Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

Sigurd Hagen Tullander

Application of a real-time operating system to enhance the microcontroller software of an autonomous model car

Nytting av eit sanntidsoperativsystem til å forbedre mikrokontrollerprogramvaren til ein førarlaus modellbil

Masteroppgave i Elektronisk Systemdesign og Innovasjon
Veileder: Per Gunnar Kjeldsberg
September 2023

NTNU
Kunnskap for en bedre verden

TECHNISCHE UNIVERSITÄT DARMSTADT

Sigurd Hagen Tullander

Application of a real-time operating system to enhance the microcontroller software of an autonomous model car

Nytting av eit sanntidsoperativsystem til å forbedre mikrokontrollerprogramvaren til ein førarlaus modellbil

Masteroppgave i Elektronisk Systemdesign og Innovasjon
Veileder: Per Gunnar Kjeldsberg
September 2023

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

NTNU
Kunnskap for en bedre verden

Application of a real-time operating system to enhance the microcontroller software of an autonomous model car

Verwendung eines Echtzeitbetriebsystems für die Verbesserung des Microcontrollersoftwares eines Autonomes Modellautos

Master thesis by Sigurd Hagen Tullander

Date of submission: 11. September 2023

1. Review: Prof. Per Gunnar Kjeldsberg
2. Review: Prof. Dr. rer. nat. Andy Schürr
3. Review: Dr.-Ing. Stefan Tomaszek

Darmstadt
ES-M-160

TECHNISCHE UNIVERSITÄT DARMSTADT

Electrical Engineering and Information Technology Department

Fachgebiet Echtzeitsysteme

NTNU
Norwegian University of Science and Technology

Application of a real-time operating system to enhance the microcontroller software of an autonomous model car
Verwendung eines Echtzeitbetriebsystems für die Verbesserung des Microcontrollersoftwares eines Autonomes Modellautos

Master thesis by Sigurd Hagen Tullander

Date of submission: 11. September 2023

Darmstadt

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Sigurd Hagen Tullander, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 11. September 2023

S. H. Tullander

Assignment

The system architecture of the vehicle used in the project seminar Autonomes Fahren (PS AF) I and II is constructed so that a self-developed Microprocessor board takes over controlling the actuators (electronic speed control and steering servo) as well as reading sensors (IMU, ultrasound, Encoder/Hall sensor) and provides a suitable interface to an on-board computer for disposal. This will take over the higher-level tasks of navigation and vehicle management.

At the current development stage, the split between the low-level control of the actuators and the higher-level control on the computer still has room for improvement. Another problem is dealing with the different variants. The firmware for the vehicles for the PS AF I and II differs only in a few points, but currently, two different versions of the firmware for the boards are maintained separately. Within the scope of this work, the following aspects should be improved:

- In a previous work, a controller for the new BLDC-Motors was developed, which should be integrated in the scope of this work. Additionally, at least a rudimentary speed controller based on the old actuators, should be implemented.
- The development of a controller for the yaw rate (steering angle) is part of this work. For this purpose, it can be beneficial also to make adaptions to the actuators.
- The tasks and the existing code structure must be analyzed, and a concept developed, these modified to support different variants and open for future developments. For this purpose, also major changes, like the usage of a real-time operating system, can be made, if this seems to be advantageous.
- Overtaken components from the existing code base must be adapted in regard to modern quality standards.

At the end of the work, there should be an interface for direct specification of speed and yaw rate (steering angle) for both cars. Additionally, a well-structured, documented code base for the microcontroller that corresponds to the usual software quality standards should be present, which represents the basis for future further developments.

The work will be carried out with support from the fachgebiet (expertise field/subject group) Control and Cyber-Physical Systems (CCPS, Dr.-Ing. Eric Lenz).

Abstract

The Project Seminar Autonomes Fahren at the Technical University of Darmstadt serves as a learning platform where students can get hands-on experience with programming an automated driving system in a 1:10 scale, targeting participation in the Carolo-Cup. In this work, we enhance the microcontroller software in charge of the underlying hardware of the driving system by applying the embedded real-time operating system FreeRTOS. Our solution is proven to increase the chances of meeting hard real-time constraints, reducing the maximum response time from $467.28\,\text{ms}$ down to only $3.24\,\text{ms}$ in a constructed overload scenario. The real-time benefits come at the cost of $45\,\%$ increased memory usage compared to the present solution, but as about $50\,\%$ of the memory is still unused, this is not a problem. With it, a low-level speed controller has been implemented with an average initial acceleration time of $478\,\text{ms}$ for speeds in the range $0.5\,\text{m/s} - 1.5\,\text{m/s}$, increasing the lifetime of currently used car models by ensuring compatibility with the upcoming generation. At last, code clones addressing hardware differences between car models have been refactored by using compile time configuration options, improving the maintainability of the software's source code.

Samandrag

Prosjektseminaret Autonomes Fahren ved det Tekniske Universitetet i Darmstadt er ein læringsplattform der studentar får praktisk erfaring med å programmere ein førarlaus bil i målestokk 1:10 med mål om deltaking i Carolo-Cupen. I dette arbeidet forbedrar vi mikrokontrollerprogramvaren som tek hand om dei underliggjande fysiske komponentene i sjølvkjøringssystemet ved å nytte innvevdsanntidsoperativsystemet FreeRTOS. Vi viser at løysinga vår aukar sjansane for å imøtekome harde sanntidskrav og reduserar den maksimale responstida frå $467.28\,\mathrm{ms}$ ned til berre $3.24\,\mathrm{ms}$ i eit konstruert overlastingsscenario. Sanntidsfordelane har ein kostnad på $45\,\%$ auka minnebruk samanlikna med den noverande løysinga, men sidan halvpartan av minnet fortsatt står ubruka er dette ikkje eit problem. Saman med det har vi implementert ein lavnivå hastigheitsregulator med gjennomsnittleg akselleresjonstid på $478\,\mathrm{ms}$ for hastigheiter i området $0.5\,\mathrm{m/s} - 1.5\,\mathrm{m/s}$, noko som forlengar brukstida til bilmodellande som vert bruka i dag ved å sørgje for kompatibilitet med den kommande generasjonen. I tillegg har vi refaktorert kodeklonar som tek omsyn om forskjellar på bilmodellane ved å nytte konfigurasjonsval på kompilatornivå, noko som gjer kildekoden enklare å vedlikehalde.

Contents

Nomenclature

Abbreviation	Description
ADS	Automated Driving System
API	Application Programming Interface
ASCII	Americation Standard Code for Information Interchange
BLDC	Brushless Direct Current
CCRAM	Core Coupled Random Access Memory
CMSIS	Common Microcontroller Software Interface Standard
COTS	Commercial-Off-The-Shelf
CPU	Central Processing Unit
DAQ	Data Acquisition Module
DARPA	Defense Advanced Research Projects Agency
DMA	Direct Memory Access
ESC	Electronic Speed Control
GPS	Global Positioning System
IMU	Inertial Measurement Unit
IRQ	Interrupt Request
ISR	Interrupt Service Routine
I^2C	Inter-Integrated Circuit
LED	Light Emitting Diode
LiDAR	Light Detection And Ranging
OBC	On-Board Computer
OS	Operating System
PC	Personal Computer
PCB	Printed Circuit Board
PS AF	Project Seminar Autonomes Fahren
PWM	Pulse-Width Modulated

Abbreviation	Description
RAM	Random Access Memory
ROM	Read-Only Memory
RTOS	Real-Time Operating System
SAE	Society of Automotive Engineers
SDG	Sustainable Development Goal
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TU-Darmstadt	Technical University of Darmstadt
UcBoard	Microcontroller Board
UN	United Nations
US	Ultrasonic Sensor
USART	Universal Synchronous and Asynchronous Receiver and Transmitter
USB	Universal Serial Bus

List of Figures

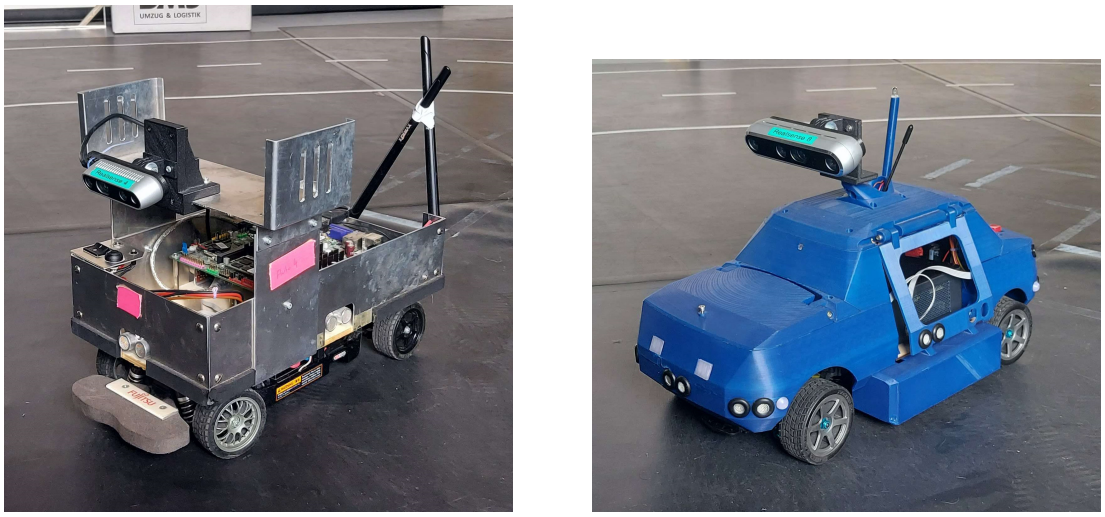
List of Tables

1 Introduction

Human errors are found to be the critical reason for 94% of road accidents, according to a technical report by the National Highway Traffic Safety Administration [1]. In other words, 19 out of 20 accidents would be avoided if the driver could be replaced by a reliable driving system that does the correct thing all of the time. Automated Driving Systems (ADSes), are being developed with a promise to prevent accidents and additionally lower transport emissions, free up driving time and increase mobility for the mobility-impaired [2, 3]. Society of Automotive Engineers (SAE) defines ADSes as automation systems that can execute dynamic driving tasks on a sustainable basis. SAE further defines six different levels of driving automation ranging from no automation (level 0) to full automation at all times in any condition in any road system (level 5) [4, 5, 6, 7]. While level 1 and 2 automation, commonly referred to as driver assistance systems, are already common in modern cars, the real challenges start at the higher levels [2]. In fact, no manufacturer in the industry is even close to attain level 5, according to the Toyota Research Institute [8]. It is still need for much more knowledge and many innovations in the field to realize the autonomously driving utopia we wish our future to become.

The first experiments on ”self-driving cars” were conducted in the 1920s in the USA. In 1925, the ”American Wonder”, a radio operated car was demonstrated for the first time by Francis P. Houdina in the streets of New York [7, 9, 10]. At that time, ”self-driving” had quite a different meaning than today, as those cars were fully operated by human drivers, there was just no human driver on board. In the 1980s, after huge advancements in computer technology, the research around autonomous driving systems gained speed by getting access to technologies such as Light Detection And Ranging (LiDAR), Global Positioning System (GPS) and computer vision [7, 6]. In 2004, the first ”DARPA Grand Challenge” was organized by the Defense Advanced Research Projects Agency (DARPA). The following years, the DARPA challenges served as the world’s primary showcase for advances in ADS technology. With the 2018 Audi A8, the world’s first production car to achieve level 3 autonomy is already a fact thanks to its AI Traffic Jam Pilot allowing the car to drive all on its own up to a maximum speed of $60\,\mathrm{km/h}$ in traffic jams [9, 2]. Several concepts for level 4 ADSes are already present, and there is reason to expect wide-spread application of highly autonomous vehicles in the near future.

At the Technical University of Darmstadt (TU-Darmstadt), students are offered to take the Project Seminar Autonomes Fahren (PS AF) to get practical experience with developing an ADS targeting participation in the Carolo-Cup, an annually held international competition for miniature vehicles founded in 2008 and carried out at the Technical University of Braunschweig in Germany [11, 12]. The project seminar has been running since 2009 and is currently in the process of developing its third car model. Pictures of the two currently developed car models are shown in Figure 1.1. Developing ADSes with model cars is associated with considerably lower costs for equipment than with real cars and thus opens the field to more people.



(a) (b)

Figure 1.1: Pictures of the PS AF car model 1 (a) and car model 2 (b).

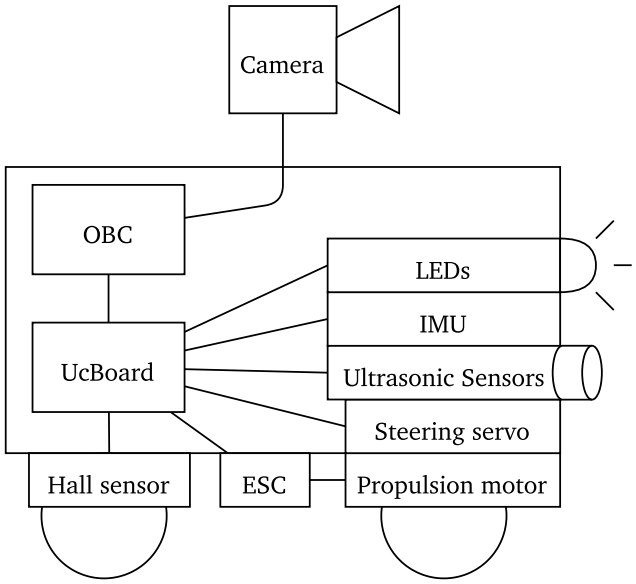


Figure 1.2: Simplified overview of the physical components of a PS AF car and their interconnections.

A simplified overview of a PS AF car's physical components is shown in Figure 1.2. Control is split across two computer systems; a Microcontroller Board (UcBoard) and a more powerful On-Board Computer (OBC). The OBC is in charge of reading camera output and doing path planning, while the UcBoard is in charge of reading outputs from all other sensors and applying inputs to the actuators. The UcBoard and the OBC communicate via Universal Serial Bus (USB). Students participating in the project seminar work in groups and their task is to implement the best possible autonomous driving algorithm on the OBC. The rest of the ADS is provided as-is for the students.

The present UcBoard software is a bare-metal C application, with no Real-Time Operating System (RTOS), meaning that routine scheduling is performed by a custom scheduling system without pre-

emptive capabilities. As we will describe later, this generally leaves the system's ability to satisfy real-time constraints more vulnerable. Also the UcBoard's current interface to the OBC does not provide a way to specify an exact driving speed, even tough the UcBoard is in control of the speed sensor as well as the motor. Improving these aspects of the UcBoard can have significant gains for the project seminar as a learning platform, as it will enable the OBC to rely more on the UcBoard for successful accomplishment of low-level tasks and enable the students to focus more on the higher-level autonomous driving algorithm and in the end develop a more sophisticated ADS.

The scope of this work is in its entirety limited to the software of the UcBoard. Even though the UcBoard communicates with the OBC under intended operation, we will always connect an off-rig computer to perform our experiments. The addition of a brushless propulsion motor will be described, but the only work we contribute in this regard is implementing support for this new component in the UcBoard's software.

This work will contribute to the United Nations's (UN) 17 Sustainable Development Goals (SDGs), adopted by all UN member states in 2015 [13], by applying a Real-Time Operating System to enhance the microcontroller software of the PS AF at the TU-Darmstadt. In particular, the work will contribute to goal 4, quality education, by providing a deeper understanding of the real-time properties of the microcontroller used in the PS AF and enhance the project seminar's capabilities as a learning platform. Since learning about autonomous driving by experimenting with model cars instead of full-scale cars requires much less energy or other resources, it will also contribute to goal 12, responsible consumption and production. And additionally, as a contribution to research in the autonomous driving field as a whole, the work will contribute to goal 9, industry, innovation and infrastructure, and goal 11, sustainable cities and communities.

1.1 Objectives and limitations

In this section, we present the objectives, or research goals, of the work. We also present the most noteworthy limitations of how the work is carried out.

In this report, we will investigate what benefits and disadvantages the application of a Real-Time Operating System can provide the UcBoard software in terms of responsiveness, robustness and memory usage. To do this, we will implement a new version of the UcBoard software which applies the embedded Real-Time Operating System FreeRTOS. By comparing the performance of the software with FreeRTOS to the current version without, we will attempt to answer the following questions:

- Responsiveness – How long time does it take for the UcBoard to respond to a command with FreeRTOS compared to without?
- Robustness – In which ways and to which degree can the response time be affected by undesirable conditions in the system, such as overload or software bugs, with FreeRTOS compared to without?
- Memory usage – How much memory does the UcBoard software use with FreeRTOS compared to without?

The work also includes the implementation of a low-level speed controller, that is, a speed controller implemented onto the microcontroller, as opposed to the current need for speed control logic in the OBC software. A major motivation for this is the fact that the next generation of PS AF cars will have a

dedicated motor controller board with integrated speed controller. When programming the OBC with the next generation in mind, there is undesirable to implement logic for speed control, since the motor controller board is more suitable for this task. By implementing a speed controller for the current motor on the UcBoard, it will make the old car model's interface to the OBC compatible with the next generation's interface, thus making the old cars more useful as development and test platforms.

Regarding the speed controller, we will quantify how well it is able to control the speed. This includes answering the following questions: How long time does it take the for the car to reach the target speed? How large is the deviation between the resulting speed and the target speed? How does these results compare to alternative approaches?

Lastly, we will try to improve the overall structure of the source code for the UcBoard software. Currently, there exist two different variants of the software, targeting two different car models. Most of the source code is the exact same, they differ only in a few points. In this work we will discuss strategies for merging the two variants together to increase the source code's maintainability.

To produce the experimental results, cars have only operated indoors on a perfectly flat and even floor. One should expect the results to differ significantly if the same experiments are conducted in rougher conditions, e.g. outdoors on grass, perhaps even with hills.

The entire work is centered around the ADS of the PS AF at the TU-Darmstadt, its entire hardware platform and the structure of the existing microcontroller software. The implementation is carried out specifically for this ADS and the results are directly applicable only for it. However, general methods and concepts we develop can be applicable for other ADSes as well.

1.2 Research method

In this section, we provide an overall description of the research methods used in the work. That is, what kind of work is done and what kind of experiments are conducted.

The research begins with a thorough analysis of the present UcBoard software system, comprising its environment, functional structure as well as source code structure. Then, relevant literature is studied to search for possible answers and solutions to identified problems. The most promising solutions are implemented and a series of experiments are conducted to verify the integrity of the theory and the quality of the implementation, as well as to find answers to our initial research goals. The proposed solutions are compared to the present UcBoard software and to other but similar software systems from related academic works.

1.3 Report structure

The report is divided into the following chapters.

1. Introduction
2. Background
    - Review of relevant topics from the literature for understanding concepts and reasoning utilized in the work.
3. Strategy
    - Our developed strategies for implementing and verifying our solutions.
4. UcBoard software implementation
    - Implementation of enhancements to the UcBoard software.
5. Experimental setup
    - Description of experimental setups used to verify our solutions and answer research questions.
6. Results
    - Results from evaluating implementation and conducting experiments.
7. Discussion
    - Discussion of results.
8. Conclusion

2 Background

In this chapter we present the background theory that is necessary to follow the rest of the report. This mostly includes theory available in the literature, as well as a thorough technical description of the present ADS used in the PS AF. In Section 2.1, we describe the linear mapping, a handy mathematical method to map values across different value ranges, in Section 2.2, fundamental concepts of real-time systems are introduced and the purpose of Real-Time Operating Systems is explained. In Section 2.4, we describe necessary concepts from the field of control theory. Section 2.6 describes the present ADS of the PS AF at the TU-Darmstadt. And in the end, in Section 2.7, we take a look at similar or related work, featuring three other projects that utilize a Real-Time Operating System to realize an autonomous model car.

2.1 Linear mapping

Linear mapping is a way to map values from one range to another range:

Definition 2.1 (Linear mapping)**.** *The linear mapping of a value x from a range $[x_0, x_1]$ to a range $[y_0, y_1]$ is given by*

$$l(x; x_0, x_1, y_0, y_1) = y_0 + \frac{y_1 - y_0}{x_1 - x_0} \cdot (x - x_0). \tag{2.1}$$

It is useful for mapping a control signal value to an appropriate operating range for a physical process. An example of a linear mapping is visualized in Figure 2.1, where an input value 0.3 from the range $[0, 1]$ is mapped linearly to an output value 13 in the range $[10, 20]$.

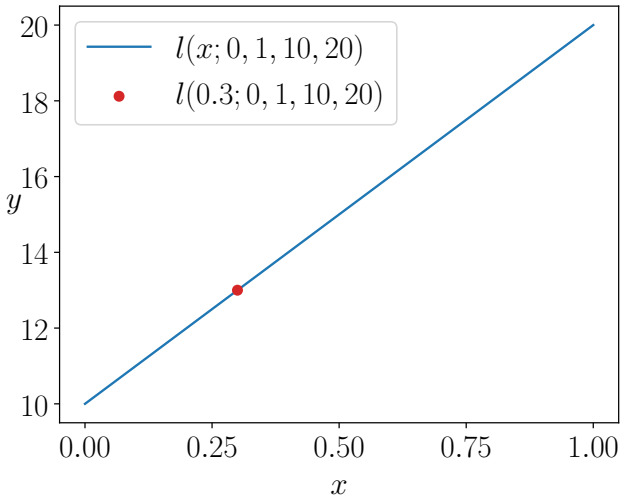


Figure 2.1: Linear mapping of 0.3 from $[0, 1]$ to $[10, 20]$.

2.2 Real-time systems

A real-time system is a computer system with real-time constraints that define how quickly the system has to respond to a given set of inputs. In [14], fundamental real-time system concepts are defined and discussed at a basic level. At the most basic level, a real-time system is defined in the following way:

Definition 2.2 (Real-time system)**.** *A real-time system is a system that must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure.*

In [14], there is further discussed how one can argue that every practical system is a real-time system, since every practical systems will be completely useless if it cannot produce outputs within a reasonable time frame. It is therefore necessary to distinguish between hard, firm and soft real-time systems.

Definition 2.3 (Hard real-time system)**.** *A hard real-time system is one in which failure to meet a single deadline may lead to complete and catastrophic system failure.*

Definition 2.4 (Firm real-time system)**.** *A firm real-time system is one in which a few missed deadlines will not lead to total failure, but missing more than a few may lead to complete and catastrophic system failure.*

Definition 2.5 (Soft real-time system)**.** *A soft real-time system is one in which performance is degraded but not destroyed by failure to meet response-time constraints.*

2.2.1 Real-time constraints

The real-time constraints of a real-time system arise from the physical environment in which the system operates. For example, when the driver of a car presses the break pedal, the break controller must

actuate the breaks within a few milliseconds to avoid a potential crash. This is a hard constraint, since missing a single deadline can have catastrophic consequences. When new speed measurement data is available, this data must be processed and the speedometer must be updated within a few tenths of a seconds, or the driver will be misinformed about how fast she/he is driving. This can lead to the driver doing bad decisions, but since the driver has an intuitive feel of how fast the car is going she/he isn't dependent on always getting instant speed data to drive safely. Therefore, this should be categorized as a firm real-time constraint. Soft real-time constraints in a car include most convenience features, such as the responsiveness of the navigation system or the air conditioner. [14] also notes that there is big room for the interpretation of hard, firm and soft real-time systems – every system can probably be categorized any way – soft, firm or hard – real-time by the construction of a proper supporting scenario.

2.2.2 Scheduling concepts

A "task" is an abstraction of a running program [14]. It can be seen as a set of instructions that need to be executed in order to achieve some functionality. Real-time systems usually have multiple tasks, and a scheduler's responsibility is to "schedule" the tasks, or in other words, decide which task to execute at any time. In a single-processor system only one task can execute at a time. A task is requested or initiated at some point in time and executes until it is completed. In a real-time system each task is associated with a deadline, a point in time before which the task has to be completed. The same task is often requested multiple times.

Definition 2.6 (Task)**.** *A task is an abstraction of a running program. A task can be preempted an resumed by the scheduler.*

One of the most influential works in real-time scheduling theory is a paper by Liu and Layland published in 1973 [15, 16]. In this paper, there is described a specific, yet common, scheduling problem consisting of a set of periodically requested tasks, with deadlines defined to be the time of the next request for the same task. All tasks are assumed to be independent and preemptible and have fixed computation times. That a task is preemptible means that the scheduler can pause its execution at any time instant to run another task for a while and resume the first task again later without modifying the task's behavior. Another assumption being made is that a task cannot itself ask the scheduler to be paused, it has to be preempted.

Based on these assumptions, Liu and Layland [15] present the rate-monotonic priority assignment rule and proves that it is the optimal way to schedule a set of fixed-priority tasks, given these assumptions. In the same paper, they also present a deadline driven scheduling algorithm. In this algorithm, the priorities are assigned based on which task has the closest deadline. Up through the years several researchers and developers have designed and implemented various additions and modifications to these algorithms to achieve different goals.

For general-purpose computing (in contrast to real-time computing) round-robin scheduling is often a beneficial approach. In a round-robin scheduler, each task or process is fairly assigned time slices that give all tasks an equal share of the Central Processing Unit (CPU) time [17]. For general purpose computers there is often a goal that all tasks get the chance to run the same amount of time, even when the system is overloaded, this is what round-robin scheduling achieves.

The term ”task” is very closely related to how the scheduler works. A set of tasks is a set of individually schedulable programs by the scheduler in question. However, in practical applications, we are often more interested in the function the task performs. And since most functions can be performed by multiple tasks and multiple functions can be performed by the same task, we will use the term ”routine” when we refer to any sort of program, regardless of implementation, that performs a specific function.

Definition 2.7 (Routine)**.** *A routine is any program that executes from it is requested until it is completed to perform a specific function.*

2.2.3 Task interaction

In most common applications some task interaction is necessary, and the exact assumptions by Liu and Layland do not hold [14]. Typical task interaction has two main objectives, to transfer data (communication) and to ensure relative timing of operations (synchronization). Proper task interaction is achieved by using synchronization primitives, such as semaphores, mutexes, events, mailboxes and so on. If task interaction is not done properly it can lead to several faults, including data corruption, infinite loops and deadlocks, to name a few.

Synchronization and communication is often two sides of the same coin. E.g. when a task A tries to fetch some data from a task B, it requires B to reach that point of execution where data is actually provided to A. If B has still not provided data when A requests it, A should wait until the data is ready. In this situation, A is blocked from executing further. When blocked, the scheduler should not execute the task it is ready. Therefore, the scheduler assigns each task one of the 4 following states [14]:

1. Executing – The task is currently executing
2. Ready – The task is currently not executing, but is ready to do it
3. Blocked – The task is blocked from executing since it has to wait for other tasks
4. Dormant – The task is either not requested yet or is already completed

2.2.4 Real-time operating systems

An Operating System (OS) is a layer of software that interacts directly with the system hardware and provides a developer-friendly interface for higher-level applications in the system. A Real-Time Operating System (RTOS) is an OS that is designed to support real-time system needs. An RTOS usually gives more control of scheduling than a general-purpose OS, and provides utilities for synchronizing threads. Many RTOSes target embedded systems, and therefore try to use as little memory and CPU resources as possible. These can also be categorized as embedded OSes, but that does not mean all embedded OSes are RTOSes [18].

FreeRTOS is an example of an embedded RTOS. FreeRTOS has a minimal Random Access Memory (RAM) usage of $236\,\mathrm{B}$ and minimum Read-Only Memory (ROM) usage of 5-10 KiB [19]. On the other hand, RTLinux is an example of an RTOS that is not meant for tiny devices. The footprint of RTLinux will even in the most stripped-down configurations be in the megabyte-range for both ROM and RAM usage [20].

