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 AUTOR PRÁCE	Bc. Matouš Hýbl
SUPERVISOR VEDOUCÍ PRÁCE	prof. Ing. Luděk Žalud, Ph.D.

BRNO 2021

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

Abstrakt práce v originálním jazyce

KEYWORDS

Klíčová slova v originálním jazyce

ABSTRAKT

Překlad abstraktu (v angličtině, pokud je originálním jazykem čeština či slovenština; v češtině či slovenštině, pokud je originálním jazykem angličtina)

KLÍČOVÁ SLOVA

Překlad klíčových slov (v angličtině, pokud je originálním jazykem čeština či slovenština; v češtině či slovenštině, pokud je originálním jazykem angličtina)

EXTENDED ABSTRACT

Výtah ze směrnice rektora 72/2017:

Bakalářská a diplomová práce předložená v angličtině musí obsahovat rozšířený abstrakt v češtině nebo slovenštině (čl. 15). To se netýká studentů, kteří studují studijní program akreditovaný v angličtině. (čl. 3, par. 7)

Nebude-li vnitřní normou stanoveno jinak, doporučuje se rozšířený abstrakt o rozsahu přibližně 3 normostrany, který bude obsahovat úvod, popis řešení a shrnutí a zhodnocení výsledků. (čl. 15, par. 5)

HÝBL, Matouš. *Two channel stepper motor controller*. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Control and Instrumentation, 2021, 43 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

*The author signs only in the printed version.

ACKNOWLEDGEMENT

Ludek, Lukas, Kluci z RG, Ondra Bastan, James Munns, Hanno Braun, Kacenka, Rodina, Kocour Rád bych poděkoval vedoucímu diplomové práce panu Ing. XXX YYY, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Contents

List of Figures

List of Tables

Listings

Introduction & Motivation

In general, the majority of embedded systems nowadays are developed in the C/C++ programming languages [?]. There are some examples - there are systems developed in Ada and currently the embedded development in Python is starting to take off in hobby projects [?]. While C and C++ are suitable for 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 nowadays cause of many critical software problems, be it security vulnerabilities or safety hazards. Recent Chrome browser analysis and report shows that around 70 % of high severity problems are memory safety problems - meaning problems with pointers and staggering half of these are use-after-free problems [3]. 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 [4].

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 cause problems during development. For example when programs behave 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 effects as the above mentioned security vulnerabilities. Imagine a robot uncontrollably spinning and destroying its surroundings because some part of a program overwritten its controls by mistake.

We believe that both of these problems can be solved by using the Rust programming language. While being a relatively novel language for systems development (development started in 2008), 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 performance of systems programming allows the developers to create powerful and performance, yet in many cases zero-cost (memory or performance) abstractions. This is especially useful as the complexity of all systems is rising and we believe that to deliver great systems, the human programmers need to be aided by all available tools. Even though the language primarily targets high 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 development approaches. 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 a dual channel stepper motor controller and develop its firmware and control application in Rust, showcasing the language's advantages and disadvantages. This project's aim was also to push forward the development of electronics devices - using high-performance MCUs (Microcontroller Unit), state of the art stepper drivers and effective 4-layer PCB design and contemporary manufacturing capabilities.

[5]

The motor controller is designed to be used 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 for students' projects and development of our robots. The current state of the stepper motor controllers in the research group is summed up in the following chapter 1.

1 Related Work

This chapter describes the current state of the stepper motor controllers in the Robotics group and the efforts to improve it. Ongoing efforts to use program embedded systems in the Rust programming language is 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

In the Robotics and AI research group, currently a second generation of the KM2 stepper motor controller is widely in use. A render of the KM2 controller can be seen in the figure 1.1. It is used primarily by the students of the BPC-PRP course for driving a simple differentially driven robot. 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 RAMPS boards. The motor controller is controlled using the I^{2}C bus. There are two major shortcomings of the driver - the used controller's I^{2}C 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, that are quite loud and do not support contemporary advanced features. The boards for 3D printers are also reportedly not well designed and they may overheat [**?**].

