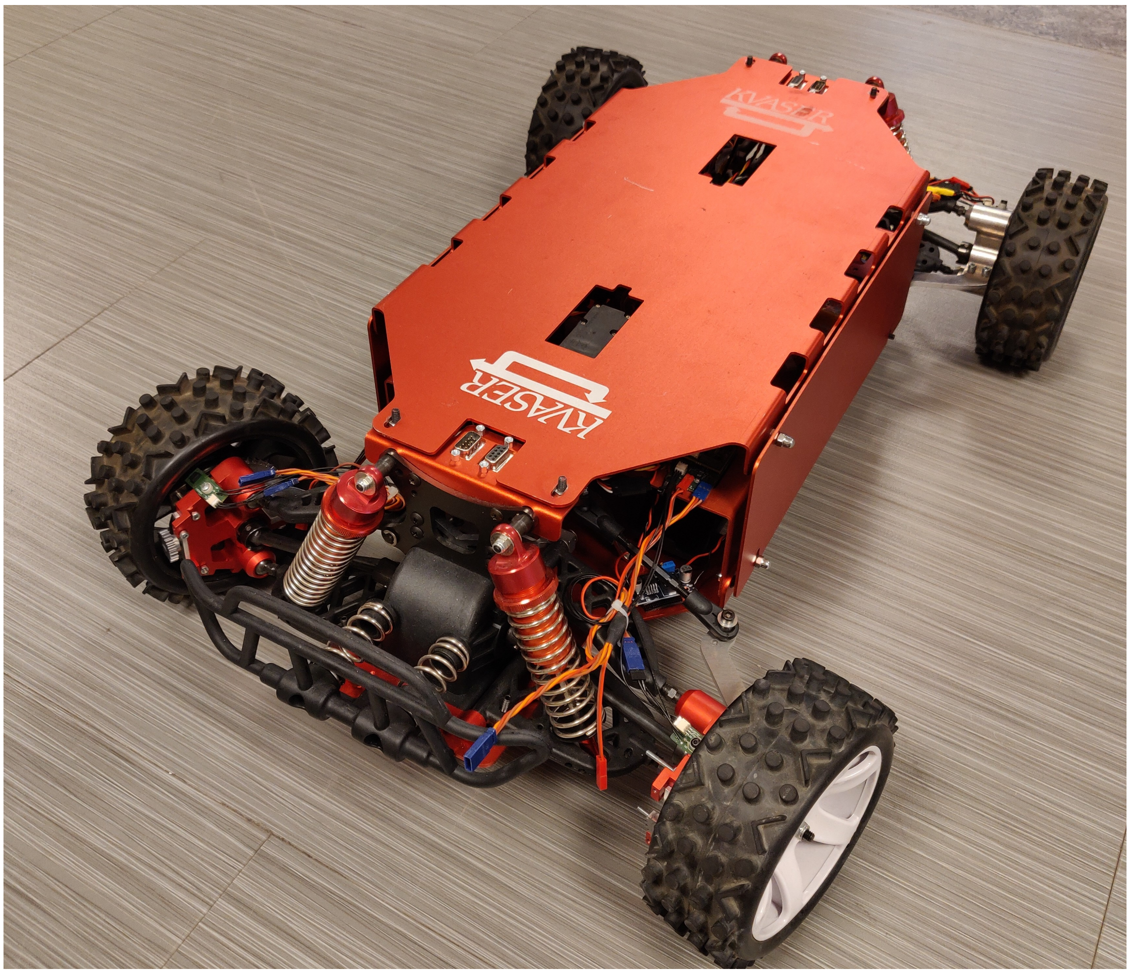

Bare-Metal Implementation of a Real-Time Distributed Control System Using CAN

Master's thesis in Embedded Electronic System Design

Anton Olsson
Robert Sjöberg

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Master's thesis 2022

Bare-Metal Implementation of a Real-Time Distributed Control System Using CAN

Anton Olsson
Robert Sjöberg

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Bare-Metal Implementation of a Real-Time Distributed Control System Using CAN
Anton Olsson, Robert Sjöberg

Supervisor: Jan Jonsson, Department of Computer Science and Engineering
Company advisor: Lars-Berno Fredriksson, Kvaser AB
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Picture of Kvaser's devKit car

Typeset in LaTeX
Gothenburg, Sweden 2022

Bare-Metal Implementation of a Real-Time Distributed Control System Using CAN
Anton Olsson, Robert Sjöberg
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Distributed control systems consisting of several nodes connected via CAN bus are commonplace in many industrial and automotive settings. These systems often require real-time capabilities and generally utilize a real-time operating system (RTOS) in the nodes to achieve this. This thesis work explored the feasibility of developing a bare-metal implementation based on a previous RTOS implementation. Additionally, we explored any real-time performance advantages of the bare-metal implementation and if there were any compatibility issues in combining bare-metal nodes with RTOS nodes.

The thesis work was conducted on a model car with several nodes with specific functions connected via a CAN bus. A node tasked with braking was converted from using an RTOS to a bare-metal implementation. The brake node was tested through a reproducible simulated test run using two different scheduling approaches. It was found that a bare-metal implementation using interrupts produced a more predictable jitter. Faster execution times were also observed due to faster driver implementation. Finally, no issues in combining nodes with different implementations were found. In conclusion, a bare-metal implementation could be beneficial in some applications with low complexity that require a small performance improvement or less jitter.

Keywords: Controller Area Network, Bare-metal, Real-time operating system

Acknowledgements

We would like to thank Kvaser AB for providing the opportunity for us to perform our thesis work at their office. Many thanks to our advisors Lars-Berno Fredriksson, for his advice and guidance, and André Idoffsson, for his technical knowledge and hardware support.

We would also like to thank our supervisor at the department of computer science and engineering, Jan Jonsson, for his commitment, support, and invaluable feedback on the report. We also wish to thank our examiner, Per Larsson-Edefors, for his swift feedback.

Finally, we would like to thank all employees at Kvaser for their helpful discussions and pleasant fika breaks.

Anton Olsson, Robert Sjöberg, Gothenburg, June 2022

Contents

1 Introduction

Control systems are necessary for a wide range of applications in the industry and in commercial products. Often these control systems are distributed over a number of embedded devices that need real-time communication to ensure correct functionality. This requires robust communication protocols that can handle strict real-time requirements, especially for safety critical systems in, for example, the automotive industry.

Controller area network (CAN) is a communication protocol widely used in the industry [1]. It is well suited for real-time communication between nodes in embedded systems due to its low overhead and timeliness [2]. Typically, these nodes run an operating system (OS) to simplify handling e.g. communication, multitasking, and I/O. Many features of an OS are however non-essential for simple nodes and removing these may provide real-time performance gains.

A tailor-made solution without an OS, referred to as a *bare-metal* solution in this thesis, can be used to optimize system performance. This could yield faster communication which is important in many applications with hard real-time requirements. An example of such applications is a control system that relies on fast sensor feedback for accurate control of, e.g. ABS in a car.

The problem addressed in this thesis report was conceived at Kvaser [3]. Kvaser supplies advanced CAN solutions for use in a wide range of applications and has more than 30 years of CAN development experience. Kvaser has a model car, called devKit, containing a distributed embedded control system (DECS) which is the environment for this thesis work. The nodes of this control system are currently using the real-time OS (RTOS) RT-Kernel.

1.1 Aim

This thesis project aims to design and implement a bare-metal solution for brake control nodes and compare its performance to that of brake control nodes implemented with RT-Kernel. The bare-metal implementation needs to be able to provide the same guarantee that all deadlines will be met, as an implementation in RT-Kernel; without the overhead of an OS. It could be expected that a bare-metal solution would result in faster task completion.

A further aim is to investigate if the potential real-time performance gain of the brake nodes can be translated into better real-time performance for the full system.

To be able to test this, however, integration is required between our bare-metal brake node and the remaining system nodes. This integration needs to be evaluated as any encountered integration difficulties could make the bare-metal solution less useful.

To summarize, the main project goals are the following:

- Create a bare-metal implementation of the brake node
- Integrate the bare-metal brake node with the remaining nodes in the devKit-car
- Measure differences in real-time performance between bare-metal and RTOS implementation
- Evaluate system issues with combining nodes with and without RTOS

1.2 Limitations

The limitations of this project are as follows: firstly, we will not create a formal proof of code accuracy and not perform code analysis for worstcase execution time (WCET), as this is outside the scope. Secondly, the control system setup and parameters will not be optimized, since the purpose of this thesis work is to evaluate the effects of the RTOS compared to a bare-metal solution, not to optimize a control system.

1.3 Thesis outline

The rest of this thesis is organized as follows: In chapter 2 the system and the various protocols and concepts it uses are covered along with background on OSes with a focus on RTOSes. In chapter 3 we discuss the testing and design methodology used in this thesis work. The implementation details of the bare-metal solution are described in chapter 4. In chapter 5, the results are presented which are then discussed in chapter 6. Finally, in chapter 7 we present our conclusions.

2

Technical background

This chapter provides the underlying theory to the concepts presented in this thesis. First, a detailed description of the target platform for this thesis is given. Then follows a presentation of the CAN protocol and how it is used in the system. After this follows some theory on operating systems and scheduling. Finally, a brief presentation of some of the concepts in use in the system is given.