Different RTOSes also have varying support for different scheduling strategies. FreeRTOS only supports one scheduling strategy, which is a fixed priority scheduling algorithm [16, 21] similar to the rate-monotonic priority assignment rule presented by Liu and Layland in 1973 [15]. The FreeRTOS scheduler can however be configured to not use task preemption and uses by default round-robin time slicing to share CPU time between tasks of the same priority [22]. RTLinux supports several more sophisticated schedulers, like ”earliest deadline first” [20]. Partly due to this, RTLinux has the potential of consuming considerably more CPU time than FreeRTOS.

2.2.5 Interrupts

Interrupts is a hardware feature present in most modern computers, including microcontrollers. It lets peripheral devices notify the CPU that some event has occurred by sending an Interrupt Request (IRQ). The CPU will then interrupt its current execution and start executing the IRQ's configured Interrupt Service Routine (ISR). Multiple IRQs might occur at the same time, or a new interrupt request might occur while another ISR is still executing. For this reason, ISRs can usually be assigned priorities, just like tasks within an RTOS. However, since ISR execution is controlled by the hardware and not the RTOS, the priorities of ISRs can usually not be interleaved with priorities of tasks within an RTOS. The interrupts and the RTOS tasks constitute two separate priority spaces, where the ISRs' priority space is above the RTOS tasks' [22, 23].

The systick interrupt is a special interrupt that is available on most microcontrollers. The systick interrupt occurs periodically with a configurable tick period T_t, configured to 1 ms for the UcBoard in the ADS of the PS AF. This makes the systick ISR a function that is ensured by the hardware itself to run periodically with period T_t, which can be used as a basis for implementing periodic routines.

2.3 FreeRTOS

FreeRTOS is a free and open-source Real-Time Operating System for embedded systems. It is suitable for microcontrollers thanks to its minimal memory usage and limited feature set. In this section, we will look more into detail of the features FreeRTOS provides that are relevant to us [24, 22].

2.3.1 The FreeRTOS API

The FreeRTOS Application Programming Interface (API) is made up of a set of functions, commonly referred to as FreeRTOS' API functions. The application interfaces the FreeRTOS through calling these functions. Like common functions, the API functions can accept function arguments and return a return value. API functions are sometime referred to as ”kernel calls” or ”system calls”, since they invoke the OS kernel. API functions are used to perform operations like creating a task, deleting a task, delaying a task or interact with synchronization primitives.

In addition to the API functions, the FreeRTOS API also includes a set of port macros. To make FreeRTOS compatible with and optimized for different CPUs architectures, each supported CPU architecture has its own port. The port is made up of a C file and a header file that defines CPU architecture-specific code. For the UcBoard software, for example, we will use the ARM Cortex-M4F port. Port macros are

conceptually similar to API functions, but they are macros, and they are provided by the port, and not by the regular kernel.

2.3.2 Tasks and scheduling

FreeRTOS features a priority-driven preemptive scheduler. The scheduler runs each time a task ”yields” and each time a systick IRQ is raised. A task can yield explicitly by calling the `portYIELD` port macro, or implicitly by calling a yielding API function.

Tasks can be created with the `xTaskCreate` API function, taking the task's priority as an argument. Each task has its own execution stack, storing function calls and local variables. FreeRTOS requires the maximum possible stack usage by each task to be allocated for its stack from the task is created until it is deleted. This ensures that the task always can be resumed if it is requested.

FreeRTOS task priorities cannot be interleaved with ISR priorities [22], which means that each task will have a lower priority than all ISRs.

FreeRTOS tasks can generally be divided into two non-overlapping categories; system tasks and application tasks. System tasks are created and managed automatically by the RTOS itself, while application tasks are created and managed by the application. In FreeRTOS, there are two system tasks; the idle task, which is run when no other task is runnable, and the timer task, also called the daemon task, which is responsible for executing timer callbacks [22].

2.3.3 The stack size of a task

At a given time instant, the stack usage of a task is equal to the sum of the stack usages of the functions that are currently in the task's call stack. Say, a task calls a function A, which calls a function B, which calls a function C, then the stack usage is the sum of the stack usages of A, B and C. But if A calls a function D after B has returned, the total stack usage is only the sum of the stack usages of A and D. In other words, the stack usage of a task varies throughout the task['s] execution. The stack size of the task has to be greater than (or equal to) the maximum stack usage it can ever have. If the stack usage becomes greater than the allocated stack size, the task stack will use memory that is not allocated to it which is likely to cause unpredictable bugs.

The total stack usage of a task, which consists on functions calling other functions, cannot generally be known exactly at compile time, since conditional branches in the program are performed dependent on dynamic input. There exist tools to estimate or calculate an upper bound for this, like for example StackAnalyzer [25].

2.3.4 Synchronization primitives

To enable safe task interaction, FreeRTOS provides synchronization primitives. The only synchronization primitive especially relevant for this report is the queue. A queue lets items be sent from one task to another. If the length of the queue is greater than 1, multiple items can be in transmit at the same time. Any task can send items to a queue with the `xQueueSend` API function, and any task can receive items from a queue with the `xQueueReceive` API function. If a task tries to receive an item from an empty queue it will be blocked until an item has arrived or until a given maximum block time has run out.

2.3.5 Timer callbacks

In addition to tasks, FreeRTOS provides a secondary routine scheduling mechanism called software timers. A software timer is a FreeRTOS object that is responsible for the execution of a timer callback. A timer callback is a function created by the application writer that executes the code that needs to be scheduled. Software timers can be created with the `xTimerCreate` API function. A software timer can be configured to run once or to run periodically until it is stopped. When running periodically, a period must be configured.

A timer callback is a more lightweight type of routine, as it doesn't have its maximum possible stack usage allocated for itself. Instead, all timer callbacks share the same allocated stack space, and only one of them can use it at a time. But this also makes them limited, since they cannot be assigned individual priorities, and if one timer callback hangs up, it will block all other timer callbacks at the same time. Software timer scheduling is performed by a special FreeRTOS system task called the timer task, and it is the timer task's priority and maximum stack size that define the priority and maximum stack size of every timer callback.

As timer callbacks are callbacks, they enforce a stricter programming pattern than tasks. Say a routine A needs to wait for another routine B to send a value to a queue, if A is a task, A can simply do a blocking API call to `xQueueReceive` and wait. If A is a timer callback, A must first check if there is something in the queue, and if it is not, it needs to to exit, store necessary state outside of its stack and check the queue again the next time it runs. It should be mentioned that since timers do run in a task, it absolutely *can* do blocking API calls, but it is bad practice, since it will block all other timer callbacks at the same times.

2.3.6 Non-RTOS real-time systems

This report might up until now have given the impression that utilizing an RTOS is an absolute necessity when designing a real-time system. But this is by no means a universal truth, many well established techniques for writing good embedded software without the use of an RTOS exist. If the system being developed is simple, not using in RTOS might very well be the most appropriate solution [22, 26]. As we will see in Section 2.6, the present PS AF UcBoard software does not use an RTOS, and is still capable of scheduling conceptually individual routines by periodically running callback functions sequentially. However, in this work, the goal is to investigate how we can use an RTOS to enhance the present system, and in this regard, investigating non-RTOS approaches is of little interest.

2.4 Control theory

In this section, we will give a brief introduction to control theory directed towards speed control of a model car. Control theory and control engineering are huge fields, of which we will only scratch the surface. We will have a highly practical view, focusing on what knowledge is needed to design a simple speed controller for the PS AF UcBoard. We will start by introducing the general concept of a control loop and then narrow it in for speed control. Then, we will describe the purpose and inner workings of a feed-forward controller and a PI-controller, since these are the two standard controllers we will use when implementing a speed controller on the PS AF UcBoard.

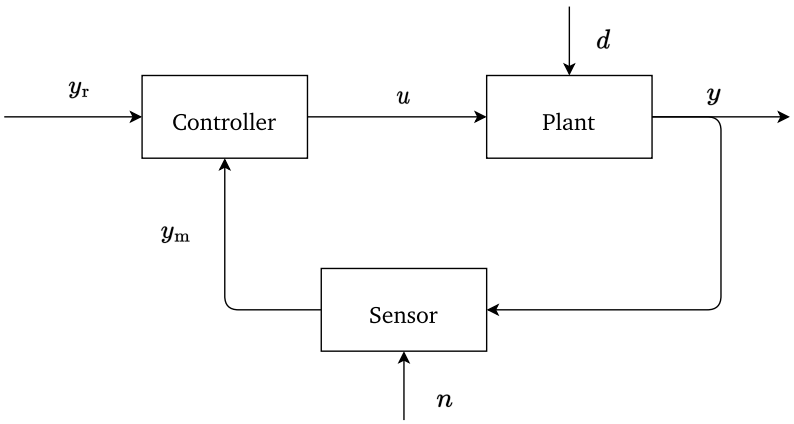


Figure 2.2: A basic feedback loop.

2.4.1 A basic control loop

In its most basic sense, a control system is a system that controls a *plant*. Generally speaking, the objective is to make some output, say y, behave in a desired way by manipulating some input, say u [27]. Most control systems use a sensor to measure the plant output y, which gives us a sensor output signal y_m that is fed back to the controller, as shown in Figure 2.2. This is called a feedback loop. The control system as a whole takes three inputs from the outside world – the reference or command input y_r, the plant disturbance d and the sensor noise n.

In control theory, there is an important distinction between process control systems, in which the goal is to make the plant output y stay close to a constant reference value y_r, and servomechanisms, in which the goal is to make the plant output y follow a variable reference value y_r. Whether the control system to design is categorized as a process control system or a servomechanism has important impacts on how the control system best can be designed [28].

2.4.2 Speed controller

Figure 2.3 shows a simplified diagram of a speed control system. It is a simple closed control loop consisting of a motor, a speed sensor and a speed controller. The speed controller provides the motor some input u. For our purpose, we will assume that u is a scalar drive value. The motor moves the car and thus produces a speed v as an output. The speed is measured by a speed sensor and provided as a feedback v_m to the speed controller. The speed controller takes the requested speed v_r as an input from the outside world. The plant disturbance d and the sensor noise n, which were depicted in Figure 2.2, are ignored for simplicity. A speed controller is a servomechanism, since v_r is variable. For this reason, we will from now on note v_r as $v_\mathrm{r}(t)$.

2.4.3 Feed-forward control

In servomechanisms, the largest impact on the system is usually the reference value. Therefore, it is often beneficial to apply feed-forward of the reference value within the controller [28].

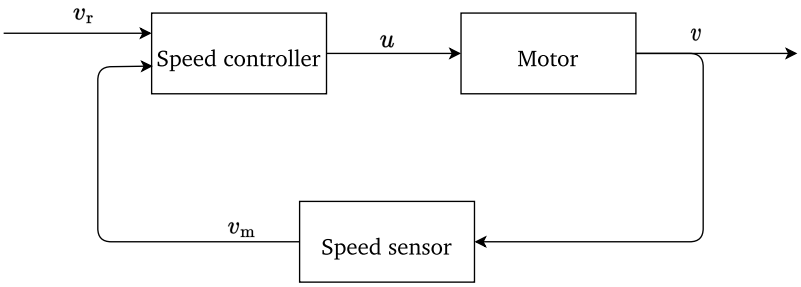


Figure 2.3: Simplified diagram of a speed control system.

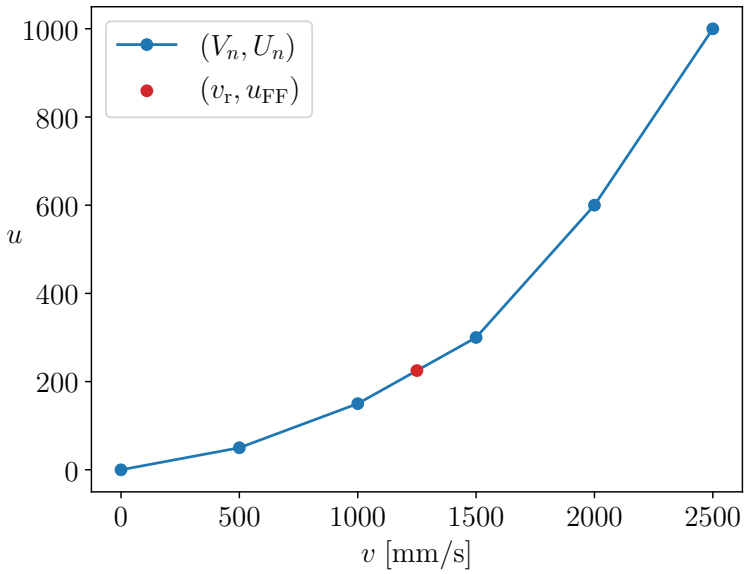


Figure 2.4: Visualization of example feed forward control.

In the case of a speed controller, we can calculate a feed-forward drive value $u_{\text{FF}}(t)$ from the currently requested speed $v_{\text{r}}(t)$ by doing a linear interpolation of $v_{\text{r}}(t)$ against a finite set of N reference pairs, each pair consisting of a speed V_n and a drive value U_n for $n \in \{0 \dots N\}$ like so:

$$\begin{aligned} u_{\text{FF}}(t) &= U_{n-1} + \frac{U_n - U_{n-1}}{V_n - V_{n-1}} \cdot (v_{\text{r}}(t) - V_{n-1}), \\ v_{\text{r}}(t) &\in [V_{n-1}, V_n\rangle, \end{aligned} \tag{2.2}$$

where V_n and U_n are sorted in ascending order. A visualization of this is shown in Figure 2.4.

2.4.4 PI-controller

A PI-controller is one of the four most common forms of standard controllers. Standard controllers are beneficial for constructing control systems, since they have wide-range tunable parameters and usually give a satisfactory result [28].

A PI-controller is a controller consisting of a proportional and an integral term. A purely proportional controller is a simple configurable amplifier, and is used to do linear compensation of the control error $e(t) = v_{\text{r}}(t) - v_{\text{m}}(t)$. The main goal of the integral term is achieve zero stationary error [28]. The proportional term is given by

$$u_{\text{P}}(t) = K_{\text{P}} \cdot (v_{\text{r}}(t) - v_{\text{m}}(t)), \tag{2.3}$$

and the integral term is given by

$$u_{\text{I}}(t) = K_{\text{I}} \cdot \int v_{\text{r}}(t) - v_{\text{m}}(t) \text{d}t \tag{2.4}$$

[29]. The proportional constant K_{P} and the integral constant K_{I} are tunable parameters. The resulting drive value u_{PI} is given by the sum,

$$u_{\text{PI}}(t) = u_{\text{P}}(t) + u_{\text{I}}(t). \tag{2.5}$$

2.4.5 Discrete-time PI-controller

While expressions in the continuous time domain are pretty to look at, they are unfeasible to implement directly on a microcontroller. We need a way to express our controller algorithm with formulas that can be computed in discrete time steps. To do this we can apply the trapezoidal integration rule on $u_{\text{PI}}(t)$ [28, 29]. Let $k \in \{0, \mathbb{N}\}$ denote a time step in which the speed controller algorithm calculates a new drive value, so that $v_{\text{m},k}$ is the speed measured in time step k. The time steps occur with a constant time interval T_{S}, the sampling period. Then, the drive value to apply until the next time step $k + 1$ can be expressed as

$$u_{\text{PI},k+1} = K_{\text{P}} \cdot (v_{\text{r},k} - v_{\text{m},k}) + K_{\text{I}} \cdot \sum_{i \leq k} (v_{\text{r},i} - v_{\text{m},i}) \cdot T_{\text{S}}. \tag{2.6}$$

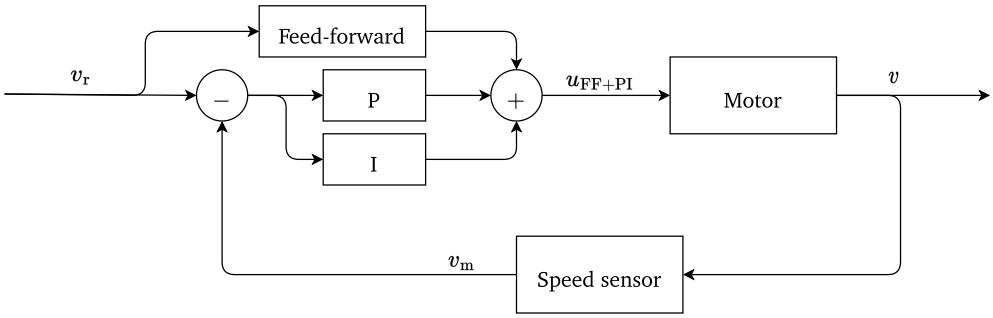


Figure 2.5: Control structure for a combined feed-forward and PI speed controller.

2.4.6 Combined feed-forward and PI-controller

In servomechanisms it is often beneficial to combine the reference-value-adaptive properties of a feed-forward controller with the disturbance resistance of a PI-controller [28]. This can be done by simply adding the terms together, like so:

$$u_{\text{FF+PI}}(t) = u_{\text{FF}}(t) + u_{\text{PI}}(t), \tag{2.7}$$

which becomes

$$u_{\text{FF+PI},k+1} = u_{\text{FF},k+1} + u_{\text{PI},k+1} \tag{2.8}$$

in the discrete time domain. The full control structure for a combined feed-forward and PI speed controller is shown in Figure 2.5.

2.5 Software evolution

The term software evolution is often used to refer to the continual development of software over longer periods of time. The need for software to evolve is generally caused by several reasons. These reasons include the need to correct and optimize the software as bugs and issues are detected, and the need to adapt the software to a changing operational environment. The operational environment can change either because the hardware on which the software runs changes, other physical components with which the software interacts changes, or because the requirements for what the software should be able to do changes [30].

Cloning or duplication of code, popularly called copy-pasting [31], is a common practice in software development. But it is also considered a serious problem in industrial software [32, 33]. Adapted from [33], we can sum up the problem of code cloning in the following way:

Code cloning is considered a serious problem in industrial software and it is suspected that many large systems contain approximately $10\,\%$ – $15\,\%$ duplicated code. Code cloning generally leads to extra maintenance overhead and risk. If a bug is identified within a piece of code that has been cloned, the same bug will have to be fixed in every clone instance. This is not always a simple copy-paste operation,

as the context or the clone itself can be modified. Bugs can also be introduced in the cloning process if the programmer lacks a proper understanding of the original code and its context. But cloning can also occur with good intentions, with regard to code understandability, evolvability, technology limitations or external business forces. Firstly, duplicating code can be used to keep software architectures clean and understandable, by keeping unreadable or complicated abstractions from entering the system. Secondly, code that is abstracted to address two or more similar, but separately evolving requirements may be difficult to modify. Thirdly, code can be cloned due to limited expressiveness in the programming language, leading to systematic use of boiler-plated solutions. And lastly, cloning code is in many situations faster than developing proper abstractions. External business forces may necessitate cloning due to the need for a short time-to-market for a software product.

Cloning code and modifying it to address separately evolving requirements is called forking. Say, we have a software A developed for a physical system A'. Then, a very similar system B' comes into play and needs a software similar to A, but adapted to the exact requirements for system B'. For solving this problem it can be a quick and efficient solution to make a new software B as a fork of A. In the long run this might or might not be a beneficial solution, depending on how the requirements for system A' and B' evolve. If the 2 systems tend to follow the same path forward, aiming to meet the same new requirements, maintaining the variants as separate forks is undesirable, as it introduces more maintenance overhead. On the other side, if the requirements for system A' and B' evolve in different directions, so new requirements for system B' do not apply for system A' and vice versa, the initial attempts at keeping the 2 softwares as 1 might have been in vain, and the benefits of maintaining the softwares as 1 do not outweigh the costs of dealing with more complex abstractions.

Parametrization by introducing configuration options can be used as a technique to refactor code clones [34, 35]. A configuration option is conceptually a key-value pair, where the key represents the configuration option's name and the value is a choice made by a specific practitioner with a specific use case for the software. Software configuration options can be stored in files, which are called configuration files. Software configuration can occur at several stages, ranging from compile-time configuration to run-time configuration. At which stage a configuration option is introduced has a large impact on what the configuration option can do. For example, a configuration option that enables or disables a complex feature at compile-time can have a big impact on the compiled program size and performance, while a run-time configuration option doing the same cannot change the actual compiled program and is more limited in this sense [34].

2.6 The ADS of the Project Seminar Autonomes Fahren

In this section we will describe the present ADS of the Project Seminar Autonomes Fahren (PS AF) at the TU-Darmstadt, including its hardware, its software and speed control mechanism. It is purely a description of the state of the ADS before any contributions from our work are applied.

The ADS of the PS AF possesses two computer systems; the On-Board Computer (OBC) and the Microcontroller Board (UcBoard). While the UcBoard is a single-core microcontroller with a total of 80 KiB of RAM, the OBC is a considerably more powerful Linux machine featuring 4 CPU cores and 8 GiB of RAM [36, 37]. The OBC and the UcBoard communicate over a custom serial protocol through a USB cable. Within the protocol, the OBC holds the role as commander, and the UcBoard holds the role as worker. By commander/worker, we refer to the same type of relationship which is historically referred to as master/slave in technical literature.

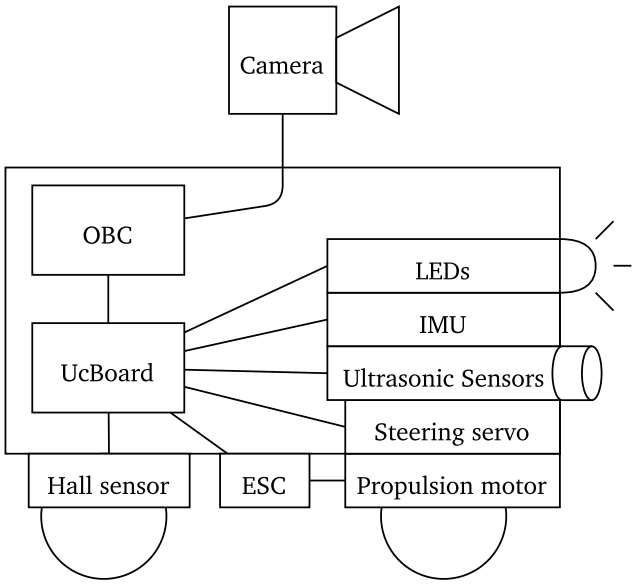


Figure 2.6: Simplified overview of the physical components of a PS AF car and their interconnections.

A simplified overview of the physical components of a PS AF car and their interconnections is shown in Figure 2.6. The OBC is responsible for reading and processing output from the camera, as this involves a lot of image processing, which is way too resource intensive for the UcBoard. The UcBoard is responsible of reading data from all other sensors; the Inertial Measurement Unit (IMU), Ultrasonic Sensors (USes) and the hall sensor. The UcBoard also controls actuators; Light Emitting Diodes (LEDs), the steering servo and the propulsion motor. The propulsion motor is operated via an Electronic Speed Control (ESC).

There currently exist two slightly different car models used in the PS AF; model 1 and model 2. Additionally, a car model 3 is currently in development. The model described above, and the model we will generally refer to if not otherwise is specified, is model 1. Model 2, however, is for the most part equal to model 1, but some aspects differ. For us, the most notable differences are:

- Model 2 uses an encoder as speed sensor instead of a hall sensor.
- Model 2 uses a different type of IMU.
- Model 2 uses a different type of Ultrasonic Sensors.

Model 3 will be mostly similar to model 2, the only interesting difference for our part is that it will have a different motor, a Brushless Direct Current (BLDC) motor, and a dedicated motor controller board with integrated speed controller, and therefore neither a hall sensor nor an encoder as speed sensor, as the BLDC motor is orientation-aware and can report the driving speed on its own.

2.6.1 Serial protocol

The OBC and the UcBoard communicate over a custom serial protocol. The serial transfer speed is configured to $921\,600$ bit/s and uses 8-bit ASCII encoding with one start bit, one stop bit, no parity bits and no hardware flow control. This gives us a data transfer speed of $92\,160$ characters/s or

92.16 characters/ms [38]. The OBC holds the role as commander and the UcBoard as worker, as the OBC sends the UcBoard commands, which the UcBoard executes and responds to. A complete list of all available commands with descriptions can be found in [39].

In principle all commands are structured so that they either set a value or get/retrieve a value. Set-commands are prefixed by an exclamation mark (`!`), while get-commands are prefixed by a question mark (`?`). For example, to drive forwards with a drive value of 500, the OBC sends the following message to the UcBoard:

```
!DRV F 500
```

. Each command sent by the OBC should have a response from the UcBoard. Responses are prefixed by a colon (`:`). For set-commands, the response is usually just the the value that was set or sometimes even just `:ok`. For get-commands, the response contains the value that was read. For example, to retrieve the current drive mode and value, the OBC sends the following message to the UcBoard:

```
?DRV
```

, and the UcBoard responds with

```
:F 500
```

. However, responses are not the only messages that the UcBoard can send to the OBC, it can also send messages through output streams, most notable of which being the Data Acquisition Module's (DAQ) output stream.

2.6.2 UcBoard

The microcontroller on the UcBoard is an STM32F303VE microcontroller. It features an ARM Cortex-M4 32-bit CPU with 512 KiB flash memory and 64 KiB Static Random Access Memory (SRAM). Additionally, it features 16 KiB routine boosting Core Coupled Random Access Memory (CCRAM) [40]. This means that CCRAM can be used by time critical routines to lower runtime, as CCRAM is faster than conventional RAM. But the CCRAM also has some limitations, it is for example not accessible by the Direct Memory Access (DMA) controller, so it can not be used for DMA interaction. Other hardware features of interest include high precision hardware timers, Universal Synchronous and Asynchronous Receiver and Transmitter (USART) controller, Serial Peripheral Interface (SPI) controller and Inter-Integrated Circuit (I^2C) controller.

Software

The UcBoard software is written in C without an RTOS, but utilizes officially provided drivers from ST as a low-level interface to peripherals. Also without an RTOS, the software is capable of executing several routines to perform all necessary functions. The routines can be categorized in the following way:

- **Motor controller** – Responsible of low-level interactions with the ESC.
- **Steering controller** – Responsible of low-level interactions with the steering servo motor.

- **Voltage measurement** – Measures voltage of the motor battery.
- **Hall sensor** – Calculates measured driving speed from hall sensor impulses.
- **IMU** – Interactions with the IMU.
- **US** – Interactions with Ultrasonic Sensors.
- **DAQ** – Implements the Data Acquisition Module (DAQ).
- **CarUI** – User interface of car (buttons and general-purpose LEDs).
- **SYS LED** – Makes sure the SYS LED blinks.
- **Communication RX** – Handle commands from the OBC.
- **Communication TX** – Transmit messages to the OBC.

Most routines are implemented as a systick callback, which is the UcBoard software's abstraction of a periodically scheduled routine, without relying on an RTOS. A systick callback is a function that is called by the systick ISR. As explained in Section 2.2.5, the systick ISR is a periodically running function that can be used as a basis for implementing periodic routines. The concept of systick callbacks is the UcBoard's way of of implementing periodic routines with the systick ISR as a basis. The routines that are not implemented as systick callbacks are SYS LED, Communication RX and Communication TX. SYS LED is implemented directly in the systick ISR itself, while Communication RX is implemented in the serial reception ISR. Communication TX is implemented as a combination of the serial transmission ISR and an infinite loop in the main thread. A major weakness of systick callbacks compared to preemptively scheduled tasks in an RTOS is that all systick callbacks are called by the same function, meaning that if one systick callback consumes a lot of processing time or even hangs up completely by e.g. entering an infinite loop, the entire system is affected. With preemptive scheduling, if a low-priority task hangs up, higher priority tasks can still run without interruptions.

