Univerzita Karlova
Pedagogická fakulta
Katedra informačních technologií a technické výchovy

BAKALÁŘSKÁ PRÁCE

Programování robotické stavebnice v jazyce Rust

Programming a robotic kits in the Rust language

Petr Novák

Vedoucí práce: PhDr. Jakub Lapeš

Studijní program: Informační technologie se zaměřením na vzdělávání (B0114A140004)

2025

Odevzdáním této bakalářské práce na téma Programování robotické stavebnice v jazyce Rust potvrzuji, že jsem ji vypracoval pod vedením vedoucího práce samostatně za použití v práci uvedených pramenů a literatury. Prohlašuji, že jsem při její tvorbě nepoužil nástrojů umělé inteligence jiným způsobem, než je uvedeno ve vyjádření, které je součástí textu práce. Dále potvrzuji, že tato práce nebyla využita k získání jiného nebo stejného titulu.

Praha, 14.4.2025

Rád bych poděkoval vedoucímu práce PhDr. Jakubu Lapešovi za vstřícnost při konzultacích. Dále bych chtěl poděkovat přátelům ze zájmového kroužku, kteří mi pomohli při ověření úloh.

ABSTRAKT

Tato bakalářská práce se zaměřuje na programování robotické stavebnice LEGO® Mindstorms EV3 v programovacím jazyce Rust. Cílem práce je představit možnosti využití Rustu při řízení robotických systémů a demonstrovat jeho výhody na konkrétní platformě. Rust je moderní jazyk s důrazem na bezpečnost a efektivní správu paměti, díky čemuž je vhodný pro nasazení i na hardwarově omezených zařízeních, jakým je i řídicí jednotka stavebnice EV3.

Součástí práce je úvod do základů jazyka Rust a přípravy vývojového prostředí, včetně instalace potřebných nástrojů. Dále je podrobně popsáno nasazení alternativního operačního systému ev3dev na stavebnici LEGO® Mindstorms EV3, jeho propojení s vývojovým počítačem a konfigurace prostředí. V praktické části jsou uvedeny ukázkové programy, které čtenáři umožní pochopit ovládání jednotlivých prvků stavebnice, tedy motorů, senzorů i vstupně-výstupních periferií.

Závěrečná část práce nabízí návrhy úloh, které mohou sloužit jak k rozvoji technických dovedností, tak k výuce programování. Práce tak poskytuje komplexní přehled o možnostech využití jazyka Rust v oblasti vzdělávací robotiky a může sloužit jako výukový materiál pro pedagogy, vedoucí informaticky zaměřených zájmových kroužků či začínající programátory.

KLÍČOVÁ SLOVA

Programovací jazyk Rust, edukační robotika, LEGO Mindstorms EV3, ev3dev, výuka programování, střední škola

ABSTRACT
This bachelor thesis focuses on the programming of the LEGO® Mindstorms EV3 robotic kit in the Rust programming language. The aim of the thesis is to present the possibilities of using Rust in controlling robotic systems and to demonstrate its advantages on a specific platform. Rust is a modern language with an emphasis on safety and efficient memory management, which makes it suitable for deployment even on hardware-constrained devices such as the EV3 controller unit.

The thesis includes an introduction to the basics of the Rust language and the setup of the development environment, including the installation of the necessary tools. Furthermore, it describes the deployment of the alternative operating system ev3dev on the LEGO® Mindstorms EV3 kit, its connection to the development computer and the configuration of the environment. In the practical section, sample programs are provided to help the reader to understand the control of the various components of the kit, i.e. motors, sensors and input/output peripherals.

The final part of the work offers suggestions for tasks that can be used to develop engineering skills as well as to teach programming. Thus, the work provides a comprehensive overview of the possibilities of using the Rust language in the field of educational robotics and can serve as a teaching material for educators, computer science interest group leaders and novice programmers.