2.1 DevKit Car

Kvaser's devKit car, pictured in Fig. 2.1, is the platform for this thesis work. It is a 1/5 scale model remote-controlled car with a distributed embedded control system (DECS) consisting of several sensor and actuator nodes (see Fig. 2.2) connected by a CAN bus. The CAN system in the car utilizes the meta protocol CAN Kingdom [4] (see section 2.3).

In total there are seven nodes in the car as well as two nodes in the remote control unit (see Fig. 2.3). The car and remote are connected to the same CAN bus via a point-to-point radio link called Air Bridge [5]. Each of the nine nodes in the system contains one ARM microcontroller (MCU) STM32F302RE [6] running the RTOS RT-Kernel [7].

Each node is responsible for a specific function. There is one node at each wheel for controlling the brakes. There is also one node for controlling the steering servo, one node for controlling the motor, and one master node responsible for setting up communication. In the remote control, each of the two nodes transmits the states of one joystick and one set of switches.

A special focus of this master thesis project is on the braking nodes. The car has been fitted with anti-locking brakes developed at Kvaser. These nodes measure and transmit the current wheel speed and use that together with the data from the other brake nodes to control the brakes. In the current control system setup only the rear wheels are used for braking. The front wheel nodes are solely used to measure the speed of the car.

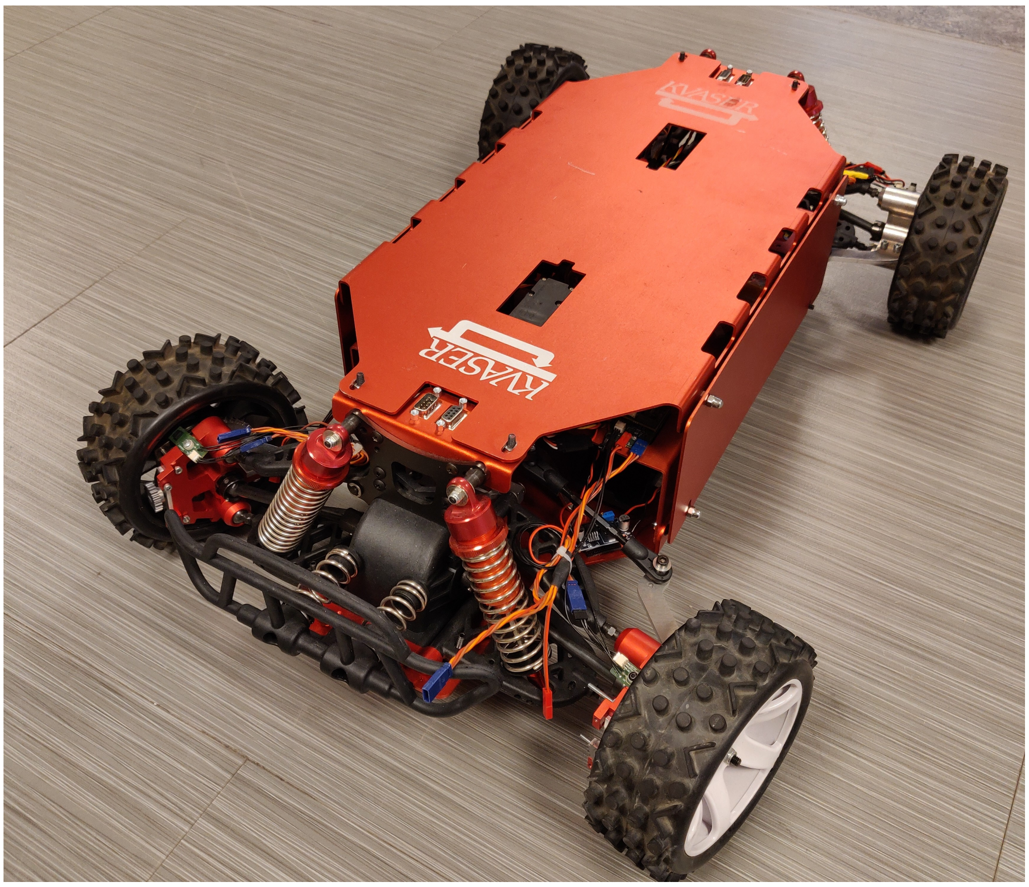

Fig. 2.1: Picture of the devKit car used for this thesis work.

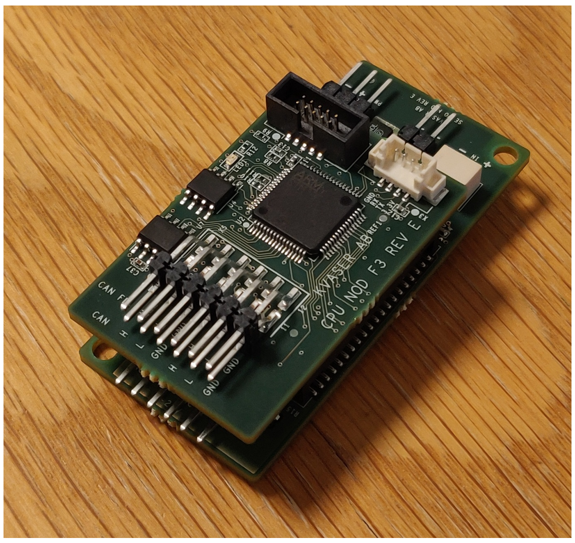

Fig. 2.2: A node from the distributed system.

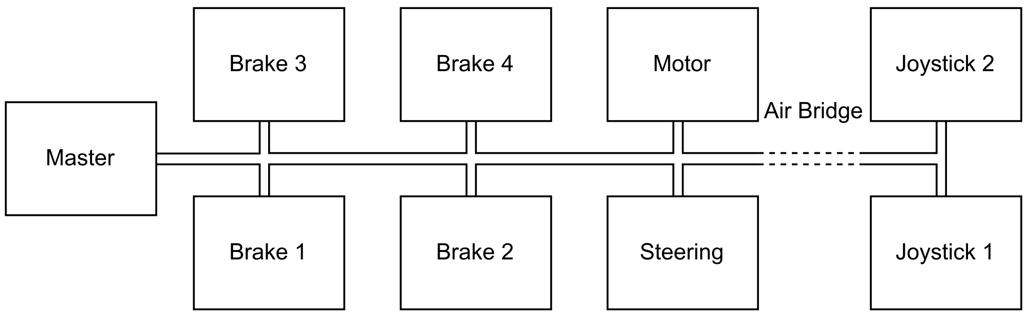

Fig. 2.3: All nodes in the system, connected by CAN bus. The nodes to the left of the Air Bridge are in the car and the nodes to the right are in the remote controller.

2.1.1 Brake node

The brake nodes measure the wheel speed using a gear tooth sensor connected to a gear with 45 teeth that rotates with the wheel. Each time a gear tooth is detected, an interrupt is generated to increment a counter. At a set time interval (10 ms) the counter is read and the wheel speed is calculated using a rolling average of the last five measurements. The calculated speed is immediately sent on the CAN bus.

The rear wheels also have active disc brakes. The disc brakes are controlled by a modified servo motor where the internal control logic has been bypassed to allow faster and more accurate control of the motor. The position sensor of the servo is also given as output for use in the motor control. The position of the servo is obtained by converting the voltage of the position sensor using the ADC. The motor is driven by the H-bridge on the board using PWM signals (see section 2.7) to control the motor power.

A block diagram of a brake node with surrounding peripherals can be seen in Fig. 2.4. Additionally, Fig. 2.5 shows how the servo motor and gear tooth sensor are mounted.

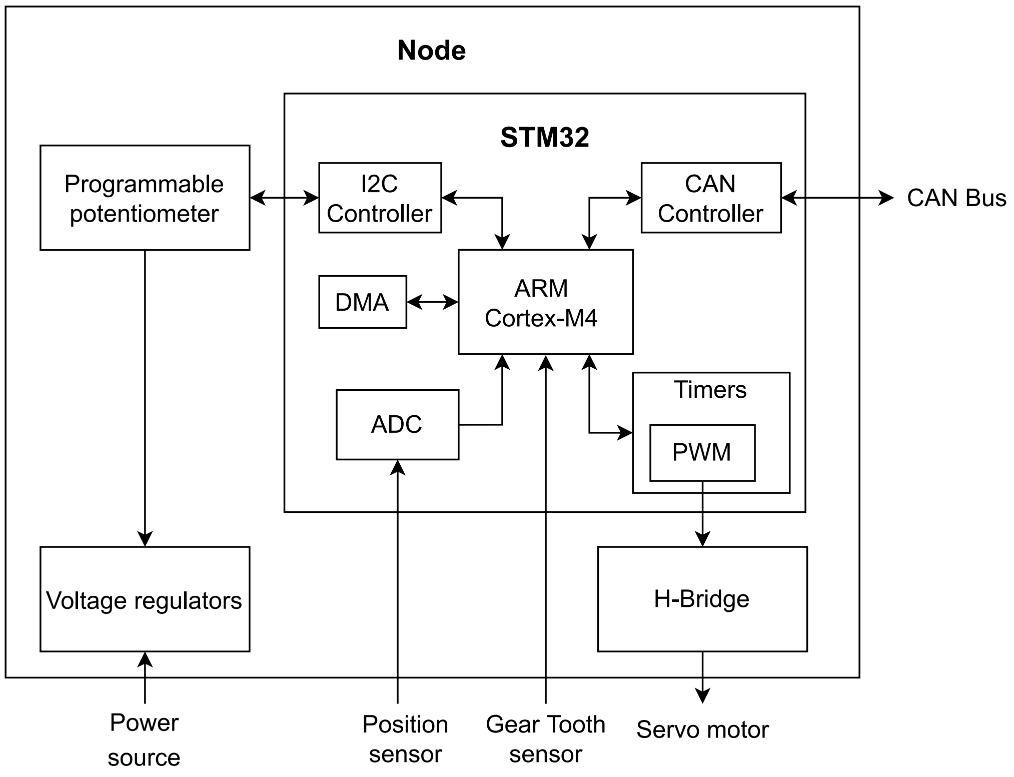

Fig. 2.4: Block diagram of a brake node in the devKit car. See sections 2.6 and 2.8 for I2C and DMA.

