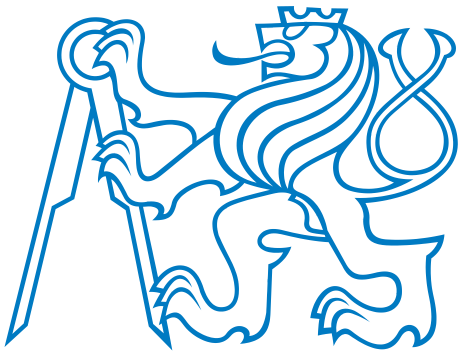
DISTRIBUTED CONTROL OF A PLATOON OF WIRELESSLY COMMUNICATING SLOTCARS USING A SIMULINK GENERATED CODE

BC. ŠIMON WERNISCH



Master's thesis
Department of Control Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague

24th May 2019

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Wernisch** Jméno: **Šimon** Osobní číslo: **412009**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra řídicí techniky**

Studijní program: **Kybernetika a robotika**

Studijní obor: **Systémy a řízení**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Distribuované řízení konvoje komunikujících autodráhových autíček s využitím kódu automaticky generovaného ze Simulinku

Název diplomové práce anglicky:

Distributed control of a platoon of wirelessly communicating slotcars using a Simulink generated code

Pokyny pro vypracování:

Seznam doporučené literatury:

[1] Documentation for Simulink Coder by The Mathworks. Available online at https://www.mathworks.com/help/pdf_doc/rtw/index.html.
[2] Documentation for Embedded Coder by The Mathworks. Available online at https://www.mathworks.com/help/ecoder/.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Zdeněk Hurák, Ph.D., katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **15.02.2019** Termín odevzdání diplomové práce: **24.05.2019**

Platnost zadání diplomové práce: **20.09.2020**

doc. Ing. Zdeněk Hurák, Ph.D. podpis vedoucí(ho) práce	prof. Ing. Michael Šebek, DrSc. podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

PROHLÁŠENÍ

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 2019

bc. Šimon Wernisch

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ABSTRACT

In this work, a managed software project is presented which aims to provide a framework for rapid prototyping of distributed control using MATLAB and Simulink software. The framework is made of a set of scripts that automate the task of defining a distributed platform and effortlessly generating code for the platform from a single Simulink scheme. For this purpose, the capabilities of networking middleware are explored to create the general-use networking interface. The distributed platform in question is the slot car platoon with BeagleBone Blue development boards at its core. Simulink implementation of the connected hardware components is provided. As part of the demonstration of the functionality of the project, the slot car system model is presented, identified and validated. This includes the friction force model, which is then compensated using feedback control. Basic controller components of the *Cooperative Adaptive Cruise Control* concept are prepared for further controller design and demonstrated through simulation and hardware experiments with the slot cars.

Keywords: *platooning, distributed control, embedded system, Simulink, wirelessly communicating experimental platform, friction compensation, CACC*

ABSTRAKT

V této práci je představen spravovaný softwarový projekt, jehož cílem je poskytnout prostředí pro rychlé prototypování distribuovaného řízení za použití MATLAB a Simulink softwaru. Toto prostředí se skládá ze sady skriptů pro automatizaci úlohy definování distribuované platformy a snadného generování kódu pro tuto platformu z jediného simulinkového schématu. Za tímto účelem jsou prozkoumány možnosti síťového middlewaru pro vytvoření síťového rozhraní s všeobecným využitím. Distribuovaná platforma, pro kterou je projekt připraven, je kolona dráhových autíček s vývojovými deskami BeagleBone Blue. Je poskytnuta simulinková implementace připojených hardwarových části. Jako součást demonstrace funkčnosti projektu je představen, identifikován a ověřen model systému dráhového autíčka. Součástí je i model třecí síly, který je poté kompenzován zpětnovazebním řízením. Základní řídící prvky konceptu *kooperativního adaptivního tempomatu* jsou připraveny pro další návrh řídícího systému a demonstrovány skrze simulační a skutečné experimenty s dráhovými autíčky.

Klíčová slova: *platooning, distribuované řízení, vestavěný systém, Simulink, bezdrátově komunikující experimentálný platforma, kompenzace tření, CACC*

ACKNOWLEDGMENTS

First, I would like to thank my supervisor doc. Ing. Zdeněk Hurák, Ph.D. for his provided insight and continued support of this project. Then I thank my family for a healthy and stable living environment. Lastly, I thank the collective of friends and coworkers at the AA^4CC laboratory for sharing their experience on tools we used.

Prague, 24th May 2019 Šimon Wernisch

ACRONYMS

AA⁴CC Advanced Algorithms for Control and Communications 1
ADAS advanced driving assistance system xvii
API application programming interface 10
CACC cooperative adaptive cruise control viii
EMF electro-magnetic field 14
GUI graphical user interface 1
IC integrated chip 31
IMU inertial measurement unit 2
MPC model predictive control 17
NNG nanomsg-next-generation 4
NoC network on chip 2
OS operating system 3
PCB printed circuit board 1
PI proportional-integral 17
PWM pulse width modulation 13
PRU programmable realtime unit 2
SiP system in package 2
SSH secure shell 3
TCP transmission control protocol 10
TF transfer function 15
UDP user datagram protocol 1
UDT UDP-based Data Transfer 5
UI user interface xvii
ØMQ ZeroMQ 10

INTRODUCTION

While adaptive cruise control and a multitude of other advanced driving assistance systems (ADASs) are common in new sold vehicles, the future where autonomous and efficient driving is utilized to the fullest may still seem like science fiction. With the development over the last decade it is not unimaginable. In fact, there are optimistic predictions that as early as 2030 there might be reliable and affordable autonomous vehicles [1]. On the other hand, the general public opinion on the safety is rather skeptical [2] and as such acceptance will most likely be gradual and slow.

The vision is to improve safety, security, fuel efficiency, road capacity, reduce travel time, congestion and more [3]. This will be achieved by eliminating human error through autonomy and intelligent traffic control by communication and planning. One of the topics in this area and a partial subject of this work is platooning. By that term we usually understand driving vehicles as part of a single lane convoy. Much theoretical work has been put into the study of vehicular platoons. Experiments have also been carried out, but due to the associated cost such experiments have been limited to relatively small convoys and overprotective margins for error. For this reason the slotcar platooning project exists at the Department of Control Engineering at the Czech Technical University.

The basis of this project is an embedded system inside a slotcar. Such a device allows design and verification of a self-controlled system similar in physical dynamics to that of a train, tram or a car. Many students have worked on this project through a few software and hardware iterations. The current one is focused on utilizing the defacto standard in control design tools, Mathworks MATLAB and Simulink software, and the low cost development board BeagleBone Blue. This small piece of hardware has often been compared to the ever popular Raspberry Pi with a larger aim on providing a feature set suitable for robotics use. It also recieves good support by the BeagleBoard.org and elinux.org communities and is supported by Mathworks in their software [4]. Other well packaged parts and a 3D printable enclosure to hold them are provided which gives a ready to use device. It is cheap to buy and time saving to assemble.

While the hardware for the current slotcars had been mostly finalized by previous students, the software part had been lacking by not providing a user interface (UI) for fast controller design and verification. The UI should allow fast online manipulation of the platoon. To explain this, through the Simulink product family, Simulink Coder and Embedded Coder, one is able to execute a model of a controller on a single development board and use many of the products' advanced features to tune and verify. Should one wish to use a multitude of development boards in their design, they will run into a hurdle with the complexity of setting up and executing their models. The complexity lies in the need to design each board in a separate model, explicitly setup communication between the boards, and manually execute each model. While it is not impossible to do, a simplifying solution should exist. That should be the major task of this work, removing this complexity and allowing fast prototyping for distributed systems.

FIRST CHAPTER tells the project's history and contributors and describes the hardware used in more detail, including preparation to obtain a working copy

SECOND CHAPTER reports the work done on the MATLAB and Simulink workflow for systematic code generation and execution on distributed systems

THIRD CHAPTER gives theoretical background and demonstrates use of the project for platooning control using the improvements described in the second chapter

FOURTH CHAPTER concludes and summarizes the work

1 SLOTCAR PLATOONING PROJECT

Distributed control of spatially distributed systems is a research topic of the Advanced Algorithms for Control and Communications (AA^{4}CC) group [5][6]. One of the experimental platforms designed there under the supervision of doc. Zdeněk Hurák is the slot car platoon. First worked on by Dan Martinec using the Lego Mindstorms NXT [7] and then using a custom printed circuit board (PCB) by Freescale Semiconductor and modified Carrera Ford Capri RS Tuner 3 slotcars [8] in 2012.

The next iteration in 2017 was the work of Martin Lád, Filip Svoboda and Filip Richter. It featured a similar Carrera slot car, this time however with a Raspberry Pi Compute Module and USB WiFi and extensive software for control of the slotcars over a network with a graphical user interface (GUI) [9, 10]. I have worked with this version and contributed to its code a little in my bachelor's studies. This version also featured custom PCB design with a `STM32F401` chip for the sensor data acquisition and motor control interface and to regulate the power supply.

This solution, while powerful, proved to be difficult to maintain as the Carrera parts were prone to wear and the PCB had to be fitted, soldered and checked for problems as the author of the current third revision also notes [11]. And this is only the hardware side of things. The software was a mixture of a monolithic Java application for both the Raspberry Pi boards and the computer running the control GUI and custom written C program for the controller. There was interfacing with Simulink software over a user datagram protocol (UDP) connection too. Simulink code could not be run on the slot car and as such the use of the tool was for the most part limited to creating reference control signals and logging scope data. Each controller had to be written inside the application Java code and selected through the Java GUI. This all was hard to read into and learn and while the choice was justified at the time, the new goal for Marek Bečka was to create a simpler maintainable platform utilizing modular, well supported and accepted components from known companies and communities.

Concurrently with Marek Bečka another bachelor student Petr Bláha tackled the issue described in the introduction which was to create a workflow inside the Mathworks ecosystem whose complexity didn't grow with the number of boards inside a modeled experiment [12]. He provided a solution which was demonstrated using two BeagleBone Blue boards exchanging measurement data between themselves and with the computer running Simulink. His script separated predefined device subsystems from a parent model and replaced the interfaces with UDP blocks. These were then distributed to their devices and ran independently.

In this work, I try to pick up after the aforementioned students following the same design philosophy to extend and improve the solution. With the BeagleBone support package by Simulink being somewhat limiting and having little documentation one could argue that using a self-prepared solution would be better. I am against this approach. The package will progressively get better and be kept up to date with the BeagleBone and Matlab development while a completely custom solution might age quickly. There is enough to work on in parts that do not seem to be on the Matlab roadmap, are absolutely needed or demonstrate how things can be done in the project.

1.1 HARDWARE

Most of this section will describe the work of the aforementioned Marek Bečka who documented his solution in his Czech bachelor's thesis [11]. His chassis work is incorporated into the repository listed below in section 1.2.

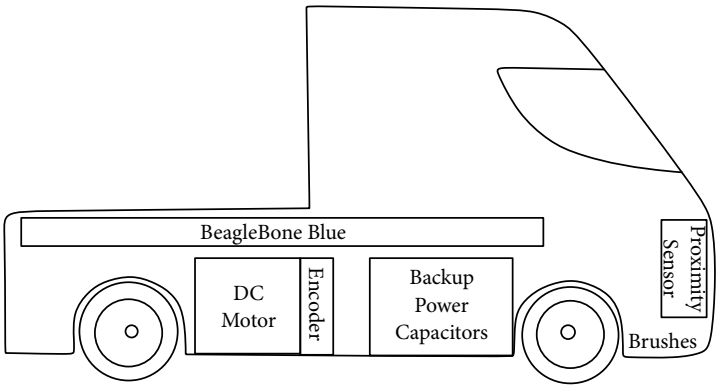


Figure 1.1: Layout of slotcar hardware from the side

For a basic picture see figure 1.1. Each slot car's basis is a `slot.it HRS2` chassis, which includes a 23000 rpm DC motor with 9:28 gear and thin copper braided brushes for drawing voltage from the track. The braids and motor are disconnected and the former is connected through a diode to a series of backup capacitors and then both are connected to the BeagleBone PCB vias for the DC jack as documented in figure 1.2. The backup power supply has a capacitance of 0.5 F, which lasts long enough for the BeagleBone to not power down in case of the temporarily lost connection between the brushes and the power rail of the track. Further filtering is done by the board itself. I have replaced the supercapacitors with smaller package supercapacitors to improve the wiring and mechanical toughness in the lower part of the slot car. The braids need to be taken care of as dust accumulates in them and they deform as the slot car moves. Otherwise, power failures may occur. The BB board sits on top of custom printed mounting columns on top of the HRS2 chassis. The outer chassis holds the proximity sensor and is mounted on the sides and front into the HRS2.

