BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2021 Bc. Matouš Hýbl

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF CONTROL AND INSTRUMENTATION
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

TWO CHANNEL STEPPER MOTOR CONTROLLER
DVOUKANÁLOVÝ KONTROLÉR KROKOVÝCH MOTORŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Matouš Hýbl**
AUTOR PRÁCE

SUPERVISOR — **prof. Ing. Luděk Žalud, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Cybernetics, Control and Measurements**

Department of Control and Instrumentation

Student: Bc. Matouš Hýbl ***ID:*** 191600

Year of study: 2 ***Academic year:*** 2020/21

TITLE OF THESIS:

Two channel stepper motor controller

INSTRUCTION:

1. Explore and describe the stepper-motor controllers currently used in DCI for the BPC-PRP course. Describe their advantages and shortcomings.
2. Research and examine various stepper motor controller chips that may be used for the design. Select the best one after a discussion with the supervisor.
3. Design and develop a new stepper motor controller with I2C and CAN bus communication interfaces.
4. Develop software that demonstrates the controller's features.
5. Demonstrate the controller in a specific application, e.g. driving a small mobile roboti with differential drive configuration.

RECOMMENDED LITERATURE:

Motors for Makers: A Guide to Steppers, Servos, and Other Electrical Machines 1st Edition, Scarpino Matthew, 2015, Que Publishing, ASIN : B018KYYDMI

Date of project specification: 8.2.2021 ***Deadline for submission:*** 17.5.2021

Supervisor: prof. Ing. Luděk Žalud, Ph.D.

doc. Ing. Petr Fiedler, Ph.D.
Chair of study program board

ABSTRACT

The goal of this thesis is the development of a dual-channel stepper motor controller. Both the development of electronics and software is described. The electronics of the controller is based on the STM32F405 microcontroller, and Trinamic manufactured stepper motor controller ICs. For communication with higher-level systems, the controller utilizes the CANOpen protocol, I^2C, and USB buses. The whole electronics was designed in the KiCAD EDA and utilizes a 4-layer PCB and contemporary manufacturing technologies. As for the software, both firmware and control software were developed. Both of these pieces of software utilize the Rust programming language, which focuses on memory safety, performance and provides useful zero-cost abstraction. The Secondary goal of this thesis is to show how the language can be utilized for low-level embedded software development. The firmware of the controller implements independent motion control for each of the axes with both velocity and position control and provides failsafe mechanisms for cases of communication failures. The controller is meant to be used by the Robotics and AI research group and by students of the DCI, FEEC BUT.

KEYWORDS

stepper motor, electronics design, KiCAD, PCB, MCU, embedded software, Rust, memory safety, zero-cost abstractions, motion control, failsafe mechanism, CANOpen, I2C, USB, robotics,

ABSTRAKT

Cílem této práce je vývoj dvoukanálového kontroléru krokových motorů. V rámci práce je popsán jak vývoj elektroniky, tak vývoj příslušného software. Elektronika kontrolŕu je založena na mikrokontroléru STM32F405 a driverů krokových motorů vyráběných firmou Trinamic. Pro komunikaci s nadřazenými systémy je implementován protokol CANOpen a sběrnice I^2C a USB. Elektronika byla navržena v software KiCAD and využívá čtyřvrstvého plošného spoje a moderních výrobních technologií. Co se týká software, byl vyvinut jak firmware pro mikrokontrolér, tak software pro ovládání kontroléru. Obě části software využívají programovacího jazyka Rust, který se zaměřuje na bezpečnost práce s pamětí, rychlost a zero-cost abstrakce. Sekundárním cílem této práce je ukázat, jak lze tento programovací jazyk s výhodou použít pro programování nízkoúrovňového embedded software. Firmware kontroléru implementuje nezávislé řízení pohybu obou os kontroléru a to jak v rychlostním, tak v pozičním režimu a zároveň implementuje bezpečnostní funkce pro případy selhání komunikace. Výsledný kontrolér by měl být použit v rámcí výzkumné skupiny Robotiky a Umělé Inteligence a studenty na Ústavu Automatizace FEKT VUT.

KLÍČOVÁ SLOVA

krokový motor, návrh elektroniky, KiCAD, DPS, MCU, vestavný software, Rust, bezpečnost práce s pamětí, zero-cost abstrakce, řízení pohybu, bezpečnostní mechanismy, CANOpen, I2C, USB, robotika

EXTENDED ABSTRACT

Úvod

Tato práce pojednává o vývoji dvoukanálového kontroléru krokových motorů. V rámci práce je popsán jak vývoj elektroniky, tak vývoj software. Uvědomujeme si, že kontroléry krokových motorů jsou již mnohokráte vyřešený problém, který má mnoho komerčně dostupných řešení. Navzdory tomuto faktu jsme se rozhodli takový kontrolér vyvinout, a to ze dvou důvodů - výsledný kontrolér bude používán v rámcí předmětu BPC-PRP, což má jisté nároky na jeho hardware i software, a proto, že jsme se rozhodli naprogramovat firmware a řídicí software v programovacím jazyce Rust. V současnosti, je většina embedded projektů programována v klasických jazycích - C a C++, s jistými výjimkami v podobě jazyků Python a Ada. I když jazyky C a C++ jsou vhodné pro embedded vývoj z důvodu snadného přístupu k periferiím, tyto jazyky si s sebou nesou problémy v podobě nedostatečné ochrany před nevalidním přístupem do paměti a velkým množstvím nedefinovaných chování, které jsou mnohdy často závislé na implementaci kompilátoru. Podle studijí, které uvádíme v originálním úvodu je špatný přístup do paměti zodpovědný až za 70 % vysoce závažných problémů v prohlížeči Chrome. Problém s nedefinovaným chováním je sice méně důležitý než špatná práce s pamětí, ale přesto způsobuje problémy zejména v rámci vývoje, kdy prodlužuje jeho čas a tedy i finanční náročnost.

Tyto problémy nejsou ale jen doménou vysokoúrovňových systémů, ale ve velké míře jsou doménou i samotných embedded systémů, kde mohou mít mnohem katastrofálnější následky než v případě oněch vysokoúrovňových systémů. Věříme, že tyto problémy lze odstranit, nebo alespoň minimalizovat jejich dopady, právě použitím programovacího jazyka Rust, který byl navržen tak, aby předcházel právě chybám při práci pamětí a to i v případě vícevláknových systémů, minimalizoval nedefinovaná chování a to vše aniž by to nějak ovlivnilo výkon programu nebo jeho paměťovou náročnost.

Myslíme si, že pokrokové myšlenky a nástroje, které jazyk nabízí mohou do embedded systémů přinést mnohem více bezpečnosti a spolehlivosti než je nyní možné dosáhnout s konvenčními nástroji. To platí zejména v současnosti, kdy složitost embedded systémů neustále roste a zvyšují se požadavky na rychlost vývoje.

Příbuzné projekty

V rámci příbuzných projektů popisujeme projekty, které souvisí ať už s vývojem embedded systémů v jazyce Rust, tak se zabývají vývojem kontrolérů pr krokové

motory, nebo nám byly přímou inspirací.

Vzhledem k tomu, že výsledný kontrolér má být použit v předmětu BPC-PRP, popisujeme kontrolér, který je v rámci tohoto kurzu již používán - tedy kontrolér KM2, který je založený na mikrokontroléru ATMega8 a s nadřazeným systémem komunikuje pomocí sběrnice I^2C. Vzhledem k použití tohoto mikrokontroléru nelze ale použít vyšší frekvenci I^2C než 30 kHz, kvůli chybě ve funkcionalitě clock-stretching.

Dále popisujeme kontrolér KM3, který využívá modernější mikrokontrolér STM32F0, ale zatím nebyl do výuky nasazen. Pro tento kontrolér jsme již dříve vyvinuly firmware v programovacím jazyce Rust, čímž jsme otestovali schopnost jazyka a jeho nástrojů fungovat na low-endovém procesoru s nedostatkem paměti FLASH. Použitý procesor byl ale největší slabinou kontroléru, protože neumožňoval implementaci pokročilých funkcí.

Jako další projekt, ve kterém jsme vyvinuli firmware v jazyce Rust popisujeme projekt DCMotor - měnič pro DC motory. V rámci vývoje jsme nahradili původní firmware naprogramovaný v C++, čímž jsme dosáhli lepších vlastností a odstranění zásadních problémů, jako je třeba extrémní hlučnost motoru. Měniče s firmware naprogramovaným v programovacím jazyce Rust byly v sedmi kusech nasazeny na roboty pro výstavu Robot 2020, kde zdárně plní svou funkcí.

V rámci projektů, které nám byly inspirací zmiňujeme projekt Mechaduino, který integruje kontrolér pro krokový motor přímo na motor, přičemž je schopen zpětnovazebního řízení pomocí integrovaného enkodéru.

Dále zmiňujeme projekt Flott, který implementuje řízení pohybu krokových motorů v programovacím jazyce Rust.

Metody

V rámci použitých metod popisujeme krokové motory a jejich řízení. Vzhledem k tomu, že jsme se rozhodli použít integrované obvody pro řízení krokových motorů od firmy Trinamic, popisujeme rovněž jejich proprietální technologie, které jsou důležité pro správné nastavení integrovaných obvodů i jejich výběr. Kromě popisu krokových motorů popisujeme také použité sběrnice, a to CAN bus s protokolem CANOpen, I^2C a USB. Následně popisujeme programovací jazyk Rust a jeho důležité koncepty - proměnné a konstanty, princip vlastnictví a tzv. borrow checker, výčtové typy a pattern matching, datové struktury, traits a generika, makra, standarní knihovnu, testování a build systém Cargo. Na základě informací o programovacím jazyce se přesouváme k popisu toho, jak lze v tomto jazyce vyvíjet pro embedded systémy. Diskutujeme podporu pro různé rodiny a jádra mikrokontrolérů, organizace vyvíjející nástroje a knihovny pro embedded Rust, přístup k periferiím, abstrakce pomocí HAL, přístup ke globálnímu stavu (který v rámci bezpečnosti považuje Rust za

nebezpečný). Dále popisujeme asynchronní programování v Rustu, které by mohlo zcela změnit způsob jakým je k embedded software přistupováno. Důležitou součástí embedded Rustu jsou nástroje, které byli vyvinuty pro snazší práci s mikrokontroléry - jsou jimi například generátor kódu pro přístup k periferiím, extrémně rychlé logování, nebo ochrana paměti před přetečením zásobníku. Jako nedílnou součást moderního vývoje software popisujeme i automatizované testy a Continuous Integration pro embedded systémy.

Po nezbytném teoretickém úvodu se dostáváme k samotnému vývoji kontroléru. Nejprve zadefinujeme požadavky na zařízení, které plynou se zadání, ale i z předchozích zkušeností a příbuzných projektů. Tyto požadavky jsou naprosto nezbytné pro kontrolu plnění cílů projektu.

Poté se dostáváme k vývoji hardware kontroléru, kdy nejprve provedeme rozhodnutí týkající se výběru mikrokontroléru a dalších obvodů a následně vyvineme schéma kontroléru, společně s deskou plošných spojů. Vývoj elektroniky byl proveden v nástroji KiCAD.

Dále následuje popis vývoje firmware kontroléru, nejprve se věnujeme architektuře firmware, na kterou navazujeme popisem kritických komponent kontroléru. Velkou pozornost věnujeme popisu vytvořených abstrakcí, díky kterým je firmware kontroléru do značné míry univerzální. Za zmínku jistě stojí abstrakce pro řízení samotných motorů nebo pro enkodéry.

Na závěr je popsán vývoj řídicí aplikace pro náš kontrolér. Původní cíl byl vytvořit řídící aplikaci s grafickým uživatelským rozhraním a možností konfigurace, ale vzhledem k nedostatku času byla vytvořena pouze jednoduchá aplikace schopná řídit obě osy kontroléru a to jak v rychlostním, tak v polohovém režimu.

Výsledky

Výsledkem práce je funkční kontrolér pro krokové motory, který je schopen tyto motory řídit jak v rychlostním, tak v pozičním módu. Pro řízení je možné použít buď sběrnici CAN, s protokolem CANOpen, nebo sběrnici I^2C. Konfigurace kontroléru je možná přes integrované USB rozhraní.

V rámci výsledků rovněž popisujeme finální stav projetku společně s přehledem plnění požadavků na kontrolér. Nedílnou součástí výsledků je i popis programátorského rozhraní a datových modelů, pomocí kterých lze kontrolér řídit.

Jako další součást výsledků popisujeme dvě demonstace funkčnosti kontroléru - jednoduchý lineární posuv řízený přes I^2C a malého robota s diferenciálním podvozkem řízeného po sběrnici CAN.

Závěr

V rámci této práce jsme navrhli, vyrobili a naprogramovali dvoukanálový kontrolér krokových motorů. Byly vytvořeny dvě verze hardware, lišící se jak zapojením, tak použitými integrovanými obvody pro řízení krokových motorů, tak designem desky plošných spojů. Druhá verze hardware je sice mnohem pokročilejší než ta první, i přesto jsme v rámci práce vymysleli další způsoby jak tuto verzi hardware dále vylepšit.

Pro kontrolér jsme vyvinuli firmware v programovacím jazyce Rust. Momentální verze firmware bohužel momentálně podporuje pouze první verzi hardware, ale doprogramování podpory pro druhou verzi by nemělo být příliš náročné. V rámci programování jsme využili všech možných nástrojů, které nám jazyk poskytuje - zejména v rámci vývoje abstrakcí, kde jsme hojně využívali traits a generiku. Věříme, že firmware byl naprogramován dostatečně abstraktně na to, aby jej bylo možné jednoduše rozšiřovat a vylepšovat. Musíme podotknout, že embedded Rust je již dostatečně vyspělý na to, aby v něm šly bezproblémově a efektivně programovat větší či menší embedded projekty.

Pro jednoduchost testování jsme rovněž vytvořili jednoduchý řídicí software schopný řídit kontrolér jak v rychlostním, tak v pozičním režimu.

Projekt kontroléru plánujeme dále vyvíjet a rozšiřovat, přičemž si uvědomujeme, že i když je momentální výsledek použitelný, tak má k dokonalosti daleko. Plánujeme například celý firmware kontrolér automatizovaně testovat, vyrobit třetí a snad poslední verzi hardware, a mnohem více. Přes toto všechno si myslíme, že by kontrolér šel i v tomto stavu nasadit do výuky jako součást předmětu BPC-PRP.

HÝBL, Matouš. *Two channel stepper motor controller*. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Control and Instrumentation, 2021, 142 p. Master's Thesis. Advised by prof. Ing. Luděk Žalud, Ph.D.

Author's Declaration

Author:	Bc. Matouš Hýbl
Author's ID:	191600
Paper type:	Master's Thesis
Academic year:	2020/21
Topic:	Two channel stepper motor controller

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno 17.5.2021 .....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I want to express my gratitude to my supervisor, prof. Ing. Luděk Žalud Ph.D. for his help and leadership. I would also like to express my gratitude to my colleagues at the Robotics and AI research group of FEEC and CEITEC BUT who supported me and helped me. Special thanks belongs to Ing. Lukáš Kopečný Ph.D. who supplied me with tasty beer, and Ing. Adam Ligocky for providing emotional support. I would also like to thank Ing. Ondřej Baštán for helping me with the direction of the project when I developed the second hardware revision. Also, I would like to thank the embedded Rust community, especially James Munns and Hanno Braun, who provided me with essential feedback at the beginning of software development. Last but not least, I would like to thank my girlfriend for her never-ending support and also to my family and friends who helped me and supported me along the way.

Contents

List of Figures

List of Tables

Listings

Introduction & Motivation

This thesis describes the design and development of a simple dual-channel stepper-motor controller. We acknowledge that driving of stepper motors is nowadays a solved problem with many solutions that are commercially available. Given this fact, we needed to differentiate the project from others. The first difference is that the target of this project is driving stepper motors in the DCI FEEC BUT's (Department of Control and Instrumentation, Faculty of Electrical Engineering and Communications, Brno University of Technology) Robotics and AI group. The controllers will be used for students' projects and development of our robots, which imposes some requirements on the PCB (Printed Circuit Board) size and used technologies. The second, albeit more important difference, is that in contrast with classical embedded systems, this stepper motor controller's firmware and service software will be developed in the Rust programming language. We believe that this difference is the core of the thesis and further distinguishes itself from other theses and projects on embedded development.

In general, the majority of embedded systems nowadays are developed in the C/C++ programming languages [3, 4, 5]. There are some exceptions - there are systems developed in Ada, and currently, the embedded development in Python is starting to take off in hobby projects [6]. While C and C++ are suitable for the development of embedded systems because they allow for direct hardware access, and the programs written in them can be extremely performant, they carry the problem of memory unsafety and undefined behavior.

Memory unsafe code is the leading cause of many critical software problems, be it security vulnerabilities or safety hazards. Recent Chrome browser analysis and report show that around 70 % of high severity problems are memory safety problems - meaning problems with pointers. A staggering half of these are use-after-free problems [7]. Similar results show other statistics, namely from the cURL project[8]. The notorious Heartbleed bug in the OpenSSL was also a problem of the ability of the program to access memory used by other parts of the program, allowing the attacker to steal confidential data from the memory [9].

The problem of undefined behaviors and the inability of the commonly used tools to spot them can be as harmful as memory safety problems, but in general causes problems mostly during development, making the development take longer and therefore become more expensive. The symptom of undefined behaviors is when the program behaves as was not intended, but with seemingly error-less code.

While the problem of memory safety and UBs (Undefined Behavior) seem to generally be problems of higher-level systems and not embedded systems, we believe that these problems apply to embedded systems as well, as these problems

can have as devastating (or even more devastating) effects as the above mentioned security vulnerabilities. Imagine a robot uncontrollably spinning and destroying its surroundings because some part of a program has overwritten its controls by mistake.

We believe that the Rust programming language can solve both of these problems. While being a relatively novel language for systems development (development started in 2006), the language is designed to be memory-safe, even delivering memory safety for state shared between threads. Its focus on type safety and strong guarantees about the performance of systems programming allows the developers to create powerful, yet in many cases zero-cost (memory or performance) abstractions. These features is especially useful as the complexity of all systems is rising. We believe that to deliver great systems, human programmers need to be aided by all available tools. Even though the language primarily targets higher-level systems, its design allows for it to be used with bare-metal embedded systems, bringing its advantages to these low-level systems.

We also believe that the novel approaches brought by the language and its ecosystem could bring improvements to the existing embedded development approaches, and also, the strictness of the language could bring more safety and reliability to embedded systems. Some of these approaches can be unit-testing and integration testing, dependency management, and embedded-systems-dedicated open-source tooling.

With this information in mind, we decided to develop the controller's firmware and control application in Rust, showcasing the language's advantages and disadvantages. This project follows the development of firmware for other motor controllers, described in the Chapter 1, which were presented at the PAIR conference [10]. Another aim of this project was to push forward the development of electronic devices at the Robotics and AI research group - using high-performance MCUs (Microcontroller Unit), state-of-the-art stepper drivers, effective 4-layer PCB design, and contemporary manufacturing capabilities.

1 Related Work

This chapter describes the current state of the stepper motor controllers in the Robotics & AI group and the efforts to improve it. Ongoing efforts to develop embedded systems in the Rust programming language are also described, both in the context of our research group and also in general. Finally, similar projects - either software or hardware-wise are reported.

1.1 KM2

Nowadays, a second generation of the KM2 stepper motor controller is widely in use in the Robotics and AI research group. It is used primarily by the students of the BPC-PRP course for driving a simple differentially driven robot. A render of the KM2 controller can be seen in the Figure 1.1. The controller utilizes an ATMega8 paired with two stepper motor controllers DRV8825, that are utilized in the form of breakout boards generally used in the now obsolete 3D printer controlling RAMPS boards. The motor controller is controlled using the I^2C bus. There are two major shortcomings of the driver - the used MCU's I^2C peripheral's clock-stretching is not compatible with Raspberry Pi's, causing problems on clock speeds higher than 30 kHz. The second shortcoming are the used driver chips which are quite loud and do not support contemporary advanced features. Overheating is also common with them.

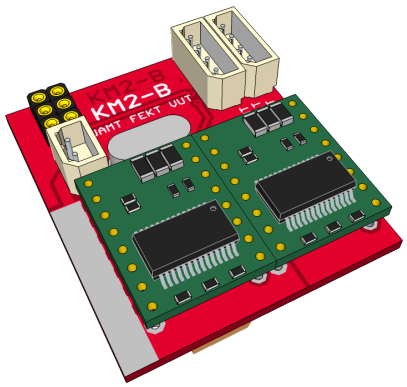


Fig. 1.1: KM2 motor controller render [11].

1.2 KM3

The KM3 (or KM2-C) was supposed to be a successor to the previously described KM2 controller, and its main goal was to solve the clock stretching problem by utilizing an STM32F031 MCU. Another advantage of this revision was that the breakout boards for motor driver chips were replaced with driver chips soldered directly on the driver PCB. The controller can be seen in the Figure 1.2. Even though the new STM32F031 MCU was an improvement over the ATMega8, it proved to be the bottleneck for implementing new functionality for the motor controller as the MCU has very limited memory, both FLASH and RAM and also limited peripherals. An example of these limitations being that the lack of pins made it impossible to directly generate pulses to control the STEP/DIR interface of the motor driver IC; therefore the control had to be done manually in the software. Another problem with this design is that the MCU utilizes a Cortex-M0 core, which means that the support for atomic instructions is missing, making it hard to work with guarantees about memory safety in cases of interrupt routine being called during memory manipulation.

We developed the Rust firmware [12] for this board and concluded that the board and its design might be suitable for the students' robot projects, but it is way too limited to be used in more serious and complex projects. We also concluded that the hardware and the technology it has been designed upon, as well as its goals, are obsolete, and that we should not pursue the development of this board further.

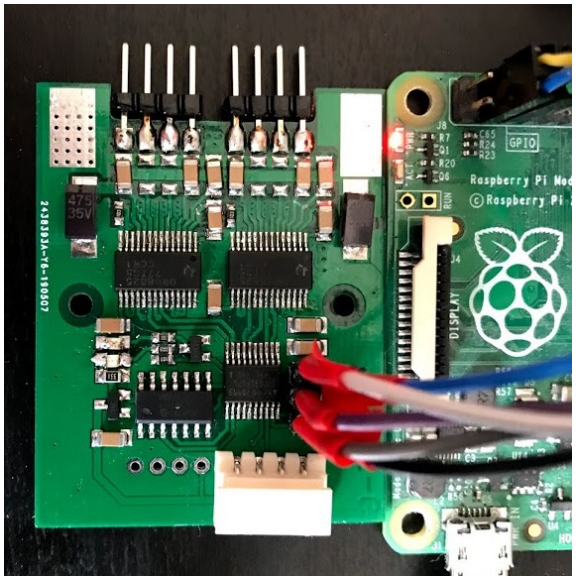


Fig. 1.2: The KM3 motor controller connected to a Raspberry Pi.

1.3 DCMotor

The DCMotor is a DC (Direct Current) motor controller, developed in the Robotics & AI research group. It was designed for feedback control of DC motors, primarily DC motors manufactured by Maxon. The main design goal was to provide a cheaper alternative to the Maxon EPOS motor controllers. The driver alongside a connected Maxon DC motor can be seen in the Figure 1.3. Originally, the firmware for the motors, developed by Ing. František Burian, Ph.D., implemented current control and velocity control. However, the firmware exhibited unwanted behavior, such as high motor temperature rises and unwanted high-pitch noise. After consulting the problem with Ing. Lukáš Kopečný Ph.D., we decided to rewrite the firmware in Rust and remove the current controller, with the reasoning that current control of such low inductance motor makes not much sense, and instead, we replaced it with current limiting and failsafe overcurrent motor disabling. The new firmware, and some hardware modifications were successfully deployed to seven DCMotor drivers as part of the exhibition robots for the Technical Museum in Brno, where they worked better than with the original firmware.

This driver was the first embedded project that used the Rust programming language to develop the firmware. We believe that using the language was the right choice and made the firmware simpler to use and made it possible to develop it in such short time. It can be said, that the work on the firmware for this board laid the foundation for the work on this thesis.

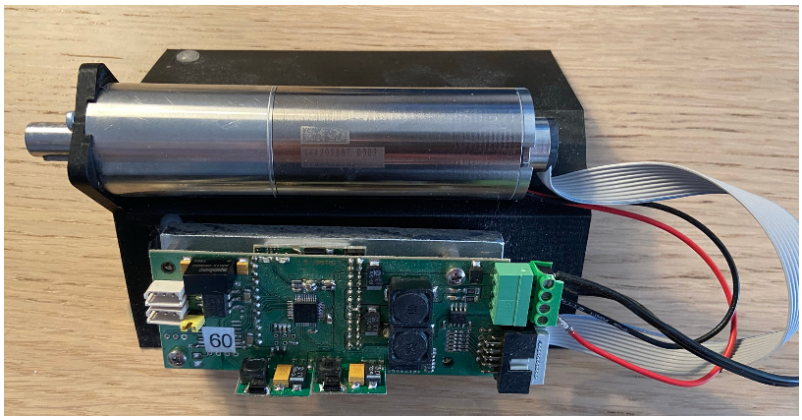


Fig. 1.3: The DC Motor driver with a connected Maxon DC motor.

1.4 Mechaduino

Mechaduino is a project that aims to create a feedback-controlled servo motor out of a stepper motor. The creators achieve that by mounting a PCB on the back of the motor that contains the power stage, a MCU, and a 14-bit magnetic encoder [13]. The mounting on the back of the stepper motor can be seen in the figure 1.4. The big advantage that this project brings is the integration of the whole system de-facto into the motor, removing any need for a separate controller board. On the other hand, the controller doesn't leverage any existing stepper motor controller solution and instead implements the winding control manually. When compared to our proposed solution, the Mechaduino has many advantages even though it is only capable of controlling only one motor, it contains an encoder for feedback control and implements servo control algorithms out of the box. On the other hand, our proposed solution leverages a state-of-the-art stepper motor controller ICs (Integrated Circuit), making it potentially less error-prone and better for future use and development. If a semestral thesis and preliminary market research was preceding this project, the Mechaduino would provide valuable information to improve the design of our project.

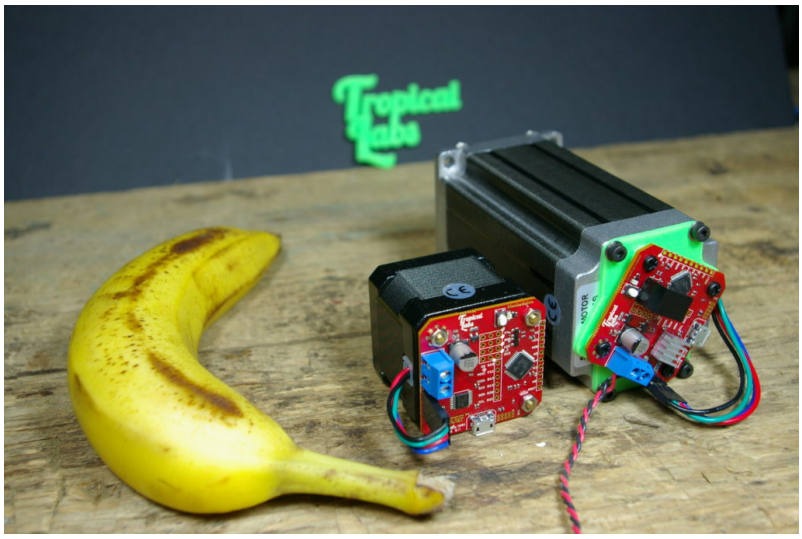


Fig. 1.4: The Mechaduino controller boards mounted on the back of a stepper motors [13].

1.5 Flott

Flott is a set of libraries suitable for developing motion controllers programmed in the Rust programming language[14]. It is a relatively new project, and as of now, it contains an abstraction layer for stepper motors and acceleration ramp generators. The project is taking a different approach to controlling the stepper motors we are. It aims to utilize software pulse generation instead of timers and uses a variable step period in ramp generation. Even though this is a good approach, we chose not to follow this model and instead implement this asynchronously using the MCU peripherals. On the other hand, the Flott project might be a great source of inspiration for future development, and maybe sometimes the SM4 motor controller might utilize it.

1.6 Takeaways from Related Work

The past stepper motor development efforts in the Robotics & AI research group showed the current solutions weak spots and advantages, which resulted in the following directions of the development of this project:

- The project shall use a powerful, modern, and capable MCU to fully support various features of the controller even in the future.
- The project shall use the state-of-art motor controller ICs.
- The project shall use the Rust programming language for firmware and control software development.

The DC Motor project showed us that writing a fully functional embedded firmware is possible and viable option.

The Mechaduino project serves as a great inspiration for what can be achieved in a servo motor based on a stepper motor.

Flott shows us that more people are trying to achieve building motion controllers in Rust and that we can get inspired from them and share knowledge with them. We've been in contact with the Flott creator and consulted some ideas with them.

2 Methods

This chapter outlines the methods used while developing the SM4 stepper motor controller.

First, an introduction is given about the problematics of stepper motors and their control and about communication buses utilized by the project. Second, a brief introduction to the Rust programming language is given, alongside a more in-depth introduction to the current state of using Rust programming language for embedded systems.

Furthermore, we declare the requirements for the resulting hardware and software. These requirements consist of functional requirements, non-functional requirements, and constraints.

The development of two hardware revisions is described in the further sections. The hardware design choices are described. These choices are based on the requirements and the Chapter 1 on related work. Electrical schematics of the vital parts of the electronics are described.

Further, the development of the firmware itself is outlined, with some interesting parts being described in detail. Finally, the development of a control software used for controlling the stepper motor controller is gone over.

2.1 Stepper motors

This section gives a brief introduction of stepper motors and their control. First, stepper motors and their types are described. Then, a comparison of stepper driver ICs is given, and some of the motion control technologies by Trinamic are described.

Stepper motors are a type of DC motor, which move in discrete steps[15, 16]. Such movement is achieved by their construction - they consist of a stator and a rotor, where the stator is made of coils(two coils form a phase) wound on ridges, whereas the rotor consists of a ferromagnetic structure - either a permanent magnet or a variable reluctance iron core[16].

2.1.1 Working principle

The basic working principle of stepper motors can be seen in the Figure 2.1. In the Figure, we can see a three-phase bipolar stepper motor. First, the coils of the stator winding **A** are energized, which causes the ferromagnetic rotor to align with the magnetic field induced by the phase winding. In the second step, the winding **B** is energized, causing the rotor magnetic field to realign with the newly induced magnetic field of the second winding. This causes the motor to move. In the next step, the winding **C** is energized, which again causes a realignment of the rotor. In the following steps, the coils are energized again but with different polarity making the rotor make a full turn.

