resistance, although not much. It is good to make a dummy ride before running a comparison experiment just to find out how the voltage varies and then put the mean into the model.

[2] The model of friction is simplified; it is not correct for velocities around zero. For a better estimation of the friction, the Karnopp model should be used.

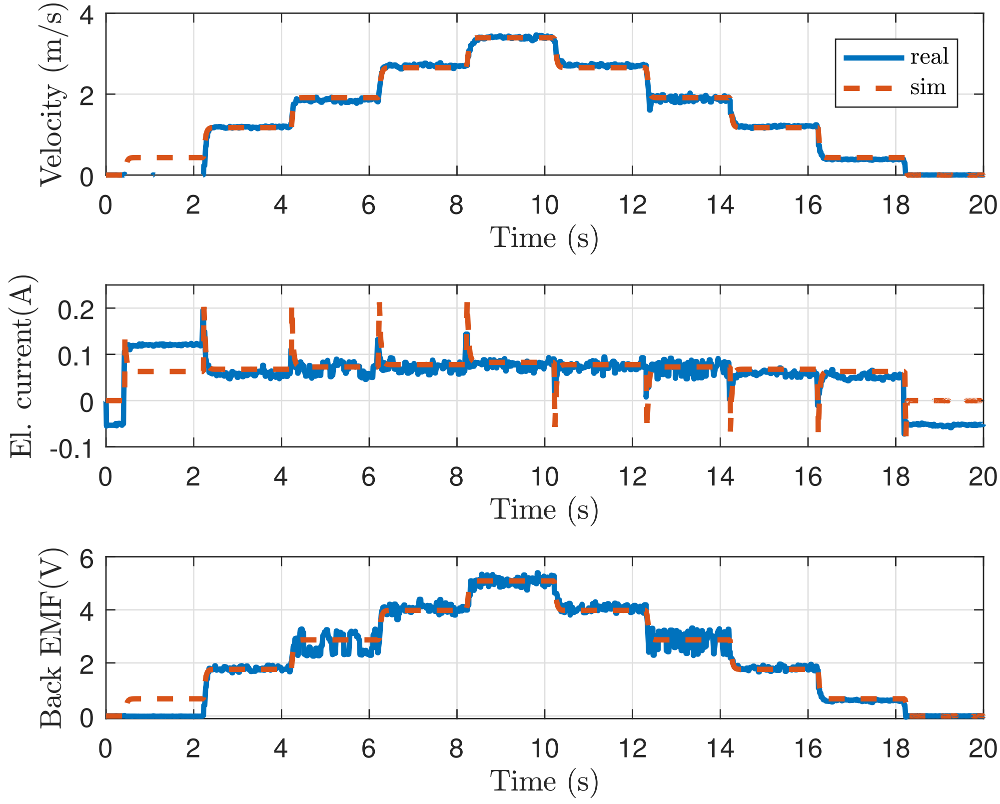


Figure 5.2: Car model: the reality compared to the simulation. In this case the car is lifted and the wheels are rotating in the air. Therefore the coefficients are different $m = 0.01$, $b_\mathrm{d} = 0.01$, $b_\mathrm{s} = 0.09$.

on the normal force (here it is not only the weight but also the attractive force of magnets that push the slot car against the track). Introducing some nonlinearity into the model seems inevitable. But temporarily, the linear model of a friction is used to get a transfer function as rough models of the overall dynamics.

The electric current dynamics is very fast compared to the mechanical dynamics of the velocity, so we can neglect it by setting $\frac{\mathrm{d}i(t)}{\mathrm{d}t} = 0$ in (5.1). Separating $i(t)$ and plugging it to (5.2), we get

$$\frac{\mathrm{d}v(t)}{\mathrm{d}t} = \frac{U_\mathrm{r}k}{Rmnr}d(t) - \frac{b}{m}v(t) - \frac{k^2}{Rmr^2n^2}v(t), \tag{5.4}$$

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = v(t). \tag{5.5}$$

Transfer functions from the input voltage are

$$G(s) = \frac{v(s)}{D(s)} = \frac{\frac{kU_\mathrm{r}}{Rmnr}}{s + \frac{b}{m} + \frac{k^2}{Rmr^2n^2}}, \tag{5.6}$$