KEYWORDS
Rust programming language, educational robotics, LEGO Mindstorms EV3, ev3dev, educational programming, high school

Obsah

Úvod ... 9

1 Jazyk Rust ... 11

1.1 Historie jazyka Rust ... 11

1.2 Proč jazyk Rust ... 12

1.3 Porovnání s ostatními jazyky ... 12

1.4 Instalace prostředí ... 15

2 Práce s jazykem Rust ... 18

2.1 Vytvoření nového projektu ... 18

2.2 Syntaxe ... 20

2.3 Proměnné ... 22

2.3.1 Proměnlivost proměnných ... 23

2.3.2 Rozsah a stínování ... 24

2.3.3 Konstanty ... 25

2.4 Datové typy ... 26

2.5 Komentáře ... 28

2.6 Funkce ... 29

2.7 Řízení toku programu ... 31

2.7.1 Podmínky ... 31

2.7.2 Cykly ... 33

2.8 Struktury ... 34

2.9 Moduly ... 37

3 Linux pro LEGO Mindstorms EV3 ... 39

3.1 Porovnání s originálním firmwarem ... 39

3.2 Instalace ... 40

3.3 Nastavení ... 41
3.3.1 Základní příkazy pro ovládání systému ev3dev ... 43
3.3.2 Zvýšení taktu procesoru ... 44
4 Programování stavebnice a vzorové programy ... 45
4.1 Nastavení projektu pro ev3dev ... 45
4.2 Vstupně-výstupní prvky kostky ... 47
4.2.1 LED signalizace ... 48
4.2.2 Zvukový výstup ... 49
4.2.3 Tlačítka jako vstup ... 50
4.2.4 Práce s obrazovkou ... 51
4.3 Ovládání motorů ... 52
4.4 Získávání dat ze senzorů ... 54
5 Úlohy a možné řešení ... 59
5.1 Motorka ... 59
5.1.1 Rozvoj kompetencí ... 60
5.2 Sledování černé čáry ... 61
5.2.1 Rozvoj kompetencí ... 63
5.3 Navigace pomocí barevných značek ... 63
5.4 Autonomní vozidlo ... 65
5.4.1 Rozvíjení kompetencí ... 66
5.5 Orientace v bludišti ... 66
5.5.1 Rozvoj kompetencí ... 67
5.6 Zpětnovazební regulátory ... 68
5.6.1 Rozvoj kompetencí ... 69
5.7 Dvoukolové balancující vozítko ... 70

5.7.1 Rozvoj kompetencí ... 70

6 Závěr ... 71

Seznam použitých informačních zdrojů ... 72

Vyjádření k využití nástrojů umělé inteligence ... 79

Úvod

K programování a robotice jsem se dostal brzy, a to především díky robotické stavebnici LEGO® Mindstorms, se kterou jsem se seznámil již ve třinácti letech na kroužku robotiky. Možnost sestavení vlastního vozítka a naučit ho k určitému chování mě téměř okamžitě nadchla. Postupem času jsem se na základní a následně i na střední škole účastnil robotických soutěžích, kde jsem získal praktické zkušenosti a motivaci se dále rozvíjet.

Velkou roli v tom měli kamarádi a spolužáci, kteří byli velmi ochotni mi pomáhat. Je možné, že bez jejich podpory by můj zájem o informační technologie nevydržel. Právě z těchto důvodů jsem si zvolil toto téma bakalářské práce, které může inspirovat další studenty k získání zájmu o svět informačních technologií.

V práci se budu věnovat programování robotické stavebnice LEGO® Mindstorms EV3, konkrétně verzi Education Core Set (sada 45544)[1)], která je určena pro školní prostředí. Na řídicí jednotku stavebnice bude nahrán alternativní operační systém, který odemkne nové způsoby využití stavebnice. K programování bude využit moderní jazyk Rust, který získává popularitu díky jeho rychlosti a bezpečnost práce s pamětí.

Pro zlepšení přehlednosti a srozumitelnosti budou v práci rozlišeny různé typy textu pomocí specifických stylů písma. Příkazy, odkazy na části příkazů a zdrojového kódu jsou označeny `tímto stylem, který používá písmo s pevnou šířkou, které je podbarveno šedou barvou.`

```
Pro odlišení samotného zdrojového kódu nebo obsahu souboru bude použit tento styl, který taktéž používá písmo s pevnou šířkou. Tento styl je navíc ohraničen a podložen jemnou texturou.
```

Cíl práce

Cílem této práce je zmapovat možnosti a využitelnost programovacího jazyka Rust při řízení robotických systémů LEGO® Mindstorms EV3, a to zejména v kontextu středního odborného vzdělávání v souladu s rámcovými vzdělávacími programy (RVP SOV). Vedle školního prostředí je práce zaměřena také na využití ve volnočasových aktivitách, jako jsou kroužky robotiky. Na základě praktického ověření budou navrženy robotické úlohy

podporující rozvoj algoritmického myšlení, které mohou sloužit jako výukový materiál pro pedagogy i studenty.