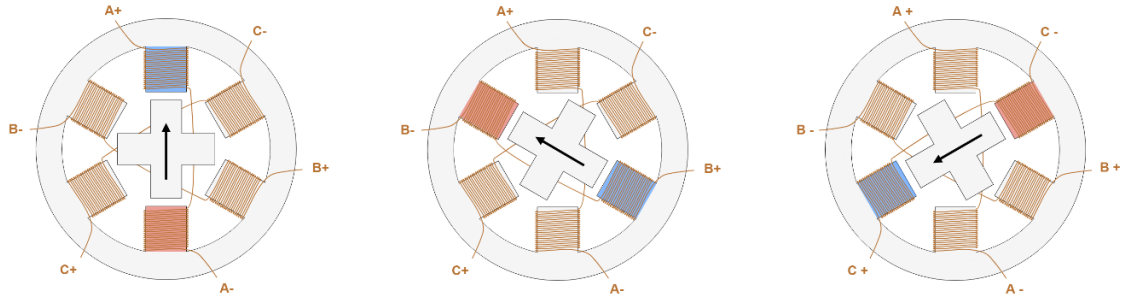


Fig. 2.1: Working principle of a stepper motor [16].

2.1.2 Rotor

Three different constructions of rotors exist [16]:

- **Permanent magnet rotor** - utilizes a permanent magnet in the place of the rotor. An advantage of this type of rotor is good torque, and also detent torque (the resistance of the motor shaft when no windings are energized) [16].
- **Variable reluctance rotor** - the rotor consists of a shaped iron core. The torques are generally lower, and there is no detent torque [16].

- **Hybrid motor** - is created by combining a permanent magnet rotor with a variable reluctance rotor. There are two magnetic caps with teeth on top of each other that have an angular shift between them. The rotor is magnetized axially [16].

2.1.3 Stator

The construction of the stator depends on the number of phases the motor has. Every phase consists of two windings, where the windings can be center-tapped or not, which determines if the motor is bipolar or unipolar. With unipolar windings, the center-tapped lead is connected to the input voltage, and the direction of the magnetic field is controlled by connecting the ground to the other leads. Bipolar motors do not have center tapped lead, and the coil itself is controlled using an H-bridge.

2.1.4 Phase Winding Energizing Techniques

The way of energizing windings described in the Subsection 2.1.1 is only one of four ways of controlling the windings. This technique, where only one of the phases is energized at a time, is called the wave mode. This mode was described in detail in the Subsection 2.1.1, and the sequence of energizing windings can be seen in the Figure 2.2.

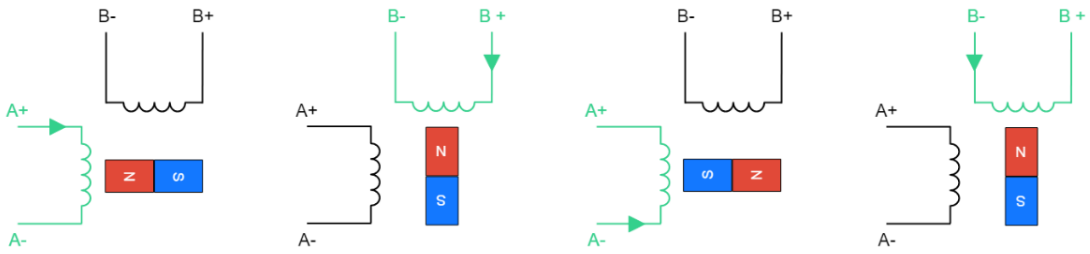


Fig. 2.2: Controlling stepper motor phase windings in wave mode [16].

Another way of driving the motor is called the full-step mode. In this mode, two phase windings are energized at the same time. Changing the current direction in the winding causes the rotor to realign. The advantage of this mode is higher torque as the magnetic field is stronger when the two of the phase windings are energized. The working principle can be seen graphically in the Figure 2.3.

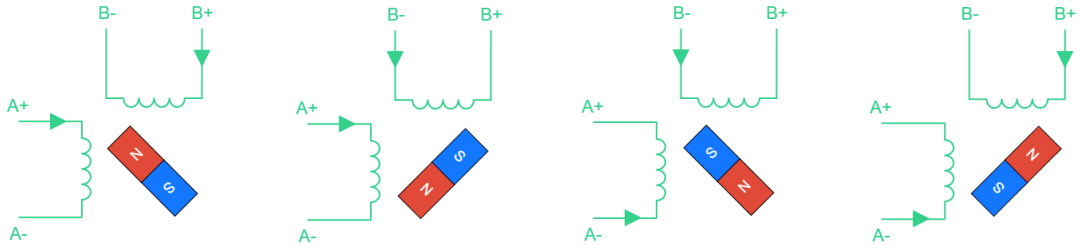


Fig. 2.3: Controlling stepper motor phase windings in full step mode [16].

Combining the wave mode, and the full step mode results in a half-step mode. In contrast to the previous driving modes, the step size of this mode is half of the previous mode - in the case of this virtual motor with permanent magnet motor 45°, instead of the original 90°. This mode alternates between energizing only one phase winding and energizing both phase windings. The disadvantage of this mode is that the output torque is not constant as the torque is different when both phase windings are energized and when only one of them is. The working principle can be seen in the Figure 2.4.

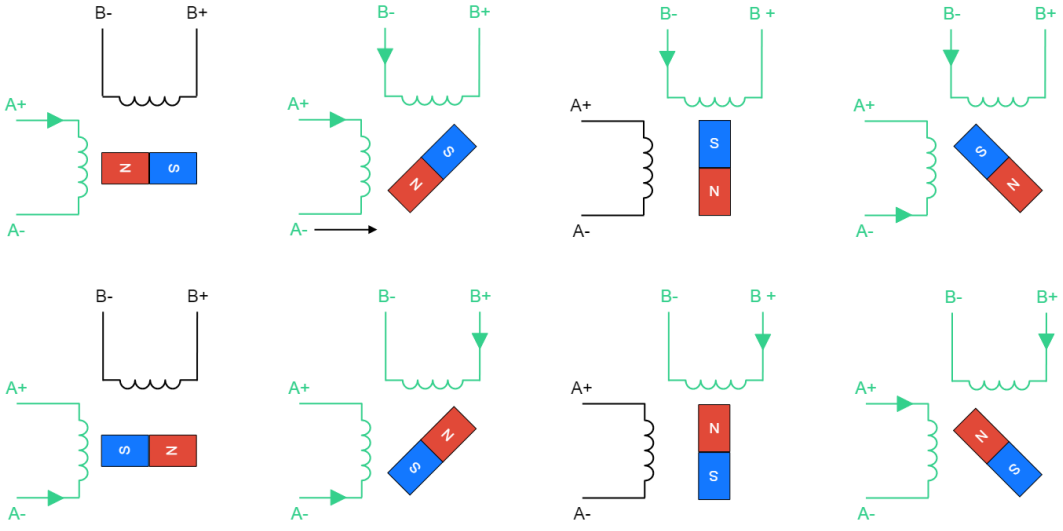


Fig. 2.4: Controlling stepper motor phase windings in half step mode [16].

The last technique for driving stepper motors is microstepping. The advantage of this mode is that it reduces step size and has constant torque output [16]. The

working principle for this mode is that the current flowing through the phase winding is controlled in some ratio, finely positioning the rotor, as shown in the Figure 2.5. Microstepping is nowadays the prevalent way of stepper motor control as it allows for precision control and allows for constant torque.

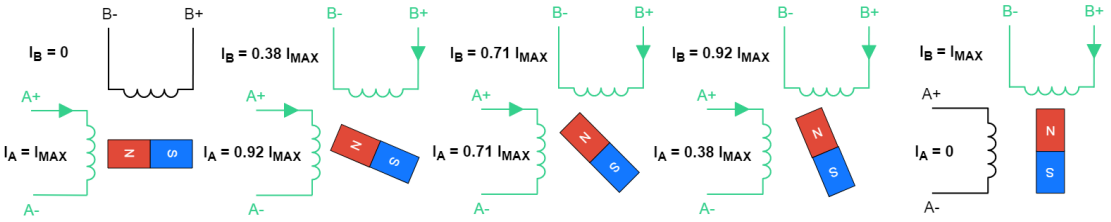


Fig. 2.5: Controlling stepper motor phase windings in microstepping mode [16].

2.1.5 NEMA17 style stepper motor

In this Section, a typical NEMA17 style motor is described. NEMA17 is a standard that describes the flange size, where the number denotes the flange size in tenths of an inch[17], in this case, NEMA17 meaning 1.7". The NEMA17 style motors are commonly used in 3D printers. The specific motor is a 17HS4401, it is a two-phase bipolar stepper motor with a step angle of 1.8°. The motor's length is 40 mm, its rated current is 1.7 A, and it has a holding torque of 0.4 Nm[18]. An image of the motor can be seen in the Figure 2.6.

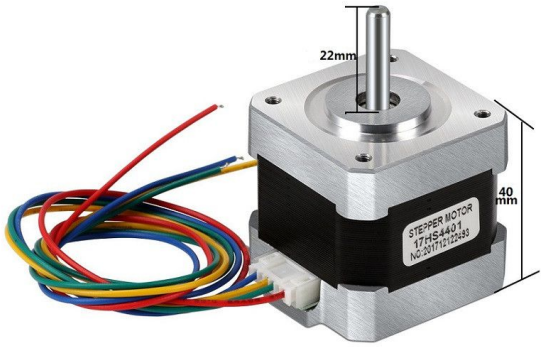


Fig. 2.6: A typical look of a NEMA17 motor.

2.1.6 Trinamic motion control technologies

As we described in the Section 2.7.2, we decided to utilize Trinamic made driver ICs, and their stepper control technologies. It is vital to describe these technologies as they have an immediate impact on the driver IC's performance and properties.

MicroPlyer™

MicroPlyer™is a microstepping interpolator. The reason for the interpolator is that the drivers feature 256 microsteps per step, and generating the stepping signal would be impractical if not impossible for some systems. The driver is configured with the number of microsteps that the driver will consider a full step, and the MicroPlyer™interpolates the rest of the microsteps up to the 256 microsteps per step[19].

Voltage Chopper Modes - SpreadCycle™, StealthChop™

To define chopper modes, we first need to define the current control modes for a bipolar stepper motor. The current control modes are the ON-phase, fast decay, and slow decay. These modes can be seen in the Figure 2.7.

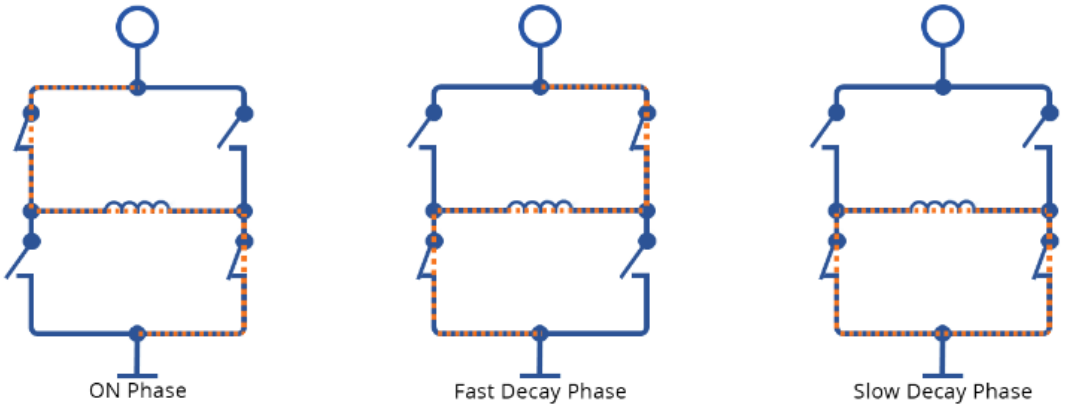


Fig. 2.7: Stepper motor winding control modes [20].

The current in the winding is controlled using voltage choppers. First, a very high voltage is applied to the winding, which causes a current rise in the winding. When the current exceeds a specific limit, the voltage is chopped (turned off). When the current drops below a specified limit, the very high voltage is turned back on. Using this approach, it is possible to maintain a relatively constant current in the winding[20].

When using a Constant T_OFF Current Chopper, the basic chopper principle is enhanced by first energizing the winding, then utilizing fast-decay, and then slow decay. This is a commonly used chopper mode as it is quite simple, but it causes motor vibration and high pitch noise. This problem is caused by the relationship between the fast decay and slow decay phase, resulting in the average current being lower than the desired target. This means that there are moments when the motor has no torque, which in turn causes vibrations[20]. The graph showing the winding current in time can be seen in the Figure 2.8.

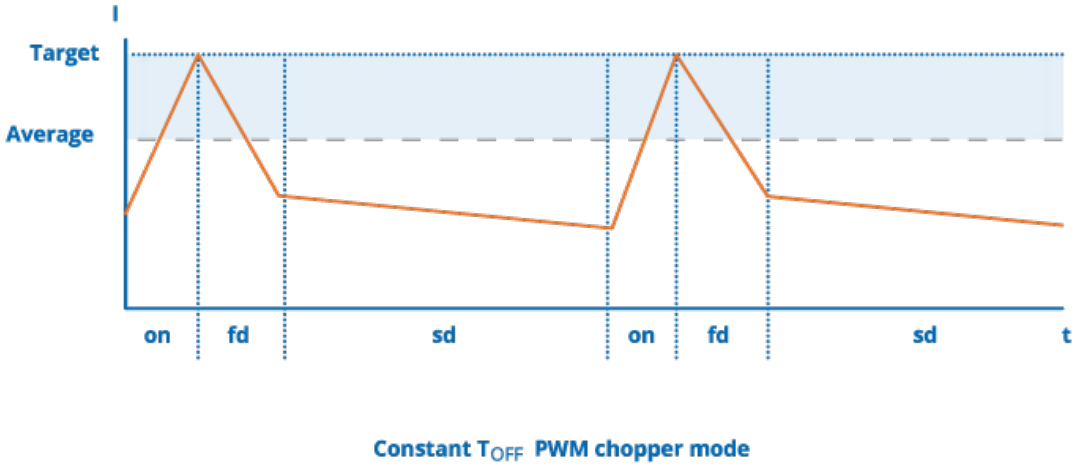


Fig. 2.8: Constant T_OFF Chopper Mode [20].

The SpreadCycle™current chopper is an improvement over the Constant T_OFF Current chopper. According to Trinamic, it automatically applies a proper relation between slow decay and fast decay to create the optimal fast decay for that cycle[20]. This technique leads to the average current matching the target current, making the current wave resembles a sine wave. This technology also remains effective at higher RPMs, where the classic constant T_OFF Chopper shows current deformations[20]. The current in time can be seen in the Figure 2.9.

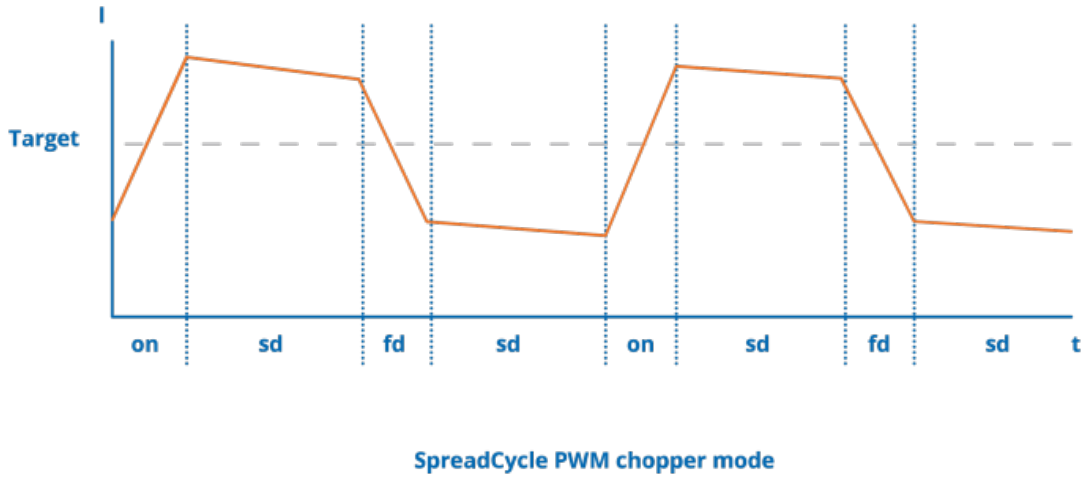


Fig. 2.9: The SpreadCycle™Mode current graph [20].

StealthChop™is the most advanced voltage chopper technology Trinamic drivers provide. The chopper completely silences stepper motors by eliminating the noise caused by unsynchronized motor coil chopper operation, PWM jitter and regulation noise at the sense resistors[20]. The chopper modulates the current using the PWM duty cycle, which minimizes the current ripple[20]. Adjusting the PWM duty cycle also results in a perfect current sine wave, and minimizing the current ripple minimizes Eddy currents in the stator, which in turn leads to less power loss and increase efficiency[20].

StallGuard™

The StallGuard™technology utilizes the back EMF (ElectroMotive Force) to analyze the load of the motor. This provides the drivers with a sensorless load measurement. This technology may be utilized for sensorless homing, self-calibration or distance measurement. The StallGuard™technology also prevents step loss when the axis is obstructed[21].

CoolStep™

CoolStep™is a technology that adjusts the motor current based on the feedback provided by the StallGuard™technology. This technology always drives the motor at the minimum required current sufficient for driving the actual load. That leads to reduced current consumption and also reduces heat generation. The technology also allows for temporary current boosts. An example of the dependency on the motor current on the load torque can be seen in the Figure 2.10.

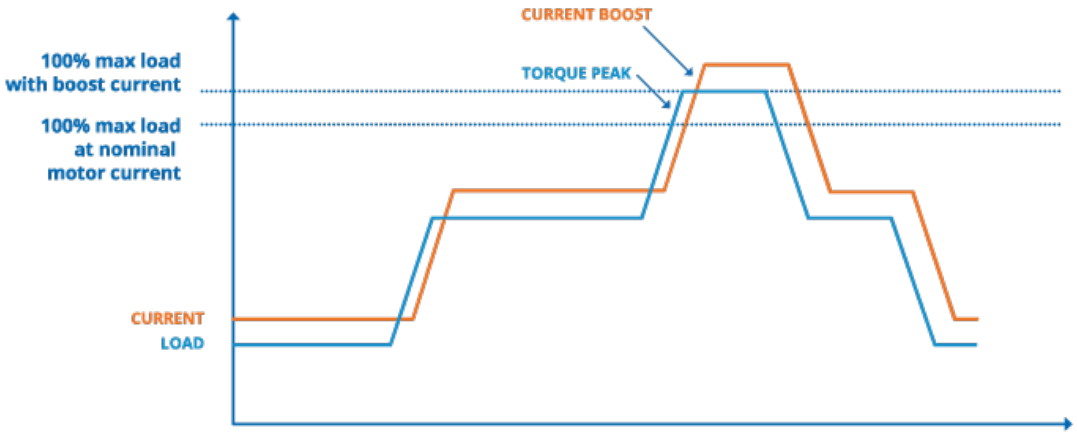


Fig. 2.10: The Trinamic CoolStep™technology [21].

2.2 Communication Protocols

According to the thesis instruction and the requirements **FR-04**, **NFR-02**, **C-03**, **C-05**, the stepper driver should feature CANOpen and I2C interfaces for control and configuration. The USB interface shall be used only for configuration. This section aims to give a brief overview of these communication interfaces and protocols utilized with them.

2.2.1 CANOpen

CANOpen is a set of higher-level protocols based on the CAN bus physical and link layer. The protocols are designed around the Master-Slave model, where there is a specific device acting as the master which controls the CANOpen network (e.g., synchronization) and up to 127 slave devices-nodes. Every device in a CANOpen network is assigned a unique ID. Within the CANOpen protocols, the CAN frame sent to the device either targets a specific device or all of them. The frames that target specific devices contain the identifier of the frame (e.g., PDO CAN ID) bit-or with the device ID. The CANOpen protocols provide standardized communication objects (COBs) with specific identifiers (IDs) for time critical processes, communication and network management[22]. The most critical parts CANOpen protocol is the SYNC protocol, the PDO protocol, the SDO protocol, and the NMT protocol. Other protocols are the EMCY protocol, TimeStamp protocol, and LSS protocol.

2.2.2 Object Dictionary

In a CANOpen device, the Object Dictionary contains the global shared state of a device. This means that the software responsible for communicating over CANOpen protocols sends out the data available in the Object Dictionary and, in turn, writes to it the received data. Apart from being a data storage for the communication interface, the Object Dictionary serves as a data source for the algorithms and systems running on the device itself. There are two numbers used to access the values in the Object Dictionary - first, the Index - a 16-bit unsigned value, and the SubIndex - an 8-bit unsigned value. Some of the Index ranges are reserved by the CANOpen specification for predefined parameters such as communication settings, while other Index ranges contain application-specific parameters[22].

SYNC protocol

The SYNC protocol is responsible for synchronizing the communication on the bus. It initiates the transfer by sending a CAN frame with the identifier **0x80**, after which every device on the bus sends/receives synchronous data objects, such as PDOs (Process Data Units). The CAN frame can also contain a single byte containing a SYNC number that can be utilized to conditionally send synchronous data or for more granular synchronization[22]. SYNC message is generally sent periodically.

PDO protocol

Process Data Objects (PDOs) are used for broadcasting high-priority status and control information[22]. Each PDO consists of a single CAN bus frame and can contain up to 8 bytes of data. The contents of the PDO can be set in some devices according to the specific application needs using a technique called PDO mapping, where PDO data are mapped to Object Dictionary fields. There are three mechanisms used to transmit PDOs - asynchronous PDOs can be sent upon an event trigger in the device. Asynchronous PDOs can also be remotely requested using the RTR bit in the CAN frame. Synchronous PDOs are broadcast as a reaction to the SYNC protocol. There are two types of PDOs - RxPDOs and TxPDOs. The RxPDOs are the PDOs that are received by the target device, while the TxPDOs are the PDOs that are transmitted by the target device. There are four available RxPDOs and four available TxPDOs each PDO has a CAN ID assigned, as can be seen in the Table 2.1.

PDO	RxPDO CAN ID	TxPDO CAN ID
PDO1	0x200	0x180
PDO2	0x300	0x280
PDO3	0x400	0x380
PDO4	0x500	0x480

Tab. 2.1: RxPDO and TxPDO CAN IDs[23]

SDO protocol

The SDO (Service Data Object) protocol is used to directly read or write entries of the device's object dictionary. This protocol utilizes the client-server model, where the device with the target Object Dictionary is the server, and the other device is the client. One SDO consists of two CAN frames with different IDs that represent the transaction. Given there are two CAN frames, the protocol is confirmed[22].

There are three variants of the SDO protocol - expedited transfer, normal (segmented) transfer, and block transfer. The expedited transfer can be utilized when the target data has a length of 4 bytes or less. To transfer data with greater length the segmented, or the block transfers may be used, where the block transfer shall have a slightly lower protocol overhead[23].

NMT protocol

NMT (Network Management) protocol is a protocol implemented by all the slave devices in the network. It consists of a finite state machine that describes the device's state with relation to the bus and the rest of the system. The states are **Initialization**, **Preoperational**, **Operational** and **Stopped**. After the device starts, it shall automatically enter the **Initialization** state. Using an NMT command CAN frame, the device goes to the next state, in this case, **Preoperational**. The state themselves have a certain meaning for the behavior of the device. For example motor movement must be disabled until the device enters the **Operational** state.

While the master controls the slave devices by commanding them to go to a certain state, the slave devices use the NMT Heartbeat protocol to periodically notify the master (and other devices) of their current state. Devices can be configured to, for example, stop movement if another slave device or master stops sending these Heartbeat messages.

CAN bus

The Controlled Area Network is a bus most commonly used in automotive for connecting ECUs (Electronic Control Units). The bus was developed by Bosch and codified into the ISO11898-1 standard. CAN bus utilizes a single differential pair making simplifying the wiring of a complex system consisting of many ECUs[24].

On the physical layer, the bus has two states - recessive and dominant, where recessive means that the differential voltage between the CANH and CANL signals is less than a minimum threshold voltage. In contrast, the dominant state means that the differential voltage is higher than the minimal threshold voltage[24]. The dominant state is achieved by sending a logical 0 through the network, while the recessive state is achieved by sending logical 1. CAN bus utilizes the CSMA/CD media access control protocol, which allows for collision detection and potential retransmission of CAN frames. For collision detection, it is vital that the dominant state overrides a recessive one.

There are two types of frames transmitted on the bus - standard frames and extended frames. These frames differ in the identifier length, where the extended frame allows for 29-bit long identifier in contrast to the standard frame, which allows for only 11-bits. Identifier length is selected on a per-frame basis using the IDE bit in the frame. Each CAN frame may contain up to 8 bytes of data, and the data length is controlled by the four DLC bits in the frame. The structure of a CAN frame can be seen in the Figure 2.11.

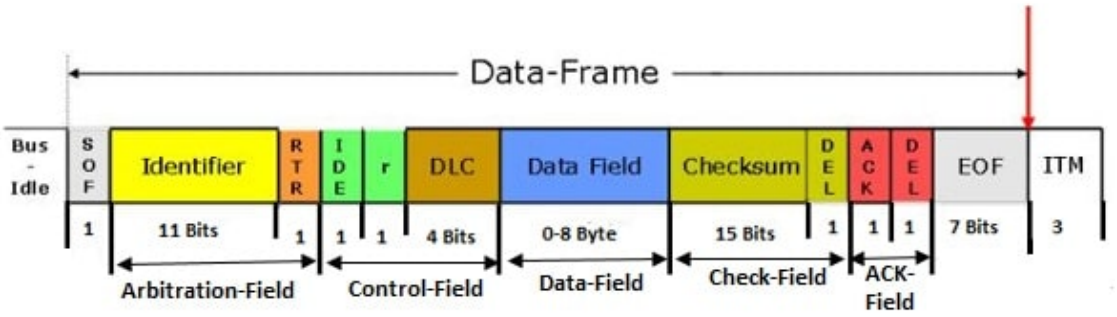


Fig. 2.11: CAN bus frame with standard identifier [25].

An important bit for CANOpen is the RTR bit which stands for Remote Transmission Request, when this bit is recessive, there are no data in the frame, and the frame asks the remote device for data. As can be seen in the Figure, the identifier and the RTR field are part of an Arbitration field, these bytes are used in the shared medium collision detection and control, and thanks to this field, frames with lower ID have a higher priority in the transmission.

2.2.3 I²C

I²C (Inter-Integrated Circuit) bus is a bus, that allows connecting multiple peripheral (slave) ICs to a controller (master) IC (multi-master mode is also supported)[26]. Hardware-wise the bus utilizes two pins in open collector configuration, which means that the high-level voltage needs to be provided externally using a pull-up resistor. The resistance value of the pull-up resistor affects the bus performance and can be fine-tuned to compensate for the parasitic capacity for the wiring. The open-collector configuration also means that when idle, the bus is pulled up to the defined voltage level, and when the device wants to transmit data, it can only pull the signal down. The I²C bus consists of two signals - the data signal (SDA) and the clock signal (SCL).

The I²C transaction begins when the controller sends a START condition on the bus, the START condition is followed by the peripheral device address, and a direction bit determining whether the controller wants to write or read from the peripheral device[26]. Then based on the direction bit, the controller either receives data or sends them. The peripheral has a mechanism of stopping the clock signal in the event the peripheral is not fast enough to produce data, which is done by pulling the clock signal low and then releasing it when the data are available. When the data is transferred, the controller finalizes the transfer by sending the STOP condition.

When interfacing with various peripherals, the read and write transactions are combined[27]. The controller first sends the peripheral address (with direction bit set to write) followed by the register it wants to access, the register is ACKed, and now the controller either sends the data to be written or issues a Repeated START, transmits peripheral address with direction bits set to read and awaits data from the peripheral. Writing to a peripheral using this approach can be seen in the Figure 2.12 and reading data can be seen in the Figure 2.13.

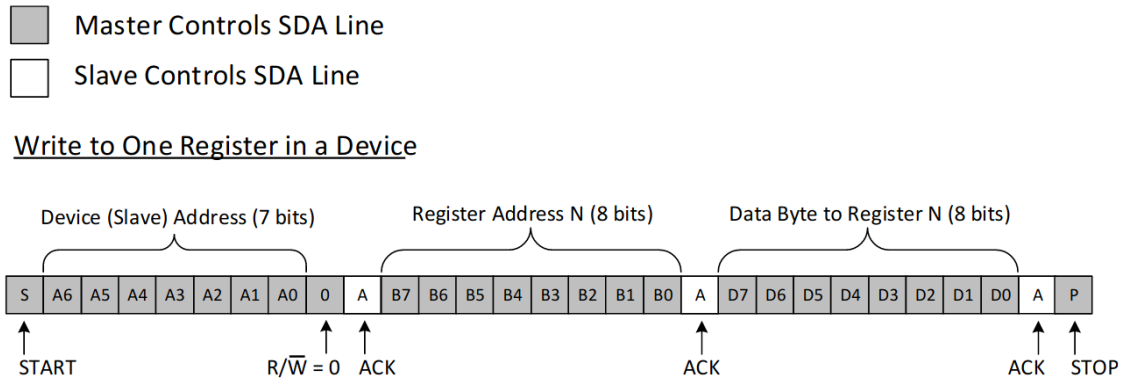


Fig. 2.12: Writing to a register of an I²C peripheral IC [27].

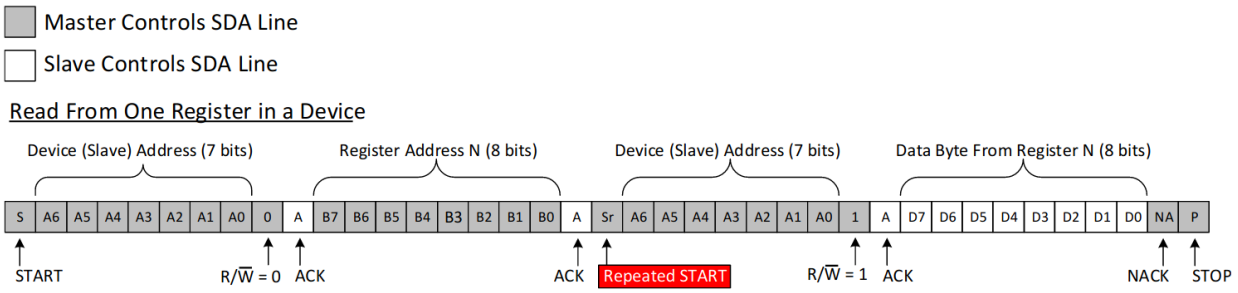


Fig. 2.13: Reading a register of an I^{2}C peripheral IC [27].