$$H(s) = \frac{x(s)}{D(s)} = \frac{1}{s}G(s). \tag{5.7}$$

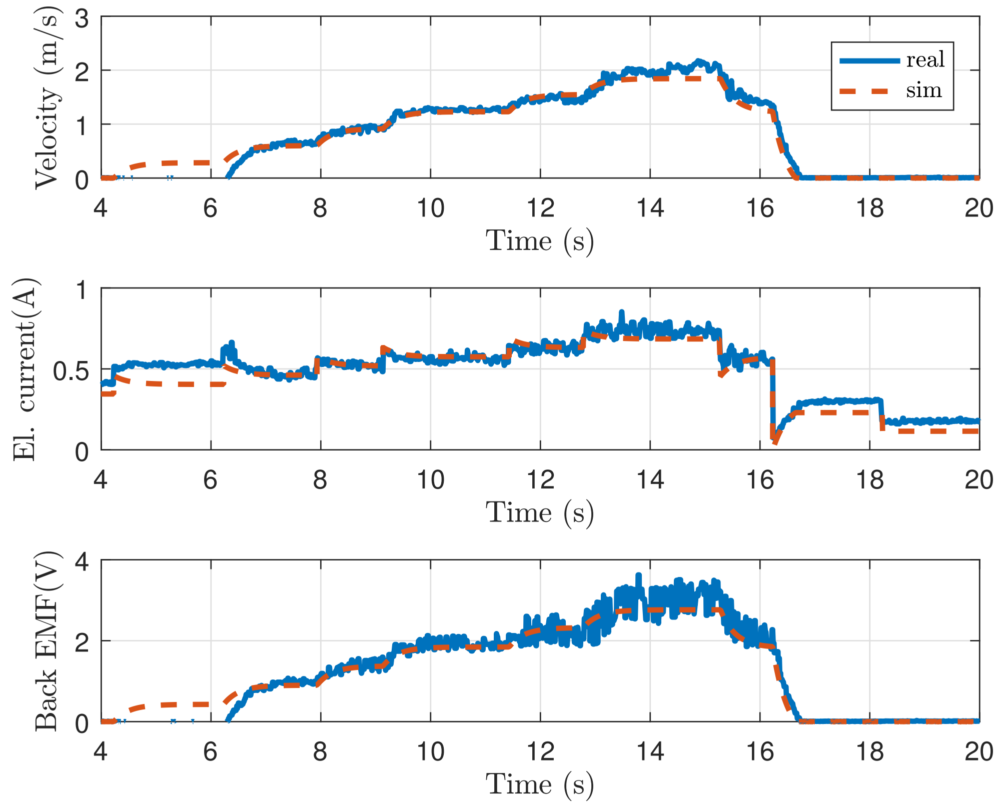


Figure 5.3: Car model: the reality compared to the simulation.

After plugging in the values of the parameters,

$$G(s) = \frac{3.2}{0.3s + 1}. \tag{5.8}$$

In combination with a *dead-zone* on input, which represents the static friction, is the linear model good approximation of reality. The dead-zone is set to range $[-0.30, 0.30]\%$ of PWM duty cycle, which is needed to overcome the static friction.

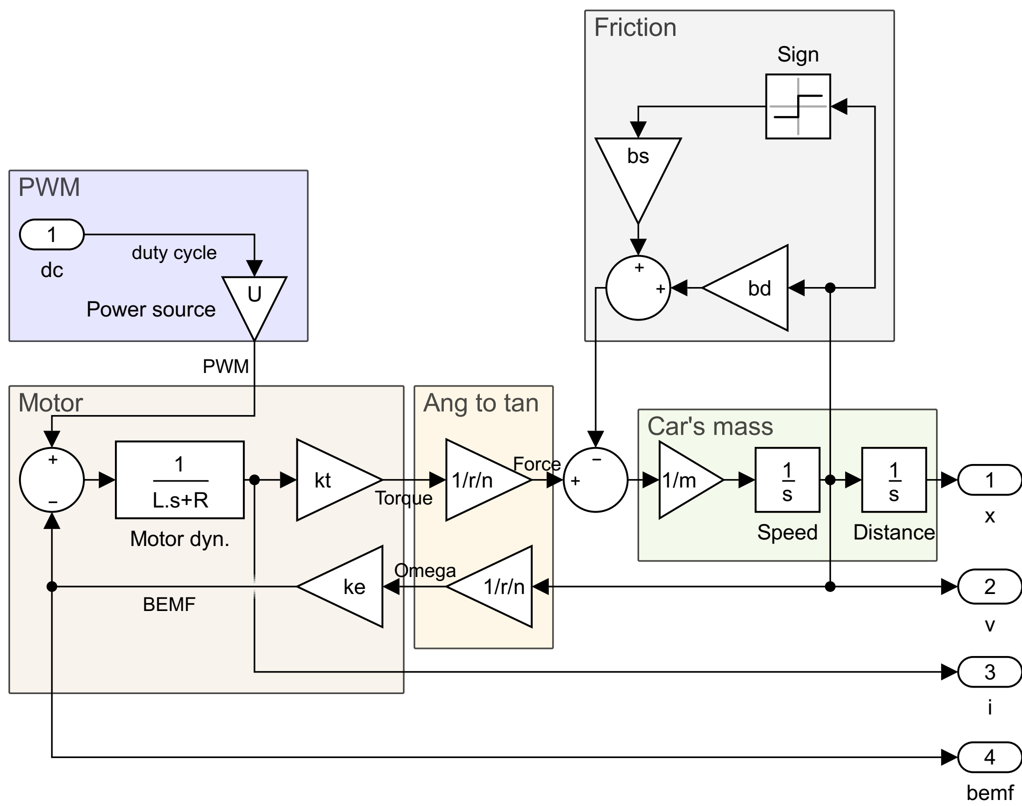


Figure 5.4: The Simulink model of the car. In the case of the car $k_e = k_t = k$.