1 Jazyk Rust

1.1 Historie jazyka Rust

Rust je relativně nový programovací jazyk, jehož prvotní vývoj započal v roce 2006 Graydon Hoare[2], který jej zprvu vyvíjel ve svém volném čase jako osobní projekt. Jeho cílem bylo vytvořit programovací jazyk, který by dosáhl podobného výkonu jako běžně používané jazyky jako C a C++, ale zároveň by poskytl výrazně vyšší bezpečnost práce s pamětí a usnadnil psaní souběžně (paralelně) běžícího kódu.

O tři roky později se k vývoji projektu připojila společnost *Mozilla*[2], která v Rustu viděla potenciál pro vytvoření bezpečného internetového prohlížeče. O rok později byl jazyk veřejně oznámen a v roce 2012 společnost představila první veřejně dostupnou verzi jazyka Rust[4]. Zároveň společnost začala vyvíjet experimentální webový prohlížeč *SERVO*[5] napsaný v Rustu, který měl demonstrovat výhody a funkce jazyka Rust, čímž přispěl k popularizaci Rustu a vedl k jeho dalšímu vývoji.

První stabilní verze Rustu byla vydána v roce 2015 jako Rust 1.0[6]. Tímto vydáním bylo oznámeno, že Rust je připraven pro produkční nasazení. Od té doby se Rust neustále vyvíjí a počet jeho uživatelů přibývá. V roce 2021 vznikla nezávislá organizace *Rust Foundation*[7], která byla založena společnostmi *Amazon*, *Google*, *Huawei*, *Microsoft* a *Mozilla*. Tato organizace spravuje vývoj Rustu, což podpořilo nejen jeho stabilitu a rostoucí popularitu, ale i zajistila budoucnost tohoto jazyka.

V průběhu let si jazyk získal oblibu nejen mezi vývojáři systémového softwaru, ale i v oblastech webového vývoje, vestavěných systémů a v herním průmyslu[8]. Rust našel uplatnění v mnoha technologických firmách[9], mezi které patří například *Amazon*, *Google*, *Meta*, *Microsoft* a *Discord*, kde je využíván například pro vývoj *backendového* softwaru nebo systémových nástrojů.

Díky svým unikátním vlastnostem a aktivní komunitě se tedy postupně stal populárním nástrojem pro psaní výkonného a bezpečného softwaru, přičemž jeho další vývoj a vylepšení pokračují díky podpoře open-source komunity a významných technologických firem.

Tento jazyk se stal natolik oblíbeným, že se stal prvním jazykem, kromě jazyka C, který byl použit pro vývoj Linuxového kernelu. Tento čin byl zpočátku kontroverzní[10], protože se

vyskytly obavy, že přidání jazyka zkomplikuje vývoj a zhoršuje udržitelnost kódu. Nakonec se ale přidání jazyka do kernelu osvědčilo.

1.2 Proč jazyk Rust

Rust je moderní programovací jazyk, který kombinuje vysoký výkon s bezpečností a spolehlivostí. Na rozdíl od starších jazyků, jako jsou například C a C++, se snaží eliminovat časté chyby při práci s pamětí, které mohou vést k pádům programů nebo bezpečnostním problémům[11)].

Jednou z hlavních předností jazyka Rust je jeho bezpečnost při práci s pamětí. To znamená, že jazyk nedovolí programátorovi udělat chyby, které by mohly poškodit běh programu. Nejčastěji se jedná o přístup k paměti, která už neexistuje, čtení paměti, která nepatří programu nebo souběžný zápis a čtení souboru. Rust tyto problémy zachytí už při překladu programu, takže se chyby nezjistí až po jeho spuštění.

Rust je kompilovaný jazyk, což znamená, že se před spuštěním přeloží do strojového kódu. Díky tomu je běh programu velmi rychlý a dokáže běžet i na méně výkonných zařízení, jako je právě řídicí kostka stavebnice LEGO Mindstorms EV3. Programy napsané v Rustu se tak mohou používat na různých zařízeních, od běžných počítačů až po vestavěné systémy.

Další výhodou Rustu je jeho podpora paralelního neboli souběžného programování. Díky přísné kontrole při překladu může jeden program bezpečně běžet ve více vláknech. To je důležité u složitých programů řídících například roboty, kteří potřebují současně řídit pohyb, vyhodnocovat data ze senzorů a komunikovat s uživatelem.