It should be noted that the motor controller is not the same as the speed controller. The term ”motor controller” always refers to the routine in the UcBoard software that is responsible for motor interactions. The term ”speed controller” always refers to a software algorithm that is used to control the speed by continuously reading speed measurements and output drive values for the motor. In the present PS AF ADS, the speed controller algorithm runs on the the OBC, while only the motor controller runs on the UcBoard. In this work, we will implement a new speed controller algorithm to run on the UcBoard. This speed controller will be executed by the motor controller routine, so at some point, they might seem to be the same thing, but the terms will still refer to two fundamentally different concepts.

To deal with differences between car model 1 and 2, there currently exist two different variants of the UcBoard software. In particular, the software for car model 2 is a fork of the software for model 1. As described in Section 2.5, forking a software to address a different set of requirements is usually a quick and efficient solution as opposed to introducing more complex abstractions into the original source code to be able to address both sets of requirements with the same code. Forking is especially paying off when the requirements for the two systems diverge later in the development process. However, in the case for the PS AF model cars, the two different systems, model 1 and model 2, aim to serve the same purpose, but model 2 has some hardware features that model 1 does not have and vice versa. Building a model car with physical components has a significant cost, which means that the older model 1 cars are still valuable for testing and development. This implies that there is also a need to

maintain the software variant for car model 1, so forking might not be an ideal solution in the long run.

2.6.3 Speed control loop

A diagram of the full speed control loop of a PS AF model car is shown in Figure 2.7. The speed controller is implemented on the OBC. The speed controller takes two inputs, the requested speed v_r and the measured speed v_m. The speed controller's output is a drive request, consisting of a driving direction and a drive value u. The UcBoard software executes the drive request by applying a Pulse-Width Modulated (PWM) signal u_PWM to the ESC. The PWM signal has a period of $20\,\mathrm{ms}$ and a pulse width T_P given by

$$\begin{aligned} T_\mathrm{P} &= 1.5\,\mathrm{ms} + \Delta T_\mathrm{P}, \\ \Delta T_\mathrm{P} &\in [-0.5\,\mathrm{ms}, 0.5\,\mathrm{ms}]. \end{aligned} \tag{2.9}$$

A visualization of one period of the PWM signal when driving backwards or breaking is shown in Figure 2.8. ΔT_P represents the pulse width deviation from the neutral position, as $T_\mathrm{P} = 1.5\,\mathrm{ms}$ corresponds to a speed of $0\,\mathrm{mm/s}$. A $\Delta T_\mathrm{P} < 0$ corresponds to forwards drive and a $\Delta T_\mathrm{P} > 0$ corresponds backwards drive or breaking. The ESC is calibrated with a dead zone, so that any $\Delta T_\mathrm{P} \in [\Delta T_\mathrm{P,NFE}, \Delta T_\mathrm{P,NBE}]$ (neutral-forward ended and neutral-backward ended) is interpreted as neutral.

Now, the conversion from drive value u to pulse width deviation ΔT_P is a piecewisely linear mapping, defined differently for the two possible driving directions. For forwards drive, the mapping is from $[1, 1000]$ to $[\Delta T_\mathrm{P,NFE}, -0.5\,\mathrm{ms}]$ for positive drive values and from $[-1, -500]$ to $[\Delta T_\mathrm{P,NBE}, 0.5\,\mathrm{ms}]$ for negative drive values. For backwards drive, the mapping is from $[1, 500]$ to $[\Delta T_\mathrm{P,NBE}, 0.5\,\mathrm{ms}]$.

The observant eye will see that the same range of ΔT_P values is used for backwards drive as well as for breaking. However, there is a significant difference between the two driving modes: When $\Delta T_\mathrm{P} > \Delta T_\mathrm{P,NBE}$, the ESC will only make the motor drive backwards if backwards drive has been unlocked since the last time it was driving forwards. Backwards drive gets unlocked whenever the speed is $0\,\mathrm{mm/s}$ while ΔT_P is neutral ($\Delta T_\mathrm{P,NFE} < \Delta T_\mathrm{P} < \Delta T_\mathrm{P,NBE}$) for $100\,\mathrm{ms}$ continuously. It is the

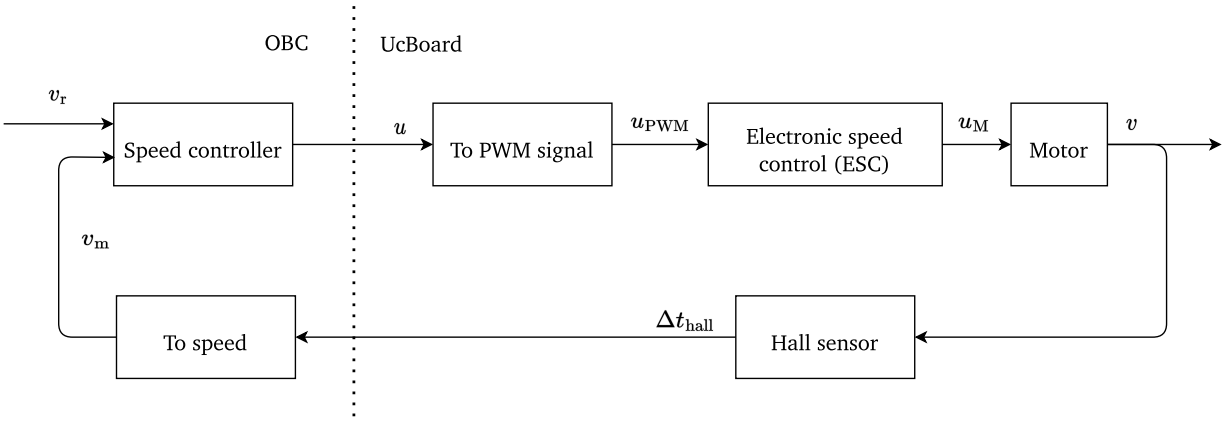


Figure 2.7: Diagram of the PSAF speed control loop (for car model 1).

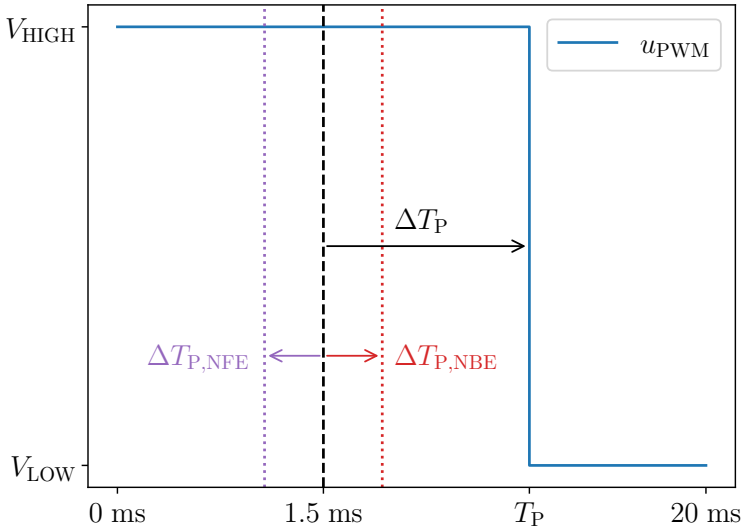


Figure 2.8: Visualization of one period of the PWM signal when driving backwards or breaking.

UcBoard software's task to keep track of whether backwards drive is unlocked in any situation and perform this unlocking process when necessary [39].

The physical PWM signal u_{PWM} is generated by a hardware timer within the microcontroller. u_{PWM} propagates to the ESC, which outputs the direct motor input u_{M}. Now, the motor makes the car accelerate and decelerate, a process resulting in a measurable speed v. The speed is measured by a speed sensor. In Figure 2.7, this is a hall sensor, which is the case for car model 1. As mentioned previously, in model 2, an encoder is used instead, and for model 3 the motor and entire speed control loop is about to totally change. Back on the OBC, the hall sensor output Δt_{hall} is converted to a speed measurement v_{m} for the speed controller.

2.6.4 BLDC motor controller

In this section we will describe the interface between the UcBoard and the BLDC motor controller. As previously mentioned, this is the motor that will be used on car model 3 and is neither present on car model 1 nor on model 2. The motor controller is a dedicated Printed Circuit Board (PCB) and is connected to the UcBoard over an SPI bus where the UcBoard is bus master. The UcBoard interacts with the BLDC motor controller by writing and reading registers. A full overview of available registers and their functions is described in [41]. An important aspect of the BLDC motor controller compared to the ESC used in car model 1 and 2 is that with the BLDC motor controller, the exact requested driving speed in mm/s can be written to a register, and the motor controller will monitor the actual speed and keep it close to this value. With the ESC, the inputted PWM does not describe the exact driving speed, but rather how much force to apply, which leaves a need to monitor and control the exact speed in an external speed control loop.

2.7 Related work

In this section we will take a look at similar research projects to the TU-Darmstadt's PS AF, including other projects targeting the Carolo-Cup, as well as similar applications of RTOSes and FreeRTOS in particular.

In the 2013 project course for the second year students from the bachelor program "Software Engineering and Management" at the Chalmers – University of Gothenburg, a self-driving model car was realized over a period of three months, based solely on Commercial-Off-The-Shelf (COTS) hardware and firmware components [42]. Just like in the PS AF, the architecture of the Chalmers car contains two separate computer systems connected by USB, a microcontroller board, in this case a STM32F4 Discovery Board, for controlling sensors and actuators, and a more powerful Linux machine in charge of high-level tasks including image processing and path planning. The software for the Chalmers microcontroller board uses the RTOS ChibiOS/RT [43] for scheduling and interfacing the steering and propulsion motors.

In a work by Perciński from 2018, the architecture of the Warsaw University of Technology's competition car for the Carolo-Cup of the same year is described [44]. Also here, computations are split across a microcontroller, in this case an STM32F7 and a more powerful Linux machine. However, interestingly enough, in the Warsaw car, the microcontroller takes the commander role, while the Linux machine acts as a worker, interfacing the camera and taking care of image processing. The Warsaw car utilizes FreeRTOS for priority-driven preemptive scheduling.

The new backbone of the 6th generation of the University of Magdeburg's Carolo-Cup car is described in the bachelor thesis of Tim Wiesner and Maximilian Grau [45]. This car has a highly modular design, featuring nine PCBs with a total of five STM32F103RE microcontrollers. Also these microcontrollers' software utilizes FreeRTOS for priority-driven preemptive task scheduling as well as task synchronization.

While both [44] and [45] claim to utilize FreeRTOS, none of these works evaluate how it actually affects the system's robustness or responsiveness. This work, on the other hand, aims to evaluate in detail how the application of an RTOS impacts the robustness and responsiveness of the microcontroller it runs on.

3 Strategy

In this chapter, we present our developed strategies for implementing and verifying our solutions for the UcBoard software. In Section 3.1, we describe methods needed when applying an RTOS to a non-RTOS codebase. In Section 3.2, we derive a suitable low-level speed controller algorithm for the UcBoard. Lastly, in Section 3.3, we will derive a strategy for merging the two UcBoard software variants into the same source code by using configuration files. In addition, in Section 3.4, we will derive a set of real-time constraints for the UcBoard software based on the system's physical environment, so that our solutions can be evaluated for their ability to meet those constraints.

3.1 Routine conversions

In this section, we will describe two methods for converting routines between different routine types. The first method converts a systick callback, a function called directly by the systick ISR, to a FreeRTOS timer callback. The primary benefit of this conversion is that it brings the routine out of the ISR priority space and into FreeRTOS' priority space, which allows its priority to be superseded by high priority tasks. However, it does not give the routine free priority assignment, since all timer callbacks have the same priority. If we want the routine to get the same individual priority assignment capabilities, as a task, we can use the second method described here to further convert it to a task.

3.1.1 Convert a systick callback to a FreeRTOS timer callback

As we saw in subsubsection 2.6.2, in the present UcBoard software, most routines are implemented as systick callbacks. In principle, we can describe the exact routine of a systick callback by its callback function C and its callback period T. The same is true for FreeRTOS timer callbacks. This means that we can implement the same routine by executing C with a period T with a FreeRTOS software timer instead of calling it manually in the systick ISR. The routine will now execute with the priority of the timer task. If individual priority assignment is needed, it can be further converted to a task with the method described in Section 3.1.2.

3.1.2 Convert a FreeRTOS timer to a FreeRTOS task

We will now present a generic method to port a FreeRTOS timer to a FreeRTOS task. This can be useful if a routine that is currently implemented as a timer callback is considered important and should be given a higher (or lower) priority than the other timer callbacks. While FreeRTOS timer callbacks actually can have a variable request period, this approach is limited to fixed-period callbacks only.

Given we have a FreeRTOS software timer executing the callback function C at a fixed period T, we can port the timer to execute C in its own task as shown in Algorithm 1, where TaskForTimer is executed as a FreeRTOS task. This is the task version of our timer. In this way, C is executed with the priority of this task, instead of the priority of the timer task. If TaskForTimer is given a higher priority than the timer task, the scheduler will, if needed, preempt the timer task for the benefit of C whenever it is time for C to execute. The downside is of course that C is now executed in a dedicated task which requires a dedicated execution stack.

Algorithm 1 A task dedicated to execute a timer callback C with a fixed interval T.

```
procedure TaskForTimer(C, T)
    loop
        Delay T
        C()
    end loop
end procedure
```

The proposed method is a very simple approach that does not require any adaptions to the implementation of the callback function C. However, in many cases, a more tailored approach would be beneficial, e.g. if C only needs to do some processing when an item can be received from a queue, the delay could be replaced by a blocking call to `xQueueReceive` and the code that should only run after an item is received from the queue can run directly afterwards. This would also have the benefit of making the queue reception event driven, rather than the previous polling-driven approach, which means that an item from a queue can be received and further processed immediately and no processing time is wasted on polling when there is nothing there.

3.2 Speed controller

In this section, we describe the speed controller algorithm we will implement on the UcBoard. As explained in Section 2.4.6, a combined feed-forward and PI-controller is often beneficial for servomechanisms, such as a speed controller. The mathematical model and control theory behind such a speed controller is already described in Section 2.4.6. The full control structure for the speed controller is shown in Figure 3.1. The speed controller is to be implemented on a microcontroller, and thus has to be evaluated in discrete time steps. We therefore propose using a discrete-time combined feed-forward

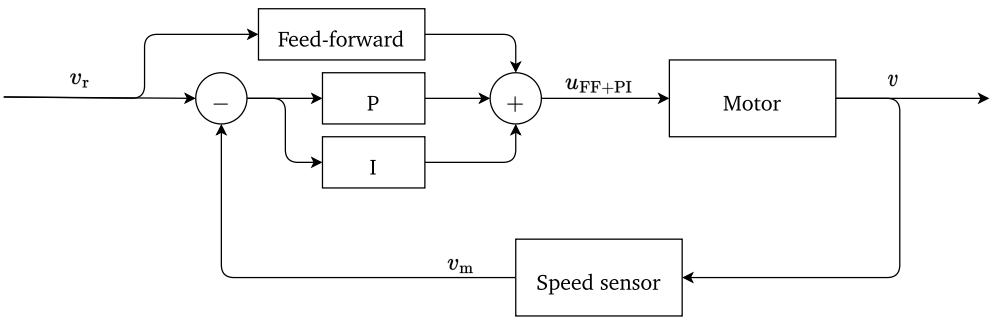


Figure 3.1: Control structure for speed controller.

and PI-controller as our speed controller as described in Section 2.4.6. The drive value to apply for time step $k+1$ is then given by

$$\begin{aligned} u_{k+1} &= U_{n-1} + \frac{U_n - U_{n-1}}{V_n - V_{n-1}} \cdot (v_{\mathrm{r},k} - V_{n-1}) \\ &\quad + K_\mathrm{P} \cdot (v_{\mathrm{r},k} - v_{\mathrm{m},k}) + K_\mathrm{I} \cdot \sum_{i \le k} (v_{\mathrm{r},i} - v_{\mathrm{m},i}) \cdot T_\mathrm{S} \\ v_\mathrm{r}(t) &\in [V_{n-1}, V_n\rangle, \end{aligned} \tag{3.1}$$

where U_n and V_n for $n \in \{0 \dots N\}$ are the reference drive values and speeds, $v_{\mathrm{r},k}$ is the requested speed in time step k, $v_{\mathrm{m},k}$ is the measured speed in time step k, T_S is the sampling period, K_P is the proportional constant and K_I is the integral constant. The reference drive values, T_S, K_P and K_I are application specific tuning parameters. Their optimal values are highly dependent on the specific case to control. This means we will consider tuning a part of our experimental setup, and the final value an experimental result.

3.3 Merge UcBoard source code variants

To merge the two variants of the UcBoard software source code, we will define a set of configuration options to store in different two configuration files, one for each car model. The configuration options must be widely enough configurable to cover all functionality currently provided by the car model 1 software as well as by the car model 2 software. Then we will modify the source code for car model 1 to use these configuration options instead of the current hard-coded values. This modified version of the car model 1 software source code will be the unified source code for the software for both car models. The modification of the car model 1 software involves to a large extent applying existing modifications in the car model 2 software. As described in Section 2.5, compile-time configuration options are generally more flexible than run-time configuration options, since they can modify bigger aspects of the software. For this application, the fact that the software must be re-compiled and the new version must be flashed onto the UcBoard when configuration options are changed is not a major drawback, as the people that will eventually change the configuration options in the future will not be non-technical consumers, but rather researchers and students at the TU-Darmstadt.

3.4 Deadlines

In this section we will derive a set of real-time constraints, resulting in two deadlines, for the UcBoard software on the PS AF model car. These deadlines are not derived from a formal specification of the car or the UcBoard, but rather from common sense. The purpose of this section is not to derive any formally required real-time deadlines for the UcBoard, but rather to establish a meaningful real-time context in which the UcBoard can be seen and evaluated.

3.4.1 DRV reaction time

The first deadline we derive is how quickly the UcBoard has to change the motor's PWM signal after receiving a DRV command. The faster the car is driving, the more crucial it is to react quickly to avoid potential crashes.

We can define a constraint that the UcBoard should never let the car drive longer than a reaction distance s_{D}, where D is short for "deadline", after a DRV command is received before the motor's PWM signal is updated accordingly. This gives a reaction time $r = \frac{s_{\mathrm{D}}}{v}$, where v is the current speed of the car. Since the speed is constantly changing, we must require the maximum reaction time, or deadline, to be $r_{\mathrm{D}} = \frac{s_{\mathrm{D}}}{v_{\mathrm{top}}}$, where v_{top} is the top speed of the car, to make sure the requirement is met for every possible speed. For example, if we require $s_{\mathrm{D}} = 100\,\mathrm{mm}$, and assume $v_{\mathrm{top}} = 2500\,\mathrm{mm/s}$, then we get $r_{\mathrm{D}} = 40\,\mathrm{ms}$.

If the UcBoard is not able to meet this deadline, it can in the worst case can make the model car crash into an object it otherwise would not crash into. As the entire purpose of the ADS is to navigate the car through a track without crashing, we can consider a crash a complete system failure. Due to this, if the UcBoard fails to meet a single DRV reaction time deadline, it might render the UcBoard completely useless, meaning that the DRV reaction time is a hard real-time constraint.

3.4.2 Response time

Partly because it important for the OBC to get timely responses to know what is going on and partly because it is a lot more practical to measure, we will to a large extent use the response time R as an estimator for the reaction time r of the respective command. The response time of a command is the time it takes from the command is received by the UcBoard until the response is returned. This metric is usable for all commands since all commands should always return a response. Depending on the case, this may or may not be a good estimator. For the cases we will study, the causes for differences in the reaction time is not the command processing and execution itself, but rather other tasks that occupy CPU time. Dependent on how the command is implemented, the requested action is sometimes guaranteed to be completed before the response is sent. For such commands, the response time is an upper bound for the reaction time ($R > r$).

As a stand-alone real-time requirement, the DRV response time is rather soft, as the only information being replied back after a DRV command is a confirmation that the order was received. This is of course dependent on how the OBC's algorithm is implemented in regard to dealing with late responses from the UcBoard. But the response time for another command could be a rather hard real-time constraint, if the response contains vital information the OBC actually needs for navigation in real-time.

4 UcBoard software implementation

In this chapter, we describe the implementation of our solutions for the UcBoard software. Firstly, in Section 4.1, we describe how we bundle the FreeRTOS source code together with the UcBoard's application source code. In Section 4.2, we describe two alternative ways to implement the UcBoard software's routines with FreeRTOS tasks and timer callbacks. In Section 4.3 we will describe how the source code variants are merged by using configuration options. The implemented interface to the BLDC motor controller is described in Section 4.4, in Section 4.5 we describe the implementation of the new motor controller, and in Section 4.6, the new speed controller. Our implementation will be compared to the present solution in the PS AF ADS where this is relevant.

An outline of the file structure for the UcBoard software source code can be found in Section A.1. The full source code can due to copyright reasons not be published, but an extract of interesting sections contributed by this work alone is given in Section A.3.

4.1 Bundle FreeRTOS together with the application source code

In our implementation, we bundle FreeRTOS together with the application source code directly, that is, the official FreeRTOS kernel source code is cloned into the FreeRTOS-Kernel subdirectory in the application source code. Then all relevant source files are added to the compilation recipe. Another opportunity would be to use a program called STM32 CubeMX to bundle the FreeRTOS source code automatically together with the application source code through the Common Microcontroller Software Interface Standard (CMSIS) RTOS API version 2 [46] like it is done for the microcontroller software for the 6th generation of the Carolo-Cup car of the University of Magdeburg [45]. A visualization of how the application source code is compiled and linked together with FreeRTOS in the two cases is shown in Figure 4.1.

We choose to bundle FreeRTOS with the application source code directly because it gives us more control of the code being added. Downloading a piece of open source software and adding it to the compile recipe does not require much effort, using an external tool to automate this is for the most part unnecessary. A major difference between bundling directly and using CubeMX is that with CubeMX, FreeRTOS is bundled through the CMSIS RTOS API, meaning that the API functions to call in the application code will have different names and potentially use different types and parameters than when the FreeRTOS API is used directly. The benefit of using the CMSIS API is that it is a standard API that can be used with a variety of different RTOSes, while the FreeRTOS API is only applicable for FreeRTOS. This makes an application written with the CMSIS RTOS API easier to adapt to a different RTOS if desired [47]. On the other hand, this leads to an extra layer of abstraction and reliance on free, but not open-source software (CubeMX) for source code management. Since the main purpose of the PS AF is to provide an education platform for students, rather than an industry-standard autonomous

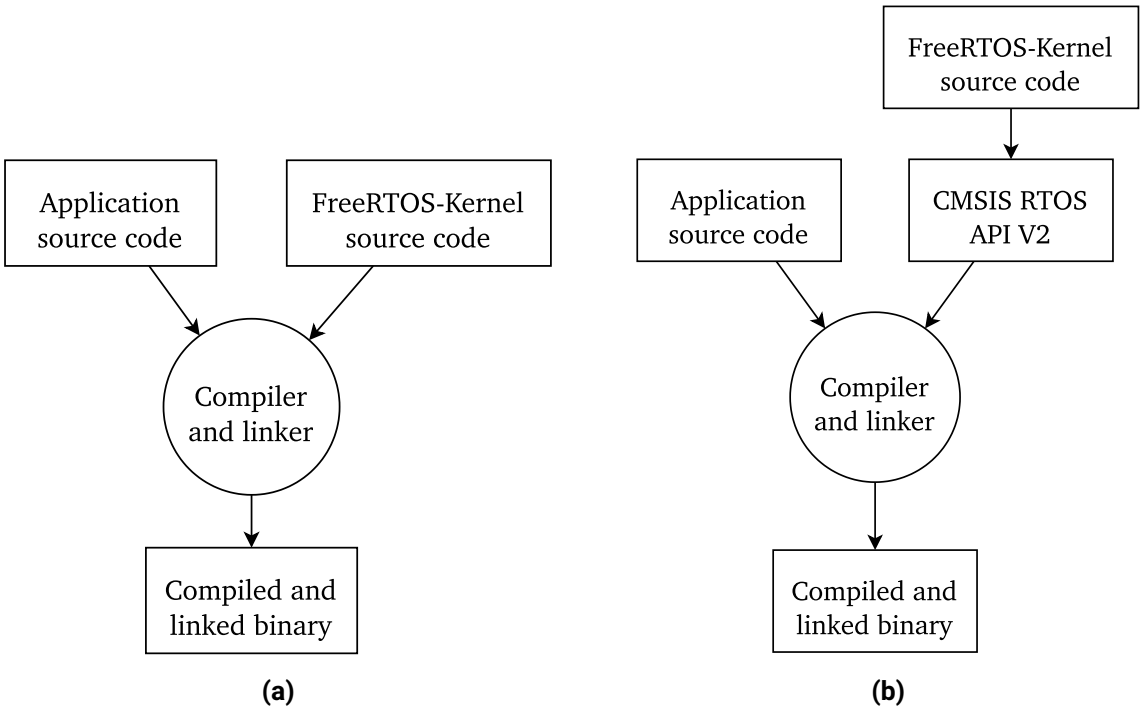


Figure 4.1: Visualization of how FreeRTOS is bundled with the application source code directly (a) and with the CMSIS RTOS API (b).

model car, it will likely not be relevant to replace FreeRTOS with a commercial RTOS anytime in the near future. Therefore, bundling FreeRTOS directly is seen as the most appropriate solution for the PS AF UcBoard software.

4.2 Routine implementation

We will utilize the methods from Section 3.1 to convert the routines of the UcBoard software from their present non-RTOS implementation to a suitable RTOS-based counterpart. The routines in question are those that were described in subsubsection 2.6.2. We implement two slightly different routine configurations to evaluate against the present non-RTOS UcBoard software. The first, configuration A, using less memory than the other, configuration B, which should have more robust reaction and response times. The only point in which they differ is whether the communications routines are implemented as routines or tasks. We want to evaluate these two configurations to see if the choice of implementing a routine as a timer callback or task actually has implications for it's ability to run, which we expect it to.

4.2.1 Configuration A

Routine configuration A uses only one application task in addition to the two system tasks. The only routine being implemented as a task in configuration A is the motor controller. The remaining 10 routines are implemented as software timers. The configuration is shown in Table 4.1. The routines

Table 4.1: Routine configuration A.

Routine	Type of implementation	Priority
Motor controller	Task	4
Steering controller	Timer callback	2
Voltage measurement	Timer callback	2
Hall sensor	Timer callback	2
IMU	Timer callback	2
US	Timer callback	2
DAQ	Timer callback	2
CarUI	Timer callback	2
SYS LED	Timer callback	2
Communication RX	Timer callback	2
Communication TX	Timer callback	2

are sorted by decreasing priority. The system tasks are not listed, since they are not routines as seen from the application's perspective. For the timer callbacks, the priority of the timer task is indicated as the timer callback's priority. Figure 4.2 shows all FreeRTOS tasks in the system and all timer callbacks as functions called by the timer task. Since most routines, including the communication routines, are implemented as timer callbacks, we should expect similar response times to the present non-RTOS solution, as the communication routines are on the critical path of the response time of all commands.