2.2.4 Universal Serial Bus

USB (Universal Serial Bus) utilizes a differential pair (or multiple) for data transfer. Nowadays, USB devices are ubiquitous and perform a variety of different functions, and according to these functions are separated into classes (Mass Storage, HID (Human Interface Devices), etc.). Through time, there were four major revisions of the USB standard as can be seen in the Table 2.2.

Version	Max. Data Rate	Code Name
1.0	1.5 Mbit/s	Low Speed
	12 Mbit/s	Full Speed
2.0	480 Mbit/s	HighSpeed
3.0	5 Gbit/s	SuperSpeed
3.1 gen 2	10 Gbit/s	SuperSpeed+
3.2	20 Gbit/s	SuperSpeed+ USB dual-line
4.0	40 Gbit/s	

Tab. 2.2: USB versions and data rates [28]

USB is nowadays everywhere, with different supported transfer data rates, different utilization of connectors, etc. With such diverse use cases and devices, the whole USB ecosystem is increasingly more difficult to orient within. For our use case, we'll be utilizing the USB controller present on the MCU, which is a USB 2.0 OTG (On The Go) Full Speed device[29], but to support better compatibility and mechanical properties, we'll be using the USB-C receptacle with both differential data lanes interconnected.

2.3 Rust Programming Language

Rust is a multi-paradigm systems programming language initially developed by Mozilla in an effort to create language suitable for the development of a safe and performant multi-threaded CSS rendering engine for the Firefox browser[30]. In recent months, the oversight of the language is done by the language's own foundation and is therefore independent of Mozilla[31].

The language itself is designed to be performant and memory-efficient - it doesn't feature a garbage collector. Memory is managed semi-manually with the leverage of many smart pointer types. The semi-automatic memory management and its type systems provide guarantees about memory and thread safety that can be evaluated at compile-time, promising that these kinds of potential bugs are found in development rather than in production.

The language itself is a part, albeit an important part, of a larger ecosystem, making the language and its tooling extremely usable, with tools almost for everything - it features seamless package management and a build system, documentation system, integrated testing, defined coding-style and more.

As we said before, the language is a multi-paradigm language, meaning that the language features parts of the functional languages paradigm and object oriented-paradigm.

In the following sections, some features of the language are described to provide some introduction into the semantics and syntax of the language.

2.3.1 Variables and Mutability

In Rust, all variables are defined as immutable by default, promoting defensive programming - meaning that no variable can be unintentionally changed. The variables are declared using the keyword **let** and the variable's mutability must be explicitly declared using the **mut** keyword. The type of variable doesn't need to be explicitly specified in most cases as the language features type inference, which is possible thanks to its powerful and strong type system. An example can be seen in the following Listing 2.1.

```
1 let a = 10; // declares an immutable variable, whose type
      is automatically inferred to i32
2 a = 11; // produces a compile-time error
3 let mut b: u8 = 0x12; // declares a mutable variable with
      explicit u8 type
4 b = 0x24; // this is ok
```

Listing 2.1: An example of declaring variables and their mutability in Rust.

Rust also supports compile-time constant evaluation using constants and constant functions. This can be achieved by using the **const** keyword, but describing this functionality is beyond the scope of this thesis.

2.3.2 Ownership and Borrow Checker

The language's semi-automatic memory management system consists of the ownership concept, move-by-default semantics, and the borrow checker.

The concept of ownership is described by the following rules[32]:

- Each value in Rust has a variable that's called its *owner*.
- There can be only one owner at a time.
- When the owner goes out of scope, the value will be dropped.

For value passing, the Rust language uses **move-by-default** semantics as opposed to **copy-by-default** present in C++. The reasoning for it is that while moving is almost zero-cost, copy almost never is.

The borrow checker is a mechanism that ensures that references to variables are always in correct state - pointing to an existing value. There are three rules to the borrow checker:

- There can be only one mutable reference to a value.
- There can be unlimited immutable references to a value.
- The first two rules are mutually exclusive - Rust forbids having both immutable and mutable reference to the same value.

The programming language also statically checks for reference lifetimes, making sure that the reference doesn't point to nonexistent memory, which is useful for returning references from functions or storing references in structs.

2.3.3 Enums and Pattern Matching

In Rust, enums are much more powerful than in C/C++. There are two significant differences - Rust enums allow adding methods and functions to them and also allow for having associated values. Consider the following code snippet:

```
1 enum Value {
2     Integer(i64),
3     Float(f64)
4 }
5
6 let int_value = Value::Integer(15);
7 let float_value = Value::Float(3.14);
8
```

```
impl Value {
    fn parse(raw: &str) -> Value {}; // implementation omitted
}
let raw_value = server.get_value();
let value = Value::parse(raw_value);
```

Listing 2.2: Definining an enum with associated values in Rust.

First, we declare the enum to have two possible values - **Integer**, with the associated value of **i64** and **Float**, with the associated value of **f64**. Then, we add a function that parses a reference to a string into our enum **Value**, and then we parse a received string into a value. The parsed Value will be one of the two values with the real numeric value embedded. Associated values in enums are a powerful concept, for example, for state machines and error handling. To access the associated value, the **match** or **if** keywords may be used as can be seen in the Listing 2.3.

```
match value {
    Value::Integer(raw) => println!("Raw integer found: {}", raw),
    Value::Float(raw) => println!("Raw float found: {}", raw),
}
if let Value::Integer(raw) = value {
    println!("Raw integer found: {}", raw);
}
```

Listing 2.3: Matching an enum variants.

2.3.4 Data Structures

The language leverages the concepts of structures to store data. These structures allow storing data with different data types. Apart from storing data, structures can have implementations associated with them which provides the ability for functions, methods, and constructors. In a broader sense, these properties conform to the object-oriented-programming paradigm where objects have properties (stored values) and behaviors (associated methods). Let's have a look at an example 2.4 of a structure definition.

```
1  // Define a structure representing a state of a motor
       axis.
2  struct AxisState {
3      pub target_velocity: f32, // define fields that are
           publicly accessible and with f32 type
4      pub actual_velocity: f32,
5  }
6
7  let mut state = AxisState {
8      target_velocity: 1.0,
9      actual_velocity: 0.0,
10 }; // create an mutable instance of the AxisState
       structure with values assigned to the fields
11
12 state.target_velocity = -1.0; // assign value to a field
       of the structure instance
```

Listing 2.4: Defining and instantiating a struct in Rust.

An **impl** block needs to be defined, to add methods to the structure. As can be seen in the following example 2.5, where we add a constructor, getter and setter methods.

```rust
// crate a block for defining methods on the AxisState structure
impl AxisState {
    // define a constructor - a method that return the AxisState structure
    pub fn new(target_velocity: f32, actual_velocity: f32) -> Self {
        Self {
            target_velocity,
            actual_velocity,
        }
    }
    // create a setter for the target_velocity, note the reference to mutable self which denotes that it is a method and not a function
    pub fn set_target(&mut self, target: f32) {
        self.target_velocity = target;
    }
    // create a getter which takes an immutable reference to the structure and returns the value of the target velocity
    pub fn target(&self) -> f32 {
        self.target_velocity // no return is needed as Rust is also an expression based language
    }
}

let mut state = AxisState::new(1.0, 0.0); // use the new function (constructor) to create an instance of the AxisState state structure
state.set_target(5.1); // set the value of the target_velocity field
println!("target velocity: {}", state.target()); // print thevalue of the target_velocity field
```

Listing 2.5: Adding methods and constructor to a struct in Rust.

2.3.5 Traits and Generics

Traits are a way to implement shared behavior (interface) for different types. Traits are similar to Java's interfaces or Swift's protocols. Together with generic types, these two features allow for creating algorithms whose inputs and outputs are generic but conform to some defined properties defined in the traits.

Let's have a look at how a motion controller can be defined and implemented using generic values in the Listing 2.6.

```
1  trait Encoder {
2      fn get_speed(&self) -> f32;
3  }
4
5  trait Motor {
6      fn set_speed(&mut self, speed: f32);
7  }
8
9  struct MotionController<E: Encoder, M: Motor> {
10     encoder: E,
11     motor: M
12 }
13
14 impl<E: Encoder, M: Motor> MotionController<E, M> {
15     fn sample(&mut self, target_speed: f32) {
16         let e = target_speed - self.encoder.get_speed();
17         // use controller to get target speed
18         let speed = psd.calculate(e);
19         self.motor.set_speed(speed);
20     }
21 }
```

Listing 2.6: Using traits and generics for shared behavior in Rust.

Such a motion controller can be used with whichever encoder and motor, that implements the **Encoder** and **Motor** traits. Traits and generics are vital for implementing HALs that are further described in the Section 2.4.

2.3.6 Macros

Another language's feature important for embedded Rust, are macros. There two types of macros in Rust - declarative macros (similar to C macros) and procedural macros, that can be used for code generation. The main distinction between C and Rust macros is that Rust macros have support for a simple type system that limits what can be passed as a function parameter - be it identifiers, expressions, etc. Macros are useful for metaprogramming - declaring code which should be generated. Many standard library features are implemented using macros. An example of a macro use can be seen in the following Listing 2.7.

```
1 let vector = vec![0.5, 0.6, 0.7]; // instantiates a
    vector with the defined values
2 println!("Value of vector is {:?}", vector); // prints
    values contained in the vector
```

Listing 2.7: Using macros in Rust to initialize a vector and print its values.

An important thing to note is that the macro processor is very capable. For example, it can evaluate values passed to them in the case of the **println** macro, which doesn't allow passing incompatible types. In embedded Rust, macros are used for generating code for different peripherals.

2.3.7 Standard Library

The Rust programming language has a rich standard library that supports widely used collections such as vectors, maps, sets, etc., communication primitives such as sockets for UDP and TCP, threads and synchronization, and much more. This makes the language ready to use out of the box, without the need to implement these primitives ourselves, which would leave room for bugs and performance problems. The following example in the Listing 2.8 shows a simple UDP communication loopback implemented using the standard library features.

```
1  use std::net::{Ipv4Addr, SocketAddrV4, UdpSocket};
2  fn main() {
3      let socket = UdpSocket::bind(SocketAddrV4::new(
           Ipv4Addr::UNSPECIFIED, 1234))
4          .expect("Failed to bind the socket.");
5      let mut buffer = [0; 1500];
6      loop {
7          match socket.recv_from(&mut buffer) {
8              Ok((len, address)) => {
9                  socket
10                     .send_to(&buffer[..len], address)
11                     .expect("Failed to send data to the
                           sender.");
12             }
13             Err(_) => {
14                 println!("Failed to receive data from the
                       socket.");
15             }
16         }
17     }
18 }
```

Listing 2.8: Using Rust standard library to implement UDP loopback.

2.3.8 Testing

The Rust programming language has support for testing built-in, meaning that no external library is needed to start writing tests for your code. Tests can be written as part of modules, which allows for testing of private members or out of the defining modules, allowing for integration testing. A simple unit testing example as a part of the defining module can be seen in the following example in the Listing 2.9.

```
1  fn adder(a: i32, b: i32) -> i32 {
2      a + b
3  }
4  #[cfg(test)]
5  mod tests {
6      use super::*;
7      #[test]
8      fn test_adding() {
9          let result = adder(1, 2);
10         assert_eq!(result, 3);
11     }
12 }
```

Listing 2.9: Writing in-module tests for Rust members.

2.3.9 Cargo

Cargo is Rust's build and dependency management system. It handles creating, building, testing, and running projects using single command without the need to call rustc and lld directly, as can be seen in the following snippet in the Listing 2.10.

```
$ cargo new sm4 --bin # creates a new Rust binary project
$ cargo new sm4-shared --lib # creates a new Rust library project
$ cargo build # builds the project in the working directory
$ cargo test # runs all the tests included in the project in the working directory
$ cargo run # runs the project in the working directory
$ cargo doc # generate documentation for the project in the working directory
```

Listing 2.10: Using cargo for project development cycle.

Apart from managing the project's development cycle, Cargo is also a dependency manager that allows for including external libraries to the project simply by specifying dependency name and version in a file called **Cargo.toml** which serves as the main configuration file of the project. An example **Cargo.toml** file can be seen in the following snippet in the Listing 2.11.

```
[package]
name = "playground"
version = "0.1.0"
authors = ["Matous Hybl <hyblmatous@gmail.com>"]
edition = "2018"

[dependencies]
parking_lot = "0.11.1"
```

Listing 2.11: An example Cargo.toml file containing project definition.

Cargo also supports other features of project management, such as enabling conditional compilation using features, etc., but the description of these features is beyond the scope of this thesis.

2.4 Embedded Rust

Embedded Rust is enabled by two crate-level attributes - **no_main** and **no_std**. The **no_main** attribute indicates that the compiler will not emit a main symbol automatically but instead expects the crate to define it itself[33]. The **no_std** attribute, on the other hand, indicates that the final program will not link against the std crate (containing functions usable with operating systems and heap allocation) but instead will link against the core crate, which contains only those parts of the standard library that are platform agnostic. The core crate includes, for example, language primitives, such as floats, slices, and support for atomic instructions. Rust program with the **no_std** attribute and therefore linking to core library can be used for bootstrapping code like bootloaders, firmware or kernels[34].

Using the **no_main** requires the programmer to write their own program entry point and some other functions, such as the reset handler and panic handlers. An example of such low-level project bring-up can be seen in [35]. Developing such low-level program can be quite hard and is usually platform-dependent. Thankfully, for the supported MCUs, this is already pre-implemented by processor low-level access crates and their startup and runtime crates.

We've mentioned that developing **no_std** does not by default use heap allocation. Heap allocation can however be added to **no_std** programs by implementing a custom allocator such as the **alloc-cortex-m** crate[36], that implements an allocator for Cortex-M MCUs. Using heap allocation, it is possible to use structures like **Vec** and **Box<T>**[37]. It also seems to be possible to run a subset of the Rust std library directly on a MCU[38], but this approach is still in the early stages of development.

2.4.1 Platform Support

To compile Rust programs for specific MCU core, it is required that target support for it is first available in LLVM and second that a target for it exists within the Rust ecosystem (existence of a target means, that the rust compiler and other tools and can be built for the target). Rust targets are also split into tiers differentiating different amounts of support from the Rust teams, where the majority of the embedded targets are in Tier 2, meaning that the compilation of the tools must succeed for every change in the compiler. Still, automated tests of the resulting builds are not guaranteed to run. As of now, there is support for different ARM architectures, such as aarch64, armv7, armv6, thumbv8, thumbv7, thumbv6 (both with hardware FPU and without) etc[39]. Apart from ARM support, there is also support for RISC-V targets, Tier 3 also contains support for AVR[39, 40] and MSP430. There also seems

to be some effort into bringing the Xtensa architecture (ESP8266 and ESP32)[41] into the Rust ecosystem as the work of supporting the architecture in LLVM seems to be done.

2.4.2 Embedded Working Group

Embedded Rust is one of the official goals of the Rust Language project, and the development of Rust for embedded devices is governed by the Embedded Devices Working Group[42]. Apart from target maintenance, the working group is directly responsible for creating and maintaining documentation - such as the Embedded Rust book[34], the Discovery book[43] and the Embedonomicon book[44] and is also responsible for developing several of the critical tools and libraries - such as **svd2rust**, **embedded-hal** or **embedded-dma**.

The working group also governs the work on low level MCU core access crates and minimal runtimes for these cores such as the **cortex-m** and **cortex-m-rt** or **risc-v** and **riscv-rt**.

2.4.3 Register Access

Peripheral and core register access is one of the most vital operations on embedded systems. Generally, in C/C++, this is done either by operating on a pointer to an integer value or by modifying a struct that contains the configuration. In embedded Rust, there are two approaches to this problem.

The first approach is the **svd2rust**[45] tool. This tool uses the manufacturer-provided SVD files that describe registers and their bits and converts them into a Rust API. There are several advantages to this approach - no code needs to be written manually, the resulting register access is compile-time safe, allows for named bit settings instead of non-descriptive setting to logical one or zero, and when there is ambiguity, for example, in integer size, it enforces adding the unsafe keyword, marking that the code should be thoroughly reviewed. An example of setting register values using this approach can be seen in the following Listing 2.12, where we are configuring channel of an advanced control timer. The disadvantage of this approach is that given the size of the SVD files, and the number of MCUs, the compilation is quite long, and given the resulting API complexity, the contemporary IDEs have problems resolving it for automatic code completion. This approach is prevalent with the majority of the PACs (Peripheral Access Crates), for example with the **stm32-rs** crate[46].

```
timer.ccmr2_output().modify(|_, w| {
    w.cc4s()
        .output() // channel 4 as output
        .oc4fe()
        .set_bit() // enable fast output
        .oc4pe()
        .set_bit() // enable preloading
        .oc4m()
        .toggle() // set mode (3bit wide register part)
});
```

Listing 2.12: Using svd2rust generated API for register access.

Another and more lightweight approach is utilization of RAL (Register Access Layer) crates, for example the **stm32ral**[1]. This approach also parsed the SVD files, but generates simpler structures, and API. An example of using this approach can be seen in the Listing 2.13.

```
modify_reg!(rcc, rcc, AHB1ENR, GPIOAEN: Enabled);
modify_reg!(gpio, gpioa, MODER, MODER1: Input, MODER2:
   Output, MODER3: Input);
```

Listing 2.13: Using RAL API for register access[1].

2.4.4 embedded–hal

The **embedded-hal** project[47] is one of the projects developed by the Embedded Devices Working Group. Its aim is to provide a HAL (Hardware Abstraction Layer) abstract enough that device drivers using it may be shared not only between different MCUs but also different platforms (MCU and Embedded Linux). This is achieved by utilizing traits described earlier in the Section 2.3.5. An example of such device driver developed using the traits available in the **embedded-hal** can be seen in the Listing 2.14[47]. As can be seen in the Listing, the driver will work with any type that implements the **WriteRead** trait. Currently, the **embedded-hal** defines traits for common embedded functionality (such as GPIO, ADC) and buses (I^2C, SPI), but only their blocking variants.

```rust
use embedded_hal::blocking::i2c::{TenBitAddress,
    WriteRead};
const ADDR: u16  = 0x158;
const TEMP_REGISTER: u8 = 0x1;
pub struct TemperatureSensorDriver<I2C> {
    i2c: I2C,
}
impl<I2C, E> TemperatureSensorDriver<I2C>
where
    I2C: WriteRead<TenBitAddress, Error = E>,
{
    pub fn read_temperature(&mut self) -> Result<u8, E> {
        let mut temp = [0];
        self.i2c.write_read(
                ADDR,
                &[TEMP_REGISTER],
                &mut temp
            )
            .and(Ok(temp[0]))
    }
}
```

Listing 2.14: Example of an device driver utilizing embedded–hal traits.

The traits from the **embedded-hal** are implemented by device HALs, such as the **stm32f4xx-hal**[48], which implements the provided traits for the peripherals of the STM32F4xx MCUs.

A crate similar to the **embedded-hal** is the **embedded-dma** defining traits for unified DMA access, so that development of unified device drivers utilizing DMA is possible. DMA itself is not that easy problem when aiming for Rust's memory safety[49], but a lot of work has been done on its support and perfecting it for safety.

2.4.5 Mutable Shared State

The Rust programming considers mutable static variables unsafe as they may cause data races in concurrency[50]. This approach is valid but causes ergonomics problems with embedded programs where storing data in static variables is used for interchanging data between interrupts and blocking code. In order to write correct, data race free code, the shared state needs to be at least wrapped in a mutex that will synchronize access to the shared resource. This, however is not enough as the

borrow checker will object with invalid access to a shared resource. Also, when using mutex on embedded systems, the block holding the lock need to be wrapped in a critical section. Otherwise, the lock could never be freed. The situation gets more complicated with different interrupt priorities. An overview of ways of solving this problem can be found in an excellent blog post by one of the members of the Embedded Devices Working group here[51].

We decided to use the RTIC (Real-Time Interrupt Driven Concurrency) framework[52] as a solution to this problem as according to us, the approach is very well thought out. The framework aims to solve the problem by providing a DSL (Domain Specific Language) based on Rust's procedural macros for concurrency on embedded systems. In short, the RTIC framework lets the user define shared resources and allows different tasks (either software tasks or interrupt triggered ones) to access those resources in a safe way, removing boilerplate code needed for shared resource handling. Imagine that there are two interrupt triggered tasks with different priorities wanting to access a shared resource. The high-priority task can easily access the resource without the need for any explicit locking as it has the privilege to do so, given its priority. On the other hand, the lower priority task needs to perform locking in order to access the shared resource to avoid the resource being accessed temporarily by the higher resource task, which could lead to a data race. This behavior is compile-time enforced and checked, bringing another level of safety and security to embedded systems. Apart from providing safe access to shared resources, this framework also contains a simple scheduler for timed tasks. Thanks to these features, this framework is nowadays the base of many embedded systems developed in Rust.

2.4.6 async/await

Async/await is a concurrency model utilizing cooperative tasks. Cooperative tasks are tasks that yield control back to the scheduler at determined points[53], and these points are when task switching occurs. Tasks can yield, for example, to wait for a blocking operation.

This concurrency model may prove extremely useful for embedded systems as it allows for concurrent code to be written in an imperative way, which increases code readability. Imagine a device driver task that resumes its operation when new data is received and doesn't use any blocking waits. The code can be read as a sequence of operations instead of, for example, a state machine.

Async/await in the Rust programming language builds upon the **Future** trait. A future can be polled, and polling results in reporting whether the **Future** can resume executing or not. An executor takes care of periodically (or as a reaction to

an event) polling **Futures** and resumes their execution if needed. Given the design of async/await, this can be easily used in a **no_std** environment, even on low computing power devices, such as the Atmel AVR[54]. As of now, there are several async runtimes for embedded systems, the most advanced one being **embassy**, the downside being that it currently requires nightly Rust to build.

An example of utilizing async/await for timed tasks, and **embassy** can be seen in the Listing 2.15

```
#[embassy::task]
async fn run1() {
    loop {
        info!("BIG INFREQUENT TICK");
        Timer::after(Duration::from_ticks(64000)).await;
    }
}
#[embassy::task]
async fn run2() {
    loop {
        info!("tick");
        Timer::after(Duration::from_ticks(13000)).await;
    }
}
#[embassy::main]
async fn main(spawner: Spawner) {
    unwrap!(spawner.spawn(run1()));
    unwrap!(spawner.spawn(run2()));
}
```

Listing 2.15: Timed tasks using async/await and embassy[2].

2.4.7 Ecosystem

The Embedded Rust ecosystem is nowadays very vast. Apart from the Embedded Devices Working Group, there are many smaller organizations governing development for specific platforms or projects. There are organizations and groups developing crates for MCU support (STM32, RISC-V, AVR, RP2040, iMXrt, SAMD, etc.), universal device drivers (using the embedded-hal described earlier), and much more. The ecosystem also consists of tooling and documentation, which is also under active development, with some tools surpassing traditional tools for embedded development. More importantly, given Rust's focus on performance and security,

eliminating most common bugs, the developers are able to work on the "harder" problems in embedded - for example, drivers for USB[55] or Ethernet[56].

2.4.8 C/C++ Interoperability

Thanks to Rust's FFI (Foreign Function Interface), it is possible to combine code written in C and Rust, even both ways[57]. For example, it is possible to utilize Rust's serialization and deserialization crates in an existing Zephyr project[58]. This is especially useful for bringing Rust into the existing embedded codebase, but also for integrating existing the C/C++ codebase into new Rust project.

2.4.9 Tooling

We've already discussed some of the Rust Embedded Tooling, such as the **svd2rust** in previous sections, but there are more tools at hand helping with the development cycle. In general, in C/C++ embedded projects, the firmware is flashed via a combination of OpenOCD and GDB. That used to be the case with Embedded Rust too before the **probe-rs** project was developed. The project aims to provide "A debugging toolset and library for debugging embedded ARM and RISC-V targets on a separate host"[59], this is achieved by developing a library for various probes (ST-Link, J-Link) handling the low-level communication with the target MCUs and then building tools upon that library, that allow for flashing and debugging the target MCUs. Those tools are called **cargo-flash**[60], enabling just flashing, and the **cargo-embed**[61], enabling flashing, debugging and logging. The **probe-rs** project is the project that implements Segger RTT (Real-Time Transfer) for seamless and fast data transport between the host and the target MCU that can be utilized for fast logging.

Thanks to **probe-rs** architecture, other tools can utilize its algorithms for accessing probes and the target **mcu**s. That is what happened with the Knurling project - an initiative to provide modern tooling for embedded development. At the moment, there are four crucial tools developed by the Knurling project - **probe-run**, **defmt**, **defmt-test** and **flip-link**.

defmt

defmt stands for deferred formatting[62], which means that the data coming from the MCU is formatted when it is received by the host. This is utilized for logging data from the MCU. The tool strips all strings from the firmware binary and replaces them with their identifier. Instead of transferring the whole strings, only their identifiers and optional arguments (such as values of variables) are transferred,

saving time and resources, making the logging extremely efficient. This logging technique relies on the fast RTT transfer from **probe-rs** described earlier. The end result is a logging interface that is fast and efficient enough to debug extremely fast and timing-sensitive interfaces, such as USB.

probe–run

probe-run is a runner for Rust programs[63], that automatically flashes the firmware binary when **cargo run** is run and opens a terminal which prints **defmt** encoded messages transferred over RTT. Another feature is that it can print stack backtraces when the firmware encounters an exception or on a breakpoint.

As of now, the runner doesn't support debugging via GDB, but there have been some developments recently about simultaneous defmt logging and debugging with GDB that stated that GDB support should be possible in the near future[64].

defmt–test

defmt-test is a component of the **defmt** tool, that can be used for running integration and unit tests directly on the target MCU. We believe that this brings new possibilities for reliability in embedded systems and software/hardware-in-the-loop testing. An example of such test can be seen in the Listing 2.16[62].

```
struct State {
    scd30: Scd30<Twim<TWIM0>>,
}
#[defmt_test::tests]
mod tests {
    use super::State;
    #[init]
    fn setup() -> State {
        // initialize the hardware (ommited)

        let scd30 = scd30::Scd30::init(i2c);
        State { scd30 }
    }
    #[test]
    fn confirm_firmware_id(state: &mut State) {
        const EXPECTED: [u8; 2] = [3, 66];
        let firmware_id = state.scd30.
           get_firmware_version().unwrap();
        assert_eq!(EXPECTED, firmware_id);
    }
}
```

Listing 2.16: Integration test written using defmt–test.

flip–link

The last Knurling tool is **flip-link**, which is a linker that adds zero-cost stack overflow protection[65]. The stack overflow protection works by smartly changing the order of **.stack** and **.bss+.data** segments of RAM in a way that when the stack overflows, it doesn't interfere with static variables but instead triggers a hardware exception and doesn't produce corrupted memory. The change in the memory layout can be seen in the Figures 2.14 and 2.15.

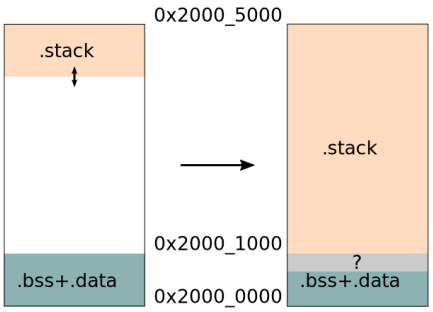


Fig. 2.14: Memory arrangement which causes memory corruption on stack overflow[65].

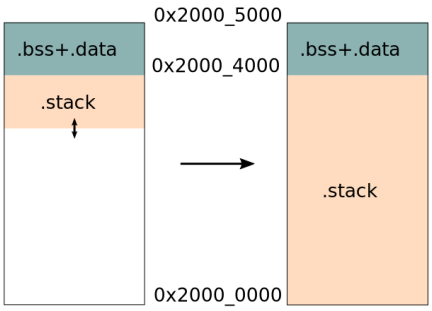


Fig. 2.15: Memory arrangement which causes hardware exception instead of corrupted memory[65].

2.4.10 Testing and CI

While we already talked about testing in the Sections 2.3.8 and 2.4.9, we believe that there is more information about testing embedded Rust software and CI (Continuous Integration).

There are two different types of tests that can be run for embedded software - tests that can be run without the hardware and tests that must be run on the target hardware as the code is dependent on it. The tests that are independent of the target hardware can be tested using the standard Rust testing harness via, for example, **cargo test**, on the other hand, to run these tests, a target that supports **std** is required, which might require some project structure changes. As for the tests that must run on the target hardware, the **defmt-test** test harness can be utilized. Approaches for testing these parts of the embedded firmware can be found here[66, 67]. Hardware in the loop-testing is also possible[68].

Nowadays, it is also possible to run the tests as part of the project's CI workflow. Apart from running tests, it is also possible to easily build the firmware itself as the only thing required is the Rust compiler with proper toolchain installed[69]. It is also possible to release the firmware, generate documentation and perform additional code checks on the codebase[70].

2.5 Embedded Rust Advantages and Disadvantages

Many of the reasons why we believe that Embedded Rust is the right choice for embedded development have already been described in the previous sections, but let us sum up what we believe are the most important points.

- The Rust language promotes and ensures code that is not prone to memory safety problems and data races.
- The Rust language ecosystem is large, and integration of libraries is seamless and promotes code reuse.
- There are advanced and modern features in the language, making it easier to write safer and more expressive code.
- A lot of the abstractions provided by the programming language are zero-cost, meaning that they consume no additional resources.
- The embedded Rust ecosystem is vast, and contains well-thought-out solutions to difficult problems in embedded - see Section 2.4.5 on RTIC.
- The embedded Rust teams seek to utilize the modern programming features, such as async/await and generics - see Section 2.4.6 on async/await.
- Some of the harder parts embedded are already developed and ready to use - such as USB and Ethernet drivers, with the community aiming to solve these hard problems having a language that helps mitigate common programming mistakes.
- The embedded tooling is gradually improving, and in some ways it has already surpassed tooling in classical embedded languages - see the Section 2.4.9 on the tooling.
- Testing embedded systems with the Rust language is developed, and testing the firmware is easy, even in the CI.