2.1.2 Brake control system

The brake control system consists of two main feedback loops. One feedback loop controls the brake torque using the speed of the car, the wheel, and the desired

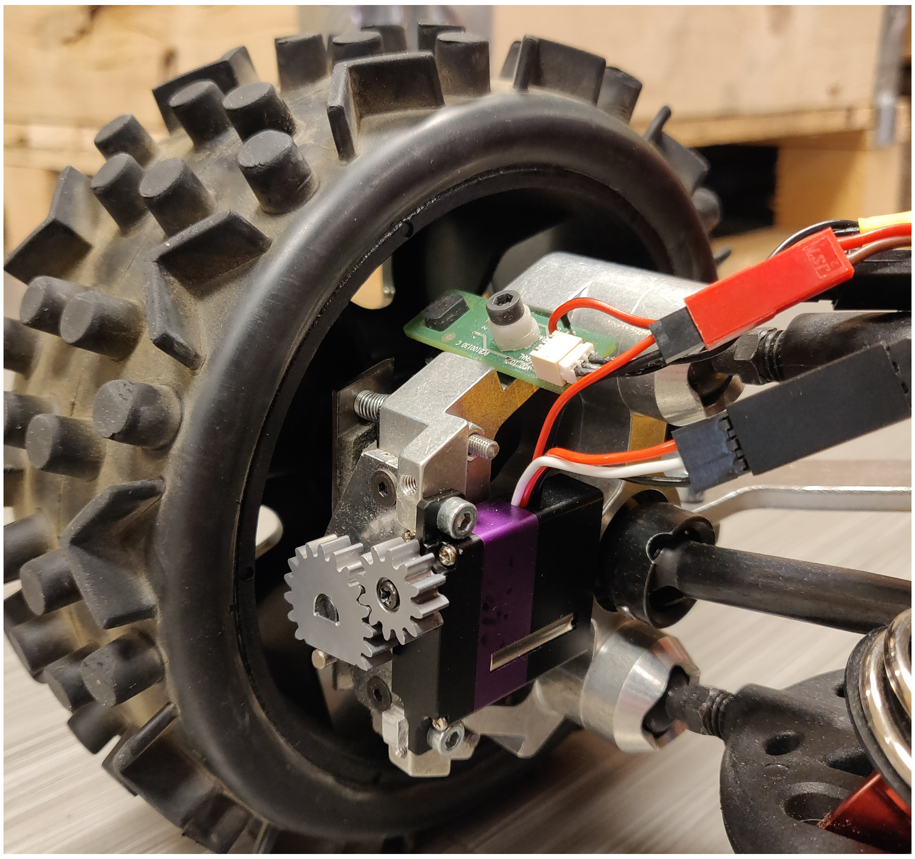

Fig. 2.5: Picture of the servo (black and purple) and gear tooth sensor (green PCB) mounted at a rear wheel.

braking. The difference in speed between the car and the braking wheel is used to determine if the brake locks the wheel and if so, the torque is reduced. The other feedback loop controls the motor position with the desired brake torque and the measured servo motor position. Calibration is necessary at every start-up to get the correct brake servo position.

2.2 Controller area network

CAN is a communication protocol first published in 1986 by Robert Bosch GmbH [8]. It describes a way for multiple nodes to access a common bus that supports distributed real-time control [9]. The CAN protocol has a maximum transfer speed of 1 Mbit/s. Communication is done through CAN messages which consist of an identifier (ID), a control field, and a data payload.

The ID for a CAN message must be unique in the sense that an ID can only be used by one node but a node can use multiple IDs. The ID serves three purposes: (1) it identifies the message, (2) it allows prioritization between messages when transmitting simultaneously and (3) allows nodes to filter out messages not intended for them. A message with a lower ID value will be prioritized over a message with a higher ID value through an arbitration process that is built into the protocol.

2.3 CAN Kingdom

CAN Kingdom is a set of protocol primitives (sometimes called a meta protocol) that enables designing a higher-level protocol for setting up a distributed control system using CAN [4]. The protocol separates the roles of the system designer and node designer. Node configuration is built into the protocol which enables the separation. This gives more freedom at the system level and promotes more general, system-independent, nodes. At least two operation modes are required to facilitate the node configuration: a start-up mode and one or more run-time modes.

The protocol places heavy emphasis on similes to aid in system comprehension. The CAN Kingdom-specific terms are marked with a capital letter in the rest of this thesis. For the reader's convenience, a short glossary is provided at the end of this section.

2.3.1 Protocol structure

The protocol divides the nodes in a DECS into sensor or actuator nodes, called Cities, and a master node, called King. These together with the CAN bus, called Postal System, make up a Kingdom. Every City has a Mayor that specifies what data it needs and what data it can transmit. The King is responsible for defining what information each City should send and when, as well as what information it should listen for so that every City can fulfill its function in the system.

The information on the bus is in the form of CAN messages, referred to as Letters, with a CAN identifier (Envelope) and a payload (Page). A Page consists of at most eight Lines with each Line being represented by one byte in the data payload. For a receiver to correctly interpret a Page it needs to be decoded. The tool for this is called a Form which says what the data on each Line represents.

When transmitting more than eight Lines multiple Letters are needed since each Letter can only carry one Page. These Letters have the same Envelope containing a Page where one or more Lines (or at least some bits of a Line) are dedicated as indexation. To interpret an Envelope with multiple Pages a set of Forms is needed (one per Page) which constitutes a Document.

To receive and interpret a Letter, its Envelope needs to be matched with a Document. This matching is done through a structure called Folder. Any Document for interpreting received Pages needs to be placed in a Folder. An Envelope is then assigned to the Folder. Now when a Letter is received it gets matched with a Document based on its Envelope and can be correctly understood. Similarly, when transmitting a Letter a Document is used to write the Pages before sending it out on the Postal System with the assigned Envelope.

2.3.2 Operation modes

The system operation is divided into at least two modes: a start-up mode and one or more run-time modes. During start-up, the King configures Cities and sets up the information exchange in the Kingdom. When the setup is done the King changes the system to a run-time mode. After setup, the King will generally be dormant as the system is now set up and running as designed. Although the King can still configure Cities or change to a different operating mode while the system is in run-time mode. This enables an implementation where the King has an active role where it can, for example, set the system in a safe mode if it detects errors.

The setup is done through a set of Pages called the King's Pages which are sent by the King. Line 0 in the King's Pages is used for the Page index which means that there are 256 possible King's Pages. The first 32 King's Pages are reserved for general system setup in the protocol (although not all are defined) and the last 128 King's Pages can be defined freely for the specific implementation. The equivalent to King's Pages for Cities are Mayor's Pages which are used for City identification at the King's request.

The start-up phase consists of two parts, initiation and configuration, that take place before the system switches to run-time mode. The initialization of each City is done by sending King's Page 1, which contains the Base Number for the Kingdom and requests the City's identification. The Base Number is used to derive the Envelope for the Mayor's Pages which contain the City identification. The identification can be either the EAN (European Article Number), a serial number, or another custom identification defined for the City.

The configuration of the Cities is needed to set up communication and to configure any City-specific parameters. The communication is established through several King's Pages, most importantly Page 2 and 16. Page 2 assigns Envelopes to a Folder in a City which is sufficient if the City has predefined Folders. Configuring Folders is done through Page 16 which assigns a Document to a Folder and sets that Folder as either receive or transmit. City-specific configuration is done through King's Pages 128-255 and can include for example setting sampling frequency or control parameters. Finally, when everything has been configured Page 0 is sent out, which terminates the start-up phase.

2.3.3 Glossary

A short glossary is enclosed, see Table 2.1, briefly describing the core CAN Kingdom terminology. It has been adapted from the glossary in the CAN Kingdom specification [4].