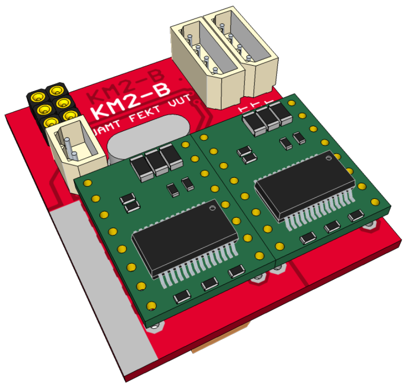


Fig. 1.1: KM2 motor controller render [1].

1.2 KM3

The KM3 (or KM2-C) was supposed to be a successor to the previously described KM2 controllers and it aimed to mainly 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 (Printed Circuit Board). The controller can be seen in the figure 1.2. Even though the new MCU was an improvement over the ATMega8, it proved to be the bottleneck for implementing new functionality to the motor controller as the MCU has very limited memory, both FLASH and RAM and peripherals. For example, the lack of pins made it impossible to directly generate pulses to control the STEP/DIR interface of the motor drivers chip and the control had therefore be done manually in software. Also the fact, that the MCU utilizes a Cortex-M0 core means that the support for atomic instructions mis 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 [6] for this board and concluded that the board and its design may be suitable for the students' robot projects, but it is way too limited to be used in more serious projects. We also concluded that the hardware used is now overcome and the development of this board shall not be further pursued as the technology it's been designed upon as well as its goals have been considered obsolete.

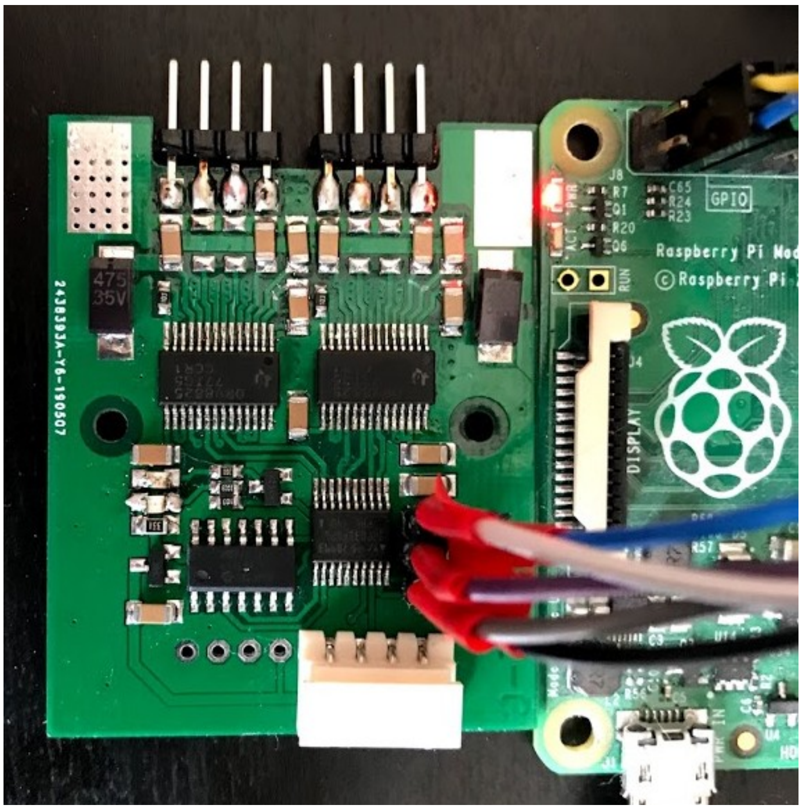


Fig. 1.2: The KM3 motor controller connected to a Raspberry Pi [1].