Rust také nabízí širokou podporu nástrojů a knihoven. Například správce balíčků s názvem *Cargo*[11)] umožňuje jednoduše přidávat další části programu (tzv. knihovny), spravovat závislosti a sestavovat programy. Díky těmto nástrojům je práce s Rustem přehledná i pro začátečníky.

1.3 Porovnání s ostatními jazyky

Rust je často srovnáván s dalšími programovacími jazyky, a to jak z hlediska výkonu, tak i bezpečnosti, přehlednosti kódu nebo vhodností pro výuku a začátečníky. V této kapitole

bude porovnán Rust s jazyky *Python* a *C*, které patří mezi doporučované jazyky pro začátečníky[12][13].

Python je široce rozšířený skriptovací jazyk, který je oblíbený především díky své jednoduché syntaxi a čitelnosti. Díky tomu je často využíván ve školství[14] jako úvodní jazyk do programování. Na druhou stranu je potřeba počítat s tím, že Python je *interpretační jazyk*, což znamená, že není překládán do strojového kódu před spuštěním. Výsledkem je výrazně nižší výkon v porovnání s kompilovanými jazyky, jako je právě Rust nebo C.

C je naopak tradiční kompilovaný jazyk, který nabízí vysoký výkon a detailní přístup k hardwaru. Jeho hlavní nevýhodou je však nízká úroveň bezpečnosti při práci s pamětí, což může vést k obtížně detekovatelným chybám. Rust se snaží být alternativou k C. Zachovává si vysoký výkon, ale zároveň přidává silné bezpečnostní mechanismy.

Pro porovnání rychlosti byly vytvořeny jednoduché testovací programy ve třech jazycích – Rust, Python a C. Tyto programy prováděly následující výpočty:

- Výpočet n-tého Fibonacciho čísla pomocí iterativního algoritmu[15]
- Eratosthenovo síto pro nalezení prvočísel menších než zadané číslo[16]

Každý program byl spuštěn s pěti různými vstupními hodnotami, které se postupně zdvojnásobovaly. Program při každém běhu změřil a vypsal čas potřebný k výpočtu. Programy nebyly optimalizovány, jelikož cílem bylo získat realistické výsledky odpovídající výchozímu nastavení.

Naměřené časy při výpočtů jsou uvedeny v následujících dvou tabulkách. Všechny časy jsou uvedeny v milisekundách. Výpočty byly provedeny na procesoru AMD Ryzen 5 3400G[17].

Vstupní hodnota	Rust	Python	C
20000	0,267	6,41	0,077
40000	0,524	18,16	0.181
80000	1,042	65.38	0,307
160000	1,904	247,2	0,617
320000	2,861	973,8	1,199

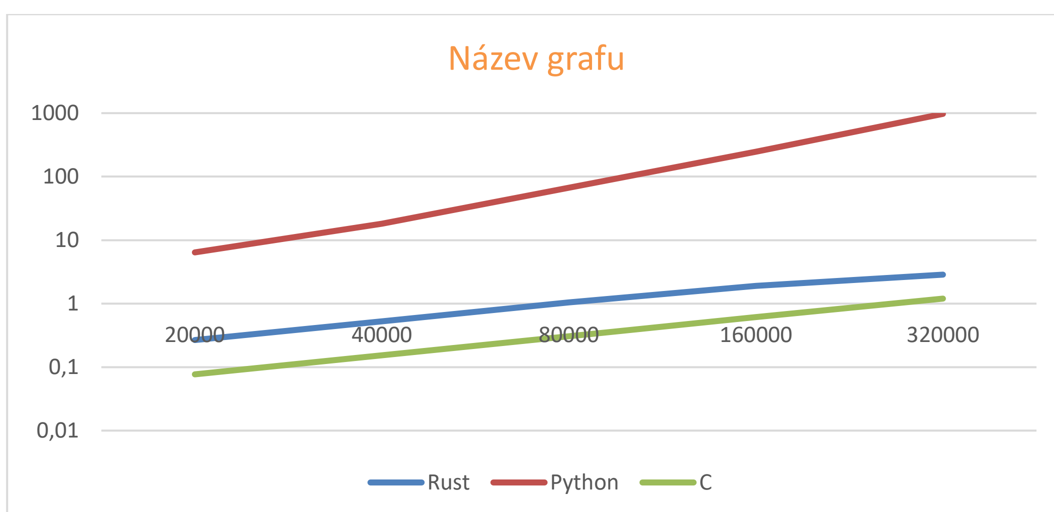
Tabulka 1: Časy výpočtu Fibonacciho čísel