Personally, I had little trouble assembling the slotcars and as such, I did not change their design. Of course, future users of the platform might select different parts, remodel them or add additional ones. The design is modular and the outer chassis and BeagleBone holders are custom 3D printed. Even in the current outer chassis design, there is enough space to fit for example a front facing camera.

It is also worthy to mention the capabilities of the BeagleBone Blue. It contains an inertial measurement unit (IMU), motor drivers, 802.11b/g/n wireless network on chip (NoC) and a large array of connectors for external devices [13]. Its system in package (SiP) features a `Cortex-A8` ARMv7 32-bit processor and two programmable realtime unit (PRU) 200 MHz microcontrollers. These PRUs add hard real-time low latency processing to the board to control peripherals. They contribute to the

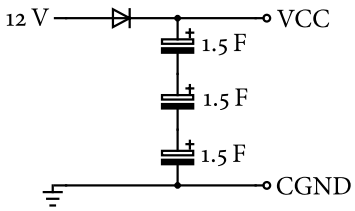


Figure 1.2: Input power backup circuit

extensibility selling point of the device as custom protocols and tight control loops can be implemented [14]. Think of a Raspberry Pi and two Arduinos [15] with a high-bandwidth connection directly, over a bus, via interrupts or DMA [16]. The BeagleBone boards are fully open-hardware and many vendors contribute, make and sell them. Overall the BeagleBone is worthy of consideration for diverse embedded and robotic applications and far exceeds the capabilities of the Raspberry Pi.

One last note I wish to say about the hardware concerns the 3D printed parts. Previously the PLA material had been used and there had been minor concerns with degradation over time, the plastic becoming brittle and with difficulty bending the plastic into place. Using the tougher ABS material has made the slot car more durable. It is, however, harder to print and has a tendency to break between print layers when printed bellow optimal printing settings on cooler temperatures. This can be an issue with the thin outer chassis and vertical joints so care needs to be taken to increase printing temperature but not damage the printer. Instead of heating up the print to bend into place it is possible and for some easier to use a modeling scalpel or other knife to remove supports and excess material in joints. The choice of printing method is free and both the mentioned materials serve their purpose with pros and cons but PLA seems to be more suited for additive printing even concerning overall mechanical properties [17].

1.2 ACCESSING THE PROJECT

The project is lead as a student project without backing and it is purposefully open-source and open-hardware. The repository can be found on Github [18]. The 3D printed chassis files are kept in the A360 web storage. Public links for these files are listed in the `cad` folder in the repository. Currently, no proper schematic of the board and connected circuits exist because the only custom circuit is the backup power capacitors, see figures 1.1 and 1.2. Other hardware is wired through standard interfaces. BeagleBone has it's own hardware documentation repository [13].

1.3 SETTING UP THE BEAGLEBONE SYSTEM

In this section I will provide the steps to setting up an operating system (OS) for the BeagleBone. Mostly to show my chosen sources and flows and to describe the steps that are generally undocumented or changed.

First, there is the choice of OS. Any OS built for the hard float ARM-v7 architecture can be used with the BeagleBone. For ease, one should pick a distribution by BeagleBone communities which include the device tree overlay, drivers and most necessary packages. BeagleBone's Ångström and Debian are commonly used [19][20] but to utilize Matlab support for the development board, Debian IoT is recommended [21]. In fact, the support package requires a specific kernel and version of `librobotcontrol` library which come preinstalled on the BeagleBone Debian image [21] [22]. On Matlab version `2018b` this is Debian `9.5` and `librobotcontrol 1.0.3`. This information can be found in the package's setup procedure mention below. The choice has been narrowed down to a single image. With the distribution image selected, the next steps are

1. editing /boot/uEnv.txt file inside the image to make sure the script used to clone the system to internal eMMC on startup is commented out
2. making a bootable SD card using the Linux `dd` tool or the Etcher utility [23]
3. booting off this SD card on a BeagleBone connected to the computer over a USB cable and connecting over secure shell (SSH)

4. making changes inside the system
5. uncommenting the uEnv line to allow flashing to internal eMMC
6. cloning the distribution to each of the devices by inserting the SD card and powering up the board

How to do these can be found in many internet tutorials and there are more ways to achieve the same goal. Doing things this way saves some time and leaves an SD card backup in case starting over on a board is needed. As for the changes done to the distribution, these are

- setting up `connman` to connect to a wireless network
- running the Matlab support package setup script to install some packages
- adding users and changing passwords
- switching to a real-time kernel
- updating the packages and installing build tools
- building libraries not in the package manager

The Simulink package must be set up before switching kernels because it checks and doesn't accept any other. This is done from the addon manager window. A network can be connected either using the package setup or from the command line using `connmanctl`. The package setup also installs needed packages if the BeagleBone is connected.

The supported 9.5 version of the distribution comes with kernel 4.14.71-ti-r80 and can be left as is. Even though the slot car can be seen as a toy, having a real-time Linux kernel will be beneficial. There are around 14 total branches available. To list available kernels, see listing 1.1. First, there are the mainline `bone` kernels which are aimed at the BeagleBone and feature the best support. Then there are the `ti-channel` and `ti-rt` kernels, which receive patches from Texas Instruments faster than mainline. These are required for the robot control library as they feature experimental PRU drivers not yet featured in the mainline. They could potentially have some broken drivers but are also generally faster. Third, there is the xenomai dual kernel. The BeagleBoard.org Foundation maintains patches for the ti kernels in their repository [24]. The latest 4.14 version in the ti-rt channel was version 4.14.108-ti-rt-r104, which I install using commands in listing 1.2.

Listing 1.1: Listing available kernels

```
$ sudo apt-get update
$ sudo apt-cache search linux-image | grep <branch>
```

Listing 1.2: Installing RT LTS kernel

```
$ cd /opt/scripts/tools/
$ git pull
$ sudo ./update_kernel.sh --ti-rt-channel --lts
$ sudo reboot
$ uname -a
Linux beaglebone 4.14.108-ti-rt-r104 #1 SMP PREEMPT RT Tue Apr 9
    18:48:02 UTC 2019 armv7l GNU/Linux
```

The additional packages I install are `cmake` and `ninja-build` recommended for building the nanomsg-next-generation (NNG) library and many other projects. For the `librobotcontrol` package check that after the upgrade the installed package version is still `1.0.3` using script `rc_version` and that the drivers are working with `rc_test_drivers`. Some drivers might not be loaded when running off the SD card

and need to be checked after flashing the eMMC. The MATLAB support package was created to utilize this library and that means it is necessary to make sure the library is installed correctly. In chapter 2 I will use two network socket application libraries UDP-based Data Transfer (UDT) and nanomsg-next-generation (NNG). Listings 1.3 and 1.4 show how these are built as dynamic libraries and placed in the /usr/local directories on Linux. After these files are placed, `ldconfig` must be called to register the libraries with the linker. The git repository used for UDT is a clone with makefile simply modified to allow building on the ARM architecture. UDT is otherwise hosted on SourceForge [25], NNG is on Github [26].

Listing 1.3: Building the UDT library

```
$ git clone https://gitlab.fel.cvut.cz/wernisim/udt-arm.git
$ cd udt-arm/udt4/src
$ make -e os=LINUX arch=ARM
$ sudo cp libudt.so /usr/local/lib
$ sudo cp udt.h /usr/local/include
```

Listing 1.4: Building the NNG library

```
$ git clone https://github.com/nanomsg/nng.git
$ mkdir nng/build cd nng/build
$ cmake -G Ninja -DBUILD_SHARED_LIBS=on ..
$ ninja
$ ninja install
```

Having prepared the SD card completely, the finishing touches are done on the boards after flashing the eMMC. These include calibrating the sensors and checking the drivers work using the Robot Control library (`librobotcontrol`) scripts. All the scripts are on the path and begin with the `rc_` prefix. See listing 1.5. As a final check, the BeagleBone `/opt/scripts/tools/version.sh` script can be used to verify the installed system components.

Listing 1.5: Calibrating the hardware for the Robot Control library

```
$ rc_test_drivers
$ rc_calibrate_accel
$ rc_calibrate_gyro
$ rc_test_imu
$ rc_test_motors -d 1 -m 1
```

1.4 SETTING UP THE SIMULINK ENVIRONMENT

To utilize the support package for the BeagleBone Blue by Mathworks, MATLAB needs to be run on the Windows OS. Hopefully, in the future the support will be extended to Linux, seeing that Mac OS X has been added in the recent versions [22]. Preparing the environment for work with the project will nevertheless be the same bar the used compilers and library installations. A compiler with support [27] should be picked, installed and set up using `mex -setup`. Generally, instead of using the support package provided by MATLAB for the MinGW compiler one should install Visual Studio or MinGW by hand. The package installs in an obfuscated location and is not suited for compiling outside of MATLAB's `mex` script. Next, the hardware support package [21] needs to be installed either from the file or by the addon explorer. After this from the addon manager window the package setup can be used to prepare the BeagleBone with the SD card as part of the steps in the above section 1.3. It is also good to install a Simulink desktop RT kernel by the Matlab command

`sldrtkernel -install`. Lastly, the project itself needs to be downloaded from the Git repository listed above in section 1.2. MATLAB features Git integration after Git is installed and added to the path variable. This integration works with Simulink projects with basic functionality.

The UDT and NNG libraries have specific ways to build them depending on the installed compiler. The compiler should be the same as the one set up for `MEX`. Otherwise there will be incompatibility and errors when producing `MEX` files inside MATLAB that link against these libraries [28]. NNG supports `cmake` and all of its generators [29]. I have tested it with the Ninja build tool inside the MSYS+MinGW environment and with Visual Studio 2017. As for UDT, the situation is slightly more difficult. Changes are required to build the library on 64-bit Windows. Either a new Makefile must be created or the Visual Studio solution files must be upgraded and an x64 configuration added. I have done the later in the UDT repository clone mentioned in listing 1.3. The build configuration is set in such a way that it reflects the flags which mex uses with the compiler. Those can be read from MATLAB command line by `mex.getCompilerConfigurations` inside the `Details` field of the returned structure. Notably, the /MD flag for Multi-threaded DLL runtime library, enabled C++ exceptions with extern C functions (/EHs), C7 compatible debug information format (/Z7) and others.

2 CODE GENERATION FOR DISTRIBUTED SYSTEMS

This chapter talks about the software prepared for the problem touched upon in the introduction. Through the model configuration using Simulink Coder and Embedded Coder, one may set up the Simulink scheme to generate efficient code which executes the simulation in real time on supported and general-use hardware. This extremely handy feature inside the already powerful tools of Matlab is what motivated the revision of the slot cars. The support package for BeagleBone Blue, which utilizes the Robot Control library by the designers of the BeagleBone Blue version, gave an easy to start platform for robotics use. Next was the question of managing more complex designs with multiple BeagleBone's. To explore the concepts of stability in mobile distributed systems, a larger set of distributed hardware is needed. In this scenario working with the bare tools proved to be a difficult menial task. By preparing a set of scripts that utilize the programmatic features, we would simplify the task dramatically.

2.1 SIMULINK FEATURES

To explain the process and reasoning on which the project stands, I would first give a list of tools that Matlab provides to make the task possible. The starting point is the support package for BeagleBone Blue [21], the hardware used in the slot cars. This is a complete set of simplifying tools to use code generation of Simulink schemes for the hardware effortlessly. The full build toolchain is set up including coordination with the BeagleBoard via SSH. Also, some of the hardware interfaces of the BeagleBone are provided in Simulink as a library of blocks. Using this package, a lot of the early work which would otherwise be needed was skipped. Almost to the point of not having to understand how the build process works.