Table 2.1: Glossary describing the CAN Kingdom terminology appearing in this thesis

Term	Description
Page	CAN data field
Envelope	CAN identifier
Letter	Complete CAN message
Line	One byte of the message payload in a Page
Form	Tool to encode/decode a Page
Document	A set of Forms
Folder	Link between Envelope and Document
Postal System	CAN Bus
City	CAN node
Mayor	Anthropomorphism of the City Code
Mayor's Pages	Pages used to identify City during start-up phase
Capital	CAN master node
King	The code in the master node
King's Pages	Pages used for system setup
Base Number	An offset to Envelop of Cities Mayor's Pages
Kingdom	CAN system

2.4 Operating system

An OS is an abstraction layer above the hardware that hides the details of the complicated hardware interfaces, providing a simpler view for an application programmer [10]. The OS also manages the hardware resources, such as processors, memory, and hard disks. Another function provided by most OSes is multitasking where multiple processes (referred to as tasks in real-time systems) can be active at the same time.

2.4.1 Real-time operating systems

In a real-time system, the correctness of the task execution is dependent on the correct result being produced within a deadline [11]. A regular OS does not provide control over task executions based on deadlines and is therefore unsuitable for real-time systems. A real-time operating system (RTOS), however, provides the time-based task control a real-time system requires, while also offering the abstractions of an OS.

The abstractions provided by both OSes and RTOSes come at the cost of some overhead. In an RTOS the overhead can for example come in the form of context switches when changing which task to execute. In many cases, the overhead can be a significant factor for the feasibility of a system [12]. If the overhead is not considered, a schedulability analysis of an infeasible system could incorrectly show it to be feasible. This would be detrimental to a hard real-time system, where one or more tasks must meet their deadlines to guarantee correct system behavior.

Some work has been done to reduce RTOS overhead. One example is [13] where a CPU and a hardware kernel were implemented on an FPGA to speed up costly operations, such as context switching. With this approach, a context switch could be achieved in only one clock cycle [13] which can be compared to 84 clock cycles in FreeRTOS [14] (under test conditions listed in the source). Another example is HartOS [15] where the operating system is designed for use on a Microblaze processor to integrate even more features in the FPGA hardware to further reduce overhead.

Another important aspect to consider in an RTOS with periodic tasks is timing accuracy. Lower timing accuracy results in larger deviations from the expected period, known as jitter, which have a negative impact on control systems [16]. Jitter characteristics can vary depending on the real-time implementation which affect control performance. For instance, jitter caused by inserting a constant time delay, such as from an interrupt, would be modeled differently from jitter found in network delay which follows a Poisson distribution [17].

An example of varying jitter in RTOSes can be seen when comparing FreeRTOS [18] and the TinyTimber RTOS [19]. TinyTimber utilizes a free running timer to keep track of time, as opposed to FreeRTOS which uses a system tick updated via interrupts. Timing accuracy measurements for these RTOSes show a jitter of only 20 µs for TinyTimber compared to 1 ms for FreeRTOS [19]. This shows that system implementation can have a large impact on jitter.

2.4.2 RT-Kernel

RT-Kernel is an RTOS developed by the company RT-labs [7]. The OS provides timers, task management, and basic synchronization primitives such as mutexes and semaphores [20]. Tasks in RT-Kernel are scheduled using preemptive priority-based scheduling and priority inheritance [21] is used to avoid priority inversion. In addition to its real-time capabilities, RT-Kernel also supports several communication protocols, file systems, and I/Os.

Tasks in RT-Kernel, as in most OSes, are in one of four states: ready, running, waiting, or dead. Starting a task sets its state to ready and places it in a queue of all tasks in the ready state, according to priority. The task with the highest priority in the ready queue enters into the running state and starts executing. A task is put in the waiting state if it is waiting for a resource or if it is stopped and needs to wait for a restart. When a task finishes it enters into the dead state. Tasks in the dead state will signal a reaper task that will free the memory.

There is a system in RT-Kernel for implementing periodic tasks called time-triggered tasks. Time-triggered tasks need static schedules that are defined at compile time. The schedules can be changed during run-time but they cannot be reconfigured which is a requirement for use with the CAN Kingdom protocol. Reconfigurable periodical tasks can instead be realized by using timer callback functions to start a task at a fixed interval.

2.4.3 Bare-metal

The concept of bare-metal is used in this thesis to refer to code that executes without an underlying OS. This approach has the benefit of reducing overhead caused by the OS. Another benefit is increased control over code execution as no OS processes will be running in the background. For small systems, this gives more opportunities to tailor the application to the system needs, while larger systems might be too complex to manage without an OS. This difficulty in managing larger systems is a major drawback of a bare-metal solution as the developer cannot leave any implementation details to the OS. Another aspect to consider is the reduced portability as code needs to be customized to the target platform.

2.5 Scheduling

Scheduling is necessary for a real-time system to ensure that all tasks meet their deadlines. There are two main types of scheduling methods: online and offline [22]. Online schedulers choose the task to execute, among the tasks that are ready to run, based on task priority. The priority can either be static (determined before run-time) or dynamic (determined during run-time). An offline scheduler has a radically different approach, as it is essentially a timetable storing a sequence of tasks and their WCET created beforehand [22]. The tasks are then executed in order according to the table, with any remaining time to WCET resulting in processor idle time.

With online schedulers, there are many algorithms available for assigning priorities to the tasks. There are several algorithms to assign static priorities that are optimal for different types of task sets e.g. rate monotonic [23] and deadline monotonic [11]. For dynamic priority assignment, earliest deadline first (EDF) is commonly used, since it is optimal for most task sets [23].

Determining if a set of tasks is schedulable requires a feasibility analysis. With an online scheduler and a general task set, this is a computationally complex problem [24]. With an off-line scheduler, however, proving the feasibility of a schedule is trivial. If the task table can be cyclically executed, then the task set is schedulable. However, general aperiodic tasks can not be scheduled offline since they occur with an irregular period. If a task set contains aperiodic tasks then online scheduling is often necessary.

2.6 I2C

I2C is a serial communication protocol invented in 1982 that is often used in embedded systems to interface with peripherals [25]. Data transfer is done over two wires, the serial data line (SDA) and the serial clock line (SCL) which carry the data and clock respectively. It uses a master-slave setup with addresses for bidirectional transfer over different speeds ranging from 100 kbit/s to 3.4 Mbit/s.

2.7 Pulse-Width Modulation

Pulse-width modulation (PWM) is a method for controlling the output power without changing the supply voltage by modulating the pulse width of a square wave signal. PWM is widely used in embedded applications to control e.g. servo position, motor power, or speaker volume. A PWM signal has two main properties: frequency and duty cycle. Frequency is application dependent and is generally fixed during operation while the duty cycle is varied.

2.8 Direct memory access

Direct memory access (DMA) allows data to be moved to or from memory, or peripherals by a DMA controller, without involving the CPU in the transfer [10]. The DMA transfer is initiated by the CPU configuring the DMA controller after which the DMA controller will handle the bus transfer and handshaking until the transfer is completed. Using DMA frees up the CPU as it does not have to constantly poll peripherals or handle interrupts to facilitate the transfer. However, it is possible for the DMA to congest the bus, negatively impacting bus access for the CPU.

3 Methods

This chapter details design decisions made related to device driver choices and methods used for testing the brake.

3.1 Device drivers

In developing the bare-metal implementation for the brake module, a few design decisions have to be made. A given design decision is to use the CAN Kingdom protocol as the brake module should be a part of a system using that protocol. The protocol implementation is, however, heavily tied to RT-Kernel and relies on several OS features which need to be adapted to work in a bare-metal setting.

One of the features RT-Kernel provides is driver support for peripherals, including CAN, ADC, PWM, I2C and external flash memory. This means that a bare-metal system needs to provide these drivers in another way. One option is to use ST Microelectronics hardware abstraction layer (HAL) library [26] which provides drivers with general abstraction for most STM32 processors and peripherals. This makes the application easier to port to other processors, and in the accompanying STM32Cube IDE [27] there are built-in code generators for setting up many of the peripherals.

A downside of using the STM HAL library is the loss of detailed control that comes with high compatibility, potentially resulting in a less efficient solution. Therefore, STM32 peripheral drivers [28] were selected for our device driver implementation which allows for more control and optimization at the cost of lower portability. Another downside of using the STM32 peripheral drivers is that a more detailed setup of peripherals is required.

3.2 Test methods

Testing of the brake node is done using a minimal test setup consisting of a brake node, a brake servo, a King node, and an Arduino UNO R3 [29] (see Fig. 3.1). For the brake node to operate it needs the remote controller throttle data, the wheel speed from all wheels, the gear tooth sensor signal, and the position of the disc brake servo. The King is used to configure and set up what data the brake node should send and receive. After the setup, the King is used to send CAN messages that imitate the remote controller and the other brake nodes. The Arduino is used to generate signals to mimic the gear tooth sensor. A realistic servo position and

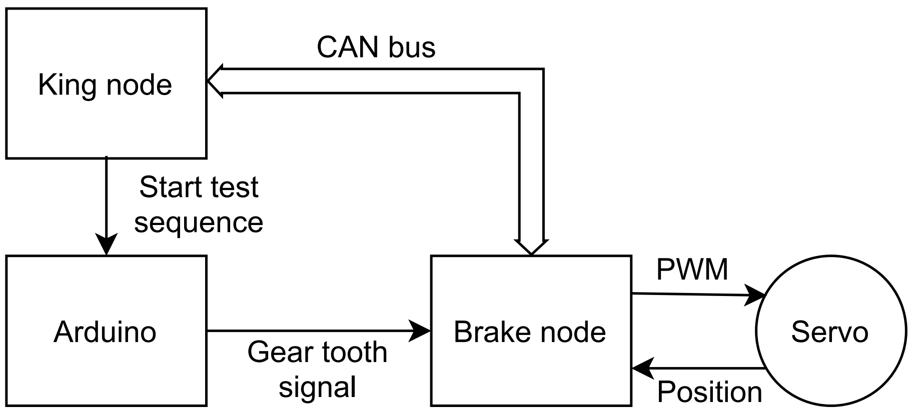

Fig. 3.1: An illustration of the test setup. The King uses the CAN bus to set up the brake and to imitate the remote controller and other brake nodes. The King also signals the Arduino to start the test sequence. When the test is started the Arduino sends the gear tooth signal to the brake which, combined with the CAN bus data and servo position, is used to control the servo.

control are obtained by mounting the servo in the disc brake used on the car. The CAN messages and Arduino signals in this test follow a predefined speed and throttle profile (see Fig. 3.2) to create a repeatable test sequence. The same test setup is used to measure the task execution time and jitter for both the RT-Kernel and bare-metal implementations as well as for functional verification of the full system software.