Vstupní hodnota	Rust	Python	C
20000	1,081	1,109	0,235
40000	2,281	2,165	0.456
80000	4,545	4,470	0,976
160000	9,361	9,219	2,106
320000	19,18	18,99	3,649

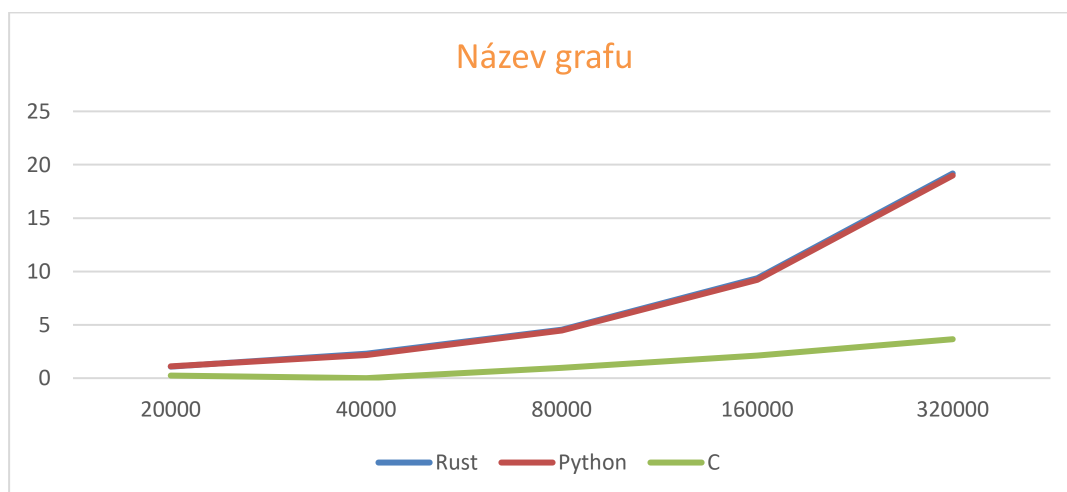
Tabulka 2: Časy nalezení prvočísel za použití Eratosthenova síta

Z výsledků si lze povšimnout, že jazyk C má nejrychlejší časy výpočtu u všech testovaných úloh. Rust se těmto časům ve většině případů přibližuje. Python je v případě Fibonacciho posloupnosti výrazně pomalejší, ale je stále použitelný pro jednodušší úlohy a výuku algoritmů.

V případě hledání prvočísel jsou časy výpočtů Rustu a Pythonu velmi podobné. Použité instrukce algoritmu jazyka Rust zřejmě nebyly vhodně zvoleny.



Graf 1: Zobrazení časové náročnosti výpočtu Fibonacciho čísla, logaritmická osa Y



Graf 2: Zobrazení časové náročnosti nalezení prvočísla, logaritmická osa Y

1.4 Instalace prostředí

Pro vývoj programů v jazyce Rust je zapotřebí nainstalovat základní komponenty – Rust samotný a vhodný editor zdrojového kódu a případně dodatečné rozšíření, které zvýší komfort během psaní kódu. V tomto oddílu je popsán postupy instalace pro operační systém Windows. Postup instalace na operační systémy typu Linux a MacOS bude podobný.

Vzhledem k zaměření práce na výuku a dostupnost nástrojů bude využit především editor *Visual Studio Code* a oficiální správce Rustu, nástroj rustup.

Nejjednodušší způsob, jak nainstalovat Rust je pomocí oficiálního instalačního souboru:

1. Otevřete webovou stránku Rustu a klikněte na tlačítko „Get Started“.
2. Stáhněte soubor „rustup-init.exe“ pro vaši architekturu počítače (pravděpodobně 64bit).
3. Po spuštění staženého souboru se zobrazí příkazový řádek. Pro zahájení instalace napište „1“ a stisknout klávesu *Enter.*
    a. Tato možnost nainstaluje potřebné knihovny, jako například „*Visual Studio C++ Build tools*“. Pokud instalační program pozná, že v počítači jsou již instalované, můžete pokračovat na krok 7.
4. Instalátor nyní začne stahovat a instalovat potřebné programy a soubory. Pro započetí instalace stačí souhlasit s podmínkami a nechat vybrané výchozí možnosti.
5. Během stahování souborů a instalace odškrtněte možnost „Spustit po instalaci“
6. Po dokončení instalace zavřete okno a vraťte se k původnímu instalátoru.
7. V původním instalačním okně, po dokončení instalace potřebných závislostí, stiskněte klávesu *Enter* pro instalaci samotného Rustu.
8. Po dokončení opět stiskněte klávesu *Enter* pro zavření instalačního programu.