The reasoning for why specifically the motor controller is implemented as a task is explained in Section 4.5. In the microcontroller software for the Warsaw University's Carolo-Cup 2018 competition car, motor control is not implemented as a dedicated task, but instead steering is. In our case, the steering controller is directly ported from a systick callback in the current non-RTOS UcBoard software, which means that it would require more rework to utilize the FreeRTOS API. However, porting it to a dedicated, as explained in Section 3.1, would let us assign the steering controller a higher priority than the other timer callbacks, giving it a better chance of running when the system is overloaded.

4.2.2 Configuration B

Routine configuration B uses three application tasks in addition to the two system tasks. The routines implemented as tasks in configuration A are the motor controller, communication RX and communication TX. The remaining eight routines are implemented as software timers. The configuration is shown in Table 4.2. Like in Table 4.1, routines are sorted by decreasing priority and system tasks are not listed. For the timer callbacks' priority, the priority of the timer task is listed. The communication tasks are given high priorities to increase the robustness of commands' response and reaction times, since they are both on the critical of the response time of all commands. Communication RX is also on the critical path of the reaction time of all commands, and has therefore a priority of 9, while communication TX only has 8. Figure 4.3 shows all FreeRTOS tasks in the system and all timer callbacks as functions called by the timer task.

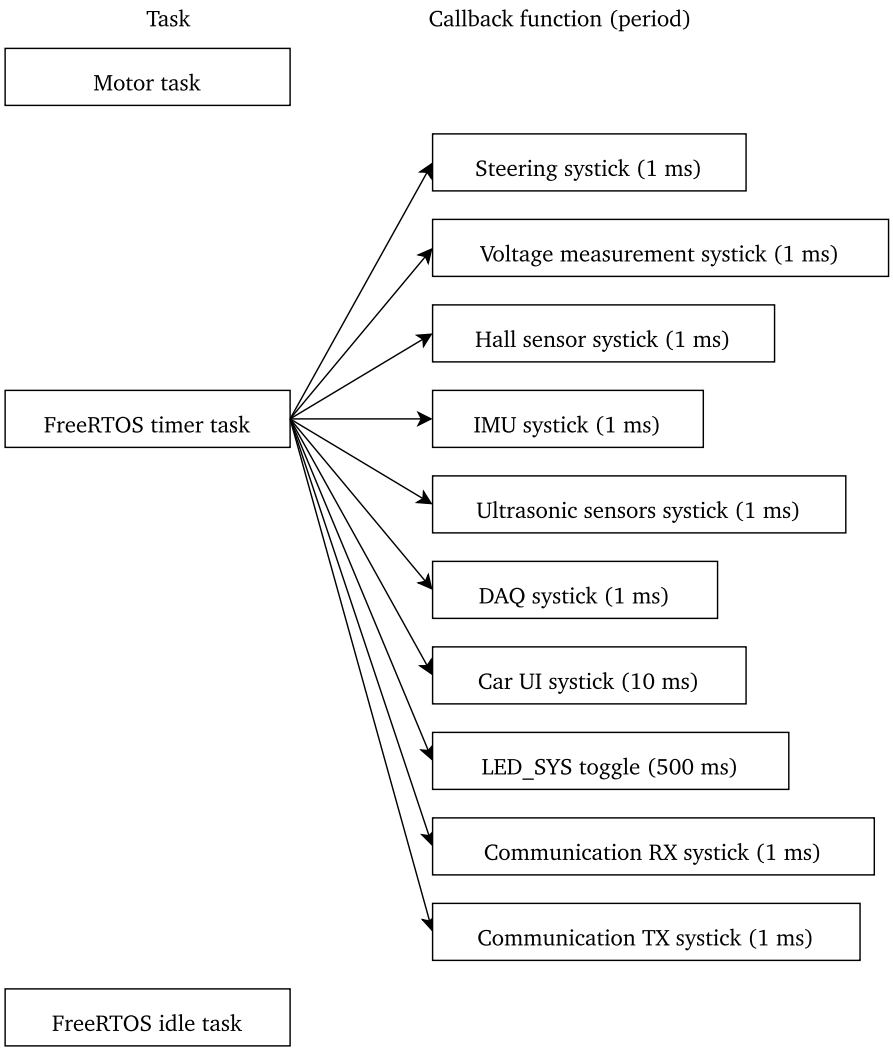


Figure 4.2: FreeRTOS tasks and timer callbacks in configuration A.

Table 4.2: Routine configuration B.

Routine	Type of implementation	Priority
Communication RX	Task	9
Communication TX	Task	8
Motor controller	Task	4
Steering controller	Timer callback	2
Voltage measurement	Timer callback	2
Hall sensor	Timer callback	2
IMU	Timer callback	2
US	Timer callback	2
DAQ	Timer callback	2
CarUI	Timer callback	2
SYS LED	Timer callback	2

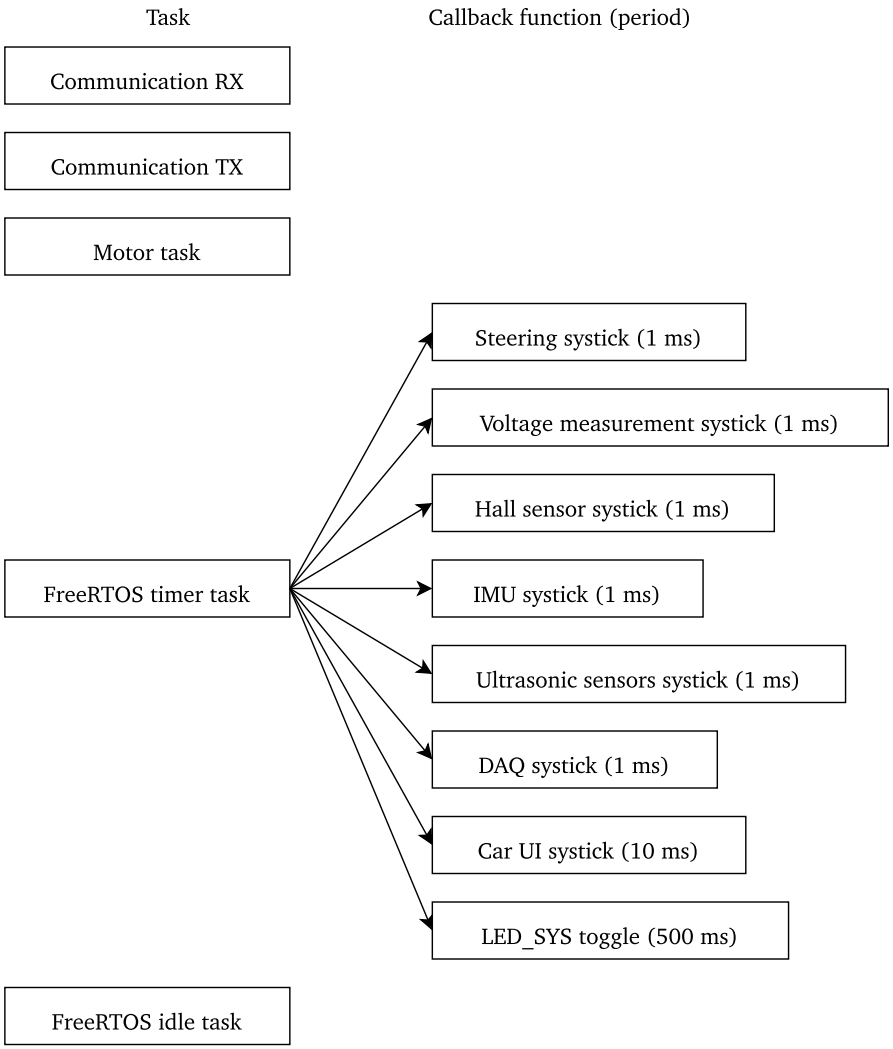


Figure 4.3: FreeRTOS tasks and timer callbacks in configuration B.

4.2.3 Task stack sizes

To get a rough idea of how large stack a task needs, we use the script analyze_stack_usage.py, which prints an overview of the functions with the highest stack usage. The script simply extracts the stack usages of functions as reported by the compiler and does not take into account the stack usage of other functions called by the function in question. Then we apply our knowledge about which functions are used by which tasks to select the stack size to allocate for each FreeRTOS task. This is not an ideal solution, finding an exact upper bound using a more sophisticated tool like StackAnalyzer would have been more beneficial, but the time did not allow for that in this work.

4.3 Merge source code variants

As described in Section 3.3, the UcBoard source code variants for the different car models are merged by defining and using a set of configuration options. As the UcBoard software is written in C, and C has a powerful preprocessor macro system, we can use this for realizing compile-time configuration options for the UcBoard software. In the finally resulting joint codebase, we can store configuration files for both cars beside each other in a directory, specifically, src/config. The config options can be imported in source files by including the main config file, config.h. config.h again includes config options from a car-specific configuration file. By changing which car-specific configuration file is included in config.h, one can change which car-specific configuration file is included in all source files. The implemented configuration options with their configured values for car model 1 can be found in Section A.3.2, and for car model 2 in Section A.3.4. A configuration file for car model 3 will at some point need to be added, but that is beyond the scope of this work.

4.4 BLDC motor interface

A higher-level interface to the motor controller board for the BLDC motor is implemented in the source file bldc.c. This interface provides a function `bldc_setTargetVelocity` which can be used by the motor controller to update the target velocity of the speed controller. To do anything else, like e.g. configure parameters for the speed controller of the BLDC motor controller, the BLDC command added to the serial protocol can be used by the OBC to write or read the BLDC motor controller's registers directly.

4.5 Motor controller

The motor controller routine is implemented as a FreeRTOS task both in routine configuration A and in routine configuration B. As we will see, the motor controller implementation can be significantly simplified by having the ability to pause for a while while preserving the execution stack.

The entrypoint of the motor controller is the function `task_motorController` (carbasicfcts.c). This function runs an infinite loop that starts each iteration with receiving a request from the drive request queue. If the drive request queue is empty, calling `xQueueReceive` will block the task until a request

has arrived or until the specified timeout has run out. If the speed controller is running, the timeout is set equal to the sampling period T_S. In this way, processing time required is minimized, as it does not spend processing time on repeatedly checking if there is a new request, but rather waits for FreeRTOS to unblock it when something can be received from the queue. Except for when the speed controller is running, since it then has to execute the speed controller algorithm at a steady rate.

The drive request is received as a `DriveModeValue_t` structure. As shown in Listing 4.1, it consists of a drive mode and a value. The drive mode is an enumerated type which tells whether to apply the value directly as T_{PWM}, as a forwards or backwards drive value u or as a cruise speed v_r for the speed controller. Storing these values in a structure is necessary to pass them together through a queue.

Listing 4.1: The enumerate type EnDrvMode_t and the structure type DriveModeValue_t.

```
typedef enum EnDrvMode_ {
    DRVMODE_OFF,
    DRVMODE_FORWARDS,
    DRVMODE_BACKWARDS,
    DRVMODE_DIRECT,
    DRVMODE_RC,
    DRVMODE_CRUISE,
} EnDrvMode_t;

typedef struct {
    EnDrvMode_t eMode;
    int32_t iValue;
} DriveModeValue_t;
```

If the drive mode is `DIRECT`, the function `motor_controller_setPwm` is used to set T_{PWM} directly. If the drive mode is managed (either `FORWARDS` or `BACKWARDS`), the function `motor_controller_managedRequest` is used to handle the managed request. If the drive mode is `CRUISE`, the function `motor_controller_cruiseRequest` is used to handle the cruise request. These three request types are hierarchically ordered into distinct abstraction levels. Internally, `motor_controller_cruiseRequest` generates a managed request and calls `motor_controller_managedRequest` to execute it, and `motor_controller_managedRequest` calls `motor_controller_setPwm` internally to apply values for T_{PWM}. A call graph for this is shown in Figure 4.4. This is a good way of structuring code, since the logic for setting PWM value directly is also needed for managed requests and cruise requests, and in this way the same logic exists only one place in the code. In comparison, in the present UcBoard software, the entire motor controller code is contained within one function. But here, there is no speed controller, and thus no ”cruise” drive mode, leading to only two possible abstraction levels for a drive request. The current implementation does not use a significant amount of code duplication to get everything into one function, but achieving this gets exponentially more complex as the number of drive request abstraction levels grows.

The motor controller implementation can be simplified by having the ability to pause while preserving the stack at three particular occasions. When changing drive direction from forwards to backwards, the motor controller has to wait $100\,\text{ms}$ in neutral for the ESC to unlock backwards drive, as explained in Section 2.6.3. To make sure the car is standing still before this happens, it should brake for $1.5\,\text{s}$. Breaking until the measured speed is 0 would be a more sophisticated option, but it is more complex, and will face issues when relying on measurements from the hall sensor, since a hall sensor stops producing measurements when the car stops. After the motor is powered on, the motor controller should wait $1.5\,\text{s}$ before applying the requested drive value, because the motor needs some time from

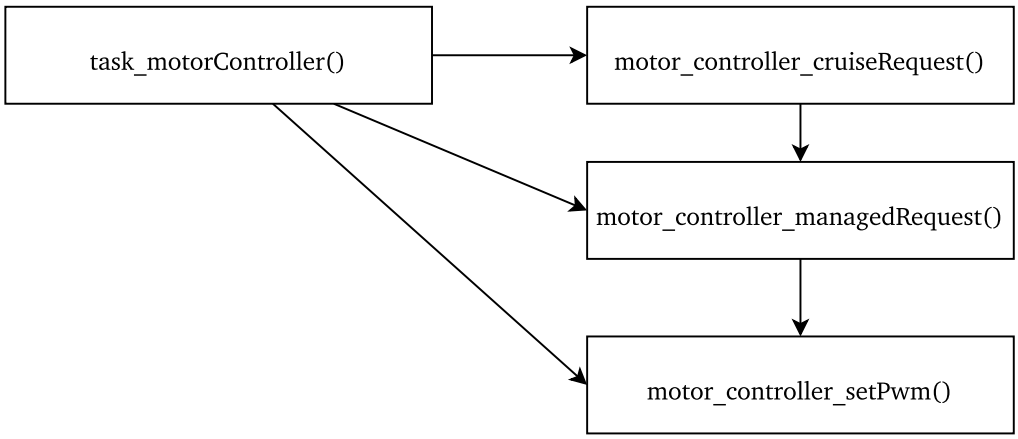


Figure 4.4: Motor controller call graph.

it is powered on until it is ready. When the motor controller is implemented as a task, it can simply use the blocking FreeRTOS API function `vTaskDelay` in these situations, and the task will be blocked for the specified amount of time. If the motor controller was implemented as a timer callback, it would require dedicated logic to store necessary state outside the stack and repeatedly increment a counter until it can proceed as explained in Section 2.3.5. This is what the present non-RTOS motor controller does when changing driving direction, and it is a major obstacle for good code readability.

If the configuration option `CONFIG_MOTOR_TYPE_BLDC` is set to 1, the behavior is a bit different, as it means that the car model used has a BLDC motor managed by a BLDC motor controller. In that case, the previously described function `bldc_setTargetVelocity` is used instead of the recently described `motor_controller_cruiseRequest` for drive mode `CRUISE`. If the drive mode is `DIRECT`, `FORWARDS` or `BACKWARDS`, the request is ignored.

4.6 Speed controller

The speed controller algorithm is described in Section 3.2. The algorithm is implemented by the function `motor_controller_cruiseRequest`, which is called once per time step by the motor controller task. The calculation of u as described in Equation 3.1 is performed by the function `cruise_control_getNewDrvModeValue`. For a time step k, the function requires two arguments, the measured speed of the car, $v_{\mathrm{m},k}$, and the requested speed, $v_{\mathrm{r},k}$. The function returns the drive value u_{k+1} to apply until the next time step as a `DriveModeValue_t` structure.

5 Experimental setup

In this chapter, we describe the experimental setup used to evaluate our solution. The results from these experiments are presented in Chapter 6. In Section 5.1, we describe how we do response time measurements, and in Section 5.2, we describe the tuning of the feed-forward reference values and PI-controller constants for the speed controller.

5.1 Response time measurements

As a metric for the robustness of the UcBoard software, we want to measure the response time R of the DRV command when the UcBoard is under heavy load. This means we need a reliable way to apply a computational load that is heavy enough to significantly affect the response time of the DRV command. By abusing the existing functionality of the UcBoard through e.g. activating a lot of background tasks and at the same time spam it with requests, the system might get overloaded, and we can see a drop in response time for certain commands. Unfortunately, the present UcBoard software is well designed and we were not able to overload it enough to make a significant impact. Therefore, we will instead implement an artificial heavy operation in the UcBoard's software on purpose to consume computational resources.

5.1.1 Artificial heavy operation

To overload the UcBoard software, we will use LED A. In detail, each time LED A is powered on, an artificial heavy operation is performed. The operation consists of writing the integer value 42 to a variable `foo` a large number of times N. The operation is performed by the CarUI routine, and not the Communication RX routine, since we would like the operation to run in the background after the response for powering the LED on is given. $N = 2\,000\,000$ was found to be a reasonable value, making the operation take approximately 500 ms.

The implementation code for the heavy operation is shown in Listing 5.1. The code is to be added to the beginning of `carui_do_systick_10ms` (carui.c). The variable `s_bDelayDone` ensures that the operation only happens once each time the LED is powered on.

Listing 5.1: Artificial heavy operation to run when LED A is powered on.

```
1 static bool s_bDelayDone = false;
2 if (f_aNewLedSeq[0].mode == LEDMODE_ON) {
3     if (!s_bDelayDone) {
4         volatile int foo;
5         for (uint64_t i = 0; i < 2000000; i++) {
6             foo = 42;
```

```
        }
        s_bDelayDone = true;
    }
}
else {
    s_bDelayDone = false;
}
```

5.1.2 Automated measurement script

A Python script `drv_response_time` (automate.py) is implemented to automate the experimental measurement process. It performs the following procedure 50 times:

1. Select a random delay value $d \in [0.1, 1]$.
2. Open a serial connection to the UcBoard.
3. Reset UcBoard software.
4. Drive forwards with a drivevalue of 300.
5. Wait $2\,\mathrm{s}$.
6. Power on LED A.
7. Wait d.
8. Brake.
9. Look through the serial response file and find the measured response time R_m for the brake command.
10. Append d and R_m to a CSV file.

The UcBoard should return a response to every command. To make sure the UcBoard is ready, `drv_response_time` always waits for the response before going further. This means that the delay d is defined from the response for powering on LED A is received until the command to brake is sent.

5.1.3 Applicability of response time as estimator for reaction time

How applicable the response time R is as an estimator for the reaction time r for the DRV command turns out to be dependent on the routine configuration. We will now explain why.

For the DRV command, actuator input is applied by the motor controller routine and not directly by the communication RX routine. But in routine configuration A and B, the motor controller is a dedicated task which uses a blocking API call to fetch drive requests from the drive request queue. In configuration A, the motor controller also has higher priority than the communication RX routine, which means that when communication RX pushes a request to the drive request queue, it will immediately be preempted by the scheduler until the motor controller is done processing it. In routine configuration A, the response time R is therefore an upper bound for the reaction time r. For routine configuration B, however both communication RX and TX have higher priority than the motor controller, which means that the response *can* be sent before actuator input is applied. For the current UcBoard software it is

a different story, but since the motor controller only polls for new requests once every 1 ms and the communication TX polls continuously, neither here is R an upper bound for r.

However, in all cases the heavy operation runs in the CarUI routine, and the CarUI routine is in none of the cases able to block the motor controller from running without also blocking communication TX from running. This means that as long as the heavy operation in the CarUI is the most significant contributor to system overload, the response time R is still a decent estimator for the reaction time r.

5.2 Speed controller tuning

In this section, we will describe how we tune the application specific parameters of the speed controller, $U_n, V_n \; \forall \; n \in \{0 \dots N\}$, T_S, K_P and K_I. We will firstly undertake the feed-forward reference values $U_n, V_n \; \forall \; n \in \{0 \dots N\}$ in Section 5.2.1, and then the PI-controller parameters T_S, K_P and K_I in Section 5.2.2.

5.2.1 Tune feed-forward reference values

Good feed-forward reference reference values, $U_n, V_n \; \forall \; n \in \{0 \dots N\}$, can quickly be found experimentally for small values of N by conducting a practical experiment: We rig up a model car with the UcBoard, place it on the floor and connect to the serial interface over bluetooth from a Personal Computer (PC). Then we let the car drive around in a circle with different drive values and record the measured speed. In particular, we apply N different drive values $U_n \; \forall \; n \in \{1 \dots N\}$ and measure their equilibrium speeds. The script `find_feed_forward_values` (automate.py) is implemented to automate this process on the PC. This is done by repeating the following steps for all $n \in \{1 \dots N\}$:

1. Setup the DAQ to log hall sensor time deltas over the serial protocol.
2. Apply U_n as the car's drivevalue.
3. Wait 15 s.
4. Stop DAQ logging and brake.
5. Calculate the average speed V_n the last 5 s.
6. Append U_n and V_n to a CSV file.

As for the selection of values for N and $U_n \; \forall \; n \in \{1 \dots N\}$, we will use all drivevalues $U_n \geq 200$ that are divisible by 100, for both driving directions. As the maximum drive value is 1000 for forwards drive and 500 for backwards drive, this gives us a total of $N = 13$ repetitions of the steps above. We expect the distribution of U_n, V_n pairs to be mostly linear, which means that $N = 4$, giving us two pairs in each driving direction would in theory be satisfactory. $N = 13$, however, gives us a better margin to cope with minor non-linearity. Additionally, the drivevalue 0 is for all purposes expected to have an equilibrium speed of 0 mm/s, so this is also added to the results, without going through the above mentioned steps. This gives us in total $N = 14$ drive value/equilibrium speed pairs for the speed controller. In addition to producing a CSV file, `find_feed_forward_reference_values` also generates `.h` file containing C macros for the UcBoard software config file. This simplifies the process of updating the feed forward reference values in the UcBoard software in the future.

5.2.2 Tune PI-controller parameters

[28] gives a profound introduction to theory and systematic rules for PI-controller tuning. However, we will not apply any of this in this work, as the main motivation for the speed controller is that the OBC's interface to car model 1 and 2 achieves compatibility with the next generation of PS AF model cars. A solution providing "good enough" performance is perfectly satisfactory. For this reason, we will take a simple try-and-observe approach, by observing the impulse response of the speed controller for a limited set of parameter values.

In a similar fashion as for finding feed forward reference values, we measure the final impulse response of the speed controller by driving the car and measure speed through the DAQ over the serial protocol. The procedure is as follows for a target speed v_r:

1. Setup the DAQ to log hall sensor time deltas over the serial protocol.
2. Apply v_r as the speed controller's target speed.
3. Wait $10\,\mathrm{s}$.
4. Stop DAQ logging and brake.
5. Write measured speed values with timestamps to a file.

The measured impulse response is specific to given values of the parameters T_S, K_P and K_I. We ended up measuring the impulse response for 56 different combinations of K_P and K_I, which are all $K_\mathrm{P} \in \{0, 50, 100, 200, 500, 1000, 2000\}$ $(\mathrm{m/s})^{-1}$ and all $K_\mathrm{I} \in \{0, 5, 10, 20, 50, 100, 200, 500\}$ m^{-1}. To limit the extensiveness of our speed controller evaluation, we did not test multiple values of T_S, and went with $50\,\mathrm{ms}$. This is a reasonable value, since the ESC is controlled by a PWM signal with a period of $20\,\mathrm{ms}$, and any value lower than this would make the speed controller update the drive value more often than the ESC can sense a drive value change. By selecting $50\,\mathrm{ms}$ our T_S is above this with a good margin. Having a too high T_S is undesired due to the sampling theorem [28].

For each combination of parameters, we measure the impulse response for three different target speeds v_r to verify how the parameters work for a greater range of target speeds. We use the three target speeds $0.5\,\mathrm{m/s}$, $1\,\mathrm{m/s}$ and $1.5\,\mathrm{m/s}$. These are selected since they represent the speed controller's typical operating range well.

As a scalar metric of how responsive the speed controller is with a given set of parameters, we will use the average acceleration time $\overline{T_\mathrm{a}}$ for the three selected target speeds. By acceleration time T_a, we mean the time passed from the request is given until the car reaches the requested speed for the first time. The average acceleration time is the arithmetic mean of the acceleration times,

$$\overline{T_\mathrm{a}} = \frac{1}{N} \sum_{i=0}^{N-1} T_{\mathrm{a},i}, \tag{5.1}$$

where N is the number of target speeds tested, 3 in our case, and $T_{\mathrm{a},i}$ is the acceleration time for target speed i. A low value for $\overline{T_\mathrm{a}}$ implies that the speed controller responds quickly to changes in the target speed, which is good. This is not a perfect way to summarize the speed controllers, since a speed controller that undershoots can get an artificially high value, even infinitely high if it never reaches the exact target speed at all. But it is very easy to understand.

6 Results

In this chapter, we will present the results from our implementation and experimental setup. First of all, we present the selection of stack sizes for FreeRTOS tasks in Section 6.1, then the resulting memory usage of the implementation in Section 6.2. In Section 6.3, we present our response time measurements, and in Section 6.4, results for the speed controller.

6.1 Stack sizes

The output from the stack usage analysis script is listed in Section B.1, and based on this, we select the stack sizes for the FreeRTOS tasks as shown in Table 6.1. The stack usage analysis showed that there are only three functions that have a stack usage above 512 B, and they are just used in a few specific contexts (`eeprom_init` is not even used at all). One of these contexts occur in the Communication RX (`display_printerror`). Most command callbacks (functions starting with `cmd_`) are among the functions with the greatest stack usage between 100 B and 512 B. The command callbacks are called by the communication RX task, but never at the same time. Based on this we select a stack size of 4 KiB for the communication RX task to have something with a good margin. Since the timer task is supposed to be flexible and be able to execute any routine in the system, it also gets a stack size of 4 KiB. The communication TX and the motor controller tasks only call a few smaller functions, so for them, 1 KiB should be enough. And the idle task does nothing, so it gets 520 B, which is FreeRTOS' officially recommended value for the Arm Cortex-M4 processor series [48].

Table 6.1: Chosen stack sizes for FreeRTOS tasks.

Task	Stack size
Communication RX	4 KiB
Communication TX	1 KiB
Motor controller	1 KiB
Timer task	4 KiB
Idle task	520 B

6.2 Memory usage

A comparison of the memory usage of the present UcBoard software compared to the new one, using FreeRTOS, with routine configuration A and with routine configuration B, is shown in Table 6.2. The exact numbers are gathered from running the script analyze_linker_map.py to analyze the linker map produced by the compiler. The raw outputs from running this script can be found in Section B.2.

Table 6.2: Comparison of memory usage for the present UcBoard software, the new one with routine configuration A and routine configuration B.