1.3 DC Motor

The DC Motor is a custom-developed DC (Direct Current) motor controller developed in the Robotics & AI research group. It was designed to feedback control DC motors, primarily DC motors manufactured by Maxon and therefore provide a cheaper alternative to the Maxon EPOS motor controllers. The driver alongside with a connected Maxon DC motor can be seen in the figure 1.3. Originally, the firmware for the motors, developed by Frantisek Burian, Ph.D., implemented current control and velocity control. However, the firmware exhibited unwanted behavior, such as extreme motor temperature rises and unwanted high-pich noise. After consultation with Lukas Kopecny Ph.D., we decided to rewrite the firmware in Rust and get rid of the current controller, with the reasoning that current control of such low inductance motor makes not much sense and instead replaced it with some form of current limiting and failsafe overcurrent stops. The new firmware alongside with some hardware modifications was successfully deployed to 7 DC Motor drivers as a 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 for the development of the firmware. We believe that using the language was the right choice and not only made the firmware simpler to use, but also made it possible to develop it in such short small time. It can be said that the work on the firmware for this board laid the foundation for the work on this thesis.

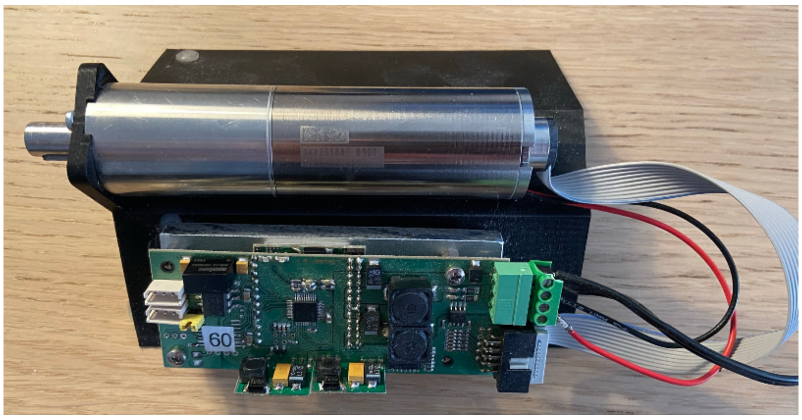


Fig. 1.3: The DC Motor driver with a connected Maxon DC motor.

1.4 Mechaduino

Mechaduino is a project that aims to create a 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 [2]. 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 integration of the whole system de-facto into the motor, removing any need for a controller board. On the other hand, the controller doesn't leverage any existing stepper motor controller solution and instead implements the servo 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 the 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 there was a semestral thesis and preliminary market research preceeding this project, the Mechaduino would provide valuable information to improve the design of our project.

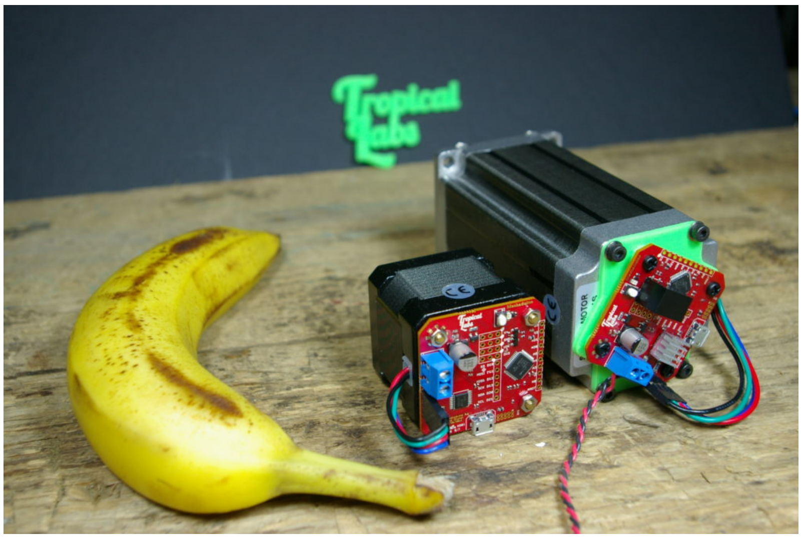


Fig. 1.4: The Mechaduino controller boards mounted on the back of a stepper motors [2].

1.5 Flott

Flott is a set of libraries suitable for developing motion controllers in the Rust programming language[7]. It is a relatively new project, but as of now, it contains an abstraction layer for stepper motors and acceleration ramp generator. The project is taking a different approach to controlling stepper motors as we are. It aims to utilize software pulse generation instead of timers and uses variable step period in ramp generation. Even though this is a valid approach we chose to not 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 be capable to fully support various features of the controller even in the future.
- The project shall use the state-of-art motor controller ICs.