On the other hand, there are some disadvantages of using embedded Rust:

- The resulting binaries can be quite large (10s of kB), making it harder to run on lower-end MCUs.
- The support for debugging using GDB exists but is not very ergonomic.
- The support for embedded in commonly used IDEs is low, and sometimes, for example, the automatic code completion of registers can fail.

2.6 Requirements

In the book Better Embedded System Software[71], the author specifies written requirements as one of the most important parts of documentation for any embedded system. The author describes requirements as rules specifying everything the system must do, everything the system must not do, and constraints the system must meet. According to the book, there are three types of requirements - functional requirements, non-functional requirements, and constraints. An essential property of requirements is that they must be easily verifiable and, if possible, directly measurable. Also, for future reference, every requirement must have a unique number.

2.6.1 Functional Requirements

Functional requirements describe properties that must be provided by the target system. These are implemented either by the firmware or the hardware.

The requirements for the stepper-motor controller are specified as follows:

- **FR-01** When no command specifying motor speed is received for a period of time (configurable, e.g., 1 s), the controller shall stop both motors.
- **FR-02** When multiple communication interfaces are connected, the system shall prioritize CAN bus, then I2C. USB has the lowest priority.
- **FR-03** The controller shall set motor current based on the ramping state. When the motor is still, low current shall be set; when the motor is accelerating, high current shall be set; when the motor is moving with constant speed, the current shall be set to some medium values. These values shall be configurable.
- **FR-04** All relevant values (currents, timings, limits, etc.) shall be configurable via USB or CANOpen SDO protocol.
- **FR-05** The controller shall be able to ramp the speeds using at least trapezoidal ramps, and their parameters shall be configurable.
- **FR-06** The controller shall support control in speed mode as well as position mode.
- **FR-07** The controller shall provide basic electrical safety features - such as fuses and reverse voltage protection.
- **FR-08** The controller shall have an interface allowing to connect external encoders (absolute or incremental) during future development.

2.6.2 Non-functional Requirements

Non-functional requirements are properties that the system must have but are not directly features or functions but rather properties of the system as a whole.

- **NFR-01** The controller shall provide developer-friendly protocol and data formats.
- **NFR-02** The controller shall be programmable without programmer, ideally using DFU (Device Firmware Update).
- **NFR-03** The controller shall be configurable using a program for personal computers.
- **NFR-04** The firmware shall be easily extensible.
- **NFR-05** The firmware shall employ unit testing for QA (Quality Assurance).
- **NFR-06** The firmware should utilize software in the loop integration testing for QA.
- **NFR-07** The firmware shall be properly documented.
- **NFR-08** The functionality of the controller shall be demonstrated using two distinctive applications.

2.6.3 Constraints

Constraints specify limitations on how the system must be built. They specify, for example, hardware limitations, technologies, protocols, conformance to standards, etc.

- **C-01** The controller shall utilize stepper drivers with silent operation, such as Trinamic StealthChop™.
- **C-02** The controller shall be able to drive motors with phase current of up to 2 A RMS.
- **C-03** The controller shall feature CAN bus, I2C bus and USB.
- **C-04** The controller shall utilize two stepper motor drivers.
- **C-05** The controller shall adhere to the CANOpen protocol.
- **C-06** The controller hardware shall be small, ideally smaller than the Raspberry Pi SBC.
- **C-07** The firmware for the controller shall be developed using the Rust programming language.

2.7 Electronics Design

This section describes the electronics of the SM4 stepper motor controller. First, the critical components are selected, and preliminary design is done based on the requirements stated in the Section 2.6. Second, the design of the electronics is described. There were two electronics revisions, where different stepper motor driver IC was used in each of them. In the description, the second revision is prioritized except for the design of the circuit of the stepper motor controller IC where both of the revisions are described.

In the Figure 2.16, we can see the block diagram of the controller. The central part is the MCU, which is connected to the two stepper motor drivers and encoders (hardware encoders are only available in the second electronics revision). Further, there are also peripheral chips and components used to utilize the CAN, I^2C and USB buses. The whole controller is powered by the power system providing correct voltages and electrical protection.

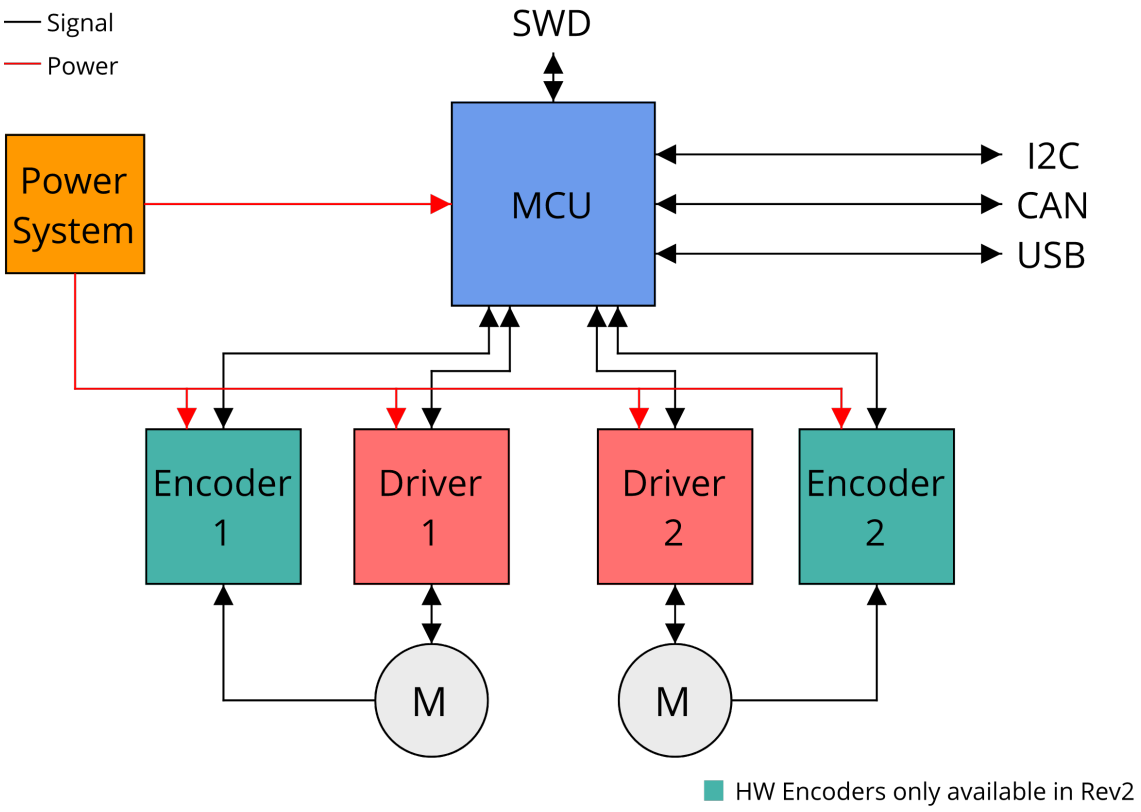


Fig. 2.16: The block diagram of the SM4 stepper motor controller.

2.7.1 Hardware Design Choices

In this section, we describe the choices made at the beginning of the design process, ones that are vital to the functionality of the whole system. The design choices are based on the requirements stated in the Section 2.6, the related work described in the Chapter 1 and our prior experience.

MCU

The most critical component of the stepper controller is the MCU. The MCU needs to accommodate for the outer communication interfaces as well as the internal ones. That means that as for the outer communication interfaces, it needs to have peripherals for CAN bus, I^2C and USB, as stated in the requirement **C-03**. The internal communication interfaces are revision dependent, however, the stepper controllers generally require GPIOs, PWM outputs, serial interfaces, and for the future encoder support, it should require incremental encoder interfaces and SPIs for SSI bit-banging (as stated in requirement **FR-08**). As was described in the Chapter on Related work 1, we decided to move from Cortex-M0 and Cortex-M0+ based ARM MCUs to more powerful Cortex-M4 MCUs. The biggest advantage of these cores is that they fully support atomic instructions, improving memory safety in ISRs, and also that they have FPU (Floating Point Unit).

Given the past experience with the STM32 family of ARM microcontrollers, we decided to select the STM32F4 product line, more specifically with the STM32F405RGT6, which features one megabyte of flash and 192 kilobytes of RAM and can be run with the 168 MHz clock[72]. The block diagram of the MCU with the core features and peripherals can be seen in the Figure 2.17.

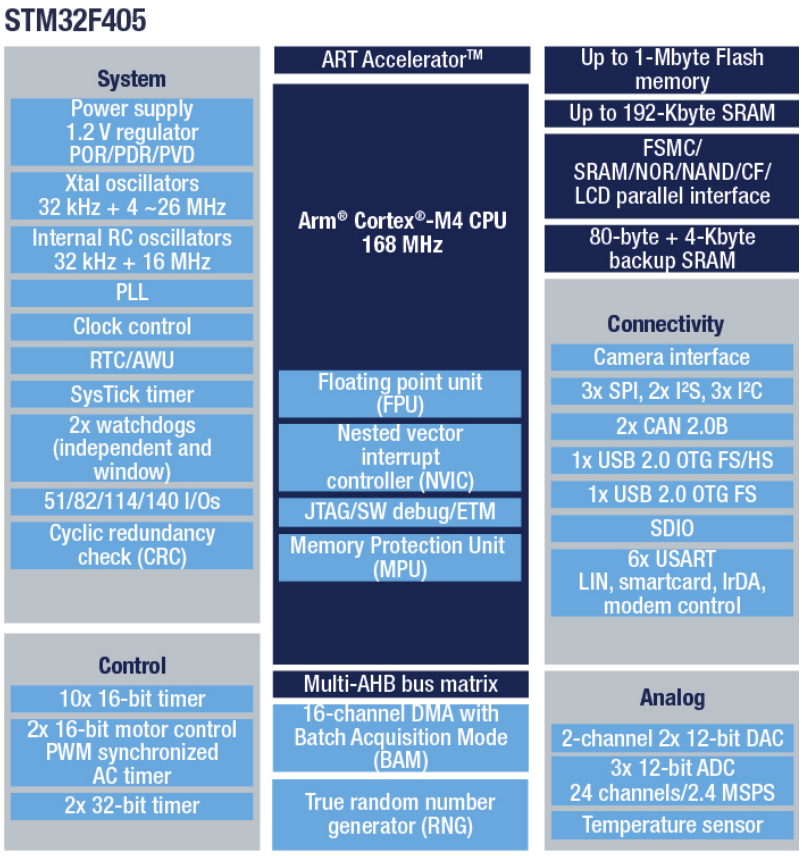


Fig. 2.17: The block diagram of the STM32F405RG MCU [73].

This MCU conforms to all the requirements and has enough peripherals to support future development.

2.7.2 Stepper motor driver IC

We performed a simplified comparison of stepper motor driver ICs in order to select a proper one for the SM4 stepper motor controller. Since the beginning, we were aiming at using the Trinamic made stepper motor driver ICs since there are very good references from the 3D printing community[74, 75, 76] for them, and we wanted to secure the silent function of the motor controller. With the prior experience with DRV8825 and the A4988 modules described in the Chapter on Related work 1, we decided to select two drivers from Trinamic - the first hardware revision utilizes the TMC2100-TA as we aimed for the as simple IC as possible and also maximum voltage of about 43 V, while the second hardware revision utilizes the TMC2226-SA, which is a more modern, and a fully-featured stepper motor driver IC, which also has higher peak current. By selecting these two stepper driver ICs, we are conforming to the requirements **C-01** and **C-02**. The comparison can be seen in the following table:

Name	Operating Voltage	Maximum current	Control Interface	Configuration Interface	Microstepping	Package	Advanced Features
DRV8825[77]	8.2-45 V	max 2.5 A (properly cooled, at 24 V, 25 °)	STEP/DIR	GPIO	up to 32	HTSSOP28	None
A4988[78]	max 35 V	max 2 A	STEP/DIR	GPIO	up to 16	QFN28ET	None
TMC2100[79]	max 46 V	max 2 A (2.5 A peaks, properly cooled)	STEP/DIR	GPIO	up to 256	TQFP48 / QFN36	MicroPlyer™, SpreadCycle™, StealthChop™
TMC2130[80]	max 46 V	max 2 A (2.5 A peaks, properly cooled)	STEP/DIR	SPI	up to 256	TQFP48 / QFN36	MicroPlyer, SpreadCycle™, StealthChop™, ChopSync™, CoolStep™, StallGuard™
TMC2209[81]	max 29 V	max 2 A (2.8 A peaks)	STEP/DIR	UART	up to 256	QFN28	MicroPlyer™, SpreadCycle™, StealthChop2™, CoolStep™, StallGuard4™
TMC2226[82]	max 29 V	max 2 A (2.8 A peaks)	STEP/DIR	UART	up to 256	HTSSOP28	MicroPlyer™, SpreadCycle, StealthChop2™, CoolStep™, StallGuard4™

Tab. 2.3: Comparison of stepper motor driver ICs

SM4 power design

The power design of the SM4 stepper motor controller is fairly simple. According to the requirements, the only requirement for it is to provide basic electrical safety features, such as fuses and reverse voltage protection **FR-07**. The controller features two power rails - one for the power electronics that can utilize quite high voltages and one 5 V for the MCU and the peripheral circuits. With the first revision, we were considering using a single power rail with all voltages derived from the power electronics one. This was, however, dismissed as a buck converter from quite a high voltage would be required, and designing a buck converter is out of the scope of this project and also the motor controller was never meant to be used as a standalone device, meaning that another device could provide the power for the 5 V rail. The buck converter would also pose EMI (ElectroMagnetic Interference) problems and would increase the price of the motor controller. In the end, the 5 V power rail is properly fused, filtered, and given that the voltage may come from different sources, also merged using diodes. The power electronics rail, on the other hand, is only filtered. A block diagram depicting the power system can be seen in the Figure 2.18.

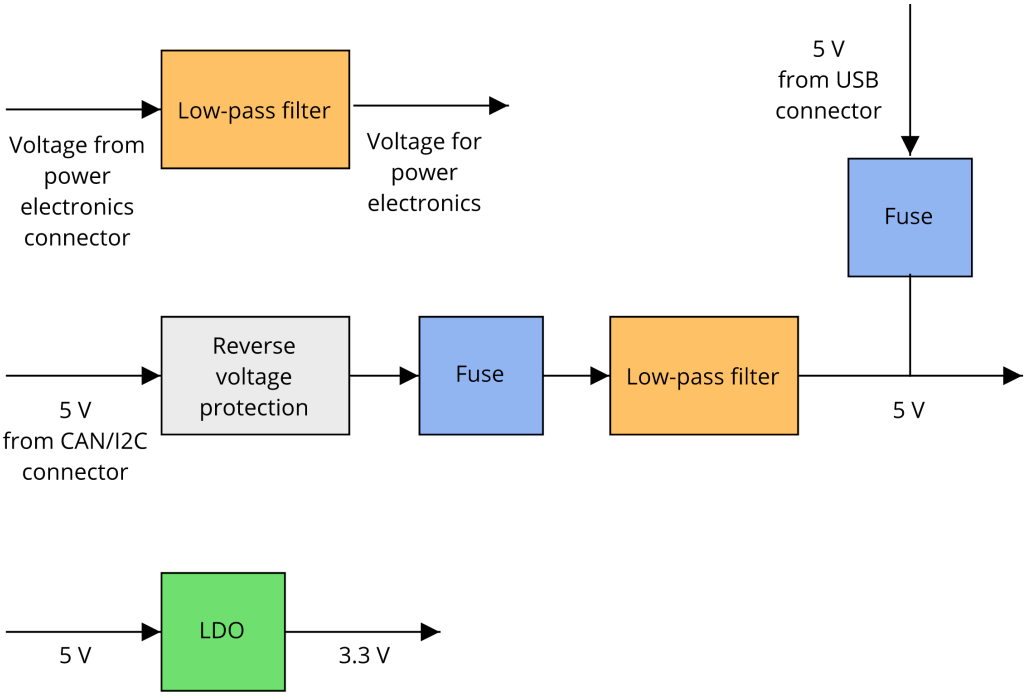


Fig. 2.18: The block diagram of the power system.

PCB

In order for this project to serve as a testbed for new manufacturing technologies, the PCB (Printed Circuit Board) was designed as a 4-layer one. The ability to design the board as a 4-layer one was enabled by the 4-layer PCB manufacturing price decrease by China-based PCB manufacturing companies. Big advantage of designing the PCB as 4-layer one was speedup of hardware development - the 4-layer stack up can be utilized so that there is no need to route power to the ICs. In our case, we chose the inner layers to be filled with copper planes - one connected to GND and the other one connected to +3.3 V. This way, whenever a connection to +3.3V or GND was required, simply connecting the pad to new via close-by was sufficient. Apart from being used for power distribution, the large copper planes allow for better PCB cooling and some minor signal connections in cases routing using the outer layers would prove difficult. The used stack up can be seen in the Figure 2.19.

Fig. 2.19: The 4-layer PCB stackup.

Another way to test manufacturing capabilities was utilizing the automated assembly service provided by the China-based PCB manufacturers. This not only saved a lot of time with manual assembly but also enabled us to use smaller components than before - the imperial size 0402.

As for testing out EDA (Electronic Design Aid) software, the KiCAD EDA was used instead of the well-known Eagle. The KiCAD EDA has improved dramatically in the past years (version 5 and soon to be released version 6), making it a great alternative to conventional EDA suites. The big advantage of KiCAD is a large footprint and symbol library, which often contains even the 3D models, and KiCAD itself is able to seamlessly integrate them and render a 3D view of the designed PCB.

2.7.3 The MCU and its Auxiliary Circuits

Designing the MCU and its auxiliary circuits is based on the STM32F405 datasheet[83] and on an example design by Philip Salmony described in a YouTube video[84]. Apart from different stepper motor driver IC connections, there are only minor differences between the hardware revisions.

The design follows the video closely, as its development approach seems sound. First, the preliminary design is done using the STM32CubeMX tool. The tool allows users to select the target MCU and graphically assign alternate functions to pins (each GPIO pin on an STM32 features multiple alternate functions, connecting for example SPI bus to the pin). Given that some internal MCU peripherals can be connected to multiple GPIOs and alternate functions, it is also possible to optimize the design and wiring based on the preliminary layout of the PCB by changing alternate function assignments. The assigned alternate functions to the MCU for the second revision of the hardware can be seen in the Figure 2.20. The STM32CubeMX is also capable of setting up the peripherals and can generate initialization code for the peripherals and the whole C/C++ project with everything set up.

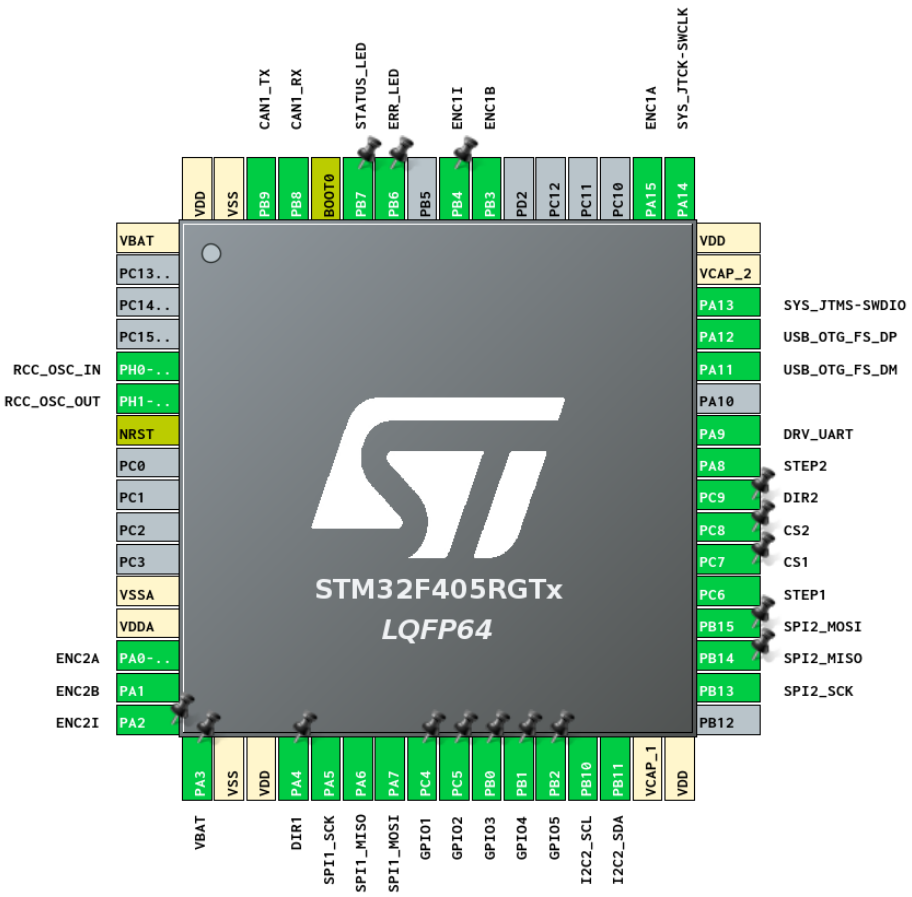


Fig. 2.20: Designing the MCU connections using STM32CubeMX.

After the pin and alternate functions assignment was done, we used the information from the datasheet to design the MCU circuitry itself. The electronic schematic can be seen in the Figure 2.21, there are not that many components in this part of the schematic. There are only two **VCAP** capacitors with values taken from the datasheet [83]. Furthermore, there are net labels connected to other parts of the schematic.

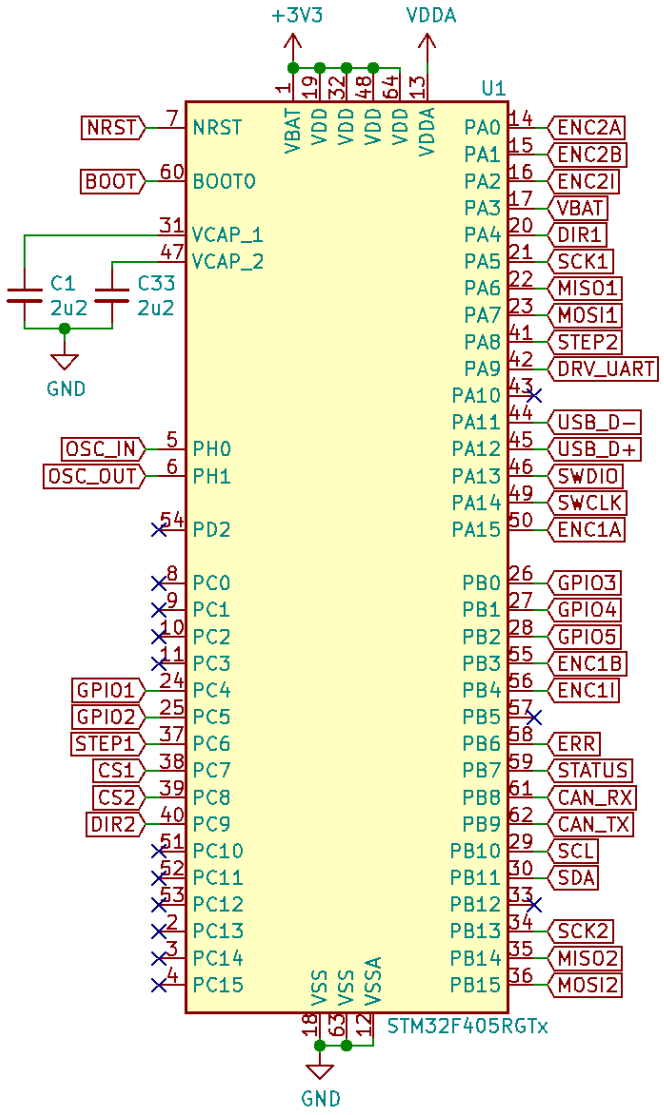


Fig. 2.21: The MCU schematic.

More interesting parts of the MCU schematic are the auxiliary circuits shown in the Figures 2.22 and 2.23. The Figure 2.22 depicts the power supply filtering circuits. As per the datasheet[83], for each **VDD** there should be a 100 nF capacitor, and there should be a single 4.7 uF capacitor on the rail.

As for the analog filtering circuit, a simple low-pass filter utilizing a ferrite bead was employed, as can be seen in the Figure 2.22, the circuit was taken from the example project[84].

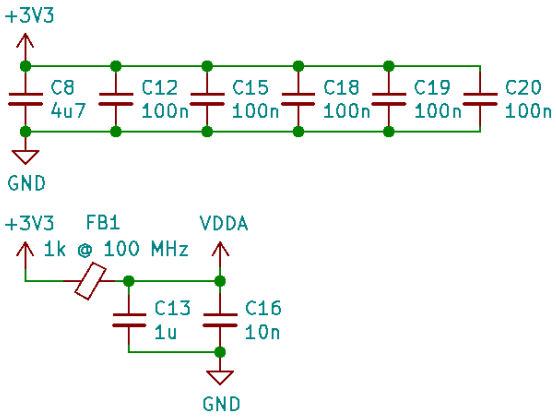


Fig. 2.22: The MCU power supply filtering schematic.

The Figure 2.23 shows other MCU auxiliary circuits - boot mode selection, reset and external crystal oscillator. The boot mode selection consists of an SPDT (Single Pole Double Throw) switch that connects the **BOOT0** GPIO to either 3.3 V or GND through a 10 kΩresistor. Pulling the **BOOT0** down to GND enables normal boot, where the MCU starts the controller's firmware, on the other hand pulling the pin up to the 3.3 V makes the MCU boot its bootloader, and it can be programmed using the DFU protocol via USB.

The reset pin on the schematic is connected to a jumper with a small filtering capacitor of 10 nF for debouncing. A pull-up resistor required for the reset to work is already included in the MCU therefore, an external one is not required.

The external crystal clock is required for USB to work. In our case, we utilized the oscillator circuit used in the example project, where we kept the value of the load capacitor of 12 pF, ignoring the stray capacity of the PCB, for improved performance of the oscillator, the following equation may be utilized to compute the load capacitance.

$$C_{load} = 2 * (C_{load-datasheet} - C_{stray}) \tag{2.1}$$

In the equation, the $C_{load-datasheet}$ is the load capacitance specified by the datasheet, and the C_{stray} is the stray capacitance of the PCB. The feed resistor value was also used as it was in the example project, but the specific value calculation can be found in the Application Note AN2867[85].

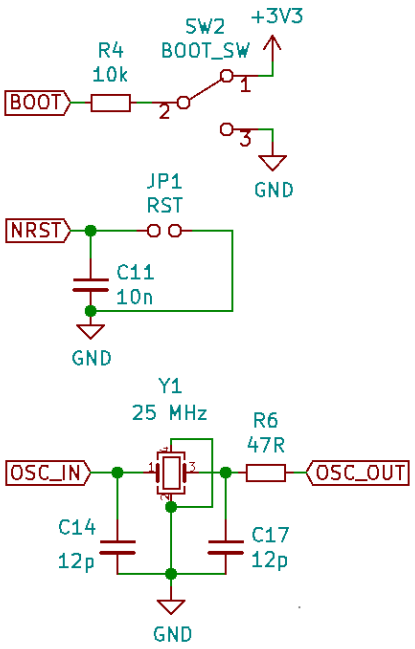


Fig. 2.23: Auxiliary circuits for the MCU - boot mode, reset and oscillator.

2.7.4 Power System

As was discussed in the Section 2.7.2, the power system consists of two rails - the power electronics rail and the 5 V rail for the MCU and supplementary circuits. In both cases, the rail should be filtered and should employ some safety features.

As for the power electronics rail, only input voltage filtering using capacitors was utilized. The main reasoning is that this should be fused on the power source side and that reverse-voltage protection would require quite large components.

The situation is different with the 5 V power rail for peripherals and the MCU. This power rail utilizes 500 mA PTC fuse, reverse-voltage protection implemented using P-channel MOSFET, and a low-pass filter comprising of a ferrite bead and a capacitor. This power rail is connected to the connectors with CAN bus and I^2C. The output of the filtered power rail is merged with a 5 V power coming from the USB-C connector (which is also fused using a 500 mA PTC fuse) using Schottky diodes. For powering the MCU with 3.3 V, the 5 V is regulated with an LDO (Low-Dropout) regulator. The whole power rail can be seen in the schematic in the Figure 2.24.

In the future revisions, the input protection circuits may be replaced by an eFuse[86, 87], an IC integrating the input power protection circuits such as overvoltage protection, undervoltage protection, overcurrent protection and reverse-voltage protection.

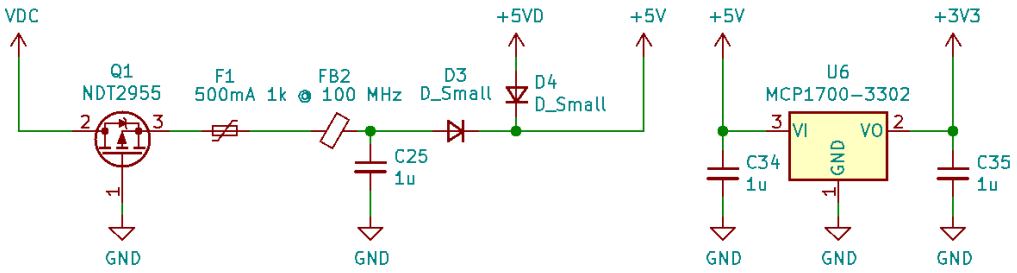


Fig. 2.24: The 5 V power rail for powering the MCU and peripherals.

2.7.5 CAN Bus Circuitry

In order to transmit CAN frames over the CAN differential pair, a transceiver that converts digital signal from the MCU to the differential signal needs to be utilized. We utilized the MCP2562 transceiver because we needed it to support 3.3 V logic levels for the MCU and 5 V levels for the differential pair, and we also had previous experience with the transceiver. In the future revisions, this transceiver will most likely be replaced by the newer MCP2562-FD transceiver as the currently used one is "Not Recommended for New Designs"[88].

The schematics of the transceiver circuits can be seen in the Figure 2.25. On the left, we can see a resistor network that can be utilized to support other transceivers as they are commonly pin-to-pin compatible except for the **Vio** and **STBY** pins that usually have a different function. On the right, we can see the bus termination resistor with a jumper and two connectors to connect the SM4 controller with other circuits. The capacitors on the bottom are used for power rail filtering.