Region	Section	Size present	Size A	Size B
FLASH (512 KiB)	Program	62.0 KiB	77.5 KiB	77.7 KiB
	Constants	6.6 KiB	7.6 KiB	7.6 KiB
	Total used	68.6 KiB	85.0 KiB	85.3 KiB
	Unused	443.4 KiB	427.0 KiB	426.7 KiB
RAM (64 KiB)	Static variables	22.9 KiB	24.5 KiB	24.6 KiB
	Idle task stack	0 B	512 B	512 B
	Timer task stack	0 B	4.0 KiB	4.0 KiB
	Motor controller task stack	0 B	1.0 KiB	1.0 KiB
	Comm RX task stack	0 B	0 B	4.0 KiB
	Comm TX task stack	0 B	0 B	1.0 KiB
	ISR stack	4.0 KiB	4.0 KiB	4.0 KiB
	Total used	26.9 KiB	34.0 KiB	39.1 KiB
	Unused	37.1 KiB	30.0 KiB	24.9 KiB
CCRAM (16 KiB)	Total used	0 B	0 B	0 B
	Unused	16.0 KiB	16.0 KiB	16.0 KiB

6.3 Response times

By using the automation script `drv_response_time` to measure the response time R of a brake command sent a delay d after powering on LED A for 50 randomly selected values of d between 100 ms and 1000 ms, we get the response time measurements R_{m} shown in Figure 6.1 for the present non-RTOS UcBoard software, Figure 6.2 for routine configuration A and Figure 6.3 for routine configuration B. The reaction time deadline, $r_{\mathrm{D}} = 40$ ms, which we derived in Section 3.4.1 is illustrated with a dashed line. It should be noted that R is only for routine configuration A an exact upper bound for r, as described in Section 5.1.3. In all cases, the software is modified to include the artificial heavy operation described in Section 5.1.1, which is what makes the software get overloaded after LED A is powered on. The minimum, maximum and average response times from the measurements are shown in Table 6.3. The values of d are selected randomly and are not the same in each of the three cases. However, the distribution of values for d can have quite some impact on the results, so for the record, the percentage of the selected values d that are lower than 500 ms are also included.

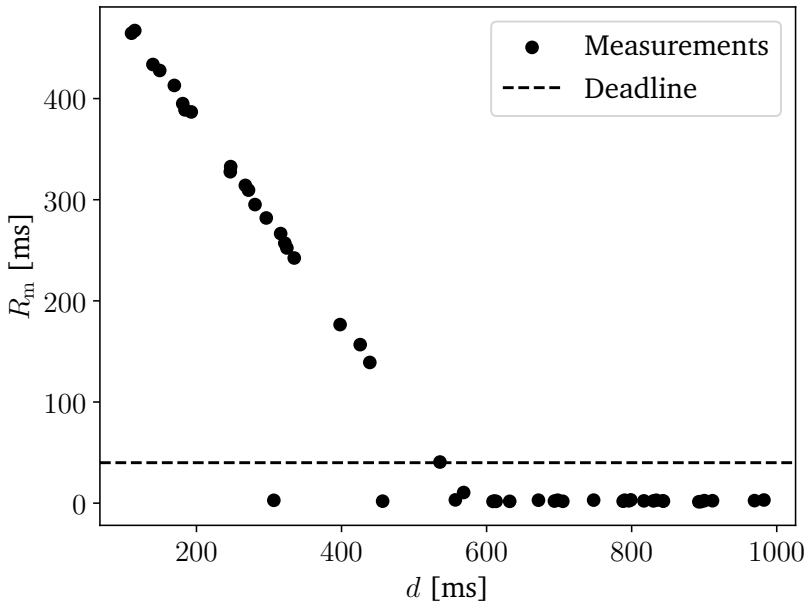


Figure 6.1: Response time for brake command when system is overloaded for present UcBoard software (modified to be overloadable).

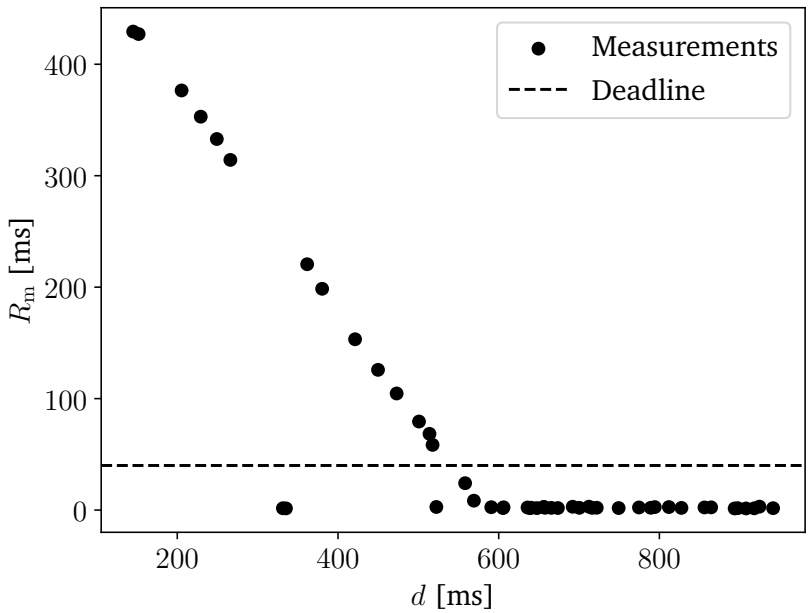


Figure 6.2: Response time for brake command when system is overloaded for routine configuration A.

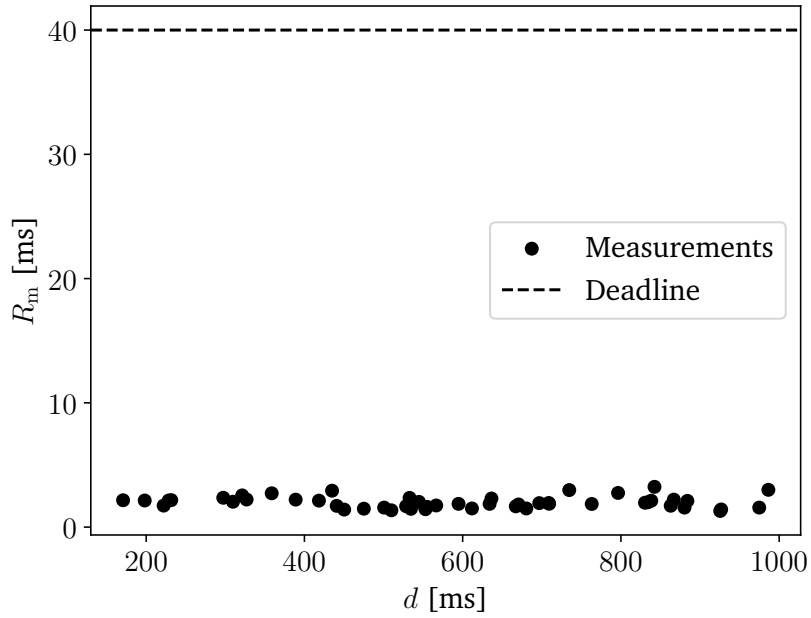


Figure 6.3: Response time for brake command when system is overloaded for routine configuration B.

Metric	Present	A	B
Percentage $d < 500\,\mathrm{ms}$	46 %	26 %	32 %
Minimum R_{m}	1.34 ms	1.45 ms	1.30 ms
Maximum R_{m}	467.28 ms	429.31 ms	3.24 ms
Average R_{m}	136.86 ms	66.96 ms	1.98 ms

Table 6.3: Minimum, maximum and average response times for present UcBoard software, routine configuration A and routine configuration B.

6.4 Speed controller

The resulting feed forward reference values found by testing are shown in Figure 6.4. A pure feed-forward speed controller using these reference values, which is the implemented speed control with $K_\mathrm{P} = 0\,(\mathrm{m/s})^{-1}$ and $K_\mathrm{I} = 0\,\mathrm{m}^{-1}$, has a step response as shown in Figure 6.5. The step response is shown for 3 different target speeds; $0.5\,\mathrm{m/s}$, $1.0\,\mathrm{m/s}$ and $1.5\,\mathrm{m/s}$.

As mentioned in Section 5.2.2, we tested a total of 56 combinations of K_P and K_I, which were all $K_\mathrm{P} \in \{0, 50, 100, 200, 500, 1000, 2000\}\ (\mathrm{m/s})^{-1}$ and all $K_\mathrm{I} \in \{0, 5, 10, 20, 50, 100, 200, 500\}\ \mathrm{m}^{-1}$. Only a selected subset is shown here. The most optimal combination is found to be $K_\mathrm{P} = 500\,(\mathrm{m/s})^{-1}$ and $K_\mathrm{I} = 10\,\mathrm{m}^{-1}$, whose step response is shown in Figure 6.6. We can quickly determine that using these values gives the speed controller a more desireable behavior than in with feed-forward only. Firstly, the equilibrium speed gets more accurate, and secondly, the acceleration time is shorter. By increasing K_P and K_I further, we get some undesirable properties. Increasing K_P to $2000\,(\mathrm{m/s})^{-1}$ gives us the step response shown in Figure 6.7, where the speed oscillates. Increasing K_I to $500\,\mathrm{m}^{-1}$ (while setting K_P back to $500\,(\mathrm{m/s})^{-1}$) gives us the step response shown in Figure 6.8, where the speed becomes too big for a while directly after the initial acceleration.

Average acceleration times $\overline{T_\mathrm{a}}$ for the mentioned 4 sets of parameters are listed in Table 6.4.

K_P	K_I	$\overline{T_\mathrm{a}}$
$0\,(\mathrm{m/s})^{-1}$	$0\,\mathrm{m}^{-1}$	$875\,\mathrm{ms}$
$500\,(\mathrm{m/s})^{-1}$	$10\,\mathrm{m}^{-1}$	$478\,\mathrm{ms}$
$2000\,(\mathrm{m/s})^{-1}$	$10\,\mathrm{m}^{-1}$	$365\,\mathrm{ms}$
$500\,(\mathrm{m/s})^{-1}$	$500\,\mathrm{m}^{-1}$	$425\,\mathrm{ms}$

Table 6.4: Initial acceleration time for speed controller. T_S is in all cases $50\,\mathrm{ms}$.

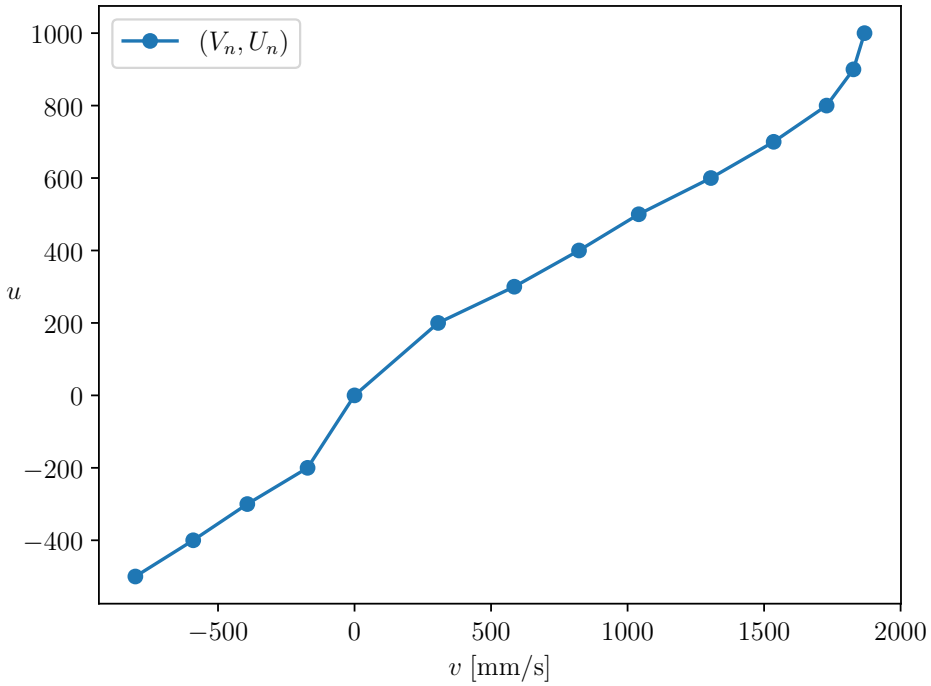


Figure 6.4: Speed controller feed forward control.

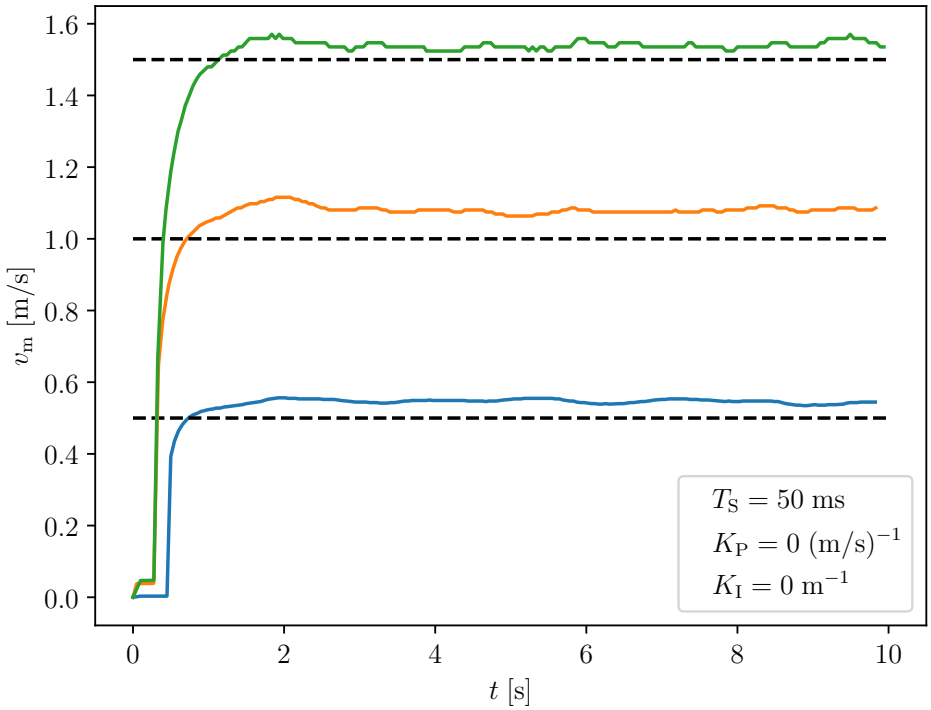


Figure 6.5: Step response of feed-forward only speed controller for 3 different target speeds.

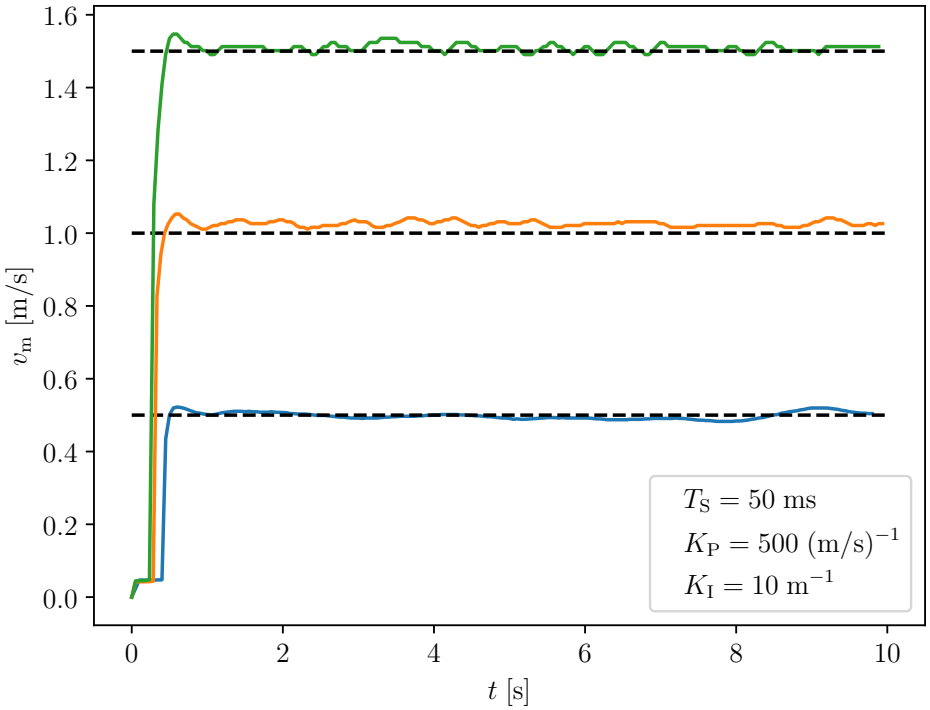


Figure 6.6: Step response of FF+P+I speed controller for 3 different target speeds.

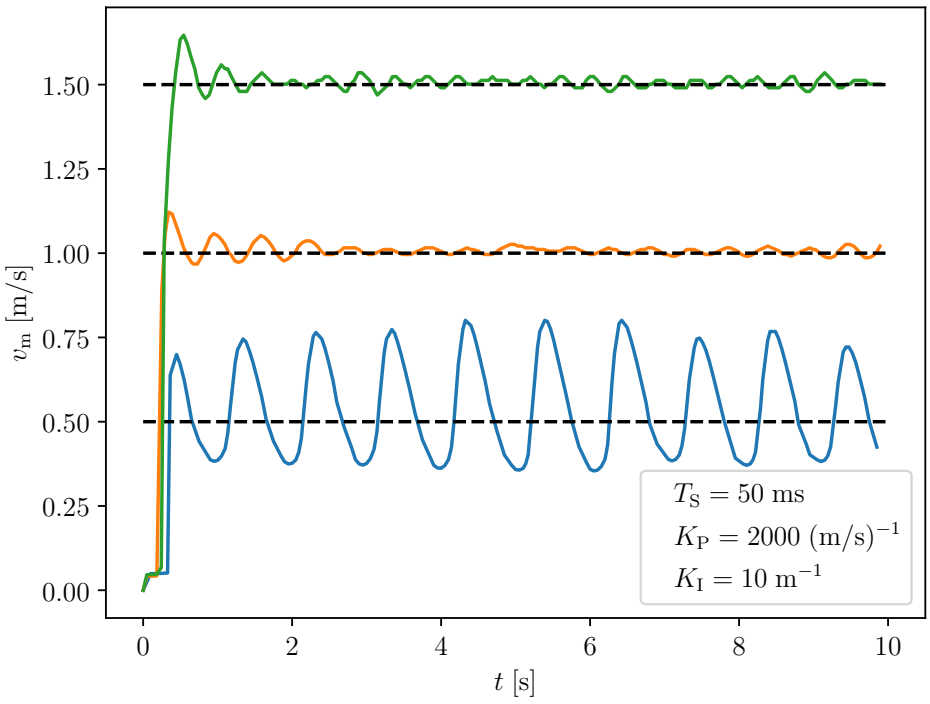


Figure 6.7: Step response of FF+P+I speed controller with a more aggressive value for K_P.

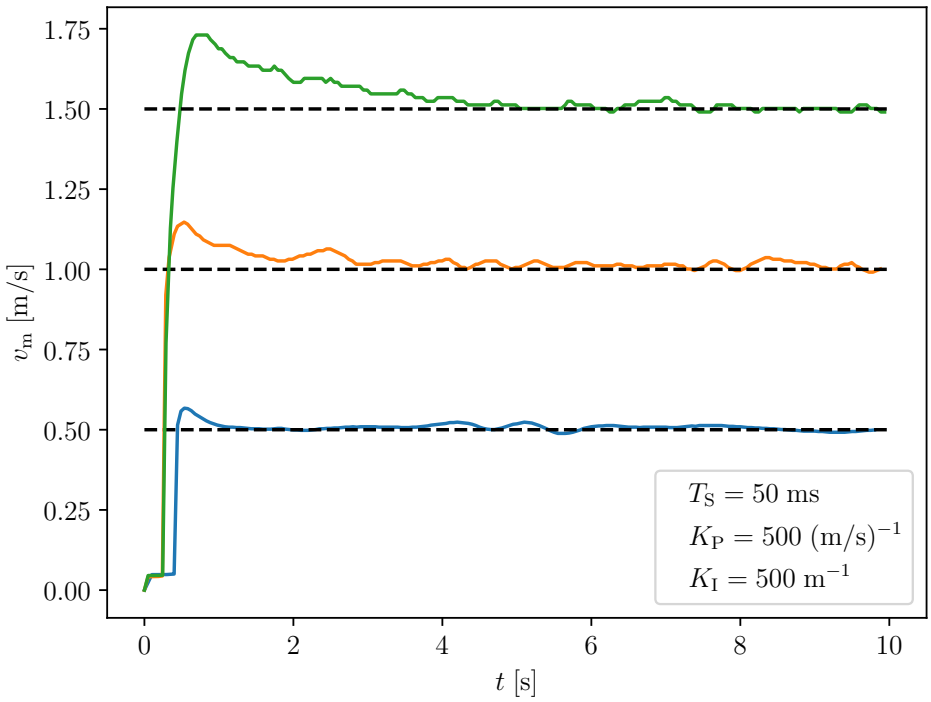


Figure 6.8: Step response of FF+P+I speed controller with a more aggressive value for K_I.

7 Discussion

In this chapter, we will discuss the results presented in Chapter 6. This includes investigating why the numbers show what they show, what they mean, and what we can learn from them. Firstly, the memory usage will be discussed in Section 7.1, then the response time measurements will be discussed in Section 7.2, and then results for the speed controller are discussed in Section 7.3. In the end, we will attempt to compare the work as a whole to similar software systems from related academic works.

7.1 Memory usage

As we saw in Section 6.2, the new UcBoard software with FreeRTOS uses considerably more memory than the present one. Table 6.2 shows that the main contribution to increased main memory usage comes from extra task stacks. The memory usage of static variables in routine configuration A is 1.6 KiB more than in the present software, and in configuration B, 1.7 KiB more than in the present software. Additionally, routine configuration A adds 5.5 KiB of memory usage by task stacks, and configuration B adds 10.5 KiB. This leads to a total of 26 % and 45 % increase in memory usage, respectively for configuration A and B compared to the present software. However, the UcBoard has a total of 64 KiB of RAM, and there is still 30.0 KiB and 24.9 KiB unused, respectively for the two configurations. Table 6.2 also shows that the UcBoard has 16 KiB of CCRAM that is unused in all 3 alternatives. As we saw in Section 2.6.2, the CCRAM of the STM32F303VE is intended for routine boosting. This means that moving the stack of high priority tasks such as communication RX and/or TX to the CCRAM can be beneficial as it could make these tasks execute faster. How considerable this performance boost actually is must be deeper investigated to know for sure.

As a conclusion, with routine configuration B, we have increased the memory usage from 26.9 KiB to 39.1 KiB. However, in total, 40.9 KiB of RAM and CCRAM is still unused, which means there is still room for developing many new features without re-engineering the entire architecture. If the extra memory usage caused by utilizing FreeRTOS provides considerable real-time benefits, this might very well be a good fit for the UcBoard software of the PS AF.

7.2 DRV response time

Our measurements from Section 6.3 show that preemptive scheduling does give the system more robust response times. In configuration A, where the communication routines don't run as dedicated tasks, they should not have a chance to run before LED A is fully turned on after about 500 ms, which we can see in Figure 6.2 is actually the case. This is also the case for the present non-RTOS UcBoard software, where they are systick callbacks, which we can see in Figure 6.1. In routine configuration B

however, where the communication routines run in dedicated tasks with higher priority than the timer task, we can see that the response time is much more consistent in Figure 6.3.

In Table 6.3, we see that the average response time is more than twice as long for the present UcBoard software than for routine configuration A. But from looking at the plots, they seem to perform about equal for a given value of d. But by looking closer, we see that the distribution of randomly selected values d is very different for the two cases; for the present software, $46\,\%$ of the values for d is lower than $500\,\text{ms}$, while the same number is only $26\,\%$ for routine configuration A. As $46\,\%$ is almost the double of $26\,\%$, this is most likely the sole explanation for this significant difference in average response time.

As we saw in Section 2.6.1, the transfer speed of the serial connection is $92.16\,\text{characters/ms}$. The full command being sent is 12 characters, which should take approximately $0.1\,\text{ms}$ to transfer. The full response is about the same length. The callback functions for both routines are called with a period of $1\,\text{ms}$, which is how often communication RX checks for new commands and how often communication TX checks for new responses to send. The internal processing should take less than $1\,\text{ms}$ in both callbacks. This means it should take $0.1\,\text{ms}$ to transfer the command to the UcBoard, maximum $1\,\text{ms}$ for communication RX to start processing it, then processing should take maximum $1\,\text{ms}$, and then it might take up to $1\,\text{ms}$ before communication TX runs again and starts processing the response. In the end, transmitting the response back to the computer takes $0.1\,\text{ms}$, which in total accounts for up to $4.2\,\text{ms}$. As we see in Table 6.3, for routine configuration B, the highest measured R_m is $3.24\,\text{ms}$, which is as expected below that.

As we just saw, there is quite a bit of waiting involved in the command-response chain, of a theoretical maximum R of $4.2\,\text{ms}$, as much as $2\,\text{ms}$ is caused by waiting time between each time the communication tasks run. This happens because even though the communication routines run as tasks in configuration B, they are converted with the universal method described in Section 3.1, and do not utilize FreeRTOS blocking API calls to operate event-driven, rather than polling-driven. By doing a more tailored conversion, utilizing event-driven programming patterns enabled by blocking FreeRTOS API calls, the waiting time can be reduced to near $0\,\text{ms}$, including only the scheduling overhead required by FreeRTOS to switch tasks, giving a theoretical maximum R of only about $2.2\,\text{ms}$.

As we noted in Section 3.4, the response time R is just an estimator for the more important reaction time r. And as we explained in Section 5.1.3, for routine configuration A, R is guaranteed to be an upper bound for r, partly due to the motor controller using an event-driven approach to fetch drive requests from the drive request queue. In configuration B, the motor controller uses the same event-driven approach to fetch drive requests from the drive request queue. This means that in both configurations, there is no polling delay needed from a drive command is received in communication RX until the drive request is received and gets processed in the motor controller. But in both configurations, communication TX uses a polling-driven approach to retrieve command responses, with a $1\,\text{ms}$ polling period. Due to this, we can assume the actual reaction times r behind the response time measurements to be up to $1\,\text{ms}$ lower than their respective values R, for routine configuration A and B. In particular, we should assume the maximum r to be about $1\,\text{ms}$ lower than the maximum R, the minimum r to be about the same as the minimum R, and the average r to be about $0.5\,\text{ms}$ lower than the average R.

7.3 Speed controller

As we saw in Section 6.4, the feed-forward only speed controller makes a good basis for the speed controller, but lacks some accuracy and responsiveness. We saw that those aspects can be improved significantly by adding a proportional term with $K_\mathrm{P} = 500\,(\mathrm{m/s})^{-1}$ and an integral term with $K_\mathrm{I} = 10\,\mathrm{m}^{-1}$. These parameter values give the speed controller quite a pleasant impulse response as shown in Figure 6.6. The average acceleration time is reduced to about the half, from 875 ms for feed-forward only down to 478 ms. An even larger $K_\mathrm{P} = 2000\,(\mathrm{m/s})^{-1}$ would further reduce the average acceleration time down to only 365 ms, but it also leads to unwanted oscillations, especially significant for lower speeds, as shown in Figure 6.7.

One could argue that it is desirable to further increase K_I to improve accuracy. We see in Figure 6.8 that setting $K_\mathrm{I} = 500\,\mathrm{m}^{-1}$ leads to the speed becoming significantly larger than v_r for a while after the initial acceleration. This could be seen as unintended behavior, since the speed is not correct. But what actually happens is that the integral term becomes just big enough to compensate for the travel distance lost during the initial acceleration, since the integral of speed is distance. If the driving algorithm on the OBC uses distance measurements and timing to drive to a given position, one can imagine that a speed controller that automatically compensates for lost travel distance during initial acceleration in many cases leads to a more precise ADS overall. While this can be a good property for precision-oriented tasks like e.g. parking, it is usually not desirable when driving longer distances on a road. Roads usually tend to have speed limits to enforce safety, as a high momentary speed is related to long brake distance, and thus high risk of causing accidents. A speed controller with high K_I will, if needed, defy speed limits for a period of time after the initial acceleration to recover lost travel distance, putting the ADS and the environment at unnecessary risk. For this reason, a high K_I is definitely not desirable in all cases. One way of mitigating this could be to conditionally set K_I based on the requested speed v_r, so that e.g. if $v_\mathrm{r} > 0.8\,\mathrm{m/s}$, $K_\mathrm{I} = 500\,\mathrm{m}^{-1}$ and if not, $K_\mathrm{I} = 10\,\mathrm{m}^{-1}$. This will on the other side make it more complex for the OBC to know what behavior to expect. The OBC could also change K_I during operation to facilitate different driving scenarios, this would presumably be the safest option.