The DC Motor project showed us, that writing a fully functional embedded firmware is not only a possible but a very viable option.

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

Flott shows us that there are more people trying to achieve building a motion controllers in Rust and that we can get inspired from and share knowledge with. 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.

Firstly, the requirements for the resulting hardware and software are set. These requirements comprise of functional requirements, non-functional requirements and constraints.

Secondly, an introduction is given about the problematics of stepper motors and their control and about communication buses utilized by the project.

Thirdly, the development of two hardware revisions is described. Hardware design choices are described and those are based on the requirements and related work. Hardware design using KiCAD is outlined and preparing source data for subsequent manufacturing is described.

Furthermore, the development of the software for the stepper motor controller is illustrated. A brief introduction to the Rust programming language is given, alongside with a more in-depth introduction to the current state of using Rust programming language for embedded systems. Further, development of the firmware itself is outlined, with some interesting parts being described in detail. Finally, the development of a control software usable for controlling the stepper motor controller is gone over.

2.1 Requirements

In Better Embedded System Software[8] 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 important 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.

2.1.1 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 requirement 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 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 interface allowing to connect external encoders (absolute or incremental) during future development.

2.1.2 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.
- **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.
- **NFR-06** The firmware should utilize hardware in the loop integration testing for QA.
- **NFR-06** The firmware shall be properly documented.
- **NFR-08** The functionality of the controller shall be demonstrated using two distinctive applications.

2.1.3 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 Stealth Chop.
- **C-02** The controller shall be able to drive motors with current of up to 2 A.
- **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.

2.2 Stepper motors

2.3 Communication protocols

2.3.1 CANOpen

2.3.2 I2C

2.3.3 USB

The USB communication with the controller is implemented using the virtual COM port protocol. Data frames have the following format:

2.4 Hardware development

This section describes the development of the hardware of the stepper motor controller. Firstly, the critical components are selected and preliminary design is done based on the requirements stated in the Section 2.1. Secondly, both of the hardware revisions and their design are described.

2.4.1 Hardware Design Choices

In this section, we describe the choices made in 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 2.1, the related work described in the Chapter 1 and our prior experience.

MCU

One of the critical components 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 possible future encoder support it should require incremental encoder interfaces and SPIs for SSI bitbanging (as stated in requirement **FR-08**). As was described in the Chapter on Related work 1, we decided to make the 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 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[17]. The block diagram of the MCU with the core features and peripherals can be seen in the Figure 2.1.