Next, the Simulink project was considered for the management of the scripts and models [30]. This adds some useful features to the work and allows a cleaner and more efficient environment. Among the most welcome additions are path management, dependency analysis, Git integration, and startup customizations. Overall, it exists to keep the project healthy over a longer period of time, with which student projects often struggle. On the same note is the object-oriented programming built into Matlab. Similar to Java code, Matlab allows classes and scripts to be organized in packages. This lets the code to be well categorized and managed even further and adds modularity and extensibility via interfaces and inheritance. The classes are not meant to be used for large data manipulations and demanding operations though, and they won't be used as such. They will be used for the top level logic.

Another crucial toolset is programmatic model editing [31]. I first considered the options of model references [32]. These are still a useful technique that helps with modularity and reducing build time, but for the purpose of generating code for multiple boards, I have found them unsuitable. First there are limitations to how they can be used [33] and second, it was hard to imagine a finalized solution in which they would work for the task. So instead the choice was to manipulate the models programmatically until they were prepared to be distributed to each device. Using this approach, where each board has its own top model, helps to simplify the understanding of code execution on the target [34]. Around the time of this choice, Petr Bláha, already mentioned in chapter 1 on the history of the project, shared his solution which worked

in this way. Sadly there was some overlap with his work because of miscommunication. However, with permission, I used his work as a kick-starter.

With the tools to enable the code generation for distributed targets out of the way, the next few are connected to implementing custom functionality. There are multiple ways to do this [35]. Each with their considerations, on which I will touch upon when talking about the custom created code. Other than that, solutions outside the Simulink software, which are useful but have to be linked via custom code to use in Simulink should be mentioned. These are the UDT [25] and NNG [36] network middleware libraries I have decided to try and the Robot Control library [37], all of which I have already mentioned in chapter 1. The Robot Control library is especially important, as it interfaces most of the possible uses of the BeagleBone Blue in a simpler way. Some of the library is already available through the support package, but other times I will use it in a custom way.

This is by no means an extensive list of what is possible. There is a huge ecosystem for many types of technical fields and there are way too many tools and options to explore. For this project, I have kept to the ones most obvious and useful but there is always the possibility of extending the solution where it would prove interesting.

2.2 WORKFLOW

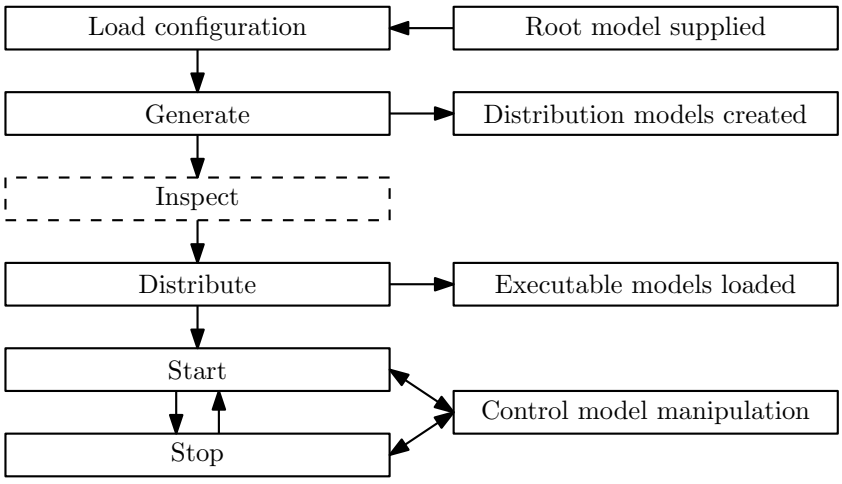


Figure 2.1: Flowchart of the script process with model interactions

The idea of working with distributed systems in Simulink, which are designed in a single model, is to prepare a set of scripts that, when called on the design model, create separate models. These are then worked with separately for the following tasks of generating code and controlling the execution of this code. This choice had the prospect of being the least problematic. For a picture of the workflow, see figure 2.1. The left column shows the scripts into which this workflow is separated and the right column shows their interactions with the Simulink model files. Each of these scripts will get their mention but first I describe the classes in which they reside.

2.2.1 Classes

To prevent the scripts from being too rigid and single-use, I have placed them in classes that also serve as storage for design choices on what the scripts should do. The main and most important class is the *Configuration* class, which is a starting point when working with the project. In this class, the options common to all parts of the process

sit, e. g. the working folder, names of models and the used communication backend. This class also defines all the scripts listed in figure 2.1 and they will be described later. The communication backend is its own class. In fact, it is an interface of which I have two implementations based on the aforementioned network middleware libraries. The backend classes define how the interface of the distributed systems is solved, more in section 2.3. Apart from this, the configuration class also holds a list of used BeagleBone boards. The class for these boards wraps the support package provided `beagleboneblue` class and it holds the login information and provides functions to manage the connection to these boards. These three classes wrap every script provided apart from some helper scripts mentioned in section 2.4 and some code not suited to the class and package format that Matlab uses. An example of how the configuration is defined is given in listing A.1 in appendix A.

2.2.2 Scripts

I will now describe the scripts in the order they are usually called which can partly be read from figure 2.1. The first in the chart is not specifically a script. It is up to the user whether to create the configuration class by hand, by script, or load from a .mat file. It is a requirement to have a configuration with all the fields defined or left at default and to have a matching root model.

GENERATE. The first script does model manipulation. It first copies the root model into the directory called `distribution`, which it creates in the configuration folder. It also creates models in this folder for each board with the content taken from a specified subsystem in the root model. After this the script calls into the specified communication backend's `createDistributionModels` script.

CREATE DISTRIBUTION MODELS. This script's task is to replace the inports and outports in the created distribution models with blocks of networking functionality. It also removes the blocks of the board subsystems from the control model after their information is parsed. Both the backend implementations work in a publisher-subscriber pattern, as that is the same as how signals in Simulink are routed between blocks, 1:N. This helped simplify the steps required to create the networking interface dramatically compared to the original work by Petr Bláha. However, there is still a lot of analysis needed for communication to be set properly. First, the model must be compiled to determine the data width of the routed signal. On the sender side, this is set dynamically, but the receiver requires the information explicitly. The script `portDetails` does this including reading the data type, which I currently lock to type double. It does so by using the `get_param('OBJ','PARAMETER')` function to read the block properties [38]. Other than that the script `getDirectConnections` ensures the connection parameters, e. g. port, and addresses, are set correctly by managing a list of information on both sides of the interface when they are both boards and not other blocks that stay in the control model.

INSPECT. Inspect is an optional script that opens all the generated models in their own windows for the user to see. This should be done a few times before one trusts the generate script to work.

DISTRIBUTE. After the previous steps are finished and the models are ready, this script is called to set important model configuration options, compile and load the models. But before that, the network connection is checked. If a board marked as crucial is unable to be connected, the script does not proceed. Then for each board, the toolchain build is sequentially executed which compiles and loads the model executable on the device. Two modes are supported in normal and external modes. Petr Bláha used a clever workaround in which after the code was generated, the compilation

happened in parallel without waiting for results. I did not maintain this variant, but in the future, it should be revised.

START / STOP. These two scripts control the execution of the distribution models as a batch. Start has the same logic as distribute where it won't start unless all crucial boards are connected. Stop script does not have this. The other part of controlling the execution is the tuning of parameters, which happens either in the control model or in one of the board models executed externally. These files are opened and closed automatically by the scripts.

OPEN SHELL. This script defined on the BeagleBoard class is worthy of final mention not because of its complexity, but for its usefulness. It is the fastest way to open a SSH window for each of the boards since the login data is stored here.

2.3 NETWORKING

The communication part of the created application is an important part of the project. While the task could be left to common UDP or transmission control protocol (TCP) blocks which Simulink provides, these blocks do not provide any good features apart from what is defined in the protocol. The features I seek are automatic congestion control for UDP, the publish-subscribe pattern, message orientation, automatic connection reestablishment and similar. The idea came from the experiences in the previous revision of slotcars, where we worked with a custom messaging implementation, and wished to instead use a standardized tool. At the time the considered tool was ZeroMQ (ØMQ). Later, the idea to implement this as a Simulink s-function was inspired by a Mathworks blog post on Co-Simulation [39]. Reading into the general opinions and comparisons on ØMQ and similar libraries I instead picked the NNG library [36] for its cleaner API and better code. The development of this library was branched from the development of ØMQ when some developers decided ØMQ needed a rewrite [40], and then rewrote it a second time [41]. The resulting library has good prospects but most importantly there is a lot of strength under a simple interface. This builds upon TCP by adding listener-dialer and publisher-subscriber paradigms for automatic one to many messaging. I use the backend based on this library when absolute reliability is desirable.

Out of interest and concern over no UDP support I have also added a communication backend based on the UDT library [25]. This, again, adds the sought features with a messaging application programming interface (API), but built on top of UDP for data-intensive tasks. It has advanced congestion control and packet multiplexing to improve the performance and reliability of the transfer [42]. It, however, does not support the publisher-subscriber pattern and is closer to traditional socket programming. I emulate the pattern in the custom implementation by using a listening server that stores its connections and pushes data to all the subscribers.

While the performance of a pure UDP approach may be faster in terms of maximum theoretical throughput, the self-awareness of UDT sockets and added mechanisms should perform much more reliably in a general scenario.

2.4 LIBRARIES FOR SLOTCARS AND OTHER TOOLS

While not directly linked to the process of generating code for distributed systems, these files are part of the project either as helper tools or to be used with the slot cars. I have prepared two model libraries that hold the sensor and actuator interfaces, simulation models, and controller examples. Their schematics are in figures 2.2 and 2.3.

The control blocks will be explained in chapter 3. There are also two libraries that hold the Simulink blocks for the communication backends. Implementation-wise, I have written both these network interfaces as C and C++ s-functions. These generally feature the highest speed and feature set out of the custom block options [43] but cannot be inlined automatically. However, using Matlab systems that evaluate c code by `coder.ceval` has been more user-friendly and cleaner in the Simulink project. Later I would reconsider moving to Matlab systems fully.

Next, to the libraries, I have shared models of experiments I have used in chapter 3 as examples. One such example is shown and listed in appendix A.

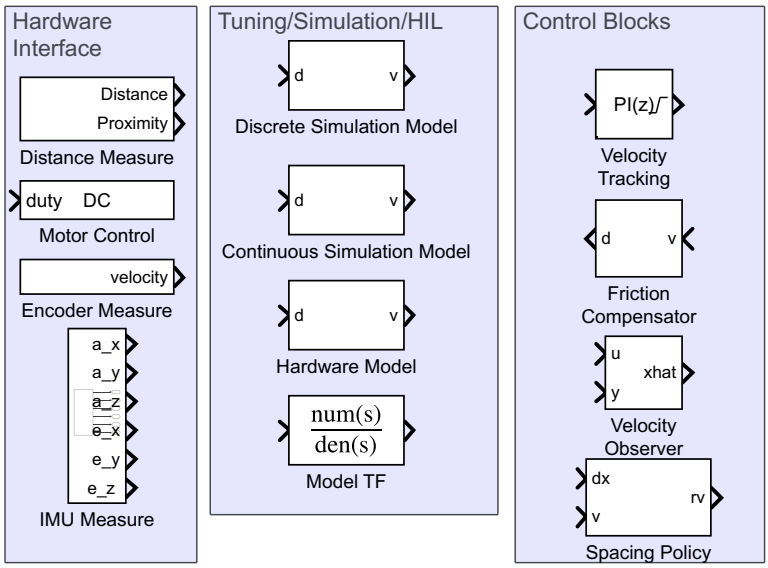


Figure 2.2: Schematic of the library with common slotcar blocks

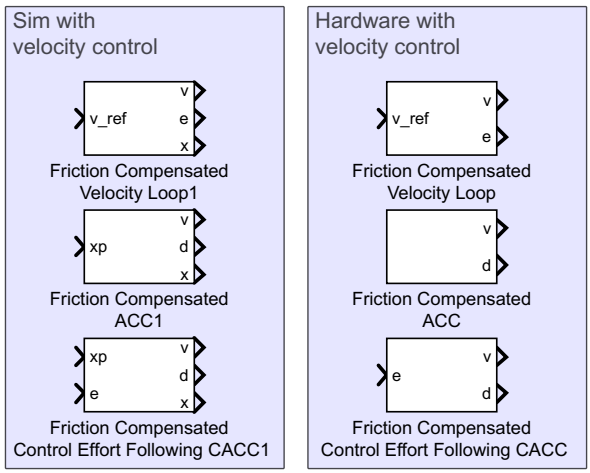


Figure 2.3: Schematic of the library with common slotcar blocks

The last files of the project are helper scripts. Currently, these are only scripts to compile mex files of the two backends' s-functions and scripts that set the temporary file folders to a work folder.