7.4 Comparisons to related work

As we have seen, in our FreeRTOS adoption, we use 5 tasks in total. In the comparable microcontroller software for the Carolo-Cup competition car of 2018 from the Warsaw University of Technology, 8 tasks are used. These 8 tasks, listed with decreasing priority, are: RC Receiver, Task Governor, Steering, IMU, Distance Sensors, Bluetooth, LED and Battery Telemetry [44]. The fact that no timer task is listed means that this software allegedly does not use any FreeRTOS timers, all routines are implemented by custom tasks. An interesting point here is that the Warsaw car has a dedicated steering task, but no dedicated motor control task. On the other hand, our implementation uses a dedicated motor control task, but implements steering as a timer callback. This shows that our software architecture is not an obvious choice of software architecture for the microcontroller of an autonomous model car; vastly different architectures can also lead to a functional solution. The most radical architectural difference between the PS AF car and the Warsaw car is, as mentioned in Section 2.7, that the microcontroller holds the commander role, while the Linux machine acts as a camera and image processing worker. This could be an important reason for why there are no dedicated tasks for communication in the

microcontroller software of the Warsaw car. When the microcontroller has the commander role, it can typically be structured quite differently, as its only mission is not to wait for commands and respond to them as quickly as possible, instead, the microcontroller gives the commands and receives responses when this is convenient. For this type of operation, implementing communication logic in functions to be called occasionally by a governor task can be more appropriate.

In [45], no further details on routine implementation, memory usage or response time for the 6th Carolo-Cup car from the University of Magdeburg are provided. But details are provided on how FreeRTOS is bundled with the application code. For the Magdeburg car, FreeRTOS is bundled through the CMSIS RTOS API by using CubeMX. In Section 4.1, we found that using the CMSIS API would not be a proper solution for the PS AF UcBoard. [45] shows that using the CMSIS API also can be a successful approach for a Carolo-Cup car.

8 Conclusion

In this chapter, we will conclude what we have accomplished in this work and give an outlook into what can be researched further based on our work in the future.

8.1 Conclusion

Throughout this work, the free and open-source Real-Time Operating System FreeRTOS has been integrated into the software for the UcBoard for the Project Seminar Autonomes Fahren at the Technical University of Darmstadt. Two different routine configurations, namely A and B, have been implemented and tested, leading to an increase in memory usage at $26\,\%$ and $45\,\%$ respectively for the two configurations. It has been shown that routine configuration B can lead to much more stable response times to commands than both configuration A and the present software. In particular the highest measured response time was reduced from $467.28\,\text{ms}$ with the present software to $3.24\,\text{ms}$ with routine configuration B in a constructed overload scenario. Even though it increases the memory usage by $45\,\%$, about $50\,\%$ of the UcBoard's total memory is still unused and we can conclude that configuration B is a definitive improvement compared to the present software. Blocking FreeRTOS API functions have been utilized to implement an event-driven motor controller routine with a low-level speed controller. This ensures that the UcBoard also on car model 1 and 2 can control speed independently of the OBC, which makes these car models more useful as test and development platforms in the future when car model 3 is finished. Support within the UcBoard software for the BLDC motor controller to be used in car model 3 is also contributed by this work. As a last piece of the work, the two different variants of the source code targeting the currently two different car models have been refactored and merged by using compile-time configuration options. Relevant aspects have been discussed and compared to similar software systems from related academic works.

8.2 Outlook

It was originally a part of the assignment to implement a yaw rate controller for the PS AF model car in the UcBoard software, but the time did not allow for this in this work. It is however still a desire to have such functionality embedded in the UcBoard software, which makes for a suitable piece of a future work. Since a yaw rate controller, just like a speed controller, is a servomechanism, a combined feed-forward and PI- controller could also here be a good fit. Considering that the UcBoard software now has an RTOS and a feed-forward and PI- speed controller, the implementation of a yaw rate controller can benefit from the API and real-time capabilities of the RTOS and re-use the structure of

the speed controller. This could allow for a straightforward yaw rate controller implementation that suits well into the source code for the UcBoard software.

As pointed out in Section 4.2.3, the selection of stack sizes for FreeRTOS tasks would be more beneficial to automate by using a sophisticated tool like e.g. StackAnalyzer to calculate stack size upper bounds, instead of the currently inexact semi-automatic approach with the script analyze_stack_usage.py. This would first and foremost improve the memory utilization of the system, since the allocated stack size for each FreeRTOS task can be lowered considerably, with increased certainty that the selected stack size actually is large enough. In addition, the stack size selection process would be fully automated, leaving more time for the developer, designer or researcher to conquer other more challenging tasks.

For routine configuration B, the communication routines have been converted to tasks, but with the universal method described in Section 3.1.2. In Section 7.2, we explained how a tailored conversion utilizing FreeRTOS blocking API calls can significantly improve response times for commands. It would also allow the communication tasks to use less processing time on polling, as explained in Section 3.1.2 and free up processing time for the UcBoard as a whole.

Bibliography

[1] S. Singh, "Critical reasons for crashes investigated in the national motor vehicle crash causation survey," tech. rep., 2015.

[2] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, "A survey of autonomous driving: Common practices and emerging technologies," *IEEE access*, vol. 8, pp. 58443–58469, 2020.

[3] T. J. Crayton and B. M. Meier, "Autonomous vehicles: Developing a public health research agenda to frame the future of transportation policy," *Journal of Transport & Health*, vol. 6, pp. 245–252, 2017.

[4] SAE, "Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles," 2021, Accessed: 2023-08-09. Available: `https://www.sae.org/standards/content/j3016_202104/`.

[5] W. Wang and L. Paulino, "Instill autonomous driving technology into undergraduates via project-based learning," in *2021 IEEE Integrated STEM Education Conference (ISEC)*, pp. 284–287, IEEE, 2021.

[6] L. Chen, Y. Li, C. Huang, B. Li, Y. Xing, D. Tian, L. Li, Z. Hu, X. Na, Z. Li, *et al.*, "Milestones in autonomous driving and intelligent vehicles: Survey of surveys," *IEEE Transactions on Intelligent Vehicles*, vol. 8, no. 2, pp. 1046–1056, 2022.

[7] V. Shreyas, S. N. Bharadwaj, S. Srinidhi, K. Ankith, and A. Rajendra, "Self-driving cars: An overview of various autonomous driving systems," *Advances in Data and Information Sciences: Proceedings of ICDIS 2019*, pp. 361–371, 2020.

[8] E. Ackerman, "Toyota's gill pratt on self-driving cars and the reality of full autonomy - ieee spectrum," 2017, Accessed: 2023-08-09. Available: `https://spectrum.ieee.org/toyota-gill-pratt-on-the-reality-of-full-autonomy`.

[9] J. Liu and J. Liu, "Intelligent and connected vehicles: Current situation, future directions, and challenges," *IEEE Communications Standards Magazine*, vol. 2, no. 3, pp. 59–65, 2018.

[10] E. Huhtamo *et al.*, "The self-driving car: A media machine for posthumans?," *Artnodes*, no. 26, pp. 1–14, 2020.

[11] C. Berger, "From a competition for self-driving miniature cars to a standardized experimental platform: concept, models, architecture, and evaluation," *arXiv preprint arXiv:1406.7768*, 2014.

[12] S. Zug, C. Steup, J.-B. Scholle, C. Berger, O. Landsiedel, F. Schuldt, J. Rieken, R. Matthaei, and T. Form, "Technical evaluation of the carolo-cup 2014-a competition for self-driving miniature cars," in *2014 IEEE International Symposium on Robotic and Sensors Environments (ROSE) Proceedings*, pp. 100–105, IEEE, 2014.

[13] U. Nations, "The 17 goals | sustainable development," 2015. Available: https://sdgs.un.org/goals, Accessed: 2023-08-23.

[14] P. A. Laplante *et al.*, *Real-time systems design and analysis*. Wiley New York, 2004.

[15] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[16] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, "Real time scheduling theory: A historical perspective," *Real-time systems*, vol. 28, pp. 101–155, 2004.

[17] A. Silberschatz, P. B. Galvin, and G. Gagne, "Process scheduling, chapter 5.3.4 round robin scheduling," *Operating System Concepts*, vol. 8, p. 194, 2010.

[18] A. Holt and C.-Y. Huang, *Embedded operating systems, chapter 1: Introduction*. Springer, 2014.

[19] FreeRTOS, "Freertos faq - memory usage, boot times and context switch times," Accessed: 2023-06-28. Available: https://www.freertos.org/FAQMem.html#ROMUse.

[20] K. Andersson and R. Andersson, "A comparison between freertos and rtlinux in embedded real-time systems," *Linköping University*, 2005.

[21] FreeRTOS, "Freertos scheduling," Accessed: 2023-06-29. Available: https://www.freertos.org/single-core-amp-smp-rtos-scheduling.html.

[22] R. Barry, *Mastering the FreeRTOSTM Real Time Kernel: A Hands-On Tutorial Guide*. Real Time Engineers Limited, 2016. Available: https://freertos.org/Documentation/RTOS_book.html.

[23] L. E. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable interrupt management for real time kernels over conventional pc hardware," in *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pp. 14–23, IEEE, 2006.

[24] FreeRTOS, "Freertos api reference," 2023. Available: https://www.freertos.org/a00106.html, Accessed: 2023-08-24.

[25] iMicro System, "Stack analyzer," 2023. Available: https://imicrosystem.com/index.php/stack-analyzer/, Accessed: 2023-08-30.

[26] M. Melkonian, "Get by without an rtos," 2000. Available: https://www.embedded.com/get-by-without-an-rtos/, Accessed: 2023-25-08.

[27] J. C. Doyle, B. A. Francis, and A. R. Tannenbaum, *Feedback control theory*. Courier Corporation, 2013.

[28] J. G. Balchen, T. Andresen, and B. A. Foss, *Reguleringsteknikk*. INSTITUTT FOR TEKNISK KYBERNETIKK, NTNU, TRONDHEIM, 2016.

[29] A. Lichtman and P. Fuchs, "Theory of pi controller and introduction to implementation for dc motor controls," in *2017 Communication and Information Technologies (KIT)*, pp. 1–5, IEEE, 2017.

[30] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of software maintenance and evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.

[31] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE'04.*, pp. 83–92, IEEE, 2004.

[32] T. Mens, S. Demeyer, and R. Koschke, "Identifying and removing software clones," *Software Evolution*, pp. 15–36, 2008.

[33] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, pp. 645–692, 2008.

[34] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software configuration engineering in practice interviews, survey, and systematic literature review," *IEEE Transactions on Software Engineering*, vol. 46, no. 6, pp. 646–673, 2018.

[35] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1055–1090, 2015.

[36] Mitac, "Pd10bi – thin mini-itx form factor intel bay trial processor," 2023. Available: https://download.mitacmct.com/Files/datasheets/motherboards/PD10BI.pdf, Accessed: 2023-09-05.

[37] Intel, "Intel celeron processor j1900 2m cache up to 2.42 ghz product specifications," 2023. Available: https://ark.intel.com/content/www/us/en/ark/products/78867/intel-celeron-processor-j1900-2m-cache-up-to-2-42-ghz.html, Accessed: 2023-09-05.

[38] IBM, "Start, stop and mark bits – ibm documentation," 2023. Available: https://www.ibm.com/docs/en/aix/7.1?topic=parameters-start-stop-mark-bits, Accessed: 2023-08-25.

[39] E. Lenz, "Beschreibung fahrzeug und ucboard," 2021. Internal documentation for the PS AF at the TU-Darmstadt.

[40] ST, "Datasheet for stm32f303xd and stm32f303xe mcus," 2016. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f303.html.

[41] E. Lenz, "Bldc," 2023. Internal documentation of the BLDC motor controller for the PS AF at the TU-Darmstadt.

[42] C. Berger, J. Hansson, *et al.*, "Cots-architecture with a real-time os for a self-driving miniature vehicle," in *SAFECOMP 2013-Workshop ASCoMS (Architecting Safety in Collaborative Mobile Systems) of the 32nd International Conference on Computer Safety, Reliability and Security*, 2013.

[43] G. D. Sirio, "Chibios homepage," 2023. Available: https://www.chibios.org/dokuwiki/doku.php, Accessed: 2023-08-22.

[44] M. Perciński and M. Marcinkiewicz, "Architecture of the system of 1: 10 scale autonomous car—requirements-based design and implementation," in *2018 International Interdisciplinary PhD Workshop (IIPhDW)*, pp. 263–268, IEEE, 2018.

[45] T. Wiesner and M. Grau, "Design und implementierung eines backbones für ein autonomes modellfahrzeug," 2020.

[46] ST, "X-cube-freertos - freertos software expansion for stm32cube - stmicroelectronics," 2023. Available: https://www.st.com/en/embedded-software/x-cube-freertos.html, Accessed: 2023-09-05.

[47] Arm, “Cmsis-rtos2 documentation,” 2023. Available: https://arm-software.github.io/CMSIS_5/RTOS2/html/index.html, Accessed: 2023-09-05.

[48] FreeRTOS, “Freertos config cortex m4f stm32f407zg-sk,” Accessed: 2023-08-03. Available: https://github.com/FreeRTOS/FreeRTOS/blob/main/FreeRTOS/Demo/CORTEX_M4F_STM32F407ZG-SK/FreeRTOSConfig.h#L55.

A Source code

A.1 UcBoard software repository outline

This section provides an outline of the file structure for the the source code for the UcBoard software. The outline describes the state of the repository at the end of the contributed work.

- ci
  - analysis-cppcheck.sh
  - format.py
  - install_arm_toolchain.sh
  - style.clang-format
- include
  - Empty directory
- lib
  - Empty directory
- src
  - common
    - ARingbuffer.c
    - ARingbuffer.h
    - common_fcts.c
    - common_fcts.h
    - crc.c
    - crc.h
    - debug.h
    - encoding.c
    - encoding.h
    - ringbuffer.c
    - ringbuffer.h

    - stdtypes.h
    - strfcts.c
    - strfcts.h
- config
    - config.h – source code listed in Section A.3.1.
    - config_ucboard1.h – source code listed in Section A.3.2.
    - config_ucboard1_bldc.h – source code listed in Section A.3.3.
    - config_ucboard2.h – source code listed in Section A.3.4.
- devices
    - bldc.c
    - bldc.h
    - carbasicfcts.c
    - carbasicfcts.h
    - carid.c
    - carui.c
    - carui.h
    - comm.c
    - comm.h
    - comm_cmdtable.c
    - comm_cmdtable.h
    - comm_public.h
    - daq.c
    - daq.h
    - display.c
    - display.h
    - eeprom.c
    - eeprom.h
    - encoder.c
    - encoder.h
    - errcodes.h
    - globalvar.c
    - globalvar.h

    - hal503.c
    - hal503.h
    - imu.c
    - imu.h
    - imu_icm20648.c
    - imu_icm20648.h
    - imu_icm20648_privatedefs.h
    - imu_mpu9250.c
    - imu_mpu9250.h
    - imu_mpu9250_privatedefs.h
    - led.c
    - led.h
    - mag_AK8963_privatedefs.h
    - pwm_motor.c
    - pwm_motor.h
    - receiver.c
    - receiver.h
    - sid.c
    - ucboard_hwdefs.h
    - ucboard_hwfcts.h
    - us.c
    - us.h
    - us_srf08.c
    - us_srf08.h
    - version.c
    - version.h
  - Drivers
    - CMSIS
      - Directory contents left out.
    - STM32F3xx_HAL_Driver
      - Directory contents left out.
  - FreeRTOS-Kernel

        - Directory contents left out.
    - stmcommon
        - atomicsection.h
        - exti_mgr.c
        - exti_mgr.h
        - i2cmgr.c
        - i2cmgr.h
        - i2cmgr_privatedefs.h
        - spimgr.c
        - spimgr.h
        - stm32_llm.c
        - stm32_llm.h
        - stopwatch.c
        - stopwatch.h
        - tim_irq_callbacks.c
    - FreeRTOSConfig.h
    - main.c
    - main.h
    - mxconstants.h
    - stm32f3xx_hal_conf.h
    - stm32f3xx_hal_msp.c
    - stm32f3xx_it.c
    - stm32f3xx_it.h
    - systick.c
    - systick.h
- svd
    - STM32F303.svd
- tests
    - Empty directory
- tools
    - analyze_linker_map.py – source code listed in Section A.3.5.
    - analyze_stack_usage.py – source code listed in Section A.3.6.

  - stlink.py
  - stlink_readme.txt
  - stlinkflash_template.bat
- CMakeLists.txt
- CMakePresets.json
- cppcheck_suppressions
- Doxyfile
- pses_ucboard.ioc
- README.adoc
- STM32F303VETx_FLASH.ld

A.2 Pyserial-UcBoard repository outline

This section provides an outline of the files in the Pyserial-UcBoard repository. This repository holds the code being run on an attached PC to automate the experiments conducted in this work.

- automate.py – source code listed in Section A.4.1.
- dummy_ucboard.py – source code listed in Section A.4.2.
- generate_commands.py – source code listed in Section A.4.3
- history.pthistory
- input_test_cruise_controller.txt
- input_test_managed.txt
- pyserial_ucboard.py – source code listed in Section A.4.4
- requirements.txt

A.3 UcBoard software source code extract

In this section a selected extract of the source code for the UcBoard software is provided. The code base in full can not be published due to copyright considerations, but a relevant extract of the source code contributed by this work is provided here.

A.3.1 config.h

```
/*
 * config.h
 *
 * Defines the actual config that is used for compilation.
 */

#ifndef CONFIG_H
#define CONFIG_H

#include "config_ucboard1.h"

#endif /* CONFIG_H */
```

A.3.2 config_ucboard1.h

```
/*
 * config_ucboard1.h
 *
 * Defines config to use when compiling for ucboard1.
 */

#ifndef CONFIG_UCBOARD1_H
#define CONFIG_UCBOARD1_H

// Number of LEDs on the car
#define CONFIG_LED_COUNT 6

// Name of LEDs, used for identification in serial protocol
#define CONFIG_LED_NAMES \
    { "A", "B", "C", "D", "E", "F" }

// A struct containing information about which GPIO ports and pins the LEDs use
#define CONFIG_LED_PORTS_AND_PINS
        \
    {
        \
        {GPIOC, GPIO_PIN_4}, {GPIOC, GPIO_PIN_5}, {GPIOE, GPIO_PIN_0}, {GPIOE,
    GPIO_PIN_1}, \
            {GPIOE, GPIO_PIN_3}, {
        \
            GPIOE, GPIO_PIN_4
        \
        }
        \
    }

// Enables support for encoder (Inkrementalgeber) for speed measurment
#define CONFIG_ENCODER_ENABLE 0

// Enables support for hall sensor for speed measurement
#define CONFIG_HALL_SENSOR_ENABLE 1

// Defines how far the car moves per 8 hall sensor pulses (unit: um)
#define CONFIG_HALL_SENSOR_DELTA_S 204204

```

```
35 // Defines which USART of the STM32F303 to use
36 // Possible values are: 2 (USART2), 3 (USART3), 4 (UART4)
37 #define CONFIG_USART 2  // USB
38 // #define CONFIG_USART 4  // Bluetooth
39
40 // Defines the baud rate used for the USART
41 #define CONFIG_USART_BAUD_RATE 921600  // USB
42 // #define CONFIG_USART_BAUD_RATE 57600  // Bluetooth
43
44 // Type of inertial measurement unit
45 #define CONFIG_IMU_TYPE_MPU9250  1
46 #define CONFIG_IMU_TYPE_ICM20648 0
47
48 // Enables support for receiver for remote control
49 #define CONFIG_RECEIVER_ENABLE 0
50
51 // Type of ultrasonic sensors
52 #define CONFIG_US_TYPE_SRF08 1
53 #define CONFIG_US_TYPE_SRF10 0
54
55 // Number of ultrasonic sensors on the car
56 #define CONFIG_US_COUNT 4
57
58 // I2C bus addresses for ultrasonic sensors
59 #define CONFIG_US_ADDRESSES \
60     { 0xE0, 0xE2, 0xE4, 0xE6 }
61
62 // DAQ channel names for ultrasonic sensors
63 #define CONFIG_US_DAQ_CHANNEL_NAMES \
64     { "USL", "USF", "USR", "USB" }
65
66 // DAQ channel descriptions for ultrasonic sensors
67 #define CONFIG_US_DAQ_CHANNEL_DESCRIPTIONS
            \
68     {
            \
69         "ultrasonic left distance", "ultrasonic front distance", "ultrasonic right
    distance", \
70             "ultrasonic back distance"
            \
71     }
72
73 #define CONFIG_MOTOR_FULLFORWARDS            -500
74 #define CONFIG_MOTOR_NEUTRAL_FORWARDS_ENDED  -52
75 #define CONFIG_MOTOR_FULLBACKWARDS           250
76 #define CONFIG_MOTOR_NEUTRAL_BACKWARDS_ENDED 53
77 #define CONFIG_MOTOR_PWMVAL_DELTA_NEGATIVE   0
78
79 // Delay between each time the cruise controller adjusts the drvval (unit: ms)
80 #define CONFIG_CRUISE_CONTROL_T_S 50
81
82 // Estimated top speed for the configuration of car and motor (unit: mm/s)
83 // Used for calibrating the cruise controller.
84 #define CONFIG_CRUISE_CONTROL_REFERENCE_TOP_SPEED_FORWARDS  (5000)
85 #define CONFIG_CRUISE_CONTROL_REFERENCE_TOP_SPEED_BACKWARDS (2500)
86
87 // Reference {speed, drvval} pairs for the cruise controller's feed forward term (unit:
    mm/s)
```

```
// The list must have minimum 2 entries.
// All values must be integers.
// Entries must be ordered by increasing speed.
// Drvval values should always be given as positive numbers. Driving direction is interpreted
// automatically.
#define CONFIG_CRUISE_CONTROL_FF_REFERENCE \
    { \
        {-803, -500}, {-591, -400}, {-393, -300}, {-172, -200}, {0, 0}, {306, 200}, {585, 300}, \
        {822, 400}, {1041, 500}, {1305, 600}, {1535, 700}, {1729, 800}, {1827, 900}, { \
        1868, 1000 \
        } \
    }
#define CONFIG_CRUISE_CONTROL_FF_REFERENCE_LENGTH 14

// Multiplication factor for the cruise controller's proportional term (unit: drvval/(m/s))
#define CONFIG_CRUISE_CONTROL_K_P 1000

// Multiplication factor for the cruise controller's integral term (unit: drvval/(m))
#define CONFIG_CRUISE_CONTROL_K_I 100

// Type of motor(s)
#define CONFIG_MOTOR_TYPE_PWM  1
#define CONFIG_MOTOR_TYPE_BLDC 0

#endif /* CONFIG_UCBOARD1_H */
```

A.3.3 config_ucboard1_bldc.h

```
/*
 * config_ucboard1_bldc.h
 *
 * Defines config to use when compiling for ucboard1 with BLDC motor.
 */

#ifndef CONFIG_UCBOARD1_BLDC_H
#define CONFIG_UCBOARD1_BLDC_H

#include "config_ucboard1.h"

#undef CONFIG_MOTOR_TYPE_PWM
#define CONFIG_MOTOR_TYPE_PWM 0

#undef CONFIG_MOTOR_TYPE_BLDC
#define CONFIG_MOTOR_TYPE_BLDC 1

#endif /* CONFIG_UCBOARD1_BLDC_H */
```

A.3.4 config_ucboard2.h

```c
/*
 * config_ucboard1.h
 *
 * Defines config to use when compiling for ucboard2.
 */

#ifndef CONFIG_UCBOARD2_H
#define CONFIG_UCBOARD2_H

// Number of LEDs on the car
#define CONFIG_LED_COUNT 9

// Name of LEDs, used for identification in serial protocol
#define CONFIG_LED_NAMES \
    { "A", "B", "U1", "U2", "U3", "U4", "BLKR", "BLKL", "BRMS" }

// A struct containing information about which GPIO ports and pins the LEDs use
#define CONFIG_LED_PORTS_AND_PINS
                \
    {
                \
        {GPIOC, GPIO_PIN_4}, {GPIOC, GPIO_PIN_5}, {GPIOD, GPIO_PIN_10}, {GPIOD, GPIO_PIN_11},      \
            {GPIOD, GPIO_PIN_13}, {GPIOB, GPIO_PIN_5}, {GPIOD, GPIO_PIN_3}, {GPIOD, GPIO_PIN_4}, { \
            GPIOD, GPIO_PIN_5
                \
        }
                \
    }

// Enables support for encoder (Inkrementalgeber) for speed measurment
#define CONFIG_ENCODER_ENABLE 1

// Enables support for hall sensor for speed measurement
#define CONFIG_HALL_SENSOR_ENABLE 0

// Defines how far the car moves per 8 hall sensor pulses (unit: um)
#define CONFIG_HALL_SENSOR_DELTA_S 0

// Defines which USART of the STM32F303 to use
// Possible values are: 2 (USART2), 3 (USART3), 4 (UART4)
#define CONFIG_USART 2  // USB

// Defines the baud rate used for the USART
#define CONFIG_USART_BAUD_RATE 921600  // USB

// Type of inertial measurement unit
#define CONFIG_IMU_TYPE_MPU9250  0
#define CONFIG_IMU_TYPE_ICM20648 1

// Enables support for receiver for remote control
#define CONFIG_RECEIVER_ENABLE 1

// Type of ultrasonic sensors
#define CONFIG_US_TYPE_SRF08 0
#define CONFIG_US_TYPE_SRF10 1

```

```
// Number of ultrasonic sensors on the car
#define CONFIG_US_COUNT 8

// I2C bus addresses for ultrasonic sensors
#define CONFIG_US_ADDRESSES \
    { 0xE0, 0xE2, 0xE4, 0xE6, 0xE8, 0xEA, 0xEC, 0xEE }

// DAQ channel names for ultrasonic sensors
#define CONFIG_US_DAQ_CHANNEL_NAMES \
    { "US1", "US2", "US3", "US4", "US5", "US6", "US7", "US8" }

// DAQ channel descriptions for ultrasonic sensors
#define CONFIG_US_DAQ_CHANNEL_DESCRIPTIONS                                      \
    {                                                                           \
        "ultrasonic 1 distance", "ultrasonic 2 distance", "ultrasonic 3 distance",     \
            "ultrasonic 4 distance", "ultrasonic 5 distance", "ultrasonic 6 distance", \
            "ultrasonic 7 distance", "ultrasonic 8 distance"                    \
    }

#define CONFIG_MOTOR_FULLFORWARDS            -500
#define CONFIG_MOTOR_NEUTRAL_FORWARDS_ENDED  -38
#define CONFIG_MOTOR_FULLBACKWARDS           500
#define CONFIG_MOTOR_NEUTRAL_BACKWARDS_ENDED 74
#define CONFIG_MOTOR_PWMVAL_DELTA_NEGATIVE   1

// Delay between each time the cruise controller adjusts the drvval (unit: ms)
#define CONFIG_CRUISE_CONTROL_T_S 1000

// Estimated top speed for the configuration of car and motor (unit: mm/s)
// Used for calibrating the cruise controller.
#define CONFIG_CRUISE_CONTROL_REFERENCE_TOP_SPEED_FORWARDS  (5000)
#define CONFIG_CRUISE_CONTROL_REFERENCE_TOP_SPEED_BACKWARDS (2500)

#endif /* CONFIG_UCBOARD2_H */
```