Chapter 6

Control

We separate the control problem into (1) control of the velocity of the car and (2) control of the inter-vehicle distances. They are connected in the cascade structure, where the velocity control loop is the inner one and the distance control loop is the outer one, as seen in Fig.1.3.

6.1 Velocity control

The velocity measurement and the control loop is implemented in STM. The velocity controller is a standard discrete PI controller with following structure

$$C_{\mathrm{v}}(z) = k_{\mathrm{p,v}} + k_{\mathrm{i,v}} \frac{T_{\mathrm{s,v}}}{z-1}, \tag{6.1}$$

where $k_{\mathrm{p,v}} = 2$, $k_{\mathrm{i,v}} = 10$ and sampling period $T_{\mathrm{s,v}} = 0.005\,\mathrm{s}$. The controller implements an anti-windup in a form of clamping $[-1, 1]$. The input control error $e_v = v_{\mathrm{ref}} - v$, where v_{ref} is reference velocity. The control effort is the PWM duty cycle D_{c}. The comparison of the simulation and the real implementation, Fig. 6.2, shows a good agreement in both, the velocity and the control effort.

For continuous analysis, we use the transfer function

$$\mathrm{C}_{\mathrm{v}}(s) = \frac{2s+10}{s}.$$

The bode diagram of the velocity loop is in Fig. 6.1.

6.2 Distance control

The platform aims for being able to run many forms of controllers. Therefore, we provide all kind of measurements (see Chap. 2), that can be used. The full states of the cars are also communicated among each other. We expect that especially the communication is the key factor for a successful controller. In this section, we provide some examples of standard controllers and some extended controller by communication. In all experiments, the leader is tracking user-defined velocity profile.

6.2.1 Bidirectional control

This experiment is a revision of an experiment, that was done previously done in [5]. We briefly describe the situation. We use a simple bidirectional control law, which accepts a single input—the weighted regulation error as in

$$e = (d_i - d^{\mathrm{ref}}) + \epsilon(d_{i+1} - d^{\mathrm{ref}}), \tag{6.2}$$

where $d_i = x_{i-1} - x_i$ is the distance to the ahead, $d_{i+1} = x_i - x_{i+1}$ is the distance to the car behind, d^{ref} is the reference distance the cars should keep among each other, and ϵ is a *constant of asymmetry*. This constant weighs the contribution of the rear spacing error. When $\epsilon = 0$, the controller only uses the distance to the car ahead (the so-called *predecessor following*), when $\epsilon = 1$, the car weighs the rear spacing error with the same weight as the front error (so called *symmetric bidirectional control*).

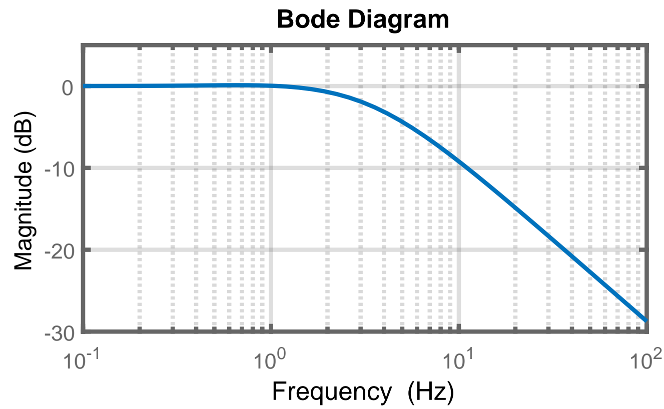


Figure 6.1: The bode diagram of the velocity control loop.