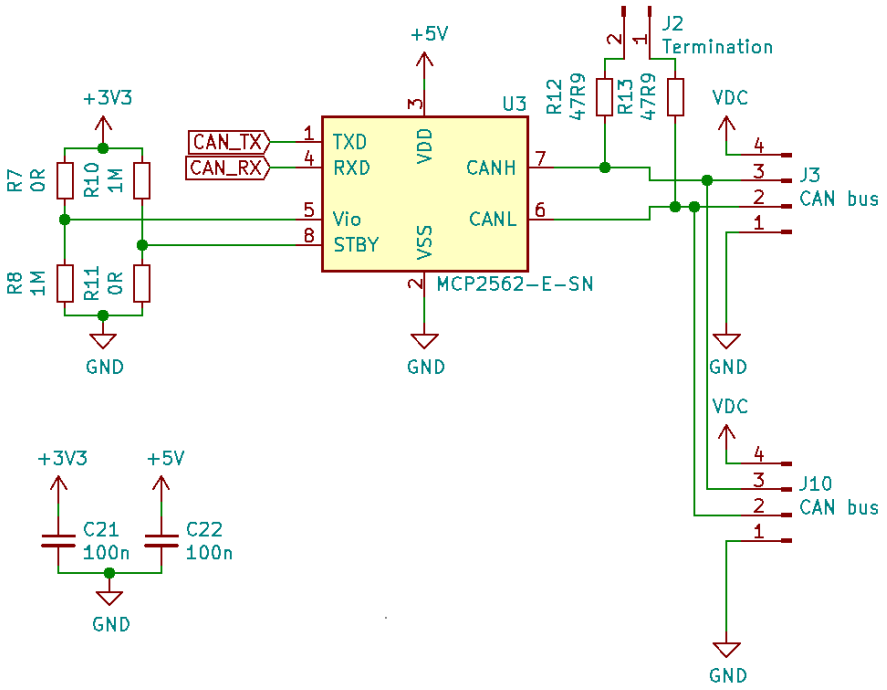


Fig. 2.25: The schematic of the CAN bus transceiver circuitry.

2.7.6 USB Circuitry

For configuration and flashing purposes, a USB connection was added. The schematic for it can be found in the Figure 2.26 and was inspired by the example project video[84]. As can be seen in the schematic, there are not many external components required for the USB to work. This is caused by the fact that the correct passive components are contained directly in the MCU. According to the AN4879[29], the USB peripheral already includes the **D+** pull-up resistor, so it is not required to be realized by an external component. On the left, we can see an ESD (Electro-Static Discharge) protection circuit, protecting the MCU from electrostatic discharge caused by human contact on the connector. It protects both the data lines and the VBUS.

On the right, we can see a USB-C™receptacle. Both data lines of the differential pairs are connected together to support USB-C™reversibility. A pull-down resistor with the value of 5.1 kΩwas connected to both of the **CC** lines of the USB-C™connector. These resistors are used for device discovery, configuration, and connection management over a USB-C™cable and also for communication in case the device has the power USB Power Delivery[89], which our device doesn't support. As an additional protection measure, a PTC fuse is added to the VBUS power rail.

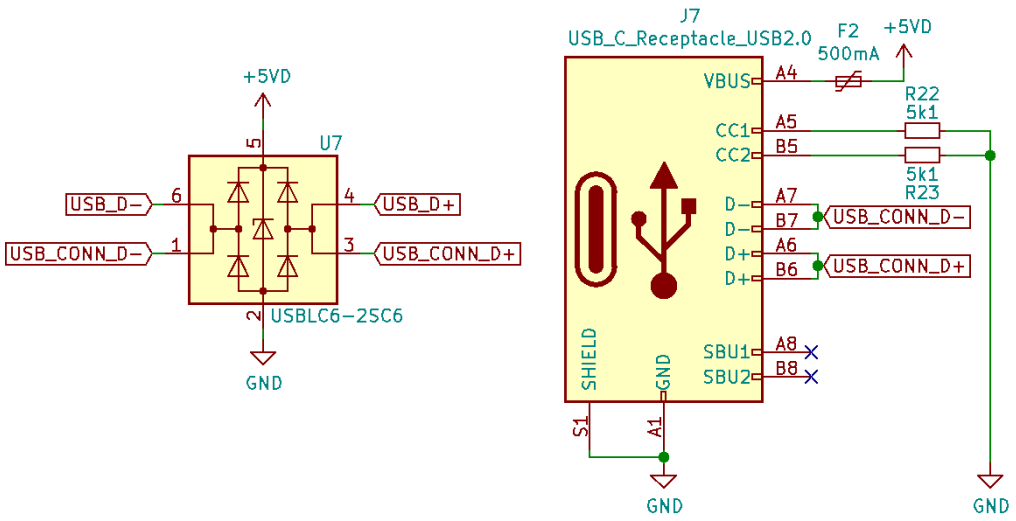


Fig. 2.26: The schematic of the USB circuit.

2.7.7 Stepper Driver Circuitry

There are two hardware revisions with two types of stepper motor driver ICs. Both of the circuits were adapted from the IC datasheet[79, 82]. The datasheet recommends motor voltage filtering using quite large electrolytic capacitors (100 µF). In order to save vertical space, these capacitors were replaced using 10 µF capacitors with the maximal voltage of 50 V.

Revision 1

The schematic for the circuit of the first revision can be seen in the Figure 2.27. This revision utilized the TMC2100-TA driver IC. This driver is configured using GPIO, and it is configured in the following way:

- use internal sense resistor with **AIN** as a current reference for internal sense resistors or use external sense resistors with **AIN** for scaling,
- use either SpreadCycle™or StealthChop™with interpolation and 16 microsteps,
- shortest slow decay phase,
- recommended chopper hysteresis - low hysteresis with 4% of full-scale current,
- shortest chopper blank time,

These values were chosen by following the recommended values from the datasheet. In the schematic, which was adapted from the recommended schematic in the datasheet[79], we can see the motor power rail filtering capacitor composed of five 10 µF capacitors. Then there are some more filtering capacitors and capacitors for the integrated charge pump. In the center of the schematic, there is the driver IC itself. The driver IC is controlled using the **REF, EN, STEP, DIR** and **MODE**

signals, to indicate for an error, the **ERR** signal is consumed by the MCU. The current sensing resistor values are set to 0.22 Ω, allowing for driving motors with 0.96 A RMS phase current.

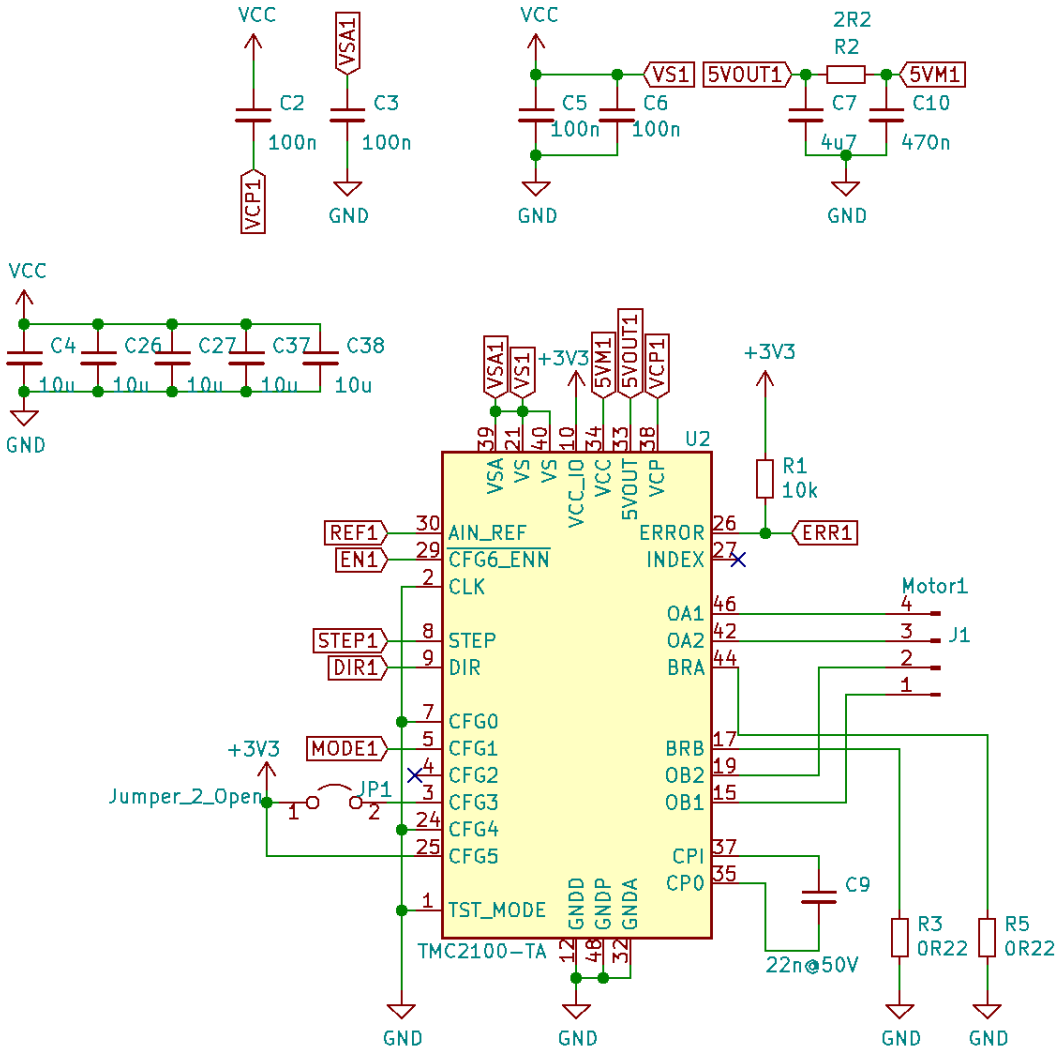


Fig. 2.27: The schematic of the stepper motor driver IC in the first revision.

Revision 2

In the Figure 2.28, we can see the schematic of the second hardware revision with the TMC2226-SA stepper motor driver IC. On the top, we can see the motor power rail filtering capacitors and also some auxiliary capacitors required by the datasheet. In the center, there is the driver IC itself. There are again some filtering capacitors and connections to the motor and to the MCU. The driver's address need to be set using the **MS1_AD0** and **MS2_AD1** pins to support multiple drivers on the same UART configuration interface. There are also some other configuration pins, but these have not been connected as their function can be replaced by the

configuration interface. For controlling the driver IC, the **STEP** and **DIR** pins are utilized. Finally, in the bottom right corner, there are the current sensing resistors for the motor phases current. Their value was chosen by the datasheet to support the maximal RMS phase current of 1.92 A - 0.1 Ω. It is important to select resistors with the proper power rating, in this case $P = R * I^2 = 0.1\Omega * (1.92A)^2 = 0.4W$, therefore we need resistors rated for at 0.5 W of power dissipation.

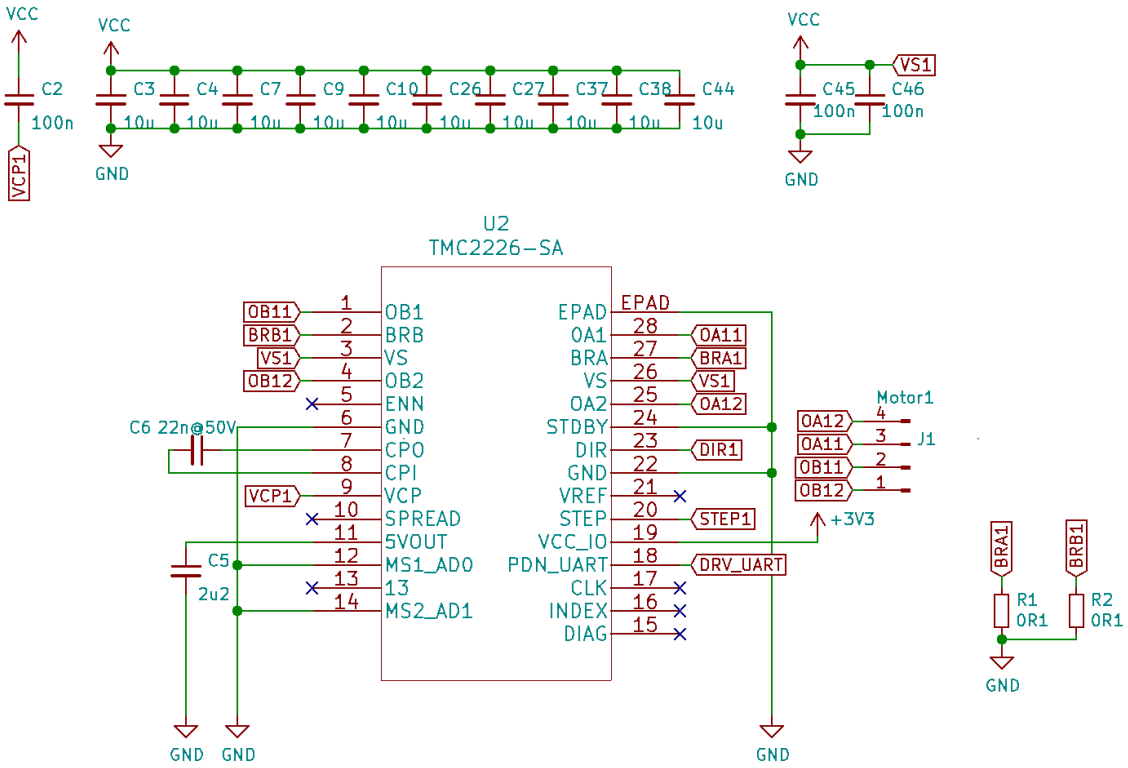


Fig. 2.28: The schematic of the stepper motor driver IC in the second revision.

2.7.8 Auxiliary Circuitry and Connectors

We added some auxiliary circuits (shown in the Figure 2.29) and future-proofing connectors (shown in the Figure 2.30) to improve the user experience, debugging, and making the controller future-proof a bit.

In the Figure 2.29, we can see that there are four LEDs for debugging and indicating the driver's state. Two of them indicate power on the 3.3 V rail and on the motor power rail (VCC). The other two LEDs are connected to the MCU and can be utilized to show the status and error states of the SM4 driver. Further, there is the voltage divider used for sensing the motor power rail voltage. Apart from the two resistors, there is also a small filtering capacitor. The resistor values were calculated for the first revision which aimed to support up to 43 V, given this voltage, the output

voltage on the divider would be $U_{out} = U_{in} * \frac{R19}{R18+R19} = 43V * \frac{4.7k\Omega}{104.7k\Omega} = 1.93V$, which is small enough for the internal ADC of the MCU.

We also added I^{2}C pull-up resistors, that can be connected to the bus using solderable NO (Normally Open) jumpers. The values for these pull-up resistors were chosen to be 2.2 kΩ. This is because we want to connect the driver using wires with unknown parasitic capacitance, which can be compensated by lower resistance of the resistors. Another reason is that the student boards have Zener diodes limiting the voltage on the I^{2}C bus lines, and these diodes require quite a high current to function properly.

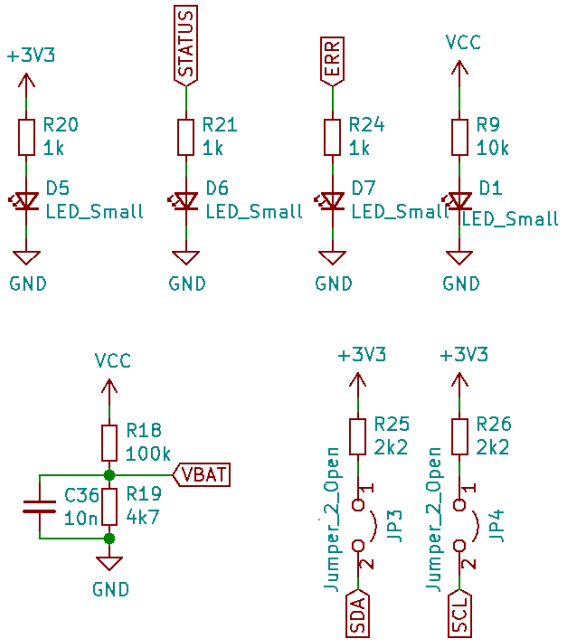


Fig. 2.29: Auxiliary circuits schematic - LEDs, I^{2}C, motor voltage measurement.

We wanted to future-proof the SM4 driver, and therefore, we added three connectors that will enable expanding the controller in the future. The connectors' schematic is depicted in the Figure 2.30. The first connector on the left is a connector designed to add two incremental encoders to the stepper driver. It supports quadrature encoders with index pulse. The second connector in the middle is the GPIO connector, which can be used to expand the driver's capabilities using five logic signals. Finally, on the right, there is a connector with two SPI buses broken out, which should allow for connecting SSI absolute encoders by bit-banging the SSI bus.

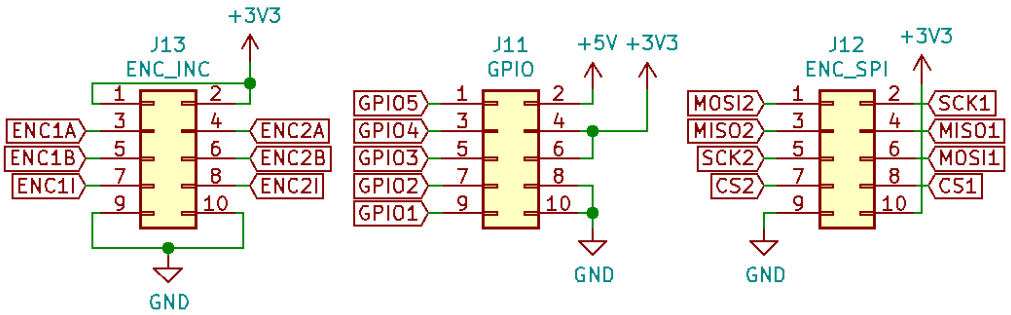


Fig. 2.30: Schematic of connectors that allow for connecting incremental and absolute encoders and extending the driver with logic signals.

2.7.9 PCB Design

When the schematic design was done, we went on to route the PCB. As for the layer stack up, we utilized the four-layer one, with internal layers used for GND and 3.3 V routing and the outside layers for signals. The stack up was previously described in the Section 2.7.1. With trace widths and via sizes, we aimed for compatibility with the technologies provided by the JLCPCB manufacturing house, but adding some margin to be safe there are no problems with manufacturing, which meant 0.2 mm minimal trace width and 0.8 mm wide vias, with 0.4 mm via drill size. Given limitations with the assembly service, all of the components are placed on the top side of the PCB. The PCB was designed to fit the area on the Raspberry Pi behind the ethernet and USB ports and is therefore 65x56 mm. When designing, we also aimed to have the power and communication connectors along a single side of the PCB and connectors for the motors on another one. The final PCB design for both of the revisions can be seen in the Figures 2.31 and 2.32.

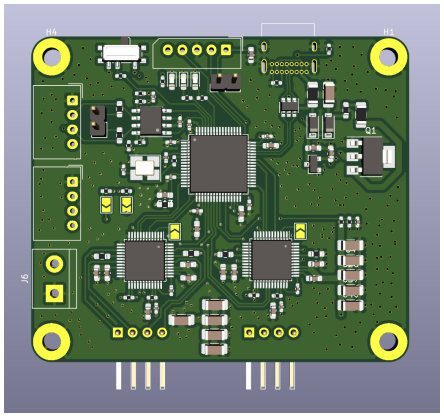


Fig. 2.31: Rendered first revision PCB.

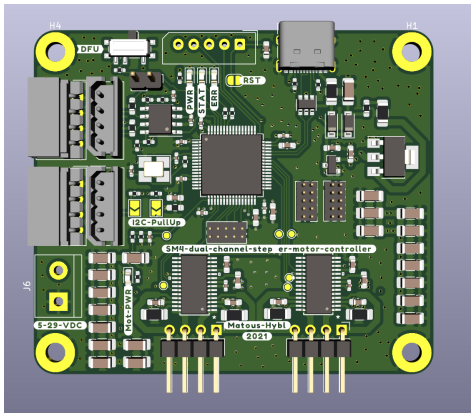


Fig. 2.32: Rendered second revision PCB.

2.8 Development of the Bare-Metal Firmware

This Section describes the development of the bare-metal firmware. First, we go over the firmware architecture, describing the technology stack and the architecture itself. Second, we go over some components of the firmware that are either crucial to the firmware, or their implementation stands out in any way.

2.8.1 Firmware Architecture

Firmware architecture can be described in many ways. We decided to show three ways to describe this firmware. First, we describe the technology stack, describing what crates and libraries upon which we built the hardware. These libraries have mostly been described in the Section 2.4. The technology stack can be seen in the Figure 2.33. When we go from the bottom, to the top, we can see that we access the hardware MCU via two crates - the **stm32-rs** used to access peripheral circuits of the MCU and the **cortex-m** crate used to access the ARM core of the MCU. The **stm32f4xx-hal** builds upon the **stm32-rs** crate and provides Hardware Abstraction Layer over the STM32F4 family and implements the **embedded-hal** traits. The **cortex-m** crate is then used by the RTIC scheduler (described in the Section 2.4.5). As can be seen in the Figure, the firmware itself then builds upon all of these technologies - it accesses the HAL as some peripherals have their abstractions developed, for some special configurations, the direct peripheral register access via **stm32-rs** is used, the RTIC scheduler is used for orchestrating the firmware and for some low-level assembly instructions the **cortex-m** crate for low-level ARM core access is used.

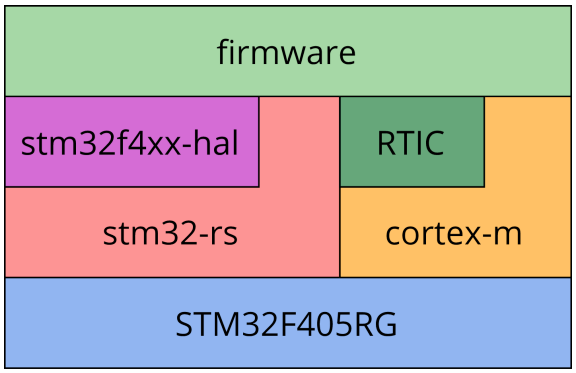


Fig. 2.33: Bare–metal firmware technology stack.

While the technology stack gives us vital information about the architecture, the architecture itself is much more complex and, in theory, could and should be independent of the used technologies[90]. The architecture overview diagram can

be seen in the Figure 2.34. The central part of the architecture is the Object Dictionary (described in Section 2.2.2 and 2.8.2 (implementation-wise)). The object dictionary contains the state of the stepper motor controller alongside configuration. Part of the object dictionary is also persistent storage where its values are retained between reboots. In the center of the firmware, there is also a general state that stores variables outside the scope of the Object Dictionary and software failsafe mechanisms that can manipulate bot the state and object dictionary. When we look lower in the diagram, we can see that there are communication interfaces that receive data from and send them to the outer world, usually to some high-level system. On the other hand, when we look higher in the diagram, that's where the motion control systems operate. They get their data from the object dictionary (control values) and encoders and use them to control the stepper motor driver ICs. The encoders and motor drivers are again connected to the outer world - meaning that they interact with it via the connected motors.

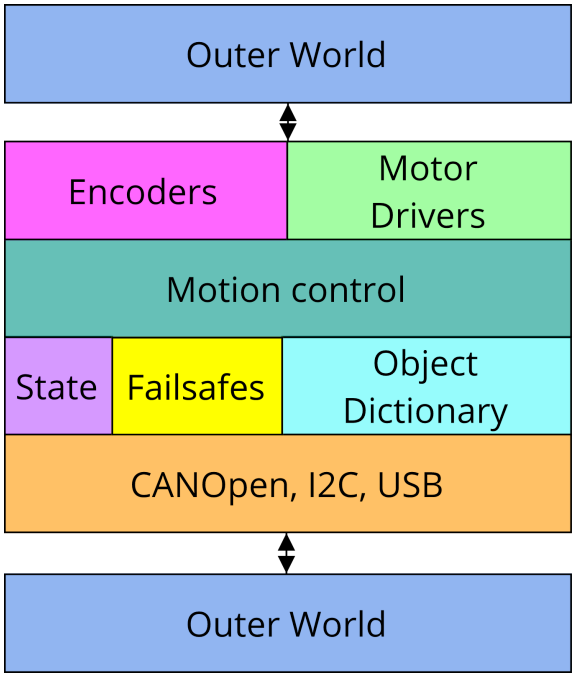


Fig. 2.34: Bare–metal firmware architecture.

There is one more final point of view on the architecture, connected with the project structure. Given there are two projects - the bare-metal firmware and the further-described control application, there is a need to share code between those two projects, e.g., share communication models. An vital distinction being that the bare-metal firmware is also cross-compiled in a **no_std** environment. Code sharing was solved by creating another project, that is **no_std** and is called **shared**, both of the application projects statically link against it. This can be seen in a

diagram presented in the Figure 2.35. We adopted the project structure from[66], which proposes a project structure that promotes code sharing and testability of all components.

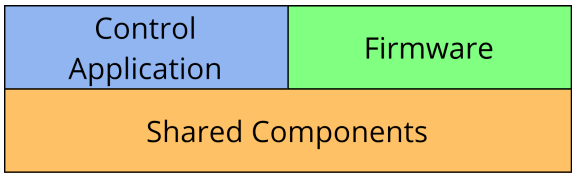


Fig. 2.35: Component sharing between the bare–metal firmware and the Control Application.

2.8.2 Object Dictionary

The importance and role of the Object Dictionary was described in the Sections 2.2.2 and 2.8.1. When designing the Object Dictionary, we wanted to design it to be universally used and implemented in multiple ways (e.g., with persistent storage or without one). The abstraction was done using traits. The most important trait is the ObjectDictionary trait displayed in the Listing 2.17. This trait describes the Object Dictionary as a type that contains information about battery voltage, temperature, and an arbitrary number of axes. The axis is also represented as a trait with accessor methods for specific axis settings (controller settings, current settings, etc.).

```rust
/// Trait for Object Dictionary abstraction
pub trait ObjectDictionary<const RESOLUTION: u32> {
    /// Returns battery voltage stored in the Object Dictionary.
    fn battery_voltage(&self) -> f32;
    /// Returns temperature stored in the Object Dictionary.
    fn temperature(&self) -> f32;
    /// Sets the battery voltage value in the Object Dictionary.
    fn set_battery_voltage(&mut self, battery_voltage: f32);
    /// Sets the temperature value in the Object Dictionary.
    fn set_temperature(&mut self, temperature: f32);
    /// Returns the configuration of a specific axis.
    fn axis(&self, axis: Axis) -> &dyn AxisDictionary<RESOLUTION>;
    /// Returns a mutable reference the configuration of a specific axis.
    /// This is the only way an axis configuration can be changed
    fn axis_mut(&mut self, axis: Axis) -> &mut dyn AxisDictionary<RESOLUTION>;
}
```

Listing 2.17: Trait for abstracting away the Object Dictionary.

To implement a simple Object Dictionary, implementing these traits would be enough. Still, since we wanted to create an Object Dictionary whose values can be saved in arbitrary storage, we needed to create another abstraction for storage. Such abstraction can be found in the Listing 2.18. As can be seen in the Listing, the trait defines loading and saving values of 32-bit floats and boolean values for an arbitrary type of key that implements the ObjectDictionaryKey.

```rust
pub trait ObjectDictionaryStorage {
    fn save_f32<KEY: ObjectDictionaryKey>(&mut self,
        key: KEY, value: f32);
    fn save_bool<KEY: ObjectDictionaryKey>(&mut self,
        key: KEY, value: bool);
    fn load_f32<KEY: ObjectDictionaryKey>(&self,
        key: KEY) -> Option<f32>;
    fn load_bool<KEY: ObjectDictionaryKey>(&self,
        key: KEY) -> Option<bool>;
}
```

Listing 2.18: Trait for abstracting away the Object Dictionary storage.

Using all the aforementioned traits, we then developed an Object Dictionary implementation called the PersistentStoreObjectDictionary, which is generic over the ObjectDictionaryStorage, meaning that an arbitrary storage type can be used with it. Similar to the Object Dictionary traits, we also needed to develop a Persistent Store Object Dictionary for axis data, which is also generic over the Object Dictionary Storage and can be seen in the Listing 2.19.

An important note is that now when any part of the code needs to access the Object Dictionary, it is done by hiding the real implementation behind dynamic dispatch based on the ObjectDictionary trait.

Since we are expecting that the only way a value can be written into the Object Dictionary is using the accessor methods, we employ buffering to make storage reads more scarce, therefore more making the ObjectDictionary more effective. This basically means that at the firmware startup, all the values are loaded to RAM, and the storage is not accessed when reading.

```rust
#[derive(Copy, Clone)]
pub struct PersistentStoreAxisDictionary<
    STORAGE: 'static + ObjectDictionaryStorage,
    const RESOLUTION: u32,
> {
    axis: Axis,
    mode: AxisMode,
    enabled: bool,
    target_velocity: Velocity,
    actual_velocity: Velocity,
    target_position: Position<RESOLUTION>,
    actual_position: Position<RESOLUTION>,
    current: CurrentSettings,
    velocity_controller_settings: ControllerSettings,
    position_controller_settings: ControllerSettings,
    velocity_feedback_control_enabled: bool,
    acceleration: f32,
    storage: &'static Mutex<RefCell<STORAGE>>,
}
impl<STORAGE: 'static + ObjectDictionaryStorage, const
   RESOLUTION: u32>
    AxisDictionary<{ RESOLUTION }> for
       PersistentStoreAxisDictionary<STORAGE, RESOLUTION>
{
    fn set_accelerating_current(&mut self, current: f32)
       {
        self.current.set_accelerating_current(current);
        self.storage.lock().borrow_mut().save_f32(
            Key::key_for_axis(AxisKey::
               AcceleratingCurrent, self.axis),
            current,
        );
    }
...
}
```

Listing 2.19: Object Dictionary for persistently storing axis data.

2.8.3 Persistent Storage Using EEPROM Emulation

In the previous Section, we discussed the development of storage-agnostic persistent Object Dictionary, and in this section, the way we implemented the persistent storage is described. Persistent storage on MCUs is generally solved by using non-volatile memory that can be either part of the MCU or an external component. Different memory technologies may be used for both types of storage. In general, FRAM (Ferroelectric Random Access Memory), EEPROM (Electrically Erasable Programmable Read-Only Memory), or flash memories are used.