2.5 PROJECT STRUCTURE

The following folder tree shows how the files described in this chapter are structured and serves as a summary of what is presented.

```
project root - Simulink .prj file and git files
├─ cad - folder for sharing of CAD work
├─ examples - example models and configurations
├─ experiments - git excluded workplace for experiments
├─ lib - submodules
│  ├─ librobotcontrol
│  ├─ nng
│  └─ udt-arm
├─ models - model libraries
├─ src - classes scripts and s-functions
│  ├─ @BeagleBoard - board class
│  ├─ @Configuration - configuration class
│  ├─ @+comms
│  │  ├─ +common - shared code between the backends
│  │  ├─ +interface - the Backend interface class
│  │  ├─ +nng - NNG backend implementation
│  │  └─ +udt - UDT backend implementation
│  ├─ +gui - GUI wrapping the configuration class
│  ├─ +slotcar - drivers for used peripherals
│  └─ sfunctions - implementations of networking
├─ utils - helper scripts
└─ work - folder for temporary files
   ├─ cache
   └─ codegen
```

3 SLOTCAR PLATOONING EXPERIMENTS

Before the experiments can be demonstrated, the control problem must be formulated. From the perspective of a single slot car inside the platoon, the speed is regulated in such a way that it is the maximal possible while ensuring that the slot car does not crash into its predecessor. Next to speed, headway may be the controlled physical variable and it is the kept distance towards the preceding car. The smaller the headway the smaller the space for overshoot when reacting to breaking but also the larger the possible benefits towards fuel consumption and congestion. These two variables make the two control loops commonly present in designed longitudinal control laws.

The term of leader is also defined as the first vehicle of the string whose reference is set in a different way from the vehicles inside the platoon. It may brake and accelerate arbitrarily. Any signal may be propagated through the platoon under the assumption that the leader's dynamics, including unmodeled errors and measurement noises, could produce it.

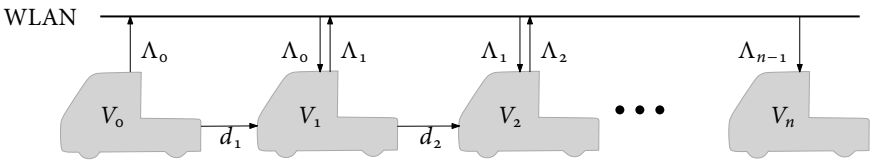


Figure 3.1: String of slotcars with a shared wireless medium

From the perspective of the platoon as a whole it is a string of vehicles with increasing complexity of dynamics in size. Under the assumption that each vehicle has similar dynamics to one another, i. e. a heterogeneous string, and that only physical propagation is present it can be shown that the last vehicle in line produces the largest reaction to the leader and overall control effort. What could be ignored in a small scale platoon may have larger effects in a platoon of modest or large sizes. For this reason, the requirement for string stability exists. It describes and limits the worst case attenuation of distance error, velocity or acceleration through the string [44]. When wireless communication of control states is included the problem solutions become more feasible. The leader's measured states may propagate to the last vehicle without any dynamics, only the delay of communications is present. This communication is the new innovation in the field and the next logical step towards safer autonomy. There exist many topologies of information flow for strings. These include predecessor-following, predecessor-leader following, two predecessor-leader following, and bidirectional [44]. Without communication, this has been demonstrated on the previous version of slotcars which had distance measurement on both front and back [10]. This chapter will additionally focus on cooperative control. On the current version of slotcars, only the front distance is measured and apart of wirelessly communicated measurements, the causality propagates from predecessor to follower only.

3.1 SLOTCAR MODEL

The slotcar is simply a physical mass moved by a DC motor. Starting from the input, the DC motor is powered by 12 V input with pulse width modulation (PWM)

Variable	Symbol	Unit
Duty cycle	$d\,(\mathrm{t})$	
Car velocity	$v_c\,(\mathrm{t})$	$m \cdot s^{-1}$
DC motor voltage	$u_m\,(\mathrm{t})$	Ω
DC motor current	$i_m\,(\mathrm{t})$	A
DC motor angular velocity	$\omega_m\,(\mathrm{t})$	$rad \cdot s^{-1}$

Table 3.1: Variables of the slotcar model

through an H-bridge. The input's actual voltage is slightly lower because of the loss on the input diode (3.1). There may also be minor fluctuations in voltage when brushes lose contact with the track. Given the PWM frequency is high enough the modulated input voltage is cut off in the frequency domain by the DC motor in a way that only the DC component remains. Numerically the input voltage is simply multiplied by the duty cycle of the PWM to obtain the regulated input voltage (3.2). Next, the motor is modeled as a resistor and back electro-magnetic field (EMF) voltage source (3.4). The inductance of the motor winding is ignored because of the fast dynamics of the motor current (compared to the car velocity) and because the current is not measured. The torque of the motor shaft is proportional to the current flowing through the motor winding and it translates into back-wheel torque. The back wheels are pushed into the track by strong magnets and are made of quality rubber. With this there is no slip of wheels and the car is pushed with the force given by the wheel torque, see equation (3.5). Equation (3.6) is the combination of equations (3.1) to (3.5) and it constitutes the controlled system from duty cycle to velocity.

$$U = U_o - U_d \tag{3.1}$$

$$u_m\,(\mathrm{t}) = Ud\,(\mathrm{t}) \tag{3.2}$$

$$v_c\,(\mathrm{t}) = nr\omega_m\,(\mathrm{t}) \tag{3.3}$$

$$Ri_m\,(\mathrm{t}) = u_m\,(\mathrm{t}) - k_m\omega_m\,(\mathrm{t}) \tag{3.4}$$

$$\frac{dv_c}{dt}\,(\mathrm{t}) = \frac{k_m}{mnr} i_m\,(\mathrm{t}) - \frac{1}{m} F_f\,(v_c\,(\mathrm{t})) \tag{3.5}$$

$$\frac{dv_c}{dt}\,(\mathrm{t}) = \frac{k_m U}{mnrR} d\,(\mathrm{t}) - \frac{{k_m}^2}{mn^2r^2R} v_c\,(\mathrm{t}) - \frac{1}{m} F_f\,(v_c\,(\mathrm{t})) \tag{3.6}$$

A fair number of simplifying assumptions have been made. That is because they pose negligible error in faultless operation. One last phenomenon present in the slotcars that cannot be ignored is friction. It is caused mostly by the car being strongly pressed into the track by its weight and the track magnets and grinding of gear and motor. It has a noticeable impact on the dynamics. In fact, it is one of two nonlinearities in the model, the other being input saturation. Especially when the car is at a halt and stiction is active. At speeds close to zero the largest nonlinearities, including discontinuity, are observable and most common friction models are not accurate or hard to use. The Continuous zero-velocity crossing model can be used to approximate friction behavior around zero velocity to remove discontinuity caused by static friction. For control purposes the static Karnopp, Armstrong and dynamic Dahl, LuGre, Bliman-Sorine, Leuven and other models exist [45][46]. The dynamic models have been shown to be more accurate while requiring a larger number of parameters in a more complex, but manageable, model [47]. Where all the friction models experience less trouble and are more accurate is at speeds further from zero. At these speeds assuming the friction consists only of Coulomb and viscous components is acceptable. Such a model is given by equation (3.8). This is sufficient for the experiments in this chapter because the goal is to regulate distance while at higher speeds. Additionally, excluding the friction part of the model, the car behaves as a linear first order system. The continuous

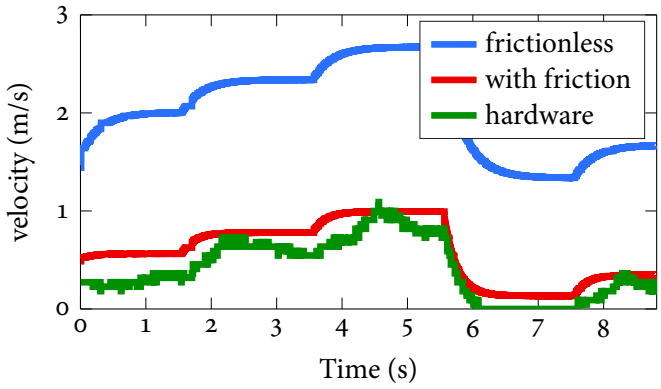


Figure 3.2: Comparison of hardware in the loop measurement and models with and without friction

transfer function (TF) of this system is in equation (3.7). It will be used for tuning of controllers.

$$\frac{V(s)}{D(s)} = \frac{KU}{mR} \frac{1}{s + \frac{K^2}{mR}} \tag{3.7}$$

Figure 3.2 shows the error of not accounting for friction. In this experiment, the duty cycle was set to predefined levels in two-second intervals and the velocity was measured with the back wheels lifted off the track to remove some of the friction. After the friction is added the model starts representing the reality much better albeit still not perfect. This non-linearity can cause issues with linear design methods. Either the velocity controller must be directly robust to this heavy model uncertainty or a friction observer and compensator must be employed. Better yet, both of these should be used at the same time. For demonstration of control purposes, which is the effort of this chapter, the first variant would suffice. Despite that I will put some effort into creating a compensator to improve the system responses in section 3.3.

$$F_f(v_c(t)) = b_d v_c(t) + b_s sign(v_c(t)) \tag{3.8}$$

The parameters used in equations (3.1) to (3.6) are listed in table 3.2. Most of these were measured directly, only the friction parameters and motor torque constant were identified or calculated. The motor constant describes both the torque constant and velocity (back EMF) constant and is calculated from the velocity constant's definition as per equation (3.9). The rating by the manufacturer which is 23 000 rpm at 12 V is plugged into the equation.

$$k_m = \frac{U_r}{\omega_r} \tag{3.9}$$

Parameter	Symbol	Value	Unit
Input voltage	U_0	12.25	V
Diode voltage drop	U_d	0.7	V
Stalled motor resistance	R	5	Ω
Motor constant	k_m	0.005	$N \cdot m \cdot A^{-1}$
Slotcar weight	m	173	g
Gear ratio	n	$9/28$	
Wheel radius	r	9	mm
Torque to linear force gain	$K = \frac{k_m}{nr}$	1.73	$N \cdot A^{-1}$
Viscous friction parameter	b_v	0.34	$kg \cdot s^{-1}$
Coulomb friction parameter	b_s	0.72	N

Table 3.2: Parameters of the slotcar model

The frictions parameters have been found by a measured experiment of controller effort with a multi-step reference of the closed velocity loop [46]. The velocity loop is described in section 3.2.

$$d(t) = \left(\frac{K}{U} - \frac{Rb_d}{KU}\right) v_c(t) - \frac{Rb_s}{KU} \operatorname{sign}(v_c(t)) \tag{3.10}$$

Transforming equations (3.6) and (3.8) in steady state where $\dot{v}_c = 0$ gives equation (3.10). By fitting the reference velocities and measured duty cycles around the tracked velocities with a line gives first-degree polynomial coefficients from which the friction parameters may be calculated. The fit is demonstrated in figure 3.3. The mean values of the duty cycles are found on different levels of the reference signal which are then fitted. The model is then validated in figure 3.2. After this the friction compensation is added in section 3.3.

3.2 CRUISE CONTROL

As mentioned in the introduction, the goal of string platooning is to control velocity and inter-vehicular distance. Controlling the motor current and acceleration is also a possibility when utilizing an ammeter and the inbuilt IMU. In fact, acceleration may be the superior choice of primary control concern over velocity because it carries the information on the dynamics of the physical system faster, being the derivative. Currently, in the project, I use velocity only because it poses less trouble as both signal and measurement.