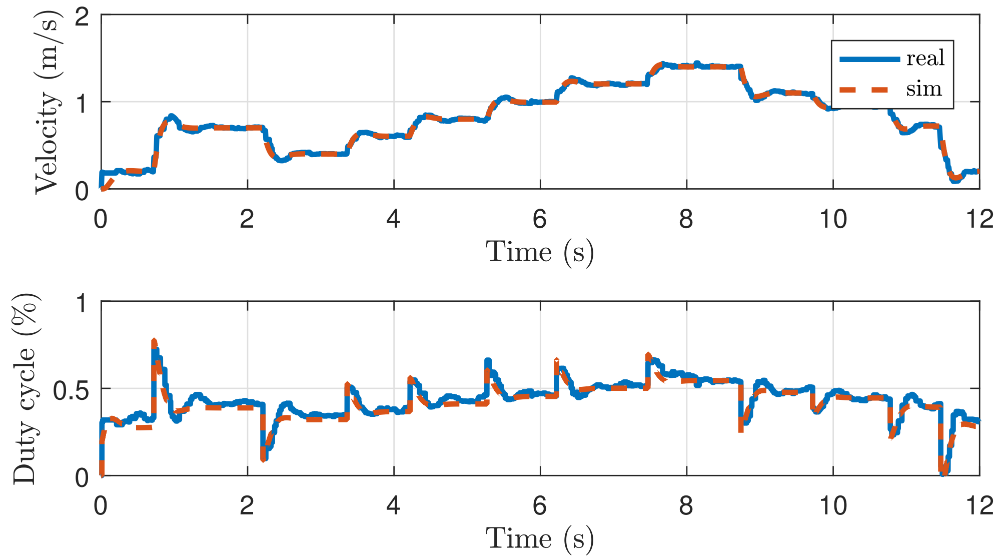


Figure 6.2: Comparison of speed controller in reality and in simulation. The PWM is the control effort.

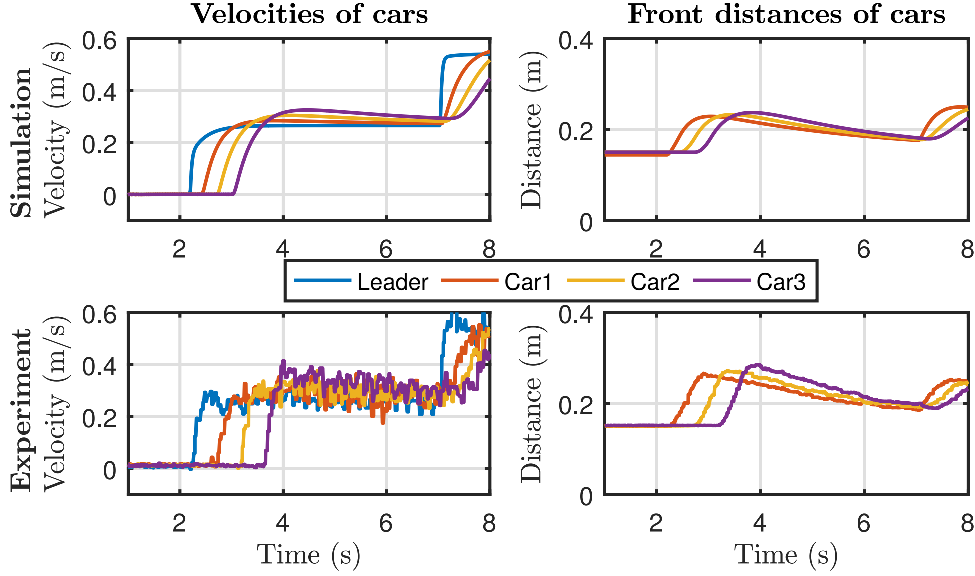


Figure 6.3: Example of predecessor following ($\epsilon = 0$) with PI controller ($k_{\mathrm{p}} = 3$, $k_{\mathrm{i}} = 0.8$, and $T_{\mathrm{s}} = 0.03$). Cars are tracking the reference distance $d^{\mathrm{ref}} = 0.15\,\mathrm{m}$. The experiment expose higher over-shoot than simulation, because the platoon starts from zero velocity and the model neglects the Stribeck effect of the mechanical friction.

In first experiment we wanted to track the reference distance. When leader moves with constant velocity, the d_i is a ramp signal–In order to track a ramp signal double integrator in open-loop is needed. One integrator is already a part of the system from v to x and we added one integrator in controller. We selected a discrete PI controller

$$C(z) = k_{\mathrm{p}} + k_{\mathrm{i}} \frac{T_{\mathrm{s}}}{z-1}, \tag{6.3}$$

where k_{p} resp. k_{i} is proportional resp. integral constant. The example of *symmetric bidirectional control* is depicted in Fig. 6.4 and *predecessor following* in Fig. 6.3. Which again proves the good agreement between the model and the reality (our platform). However, having another integrator in the open open loop means slow and oscillatory transient.

In the second experiment, we only required tracking of the leader's velocity and the steady-state distance error could be nonzero. Therefore, we needed only one integrator in open-loop (one integrator is already a part of the system). An instance of such a controller is the following filtered PD controller

$$C_d(z) = k_{\mathrm{p}} + k_{\mathrm{d}} \frac{N}{1 + N \frac{T_{\mathrm{s}}}{z-1}}, \tag{6.4}$$

where k_{p} resp. k_{d} is proportional resp. derivative constant, N is filter coefficient. In our experiments, we use the PD controller with parameters $k_{\mathrm{p}} = 5$, $k_{\mathrm{d}} = 0.2$, $N = 50$, and $T_{\mathrm{s}} = 0.03$. The example using PD controller is depicted in Fig. 6.5.