A.3.5 analyze_linker_map.py

```
#!/usr/bin/python3

import argparse
import inspect
import os
import subprocess
import time
import re
from dataclasses import dataclass


def r(*path):
    """
    Takes a relative path from the directory of this python file and returns the
    absolute path.
    """
    return os.path.join(os.path.dirname(os.path.abspath(__file__)), *path)


def run_in_dir(directory, callable):
    cwd = os.getcwd()
```

```
    os.chdir(directory)
    result = callable()
    os.chdir(cwd)
    return result


def main(linker_map_path):
    with open(linker_map_path) as file:
        linker_map = file.read()

    linker_map = linker_map[linker_map.index("Linker script and memory map") :]

    def get_section_size(section):
        match re.search(
            r"\n(" + re.escape(section) + r")\s+(0x[0-9a-f]+)\s+(0x[0-9a-f]+)",
            linker_map,
        ):
            case None:
                raise Exception(f"Section {section} was not found")
            case m:
                return int(m.group(3), 16)

    def format_bytes(num_of_bytes):
        if num_of_bytes < 1024:
            return f"{num_of_bytes} B"
        elif num_of_bytes < 1024**2:
            return f"{num_of_bytes/1024:.1f} KiB"

    text_size = get_section_size(".text")
    rodata_size = get_section_size(".rodata")
    data_size = get_section_size(".data")
    bss_size = get_section_size(".bss")
    main_stack_size = 0x1000

    @dataclass
    class SpecialStaticVariable:
        description: str
        variable_name: str

        def size(self):
            try:
                return get_section_size(f" .bss.{self.variable_name}")
            except:
                return 0

    special_static_variables = [
        SpecialStaticVariable(
            description="idle task stack", variable_name="f_xIdleTaskStack"
        ),
        SpecialStaticVariable(
            description="timer task stack", variable_name="f_xTimerTaskStack"
        ),
        SpecialStaticVariable(
            description="motor controller stack",
            variable_name="f_axMotorControllerTaskStack",
        ),
        SpecialStaticVariable(
            description="comm rx task stack",
```

```
            variable_name="f_comm_rx_do_systickTaskStack",
        ),
        SpecialStaticVariable(
            description="comm tx task stack",
            variable_name="f_comm_tx_do_systickTaskStack",
        ),
    ]

    flash_size = 512 * 1024
    flash_used = text_size + rodata_size
    ram_size = 64 * 1024
    ram_used = data_size + bss_size + main_stack_size
    ccram_size = 16 * 1024
    ccram_used = 0

    print(f'+{"-"*24}+{"-"*24}+{"-"*24}+')
    print(f'|{f"FLASH":^24}|{f"program ":>24}|{f"{format_bytes(text_size)} ":>24}|')
    print(f'|{"":^24}|{f"constants ":>24}|{f"{format_bytes(rodata_size)} ":>24}|')
    print(f'|{" "*24}+{"-"*24}+{"-"*24}+')
    print(f'|{"":^24}|{f"total used ":>24}|{f"{format_bytes(flash_used)} ":>24}|')
    print(
        f'|{"":^24}|{f"not used ":>24}|{f"{format_bytes(flash_size - flash_used)}
":>24}|'
    )
    print(f'+{"-"*24}+{"-"*24}+{"-"*24}+')
    print(
        f'|{f"RAM":^24}|{f"static variables ":>24}|{f"{format_bytes(data_size + bss_size
 - sum([variable.size() for variable in special_static_variables]))} ":>24}|'
    )
    for variable in special_static_variables:
        print(
            f'|{"":^24}|{f"{variable.description} ":>24}|{f"{format_bytes(variable.size
())} ":>24}|'
        )
    print(f'|{"":^24}|{f"ISR stack ":>24}|{f"{format_bytes(main_stack_size)} ":>24}|')
    print(f'|{" "*24}+{"-"*24}+{"-"*24}+')
    print(f'|{"":^24}|{f"total ":>24}|{f"{format_bytes(ram_used)} ":>24}|')
    print(
        f'|{"":^24}|{f"not used ":>24}|{f"{format_bytes(ram_size - ram_used)} ":>24}|'
    )
    print(f'+{"-"*24}+{"-"*24}+{"-"*24}+')
    print(f'|{f"CCRAM":^24}|{f"total ":>24}|{f"{format_bytes(ccram_used)} ":>24}|')
    print(
        f'|{"":^24}|{f"not used ":>24}|{f"{format_bytes(ccram_size - ccram_used)}
":>24}|'
    )
    print(f'+{"-"*24}+{"-"*24}+{"-"*24}+')
    print()

    print("32 largest static variables")
    print(f'+{"-"*32}+{"-"*12}+')
    total_size_highest_32 = 0
    for variable_name, address, size, file_path in sorted(
        (
            re.findall(
                r"\n \.bss\.([\w\.]+)\s+(0x[0-9a-f]+)\s+(0x[0-9a-f]+) (.+)", linker_map
            )
            + re.findall(
```

```
                r"\n \.data\.([\w\.]+)\s+(0x[0-9a-f]+)\s+(0x[0-9a-f]+) (.+)", linker_map
            )
        ),
        key=lambda m: int(m[2], 16),
        reverse=True,
    )[:32]:
        if variable_name in [
            variable.variable_name for variable in special_static_variables
        ]:
            continue
        size = int(size, 16)
        total_size_highest_32 += size
        if file_path.startswith("CMakeFiles/pses_ucboard.dir/"):
            file_path = file_path[len("CMakeFiles/pses_ucboard.dir/") :]
        print(f'|{f" {variable_name}":<32}|{f"{format_bytes(size)} ":>12}| {file_path}')
    print(f'|{f" total":<32}|{f"{format_bytes(total_size_highest_32)} ":>12}|')
    print(f'+{"-"*32}+{"-"*12}+')


if __name__ == "__main__":
    argument_parser = argparse.ArgumentParser()
    for parameter in inspect.signature(main).parameters:
        argument_parser.add_argument(parameter)
    arguments = argument_parser.parse_args()
    main(**vars(arguments))
```

A.3.6 analyze_stack_usage.py

```
#!/usr/bin/python3

import argparse
import inspect
import os
import subprocess
import time
import re
from dataclasses import dataclass


def r(*path):
    """
    Takes a relative path from the directory of this python file and returns the
    absolute path.
    """
    return os.path.join(os.path.dirname(os.path.abspath(__file__)), *path)


def run_in_dir(directory, callable):
    cwd = os.getcwd()
    os.chdir(directory)
    result = callable()
    os.chdir(cwd)
    return result


def main(compile_output_dir_path):
    def format_bytes(num_of_bytes):
        if num_of_bytes < 1024:
```

```
            return f"{num_of_bytes} B"
        elif num_of_bytes < 1024**2:
            return f"{num_of_bytes/1024:.1f} KiB"

    @dataclass
    class FunctionStackUsage:
        su_file_path: str
        file_path: str
        line_number: int
        column_number: int
        function_name: str
        stack_usage: int
        staticness: str

    all_functions = []
    for root, dirs, filenames in os.walk(compile_output_dir_path):
        for filename in filter(lambda filename: filename.endswith(".su"), filenames):
            su_file_path = os.path.join(root, filename)
            with open(su_file_path) as file:
                stack_usage_info = file.read()
            for (
                file_path,
                line_number,
                column_number,
                function_name,
                stack_usage,
                staticness,
            ) in re.findall(
                r"(.+):(\d+):(\d+):([\w\.]+)\s+(\d+)\s*(.*)\n", stack_usage_info
            ):
                all_functions.append(
                    FunctionStackUsage(
                        su_file_path,
                        file_path,
                        int(line_number),
                        int(column_number),
                        function_name,
                        int(stack_usage),
                        staticness,
                    )
                )

    print("64 functions with the highest stack usage")
    print(f'+{"-"*48}+{"-"*12}+')
    for function in sorted(all_functions, key=lambda f: f.stack_usage, reverse=True)[
        :64
    ]:
        print(
            f'|{f" {function.function_name}":<48}|{f"{format_bytes(function.stack_usage)
    } ":>12}|'
        )
    print(f'+{"-"*48}+{"-"*12}+')
    print()
    print(
        "NOTE: This analysis only looks at the stack usage of each function isolated and
    "
    )
    print(
```

```
        "does not take into account how much stack each function actually needs when calling"
    )
    print(
        "other functions. This is not even possible to determine by static analysis alone."
    )


if __name__ == "__main__":
    argument_parser = argparse.ArgumentParser()
    for parameter in inspect.signature(main).parameters:
        argument_parser.add_argument(parameter)
    arguments = argument_parser.parse_args()
    main(**vars(arguments))
```

A.4 Pyserial-UcBoard source code

This section provides the source code in the Pyserial-UcBoard repository. This is the code being run on an attached PC to automate the experiments conducted in this work. The code is also available online at `https://gitlab.stud.idi.ntnu.no/sigurdht/pyserial-ucboard`.

A.4.1 automate.py

```
#!/usr/bin/python3

import argparse
import inspect
import os
import sys
import subprocess
import time
import itertools
import re
import numpy as np
import random


def r(*path):
    """
    Takes a relative path from the directory of this python file and returns the absolute path.
    """
    return os.path.join(os.path.dirname(os.path.abspath(__file__)), *path)


def run_in_dir(directory, callable):
    cwd = os.getcwd()
    os.chdir(directory)
    result = callable()
    os.chdir(cwd)
    return result


```

```
def parse_data_from_serial_output(serial_output_file, group_config: dict):
    result = {
        group_number: {channel: [] for channel in channels}
        for group_number, channels in group_config.items()
    }
    with open(serial_output_file) as file:
        for i, line in enumerate(filter(lambda line: line[0] == "#", file.readlines())):
            if "\x02" in line:
                continue
            group_number_match = re.compile(r"#(\d+):").search(line)
            if group_number_match is None or group_number_match.start() != 0:
                # raise Exception(
                #     f"Line {i} starts with # but has no recognizable group number"
                # )
                continue
            group_number = group_number_match.group(1)
            if group_number not in group_config.keys():
                continue
            channel_values = line[group_number_match.end() :].split(" | ")
            if len(channel_values) != len(group_config[group_number]):
                continue
            for i, value in enumerate(channel_values):
                try:
                    value = float(value)
                except ValueError:
                    value = None
                result[group_number][group_config[group_number][i]].append(value)
    return result


HALL_SENSOR_DELTA_S = 0.204204  # Diameter of wheel in meters


def cruise_controller_step_response(python_path: str, serial_port: str):
    setup_commands = "\n".join(
        [
            "set command_delay 0",
            "",
            "!reset now",
            "wait 2",
            "",
            "!steer 450",
            "!drv f 100",
            "wait 5",
            "",
            "!drv f -100",
            "wait 2",
        ]
    ).encode("ascii")
    # subprocess.run([python_path, r("pyserial_ucboard.py"), "-e", serial_port], input=
    setup_commands, stdout=sys.stdout)
    t_s = 50
    for k_p in [0, 50, 100, 200, 500, 1000, 2000]:
        for k_i in [0, 5, 10, 20, 50, 100, 200, 500]:
            for requested_speed in [500, 1000, 1500]:
                output_file = r(
                    f"automated_outputs/cruise_controller_step_response/serial_output_ts
{t_s}_ff+p{k_p}+i{k_i}_{requested_speed}mmps.txt"
```

```
                )
                if os.path.exists(output_file):
                    continue
                input_commands = "\n".join(
                    [
                        "set command_delay 0",
                        "",
                        f"!globalvar g_carbasicfcts_cruiseTS {t_s}",
                        f"!globalvar g_carbasicfcts_cruiseKP {k_p}",
                        f"!globalvar g_carbasicfcts_cruiseKI {k_i}",
                        "wait 1",
                        "",
                        "!daq grp 4 ~all _tics cruise_drvval hall_cnt hall_dt hall_dt8",
                        "!daq start",
                        "",
                        f"!drv c {requested_speed}",
                        "wait 10",
                        "!daq stop",
                        "",
                        "!drv f -100",
                        "wait 2",
                        "",
                    ]
                ).encode("ascii")
                # print(input_commands.decode("ascii"))
                subprocess.run(
                    [
                        python_path,
                        r("pyserial_ucboard.py"),
                        "--output-file",
                        output_file,
                        "-ce",
                        serial_port,
                    ],
                    input=input_commands,
                    stdout=sys.stdout,
                )


def find_feed_forward_values(python_path: str, serial_port: str):
    setup_commands = "\n".join(
        [
            "set command_delay 0",
            "",
            "!reset now",
            "wait 2",
            "",
            "!steer -800",
            "!drv f 300",
            "wait 5",
            "",
            "!drv f -100",
            "wait 2",
        ]
    ).encode("ascii")
    # subprocess.run([python_path, r("pyserial_ucboard.py"), "-e", serial_port], input=
    setup_commands, stdout=sys.stdout)
    results = {
```

```
        # "f 50": None,
        # "f 100": None,
        "f 200": None,
        "f 300": None,
        "f 400": None,
        "f 500": None,
        "f 600": None,
        "f 700": None,
        "f 800": None,
        "f 900": None,
        "f 1000": None,
        # "b 50": None,
        # "b 100": None,
        "b 200": None,
        "b 300": None,
        "b 400": None,
        "b 500": None,
    }
    for drvcommand in results.keys():
        output_file = r(
            f"automated_outputs/find_feed_forward_values/serial_output_drv_{drvcommand}.txt"
        )
        # if os.path.exists(output_file):
        #     continue
        input_commands = "\n".join(
            [
                "set command_delay 0",
                "",
                "!daq grp 4 ~all _tics hall_cnt hall_dt hall_dt8",
                "!daq start",
                "",
                f"!drv {drvcommand}",
                "wait 15",
                "!daq stop",
                "",
                "!drv f -100",
                "wait 2",
                "",
            ]
        ).encode("ascii")
        # print(input_commands.decode("ascii"))
        # subprocess.run([python_path, r("pyserial_ucboard.py"), "--output-file", output_file, "-ce", serial_port], input=input_commands, stdout=sys.stdout)
        data = parse_data_from_serial_output(
            output_file,
            {
                "4": ["_tics", "hall_cnt", "hall_dt", "hall_dt8"],
            },
        )
        speed = HALL_SENSOR_DELTA_S / (np.array(data["4"]["hall_dt8"]) * 1e-3)
        # print(speed)
        average_speed = np.average(speed[-len(speed) // 3 :])
        results[drvcommand] = average_speed
    results["f 0"] = 0
    print(results)

    results_csv_file = r("automated_outputs/find_feed_forward_values/results.csv")
```

```
    results_config_file = r("automated_outputs/find_feed_forward_values/results.h")
    # os.makedirs(os.path.dirname(results_csv_file), exist_ok=True)
    result_numbers = [
        (
            int(key[2:]) * (1 if key[0] == "f" else -1),
            results[key] * (1 if key[0] == "f" else -1),
        )
        for key in results
    ]
    result_numbers = sorted(result_numbers, key=lambda pair: pair[0])

    with open(results_csv_file, "w") as file:
        file.write(
            "\n".join([",".join([str(col) for col in cols]) for cols in result_numbers])
        )
    with open(results_config_file, "w") as file:
        pairs = [f"{{{1000*speed:.0f},{drvval}}}" for drvval, speed in result_numbers]
        content = (
            "#define CONFIG_CRUISE_CONTROL_FF_REFERENCE {"
            + ",".join(pairs)
            + "}\n#define CONFIG_CRUISE_CONTROL_FF_REFERENCE_LENGTH "
            + str(len(pairs))
            + "\n"
        )
        file.write(content)


def dos_attack(python_path: str, serial_port: str):
    setup_commands = "\n".join(
        [
            "set command_delay 0",
            "",
            "!reset now",
            "wait 2",
            "",
            # "!steer 100",
            # "!drv f 100",
            # "wait 5",
            # "",
            # "!drv f -100",
            # "wait 2",
        ]
    ).encode("ascii")
    subprocess.run(
        [python_path, r("pyserial_ucboard.py"), "-e", serial_port],
        input=setup_commands,
        stdout=sys.stdout,
    )
    input_commands = "\n".join(
        [
            "set command_delay 0",
            "",
            "!us on",
            "",
            # *[
            #     f"!daq grp {group_number} ~any _tics _dly ax gy mz usf usl usr usb
    vdbat"
            #     for group_number in range(20)
```

```
            # ],
            # "!daq start",
            # "!drv c 1000",
            *itertools.chain(
                *[
                    [
                        "!led a on",
                        "!led a off",
                    ]
                    for _ in range(3)
                ]
            ),
            "",
            "!drv off",
            "",
        ]
    ).encode("ascii")
    output_file = r("automated_outputs/dos_attack/serial_output.txt")
    # print(input_commands.decode("ascii"))
    subprocess.run(
        [
            python_path,
            r("pyserial_ucboard.py"),
            "--output-file",
            output_file,
            "-cer",
            serial_port,
        ],
        input=input_commands,
        stdout=sys.stdout,
    )


def drv_response_time(python_path: str, serial_port: str):
    for _ in range(50):
        setup_commands = "\n".join(
            [
                "set command_delay 0",
                "",
                "!reset now",
                "wait 2",
                "",
                # "!steer 100",
                # "!drv f 100",
                # "wait 5",
                # "",
                # "!drv f -100",
                # "wait 2",
            ]
        ).encode("ascii")
        subprocess.run(
            [python_path, r("pyserial_ucboard.py"), "-e", serial_port],
            input=setup_commands,
            stdout=sys.stdout,
        )
        delay = random.uniform(0.1, 1)
        # delay = 0
        input_commands = "\n".join(
```

```
            [
                "set command_delay 0",
                "",
                "!drv f 300",
                "wait 2",
                "",
                "!led a on",
                f"wait {delay:.3f}",
                "",
                "# Measure response time",
                "!drv f -300",
                "",
                "!led a off",
                "",
            ]
        ).encode("ascii")
        output_file = r("automated_outputs/drv_response_time/serial_output.txt")
        results_file = r("automated_outputs/drv_response_time/results.csv")
        # print(input_commands.decode("ascii"))
        subprocess.run(
            [
                python_path,
                r("pyserial_ucboard.py"),
                "--output-file",
                output_file,
                "-cer",
                serial_port,
            ],
            input=input_commands,
            stdout=sys.stdout,
        )
        with open(output_file) as serial_output:
            for line in serial_output.readlines():
                response_time_match = re.compile(r":F -300 \((\d+\.\d+) ms\)").search(
                    line
                )
                if response_time_match is None:
                    continue
                response_time = float(response_time_match.group(1))
                os.makedirs(os.path.dirname(results_file), exist_ok=True)
                with open(results_file, "a") as results:
                    delay_ms = delay * 1000
                    results.write(f"{delay_ms:.2f},{response_time:.2f}\n")
                break


def drv_response_time_static(python_path: str, serial_port: str):
    for _ in range(50):
        delay = random.uniform(0.1, 1)
        # delay = 0
        input_commands = "\n".join(
            [
                "set command_delay 0.1",
                "",
                "!reset now",
                "wait 2",
                "",
                "!drv f 300",
```

```
                "wait 2",
                "",
                "set command_delay 0",
                "!led a on",
                f"wait {delay:.3f}",
                "set command_delay 0.1",
                "",
                "# Measure response time",
                "!drv f -300",
                "",
                "!led a off",
                "",
            ]
        ).encode("ascii")
        output_file = r("automated_outputs/drv_response_time/serial_output.txt")
        results_file = r("automated_outputs/drv_response_time/results.csv")
        # print(input_commands.decode("ascii"))
        subprocess.run(
            [
                python_path,
                r("pyserial_ucboard.py"),
                "--output-file",
                output_file,
                "-cer",
                "-t",
                "0",
                serial_port,
            ],
            input=input_commands,
            stdout=sys.stdout,
        )
        with open(output_file) as serial_output:
            for line in serial_output.readlines():
                response_time_match = re.compile(r":F -300 \((\d+\.\d+) ms\)").search(
                    line
                )
                if response_time_match is None:
                    continue
                response_time = float(response_time_match.group(1))
                os.makedirs(os.path.dirname(results_file), exist_ok=True)
                with open(results_file, "a") as results:
                    delay_ms = delay * 1000
                    results.write(f"{delay_ms:.2f},{response_time:.2f}\n")
                break


def main(function_name: str, python_path: str, serial_port: str):
    try:
        globals().get(function_name)(python_path, serial_port)
    except KeyboardInterrupt:
        subprocess.run(
            [python_path, r("pyserial_ucboard.py"), "-e", serial_port],
            input="!led a off".encode("ascii"),
            stdout=sys.stdout,
        )
        subprocess.run(
            [python_path, r("pyserial_ucboard.py"), "-e", serial_port],
            input="!drv off".encode("ascii"),
```

```
            stdout=sys.stdout,
        )


if __name__ == "__main__":
    argument_parser = argparse.ArgumentParser()
    for parameter in inspect.signature(main).parameters:
        argument_parser.add_argument(parameter)
    arguments = argument_parser.parse_args()
    main(**vars(arguments))
```

A.4.2 dummy_ucboard.py

```
#!/usr/sbin/python3

# usage: dummy_ucboard.py [-h] [--baudrate BAUDRATE] [--delay DELAY] serial_port

# This script can be used for testing/debugging `pyserial_ucboard.py` or other command
# passing workflows. To use it, firstly set up a virutal serial port connection. On
# linux this can be done with socat: `socat -d -d pty,raw,echo=0 pty,raw,echo=0` This
# command will output two serial devices (e.g. `/dev/pts/X` and `/dev/pts/Y`) that are
# now connected. Leave this process running. Now, in another terminal, run `python
# dummy_ucboard.py /dev/pts/X` and leave also this process running. Then, run your
# command passing workflow with `/dev/pts/Y` as serial port. The dummy is, as the name
# suggests, quite stupid and can only print the commands it receives and respond with
# `:ok`.

# positional arguments:
#   serial_port           Serial communication port to use

# options:
#   -h, --help            show this help message and exit
#   --baudrate BAUDRATE, -b BAUDRATE
#                         Baudrate for serial communication (default: 921600)
#   --delay DELAY, -d DELAY
#                         Delay in seconds before responding to commands (default:
#                         0.001)

import serial
import serial.tools.list_ports
import sys
import time
import threading
import argparse
import os
import time

def r(*path):
    """
    Takes a relative path from the directory of this python file and returns the
    absolute path.
    """
    return os.path.join(os.path.dirname(os.path.abspath(__file__)), *path)


if __name__ == "__main__":
    class ArgumentParserCustom(argparse.ArgumentParser):
        def error(self, message):
```

```
            sys.stderr.write(f"error: {message}\n")
            self.print_help()
            sys.exit(2)

    argument_parser = ArgumentParserCustom(
        epilog="Available serial ports are: "+", ".join([str(port) for port in  serial.tools.list_ports.comports()]),
        formatter_class=argparse.ArgumentDefaultsHelpFormatter,
        description="""\
This script can be used for testing/debugging `pyserial_ucboard.py` or other command passing workflows.
To use it, firstly set up a virutal serial port connection. On linux this can be done with socat:
`socat -d -d pty,raw,echo=0 pty,raw,echo=0`
This command will output two serial devices (e.g. `/dev/pts/X` and `/dev/pts/Y`) that are now connected. Leave this process running.
Now, in another terminal, run `python dummy_ucboard.py /dev/pts/X` and leave also this process running.
Then, run your command passing workflow with `/dev/pts/Y` as serial port.
The dummy is, as the name suggests, quite stupid and can only print the commands it receives and respond with `:ok`."""
    )
    argument_parser.add_argument("--baudrate", "-b", type=int, default=921600, help="Baudrate for serial communication")
    argument_parser.add_argument("--delay", "-d", type=float, default=.001, help="Delay in seconds before responding to commands")
    argument_parser.add_argument("serial_port", type=str, help="Serial communication port to use")
    args = argument_parser.parse_args()

    serial_port = args.serial_port

    try:
        serial_connection = serial.Serial(port=serial_port, baudrate=args.baudrate, timeout=.1)
    except Exception as exception:
        print(exception)
        print("Something went wrong. Did you enter the correct serial port?")
        print("Available ports are:")
        for port in serial.tools.list_ports.comports():
            print(f"    {port}")
        print("If you are on Linux you might have to get permission to read/write to the device with")
        print(f"    sudo chmod a+rw {serial_port}")
        sys.exit(1)

    print(f"Successfully connected to commander on {serial_port}")
    print("Press Ctrl+C to exit")

    try:
        while True:
            command = serial_connection.read_until(expected=b"\n")
            if command == b"":
                continue
            command = command.decode(encoding="ascii")
            print(command, end="")
            time.sleep(args.delay)
            serial_connection.write(b"\x02:ok\x03")
```

```
    except (KeyboardInterrupt):
        # Exit on Ctrl+C
        pass
```

A.4.3 generate_commands.py

```
#!/usr/sbin/python3

# usage: generate_commands.py [-h] [--output-file OUTPUT_FILE]

# This scripts generates command input for `pyserial_ucboard.py`. Edit this script
# directly to change which commands it generates. Commands could either be stored to a
# file which is then used as input file for `pyserial_ucboard.py` like so: `python
# generate_commands.py -o commands.txt && python pyserial_ucboard.py -i commands.txt -e
# /dev/ttyUSBX`, or you can pipe the commands directly like so: `python
# generate_commands.py | python pyserial_ucboard.py -e /dev/ttyUSBX`. `

# options:
#   -h, --help            show this help message and exit
#   --output-file OUTPUT_FILE, -o OUTPUT_FILE
#                         Path to file for storing output

import sys
import time
import threading
import argparse
import os
import time
import itertools


def r(*path):
    """
    Takes a relative path from the directory of this python file and returns the
    absolute path.
    """
    return os.path.join(os.path.dirname(os.path.abspath(__file__)), *path)

def run_in_dir(directory, callable):
    cwd = os.getcwd()
    os.chdir(directory)
    result = callable()
    os.chdir(cwd)
    return result

commands = [
    "set command_delay 0",
    "",
    "!reset now",
    "wait 2",
    "",
    "!steer 100",
    "wait 1",
    "",
    *itertools.chain(
        *[[
            f"# Driving 10s with drvval {drvval}",
            f"!drv f {drvval}",
```

```
            "wait 10",
            "!drv f -100",
            "wait 2",
            "",
        ] for drvval in range(100, 1001, 100)]
    ),
]

# print(commands); exit()

commands_txt = "\n".join(commands)

if __name__ == "__main__":
    argument_parser = argparse.ArgumentParser(
        description="""\
This scripts generates command input for `pyserial_ucboard.py`.
Edit this script directly to change which commands it generates.
Commands could either be stored to a file which is then used as input file for `pyserial_ucboard.py` like so:
`python generate_commands.py -o commands.txt && python pyserial_ucboard.py -i commands.txt -e /dev/ttyUSBX`,
or you can pipe the commands directly like so:
`python generate_commands.py | python pyserial_ucboard.py -e /dev/ttyUSBX`.
""",
    )
    argument_parser.add_argument("--output-file", "-o", type=str, default=None, help="Path to file for storing output")
    args = argument_parser.parse_args()
    output_file = sys.stdout
    if args.output_file is not None:
        output_file = open(args.output_file, "w")
    print(commands_txt, file=output_file)
```