In addition to the test setup used for the execution time and jitter measurements, each individual device driver had to be tested. The functional verification of the device drivers had various test setups depending on the device driver that was tested.

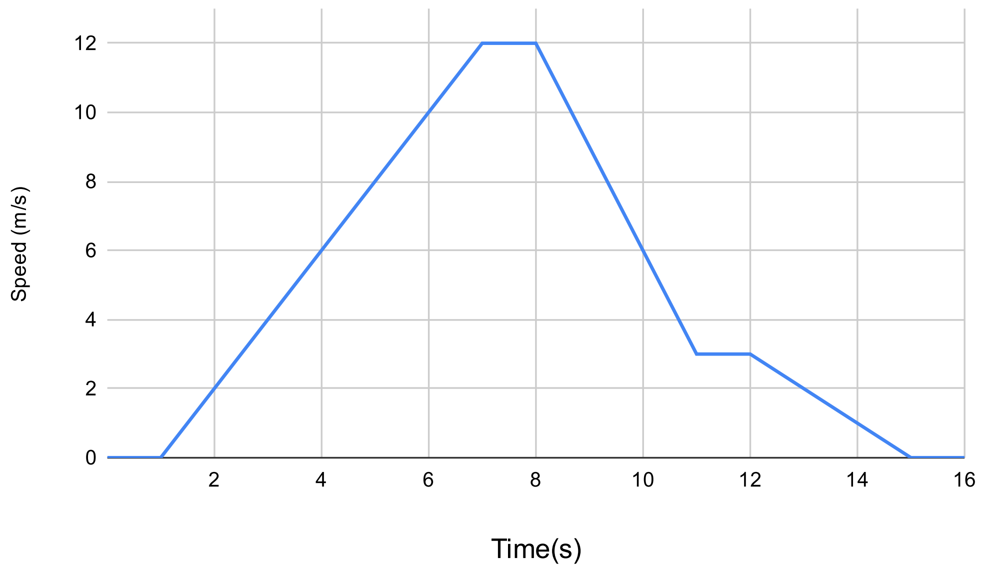

Fig. 3.2: The speed profile used to generate CAN data and gear tooth pulses.

3.2.1 Time measurements

There are a few different methods for measuring the execution time of the tasks. One is to use the Arduino to measure execution time by configuring the brake node to generate signals at the start and end points of the task and measuring the time between these points using the Arduino clock. Another method is to set up a dedicated hardware timer on the STM32 processor which is started at the beginning of the code block and checked at the end. The time difference can then be sent via the CAN bus to a computer to log the data. For both these methods, the jitter can be measured by time stamping the execution time measurements.

The measuring approach with the Arduino was originally used for two main reasons: first, it is easy to set up and gather test data, and second, it gives the possibility to measure execution time while making minimal changes to the existing code. An issue with this approach is that the Arduino only has a 4 µs resolution [29, 30] which with execution times of between 20 and 60 µs would make it almost impossible to detect jitter in task execution unless it is very large. There might also be an issue with the Arduino not being fast enough to catch all interrupts. For these reasons, this approach was abandoned.

A way to get better precision of the time measurements is to use the built-in hardware timer of the STM32 processor as this has a tunable resolution down to one clock period which is 14 ns at the maximum frequency of 72 MHz. This does not require any external measurement equipment but it does require additional code to use the timer which increases the task execution time. The time increase is however minor and similar in both the RT-Kernel and bare-metal implementation. Therefore, hardware timers are used for measuring jitter and execution times.

3.2.2 Overhead measurements for RT-Kernel and bare-metal

An important factor to consider and include in the measurement of task execution time is the overhead associated with the task. To execute a task in RT-Kernel a context switch is required (even if no other task is executing a switch from the idle task is still required) which adds overhead and increases the task completion time. To account for the overhead the timer is started just before the task is placed in the ready queue.

To ensure a fair comparison between the implementations, the overhead of the bare-metal solution must be included in the execution measurement. This is achieved by starting the timer at the beginning of a new time slot. In both implementations, the time is measured at the end of the task.

An argument could be made that there is additional overhead when returning from a task in the RT-Kernel case. How you define what part of the total execution time of the RT-Kernel code should be associated with a specific task is not obvious and it is also application dependent. We choose to only include the initial context switch to give a minimum overhead measurement for the RT-Kernel implementation.

3.2.3 Jitter measurements

The jitter of the tasks is measured at the start of the tasks. Variations in execution time will be visible in the execution time measurements but these measurements will not show variation in task starting time due to the scheduling. The measurements are made by taking a time stamp from the hardware timer at each start of the task. The period of the task is obtained from the difference between consecutive time stamps. The jitter is then defined as the absolute value of the difference between the desired period and the measured period.

3.2.4 Device driver verification

Functional verification of device drivers was done through a few different test setups. CAN communication was observed to be working correctly through monitoring of the CAN bus with Kvaser's leaf light v2 [31] interface and Kvaser CANKing [32] software. I2C and PWM drivers were tested using an oscilloscope to observe the signals on the corresponding output where timing and duty cycles were measured. The ADC driver was tested using a potentiometer to vary the input voltage and send the measured value of the ADC conversion via the CAN bus for comparison.

4 Implementation

In this chapter, the implementation will be described. First, we discuss the implementation of the different drivers needed for the system. Then we discuss the adaptations made to the CAN Kingdom stack. Finally, we discuss the scheduling implementation of the system tasks.

4.1 Driver implementation

With the help of the STM32 peripheral drivers, the vital functions for the brake to operate were realized in the bare-metal solution. This includes support for sending and receiving messages with CAN, using I2C to set a potentiometer for voltage regulation, PWM for controlling the brake disc servo and, ADC for reading servo position. Both I2C and ADC were implemented with DMA to avoid unnecessary CPU usage.

4.2 CAN Kingdom

The CAN Kingdom implementation consists of two parts: one part for handling messages and one part, a HAL[1], for interfacing with peripherals. The message handling procedure is the same as in RT-Kernel, as they follow the same protocol, and only needed a few minor modifications to work with the bare-metal driver implementation. The HAL was converted to use the new drivers since RT-Kernel uses a Unix-like driver system where peripherals are addressed using file descriptors which would be unnecessarily complex to reuse. While the basic functionality was retained, the current implementation does not support storing City and Folder configurations on the external flash memory, as was possible in the RT-Kernel implementation. This functionality was not implemented as it is not utilized and would need driver support for external flash. Loading configurations now only returns the default reset values. If flash drivers were to be developed this functionality could be implemented in the bare-metal implementation as well.

The received CAN messages are handled by the main CAN Kingdom function which can either be polled to check for new messages or called through an interrupt. If a received message is a King's Letter, it is processed immediately by the protocol, otherwise, an application-defined handler function is called. These handler functions

[1]Note: This HAL does not have any relation to the STM HAL library

will store the received data in a common system struct to make it available to other parts of the application. If the message requires additional processing, a flag is set to trigger a task to process the message data.

4.3 Task management

The task management of the bare-metal implementation consists of a static schedule with time slots and a scheduler. The schedule can be configured during the set-up phase by sending King's Pages to configure the period and offset of the periodic tasks. The scheduler keeps track of when each task should be executed during the run-time phase.

The tasks are adapted from the RT-Kernel implementation with only slight modifications. In RT-Kernel the tasks consist of an initialization and an execution loop where the initialized variables are stored on the task stack between executions. However to be able to share data between tasks most variables are stored in a shared struct. The execution loop executes once by having a taskStop function call at the beginning of the loop. When the task is started again, it will resume execution after the taskStop call. See Listing 4.1 for a pseudocode example.

The RT-Kernel task was converted to the implementation seen in Listing 4.2 where task execution is done by calling the corresponding task function. The bare-metal task implementation has no stack, but instead variables are stored in the shared struct. The initialization is now performed once every task execution. The only modifications made to the task code were switching to our bare-metal device drivers.