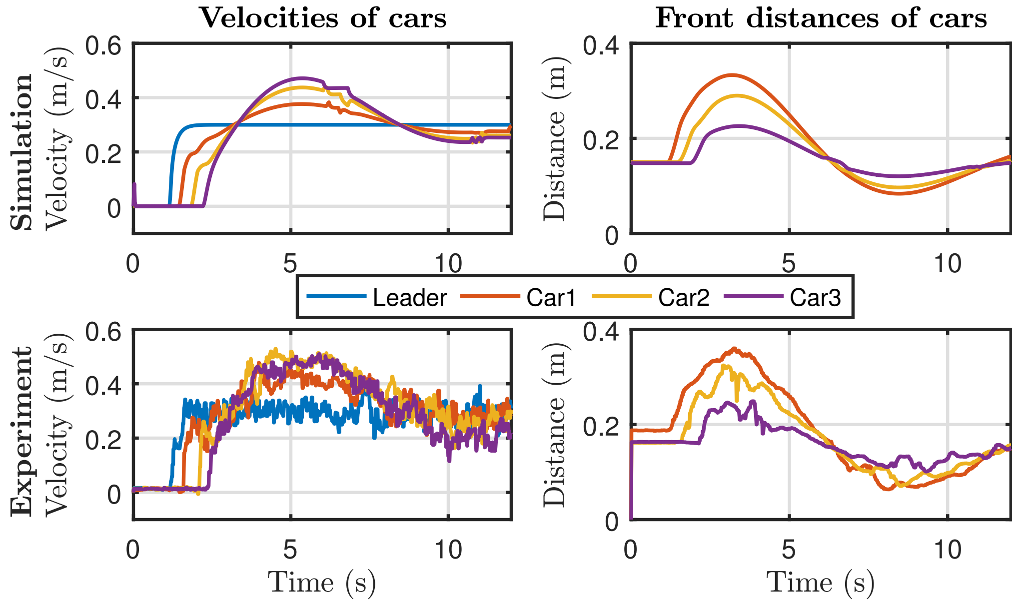


Figure 6.4: An example of symmetric bidirectional control ($\epsilon = 1$) with PI controller, where $k_\mathrm{p} = 2$, $k_\mathrm{i} = 2$, $T_\mathrm{s} = 0.03$.

The steady-state error was able to achieve by using the wireless communication, where leader send its velocity $v_0(t)$ to all the cars. The $v_0(t)$ was added as a feed-forward signal to the desired velocity produces by the distance controller. The control law for the ith car becomes

$$v_i^{\mathrm{ref}}(t) = v_0(t) + r_i(t), \tag{6.5}$$

where r_i is the output of the distance controller (6.4). We can see, that unlike the controller without communication (Fig. 6.5, the same controller with augmented with communication (Fig. 6.6) is able to track the reference distance.

6.2.2 Cooperative adaptive cruise control

This example is similar to cooperative adaptive cruise control (CACC) as presented in [8]. The structure, Fig. 6.7, shows that we augment our control loop with standard PD controller, as was previously used in *predecessor following*, with time-headway and the target speed v^{ref} (obtained by wireless communication) of the predecessor. The preceding car-following policy has following structure

$$P(s) = hs + 1, \tag{6.6}$$

where h is time-headway. The result of such control can be seen in Fig 6.8. Additionally, forwarding the target speed of the leader v^{ref_0} was tested; see 6.9.

In these cases of CACC experiments, we can see, that the advantage of communication is in fast reaction time and, although not clearly shown, it allows tracking the target distance, which is where a standard PD fails. Moreover,