As for the overall design of the working controller for the described problem in the introduction to this chapter, the structure is called cooperative adaptive cruise control (CACC). There are variations of CACC, but it generally may be broken down in the following fashion. The basic part of the structure is the cruise control. This is for maintaining a constant reference velocity in the presence of disturbances. Next, the adaptive part is added by measuring the distance to the vehicle in front and matching its speed to keep a set distance. Last the cooperation is added, where the car exchanges it's own measurements or input signals with other cars and uses them in a feed-forward way to improve reaction time and reference tracking properties.

For an overall picture of one such controller see figure 3.4. This scheme has three blocks representing control laws C, H, L, and the car model G defined in section 3.1. C is a controller meant to improve the reference tracking of the velocity closed loop. H is the control law of the spacing policy. This control law acts on the relative distance between the car and its predecessor and it may also vary with the speed. L is a general control law acting on some network transmitted values. Most often these values are

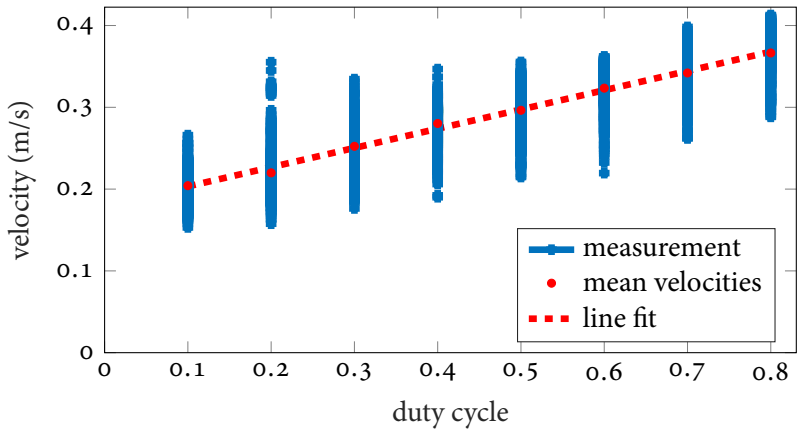


Figure 3.3: Measured controller effort (duty cycle) for different reference velocities with mean values fitted by a line

the reference signals the followed vehicles act upon or their actual measurements of inner variables like acceleration or velocity. In the case of actual vehicle traffic these will also include planned changes based on the road ahead, e. g. the distance to known traffic light stops. The control laws may be designed independently. In the end, string stability will have to be ensured for the safety of CACC systems but not at the current early stage of controller design for the slot cars.

3.2.1 Velocity tracking

The velocity loop is the basis of the slot car control. Its goal is to maintain the required speed from feedback. Most of the time a classic proportional-integral (PI) controller is used for this task for its simplicity but more modern methods have also been demonstrated in the form of model predictive control (MPC) [48][49] and H_∞ controllers [50, 51]. in the PI controller the integral term removes the steady state error caused by friction and modeling uncertainties. For a stable system like the slot car's equation (3.6) such control can be considered robust. The input saturation is present here so the controller should use some form of anti-windup. Else the car might not react to a braking reference by the spacing policy after a while of maximum effort. The input is saturated not to the range -1 to 1 specific to PWM but to -0.4 to 0.4. This comes from the thermal limits of the controlling H-bridge controller. The controller is tuned with this limitation in mind. The specific tuning requirements are not set but it is better for the slotcars to be tuned more robust than fast.

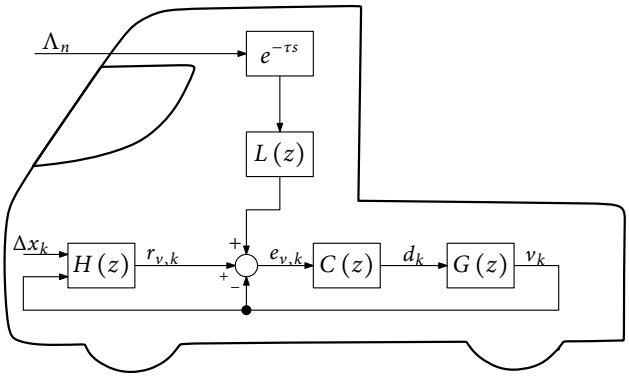


Figure 3.4: Slotcar with CACC

3.2.2 Spacing policy

The goal of the spacing policy is to define a desired distance to the predecessor vehicle and calculate the necessary speed to keep it. It is a form of a position controller where the position error is given directly by the measurement of inter-vehicle distance $\Delta x = r_x - x$. This distance should be close but with enough headway to react to immediate braking by the predecessor without crashing. The headway needed generally grows with the velocity of the vehicles and the most common policy is given by equations (3.11) and (3.12).

$$r_v(k) = r_v(k-1) + ce(k) \tag{3.11}$$
$$e(k) = \Delta x(k) - (h_o + h_v v(k)) \tag{3.12}$$

The change in velocity is given by the measured distance to the predecessor and the headway which grows with speed. This headway is set by parameters h_o and h_v and the aggressiveness of acceleration is given by constant c. This policy works in such a way that the car stays at a specific point distance to the predecessor. This means that there is constant action e because of measurement error. Another variant is given in equation (3.13) [49].

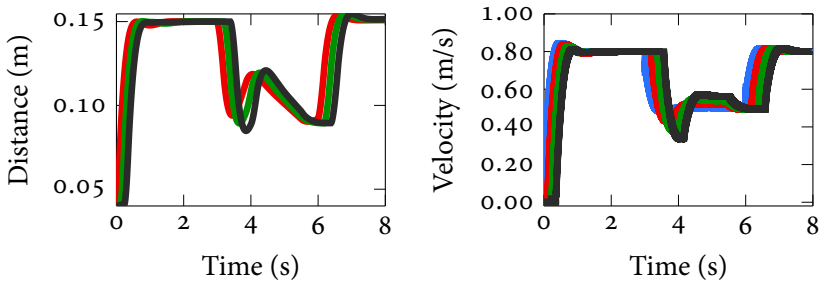
$$e(k) = \frac{c_1}{\Delta x(k) - \Delta x_{\min}} - e^{\frac{\Delta x(k) - \Delta x_{\max}}{c_2}} \tag{3.13}$$

This equation creates a distance corridor as opposed to the point in which no action is required. Only once the distance measurement falls outside of this corridor is additional action produced. The minimal corridor distance $\Delta x_{\min}$ is set using the same spacing policy as in equation (3.12). So first the speed variable headway is selected and tested to satisfy string stability and then the corridor is added larger than some multiple of the standard deviation of distance measurement error.

$$\Delta x_{\min}(k) = h_o + h_v v(k) \tag{3.14}$$
$$\Delta x_{\max}(k) = \Delta x_{\min} + d_{\text{cor}} \tag{3.15}$$
$$d_{\text{cor}} = p\sigma^2_{\Delta x} \tag{3.16}$$

In the form of equation (3.13) it is used in the optimization problem of model predictive control. For an explicit policy, I will adopt only the idea of the dead-zone and the coefficients. The spacing policy I use is given by equations (3.11) and (3.14) to (3.17).

$$e(k) = \begin{cases} \Delta x(k) - \Delta x_{\min}(k) & \Delta x(k) \le \Delta x_{\min}(k) \\ \Delta x(k) - \Delta x_{\max}(k) & \Delta x(k) \ge \Delta x_{\max}(k) \\ 0 & \text{otherwise} \end{cases} \tag{3.17}$$



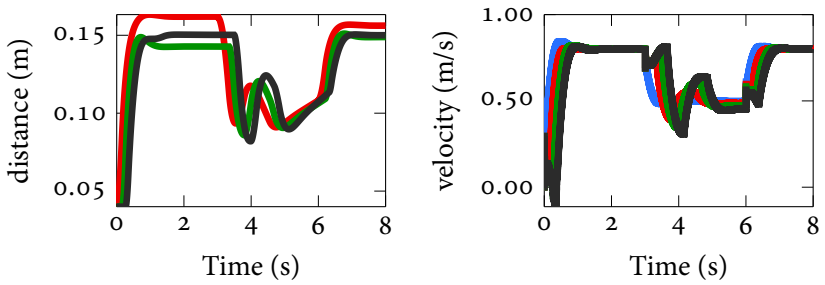
(a) Distance **(b)** Velocity

Figure 3.5: Simulated ACC string of 4 cars

3.2.3 Wireless feedforward

The first thing to note about this feedforward path of wirelessly transmitted signals is that the communication poses many issues. These cumulate into two things. First the random delay with time changing properties. This is assumed to be nonexistent when designing the control law and then acknowledged as causing some error. Then there is the possibility of a broken communication link, which may or may not recover. In such a situation the controller must detect this break and switch to a different control law. This is, for example, the nulling of the feedforward output to return to the ACC controller or degrading the CACC to estimate the lost network transmitted signal when possible [52]. I will assume the simpler former variant. In case of not receiving any packets for some duration, the path is disabled.

Out of the topologies already mentioned in the introductory part to this chapter I consider leader following in the communications without delving into the analysis of string stability. The leader transmits its immediate intention to move - the control error of the velocity loop. This control error is then added to the control error of the receiving vehicle. In another variant, the leader transmits its measured velocity for the popularity of stopping the slotcars by hand when demonstrating the platform. In this early stage of controller development, a simpler approach will do to demonstrate the ability to include wireless communications in the controller design. Otherwise, these cooperative paths without any filtering do not benefit the controller. No specific error propagation is suppressed and some undesirable effects occur. This can be seen when comparing the simulations in figures 3.5 and 3.6. The CACC system produces a slightly faster response at the last vehicle but at the cost of more perturbation



(a) Distance **(b)** Velocity

Figure 3.6: Simulated CACC string of 4 cars

3.3 FRICTION COMPENSATION

In section 3.1 a model for friction was selected and verified. Now a friction observer can be added to the controller. This task is separate from the CACC control problem, only the velocity loop with the PI control is adopted from the previous section. Effective friction compensation relies on velocity measurement with good resolution and small time delay [45] and a good model. For this reason, I expect the compensator to not be robust on the slotcars at all. When paired with the velocity loop, however, which uses the integral term to compensate for uncertainties, the compensator should improve properties.

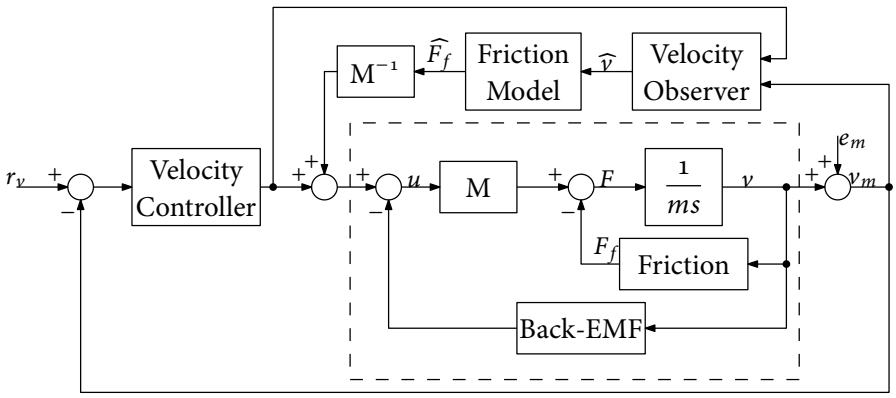


Figure 3.7: Block diagram of friction compensation

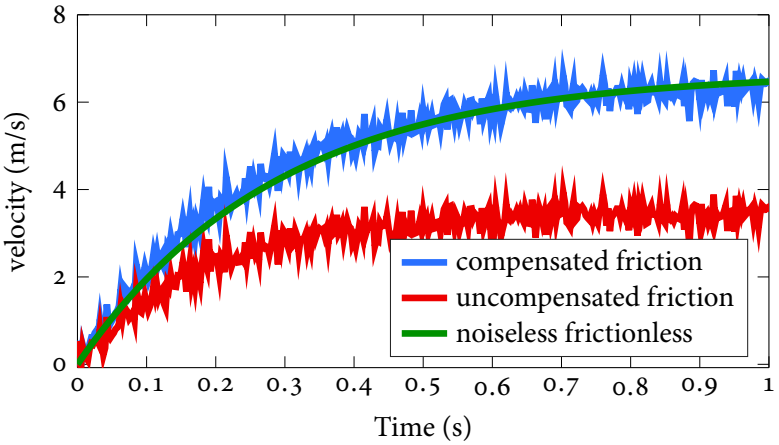


Figure 3.8: Simulated comparison of step response of slotcar models with friction and Gaussian measurement noise to the linear transfer function

The scheme of the compensator is in figure 3.7. The idea is to create a friction observer whose output is added to the plant input as to compensate for the actual friction or at least a part of it [45, 53]. Figure 3.8 demonstrates the benefit of a model perfect feedback friction observer in a direct step reference tracking situation. The model with friction, measurement noise, and friction compensation follows the frictionless linear model unlike the one without compensation.