Listing 4.1: Pseudocode of the RT-Kernel task implementation

```
taskFunc()
    Initialization
    while(true){
        taskStop();

        Task Code
    }
```

Listing 4.2: Pseudocode of the Bare-metal task implementation

```
taskFunc()
    Initialization
    Task Code
```

The schedule is implemented using an array of function pointers to tasks where each entry corresponds to one time slot. The length of the schedule, which can be configured by the King, is equal to the number of time slots and need to be divisible by the period of each scheduled task. A task must finish within its time slot which has a default length of 1 ms. In the brake node implementation, there are two scheduled tasks: CALCSpeed, which calculates the wheel speed with a period of 10 ms, and ABS, which controls the brakes with a period of 5 ms. The offsets vary between the different brake nodes.

In this schedule, it is possible to schedule the processing of incoming periodic CAN

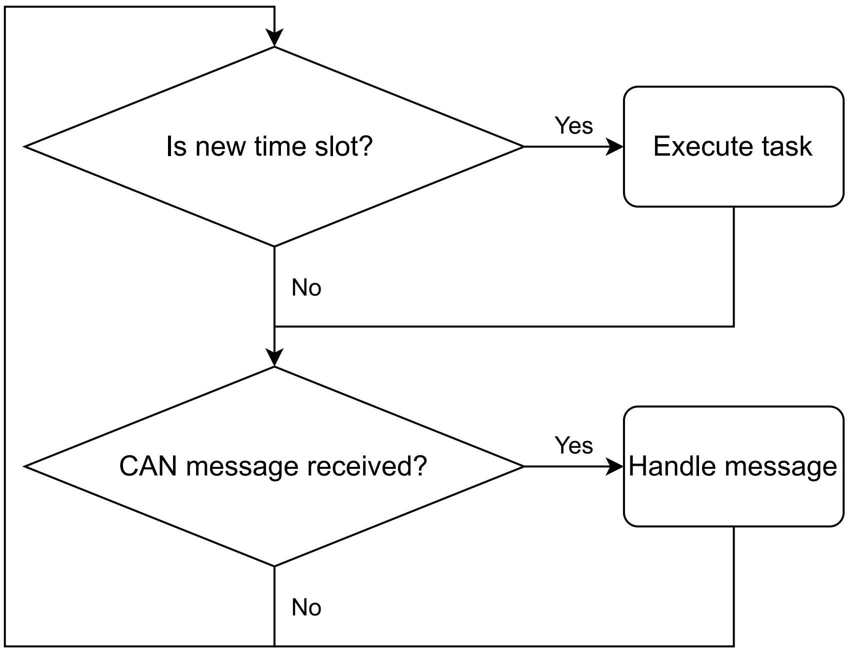

Fig. 4.1: Program flow of the polling-based scheduler.

messages similarly to other tasks. However, it may be difficult to predict when these messages arrive since there is no time synchronization. Additionally, if each message should get its own time slot, it would require more King's Pages to set up and may lead to scheduling conflicts. Therefore each empty schedule slot is assigned by default to process all incoming messages. This requires intentionally leaving free slots unless all message processing is explicitly scheduled.

The scheduling has two main parts: handling incoming CAN messages by calling the main CAN Kingdom function and starting scheduled tasks. Both of these could be accomplished either by polling repeatedly to check if a new time slot or CAN message has arrived or using an interrupt as a trigger. We have therefore developed two schedulers, one with interrupts and one without.

4.3.1 Polling-based scheduler

The polling-based scheduler does not use any interrupts, but instead continuously check if there is a new time slot or a new CAN message. The program flow of this scheduling approach can be seen in Fig. 4.1. The first step of the polling loop is to check the timer if a new time slot has arrived. If this is the case the task is fetched from the schedule and executed. When the task is done, or if there was no new time slot, the CAN Kingdom protocol is called to check for new messages. If a new message is received, it is handled and a flag is set for messages that require a separate task to handle. These tasks are then executed in the next free time slot.

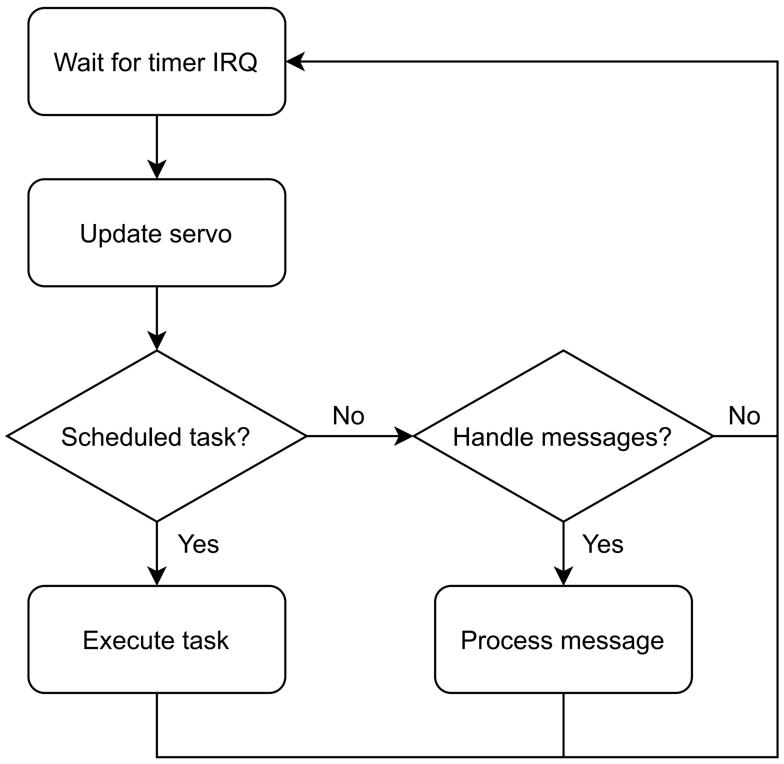

Fig. 4.2: Program flow of the interrupt-based scheduler.

4.3.2 Interrupt-based scheduler

Polling to check if there is a new time slot can be inaccurate and could cause jitter. One way to get very accurate time slots is to use a hardware timer to generate an interrupt to signify a new slot. Interrupts could also be used to indicate when a CAN message has arrived, removing the need to repeatedly check if a new CAN message was received.

Interrupts that appear during tasks do, however, cause varying execution times for those tasks which lead to jitter. To combat this the calculation part and the actuation part of a task have been split. At the beginning of every time slot, the servo is updated with the data calculated during the latest task execution. This guarantees accurate control of the servo at the cost of some additional overhead. This results in the program flow that can be seen in Fig. 4.2.

The interrupts are handled through the STM32's Nested Vectored Interrupt Controller (NVIC). The NVIC is configured to have four priority levels but only three of them are currently in use. The interrupt for time slots has the highest priority to give as low jitter as possible to the servo actuation. The second highest priority is used for interrupts generated by the gear tooth sensor. The lowest priority is assigned to interrupts from received CAN messages since these have a buffer that can handle a delay.

5
Results

A bare-metal implementation of the brake node has been created and verified to work together with the rest of the nodes in the devKit car. This chapter presents the results from execution time and jitter measurements as well as an evaluation of the system integration.

5.1 Execution time

The execution time of the ABS task varies depending on the speed of the car and internal flags that are part of the brake control loop. Generally, tasks that fluctuate in execution time are undesirable in real-time systems and control settings. This behavior has been kept to provide a fair comparison between the bare-metal and RT-Kernel solutions.

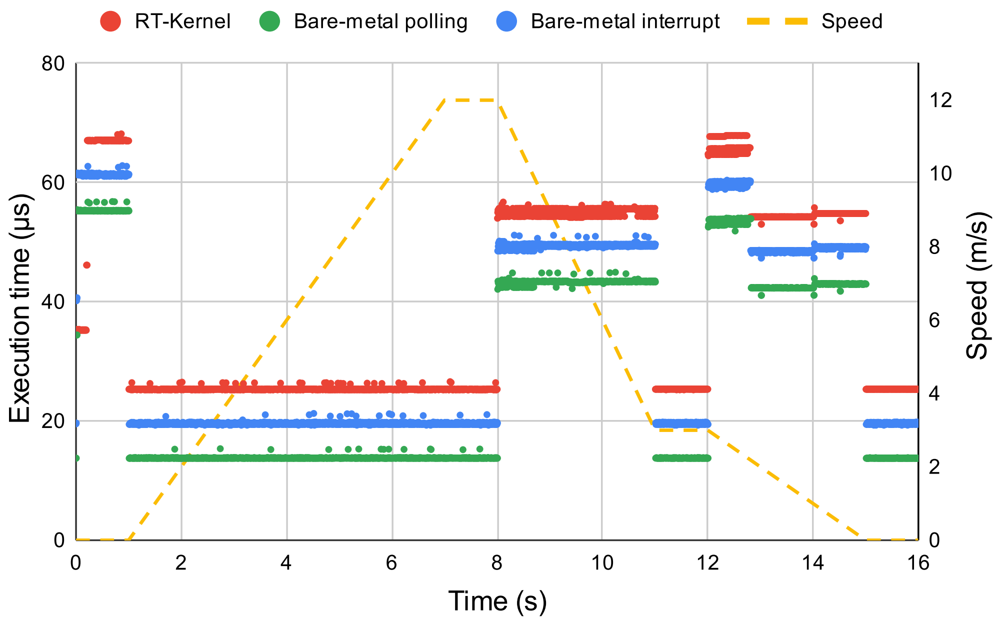

Fig. 5.1: Execution time of the ABS task for RT-Kernel and Bare-metal application with both polling and interrupt-based schedulers along with speed data sent to the systems. Overhead is included in measurements.