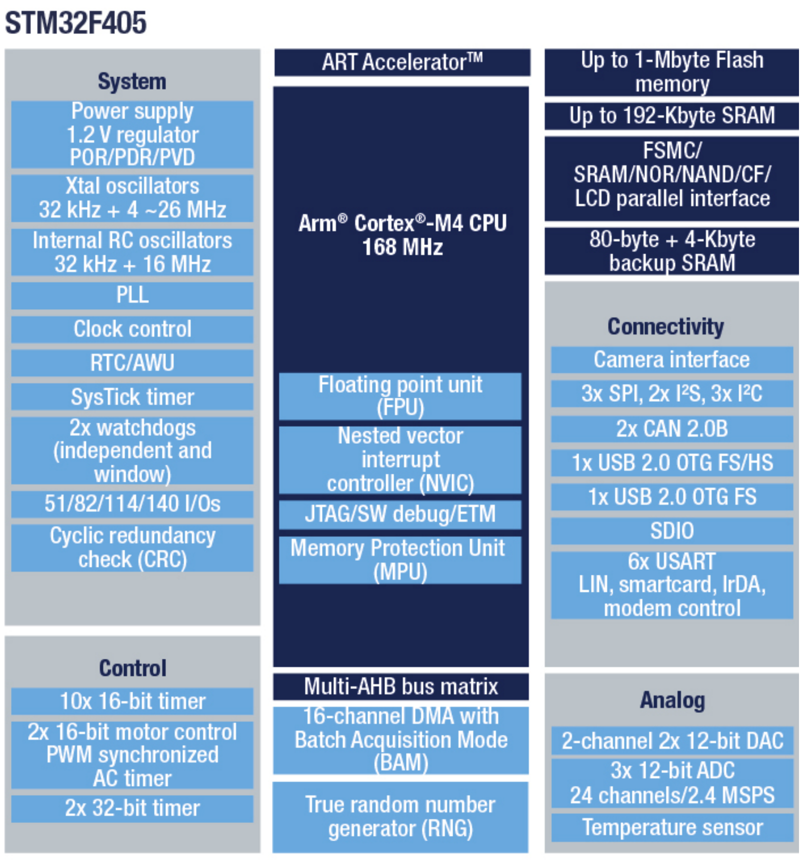


Fig. 2.1: The block diagram of the STM32F405RG MCU [17].

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

Stepper driver

As per the requirements **C-01** and **C-02**, the stepper driver shall be able to drive stepper motors with current of up to 2A and feature silent operation. We decided to use the stepper motor driver ICs developed by Trinamic. These drivers are nowadays being used for motion control of 3D printers [**?**] and allow for silent operation with their StealthChop technology. When selecting the driver ICs there were different priorities in mind - first the package should be solderable by hand, secondly with the first revision of the SM4 stepper motor controller we aimed for the ability to power the drivers via a 10-cell Li-Ion power pack, which required maximal input voltage of more than 42 Volts. The first hardware revision features the TMC2100-TA driver IC, whose main features are shown in the Table 2.1.

Parameter	Value
Motor supply voltage	5-46 V
Microsteps	up to 256
Control interface	Step/Dir
Configuration interface	GPIO
Phase current (RMS)	1.4 A
Advanced stepper control technologies	MicroPlyer, SpreadCycle, StealthChop
Package	eTQFP48

Tab. 2.1: Main parameters of the Trinamic TMC2100-TA driver IC [18].

With the second hardware revision, the priorities have shifted and higher phase current was required (**C-01**) even at the expense of the supply voltage. The second hardware revision features the TMC2226-SA driver IC, whose properties are shown in the Table 2.2.

Parameter	Value
Motor supply voltage	4.75-29 V
Microsteps	up to 256
Control interface	Step/Dir or UART
Configuration interface	UART
Phase current (RMS)	2.0 A
Advanced stepper control technologies	MicroPlyer, CoolStep, SpreadCycle, StealthChop2, StallGuard4
Package	HTSSOP28

Tab. 2.2: Main parameters of the Trinamic TMC2226-SA driver IC [**?**].

SM4 stepper driver 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 use 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.

As for the power electronics, only input voltage filtering using capacitors was utilized. The main reasoning being that this should be fused on the side of the power source 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 2.2.

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

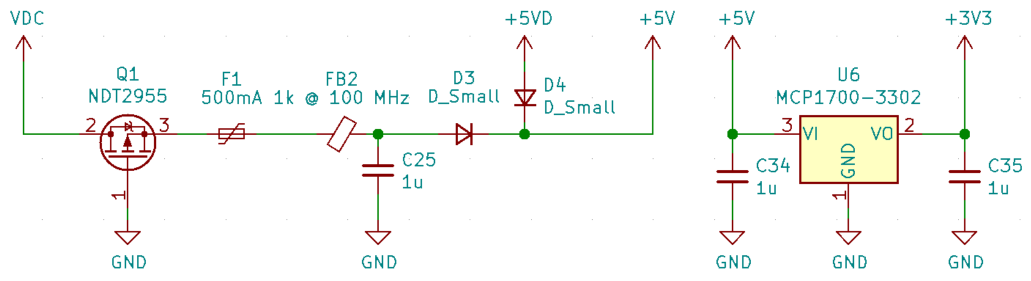


Fig. 2.2: The 5 V power rail for powering the MCU and peripherals.