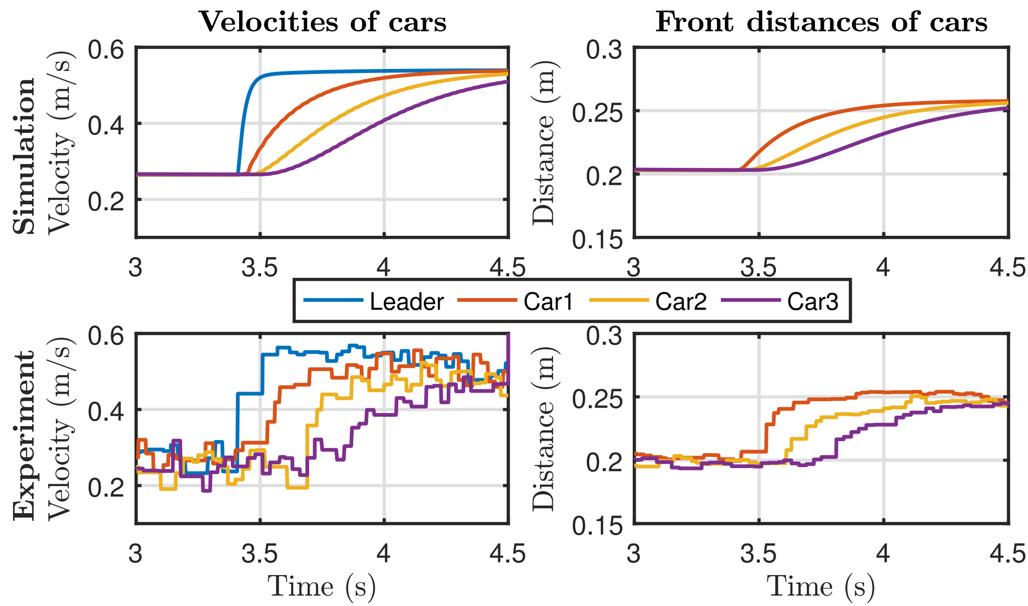


Figure 6.5: An example of predecessor following ($\epsilon = 0$) with PD controller. We can see, that the controller alone is not able to track the desired distance $d_{\text{ref}} = 0.15\,\text{m}$.

forwarding the target speed of leader gives almost immediate response of other cars. Whereas in predecessor forwarding, it takes some time to propagate the change in target velocity of the leader. The simulations exposes, that forwarding the target speed of the predecessor gives no overshoot unlike forwarding target speed of the leader; however, not proved by experiments.

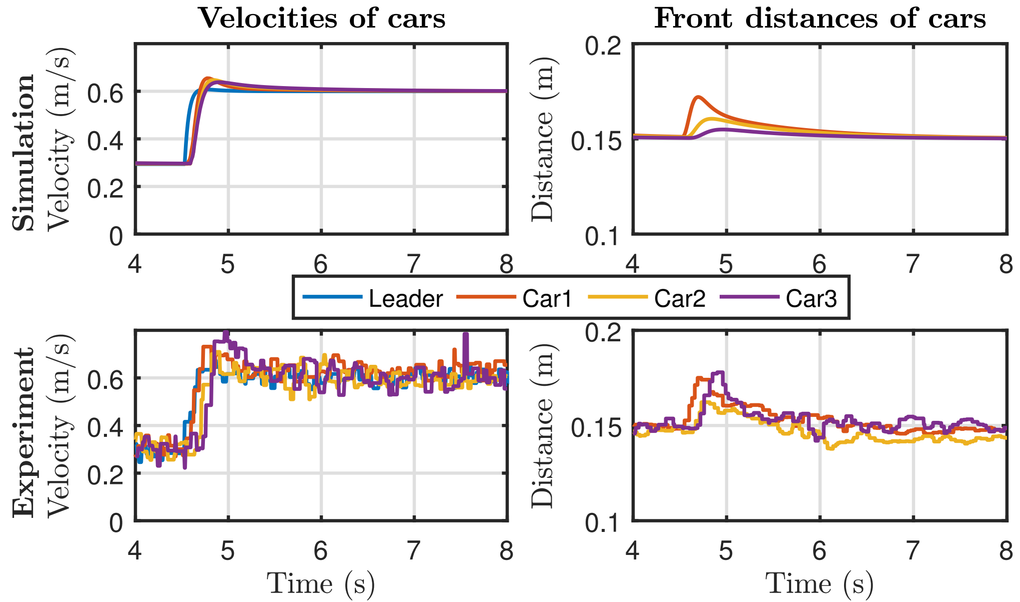


Figure 6.6: An example of symmetric bidirectional control ($\epsilon = 1$) with PD controller extend with the feed-forward from the speed of the leader v_0, because of which the cars are able to track the reference distance $d^{\text{ref}} = 0.15\,\text{m}$.

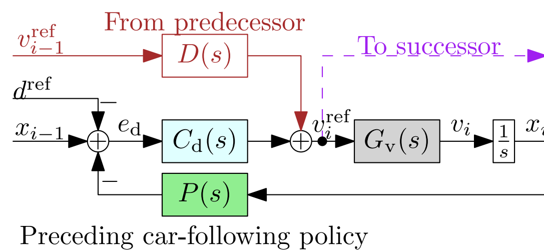


Figure 6.7: The structure of CACC. Distance controller C_d extended with feed-forwarded information of predecessor by using wireless communication.