To save space on the PCB, save cost, and better utilize the MCU resources, we decided to use the internal flash memory to store the user data apart from the driver firmware. Even though flash memory may seem straightforward to use since they are ubiquitous, its low-level use is not that simple. Flash memory is generally divided into sectors that can be several kilobytes or megabytes large. These sectors can be electrically erased - which means that every bit in the sector is set to 1. Depending on the memory, a word of a specific size can be programmed, but it is only possible to flip the bits in the word to zero [91, 92]. The sector needs to be first erased and then programmed to write a higher number to the word in the sector. This is problematic for two reasons:

1. sectors generally have the size of several kilobytes, meaning that when you'd want to update the value in the desired word, the whole sector would have to be read to some other memory, erased, and then programmed again with the new, updated value,
2. there is a limited number of the whole sector erases, caused by the limitation of the hardware.

Fortunately, this problem can be solved by emulating the EEPROM memory as described in ST Application Note AN3969 [93]. The application note leverages two FLASH sectors of the same size, where one of them is marked as the active one and the second one is used when the first sector is full. The working principle is described in the following paragraphs and can be seen in the Figure 2.36.

In the beginning, both of the sectors are erased, and one of them is marked as active. Data are then written to the first sector into simulated cells. The cells contain a header (which can be understood as a key or a virtual address) and the data. When a new write is requested, the data are appended behind the already stored data. When data is read using the virtual address, or a key, the sector is traversed from its end, searching for the first occurrence of the key or address. The first occurrence is the most recent value of the cell marked by the key. This way, we are able to store the value with a specific identifier (key, virtual address) in the flash multiple times.

When no more cells can be written to the active sector, the second sector is marked active. The data are transmitted to the second sector, taking only the latest value of an identifier into account. After the transfer, the first sector is erased.

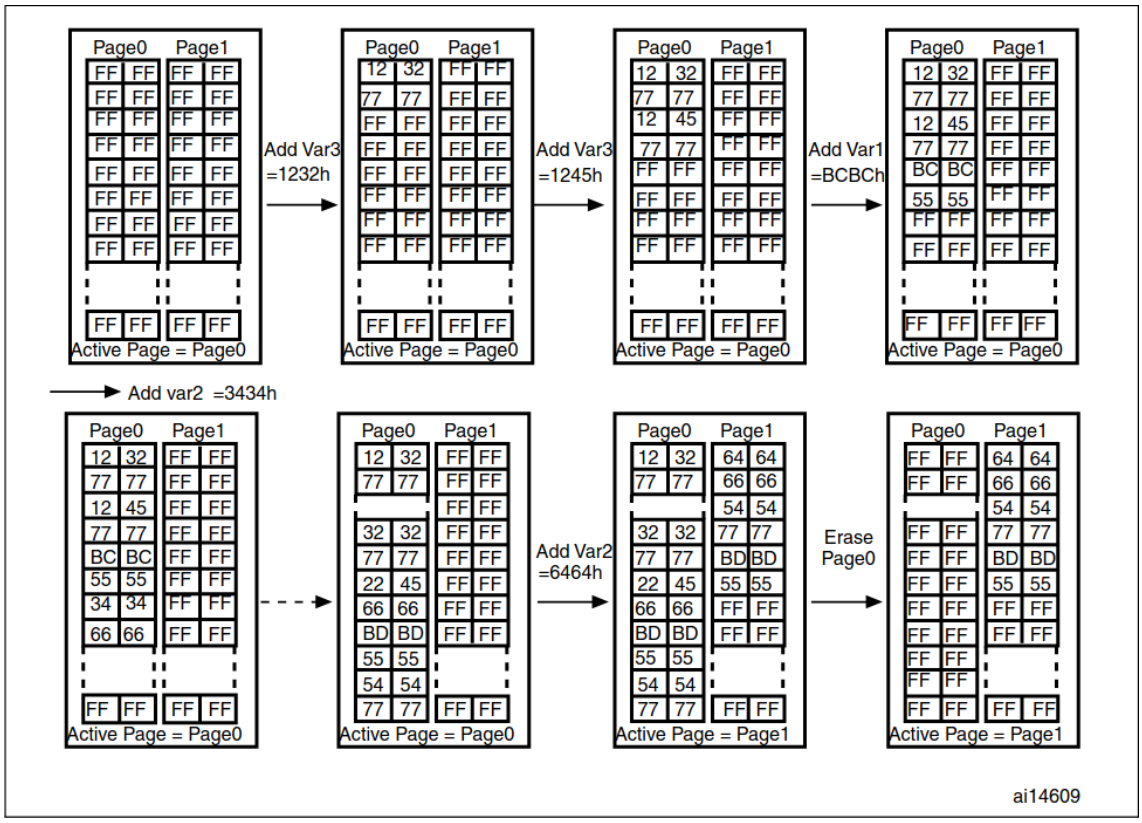


Fig. 2.36: EEPROM emulation working principle [93].

Even though the working principle of the EEPROM emulation is simple, there are some technical obstacles in the implementation. The first obstacle is that the flash memory on the STM32 MCU is split into differently sized sectors, and it is required that the sectors have the same size. Referring to the Reference Manual [72] there are some 16 kilobyte sectors that could be used for the emulation, as can be seen in the Figure 2.37. Using the 128 kilobyte sectors would also be possible, but given their size, copying values from one sector to another would take too much time, and also, read access times would be higher.

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	. . .	. . .	. . .
	Sector 11	0x080E 0000 - 0x080F FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

Fig. 2.37: FLASH layout of the STM32F405 MCU [72].

There is, however a problem with using the sectors in the beginning of the flash memory as that is where the firmware is usually stored. The solution to this problem is by leaving the first sector (Sector 0) for the vector table and instructing the linker to place the **.text** section of the program further in the memory. According to the documentation of the **cortex-m** Rust crate [94], this can be achieved by adding the line **_stext = ORIGIN(FLASH) + OFFSET** to the linker script, where the **OFFSET** shall be replaced with the offset of the target sector where we want our program to be stored, in our case **0x0000C000**, which indicates the start of the Sector 3.

As for the actual implementation of the emulation for the STM32F405, we decided to develop our own, as no suitable Rust crate was available for it. The development was inspired by a crate that implemented the emulation for STM32F103 [95] and by following the implementation in the Application Note. The functions from flash memory access were adopted from an as of the time of writing unmerged Pull Request into the STM32F4 HAL [96]. An example of accessing the emulated persistent storage can be seen in the following Listing 2.20.

```
let mut store = Storage::new(device.FLASH);
store
    .init()
    .expect("Failed to initialize emulated storage.");
store.write_f32(0xbeef, 3.14);
let read = store.read_f32(0xbeef).unwrap();
assert_eq!(read, 3.14);
```

Listing 2.20: An example use of the emulated persistent storage.

As can be seen in the Listing 2.20, first we create the object with a parameter of the flash peripheral, then we initialize the emulated storage - this prepares the sectors that are supposed to be used, and then we perform a simple write and read operations on the storage.

2.8.4 CANOpen Implementation

The CAN bus on STM32 is implemented in hardware in the bxCAN peripheral. This peripheral is the same for all of the STM32F MCUs, only with slight differences in filter banks and message mailbox sizes. Given this fact, it was possible to develop a crate that supports all of the STM32 MCUs, the crate is called **bxcan** and is developed the stm32-rs organization.

In the past projects described in Chapter 1, we utilized our own implementation of a driver for the bxCAN peripheral, but since the **bxcan** crate is well maintained, documented, and tested, we decided to discontinue the development of the driver and migrate to **bxcan**.

In order to implement the CANOpen protocols described in the Section 2.2.1, we developed a simple wrapper over the **bxcan** crate. The wrapper is basically a simple proxy that transforms CANOpen protocol data to CAN frames and sends them and parses received CAN frames into CANOpen protocol data. Upon initialization, the wrapper first configures the bxCAN peripheral, enables interrupts, and enables the peripheral itself.

The bxCAN initialization in the wrapper can be seen in the Listing 2.21.

```
let mut bus = Can::new(bus);
bus.configure(|config| {
    config.set_bit_timing(0x001a000b);
});
bus.enable_interrupts(
    Interrupts::FIFO0_MESSAGE_PENDING | Interrupts::
       FIFO1_MESSAGE_PENDING,
);
bus.modify_filters()
    .clear()
    .enable_bank(0, Mask32::accept_all());
bus.set_automatic_wakeup(true);
nb::block!(bus.enable()).unwrap();
```

Listing 2.21: Initializing the bxCAN peripheral in the CANOpen wrapper.

When an interrupt handler is invoked, the function **process_incoming_frame** is called, and it returns the type of the CANOpen protocol message alongside with received frame. If the frame contains a valid CANOpen message, the CAN frame is further parsed, for example, to decode the contents of the PDO as can be seen in the Listing 2.22 In the Listing, we can also see accessing the generic ObjectDictionary via the structure **DriverState**.

```rust
pub fn rx_pdo2<OD, const R: u32>(frame: &Frame, state: &mut DriverState<OD, R>)
where
    OD: ObjectDictionary<R>,
{
    if frame.data().is_none() {
        defmt::warn!("Invalid RxPDO2 received.");
        return;
    }
    if let Ok(pdo) = RxPDO2::try_from(frame.data().unwrap().as_ref()) {
        state
            .object_dictionary()
            .axis_mut(Axis::Axis1)
            .set_target_velocity(Velocity::new(pdo.axis1_velocity));
        state
            .object_dictionary()
            .axis_mut(Axis::Axis2)
            .set_target_velocity(Velocity::new(pdo.axis2_velocity));

        state.invalidate_last_received_speed_command_counter();
    } else {
        defmt::warn!("Malformed RxPDO2 received.");
    }
}
```

Listing 2.22: Accessing the Object Dictionary when a RxPDO2 is received.

2.8.5 I²C Slave Implementation

Implementing I²C slave on a MCU is a cumbersome problem, as there is not enough documentation or example code. Timing of the bus, requiring the slave to react within relatively short time, makes development and debugging harder. On the other hand with the help of **defmt** and RTT this is not a problem anymore. We had already developed one implementation of the I²C Slave handling for STM32F0

MCU as a part of the KM3 firmware described in the Section 1.2. Unfortunately, the STM32F4 family has a different version of the I^2C peripheral than the STM32F0, resulting with significant changes to the peripheral handling code, even though the external API remained the same. The I^2C2 peripheral is utilized, and it is configured to use the 7-bit addressing mode, with enabled clock stretching to give the MCU more time to prepare the data for transfer. Finally, all the interrupt sources are enabled. The peripheral has two interrupt vectors - one for data-related interrupts **I2C2_EV** and one for error-related interrupts **I2C2_ER**. When the error-related interrupt handler is invoked, the error flags are cleared, and the transfer is reset. In the data-related interrupt handler, the state machine implementing the I^2C slave register-based protocol (described in the Section 2.2.3) is implemented. The result of calling the data interrupt handling function is a state which denotes what is expected by the higher-level system - the data can be either requested or received. The upper-level system then handles transferring the data between the I^2C slave implementation's buffers and the Object Dictionary.

2.8.6 USB

The integrated USB on the SM4 stepper motor controller has two functions - it enables the possibility to upgrade the firmware via DFU (handled by the internal MCU bootloader) and provides an interface for configuration (developed by us in the firmware). For the configuration, the USB utilizes the CDC-ACM device class, which means that to the host device, the SM4 controller turns up as a serial port. Thankfully, given the ubiquity of USB devices, crates for implementing USB devices on the STM32 are already developed. There is the crate **usb-device** providing the USB stack and abstractions for hardware[97], the **stm32-usbd**[55] implementing the hardware abstraction and **usbd-serial**[98] implementing the CDC-ACM device class for the **usb-device** stack.

The implementation of this functionality in the firmware is done via the **USBProtocol** struct, which is basically a wrapper over the raw USB Serial port functionality. This wrapper is interrupt-driven, quite similarly to the I^2C Slave driver, meaning that the received data is parsed when an interrupt handler is invoked, and the data is then exchanged between the wrapper and the Object Dictionary.

An example of configuring the USB device can be found in the Listing 2.23.

```
let usb_dev = unsafe {
    UsbDeviceBuilder::new(USB_BUS.as_ref().unwrap(),
       UsbVidPid(0x16c0, 0x05e1))
      .manufacturer("MH Robotics")
      .product("SM4")
      .serial_number("sm4202101")
      .device_class(usbd_serial::USB_CLASS_CDC)
      .build()
};
```

Listing 2.23: Initializing the USB device with the CDC-ACM class.

The used device **VID** and **PID** were adopted from the VUSB project[99], that provides them for free with some rules about their use.

2.8.7 Stepper control

Movement of stepper motors, controlled using the stepper motor driver ICs is generally controlled using the **STEP/DIR** interface. This interface consists of two signals, the first of them named **STEP** is a square wave signal with variable frequency, where the edges (rising, falling, or both) instruct the driver to move the motor by a microstep. On the other hand, the second signal named **DIR** is generally a logic signal whose logic level denotes the direction of the shaft movement.

Apart from these two digital signals, there is usually an analog signal that is used to set the motor phase current. In more modern stepper drivers, this analog signal is replaced by a serial digital interface, allowing for a finer current setting.

The **STEP** signal can be generated either in software (by changing the output logic level of a GPIO pin) or in hardware by utilizing a pwm signal with the duty cycle of 50 %. Both of these approaches have their limitations and advantages. Generating the signal in software has the advantage of being able to easily count the number of microsteps the motor was commanded to do. On the other hand the upper limit of the maximal frequency is much lower than with hardware generation, and this technique uses more computational resources as the signal is generally generated using an interrupt on timer overflow, where the signal logic level needs to be toggled (which requires branching) and the pulse counter needs to be accessed. Using a timer with pwm allows for much higher maximal frequencies. On the other hand counting the pulses can prove to be quite hard. This issue will be described in more detail in the Section 2.8.8.

We decided to utilize the **STEP** signal generation done by the hardware as we wanted to offload the work from the MCU core.

To abstract the real implementation, we created a trait (see Section 2.3.5) for setting the microstepping frequency, as can be seen in the Listing 2.24. Using this trait, the software controlling the stepper driver IC can use either hardware or software **STEP** signal generator.

```
1  /// This trait is an abstraction over hardware/software
       that is capable of generating square wave signal of
       specific frequency.
2  /// It is generally implemented by timers.
3  pub trait StepGenerator {
4      /// Sets output frequency of the generator.
5      ///
6      /// # Arguments
7      /// * `frequency` - frequency of the output square
           wave signal
8      fn set_step_frequency(&mut self, frequency: Hertz);
9  }
```

Listing 2.24: Trait for abstracting STEP generation.

In our case, the trait is implemented by the abstractions over the MCU's advanced control timers 1 and 8. The abstractions over the timers are based on the timer implementations found in the **stm32f4xx-hal**, but are preconfigured to generate output pwm signals and also to act as a master timer generating the clock signal for other timers on compare.

Using this trait, we were able to define a struct that describes the TMC2100 driver, as can be seen in the Listing 2.25.

```
1  pub struct TMC2100<G, STEP, DIR, DAC> {
2      generator: G,
3      _step_pin: STEP,
4      dir_pin: DIR,
5      current_dac: DAC,
6      sense_r: f32,
7      microsteps_per_revolution: f32,
8  }
```

Listing 2.25: TMC2100 driver.

In the Listing, we can see that the driver contains a generic generator, has ownership of the **STEP** pin (so that no other peripheral can access it), has ownership of the **DIR** pin, DAC current reference and knows the value of the sense resistor for current setting and microsteps per revolution for **STEP** output frequency setting.

For a more seamless integration with the motion controller described in the Section 2.8.10 the trait **StepperDriver** was declared as can be seen in the Listing 2.26.

```
1  /// This trait is an abstraction over stepper drivers.
2  /// Generally the drivers have two functions - generate
       steps and set output current.
3  pub trait StepperDriver {
4      /// Sets output frequency of the driver.
5      /// this shall be the angular frequency of the output
           shaft in revolutions per second.
6      ///
7      /// # Arguments
8      /// * `frequency` - frequency of the output motor
           shaft in revolutions per second
9      fn set_output_frequency(&mut self, frequency: f32);
10
11     /// Sets the target current the driver shall drive
           the stepper motor with.
12     ///
13     /// # Arguments
14     /// * `current` - the desired current in Amps
15     fn set_current(&mut self, current: f32);
16 }
```

Listing 2.26: Trait for abstracting the stepper motor driver IC.

This trait was then implemented for the TMC2100 structure, implementing the stepper control itself, as is shown in the Listing 2.27.

```rust
impl<G, STEP, DIR, DAC> StepperDriver for TMC2100<G, STEP, DIR, DAC>
where
    G: StepGenerator,
    DIR: embedded_hal::digital::v2::OutputPin,
    DAC: DACChannel,
{
    fn set_output_frequency(&mut self, frequency: f32) {
        if frequency < 0.0 {
            self.dir_pin.set_high().ok();
        } else {
            self.dir_pin.set_low().ok();
        };
        self.generator.set_step_frequency(Hertz::new(
            (frequency.abs() * self.microsteps_per_revolution) as u32,
        ))
    }
    fn set_current(&mut self, current: f32) {
        let voltage = (current.abs() * MAX_V_REF as f32 / V_FS * (self.sense_r + R_OFFSET) / 0.707) as u16;
        self.current_dac.set_output_voltage(voltage.min(MAX_V_REF));
    }
}
```

Listing 2.27: Implementing the StepperDriver trait for TMC2100.

2.8.8 Simulated encoders

The SM4 stepper motor controller is meant to be used without a hardware encoder in its default hardware configuration. This requirement was caused by the reasoning that the motor controller is also targeted for the BPC-PRP course, where students use the distance driven by the wheels to calculate the robot's position in the world's reference frame. Thankfully, with a stepper motor, it is quite easy to simulate encoders by counting the number of microsteps the motor was supposed to move by. The simulated encoders do not provide real feedback from the system On the other

hand, when some conditions are met (no step skipping - no motor overloading), the measurement from them can be quite reliable.

As we already discussed in the Section 2.8.7, we decided not to use counting the microsteps in software but in hardware instead. This is done by chaining timers in the MCU. We already mentioned that the timers used to generate **STEP** signal are configured to be the clock source for other timers, and by counting the clock cycles, we are able to count the number of microsteps the motor was commanded to turn by.

The working principle is really simple and also saves computational power of the MCU, but has one disadvantage - the amount of microsteps per sampling period is too low to be used for velocity control, as the difference between the consecutive position reads is usually zero and only sometimes 1. This poses a big problem for motor control systems of any kind.

Let's have a look at the implementation. Similar to what we did with the **STEP** signal generators, we also declared a trait to abstract away the thing that counts the pulses so that on the outside, it doesn't matter how the counting works. The trait can be seen in the Listing 2.28.

```
1 /// Trait used to abstract STEP pulse and other counters.
2 pub trait Counter {
3     /// Return the current internal value of the counter.
4     fn get_value(&self) -> u32;
5     /// Resets the internal value of the counter.
6     fn reset_value(&mut self);
7 }
```

Listing 2.28: Counter trait for counting STEP pulses.

The trait was implemented for timers 2 and 5 of the MCU.

Apart from this trait, an abstraction over the whole encoded was developed. The abstraction utilizes const generics to define encoder resolution and can be seen in the Listing 2.29.

```rust
/// A trait abstracting common encoder functionality.
/// It is suitable for both incremental and absolute encoders.
/// It is designed so its [`Self::sample()`] shall be periodically called with known fixed period,
/// which allows for velocity calculations.
pub trait Encoder<const RESOLUTION: u32> {
    /// Returns the velocity measured by the encoder.
    /// This value is generally calculated from consecutive position readings.
    fn get_velocity(&self) -> Velocity;
    /// Returns the current position of the shaft.
    fn get_position(&self) -> Position<RESOLUTION>;
    /// Sets the sampled position to zero.
    /// This is applicable only with incremental encoders.
    /// Absolute encoders might offset the zero by software.
    fn reset_position(&mut self) -> Position<RESOLUTION>;
    /// This function shall be periodically called to sample the encoder.
    /// Sampled values are used for position and velocity readings.
    fn sample(&mut self);
    /// This method shall be called with non-directional encoders whenever there is a change of rotation direction.
    /// # Arguments
    /// * `direction` - indicates whether the shaft is now turning in the clockwise or counterclockwise direction.
    fn notify_direction_changed(&mut self, direction: Direction);
}
```

Listing 2.29: Encoder trait for abstracting encoders.

Based on this trait and the Counter trait, a simulated encoder was developed. The big advantage is that the simulated encoder can be easily replaced by another

hardware or software realized encoders. This Encoder trait is further used in the motion controller.

For representing the position, we utilized the model used in servo controllers by TGDrives - a signed integer denoting the number of revolutions and an unsigned integer indicating the angle. For example, assume that we have a two-bit encoder, and a position of 1 revolution, and the angle with the value of 2, meaning that the position is equal to 1.5 revolutions. Now, assume that the number of revolutions is -1, and the angle is again 2, which means that the resulting position represented by these two numbers is -0.5. More on this implementation, alongside with tests, can be found in the shared library in the module defining the structure Position.

2.8.9 Device Monitoring

In order to provide status and health information, simple device monitoring is employed. The monitoring system periodically reads internal MCU temperature and the motor voltage. This functionality is implemented by accessing the internal MCU ADC. The internal ADC utilizes the Successive approximation principle and supports up-to 19 channels[83]. The ADC is configured to read the two channels and to use DMA to transfer the values from the peripheral to the program's memory. Both ADC configuration and DMA transfer are already implemented in the **stm32f4xx-hal** crate, meaning that no low-level peripheral access code was required. The monitoring data are periodically transferred to the Object Dictionary by the higher-level code.

First, we configure the ADC as can be seen in the Listing 2.30. We set the DMA transfer mode to **Continuous** (which reissues a DMA request on every start of conversion) and enable scan mode, which scans all the channels in the sequence. Further, both of the channels are configured, with the assignment to pin or special channel, the order in the conversion sequence, and sample time, which denotes for how many clock cycles the sample-and-hold circuits samples. Finally, the temperature and VRef channel measurement is enabled as it is not enabled by default by the hardware.

```
let mut adc = Adc::adc1(raw_adc, true, adc_config);
let adc_config = AdcConfig::default()
     .dma(Dma::Continuous)
     .scan(Scan::Enabled);
adc.configure_channel(&Temperature, Sequence::One,
   SampleTime::Cycles_480);
adc.configure_channel(&battery_voltage, Sequence::Two,
   SampleTime::Cycles_480);
adc.enable_temperature_and_vref();
```

Listing 2.30: Configuring ADC for temperature and voltage monitoring.

Next, we configure the DMA transfer, as can be seen in the Listing 2.31. We configure the DMA controller to issue an interrupt when the transfer is complete, to increment addresses only in memory and not in the peripheral, and we disable double buffering.

```
let first_buffer = singleton!(: [u16; 2] = [0; 2]).unwrap
   ();
let config = DmaConfig::default()
    .transfer_complete_interrupt(true)
    .memory_increment(true)
    .double_buffer(false);
let transfer = Transfer::init(dma, adc, first_buffer,
   None, config);
```

Listing 2.31: Configuration of the DMA controller for ADC transfers.

The monitoring system is then periodically asked to poll data from the ADC by starting the transfer as can be seen in the following Listing 2.32.

```
self.transfer.start(|adc| {
    adc.start_conversion();
});
```

Listing 2.32: Polling the ADC.

When the DMA transfer complete interrupt routine is called, the subsequent transfer is prepared, and the measured data is processed using factory calibration and formulae that can be found in the reference manual[83]. The final two coefficients in the battery_voltage calculation represent the voltage divider described in the Section 2.7.8. The preparation and calculation is shown in the Listing 2.33.

```
let (buffer, _) = self
    .transfer
    .next_transfer(self.buffer.take().unwrap())
    .unwrap();
let raw_temp = buffer[0];
let raw_volt = buffer[1];
self.buffer = Some(buffer);
let cal30 = VtempCal30::get().read() as f32;
let cal110 = VtempCal110::get().read() as f32;
self.temperature = (110.0 - 30.0) * ((raw_temp as f32) -
   cal30) / (cal110 - cal30) + 30.0;
self.battery_voltage = (raw_volt as f32) / ((2_i32.pow
   (12) - 1) as f32) * 3.3 / 4.7 * 104.7;
```

Listing 2.33: Processing the data measured by the ADC.

2.8.10 Motion Control

Motion control is a crucial component of the firmware. Based on the values stored in the Object Dictionary, it calculates the output control signal values for the stepper motor driver ICs. In our case, the core of the motion control component are two nested PSD controllers. The controllers are implemented with anti-windup realized by summator and output value clamping The algorithm used for the PSD controller can be seen in the Listing 2.34. For ease of use, all the constants are stored in a structure **ControllerSettings**.

```
let error = desired - actual;
self.sum += error * self.sampling_period;
self.sum = self.sum.clamp(
    -settings.max_output_amplitude,
    settings.max_output_amplitude,
);
let action = error * settings.proportional
    + settings.derivative * (error - self.previous) /
       self.sampling_period
    + settings.integral * self.sum;
self.previous = error;
action.clamp(
    -settings.max_output_amplitude,
    settings.max_output_amplitude,
)
```

Listing 2.34: Implementation of the PSD controller with integrator.

One of these controllers (the inner one) is used for controlling output speed based on the target speed, and the second one (the outer one) is used to control the position. The structure of the motion controller can be seen in the Figure 2.38.

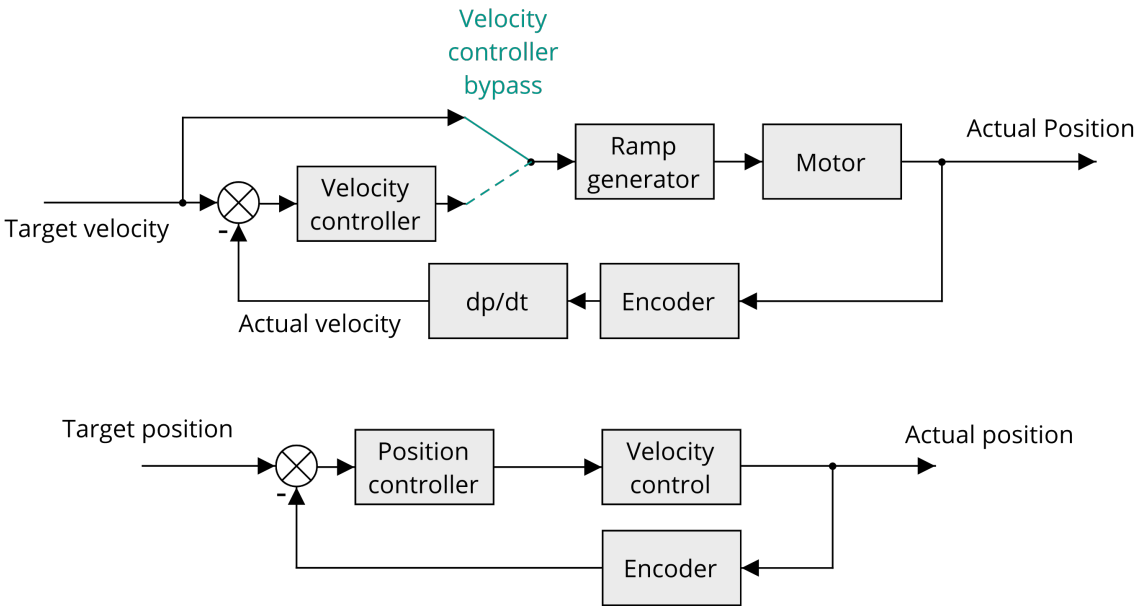


Fig. 2.38: Motion control schematic.

In the first half of the Figure, there is the velocity controller, which is used both when the SM4 stepper motor controller's axis is running in the velocity or position mode. Given that the stepper motor controllers are often driven in the open-loop, there is an option to bypass the velocity PSD controller, meaning that the target speed is directly passed to the ramp generator. The ramp generator is capable of generating trapezoidal ramps with defined acceleration. An important note being that the ramp generator is sampled at a higher frequency than the rest of the motion controller. The algorithm for calculating trapezoidal ramps can be seen in the Listing 2.35.

```
1  pub fn generate(&mut self, target_speed: f32,
       target_acceleration: f32) -> f32 {
2      let step = target_acceleration / self.
           generation_frequency;
3      let diff = target_speed - self.current_speed;
4      if diff.abs() < step {
5          self.current_speed = target_speed;
6      } else {
7          self.current_speed += diff.signum() * step;
8      }
9      self.current_speed
10 }
```

Listing 2.35: Calculating trapezoidal ramps.

Velocity calculated by the ramp generator is then passed to the stepper motor driver IC via an abstraction. In the case of the closed-loop control, the bypass is not active, and the velocity controller regulates the velocity based on the value speed measured by the encoder. It is important to note that to successfully control in closed-loop mode the difference between encoder samples must be higher than 0, ideally several orders of magnitude higher.

When the SM4 stepper motor controller's axis is running in the position control mode, the second controller shown in the second half of the Figure is utilized. In this case, the feedback from the encoder is utilized and this controller is a proper closed-loop controller.

The actual motion controller implementation utilizes the **StepperDriver** and **Encoder** traits described in the Sections 2.8.7 and 2.8.8. It is declared as follows in the Listing 2.36.

```
1  /// Motion controller of an arbitrary axis.
2  /// This motion controller expects that the target driver
       is controlled either in velocity or position mode.
3  /// Trapezoidal ramp generator is utilized.
4  pub struct AxisMotionController<D: StepperDriver, E:
       Encoder<RESOLUTION>, const RESOLUTION: u32> {
5      /// The target stepper motor driver, that will be
           controlled by this motion controller.
6      driver: D,
7      /// The encoder, that will be used to provide
           feedback for closed loop control
8      encoder: E,
9      velocity_controller: PSDController,
10     position_controller: PSDController,
11     ramp_generator: TrapRampGen,
12     /// Variable used to store the calculated velocity
           action for ramp generator.
13     axis_velocity_action: f32,
14 }
```

Listing 2.36: Implementation of the PSD controller with integrator.

Given there are two sampling frequencies - one for control and one for ramping (the higher one), there are two methods used to actually control the axis. The **control** method calculated the output velocity based on the requirements imposed by the object dictionary and stores it into a member variable. The ramping method

then utilizes this value to change the output velocity. Headers of these functions can be seen in the Listing 2.37.

```
1 pub fn ramp(&mut self, global_disable: bool, dictionary:
    &mut dyn AxisDictionary<RESOLUTION>) { ... }
2 pub fn control(&mut self, global_disable: bool,
    dictionary: &mut dyn AxisDictionary<RESOLUTION>) { ...
     }
```

Listing 2.37: Headers of the ramp and control functions.

As can be seen in the Listing, the motion controller operates over the generic AxisDictionary trait and also features a **global_disable** option to disable motion in the case a failsafe mechanism triggers.