Since the friction model gives the dependence of friction force on velocity, only the velocity estimate is needed. To reduce the measurement noise and in the future to incorporate acceleration measurements a Kalman filter with the linear car model can be used. The inputs to the Kalman filter are the velocity measurements and the system input before friction compensation is added. From the estimate of velocity, the estimate of the friction force is calculated and then gain is applied. This gain is set such that when the friction force is substituted into the duty cycle in equation (3.6) the two terms cancel each other. Because the friction estimate might not be very good in practice the gain is lowered as it is better to undercompensate than to overcompensate since the later may lead to instability [45]. However, the experiments performed show that the velocity observer in this form performs well as the simulation suggested.

3.4 SAMPLING RATES

As part of the controller design, I shortly consider each of the sampling rates. The car dynamics, given as a TF in equation (3.7), has a frequency cutoff given by the magnitude of its pole. As per the Nyquist theorem, the sampling rate should be safely

larger with a minimum ratio of 2 than this frequency or some higher frequency which is dampened enough to be considered a marginal error, e. g. at the $-20\,dB$ level. Good engineering practice is to select an even smaller sampling rate which is ten times the frequency or more depending on the application.

The rate of networked messages is a different problem altogether which is too variable to solve easily, in a stochastic time variant fashion [54]. Some talk about this was presented in sections 2.3 and 3.2.3. The controller uses buffers to synchronize and change the rate of the signal transmitted over the network and should have a strategy when the buffers are empty. Otherwise, the rate should be as high as possible without experiencing an over-utilized communication channel.

The last sampling rate which should be noted is the sampling rate of velocity measurement. By this, I do not mean the rate of the microcontroller sampling the encoder signals. I mean the rate used to calculate velocity by differentiating the change of encoder's tick count and the period. There are two approaches with different precision at different speeds. At high speeds, the simpler constant sampling time is preferable with good accuracy while at low speeds a constant position is sometimes used with variable sampling time as the measure between ticks of the encoder. I consider the first approach only. The longer the sampling period the better the accuracy but at the cost of delay of the measurement. It is somewhat analogous to filtering. The larger the contribution of past data in a filter, e. g. the window size of a moving mean filter, the more lag is added and bandwidth is limited. Such delay is usually not mentioned to not complicate the control problem but it may even destabilize the designed control loop when too large. Considering the control loop may be robust to sensor noise or a model-based estimator may be employed it is better to pick the noisier variant with less lag. Of course in another path where the added lag does not play much role, the velocity can be filtered. This can be for example when transmitting the values over a network whose delay will overshadow the smaller one.

3.5 EXPERIMENTS

PID controller		spacing policy		sampling rates	
k_p	0.20	h_o	0.04	fundamental	400 Hz
k_i	2.15	h_d	0.05	communication	100 Hz
k_d	0	d_{cor}	0.03	encoder	200 Hz
		c	100		

Table 3.3: Selected parameters of the CACC controller for experiments

In the sections above a designed controller has been created which incorporates friction compensation and CACC and in this section I will show them through experiments with slotcars. More than the validity of the designed control I will show the schemes from which the experiments were run. At the time of these experiments, I had four slotcars prepared and planned for. Two of which were working previously, one I completed by adding power backup and one I built myself. One of them showed a faulty motor drive however and so the experiments were done with three slotcars. Table 3.3 lists the coefficients selected for the controller described in section 3.2.

The first experiment is shown in figure 3.9. This one shows the validity of the spacing policy against stationary objects. At full speed, the car stops in time before crashing and backs away into the required distance. This is a basic requirement for the spacing policy. In the case of detecting an object while turning which will be evaded, the car slows down or comes to a halt and then proceeds if there is enough space.

The second experiment is simply driving the platoon by cycling through reference speeds for the leader. Here because of measurement error, the cars occasionally crash

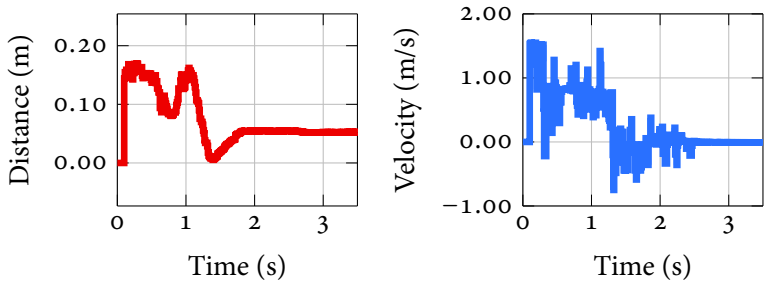


Figure 3.9: Testing ACC controller when driving into a wall at maximum speed

one another. In this way, the controller is not robust. For the Simulink schematics used for these experiments please see appendix A in which I show all the block diagrams, the used configuration script.

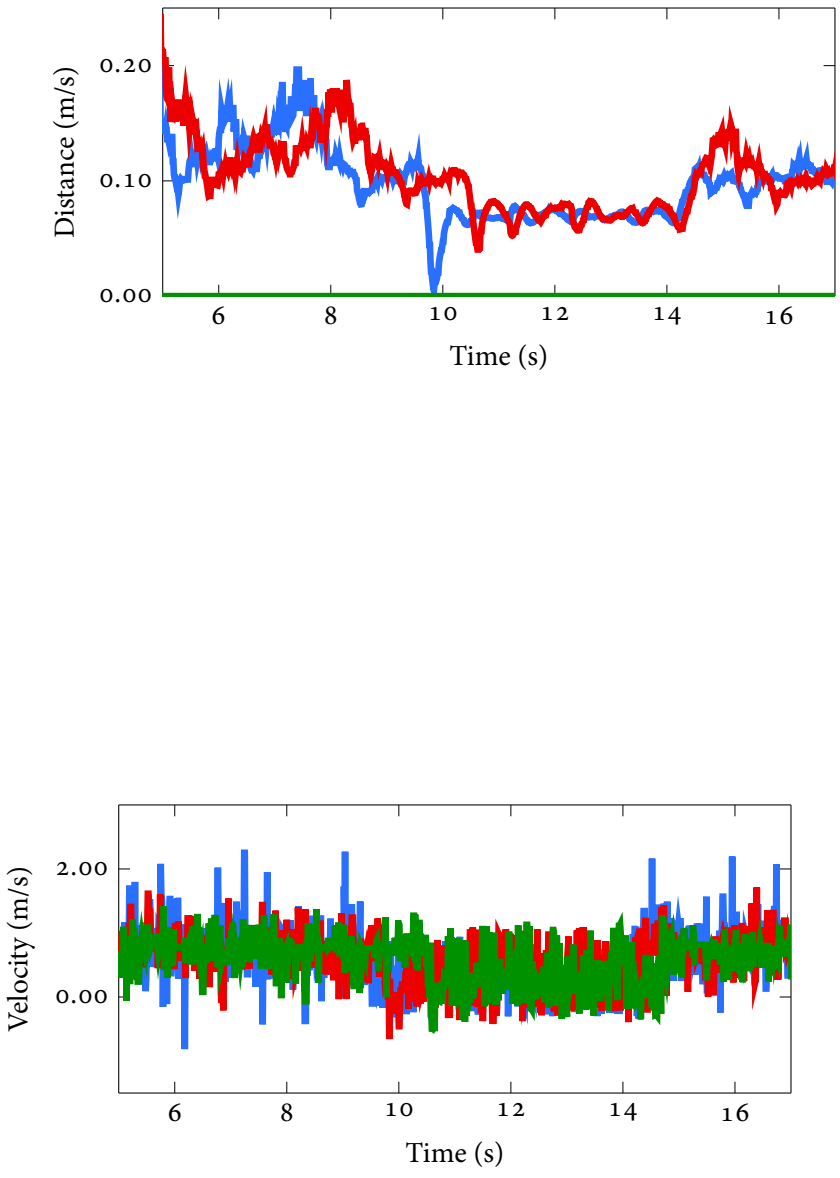


Figure 3.10: Testing CACC controller with 3 slotcars

4 CONCLUSION

The goal of the work in this thesis was to prepare a workflow for code generation for distributed systems and to demonstrate this using the slot cars which were prepared by previous students. I have prepared an extensible Simulink project for this task which includes both the classes and scripts for manipulating models of distributed systems and the drivers and model files for slot car control via Simulink. This allows utilizing model-based project management of Simulink software. Modest use of object-oriented programming features in Matlab gives the code readability, additional extensibility, and some modularity. In connection to the development of custom functionality, I have also explored the options of including custom algorithms as part of the code generation process and the options of incorporating networking application cleverness into the distributed system's communication segment. Two example networking solutions using middleware both on top of the TCP and UDP protocols have been tested.

The added value of this project over the bare tools provided by the Mathworks ecosystem is in the utilization of the programmatic approach to model manipulation and in adding communication middleware which serves as the backbone of the distributed application. This gives a serviceable alternative to manual manipulation which in the case of working with multiple hardware pieces at once becomes difficult. While the possibilities that Mathworks provides could not be fully explored and not all parts of the solution could be made problem-free, a stable workflow has been created to allow design and validation of complex distributed systems.

For the purpose of slot car control, I have also prepared a model with identified parameters including the non-linear friction phenomenon. While not perfect, the model has been shown to perform reasonably well. This was demonstrated by experiments with basic controllers of the CACC structure which include networking. This should complete the early controller development for the slot cars with better sensing and more robust control to follow. As part of the project, model libraries have been created which hold the blocks used in the experiments for future use.

In the current state, the hardware and software for the third slot car revision have come together and hopefully become transparent enough for interested parties to try out. While some of the features are not as polished as on the previous slot cars, the modularity and relative simplicity this version provide both software and hardware allow more improvements to come. These improvements should include better sensor and actuator hardware and fixes to software issues that went under the radar or were hardly solvable in the releases of utilized tools available at the time.

A VISUAL DEMONSTRATION OF THE PROJECT WORKFLOW

In the first appendix I show some of the Simulink schemes used and how the experiments were run. As stated in the chapter 2 the base of defining an experiment is the *Configuration* class. I define this by running a script in which the properties are set.

Listing A.1: Example of defining the Configuration class

```
%% define experiment options
conf = Configuration;
conf.Folder = pwd;
conf.RootModel = 'drive4CACC';
conf.MatlabIpv4 = '192.168.88.11';
conf.CommsBackend = comms.udt.UdtBackend;
conf.CommsBackend.SampleTime = 0.01;
conf.ParallelCompilation = false;

%% define used boards
conf.Boards(1).Ipv4 = '192.168.88.19';
conf.Boards(1).ModelName = 'M1';
conf.Boards(1).External = false;
conf.Boards(1).Crucial = true;

conf.Boards(2).Ipv4 = '192.168.88.20';
conf.Boards(2).ModelName = 'M2';
conf.Boards(2).External = false;
conf.Boards(2).Crucial = false;

conf.Boards(3).Ipv4 = '192.168.88.15';
conf.Boards(3).ModelName = 'M3';
conf.Boards(3).External = false;
conf.Boards(3).Crucial = false;

conf.Boards(4).Ipv4 = '192.168.88.18';
 conf.Boards(4).ModelName = 'M2';
conf.Boards(4).External = false;
conf.Boards(4).Crucial = false;
```

Tied to this configuration is the root model sitting in the same folder or in the folder specified. For the CACC experiment this model is the block schematic in figure A.1. For this experiment the driving reference of the leader is generated in the leader car and the PC model serves for logging and viewing of received data. When the speed is to be interactive, I drag the generating block left outside the subsystem. After that the generated control scheme serves also as tunable user input. Calling `gui.default\_gui(conf)` opens the GUI which binds to the configuration, see figure A.2. From this the workflow can be initiated.