PCB Design

In order for this project to serve as a testbed for new manufacturing technologies, the PCB (Printed Circuit Board) was designed as 4-layer. 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 stackup 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 also for some minor signal connections in cases routing using the outer layers would prove difficult. The used stackup can be seen in the Figure 2.3.

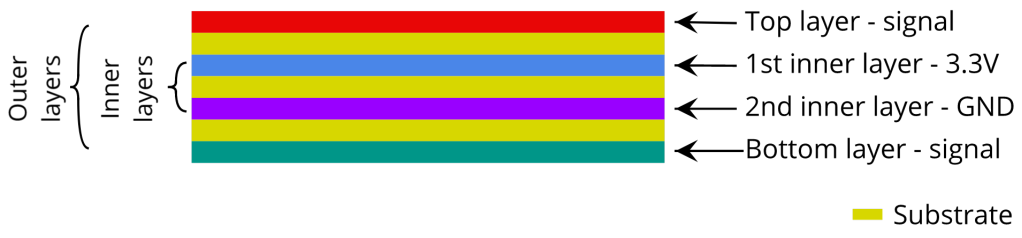


Fig. 2.3: The 4-layer PCB stackup.

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 great competitor 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.

2.4.2 Schematics and PCB Design using KiCAD

2.4.3 PCB manufacturing using JLCPCB and KiCAD

2.5 Development of the bare metal firmware

This section describes the development of the bare-metal firmware. A brief introduction to the Rust programming language is given and the usage of the language for embedded systems is described in detail. Further, some interesting parts of the firmware itself is described, alongside with the project structure, testing, etc.

2.5.1 Rust programming language

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

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 provides 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 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 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 in order to provide some introduction into the semantics and syntax of the language.

Variables and Mutability

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

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

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.

Ownership and Borrow Checker

The languages semi-automatic memory management system comprises of the ownership concept, move-by-default semantics and the borrow checker.

The concept of ownership is described by these rules[**?**]:

- Each value in Rust has a variable that's called its *owner*.
- There cane 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 move is almost zero-cost, copy almost never is.

The borrow checker is a mechanism 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 an 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 to values in structs.

Enums and Pattern Matching

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

```
enum Value {
```

```rust
    Integer(i64),
    Float(f64)
}

let int_value = Value::Integer(15);
let float_value = Value::Float(3.14);

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

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 for example state machines and error handling. To access the associated value, the **match** or **if** keywords may be used as can be seen in the following snippet:

```rust
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);
}
```

Objects and Methods

Traits and Generics

Macros

Standard Library

Unit Testing

2.5.2 Embedded Rust

2.5.3 Persistent storage using EEPROM emulation

The SM4 stepper motor controller needs persistent storage to save configuration and data. 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 the storage. In general, FRAM (Ferroelectric Random Access Memory), EEPROM (Electrically Erasable programmable read-only memory), or flash memories are used.

In order 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 the flash memory may seem straightforward to use since they are ubiquitous, their low level use is not that simple. A 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 [?]. That means that to write a higher number to the word of the memory, the flash needs to be first erased and then programmed. 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 whole sector erases, caused by the limitation of hardware.

Fortunately, this problem can be solved by emulating the EEPROM memory as described in ST Application Note AN 3969 [?]. 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 2.4.

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 a data with 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 and 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.

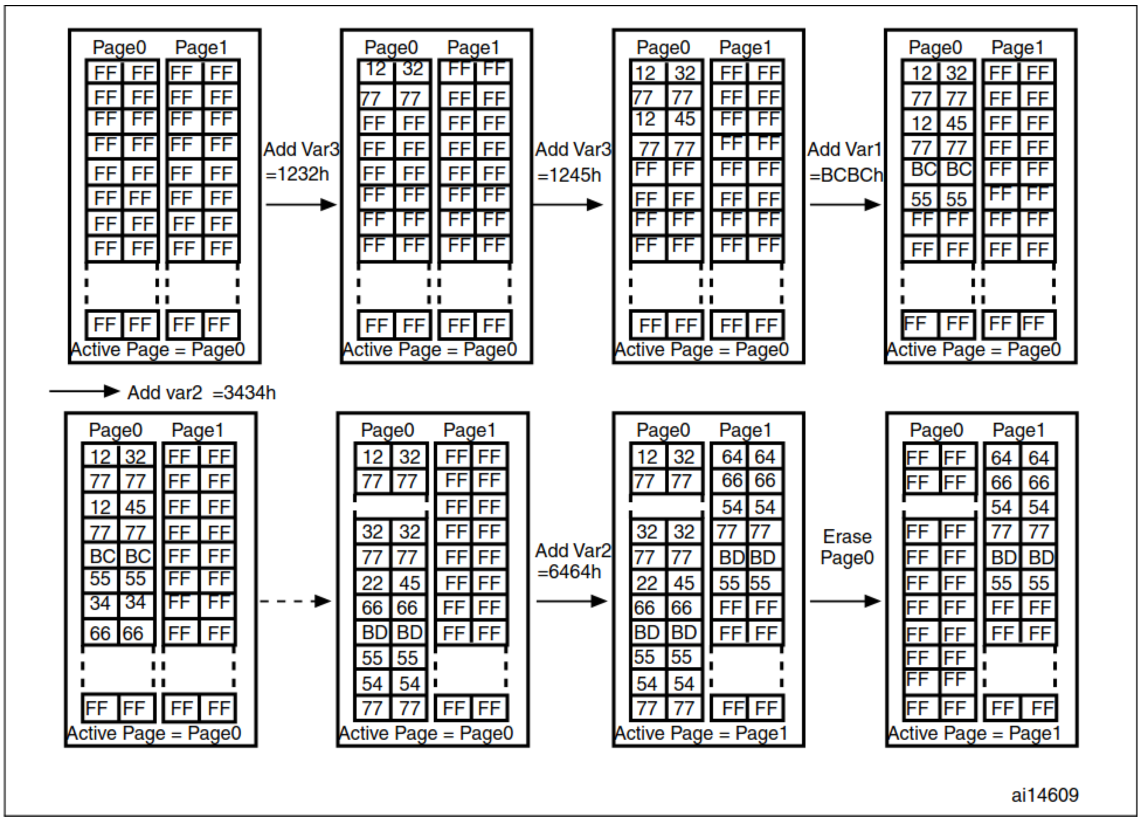
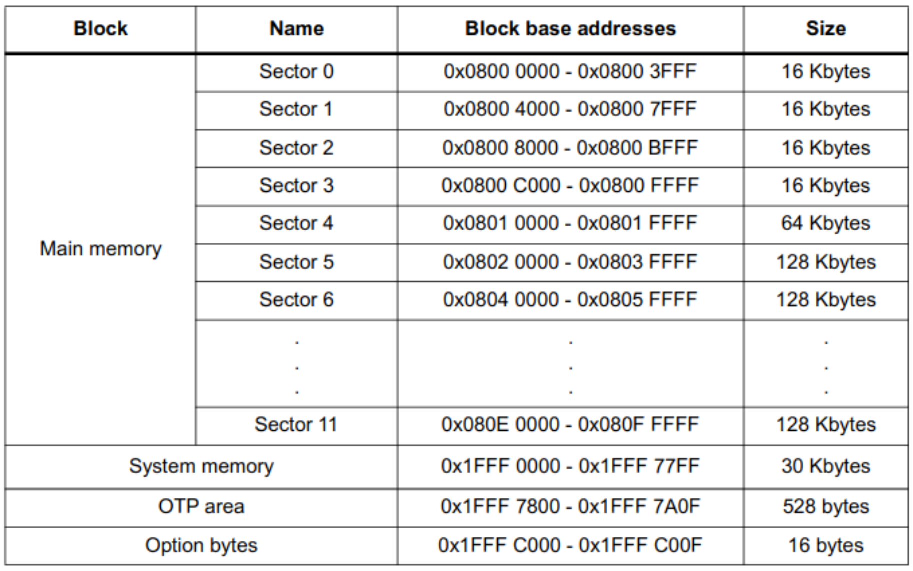


Fig. 2.4: EEPROM emulation working principle [?].

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 [?] there are a some 16 kilobyte sectors that could be used for the emulation, as can be seen in the Figure 2.5. 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. 2.5: EEPROM emulation working principle [?].

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 [?], 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 [?] 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 [?]. An example of accessing the emulated persistent storage can be seen in the following Listing 2.1.