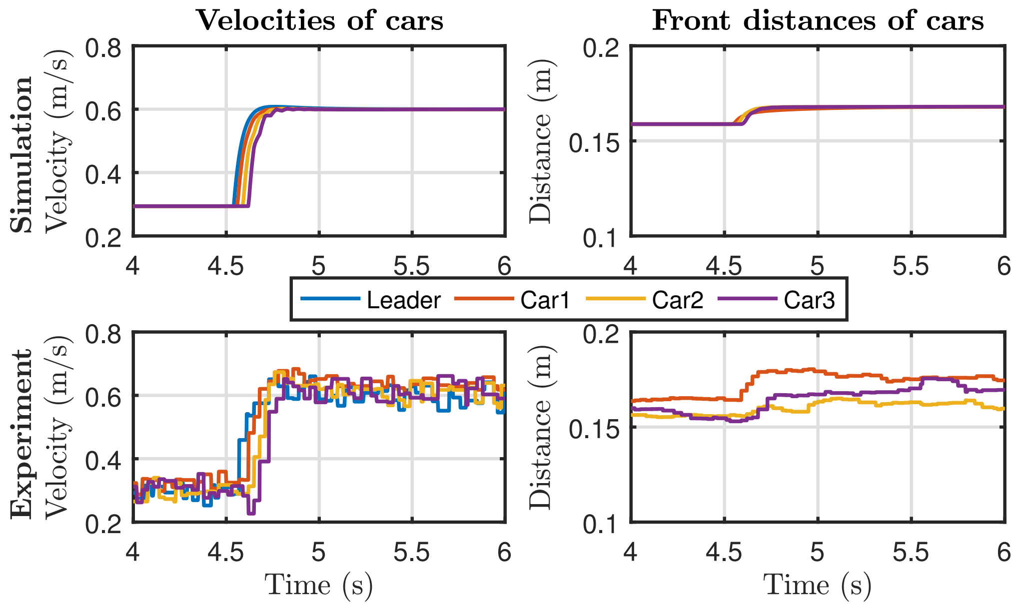


Figure 6.8: An example of CACC; the **target speed of the predecessor** as the feed-forward. the Tracking distance $d^{\text{ref}} = 0.15\,\text{m}$ and time-headway $h = 0.03$.

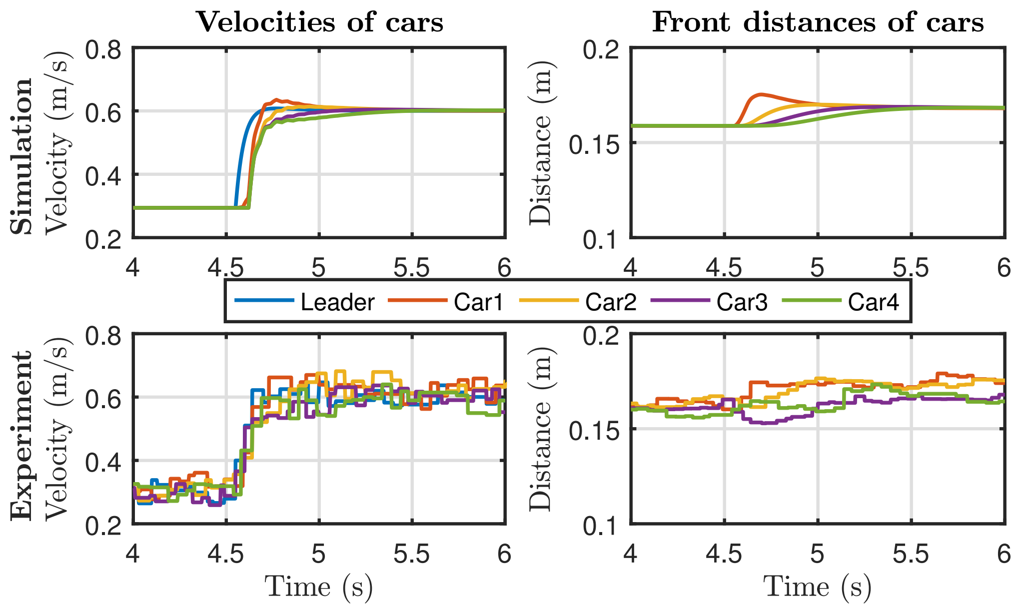
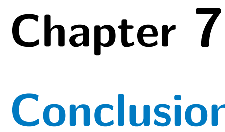


Figure 6.9: An example of CACC; the **target speed of the leader** as the feed-forward. Tracking distance $d^{\text{ref}} = 0.15\,\text{m}$ and time-headway $h = 0.03$. Shows immediate response in velocity.



Chapter 7
Conclusion

This thesis describes the progress of work done on the continuously-evolving experimental slotcar platform. My goal was to develop the software and the firmware parts of the platform and to implement some controllers and evaluate them in experiments. The platform, as it is designed today, is fairly user-friendly. A user just needs to implement the controller interface and does not need to understand the very platform details. The inputs of interface are state of the car running a controller and states of other cars (obtained by the communication). The state includes every quantity for which we have a sensor or which is calculated somehow else. However, It is necessary to mention that operator still needs to be familiar with the platform when doing maintenance. It should be covered within this work. The description here should also serve as a manual for new contributors, as well as for users as an overview of capabilities.

The experiment evaluations showed that the platform is ready for control algorithms based on wireless communication. It was also shown that the communication, as it was expected, improves the transient response of the platoon. As seen in an example, when PD controller is extended by the target velocity of the predecessor, the response of the car is faster and can track the reference inter-vehicle distance. However, the communication interface itself needs to be inspected more closely, and more precise statistics need to be made. The results showed a positive agreement between simulation and the real data. Therefore, it fulfills the essence of a platform, and it is able to verify theoretical findings in practice. Of course, there are some limitations in a way that the simulation is ideal and does not include turnings, a noise of the sensors, and other disturbances.