After the generate script is run, separate model files are created in the distribution folder. The control model looks the same as figure A.1 but with the insides of the areas replaced by communication blocks. That is why in the design model the lines were muxed and then demuxed, to keep both data streams in a single connection. As for the

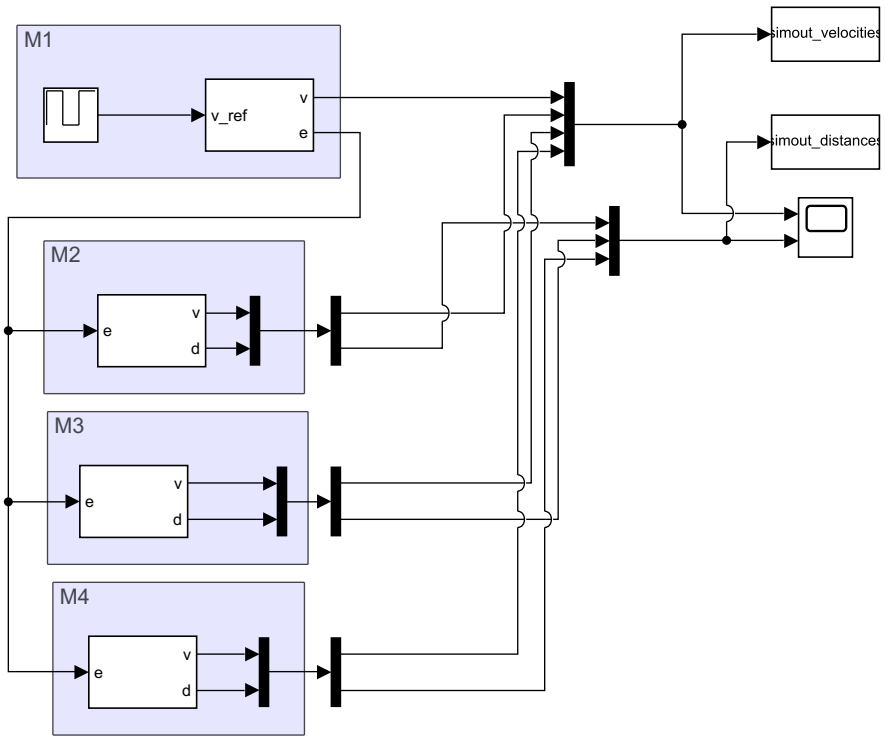


Figure A.1: Simulink block diagram of drive4CACC

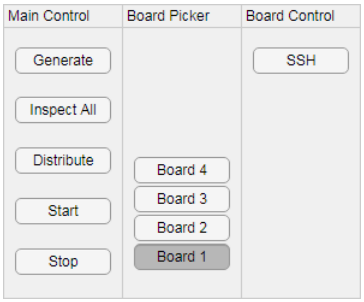


Figure A.2: Gui for the experiment

created models for each board, the top block diagram is in figure A.3 for the second board. This diagram has communication blocks with matching parameters to the other side, the control scheme. Inside the CACC block is the schematic of the controller, figure A.4. Inside this schematic are the controllers and the blocks implementing the hardware interface. Outside of the target for generated code these blocks do nothing so it is safe to leave them unused in the control scheme after removing a board from the configuration.

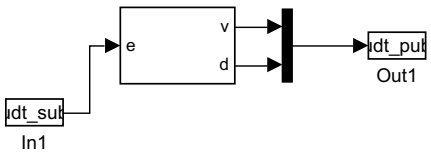


Figure A.3: Top scheme running in the slotcar

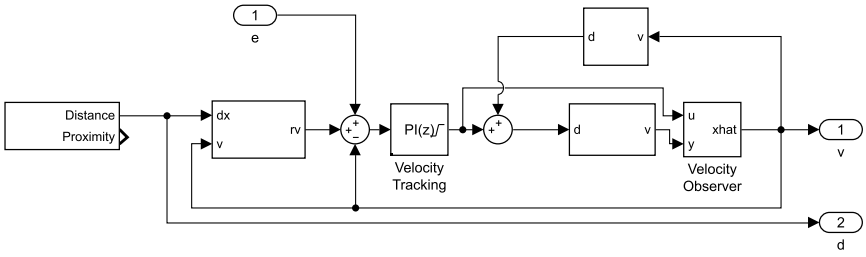


Figure A.4: Inner scheme implementing the CACC controller

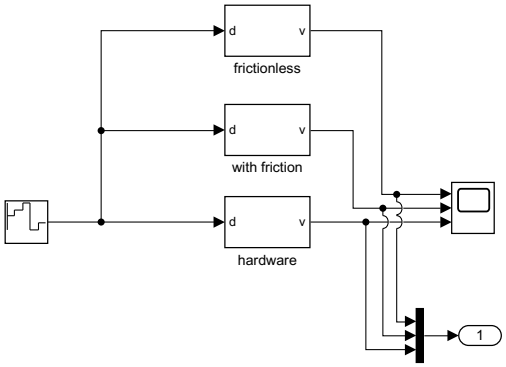


Figure A.5: Scheme running on the board for model verification

The other experiment I wish to show is the one used for model verification, figure A.5, which was used in section 3.1. In this model running on the slotcar the hardware interface and the two simulation models are running side-by-side. The results are then viewed directly in the scope when running external mode but also they are sent to the control model for viewing and logging.

B UNUSED IMPROVEMENTS AND IDEAS

In this appendix I wish to mention some things that would not fit into the main text. Perhaps to inspire or motivate some future changes and fixes to the platform.

OVERHEATING. First thing concerns the hardware oversight of combining the slot-car racing motor with the Toshiba `TB6612FNG` H-bridge integrated chip (IC) present on the BeagleBone. What happens is that after a while of riding the motors stop. From the data sheet of the IC it is clear that it is not rated for operation with a relatively low resistance racing motor at the higher voltages. The chip's thermal protection turns off functions before it blows up. Currently I have set a limit on the duty cycle the controller is allowed to use to 40 %. This does put acceptable ranges on the IC. The motor circuitry should be revised a little in the future. That brings me to a complementary thought. That is to put a resistor in series with the motor and place voltage measurement on it to determine the motor current. This should not detriment the velocity of the car because limits have to be placed on the motor anyway to help dimension the motor into the 1.2 *A* average that the IC can handle. Instead it could allow some more intelligent switching or control. In fact this is a common procedure in more sophisticated H-bridge chips. Or just replace the H-bridge in the next revision altogether.

BOARD ACCESSIBILITY. Make holes to be able to click the power and reset buttons on the BB. The BeagleBone has startup on receiving power so the buttons do not have to be clicked but when the contact is lost for a little bit and the power backup condensators do not drain fully then the board remains in a battery monitoring state and waits for the user to click the power button. The chassis hides the buttons completely and so I made larger circular cutouts into the chassis where the two buttons are so they can be pressed with a straightened paper clip or a similar, even thicker object. This change is prepared in the 3D work for the next print. Also under the keyword accessibility is the BeagleBone's NoC power saving feature. After a while of inactivity the antenna goes into such a low powered mode that the router must be moved in a 10 *cm* distance for connection to be remade and the NoC to go full power. This happens even with power management turned off.

INERTIAL MEASUREMENT. The matlab driver for the IMU is now rewritten to utilize the Robot Control library and fully working. I did not incorporate it into the platooning experiments but acceleration and perhaps even the angular velocity (steering) do have a place in the control strategies. The measurement could be used to improve velocity measurement by sensor fusion, e. g. Kalman filter, or to implement acceleration control.

RELIABLE POWER COLLECTION. The brushes on the slotcar base have not been particularly sturdy or contact reliable. However it is simple to remove them from the holder and replace them because they are not soldered or glued on. Getting copper braids with a larger width should help because the current ones are so thin that they can get pressed into the gap between the power rails and lose contact.

ZIGBEE. Where wireless networking is concerned, `802.11` is the most popular family of technologies but all of them have quirks which may be unfavorable sometimes.

One alternative of interest is the ZigBee technology. Without running into details, the advantage of ZigBee over Wi-Fi is the use of ad-hoc mesh type topology which does not really work well in Wi-Fi. It also has some differences in the physical layer which might make it more suitable. ZigBee has been explored as a choice of physical layer for meshed applications and it seems to be robust [55]. Products like the XBee have many variants and extensions for robotics projects so one that would fit into the slotcar should not be difficult to find.

BIBLIOGRAPHY

[1] Todd Litman. *Implications for Transport Planning*. Victoria Transport Policy Institute, p. 39 (cit. on p. xvii).

[2] Charlie Hewitt et al. "Assessing Public Perception of Self-Driving Cars: The Autonomous Vehicle Acceptance Model". In: *Proceedings of the 24th International Conference on Intelligent User Interfaces - IUI '19*. The 24th International Conference. Marina del Ray, California: ACM Press, 2019, pp. 518–527. ISBN: 978-1-4503-6272-6. DOI: 10.1145/3301275.3302268. URL: http://dl.acm.org/citation.cfm?doid=3301275.3302268 (visited on 04/29/2019) (cit. on p. xvii).

[3] Steven E Shladover, Christopher Nowakowski, and Xiao-Yun Lu. *Using Cooperative Adaptive Cruise Control (CACC) to Form High-Performance Vehicle Streams*, p. 51 (cit. on p. xvii).

[4] *BeagleBone Blue Support from Simulink Coder*. URL: https://www.mathworks.com/hardware-support/beaglebone-blue.html (visited on 04/29/2019) (cit. on p. xvii).

[5] *Advanced Algorithms for Control and Communications*. URL: http://aa4cc.dce.fel.cvut.cz/ (cit. on p. 1).

[6] *Distributed Control of Spatially Distributed Systems*. URL: http://aa4cc.dce.fel.cvut.cz/content/distributed-control-spatially-distributed-systems (cit. on p. 1).

[7] Dan Martinec and Zdenek Hurak. "Vehicular Platooning Experiments with LEGO MINDSTORMS NXT". In: *2011 IEEE International Conference on Control Applications (CCA)*. Control (MSC). Denver, CO, USA: IEEE, Sept. 2011, pp. 927–932. ISBN: 978-1-4577-1062-9. DOI: 10.1109/CCA.2011.6044393. URL: http://ieeexplore.ieee.org/document/6044393/ (visited on 04/29/2019) (cit. on p. 1).

[8] Dan Martinec, Michael Sebek, and Zdenek Hurak. "Vehicular Platooning Experiments with Racing Slot Cars". In: *2012 IEEE International Conference on Control Applications*. 2012 IEEE International Conference on Control Applications (CCA). Dubrovnik, Croatia: IEEE, Oct. 2012, pp. 166–171. ISBN: 978-1-4673-4505-7 978-1-4673-4503-3 978-1-4673-4504-0. DOI: 10.1109/CCA.2012.6402709. URL: http://ieeexplore.ieee.org/document/6402709/ (visited on 04/29/2019) (cit. on p. 1).

[9] Martin Lád, Ivo Herman, and Zdeněk Hurák. "Vehicular Platooning Experiments Using Autonomous Slot Cars". In: *IFAC-PapersOnLine*. 20th IFAC World Congress 50.1 (July 1, 2017), pp. 12596–12603. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2017.08.2201. URL: http://www.sciencedirect.com/science/article/pii/S2405896317328719 (visited on 05/24/2019) (cit. on p. 1).

[10] Martin Lád. "Experimental Slotcar-Based Platform for Distributed Control of Vehicular Platoons". master's thesis. May 26, 2017. URL: https://dspace.cvut.cz/handle/10467/68381 (visited on 05/05/2019) (cit. on pp. 1, 13).

[11] Bečka Marek. "Návrh a realizace řídicího systému pro autonomní autodráhové autíčko určené pro experimentování s inteligentními konvoji". bachelor thesis. Praha: ČVUT, June 12, 2018. URL: https://dspace.cvut.cz/handle/10467/76622 (visited on 04/30/2019) (cit. on pp. 1, 2).