Po dokončení instalace je dobré si ověřit, zda instalace opravdu proběhla úspěšně. Pro ověření bude zapotřebí příkazový řádek (program cmd.exe). Pomocí kombinací kláves *Win + R* se otevře dialogové okno. Pro spuštění příkazového řádku do textového pole napište „cmd“ a stiskněte *Enter*.

Pro zobrazení verze Rustu zadejte příkaz `rustc --version`. Jako výstup se mohou zobrazit následující možnosti:

A. Výstup je podobný jako `rustc 1.85.0 (4d91de4e4 2025-02-17)`
    - Instalace proběhla úspěšně a můžete pokračovat dál
B. Výstup říká, že „rustc“ nebyl rozpoznám jako příkaz nebo program
    - Instalace neproběhla úspěšně. Pro pokračování musíte problém vyřešit, aby byla instalace úspěšná.

Nyní je čas na instalaci editoru *Visual Studio Code*, také známý jako *VS Code*. Postup instalace je následující:

1. Otevřete webovou stránku softwaru Visual Studio Code[18)] a stáhněte verzi pro Windows
2. Po spuštění instalátoru vyberte Anglický jazyk a souhlaste se smluvními podmínky programu. Ostatní možnosti můžete ponechat ve výchozím nastavení, a klikněte na tlačítko „Install“.
3. Po dokončení klikněte na tlačítko „Finish“. Otevře se okno programu.

Teď je potřeba nainstalovat rozšíření pro jazyk Rust, které bude například zvýrazňovat syntaxi pro lepší orientaci ve zdrojovém kódu[19)].

1. Pomocí kombinací kláves „*Ctrl + Shift + X*“ otevřete postranní panel s rozšířeními.
2. Pomocí vyhledávacího pole vyhledejte balíček „rust-analyzer“ a nainstalujte jej.

Po úspěšné instalaci Rustu a editoru má čtenář k dispozici plně funkční vývojové prostředí. V následující kapitole bude popsán způsob vytvoření a základní správy Rust projektů pomocí nástroje Cargo.

2 Práce s jazykem Rust

Po instalaci všech potřebných prvků pro programování v Rustu je možné začít se samotným programováním. Tato kapitola se zaměřuje na základy práce s jazykem Rust, jako je správa projektů, psaní zdrojového kódu, kompilace a spouštění programů. Následují podkapitoly se dále věnují klíčovým aspektům, jako jsou datové typy a funkce. Tyto koncepty hrají zásadní roli pro psaní zdrojového kódu a porozumění způsobu, jak Rust pracuje.

2.1 Vytvoření nového projektu

Rust nabízí několik nástrojů, které pomáhají při vývoji aplikací, přičemž hlavní z nich je *Cargo*. Tento nástroj usnadňuje vytváření nových projektů, správu závislostí a sestavování programů.

Práce s Rustem začíná vytvořením nového adresáře, ve kterém se budou nacházet projektové soubory. Nejlepší postup je vytvořit pracovní adresář, ve kterém bude každý projekt uložený zvlášť. Tento adresář lze zvolit podle osobních preferencí, například v adresáři `Dokumenty/Rust`.

Nejprve je nutné otevřít program *Visual Studio Code*, na který budu referovat jako *editor*, a vybrat pracovní složku. To lze provést například tímto postupem:

1. V horním menu zvolit **File →Open Folder…**, případně **Soubor →Otevřít složku…**
2. Pomocí dialogového okna vybrat požadovanou složku
3. Po potvrzení se složka zobrazí v postranním panelu editoru

Jakmile je složka v editoru otevřená, lze v ní vytvořit nový Rust projekt z příkazového řádku. Ten lze v editoru otevřít z horního menu, a to Terminál → Nový terminál. V terminálu je nyní možné vytvořit nový projekt příkazem `cargo new <název_projektu>`. Pro vytvoření projektu „ahoj_svete" tedy zadám příkaz `cargo new ahoj_svete`.