There is still some room for improvement and some tasks to be done. To get the platform closer to perfection, some breakthrough decisions must be made. Our communication interface should be replaced by something more standard such as ZeroMQ. Service messages are now hard coded into the communication interface this should be replaced by JSON to make it more flexible. Next, there is a possibility of using the electric current (measured on the motor) to improve velocity estimation or to compensate the mechanical friction. The measurement of current must be improved; It is heavily filtered in order to get reasonable values; a lot of noise is present. The source of the

noise was not found. There is also a possibility of using other measurements for instance acceleration and rotation to get a better estimation of velocity. In future, IRC sensor should be removed because of the mechanical construction and complicated maintenance. After all, it does not provide excellent precision. Its only advantage is the simplicity of the measurement. Therefore, it can be taken as a reference of a steady velocity value. For having the cars even closer to the model, it is needed to add a stabilization of rail voltage to each car. As cars are moving along the rail, the resistance of the track and therefore the voltage slightly varies; however, it is noticeable in the simulation. These are just tips how to make the platform better; not failures.

The only serious mistake is that the *bottom* board contains an error in the design. This error causes an occasional burn out of the board. It is rare; yet several boards were destroyed, and the reason is still not clear. It is probably connected with the power supply chain.

As a final conclusion the platform works well as you can see on videos at [6].

Bibliography

[1] P. Barooah and J. P. Hespanha. “Error Amplification and Disturbance Propagation in Vehicle Strings with Decentralized Linear Control”. In: *Proceedings of the 44th IEEE Conference on Decision and Control*. Dec. 2005, pp. 4964–4969. DOI: 10.1109/CDC.2005.1582948.

[2] Fairchild Semiconductor. *QRE1113, QRE1113GR Minature Reflective Object Sensor*. 2011. URL: http://cdn.sparkfun.com/datasheets/Sensors/Proximity/QRE1113.pdf.

[3] Inc. Freescale Semiconductor. *Xtrinsic MAG3110 Three-Axis, Digital Magnetometer*. 2013. URL: http://www.nxp.com/assets/documents/data/en/data-sheets/MAG3110.pdf.

[4] Martin Lád. “Design and implementation of a control system for a slot car”. Bachelor thesis. Czech Technical University in Prague, 2014. URL: https://support.dce.felk.cvut.cz/mediawiki/index.php/Bp_414_en.

[5] Martin Lád, Ivo Herman, and Zdeněk Hurák. “Vehicular platooning experiments using autonomous slot cars”. In: Toulouse, France, July 2017. URL: http://aa4cc.dce.fel.cvut.cz/content/vehicular-platooning-experiments-using-autonomous-slot-cars.

[6] Martin Lád and Zdeněk Hurák. *Slotcars for vehicular platooning*. 2017. URL: https://hackaday.io/project/19087-slotcars-for-vehicular-platooning.

[7] Dan Martinec. “Distributed control of platoons of racing slot cars”. MA thesis. Czech Technical University in Prague, 2012. URL: https://support.dce.felk.cvut.cz/mediawiki/index.php/Dp_480_en.

[8] V. Milanés et al. “Cooperative Adaptive Cruise Control in Real Traffic Situations”. In: *IEEE Transactions on Intelligent Transportation Systems* 15.1 (Mar. 2014), pp. 296–305. ISSN: 1524-9050. DOI: 10.1109/TITS.2013.2278494.

[9] G. J. L. Naus et al. “String-Stable CACC Design and Experimental Validation: A Frequency-Domain Approach”. In: *IEEE Transactions on Vehicular Technology* 59.9 (Nov. 2010), pp. 4268–4279. ISSN: 0018-9545. DOI: 10.1109/TVT.2010.2076320.

[10] P. Seiler, A. Pant, and K. Hedrick. "Disturbance propagation in vehicle strings". In: *IEEE Transactions on Automatic Control* 49.10 (Oct. 2004), pp. 1835–1842. ISSN: 0018-9286. DOI: 10.1109/TAC.2004.835586.

[11] STMicroelectronics. *LSM330DLC, iNEMO inertial module: 3D accelerometer and 3D gyroscope.* 2012. URL: http://www.st.com/en/mems-and-sensors/lsm330dlc.html.

[12] STMicroelectronics. *STM32F401RB.* 2017. URL: http://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32f4-series/stm32f401/stm32f401rb.html.

[13] Andras Tantos. *H-Bridge Secrets.* 2011. URL: http://www.modularcircuits.com/blog/articles/h-bridge-secrets/.

[14] Texas Instruments. *DRV8816 DMOS Dual 1/2-H-Bridge Motor Drivers.* 2013. URL: http://www.ti.com/lit/ds/symlink/drv8816.pdf.

Appendix A

The contents of the enclosed CD

The CD includes:

- Text of this thesis in pdf format.
- All git repositories as they were listed in Chapter 1.