[12] Petr Bláha. "Generování kódu pro distribuované řízení z modelů v Simulinku". bachelor thesis. Praha: ČVUT, Jan. 29, 2019. URL: https://dspace.cvut.cz/handle/10467/81244 (visited on 04/30/2019) (cit. on p. 1).

[13] *BeagleBoard.Org BeagleBone Blue: BeagleBone Optimized for Educational Robotics - Beagleboard/Beaglebone-Blue*. BeagleBoard.org, Apr. 28, 2019. URL: https://github.com/beagleboard/beaglebone-blue (visited on 05/02/2019) (cit. on pp. 2, 3).

[14] Jason Kridner. "Using the BeagleBone Real-Time Microcontrollers". URL: https://beagleboard.org/static/presentations/MakerFaireNY20140920_UsingBeagleBoneRealTimeMicrocontrollers.pdf (visited on 05/02/2019) (cit. on p. 3).

[15] *Case Studies - Introduction*. URL: https://markayoder.github.io/PRUCookbook/01case/case.html (visited on 05/02/2019) (cit. on p. 3).

[16] *Ti AM33XX PRUSSv2 - eLinux.Org*. URL: https://elinux.org/Ti_AM33XX_PRUSSv2 (visited on 05/02/2019) (cit. on p. 3).

[17] Adrián Rodríguez-Panes, Juan Claver, and Ana María Camacho. "The Influence of Manufacturing Parameters on the Mechanical Behaviour of PLA and ABS Pieces Manufactured by FDM: A Comparative Analysis". In: *Materials* 11.8 (Aug. 1, 2018). ISSN: 1996-1944. DOI: 10.3390/ma11081333. pmid: 30071663. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6119930/ (visited on 05/02/2019) (cit. on p. 3).

[18] *Aa4cc/Slotcar*. AA4CC - Advanced Algorithms for Control and Communications, May 6, 2019. URL: https://github.com/aa4cc/slotcar (visited on 05/12/2019) (cit. on p. 3).

[19] *BeagleBoard.Org - Latest-Images*. URL: http://beagleboard.org/latest-images (visited on 05/02/2019) (cit. on p. 3).

[20] *Beagleboard:BeagleBoneBlack Debian - eLinux.Org*. URL: https://elinux.org/Beagleboard:BeagleBoneBlack_Debian (visited on 04/28/2019) (cit. on p. 3).

[21] *Simulink Coder Support Package for BeagleBone Blue Hardware - File Exchange - MATLAB Central*. URL: https://www.mathworks.com/matlabcentral/fileexchange/64925 (visited on 04/01/2019) (cit. on pp. 3, 5, 7).

[22] *Release Notes for Simulink Coder Support Package for BeagleBone Blue Hardware - MATLAB & Simulink*. URL: https://www.mathworks.com/help/supportpkg/beagleboneblue/release-notes.html (visited on 05/02/2019) (cit. on pp. 3, 5).

[23] *balenaEtcher - Home*. URL: https://www.balena.io (visited on 05/02/2019) (cit. on p. 3).

[24] *BeagleBoard.Org*. URL: https://github.com/beagleboard (visited on 05/02/2019) (cit. on p. 4).

[25] *UDT: Breaking the Data Transfer Bottleneck*. URL: http://udt.sourceforge.net/doc.html (visited on 04/17/2019) (cit. on pp. 5, 8, 10).

[26] *NNG - Nanomsg-NG*. URL: https://nanomsg.github.io/nng/ (visited on 05/04/2019) (cit. on p. 5).

[27] *Compilers*. URL: https://www.mathworks.com/support/requirements/supported-compilers.html (visited on 05/03/2019) (cit. on p. 5).

[28] *Interoperability of Libraries Created by Different Compiler Brands | MinGW*. URL: http://mingw.org/wiki/Interoperability_of_Libraries_Created_by_Different_Compiler_Brands (visited on 05/12/2019) (cit. on p. 6).

[29] *Cmake-Generators(7) — CMake 3.0.2 Documentation*. URL: https://cmake.org/cmake/help/v3.0/manual/cmake-generators.7.html (visited on 05/05/2019) (cit. on p. 6).

[30] *Project Management - MATLAB & Simulink*. URL: https://www.mathworks.com/help/simulink/project-management.html?s_tid=CRUX_lftnav (visited on 05/08/2019) (cit. on p. 7).

[31] *Programmatic Model Editing - MATLAB & Simulink*. URL: https://www.mathworks.com/help/simulink/programmatic-modeling.html?s_tid=CRUX_lftnav (visited on 05/11/2019) (cit. on p. 7).

[32] *Model References - MATLAB & Simulink*. URL: https://www.mathworks.com/help/simulink/model-reference.html (visited on 05/23/2019) (cit. on p. 7).

[33] *Model Reference Requirements and Limitations - MATLAB & Simulink*. URL: https://www.mathworks.com/help/simulink/ug/model-referencing-limitations.html (visited on 05/23/2019) (cit. on p. 7).

[34] *Design Models for Generated Embedded Code Deployment - MATLAB & Simulink*. URL: https://www.mathworks.com/help/releases/R2018b/ecoder/ug/design-models-for-generated-embedded-code-deployment.html (visited on 05/16/2019) (cit. on p. 7).

[35] *Implement New Algorithm - MATLAB & Simulink*. URL: https://www.mathworks.com/help/simulink/implement-new-algorithm.html?s_tid=CRUX_lftnav (visited on 05/07/2019) (cit. on p. 8).

[36] *Nanomsg-next-Generation – Light-Weight Brokerless Messaging: Nanomsg/Nng*. nanomsg, Mar. 10, 2019. URL: https://github.com/nanomsg/nng (visited on 03/11/2019) (cit. on pp. 8, 10).

[37] *Robot Control Library*. URL: http://strawsondesign.com/docs/librobotcontrol/manual.html (visited on 04/24/2019) (cit. on p. 8).

[38] *Common Block Properties - MATLAB & Simulink*. URL: https://www.mathworks.com/help/simulink/slref/common-block-parameters.html (visited on 04/28/2019) (cit. on p. 9).

[39] *Examples of Co-Simulation with Simulink*. MathWorks, Oct. 30, 2018. URL: https://github.com/mathworks/SimulinkCoSimulationExample (visited on 03/11/2019) (cit. on p. 10).

[40] *Differences between Nanomsg and ZeroMQ*. URL: https://nanomsg.org/documentation-zeromq.html (visited on 05/24/2019) (cit. on p. 10).

[41] *Rationale: Or Why Am I Bothering to Rewrite Nanomsg?* URL: https://nanomsg.github.io/nng/RATIONALE.html (visited on 05/24/2019) (cit. on p. 10).

[42] Yunhong Gu and R.L. Grossman. "Supporting Configurable Congestion Control in Data Transport Services". In: *ACM/IEEE SC 2005 Conference (SC'05)*. ACM/IEEE SC 2005 Conference (SC'05). Seattle, WA, USA: IEEE, 2005, pp. 31–31. ISBN: 978-1-59593-061-3. DOI: 10.1109/SC.2005.68. URL: http://ieeexplore.ieee.org/document/1559983/ (visited on 05/24/2019) (cit. on p. 10).

[43] *Comparison of Custom Block Functionality - MATLAB & Simulink*. URL: https://www.mathworks.com/help/simulink/ug/comparison-of-custom-block-functionality.html (visited on 03/11/2019) (cit. on p. 11).

[44] Ziran Wang, Guoyuan Wu, and Matthew J. Barth. "A Review on Cooperative Adaptive Cruise Control (CACC) Systems: Architectures, Controls, and Applications". In: *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. 2018 21st International Conference on Intelligent Transportation Systems (ITSC). Maui, HI: IEEE, Nov. 2018, pp. 2884–2891. ISBN: 978-1-72810-321-1 978-1-72810-323-5. DOI: 10.1109/ITSC.2018.8569947. URL: https://ieeexplore.ieee.org/document/8569947/ (visited on 05/18/2019) (cit. on p. 13).

[45] H. Olsson et al. "Friction Models and Friction Compensation". In: *European Journal of Control* 4.3 (Jan. 1, 1998), pp. 176–195. ISSN: 0947-3580. DOI: 10.1016/S0947-3580(98)70113-X. URL: http://www.sciencedirect.com/science/article/pii/S094735809870113X (visited on 05/18/2019) (cit. on pp. 14, 20, 21).

[46] Claudiu Iurian et al. "Identification of a System with Dry Friction". In: (Sept. 2005). URL: https://upcommons.upc.edu/handle/2117/511 (visited on 05/13/2019) (cit. on pp. 14, 16).

[47] V van Geffen. *A Study of Friction Models and Friction Compensation*. Technische Universiteit Eindhoven, Dec. 2009, p. 24 (cit. on p. 14).

[48] Ellen van Nunen et al. "Robust Model Predictive Cooperative Adaptive Cruise Control Subject to V2V Impairments". In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. 2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC). Yokohama: IEEE, Oct. 2017, pp. 1–8. ISBN: 978-1-5386-1526-3. DOI: 10.1109/ITSC.2017.8317758. URL: http://ieeexplore.ieee.org/document/8317758/ (visited on 05/18/2019) (cit. on p. 17).

[49] Roman Schmied et al. "Nonlinear MPC for Emission Efficient Cooperative Adaptive Cruise Control". In: *IFAC-PapersOnLine* 48.23 (2015), pp. 160–165. ISSN: 24058963. DOI: 10.1016/j.ifacol.2015.11.277. URL: https://linkinghub.elsevier.com/retrieve/pii/S2405896315025616 (visited on 04/30/2019) (cit. on pp. 17, 18).

[50] Yang Zheng et al. "Platooning of Connected Vehicles with Undirected Topologies: Robustness Analysis and Distributed H-Infinity Controller Synthesis". In: (Nov. 4, 2016). arXiv: 1611.01412 [math]. URL: http://arxiv.org/abs/1611.01412 (visited on 05/18/2019) (cit. on p. 17).

[51] E. Kayacan. "Multiobjective $_{\text{infty}}$ Control for String Stability of Cooperative Adaptive Cruise Control Systems". In: *IEEE Transactions on Intelligent Vehicles* 2.1 (Mar. 2017), pp. 52–61. ISSN: 2379-8904. DOI: 10.1109/TIV.2017.2708607 (cit. on p. 17).

[52] Jeroen Ploeg et al. "Graceful Degradation of Cooperative Adaptive Cruise Control". In: *IEEE Transactions on Intelligent Transportation Systems* 16.1 (Feb. 2015), pp. 488–497. ISSN: 1524-9050, 1558-0016. DOI: 10.1109/TITS.2014.2349498. URL: http://ieeexplore.ieee.org/document/6907977/ (visited on 05/20/2019) (cit. on p. 19).

[53] C. Canudas de Wit et al. "A New Model for Control of Systems with Friction". In: *IEEE Transactions on Automatic Control* 40.3 (Mar. 1995), pp. 419–425. ISSN: 0018-9286. DOI: 10.1109/9.376053 (cit. on p. 21).

[54] Johan Nilsson. "Real-Time Control Systems with Delays". Department of Automatic Control, Lund Institute of Technology (LTH), 1998. URL: http://portal.research.lu.se/portal/en/publications/realtime-control-systems-with-delays(a7fa0a2d-09ac-4630-bd35-e9981735db27)/bibtex.html (visited on 05/24/2019) (cit. on p. 22).

[55] R. Fitch and R. Lal. "Experiments with a Zigbee Wireless Communication System for Self-Reconfiguring Modular Robots". In: *2009 IEEE International Conference on Robotics and Automation*. 2009 IEEE International Conference on Robotics and Automation. May 2009, pp. 1947–1952. DOI: 10.1109/ROBOT.2009.5152807 (cit. on p. 32).

bc. Šimon Wernisch: *Distributed control of a platoon of wirelessly communicating slotcars using a Simulink generated code,* Master's thesis, 24th May 2019

SUPERVISOR:
doc. Ing. Zdeněk Hurák, Ph.D.

LOCATION:
Prague