```
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 2.1: An example use of the emulated persistent storage.

As can be seen in the Listing 2.1, 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.

2.6 Development of the control application

3 Results

3.1 Demonstration #1 - A simple mobile robot

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[8]

3.2 Demonstration #2 - A linear rail actuator for camera movement

Conclusion, Discussion and Future work

Bibliography

[1] KM2.render.png (PNG Image, 1024 × 735 pixels) — Scaled (92%). URL: `https://student.robotika.ceitec.vutbr.cz/PCBS/KM2/raw/master/bin/RevB/KM2.render.png`.

[2] Mechaduino – Tropical Labs. URL: `https://tropical-labs.com/mechaduino/`.

[3] Memory safety - The Chromium Projects. URL: `https://www.chromium.org/Home/chromium-security/memory-safety`.

[4] Heartbleed Bug. URL: `https://heartbleed.com/`.

[5] Program | Student Conference on Planning in Artificial Intelligence and Robotics. URL: `./index.html`.

[6] Robotics-BUT/KM3-rs, December 2020. original-date: 2020-10-18T19:51:28Z. URL: `https://github.com/Robotics-BUT/KM3-rs`.

[7] Flott - Motion Control in Rust. URL: `https://flott-motion.org/`.

[8] Phillip Koopman. *Better Embedded System Software*. Carnegie Mellon University, 1.st edition, revised 2021 edition, 2010. URL: `http://www.koopman.us/`.

[9] Akin's Laws of Spacecraft Design. URL: `https://spacecraft.ssl.umd.edu/akins_laws.html`.

[10] Robotics-BUT/DCMotor-firmware, February 2021. original-date: 2020-09-27T19:09:49Z. URL: `https://github.com/Robotics-BUT/DCMotor-firmware`.

[11] Modul KM2 - BPRP - Robotika a počítačové vidění. URL: `https://sites.google.com/a/vutbr.cz/bprp/kambot/dily/modul-km2`.

[12] 02 - Platforma KAMbot, GIT (dokončení) - BPRP - Robotika a počítačové vidění. URL: `https://sites.google.com/a/vutbr.cz/bprp/prednasky/2018/02`.

[13] PCBS / KM2. URL: `https://student.robotika.ceitec.vutbr.cz/PCBS/KM2`.

[14] eCorax - As above, so below: Bare metal Rust generics 1/2. URL: `https://www.ecorax.net/as-above-so-below-1/`.

[15] [RFC] user defined memory layout · Issue #164 · rust-embedded/cortex-m-rt. URL: `https://github.com/rust-embedded/cortex-m-rt/issues/164`.

[16] rust-embedded/cortex-m-rt. URL: `https://github.com/rust-embedded/cortex-m-rt`.

[17] STM32F405RG - High-performance foundation line, Arm Cortex-M4 core with DSP and FPU, 1 Mbyte of Flash memory, 168 MHz CPU, ART Accelerator - STMicroelectronics. URL: `https://www.st.com/en/microcontrollers-microprocessors/stm32f405rg.html`.

[18] TMC2100. URL: `https://www.trinamic.com/products/integrated-circuits/details/tmc2100/`.

[19] en.bd_stm32f405_1mb.jpg (JPEG Image, 700 × 754 pixels). URL: `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`.

[20] Implement flash read/erase/program by astro · Pull Request #239 · stm32-rs/stm32f4xx-hal. URL: `https://github.com/stm32-rs/stm32f4xx-hal/pull/239`.

Symbols and abbreviations

Šířka levého sloupce Seznamu symbolů a zkratek	je určena šířkou parametru prostředí `acronym` (viz řádek 1 výpisu zdrojáku na str. **??**)
KolikMista	pouze ukázka vyhrazeného místa
DSP	číslicové zpracování signálů – Digital Signal Processing
f_{vz}	vzorkovací kmitočet

List of appendices