A comparison of the execution times, including overhead, between the different solutions for the ABS task can be seen in Fig. 5.1. The simulated car speed is also included to demonstrate that it impacts the task completion time. The bare-metal implementation with interrupts is on average 5.4 µs faster than the RT-Kernel implementation.

The polling-based approach is further 5.9 µs faster on average than the interrupt-based scheduler. This is due to the update servo function adding overhead for the scheduling with interrupts. If the overhead is excluded then both bare-metal schedulers have very similar execution times, as seen in Fig. 5.2. However one might note that the interrupt-based scheduler, without overhead, is slightly (1.3 µs) faster due to the driver calls being moved to the update servo routine. We can also see that the RT-Kernel implementation still has on average 6 µs slower execution times.

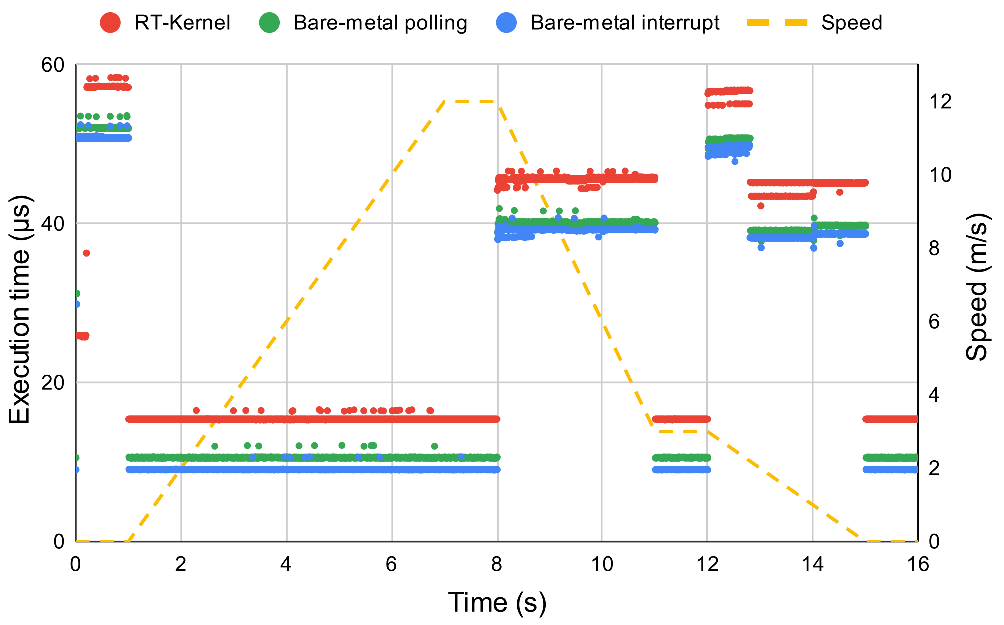

Fig. 5.2: Execution time of the ABS task for RT-Kernel and Bare-metal application with both polling and interrupt-based schedulers along with speed data sent to the systems. Initial task overhead is not included.

The execution time of the CALCSpeed task, see Fig. 5.3, is not dependent on the input data from the Arduino or CAN bus as the same calculations are done regardless. Similar to the ABS task, the polling approach is slightly faster than the interrupt-based one due to the update servo function. The varying execution time for the RT-Kernel implementation is noteworthy.

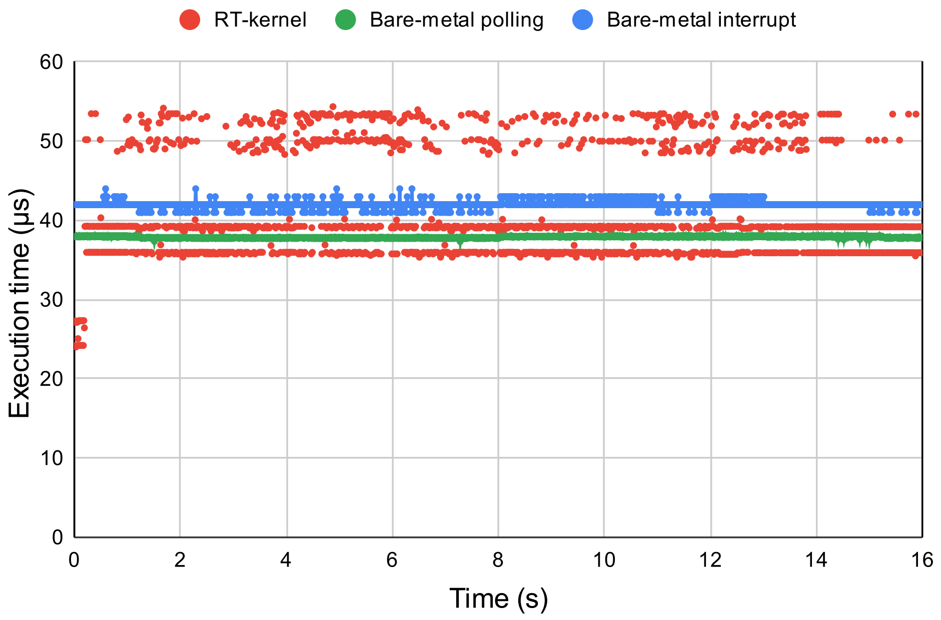

Fig. 5.3: Execution time of the CALCSpeed task for RT-Kernel and Bare-metal application with both polling and interrupt-based schedulers along with speed data sent to the systems. Overhead is included in measurements.

5.2 Jitter measurements

Jitter measurements were performed on the CALCSpeed and ABS tasks in the RT-Kernel implementation and the bare-metal implementation (using both polling and interrupt-based schedulers). From Tables 5.1 and 5.2 it is clear that the bare-metal interrupt-based scheduler had the least jitter with under 1 µs maximum jitter and an average of at most 0.25 µs (about 18 clock cycles). Both bare-metal schedulers showed very similar jitter for both tasks. RT-Kernel however had an average jitter that varied greatly from 0.25 µs to 4.57 µs.

An additional jitter measurement was taken for the update servo function used in the interrupt-based scheduler. The measurement presented in Table 5.3 shows maximum jitter of 0.28 µs which is equal to 20 clock cycles. The average jitter is less than seven clock cycles.

5.3 System integration

The bare-metal brake nodes were tested in the devKit car together with the other system nodes that are based on RT-Kernel. The system was also tested using different combinations of RT-Kernel brake nodes and bare-metal brake nodes. The system performed according to expectations, that is, it shows the same behavior as when only RT-Kernel nodes were used, for all the tested combinations of nodes.

Table 5.1: Maximum and minimum period and the average jitter of the CALC-Speed task

Implementation	Max period [µs]	Min period [µs]	Average jitter [µs]
RT-Kernel	10002.22	9997.97	0.25
Bare-metal polling	10004.89	9995.11	1.67
Bare-metal interrupt	10000.69	9999.28	0.21

Table 5.2: Maximum and minimum period and the average jitter of the ABS task

Implementation	Max period [µs]	Min period [µs]	Average jitter [µs]
RT-Kernel	5006.53	4993.44	4.57
Bare-metal polling	5005.00	4995.19	1.60
Bare-metal interrupt	5000.72	4999.28	0.25

Table 5.3: Maximum and minimum period and the average jitter of servo update on the bare-metal implementation using the interrupt-based scheduler

Max period [µs]	Min period [µs]	Average jitter [µs]
1000.28	999.72	0.093

6 Discussion

This chapter discusses the previously presented results and comments on the choice of device driver and scheduling implementation. There are also some reflections on using RT-Kernel compared to bare-metal as well as ethical considerations for this thesis work.

6.1 Device drivers

The device drivers used for the bare-metal implementation are developed specifically for this application. This gives much better performance compared to the device driver implementation in RT-Kernel where each driver call requires a lookup of a file descriptor similarly to a Linux driver. This can especially be seen in Fig. 5.2 where the bare-metal implementations have a significantly shorter execution time while the code segments are almost identical, apart from the device drivers. The constant decrease in execution time is expected since the number of driver calls is constant, even though the total task execution time varies.

6.2 Task implementation

The only modifications to the task code were switching drivers. This was done to provide a fair comparison between the RT-Kernel and bare-metal implementations. One could, however, argue that there are a few performance improvements that could be made to the bare-metal implementation without affecting the fairness of the comparison. One example is to initialize task variables separately to avoid constant re-initialization. It is possible that removing the re-initialization in the CALCSpeed task could result in a significant reduction in execution time for the bare-metal implementation, since the initialization contains a floating-point calculation. This could bring the execution time closer to the shortest execution times for RT-Kernel.

6.3 Scheduling

The jitter observed from the polling-based scheduler is due to alternately checking if there is a new time slot and if a CAN message was received. If the check for a new time slot just misses the event of a new slot, then the next slot could be delayed by the time it takes to handle a CAN message. Even if no CAN message was received it will still result in a small delay to the start of the task execution. A solution to this is to not perform the check for CAN messages when the start of a new time slot is within a certain time frame, for example, 10 µs, to make sure the task will always start on time. This would however mean introducing additional overhead which would offset most execution time improvements from moving to bare-metal.