2.9 Development of the Control Application

Control software was developed to ease testing of the stepper motor controller functionality. The control software was also developed in the Rust programming language, utilizing a crate for TUI (Terminal User Interface) and the shared components library from the bare-metal firmware. Initially, we aimed to develop a fully-featured GUI (Graphical User Interface) based on the GTK framework, but this work hasn't left the prototyping stage as the learning curve for integrating GTK is quite steep. In the Figure 2.39 we can see the control application, and in the Figure 2.40, there is a screenshot of the proposed GUI developed with the GTK framework.

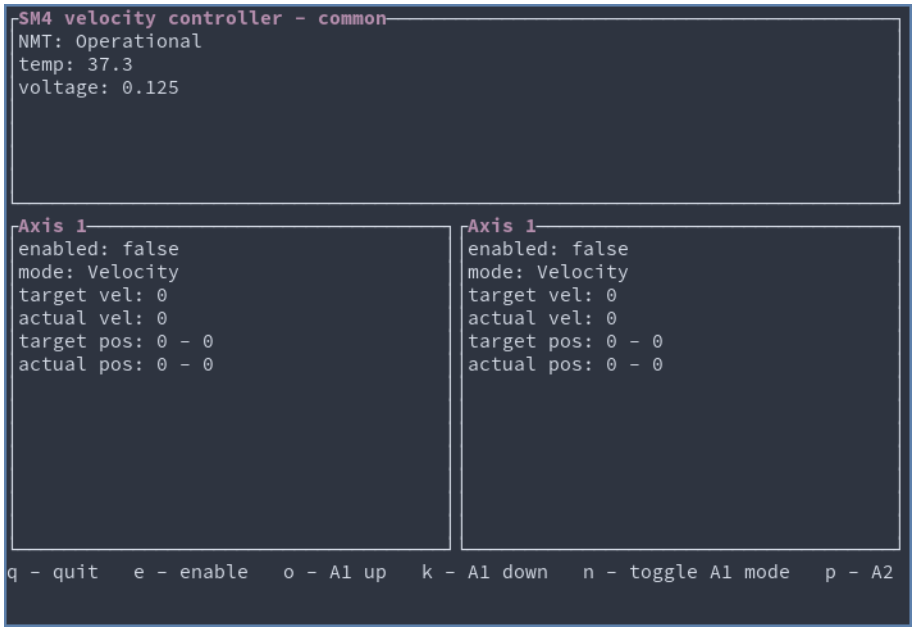


Fig. 2.39: The TUI control application.

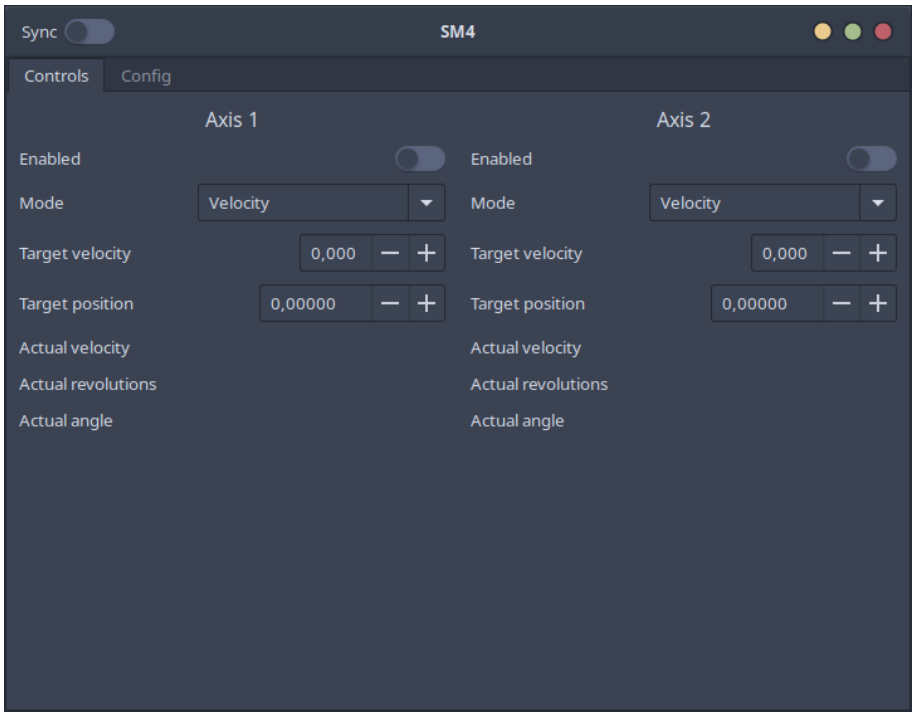


Fig. 2.40: The proposed GUI control application.

For communication with the stepper motor controller, the control application utilizes CANOpen protocol. It periodically transmits **SYNC** frames and displays the contents of the received **PDO**s. On various keypresses, it changes the values of the control variables (such as axis target velocity and target position) and sends them to the controller. This way, it can be easily tested whether the controller works as expected.

In the future, apart from developing a proper GUI for the control application, we want to add support for all the remaining communication interfaces and configuring all the parameters of the stepper motor controller.

3 Results

This chapter discusses the implemented functionality, features and the final state of the project in general. Apart from the final project state, two demonstrations are showcased - one of them being a small mobile robot for indoor mapping, and the second one being a stepper motor driven linear rail useful, for example, for camera movement.

3.1 Final Project State

This section describes the final state of the SM4 stepper motor controller project. In the Figures below, we can see both of the manufactured PCB revisions. There were two pieces of each of the PCB revisions manufactured. Furthermore, the **API!**s for all of the buses are described.

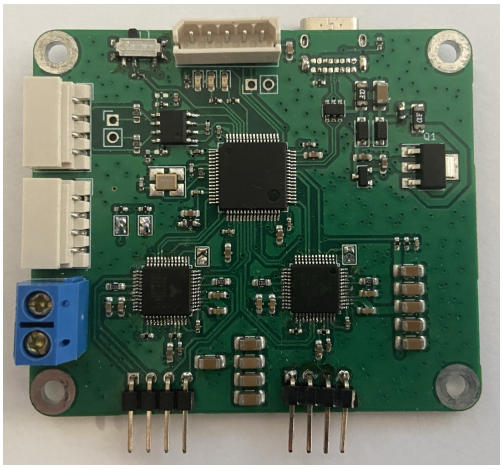


Fig. 3.1: The manufactured revision 1 PCB.

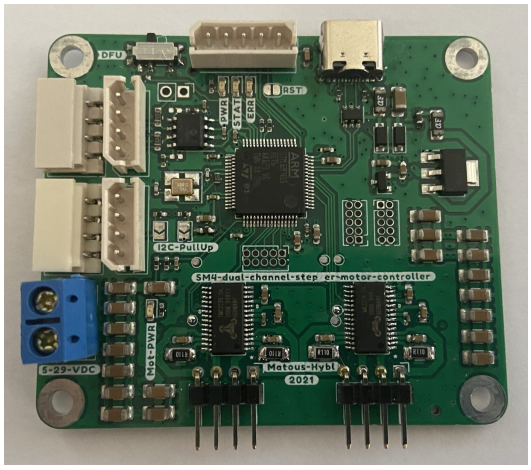


Fig. 3.2: The manufactured revision 2 PCB.

In general, two revisions of the hardware have been developed, but only the first one has been used for firmware development due to time constraints. The majority of the requirements have been fulfilled. The stepper motor controller is capable of independently driving two stepper motors in either velocity or position mode. Thanks to the Trinamic driver ICs, the motor operation is silent. The driver now utilizes simulated encoders to measure output shaft position. Given the controller's design, it can be controlled using the CANOpen protocol or via I^2C and can be configured using the USB interface.

3.1.1 Requirements Fulfilment

At the beginning of the development, we specified requirements in the Section 2.6. The majority of the requirements have been fulfilled, but we will describe the unfulfilled requirements alongside their current fulfilment status in the Table 3.1.

ID	Requirement	Fulfilment status
FR-02	When multiple communication interfaces are connected, the system shall prioritize CAN bus, then I2C. USB has the lowest priority.	We haven't been able to find a mechanism for prioritizing, we believe that this requirement might be changed as it should be the integrator's responsibility to choose only one of the communication interfaces.
FR-04	All relevant values (currents, timings, limits, etc.) shall be configurable via USB or CANOpen SDO protocol.	Now, only a subset of the values is configurable. More values (such as the configuration of communication) will be implemented in future revisions.
NFR-03	The controller shall be configurable using a program for personal computers.	The software for personal computers was developed, but the configuration part is missing as of now.
NFR-6	The firmware should utilize software in the loop integration testing for QA	Some software in the loop testing has been drafted with the Encoder abstraction, but more of the software in the loop testing will be done in the future.
NFR-7	The firmware shall be properly documented.	Some parts of the firmware are already documented via the inline documentation, but the majority of the other code is not, but it will be done in the future revision.

Tab. 3.1: Unfulfilled requirements

3.1.2 CANOpen API

As we described in the Sections on firmware development (2.8) and the one on CANOpen (2.2.1), the firmware utilizes the Object Dictionary as the central con-

figuration storage. From the higher level systems, this Object Dictionary can be accessed via the SDO protocol using indexes and subindexes which are specified in the Table D.1 in the Appendix D.

Apart from accessing the Object Dictionary using SDOs, some important control and configuration values have been made accessible via the PDO protocol. The structure of the PDOs can be seen in the Tables D.3, D.4, D.5, D.6 in the Appendix D. For completeness, the description of the PDOs can be found in the Table 3.2.

Name	Contents
TxPDO1	Contains device status - battery voltage and temperature
TxPDO2	Contains actual velocities of both axes
TxPDO3	Contains actual position of axis 1
TxPDO4	Contains actual position of axis 2
RxPDO1	Sets axis mode and enables axes
RxPDO2	Sets target velocities for both axes
RxPDO3	Sets target position for axis 1
RxPDO4	Sets target position for axis 2

Tab. 3.2: Simplified PDO contents

Alongside the SDO and PDO protocols, also the SYNC and NMT protocols have been implemented. Note that the device must reach the NMT State Operational in order to enable motion control.

3.1.3 I²C API

For the controller to be usable with the BPC-PRP course, we developed a simplified API for the I²C bus. Using this API, it is only possible to set the axes mode, enable or disable them and then to set control variables and read them. In the course, the students utilize control in the velocity mode and then acquire the positions of both axes for odometry calculation. In the previous Sections, we described that in order for motion control to work, the driver needs to reach NMT State Operational. With the I²C API, this transition is performed automatically when the axes are enabled.

The API can be found in the Table 3.3 and is based on the I²C protocol described in the Figures 2.13 and 2.12.

Address	R/W	Length [B]	Description
0x10	R/W	2	axis mode and axis enable
0x2X	R/W	4	set or read axis velocity, where X denotes the axis (1, 2)
0x3X	R/W	8	set or read axis position, where X denotes the axis (1, 2)
0x40	W	8	set velocity for both axes
0x50	R	16	read position from both axes

Tab. 3.3: I^2C API

3.1.4 USB API

The implemented USB API was designed to only allow for configuring the device as per the requirements. The protocol is fairly simple. There are two types of messages - a request and a transfer message, that can be seen in the Figures 3.3 and 3.4.

When the higher-level device wants to write a value to the Object Dictionary, it simply sends the data transfer message, where the OD KEY contains the index and the subindex. On the other hand, when the higher-level device wants to read a value from the Object Dictionary, it sends the data request message, and the controller responds with the data transfer message containing the requested value.

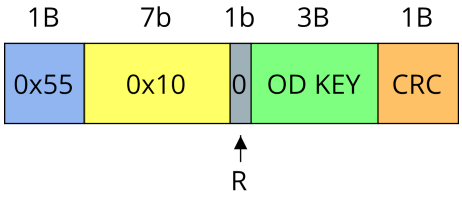


Fig. 3.3: A data request in the USB API.

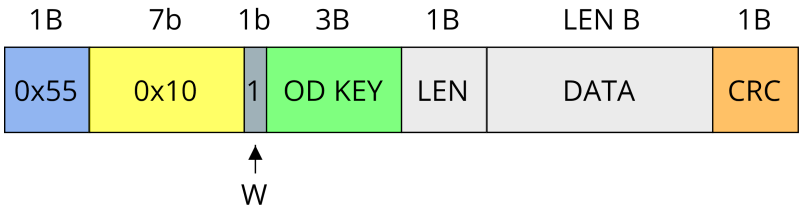


Fig. 3.4: A data transfer in the USB API.

3.1.5 Communication Failure Failsafe

A simple failsafe for the cases when the communication stops was employed. The failsafe triggers when no control message is received for a specific amount of time. Therefore it is vital to send the control data periodically. This is true for both the CANOpen and I^2C APIs.

3.1.6 Control Application

A simple control application utilizing the TUI and CANOpen protocol was developed. Controlling the stepper motor controller both in velocity and position mode is implemented, but it is not possible to change controller parameters, such as controller constants. In the future, a fully-featured control application with proper GUI, support for all the buses and configuration will be developed in order to allow for seamless configuration. Adding support for flashing via USB DFU is planned too.

3.1.7 Takeaways for Future Revisions

In this section, we describe what changes we'd made in a possible future revision. The changes are:

- Use the angular XT-30 connector for motor power.
- Use different connectors for motors.
- Use standardized 10-pin JTAG connector for SWD (Serial Wire Debug).
- Use eFuse for electronics protection.
- Add more status LEDs.
- Improve encoder connector placement, select appropriate connectors.
- Remove compatibility resistors around the CAN transceiver.
- Attempt to utilize async Rust for easier development.
- Rework the schematic to include more information, such as maximal capacitor voltages, add reasoning about component values.
- Improve current sensing circuitry to support motors with different phase currents and use sensing resistors with proper power rating. The current should be easily configurable.
- Use 4 byte key for object dictionary values.
- Properly configure bxCAN filters for CANOpen operation.

3.2 Demonstration #1 - Linear Rail Actuator for Camera Movement

The first demonstration for the SM4 motor controller is controlling a single-axis linear rail actuator. This actuator was provided by the thesis supervisor and is equipped with a NEMA17 style stepper motor that drives a belt with an attached table. Originally, this rail was meant to be used for camera movement when recording a musician.

This demonstration is controlled using a Raspberry Pi 4B, and the SM4 motor controller is controlled using the I^2C bus and the corresponding API. This showcase aims to demonstrate the controller's ability to control in position mode and, therefore, to position the table on the rail according to the commands of the higher-level system, and also the usability of I^2C bus and API. In the future, this demonstration will be equipped with a second axis allowing for tilting the mounted camera. The finished demonstrator can be seen in the Figure 3.5.

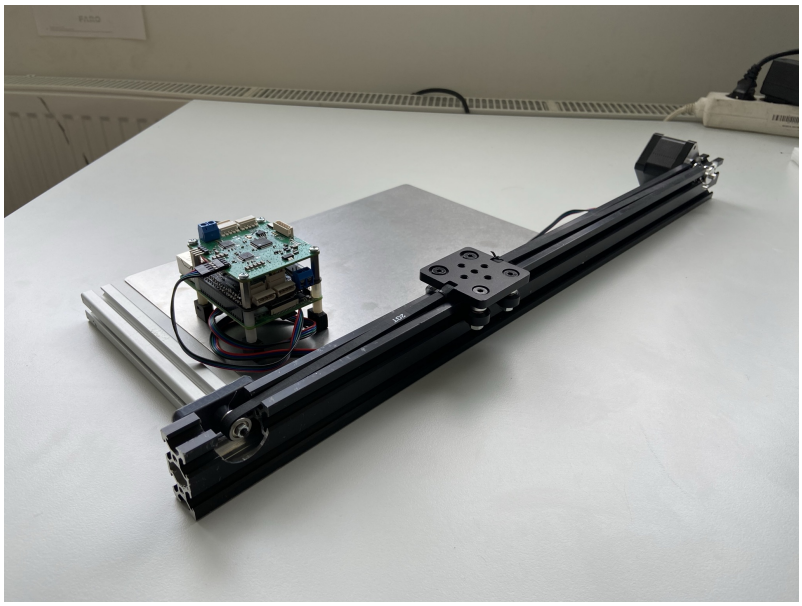


Fig. 3.5: Linear rail for camera movement - the first demonstration of the SM4 stepper motor controller.

3.3 Demonstration #2 - Small Mobile Robot for Indoor Mapping

> Any exploration program which "just happens" to include a new launch vehicle is, de facto, a launch vehicle program.
>
> (alternate formulation) The three keys to keeping a new human space program affordable and on schedule:
> 1) No new launch vehicles.
> 2) No new launch vehicles.
> 3) Whatever you do, don't develop any new launch vehicles.
>
> Akin's Laws of Spacecraft Design[100]

The second demonstration, where we showcase the SM4 stepper motor controller, is a small differentially driven robot aimed for indoor mapping and self-localization. The chassis is differential with a motor on two sides of the robot, which showcases the driver's ability to control both of the motors and calculate odometry. Apart from the driver itself, the robot features a Raspberry Pi 4B SBC, 4-cell Li-Ion battery, step-down converter, and a simple planar LIDAR for the mapping task. A block diagram of this demonstration can be seen in the Figure 3.6

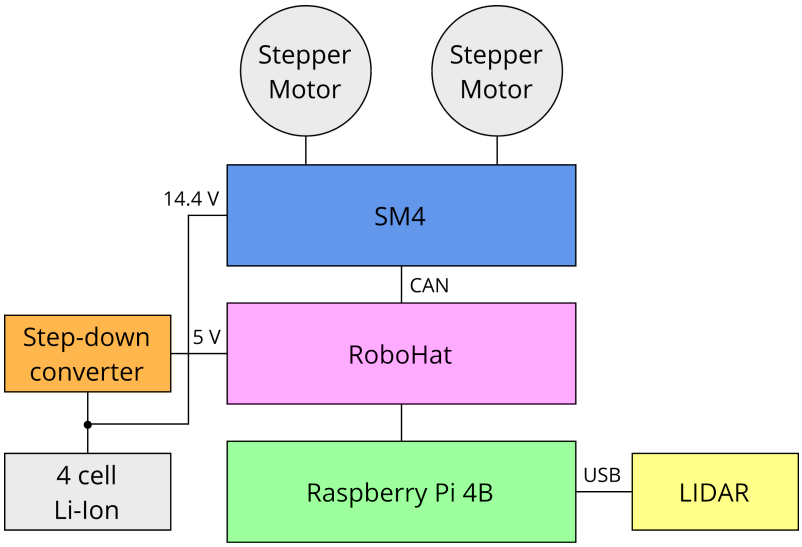


Fig. 3.6: The block diagram of the robot used for the second demonstration.

It is projected that the robot will be utilized for algorithm demonstration as part of the MPC-MAP - Advanced Mapping and Self-Localization for Robotics course. The finalized robot can be seen in the Figure 3.7.

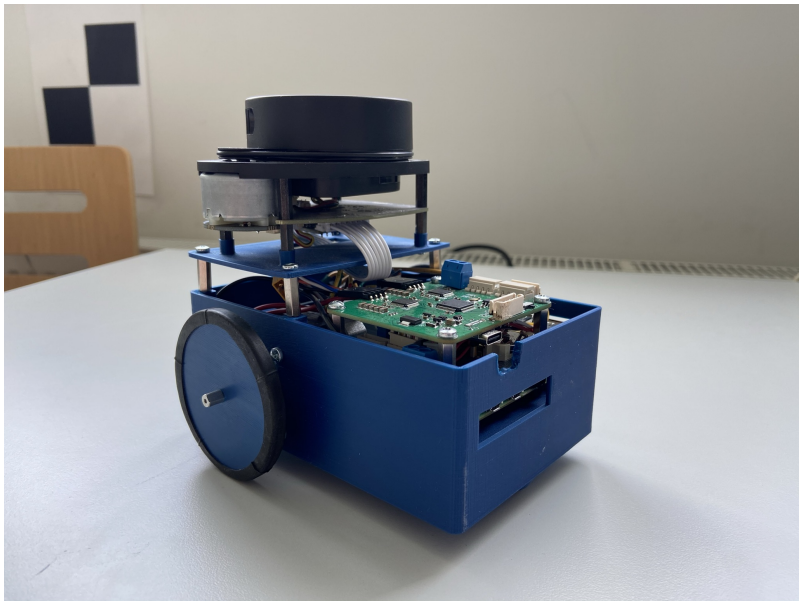


Fig. 3.7: The MAP-bot robot - the second demonstration of the SM4 stepper motor controller.

Conclusion, Discussion and Future work

In this thesis, we described the development of a dual-channel stepper motor controller we named SM4. We developed both the hardware and software. As for the hardware development, we utilized the STM32F405 MCU and Trinamic stepper motor driver ICs as the basis of the design. As for PCB design, we utilized a 4-layer PCB to decrease the development time, and we utilized the JLCPCB's manufacturing service alongside the PCB assembly service. Two revisions of the PCB were designed and manufactured, two of each revision PCBs were assembled. Even though the second revision of the PCB is more advanced than the first one, there is still plenty of room for improvements, which we came across during the development. The schematic and PCBs were designed in the KiCAD EDA suite, which proved useful, given the fact that KiCAD has a large footprint and symbol library.

As for the development of the software, we utilized the Rust programming language for both the bare-metal firmware and the control application. Unfortunately, the firmware, as of now, supports only the first revision of the hardware. Developing the firmware in Rust proved useful, as there is great support for developing on bare-metal, given there is a large community for developing device drivers, HALs, and tooling. During the development, we utilized many of the language's features, especially when developing abstractions over the hardware where we utilized traits and generics. We believe that given the developed abstractions the firmware can be easily extended and improved.

We believe that the tooling that currently exists surpasses the tooling available for other languages and ecosystems. The language's features provide memory and data race safety for embedded systems while not impeding the code size or performance. Given our experience with developing embedded firmware for this project and several other ones described earlier, we firmly believe that the Rust programming language is the right way forward as it brings features never deemed possible for embedded systems development.

Apart from the firmware, we also developed a simple control application for personal computers that is now capable of only controlling the stepper motor controller but not of configuring it. The control application is now able to control the controller's axis in both velocity and position modes.

Even though there is still a lot of work to be done, we believe that the controller is currently usable and be deployed, for example, to be part of the BPC-PRP course.

3.4 Future work

We are hoping to continue working on the stepper motor controller in the future. We would like to greatly extend and improve both the hardware and software (and both the firmware and the control application). Some of the requirements were not completely fulfilled, and we are aiming to revisit them. A big part of the future development will be finishing the documentation and automated testing. Extending the firmware with support for the second hardware revision will be a priority since the new stepper motor controller IC is much more powerful than the one in the first revision. We are also planning to try integrating real hardware encoders to try out the suitability of the abstractions and the ability to control the motors with proper feedback. We will also aim for full conformance with the CANOpen standard to allow for seamless integration with other systems. We are also looking into continuing the development using Rust programming language for embedded systems, either by developing more projects using it or contributing to the existing ecosystem by developing the tooling and libraries.

Bibliography

1. GREIG, Adam. *adamgreig/stm32ral* [online]. 2021 [visited on 2021-05-07]. Available from: `https://github.com/adamgreig/stm32ral`. original-date: 2018-06-25T22:35:57Z.
2. *embassy-rs/embassy* [online]. embassy-rs, 2021 [visited on 2021-05-08]. Available from: `https://github.com/embassy-rs/embassy`. original-date: 2020-09-22T16:04:20Z.
3. COHEN, Perry. *Tech Market Madness: The State of Embedded Programming Languages* [Embedded Computing Design] [online] [visited on 2021-05-06]. Available from: `http://www.embeddedcomputing.com/technology/software-and-os/ides-application-programming/tech-market-madness-analyst-insights-for-engineers-embedded-programming-languages`.
4. DUBOIS, Chantelle. *Programming Languages for Embedded Systems 101: Background and Resources - News* [online] [visited on 2021-05-06]. Available from: `https://www.allaboutcircuits.com/news/programming-languages-for-embedded-systems-101-background-and-resources/`.
5. Embedded software. In: *Wikipedia* [online]. 2021 [visited on 2021-05-06]. Available from: `https://en.wikipedia.org/w/index.php?title=Embedded_software&oldid=1021501630`. Page Version ID: 1021501630.
6. CIRCUITPYTHON. *CircuitPython* [CircuitPython] [online]. 2021 [visited on 2021-05-06]. Available from: `https://circuitpython.org/`.
7. CHROMIUM PROJECTS. *Memory safety - The Chromium Projects* [The Chromium Projects] [online] [visited on 2021-03-31]. Available from: `https://www.chromium.org/Home/chromium-security/memory-safety`.
8. STENBERG, Daniel. *half of curl's vulnerabilities are C mistakes | daniel.haxx.se* [online] [visited on 2021-05-05]. Available from: `https://daniel.haxx.se/blog/2021/03/09/half-of-curls-vulnerabilities-are-c-mistakes/`.
9. SYNOPSYS. *The Heartbleed Bug* [The Heartbleed Bug] [online]. 2020-03-06 [visited on 2021-03-31]. Available from: `https://heartbleed.com/`.
10. FAIGL, Jan. *Program | Student Conference on Planning in Artificial Intelligence and Robotics* [online] [visited on 2021-03-31]. Available from: `./index.html`.
11. BURIAN, Frantisek. *KM2.render.png (PNG Image, 1024 × 735 pixels) — Scaled (92%)* [online] [visited on 2021-04-03]. Available from: `https://student.robotika.ceitec.vutbr.cz/PCBS/KM2/raw/master/bin/RevB/KM2.render.png`.

12. HYBL, Matous. *Robotics-BUT/KM3-rs* [online]. BUT FEEC Robotics, 2020 [visited on 2021-03-31]. Available from: https://github.com/Robotics-BUT/KM3-rs. original-date: 2020-10-18T19:51:28Z.

13. TROPICAL LABS. *Mechaduino – Tropical Labs* [Tropical Labs] [online]. 2021 [visited on 2021-04-05]. Available from: https://tropical-labs.com/mechaduino/.

14. BRAUN, Hanno. *Flott - Motion Control in Rust* [Flott] [online] [visited on 2021-04-05]. Available from: https://flott-motion.org/.

15. EARL, Bill. *All About Stepper Motors* [Adafruit Learning System] [online] [visited on 2021-04-30]. Available from: https://learn.adafruit.com/all-about-stepper-motors/what-is-a-stepper-motor.

16. CARMINE FIORE. *Stepper Motors: Types, Uses and Working Principle | Article | MPS* [MPS] [online]. 2021 [visited on 2021-04-30]. Available from: https://www.monolithicpower.com/en/stepper-motors-basics-types-uses.

17. REPRAP. *NEMA Motor - RepRap* [online] [visited on 2021-05-10]. Available from: https://reprap.org/wiki/NEMA_Motor.

18. *NEMA 17HS4401 Bipolar Stepper Motor* [online] [visited on 2021-05-10]. Available from: https://www.cytron.io/p-nema-17hs4401-bipolar-stepper-motor.

19. TRINAMIC. *Microstepping* [Trinamic] [online] [visited on 2021-05-02]. Available from: https://www.trinamic.com/technology/motor-control-technology/microstepping/.

20. TRINAMIC. *Chopper Modes* [Trinamic] [online] [visited on 2021-04-30]. Available from: https://www.trinamic.com/technology/motor-control-technology/chopper-modes/.

21. TRINAMIC. *Trinamic StallGuard and CoolStep Technology* [Trinamic] [online] [visited on 2021-05-02]. Available from: https://www.trinamic.com/technology/motor-control-technology/stallguard-and-coolstep/.

22. CAN IN AUTOMATION. *CAN in Automation (CiA): CANopen* [CAN In Automation] [online]. 2021 [visited on 2021-05-04]. Available from: https://www.can-cia.org/canopen/.

23. CANopen. In: *Wikipedia* [online]. 2021 [visited on 2021-05-04]. Available from: https://en.wikipedia.org/w/index.php?title=CANopen&oldid=1021343213. Page Version ID: 1021343213.

24. ST. MICHAEL, Stephen. *Introduction to CAN (Controller Area Network) - Technical Articles* [All About Circuits] [online]. 2019-02-19 [visited on 2021-05-03]. Available from: https://www.allaboutcircuits.com/technical-articles/introduction-to-can-controller-area-network/.

25. PIEMBSYSTECH. *CAN Protocol* [PiEmbSysTech] [online] [visited on 2021-05-03]. Available from: https://piembsystech.com/can-protocol/.

26. SFUPTOWNMAKER. *I2C* [Sparkfun - start something] [online]. 2021 [visited on 2021-05-05]. Available from: https://learn.sparkfun.com/tutorials/i2c/all.

27. VALDEZ, Jonathan; BECKER, Jared. Understanding the I2C Bus. 2015, p. 8.

28. USB. In: *Wikipedia* [online]. 2021 [visited on 2021-05-05]. Available from: https://en.wikipedia.org/w/index.php?title=USB&oldid=1021306944. Page Version ID: 1021306944.

29. STMICRO. *AN4879 - USB hardware and PCB guidelines using STM32 MCUs* [online]. STMicro, 2018 [visited on 2021-05-05]. Available from: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwito5urq7LwAhXPFXcKHWKxDfAQFjAEegQIBhAD&url=https%3A%2F%2Fwww.st.com%2Fresource%2Fen%2Fapplication_note%2Fdm00296349-usb-hardware-and-pcb-guidelines-using-stm32-mcus-stmicroelectronics.pdf&usg=AOvVaw1eXmSESQxeC6yMXNhB_H59.

30. *Servo* [online] [visited on 2021-05-16]. Available from: https://servo.org/.

31. *Rust and C++ interoperability - The Chromium Projects* [online] [visited on 2021-05-05]. Available from: https://www.chromium.org/Home/chromium-security/memory-safety/rust-and-c-interoperability.

32. KLABNIK, Steve; NICHOLS, Carol. *The Rust Programming Language - The Rust Programming Language* [online] [visited on 2021-04-19]. Available from: https://doc.rust-lang.org/book/title-page.html.

33. *Crates and source files - The Rust Reference* [The Rust Reference] [online]. 2021 [visited on 2021-05-06]. Available from: https://doc.rust-lang.org/reference/crates-and-source-files.html#the-no_main-attribute.

34. WG, Rust Embedded Devices. *Introduction - The Embedded Rust Book* [online]. 2021 [visited on 2021-05-06]. Available from: https://docs.rust-embedded.org/book/intro/index.html.

35. MUNNS, James. *From Zero to main(): Bare metal Rust* [Interrupt] [online]. 2019-12-17 [visited on 2021-05-06]. Available from: https://interrupt.memfault.com/blog/zero-to-main-rust-1.