A.4.4 pyserial_ucboard.py

```
#!/usr/sbin/python3

# usage: pyserial_ucboard.py [-h] [--baudrate BAUDRATE] [--timeout TIMEOUT]
#                            [--history-file HISTORY_FILE] [--input-file INPUT_FILE]
#                            [--exit-after-input-commands] [--output-file OUTPUT_FILE]
#                            [--clear-output-file] [--response-time]
#                            serial_port [command ...]

# positional arguments:
#   serial_port           Serial communication port to use
#   command               Send command, listen for response and exit (default: None)

# options:
#   -h, --help            show this help message and exit
#   --baudrate BAUDRATE, -b BAUDRATE
#                         Baudrate for serial communication (default: 921600)
#   --timeout TIMEOUT, -t TIMEOUT
#                         Timeout in seconds when waiting for response (default: 3)
#   --history-file HISTORY_FILE, -f HISTORY_FILE
#                         Path to file for reading and storing history of previously
#                         sent commands (default: None)
#   --input-file INPUT_FILE, -i INPUT_FILE
#                         Path to file for input commands. Will be overridden by
```

```
#                         command provided directly through positional argument
#                         (default: None)
#   --exit-after-input-commands, -e
#                         Exit after provided input commands are executed (default:
#                         False)
#   --output-file OUTPUT_FILE, -o OUTPUT_FILE
#                         Path to file for storing output (default: None)
#   --clear-output-file, -c
#                         Clear output file on startup (default: False)
#   --response-time, -r   Prints how long time it takes from a command is sent until
#                         the direct response is received (default: False)

import serial
import serial.tools.list_ports
import sys
import time
import threading
import argparse
import os
import time

from prompt_toolkit import PromptSession
from prompt_toolkit.key_binding import KeyBindings
from prompt_toolkit.key_binding.bindings.named_commands import end_of_line
from prompt_toolkit.auto_suggest import AutoSuggestFromHistory
from prompt_toolkit.history import FileHistory
from prompt_toolkit.patch_stdout import patch_stdout

def r(*path):
    """
    Takes a relative path from the directory of this python file and returns the
    absolute path.
    """
    return os.path.join(os.path.dirname(os.path.abspath(__file__)), *path)


def read_message_from_ucboard(serial_connection: serial.Serial):
    # output = serial_connection.read()
    # return None if output == b"" else output.decode(encoding="ascii")
    # Read until start of message
    output = serial_connection.read_until(expected=b"\x02")

    if output == b"":
        return None

    # Read until end of message, do not include the \x03 in result, strip eventual extra
     leading \x02
    # and decode (convert from bytes to str)
    return (serial_connection.read_until(expected=b"\x03")[:-1]).strip(b"\x02").decode(
    encoding="ascii")


if __name__ == "__main__":
    class ArgumentParserCustom(argparse.ArgumentParser):
        def error(self, message):
            sys.stderr.write(f"error: {message}\n")
            self.print_help()
            sys.exit(2)
```

```
79
80      argument_parser = ArgumentParserCustom(
81          epilog="Available serial ports are: "+", ".join([str(port) for port in  serial.
        tools.list_ports.comports()]),
82          formatter_class=argparse.ArgumentDefaultsHelpFormatter,
83      )
84      argument_parser.add_argument("--baudrate", "-b", type=int, default=921600, help="
        Baudrate for serial communication")
85      argument_parser.add_argument("--timeout", "-t", type=float, default=3, help="Timeout
         in seconds when waiting for response")
86      argument_parser.add_argument("--history-file", "-f", type=str, default=None, help="
        Path to file for reading and storing history of previously sent commands")
87      argument_parser.add_argument("--input-file", "-i", type=str, default=None, help="
        Path to file for input commands. Will be overridden by command provided directly
        through positional argument")
88      argument_parser.add_argument("--exit-after-input-commands", "-e", action="store_true
        ", help="Exit after provided input commands are executed")
89      argument_parser.add_argument("--output-file", "-o", type=str, default=None, help="
        Path to file for storing output")
90      argument_parser.add_argument("--clear-output-file", "-c", action="store_true", help=
        "Clear output file on startup")
91      argument_parser.add_argument("--response-time", "-r", action="store_true", help="
        Prints how long time it takes from a command is sent until the direct response is
        received")
92      argument_parser.add_argument("serial_port", type=str, help="Serial communication
        port to use")
93      argument_parser.add_argument("command", type=str, nargs="*", help="Send command,
        listen for response and exit")
94      args = argument_parser.parse_args()
95
96      serial_port = args.serial_port
97
98      try:
99          serial_connection = serial.Serial(port=serial_port, baudrate=args.baudrate,
        timeout=.1)
100     except Exception as exception:
101         print(exception)
102         print("Something went wrong. Did you enter the correct serial port?")
103         print("Available ports are:")
104         for port in serial.tools.list_ports.comports():
105             print(f"    {port}")
106         print("If you are on Linux you might have to get permission to read/write to the
         device with")
107         print(f"    sudo chmod a+rw {serial_port}")
108         sys.exit(1)
109
110     time.sleep(0.1)
111
112     if args.output_file is not None:
113         os.makedirs(os.path.dirname(args.output_file), exist_ok=True)
114         if args.clear_output_file:
115             with open(args.output_file, "w") as output_file:
116                 output_file.write("")
117
118     exit_event = threading.Event()
119     response_received_event = threading.Event()
120
121     output_lock = threading.Lock()
```

```
122
123     last_command_time = time.time()
124
125     def print_to_output_file(output, *print_args, **print_kwargs):
126         if args.output_file is not None:
127             with open(args.output_file, "a") as output_file:
128                 print(output, *print_args, file=output_file, **print_kwargs)
129
130     def read_output_thread():
131         global serial_connection, exit_event, last_command_time
132
133         while not exit_event.is_set():
134             message = read_message_from_ucboard(serial_connection)
135
136             if message is None or message == "":
137                 continue
138
139             output_lock.acquire()
140             if message[0] == ":":
141                 response_received_event.set()
142                 if args.response_time:
143                     time_since_last_command_ms = (time.time() - last_command_time) *
    1000
144                     message = f"{message} ({time_since_last_command_ms:.2f} ms)"
145             print(message)
146             print_to_output_file(message)
147             output_lock.release()
148
149     def send_command(command):
150         global serial_connection, last_command_time
151         response_received_event.clear()
152         serial_connection.write((command + "\n").encode(encoding="ascii"))
153         last_command_time = time.time()
154
155     if len(args.command) > 0:
156         command = " ".join(args.command)
157         command = (command + "\n").encode(encoding="ascii")
158         serial_connection.write(command)
159         print_to_output_file(f"> {command}")
160         while True:
161             answer = read_message_from_ucboard(serial_connection)
162             if answer is None:
163                 break
164             print(answer)
165             print_to_output_file(answer)
166         if args.exit_after_input_commands:
167             sys.exit()
168
169     threading.Thread(target=read_output_thread).start()
170
171     print(f"Successfully connected to ucboard on {serial_port}")
172     print("Type the command you want to send below and press ENTER to send it")
173     print("Press Ctrl+C to exit")
174
175     try:
176         input_commands = None
177         if args.input_file is not None:
178             with open(args.input_file) as input_file:
```

```
            input_commands = input_file.read().split("\n")
    elif not sys.stdin.isatty():
        input_commands = sys.stdin.read().split("\n")

    if input_commands is not None:
        command_delay = 0.1
        for command in input_commands:
            if command == "" or command[0] == "#":
                # Ignore empty lines and comments
                continue
            if command[0] in ["!", "?"]:
                # Command to ucboard
                output_lock.acquire()
                send_command(command)
                print(f"> {command}")
                print_to_output_file(f"> {command}")
                output_lock.release()
                time.sleep(command_delay)
                response_received_event.wait(timeout=args.timeout)
                continue
            if command.startswith("wait"):
                # Wait for the specified number of seconds
                time.sleep(float(command[len("wait "):]))
            if command.startswith("set command_delay"):
                command_delay = float(command[len("set command_delay "):])
        if args.exit_after_input_commands:
            raise KeyboardInterrupt()
    prompt_session = PromptSession(
        history=FileHistory(args.history_file or r("history.pthistory")),
        auto_suggest=AutoSuggestFromHistory(),
    )
    while True:
        with patch_stdout():
            command = prompt_session.prompt("> ")
        output_lock.acquire()
        send_command(command)
        print_to_output_file(f"> {command}")
        output_lock.release()
        response_received_event.wait(timeout=args.timeout)
except (KeyboardInterrupt, EOFError):
    # Exit on Ctrl+C and Ctrl+D
    exit_event.set()
```

B Compile time analysis output

B.1 Analyze stack usage

In this section, output from running the script analyze_stack_usage.py on our implementation (routine configuration B) is listed.

64 functions with the highest stack usage

eeprom_init	2.1 KiB
display_printerror	1016 B
parseGrpDef	528 B
cmd_drv	408 B
cmd_imu	320 B
cmd_us	320 B
cmd_daq	312 B
cmd_mag	304 B
cmd_bldc	288 B
streamout	256 B
cmd_encoder	232 B
us_do_systick	224 B
cruise_control_getNewDrvVal	176 B
daq_do_systick	168 B
SystemClock_Config	160 B
cmd_dbg	152 B
cmd_led	144 B
us_init	136 B
cmd_commstats	136 B
cmd_reset	120 B
cmd_tics	120 B
cmd_sessionid	120 B
cmd_version	120 B
cmd_vout12v	120 B
cmd_steer	120 B
comm_parseSubcmdArgs	120 B
cmd_eeprom	120 B
cmd_globalvar	120 B
cmd_rcmode	120 B
cmd_carid	120 B

comm_parseArgs	112 B
HAL_ADCEx_MultiModeStart_DMA	112 B
HAL_ADC_Init	104 B
HAL_ADC_DeInit	104 B
HAL_ADCEx_MultiModeStop_DMA	104 B
HAL_ADCEx_RegularMultiModeStop_DMA	104 B
HAL_ADCEx_InjectedConfigChannel	104 B
HAL_ADCEx_MultiModeConfigChannel	104 B
HAL_ADC_ConfigChannel	96 B
receiver_init	88 B
bldc_init	80 B
imuMPU9250_init	80 B
imuICM20648_init	80 B
MX_GPIO_Init	72 B
eeprom_streamout	72 B
getGrpDataStringAscii_returnend	64 B
getDataString_returnend	64 B
encoder_init	64 B
spimgr_init	64 B
getIMUConfString_returnend	56 B
createErrStr_returnend	56 B
processUsartCommand	56 B
HAL_ADC_MspInit	48 B
HAL_I2C_MspInit	48 B
HAL_TIM_IC_MspInit	48 B
HAL_TIM_MspPostInit	48 B
MX_TIM2_Init	48 B
initPWM	48 B
display_println_bits	48 B
hal503_init	48 B
imu_do_systick	48 B
utoa_bits	48 B
HAL_I2C_Mem_Write	48 B
HAL_I2C_Mem_Read	48 B

NOTE: This analysis only looks at the stack usage of each function isolated and
does not take into account how much stack each function actually needs when calling
other functions. This is not even possible to determine by static analysis alone.

B.2 Analyze linker map

In this section, output from running the script analyze_linker_map.py on the present UcBoard software, our implementation with routine configuration A and our implementation with routine configuration B is listed.

B.2.1 Present

```
+-------------------------+-------------------------+-------------------------+
|          FLASH          |                 program |                    62.0 KiB |
|                         |               constants |                     6.6 KiB |
|                         +-------------------------+                             |
|                         |              total used |                    68.6 KiB |
|                         |                not used |                   443.4 KiB |
+-------------------------+-------------------------+-------------------------+
|           RAM           |        static variables |                    22.9 KiB |
|                         |         idle task stack |                       0 B   |
|                         |        timer task stack |                       0 B   |
|                         | motor controller stack |                       0 B   |
|                         |     comm rx task stack |                       0 B   |
|                         |     comm tx task stack |                       0 B   |
|                         |              ISR stack |                     4.0 KiB |
|                         +-------------------------+                             |
|                         |                   total |                    26.9 KiB |
|                         |                not used |                    37.1 KiB |
+-------------------------+-------------------------+-------------------------+
|          CCRAM          |                   total |                       0 B   |
```

```
|                          |                not used |                16.0
   KiB |
+--------------------------+-------------------------+-------------------------+


32 largest static variables
+-----------------------------------+-------------+
| f_grps                            |     9.1 KiB | devices/daq.c.obj
| f_eepromdata                      |     2.0 KiB | devices/eeprom.c.obj
| f_vals                            |     1.1 KiB | devices/daq.c.obj
| f_acRxBuf                         |      1002 B | devices/comm.c.obj
| f_acTxSecondaryBuf                |      1001 B | devices/comm.c.obj
| f_acTxOutstreamBuf                |      1001 B | devices/comm.c.obj
| f_acTxRespBuf                     |      1001 B | devices/comm.c.obj
| f_acBuffer                        |      1000 B | devices/display.c.obj
| f_aSPIConfigs                     |       572 B | stmcommon/spimgr.c.obj
| f_chs                             |       480 B | devices/daq.c.obj
| f_usdevices                       |       400 B | devices/us.c.obj
| __global_locale                   |       364 B | /home/sigurd/
   masteroppgave/uc-board-os/arm-toolchain/arm-gnu-toolchain-12.2.
   mpacbti-rel1-x86_64-arm-none-eabi/bin/../lib/gcc/arm-none-eabi
   /12.2.1/../../../../arm-none-eabi/lib/thumb/v7e-m+fp/hard/libc_nano.a
   (libc_a-locale.o)
| f_aMsgs                           |       320 B | stmcommon/i2cmgr.c.obj
| __sf                              |       312 B | /home/sigurd/
   masteroppgave/uc-board-os/arm-toolchain/arm-gnu-toolchain-12.2.
   mpacbti-rel1-x86_64-arm-none-eabi/bin/../lib/gcc/arm-none-eabi
   /12.2.1/../../../../arm-none-eabi/lib/thumb/v7e-m+fp/hard/libc_nano.a
   (libc_a-findfp.o)
| f_aCurLedSeq                      |       264 B | devices/carui.c.obj
| f_aNewLedSeq                      |       264 B | devices/carui.c.obj
| f_acErrMsg                        |       255 B | devices/daq.c.obj
| f_acErrMsg                        |       255 B | devices/display.c.obj
| f_acBuffer                        |       200 B | devices/daq.c.obj
| f_aI2C                            |       192 B | stmcommon/i2cmgr.c.obj
| g_commCmdFctTable                 |       136 B | devices/comm_cmdtable.c.
   obj
| f_aDevices                        |       120 B | stmcommon/i2cmgr.c.obj
| f_aDevices                        |       120 B | stmcommon/spimgr.c.obj
| huart2                            |       112 B | Src/main.c.obj
| hspi1                             |       100 B | Src/main.c.obj
| f_guestdevice                     |       100 B | devices/us_srf08.c.obj
| hadc4                             |        84 B | Src/main.c.obj
| hadc3                             |        84 B | Src/main.c.obj
| hadc1                             |        84 B | Src/main.c.obj
| hi2c1                             |        76 B | Src/main.c.obj
```

```
| _impure_data                    |       76 B | /home/sigurd/
  masteroppgave/uc-board-os/arm-toolchain/arm-gnu-toolchain-12.2.
  mpacbti-rel1-x86_64-arm-none-eabi/bin/../lib/gcc/arm-none-eabi
  /12.2.1/../../../../arm-none-eabi/lib/thumb/v7e-m+fp/hard/libc_nano.a
  (libc_a-impure.o)
| htim6                           |       60 B | Src/main.c.obj
| total                           |    21.9 KiB |
+---------------------------------+-------------+
```

B.2.2 Routine configuration A

```
+-------------------------+-------------------------+-------------------------+

|          FLASH          |                 program |                    77.5
  KiB |
|                         |               constants |                     7.6
  KiB |
|
  +-------------------------+-------------------------+
|                         |              total used |                    85.0
  KiB |
|                         |                not used |                   427.0
  KiB |
+-------------------------+-------------------------+-------------------------+

|           RAM           |        static variables |                    24.5
  KiB |
|                         |         idle task stack |                     512
   B |
|                         |        timer task stack |                     4.0
  KiB |
|                         | motor controller stack |                     1.0
  KiB |
|                         |     comm rx task stack |                       0
   B |
|                         |     comm tx task stack |                       0
   B |
|                         |               ISR stack |                     4.0
  KiB |
|
  +-------------------------+-------------------------+
|                         |                   total |                    34.0
  KiB |
|                         |                not used |                    30.0
  KiB |
+-------------------------+-------------------------+-------------------------+
```

```
|          CCRAM          |                  total |                      0
    B |
|                         |               not used |                 16.0
   KiB |
+-------------------------+------------------------+-------------------------+
```

32 largest static variables

f_grps	9.1 KiB	src/devices/daq.c.obj
f_eepromdata	2.0 KiB	src/devices/eeprom.c.obj
f_vals	1.1 KiB	src/devices/daq.c.obj
f_sUsart2RxBuffer	1012 B	src/devices/comm.c.obj
f_acTxOutstreamBuf	1001 B	src/devices/comm.c.obj
f_acTxRespBuf	1001 B	src/devices/comm.c.obj
f_acBuffer	1000 B	src/devices/display.c.obj
f_aSPIConfigs	572 B	src/stmcommon/spimgr.c.obj
f_chs	480 B	src/devices/daq.c.obj
f_usdevices	400 B	src/devices/us.c.obj
__global_locale	364 B	/home/sigurd/masteroppgave/uc-board-os/arm-toolchain/arm-gnu-toolchain-12.2.mpacbti-rel1-x86_64-arm-none-eabi/bin/../lib/gcc/arm-none-eabi/12.2.1/../../../../arm-none-eabi/lib/thumb/v7e-m+fp/hard/libc_nano.a(libc_a-locale.o)
f_aMsgs	320 B	src/stmcommon/i2cmgr.c.obj
pxReadyTasksLists	320 B	src/FreeRTOS-Kernel/tasks.c.obj
__sf	312 B	/home/sigurd/masteroppgave/uc-board-os/arm-toolchain/arm-gnu-toolchain-12.2.mpacbti-rel1-x86_64-arm-none-eabi/bin/../lib/gcc/arm-none-eabi/12.2.1/../../../../arm-none-eabi/lib/thumb/v7e-m+fp/hard/libc_nano.a(libc_a-findfp.o)
f_aCurLedSeq	264 B	src/devices/carui.c.obj
f_aNewLedSeq	264 B	src/devices/carui.c.obj
f_acErrMsg	255 B	src/devices/daq.c.obj
f_acErrMsg	255 B	src/devices/display.c.obj
f_acBuffer	200 B	src/devices/daq.c.obj
f_aI2C	192 B	src/stmcommon/i2cmgr.c.obj
g_commCmdFctTable	160 B	src/devices/comm_cmdtable.c.obj
f_aDevices	120 B	src/stmcommon/i2cmgr.c.obj

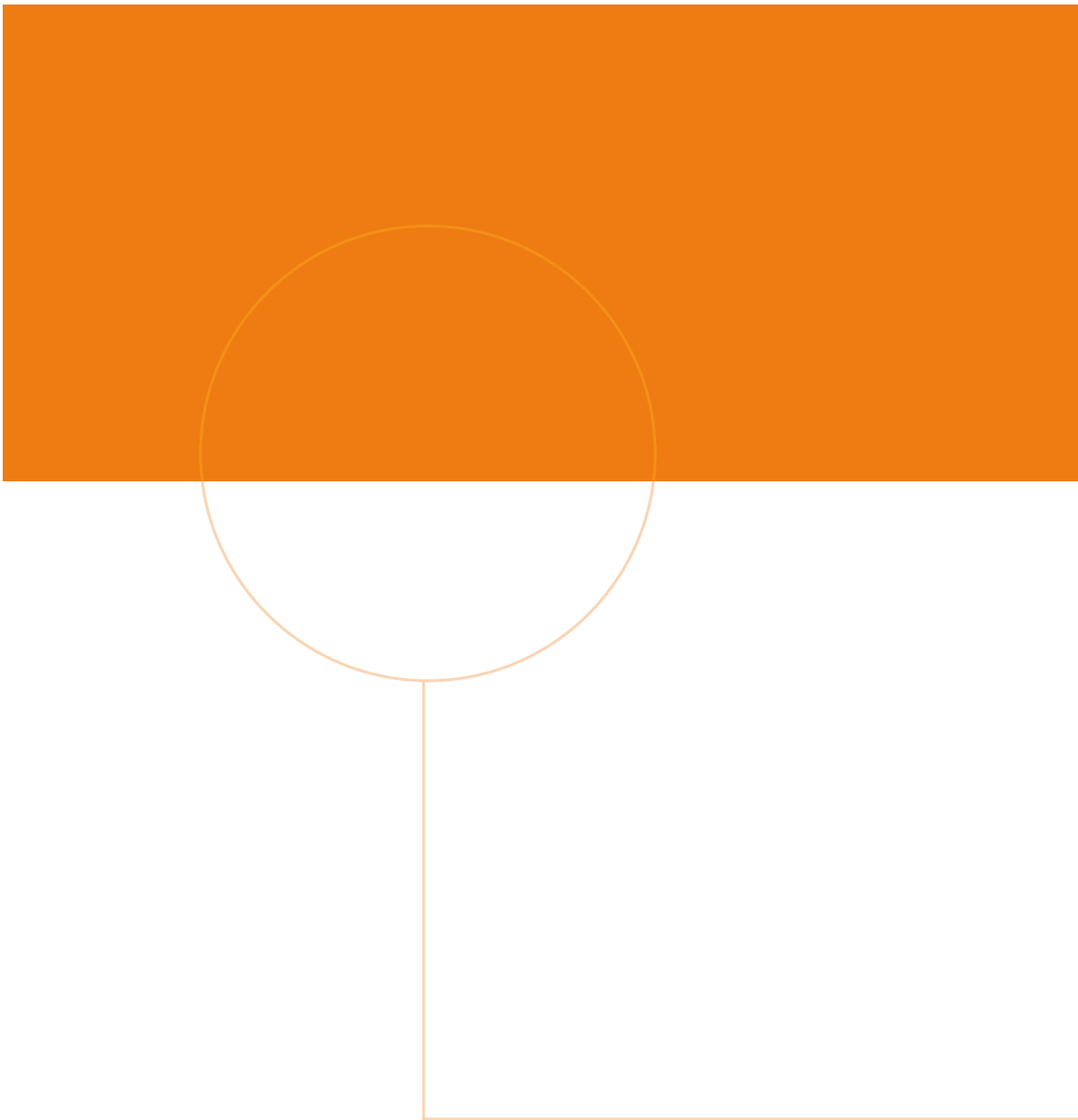
```
| f_aDevices                        |       120 B | src/stmcommon/spimgr.c.
  obj
| ucStaticTimerQueueStorage.1       |       120 B | src/FreeRTOS-Kernel/
  timers.c.obj
| huart                             |       112 B | src/main.c.obj
| hspi2                             |       100 B | src/main.c.obj
| hspi1                             |       100 B | src/main.c.obj
| f_guestdevice                     |       100 B | src/devices/us_srf08.c.
  obj
| f_xTimerTaskStatic                |        92 B | src/main.c.obj
| total                             |   21.1 KiB |
+-----------------------------------+-------------+
```

B.2.3 Routine configuration B

```
+-------------------------+-------------------------+-------------------------+

|          FLASH          |                 program |                 77.7
  KiB |
|                         |               constants |                  7.6
  KiB |
|
  +-------------------------+-------------------------+
|                         |              total used |                 85.3
  KiB |
|                         |                not used |                426.7
  KiB |
+-------------------------+-------------------------+-------------------------+

|           RAM           |        static variables |                 24.6
  KiB |
|                         |         idle task stack |                    512
   B |
|                         |        timer task stack |                  4.0
  KiB |
|                         | motor controller stack |                  1.0
  KiB |
|                         |      comm rx task stack |                  4.0
  KiB |
|                         |      comm tx task stack |                  1.0
  KiB |
|                         |               ISR stack |                  4.0
  KiB |
|
  +-------------------------+-------------------------+
|                         |                   total |                 39.1
  KiB |
```

```
|                          |                 not used |                 24.9
   KiB |
+--------------------------+--------------------------+---------------------+

|          CCRAM           |                    total |                    0
    B |
|                          |                 not used |                 16.0
   KiB |
+--------------------------+--------------------------+---------------------+


32 largest static variables
+--------------------------------+-------------+
| f_grps                         |    9.1 KiB | src/devices/daq.c.obj
| f_eepromdata                   |    2.0 KiB | src/devices/eeprom.c.obj
| f_vals                         |    1.1 KiB | src/devices/daq.c.obj
| f_sUsart2RxBuffer              |     1012 B | src/devices/comm.c.obj
| f_acTxOutstreamBuf             |     1001 B | src/devices/comm.c.obj
| f_acTxRespBuf                  |     1001 B | src/devices/comm.c.obj
| f_acBuffer                     |     1000 B | src/devices/display.c.
   obj
| f_aSPIConfigs                  |      572 B | src/stmcommon/spimgr.c.
   obj
| f_chs                          |      480 B | src/devices/daq.c.obj
| f_usdevices                    |      400 B | src/devices/us.c.obj
| __global_locale                |      364 B | /home/sigurd/
   masteroppgave/uc-board-os/arm-toolchain/arm-gnu-toolchain-12.2.
   mpacbti-rel1-x86_64-arm-none-eabi/bin/../lib/gcc/arm-none-eabi
   /12.2.1/../../../../arm-none-eabi/lib/thumb/v7e-m+fp/hard/libc_nano.a
   (libc_a-locale.o)
| f_aMsgs                        |      320 B | src/stmcommon/i2cmgr.c.
   obj
| pxReadyTasksLists              |      320 B | src/FreeRTOS-Kernel/
   tasks.c.obj
| __sf                           |      312 B | /home/sigurd/
   masteroppgave/uc-board-os/arm-toolchain/arm-gnu-toolchain-12.2.
   mpacbti-rel1-x86_64-arm-none-eabi/bin/../lib/gcc/arm-none-eabi
   /12.2.1/../../../../arm-none-eabi/lib/thumb/v7e-m+fp/hard/libc_nano.a
   (libc_a-findfp.o)
| f_aCurLedSeq                   |      264 B | src/devices/carui.c.obj
| f_aNewLedSeq                   |      264 B | src/devices/carui.c.obj
| f_acErrMsg                     |      255 B | src/devices/daq.c.obj
| f_acErrMsg                     |      255 B | src/devices/display.c.
   obj
| f_acBuffer                     |      200 B | src/devices/daq.c.obj
| f_aI2C                         |      192 B | src/stmcommon/i2cmgr.c.
   obj
```

```
| g_commCmdFctTable                 |       160 B | src/devices/
   comm_cmdtable.c.obj
| f_aDevices                        |       120 B | src/stmcommon/i2cmgr.c.
   obj
| f_aDevices                        |       120 B | src/stmcommon/spimgr.c.
   obj
| ucStaticTimerQueueStorage.1       |       120 B | src/FreeRTOS-Kernel/
   timers.c.obj
| huart                             |       112 B | src/main.c.obj
| hspi2                             |       100 B | src/main.c.obj
| hspi1                             |       100 B | src/main.c.obj
| total                             |    20.9 KiB |
+-----------------------------------+-------------+
```



NTNU
Kunnskap for en bedre verden

TECHNISCHE UNIVERSITÄT DARMSTADT