The interrupt-based approach does come with the drawback of interrupts affecting the completion time for tasks. This was remedied by separating computation and actuation, as is common practice in control systems. Instead of having a task directly controlling the servo, the output values are stored to be able to actuate at the start of every time slot. This approach also addresses the jitter in servo actuation caused by the variation in execution time of the control logic.

Breaking out the updating of the servo adds a small overhead of about 5 µs but in return gives negligible actuation time jitter. Another benefit is that the execution time of the ABS task is slightly reduced as all driver calls are now made in the update servo function. This small increase in overhead could therefore be acceptable if very low jitter is required. The overhead caused by updating the servo could be reduced by only calling the update servo function when an update is required and not in every time slot. This could result in jitter in the starting time of some tasks (depending on where servo updates are triggered in the scheduled) but since jitter-sensitive operations should be made in the update servo function, this should not be a problem.

Unfortunately, most of the improvements observed in the bare-metal implementation have little real impact on the system. The processor in the nodes is much faster than required with the scheduled tasks using less than 3% of the processor time. It is possible that there could be larger performance gains for a node with a higher processor load since not all processor time used by the RTOS is measured when comparing the tasks.

6.4 RT-Kernel

During testing, we found a few interesting issues that affected the real-time performance of the RT-Kernel application. For instance, changes in jitter could be observed due to very small changes to the execution time of unrelated code sections. Additionally, the large variations in execution time found in the CALCSpeed task are hard to explain as this task only has one code path. It might be possible to explain this jitter with interrupts but it is unlikely that this would only be present in the CALCSpeed task of the RT-Kernel implementation.

It is difficult to know the exact reason for these behaviors due to the complexity of the RTOS. These types of issues would be easier to debug in the bare-metal application as the program flow is easier to follow, thereby simplifying program analysis. This is another reason why a bare-metal implementation might be advantageous for some applications.

6.5 Ethical considerations

This project is about the CAN system in a model car. It is unlikely that our bare-metal solution will be the basis of a system in a real car, or in another industrial setting. However, the intended use case of our code includes safety-critical systems where failure to meet deadlines could give potentially catastrophic consequences. It is therefore important that any claim about system performance is backed by rigorous testing to guarantee the accuracy and that any uncertainties are clearly stated.

7
Conclusion

This thesis work set out to create a bare-metal implementation of a brake node, evaluate its real-time performance and assess the impacts of combining different types of nodes in a DECS. This work has shown that it is feasible to develop a real-time bare-metal implementation from an RTOS application. This resulted in lower, more predictable, jitter and shorter execution time due to reduced overhead. It was also found that integrating bare-metal and RT-Kernel nodes in a system using the CAN Kingdom protocol was possible with no issues.

The key advantages of a bare-metal solution compared to an RTOS implementation are reduced overhead and jitter, increased control over low-level details, and simplified program analysis. The main drawbacks are worse portability and the lack of ready-made drivers. Moving from an RTOS to a bare-metal solution could be a worthwhile endeavor for applications where extremely low jitter is required or hardware limitations constrain system performance.

Bibliography

[1] M. Nahas, M. J. Pont, and M. Short, “Reducing message-length variations in resource-constrained embedded systems implemented using the Controller Area Network (CAN) protocol,” *Journal of Systems Architecture*, vol. 55, no. 5, pp. 344–354, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762109000368

[2] K. Zuberi and K. Shin, “Non-preemptive scheduling of messages on controller area network for real-time control applications,” in *Proceedings Real-Time Technology and Applications Symposium*, 1995, pp. 240–249.

[3] Kvaser AB. Kvaser About us. Accessed on: April 4, 2022. [Online]. Available: https://www.kvaser.com/

[4] ——. *CanKingdom*. Accessed on: Jan 10, 2022. [Online]. Available: https://www.kvaser.com/about-can/higher-layer-protocols/cankingdom/

[5] ——. *Kvaser Air Bridge Light HS (FCC)*. Accessed on: Jan 10, 2022. [Online]. Available: https://www.kvaser.com/product/kvaser-air-bridge-light-hs-fcc/

[6] STMicroelectronics. *STM32F3 Series*. Accessed on: Jan 10, 2022. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f3-series.html

[7] RT-Labs. *RT-Kernel*. Accessed on: Jan 3, 2022. [Online]. Available: https://rt-labs.com/product/rt-kernel/

[8] CAN in Automation. History of CAN technology. Accessed on: Mar 28, 2022. [Online]. Available: https://www.can-cia.org/can-knowledge/can/can-history/

[9] Bosch, “CAN Specification Version 2.0,” Robert Bosch GmbH, Stuttgart, Standard, 1991.

[10] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Pearson Education, 2014.

[11] J. Y.-T. Leung and J. Whitehead, “On the complexity of fixed-priority scheduling of periodic, real-time tasks,” *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, 1982.

[12] M. Brand, A. Mayer, and F. Slomka, “A matter of overhead – response time analysis of hard real-time systems in theory and practice,” in *MBMV 2021; 24th Workshop*, 2021, pp. 1–7.

[13] M. Song, S. H. Hong, and Y. Chung, “Reducing the overhead of real-time operating system through reconfigurable hardware,” in *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. IEEE, 2007, pp. 311–316.

[14] FreeRTOS. *FreeRTOS FAQ - Memory Usage, Boot Times & Context Switch Times*. Accessed on: Jan 10, 2022. [Online]. Available: https://www.freertos.org/FAQMem.html#ContextSwitchTime

[15] A. B. Lange, K. H. Andersen, U. P. Schultz, and A. S. Sørensen, “HartOS - a Hardware Implemented RTOS for Hard Real-time Applications,” *IFAC Proceedings Volumes*, vol. 45, no. 7, pp. 207–213, 2012, 11th IFAC,IEEE International Conference on Programmable Devices and Embedded Systems.

[16] F. Proctor and W. Shackleford, “Real-time operating system timing jitter and its impact on motor control,” *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 4563, 12 2001.

[17] A. Stothert and I. MacLeod, “Effect of timing jitter on distributed computer control system performance,” *IFAC Proceedings Volumes*, vol. 31, no. 32, pp. 25–29, 1998, 15th IFAC Workshop on Distributed Computer Control Systems (DCCS’98), Como, Italy, 9-11 September 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667017363310

[18] FreeRTOS. *The FreeRTOS™ Kernel*. Accessed on: May 23, 2022. [Online]. Available: https://www.freertos.org/RTOS.html

[19] P. Lindgren, J. Eriksson, S. Aittamaa, and J. Nordlander, “Tinytimber, reactive objects in c for real-time embedded systems,” in *2008 Design, Automation and Test in Europe*, 2008, pp. 1382–1385.

[20] RT-Labs. *RT-Kernel: User & Reference Manual*. Accessed on: Mar 25, 2022. [Online]. Available: https://rt-labs.com/refman/rt-kernel/

[21] L. Sha, R. Rajkumar, and J. Lehoczky, “Priority inheritance protocols: an approach to real-time synchronization,” *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[22] J. Goossens and P. Richard, “Overview of real-time scheduling problems,” in *Euro Workshop on Project Management and Scheduling*. Citeseer, 2004.

[23] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *J. ACM*, vol. 20, no. 1, p. 46–61, jan 1973.

[24] P. Ekberg and W. Yi, “Fixed-priority schedulability of sporadic tasks on uniprocessors is np-hard,” in *2017 IEEE Real-Time Systems Symposium (RTSS)*, 2017, pp. 139–146.

[25] NXP. I2c-bus specification and user manual. Accessed on: Mar 22, 2022. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[26] STMicroelectronics. STM32Cube MCU Package for STM32F4 series (HAL, Low-Layer APIs and CMSIS, USB, TCP/IP, File system, RTOS, Graphic - and examples running on ST boards). Accessed on: April 4, 2022. [Online]. Available: https://www.st.com/en/embedded-software/stm32cubef4.html

[27] ——. Integrated Development Environment for STM32. Accessed on: April 4, 2022. [Online]. Available: https://www.st.com/en/development-tools/stm32cubeide.html

[28] ——. STM32F301x/302x/303x/334x DSP and standard peripherals library, including 81 examples for 25 different peripherals and template project for 5 different IDEs (UM1581). Accessed on: April 4, 2022. [Online]. Available: https://www.st.com/en/embedded-software/stsw-stm32108.html

[29] Arduino. Arduino UNO R3. Accessed on: April 4, 2022. [Online]. Available: https://docs.arduino.cc/hardware/uno-rev3

[30] ——. Language reference: micros(). Accessed on: May 12, 2022. [Online]. Available: https://www.arduino.cc/reference/en/language/functions/time/micros/

[31] Kvaser AB. Kvaser Leaf Light HS v2. Accessed on: May 23, 2022. [Online]. Available: https://www.kvaser.com/product/kvaser-leaf-light-hs-v2/

[32] ——. Kvaser's CanKing - Free Bus Monitor Software. Accessed on: May 23, 2022. [Online]. Available: https://www.kvaser.com/software/kvaser-canking/