36. WG, Rust Embedded Devices. *rust-embedded/alloc-cortex-m* [online]. Rust Embedded, 2021 [visited on 2021-05-06]. Available from: https://github.com/rust-embedded/alloc-cortex-m. original-date: 2016-11-19T15:34:18Z.

37. *alloc - Rust* [online] [visited on 2021-05-06]. Available from: https://doc.rust-lang.org/alloc/.

38. HUTT, Tim. *Using std in embedded Rust* [online] [visited on 2021-05-05]. Available from: http://blog.timhutt.co.uk/std-embedded-rust/index.html.

39. *Platform Support - The rustc book* [online] [visited on 2021-05-06]. Available from: https://doc.rust-lang.org/nightly/rustc/platform-support.html.

40. RAHUL, CreativeCoder. *How to run Rust on Arduino Uno - Our first blink* [online] [visited on 2021-05-05]. Available from: https://creativcoder.dev/rust-on-arduino-uno.

41. MABIN, Scott. *MabezDev/xtensa-rust-quickstart* [online]. 2021 [visited on 2021-05-06]. Available from: https://github.com/MabezDev/xtensa-rust-quickstart. original-date: 2019-03-05T23:57:17Z.

42. *Embedded devices working group* [online] [visited on 2021-05-07]. Available from: https://www.rust-lang.org/governance/wgs/embedded.

43. RUST EMBEDDED DEVICES WG. *Introduction - Discovery* [online] [visited on 2021-05-07]. Available from: https://docs.rust-embedded.org/discovery/index.html.

44. RUST EMBEDDED DEVICES WG. *Preface - The Embedonomicon* [online] [visited on 2021-05-07]. Available from: https://docs.rust-embedded.org/embedonomicon/index.html.

45. RUST EMBEDDED DEVICES WG. *rust-embedded/svd2rust* [online]. Rust Embedded, 2021 [visited on 2021-05-06]. Available from: https://github.com/rust-embedded/svd2rust. original-date: 2016-10-09T02:47:52Z.

46. *stm32-rs/stm32-rs* [online]. stm32-rs, 2021 [visited on 2021-05-06]. Available from: https://github.com/stm32-rs/stm32-rs. original-date: 2017-05-31T02:43:32Z.

47. RUST EMBEDDED DEVICES WG. *rust-embedded/embedded-hal* [online]. Rust Embedded, 2021 [visited on 2021-05-06]. Available from: https://github.com/rust-embedded/embedded-hal. original-date: 2017-06-09T20:57:29Z.

48. *stm32-rs/stm32f4xx-hal* [online]. stm32-rs, 2021 [visited on 2021-05-06]. Available from: https://github.com/stm32-rs/stm32f4xx-hal. original-date: 2018-10-01T22:03:48Z.

49. APARICIO, Jorge. *Memory safe DMA transfers - Embedded in Rust* [online] [visited on 2021-05-08]. Available from: https://blog.japaric.io/safe-dma/.

50. *Static items - The Rust Reference* [online] [visited on 2021-05-08]. Available from: https://doc.rust-lang.org/reference/items/static-items.html.

51. EGGER, Daniel. *A look into ways to implement and share data with interrupt handlers in Rust (Update 1)* [The subconscious mumblings of therealprof] [online] [visited on 2021-05-05]. Available from: https://therealprof.github.io/blog/interrupt-comparison/.

52. *Preface - Real-Time Interrupt-driven Concurrency* [online] [visited on 2021-05-06]. Available from: https://rtic.rs/0.5/book/en/.

53. APARICIO, Jorge. *Concurrency Patterns in Embedded Rust* [online] [visited on 2021-05-05]. Available from: https://ferrous-systems.com/blog/embedded-concurrency-patterns/.

54. SCHATTINGER, Ben. *Async/Await for AVR with Rust* [Ben Schattinger] [online]. 2020-07-25 [visited on 2021-05-05]. Available from: https://lights0123.com/blog/2020/07/25/async-await-for-avr-with-rust/.

55. *stm32-rs/stm32-usbd* [online]. stm32-rs, 2021 [visited on 2021-05-06]. Available from: https://github.com/stm32-rs/stm32-usbd. original-date: 2019-03-24T19:30:20Z.

56. *stm32-rs/stm32-eth* [online]. stm32-rs, 2021 [visited on 2021-05-06]. Available from: https://github.com/stm32-rs/stm32-eth. original-date: 2018-02-06T02:18:56Z.

57. RUST EMBEDDED DEVICES WG. *Interoperability - The Embedded Rust Book* [online] [visited on 2021-05-08]. Available from: https://docs.rust-embedded.org/book/interoperability/index.html.

58. WOLFF, Jared. *Embedding Rust Into Zephyr Using Cbindgen* [online] [visited on 2021-05-05]. Available from: https://www.jaredwolff.com/embedding-rust-into-zephyr-using-cbindgen/.

59. PROBE RS PROJECT. *probe-rs/probe-rs* [online]. probe.rs, 2021 [visited on 2021-05-09]. Available from: https://github.com/probe-rs/probe-rs. original-date: 2019-02-10T20:13:01Z.

60. PROJECT, Probe RS. *probe-rs/cargo-flash* [online]. probe.rs, 2021 [visited on 2021-05-09]. Available from: https://github.com/probe-rs/cargo-flash. original-date: 2020-01-29T19:52:01Z.

61. PROBE RS PROJECT. *probe-rs/cargo-embed* [online]. probe.rs, 2021 [visited on 2021-05-09]. Available from: https://github.com/probe-rs/cargo-embed. original-date: 2020-04-09T17:30:24Z.

62. KNURLING PROJECT. *knurling-rs/defmt* [online]. Knurling, 2021 [visited on 2021-05-09]. Available from: https://github.com/knurling-rs/defmt. original-date: 2020-07-24T16:04:15Z.

63. KNURLING PROJECT. *knurling-rs/probe-run* [online]. Knurling, 2021 [visited on 2021-05-09]. Available from: https://github.com/knurling-rs/probe-run. original-date: 2020-08-14T08:46:46Z.

64. APARICIO, Jorge. *Using GDB and defmt to debug embedded programs* [online] [visited on 2021-05-09]. Available from: https://ferrous-systems.com/blog/gdb-and-defmt/.

65. KNURLING PROJECT. *knurling-rs/flip-link* [online]. Knurling, 2021 [visited on 2021-05-09]. Available from: https://github.com/knurling-rs/flip-link. original-date: 2020-09-07T17:05:51Z.

66. APARICIO, Jorge. *Testing an embedded application* [online] [visited on 2021-05-09]. Available from: https://ferrous-systems.com/blog/test-embedded-app/.

67. APARICIO, Jorge. *Testing a Hardware Abstraction Layer (HAL)* [online] [visited on 2021-05-09]. Available from: https://ferrous-systems.com/blog/defmt-test-hal/.

68. SCHIEVINK, Jonas. *Running hardware-in-the-loop tests on GitHub Actions* [online] [visited on 2021-05-09]. Available from: https://ferrous-systems.com/blog/gha-hil-tests/.

69. VAHTER, Andres. *Release Rust embedded firmware using Github Actions* [Andres Vahter on Svbtle] [online] [visited on 2021-05-05]. Available from: https://andres.svbtle.com/release-rust-embedded-firmware-using-github-actions.

70. CHIOVOLONI, Thom. *A Few Github Action "Recipes" for Rust* [shift.click] [online]. 2020-09-06 [visited on 2021-05-05]. Available from: https://shift.click/blog/github-actions-rust/.

71. KOOPMAN, Phillip. *Better Embedded System Software* [online]. 1.st Edition, revised 2021. Carnegie Mellon University, 2010 [visited on 2021-02-24]. ISBN 979-8596008050. Available from: http://www.koopman.us/.

72. STMICRO. *STM32F405RG - High-performance foundation line, Arm Cortex-M4 core with DSP and FPU, 1 Mbyte of Flash memory, 168 MHz CPU, ART Accelerator - STMicroelectronics* [online] [visited on 2021-04-11]. Available from: https://www.st.com/en/microcontrollers-microprocessors/stm32f405rg.html.

73. STMICRO. *en.bd_stm32f405_1mb.jpg (JPEG Image, 700 × 754 pixels)* [online] [visited on 2021-04-11]. Available from: https://www.st.com/content/ccc/fragment/product_related/rpn_information/product_circuit_diagram/group0/d1/5f/28/7e/77/7a/49/c0/bd_stm32f405_1m/files/bd_stm32f405_1mb.jpg/_jcr_content/translations/en.bd_stm32f405_1mb.jpg.

74. PRUSA, Josef. *Original Prusa i3 MK3 is out! And it's bloody smart!* [Prusa Printers] [online]. 2017-09-22 [visited on 2021-05-02]. Available from: https://blog.prusaprinters.org/original-prusa-i3-mk3-bloody-smart_7201/. Section: Highlights.

75. PRUSA, Josef. *Original Prusa MINI is here: Smart and compact 3D printer for everyone!* [Prusa Printers] [online]. 2019-10-12 [visited on 2021-05-02]. Available from: https://blog.prusaprinters.org/original-prusa-mini-is-here-smart-and-compact-3d-printer_30887/. Section: Highlights.

76. 3DADDICT. *Stepper Driver Comparison 3D Printer Upgrade* [3DAddict] [online]. 2020-10-04 [visited on 2021-05-02]. Available from: https://3daddict.com/stepper-driver-comparison-3d-printer-upgrade/.

77. TEXAS INSTRUMENTS. *DRV8825 Stepper Motor Controller IC* [online]. Texas Instruments, 2014 [visited on 2021-05-02]. Available from: https://www.ti.com/lit/ds/symlink/drv8825.pdf?ts=1619898319490&ref_url=https%253A%252F%252Fwww.google.com%252F.

78. ALLEGRO MICROSYSTEMS. *A4988 DMOS Microstepping Driver with Translator And Overcurrent Protection* [online]. Allegro Microsystems, 2014 [visited on 2021-05-02]. Available from: https://www.pololu.com/file/0J450/a4988_DMOS_microstepping_driver_with_translator.pdf.

79. TRINAMIC. *TMC2100-LA Datasheet* [online]. Trinamic, 2018 [visited on 2021-05-02]. Available from: https://www.trinamic.com/fileadmin/assets/Products/ICs_Documents/TMC2100_datasheet_Rev1.08.pdf.

80. TRINAMIC. *TMC2130-LA Datasheet* [online]. Trinamic, 2018 [visited on 2021-05-02]. Available from: https://www.trinamic.com/fileadmin/assets/Products/ICs_Documents/TMC2130_datasheet.pdf.

81. TRINAMIC. *TMC2209 Datasheet* [online]. Trinamic, 2019 [visited on 2021-05-02]. Available from: https://www.trinamic.com/fileadmin/assets/Products/ICs_Documents/TMC2209_Datasheet_V103.pdf.

82. TRINAMIC. *TMC2226 Datasheet* [online]. Trinamic, 2020 [visited on 2021-05-02]. Available from: https://www.trinamic.com/fileadmin/assets/Products/ICs_Documents/TMC2226_Datasheet_V106.pdf.

83. STMICRO. *STM32F405xx STM32F407xx* [online]. STMicro, 2020 [visited on 2020-09-05]. Available from: https://www.st.com/resource/en/datasheet/stm32f405rg.pdf.

84. SALMONY, Philip. *KiCad STM32 + USB + Buck Converter PCB Design and JLCPCB Assembly (Update)* [online]. 2020 [visited on 2021-05-09]. Available from: https://www.youtube.com/watch?v=C7-8nUU6e3E.

85. STMICRO. *AN2867 - Oscillator design guide for STM8AF/AL/S, STM32 MCUs and MPUs* [online]. STMicro, 2020 [visited on 2021-10-05]. Available from: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjXke-Vhr_wAhVaPuwKHWkLCREQFjAAegQIBxAD&url=https%3A%2F%2Fwww.st.com%2Fresource%2Fen%2Fapplication_note%2Fcd00221665-oscillator-design-guide-for-stm8afals-stm32-mcus-and-mpus-stmicroelectronics.pdf&usg=AOvVaw0W61KfA6_gLIcIPUfy7taA.

86. GREATSCOTT! *The Best Protection for your Circuits? eFuse! Here is why they are awesome! EB#48* [online]. 2021 [visited on 2021-05-05]. Available from: https://www.youtube.com/watch?v=JOhQ3nsR7xo.

87. TEXAS INSTRUMENTS. *eFuse and Hot Swap Controllers* [Texas Instruments] [online]. 2021-05-05 [visited on 2021-05-05]. Available from: https://www.ti.com/power-management/power-switches/efuse-hotswap-controllers/overview.html.

88. MICROCHIP. *MCP2562 - Interface - Interface- Controller Area Network (CAN)* [Microchip] [online] [visited on 2021-05-10]. Available from: https://www.microchip.com/wwwproducts/en/MCP2562.

89. STMICRO. *TA0357 - Overview of USB Type-C and Power Delivery technologies* [online]. STMicro, 2018 [visited on 2021-10-05]. Available from: https://www.st.com/content/ccc/resource/technical/document/technical_article/group0/59/a8/53/96/fe/a8/45/35/DM00496853/files/DM00496853.pdf/jcr:content/translations/en.DM00496853.pdf.

90. THOMAS, David; HUNT, Andrew. *The pragmatic programmer, 20th anniversary edition: journey to mastery*. Second edition. Boston: Addison-Wesley, 2019. ISBN 978-0-13-595705-9.

91. MANSANET, Pablo. *eCorax - As above, so below: Bare metal Rust generics 1/2* [online] [visited on 2021-04-06]. Available from: `https://www.ecorax.net/as-above-so-below-1/`.

92. MANSANET, Pablo. *eCorax - As above, so below: Bare metal Rust generics 2/2* [online] [visited on 2021-05-05]. Available from: `https://www.ecorax.net/as-above-so-below-2/`.

93. STMICRO. *AN3969 - EEPROM emulation in STM32F40x/STM32F41x microcontrollers* [online]. STMicro, 2011 [visited on 2021-04-06]. Available from: `https://www.st.com/content/ccc/resource/technical/document/application_note/ec/dd/8e/a8/39/49/4f/e5/DM00036065.pdf/files/DM00036065.pdf/jcr:content/translations/en.DM00036065.pdf`.

94. RUST EMBEDDED DEVICES WG. *rust-embedded/cortex-m* [online]. Rust Embedded, 2021 [visited on 2021-05-11]. Available from: `https://github.com/rust-embedded/cortex-m`. original-date: 2016-09-27T23:35:04Z.

95. DUBROV, Ivan. *idubrov/eeprom* [online]. 2020 [visited on 2021-05-05]. Available from: `https://github.com/idubrov/eeprom`. original-date: 2017-09-28T06:54:38Z.

96. ASTRO. *Implement flash read/erase/program by astro · Pull Request #239 · stm32-rs/stm32f4xx-hal* [GitHub] [online]. 2020-10-12 [visited on 2021-04-15]. Available from: `https://github.com/stm32-rs/stm32f4xx-hal/pull/239`.

97. VIRKKUNEN, Matti. *mvirkkunen/usb-device* [online]. 2021 [visited on 2021-05-14]. Available from: `https://github.com/mvirkkunen/usb-device`. original-date: 2018-08-28T16:01:57Z.

98. VIRKKUNEN, Matti. *mvirkkunen/usbd-serial* [online]. 2021 [visited on 2021-05-14]. Available from: `https://github.com/mvirkkunen/usbd-serial`. original-date: 2019-05-23T15:14:29Z.

99. *obdev/v-usb* [GitHub] [online] [visited on 2021-05-14]. Available from: `https://github.com/obdev/v-usb`.

100. AKIN, David L. *Akin's Laws of Spacecraft Design* [online] [visited on 2021-02-24]. Available from: `https://spacecraft.ssl.umd.edu/akins_laws.html`.

Symbols and abbreviations

ADC	Analog to Digital Converter
AI	Artificial Intelligence
API	Application Programming Interface
ARM	Acorn RISC Machine
CAN	Controlled Area Network
CI	Continuous Integration
DC	Direct Current
DAC	Digital to Analog Converter
DMA	Direct Memory Access
DSL	Domain Specific Language
DFU	Device Firmware Update
EEPROM	Electrically Erasable Programmable Read-Only Memory
EMI	ElectroMagnetic Interference
ESD	ElectroStatic Discharge
FFI	Foreign Function Interface
FRAM	Ferroelectric Random Access Memory
GND	Ground
GPIO	General Purpose Input Output
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
IC	Integrated Circuit
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
LDO	Low DropOut

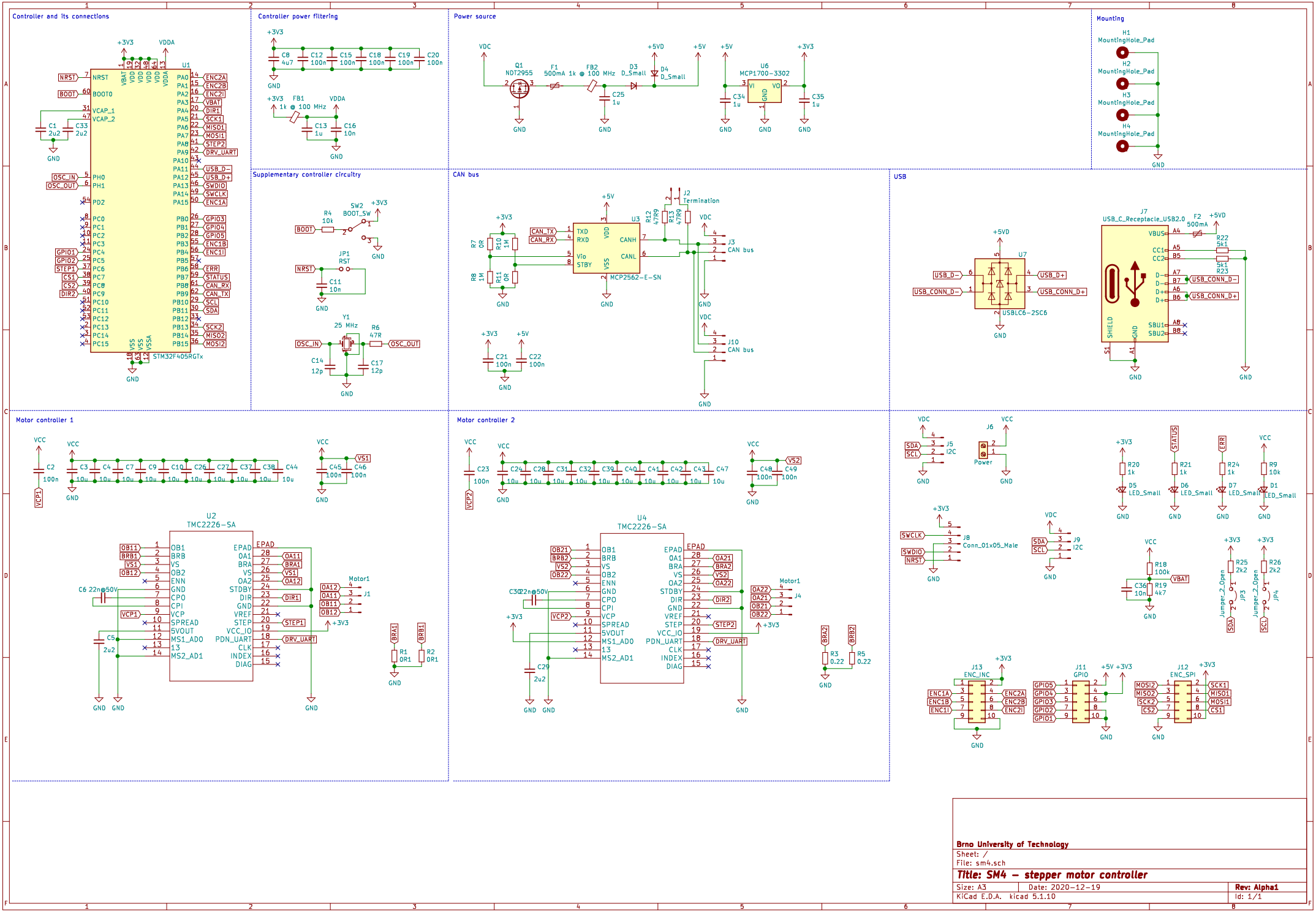
LED	Light Emitting Diode
LIDAR	Light Detection and Ranging
MCU	Microcontroller Unit
NO	Normally Open
PAC	Peripheral Access Crate
PCB	Printed Circuit Board
PTC	Positive Temperature Coefficient
pwm	PWMPulse Width Modulation
QA	Quality Assurance
RAM	Random Access Memory
RAL	Register Access Layer
RISC	Reduced Instruction Set Computer
RMS	Root-Mean Square
RPS	Revolutions per Second
RTIC	Real-Time Interrupt Driven Concurrency
RTT	Real-Time Transfer
SBC	Single Board Computer
SPI	Serial Peripheral Interface
SPDT	Single Pole Double Throw
SVD	System View Description
TUI	Terminal User Interface
UB	Undefined Behavior
USB	Universal Serial Bus

List of appendices

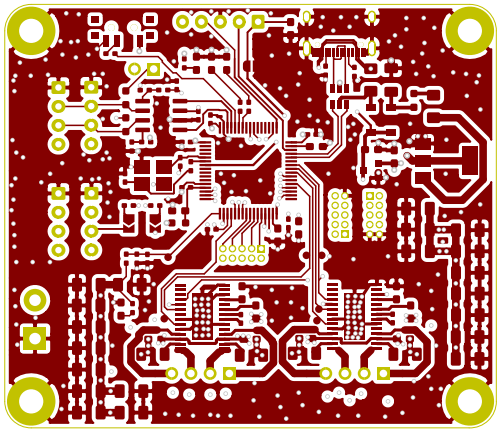
A Contents of the Enclosed Electronic Medium

- `/` — Root
  - `.github` — CI workflows and auxiliary files
  - `Hardware` — Hardware resources for both HW revisions
    - `Cube` — STM32CubeMX project for pin assigment
    - `Docs` — Datasheets of components
    - `Libs` — KiCAD component and footprint libraries
    - `rev1` — KiCAD project for the first revision
    - `rev2` — KiCAD project for the second revision
  - `Poster` — Source file for a future poster
    - `poster_template.pptx`
  - `Software` — Software projects and source code
    - `controller` — The control software
    - `embedded` — Workspace containing cross-compiled projects
      - `firmware` — The motor controller's firmware
      - `testsuite` — Tests for the motor controller's firmware
    - `shared` — Project with code shared between firmware and controller
    - `Cargo.toml` — Cargo project file
    - `Cargo.lock` — Cargo project file lock
    - `LICENSE-MIT` — Software License
    - `README.md` — Read me for software
  - `Thesis` — Source code for this thesis
  - `README.md` — Readme for this master's project

B Schematic of the Second Electronics Revision


Controller and its connections
Controller power filtering
Power source
Mounting
Supplementary controller circuitry
CAN bus
USB
Motor controller 1
Motor controller 2
STM32F405RGTx
TMC2226-SA
MCP2562-E-SN
USBLC6-2SC6
Brno University of Technology
Sheet: /
File: sm4.sch
Title: SM4 – stepper motor controller
Size: A3
Date: 2020-12-19
Rev: Alpha1
KiCad E.D.A. kicad 5.1.10
Id: 1/1

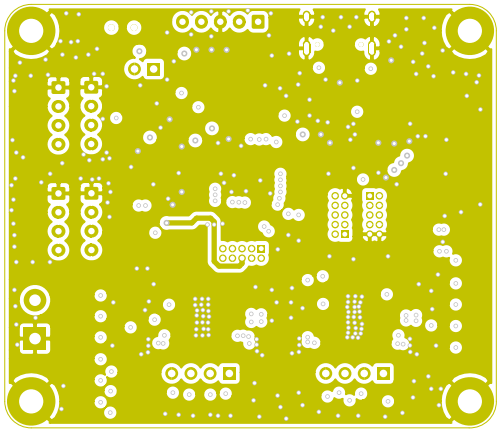
C PCB of the Second Electronics Revision



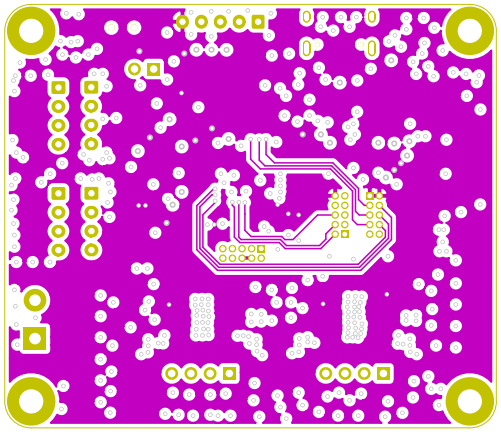
Sheet:
File: sm4.kicad_pcb

Title:

Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad 5.1.10		Id: 1/1



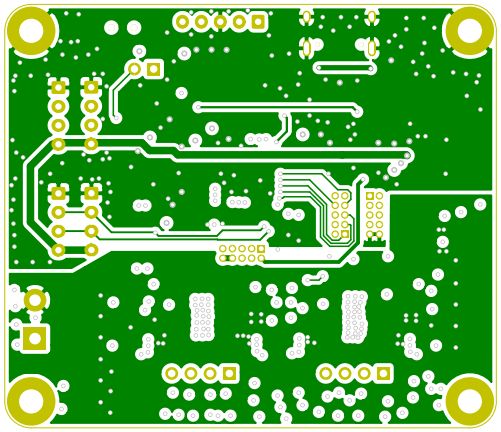
Sheet:		
File: sm4.kicad_pcb		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad 5.1.10		Id: 1/1



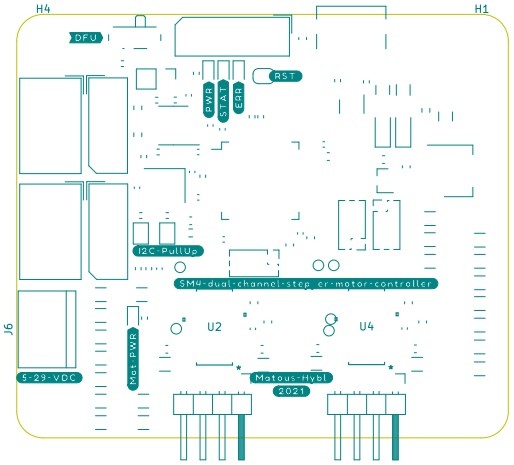
Sheet:
File: sm4.kicad_pcb

Title:

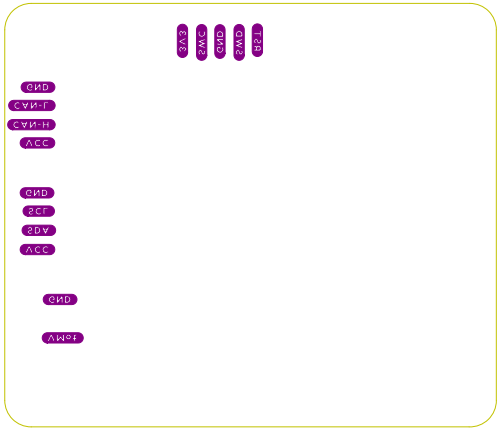
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad 5.1.10		Id: 1/1



Sheet:			
File: sm4.kicad_pcb			
Title:			
Size: A4	Date:		**Rev:**
KiCad E.D.A. kicad 5.1.10			Id: 1/1



Sheet:		
File: sm4.kicad_pcb		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad 5.1.10		Id: 1/1



Sheet:		
File: sm4.kicad_pcb		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad 5.1.10		Id: 1/1

D CANOpen PDOs and Object Dictionary

Index	Subindex	Type	Length [B]	Description
2000	1	f32	4	Battery Voltage in Volts
2000	2	f32	4	MCU Temperature in °C
2[1,2]00	1	AxisMode	1	mode of the axis - 0 for velocity, 1 for position
2[1,2]00	2	bool	1	axis enabled
2[1,2]00	3	f32	4	target axis velocity in RPS
2[1,2]00	4	f32	4	actual axis velocity in RPS
2[1,2]00	5	i32	4	target axis position - revolutions
2[1,2]00	6	u32	4	target axis position - angle
2[1,2]00	7	i32	4	actual axis position - revolutions
2[1,2]00	8	u32	4	actual axis position - angle
2[1,2]00	9	f32	4	target ramp acceleration in RPS per second
2[1,2]00	10	bool	1	velocity controller bypass enabled
2[1,2]00	11	f32	4	current applied to motor winding during acceleration in Amperes
2[1,2]00	12	f32	4	current applied to motor winding when idle in Amperes
2[1,2]00	13	f32	4	current applied to motor winding when moving with constant speed in Amperes
2[1,2]00	14	f32	4	velocity controller proportional gain
2[1,2]00	15	f32	4	velocity controller summation gain
2[1,2]00	16	f32	4	velocity controller differential gain
2[1,2]00	17	f32	4	velocity controller maximal action value

Tab. D.1: The Object Dictionary

Index	Subindex	Type	Length [B]	Description
2[1,2]00	18	f32	4	position controller proportional gain
2[1,2]00	19	f32	4	position controller summation gain
2[1,2]00	20	f32	4	position controller differential gain
2[1,2]00	21	f32	4	position controller maximal action value

Tab. D.2: The Object Dictionary (Continued)

Value	Length [B]	Description
axis mode	1	LSB contains axis 1 mode - 0 means velocity mode, 1 means position mode, first bit of the second nimble contains axis 2 mode
axis enabled	1	LSB sets axis 1 enabled - 0 means disabled, 1 means enabled, second lowest bit sets axis 2 enabled

Tab. D.3: RxPDO1 mapping

Value	Length [B]	Description
battery voltage	2	battery voltage in millivolts
temperature	2	temperature in tenths of °C

Tab. D.4: TxPDO1 mapping

RxPDO2, TxPDO2

Value	Length [B]	Description
axis 1 velocity	4	32-bit float indicating axis 1 velocity in revolutions per second
axis 2 velocity	4	32-bit float indicating axis 2 velocity in revolutions per second

Tab. D.5: Mapping of PDOs containing velocity information - RxPDO2 and TxPDO2

Value	Length [B]	Description
axis revolutions	4	signed 32-bit integer denoting the number of axis shaft revolutions
axis angle	4	unsigned 32-bit indicating axis shaft angle

Tab. D.6: Mapping of PDOs containing position information - RxPDO3, RxPDO4, TxPDO3 and TxPDO4