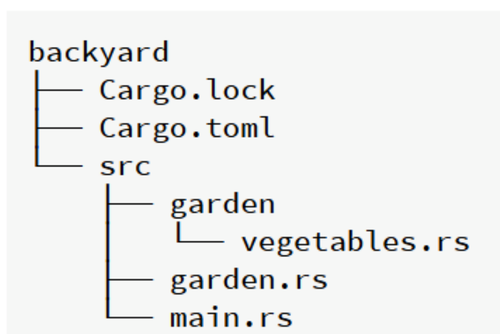
Rust má pravidla pro pojmenování, která pomáhají udržet zdrojový kód čitelný a konzistentní. Pro názvy proměnných a funkcí se používá formát zvaný *snake_case*[20)] (lze přeložit jako „hadí písmo"). Každé slovo je psáno malými písmeny a jednotlivá slova jsou oddělená podtržítkem. Například pokud se pokusím vytvořit nový projekt v jiném formátu,

Cargo napíše varování, že jméno není ve formátu *snake_case* nebo *kebab-case*[21)] (podobný formátu *snake_case*, ale slova jsou oddělená pomlčkou).

Doporučuji pojmenovávat vše bez použití diakritiky. Tyto znaky jsou specifické pro český jazyk a na cizojazyčných klávesnicích je nelze zapsat. To je problematické zejména v situacích, kdy na vývoji zdrojového kódu pracuje více lidí. Navíc se může stát, že se projekt uloží v jiném formátování nebo jazykové sadě (například ASCII), ve které diakritické znaky neexistují.

Po zadání příkazu se v pracovním adresáři vytvoří nová složka s názvem projektu, která obsahuje základní strukturu projektu, která se nyní zobrazuje v postranním panelu. V této složce se nachází automaticky vygenerované soubory, přičemž nejpodstatnější jsou:

- `Cargo.toml` – konfigurační soubor, ve kterém jsou informace o projektu a jeho závislostech
- `src/main.rs` – hlavní soubor, ve kterém se nachází výchozí zdrojový kód



Obrázek 1: Struktura projektu[22)]

Tento jednoduchý program slouží jako test funkčnosti prostředí a zároveň ukazuje základní strukturu programu.

Tímto krátkým odstavcem chci čtenáři doporučit, aby následující programy použil a experimentoval s nimi. Pouze čtením textu není možné se naučit programovat a nejlepší formou učení je programování v praxi.

2.2 Syntaxe

Po vytvoření nového projektu se ve výchozím souboru `main.rs` nachází jednoduchý testovací program, kterému se říká „Hello World!“. Tento program slouží k ověření správnosti prostředí a zároveň ukazuje základní syntaxi jazyka.

Zdrojový kód vypadá následovně:

```
fn main() {
    println!("Hello, world!");
}
```

Tento kód představuje nejjednodušší program v Rustu, ale nejdřív popíšu jeho strukturu.

Každý spustitelný program začíná funkcí `main`, které představuje vstupní bod programu. Funkce se definuje klíčovým slovem `fn` (zkratka pro *function*), za kterým následuje název funkce a kulaté závorky.

Tělo funkce je uzavřeno složenými závorkami. Vše, co se nachází uvnitř těchto závorek, tvoří takzvaný *blok*. Bloky se využívají ke strukturování kódu funkcí, podmínek a cyklů.

Řádek s příkazem `println!("Hello, world!");` slouží k výpisu na standartní výstup, což je obvykle terminál nebo příkazová řádka. V tomto případě se nejedná o běžnou funkci, ale o tzv. *makro*. Makra[23)] v jazyce Rust se poznají podle vykřičníku za názvem a mají tu vlastnost, že se zpracovávají už při překladu programu. To znamená, že mohou upravit nebo rozšířit kód ještě před jeho spuštěním.

Uvnitř kulatých závorek se nachází řetězec "Hello, world!", který je uzavřen v uvozovkách. Každý příkaz je zakončen středníkem, který označuje konec příkazu.

Teď po rozboru syntaxe je na čase program spustit pomocí příkazového řádku. Aby bylo možné s projektem pracovat, je nutné přepnout se do jeho složky v příkazovém řádku. To lze provést pomocí příkazu `cd ahoj_svete`. Tímto příkaz se přesunu do složky projektu, odkud lze provádět operace, jako je spouštění programu. Pokud se omylem přesunu do nesprávné složky, mohu přejít do nadřazené složky příkazem `cd ..` .

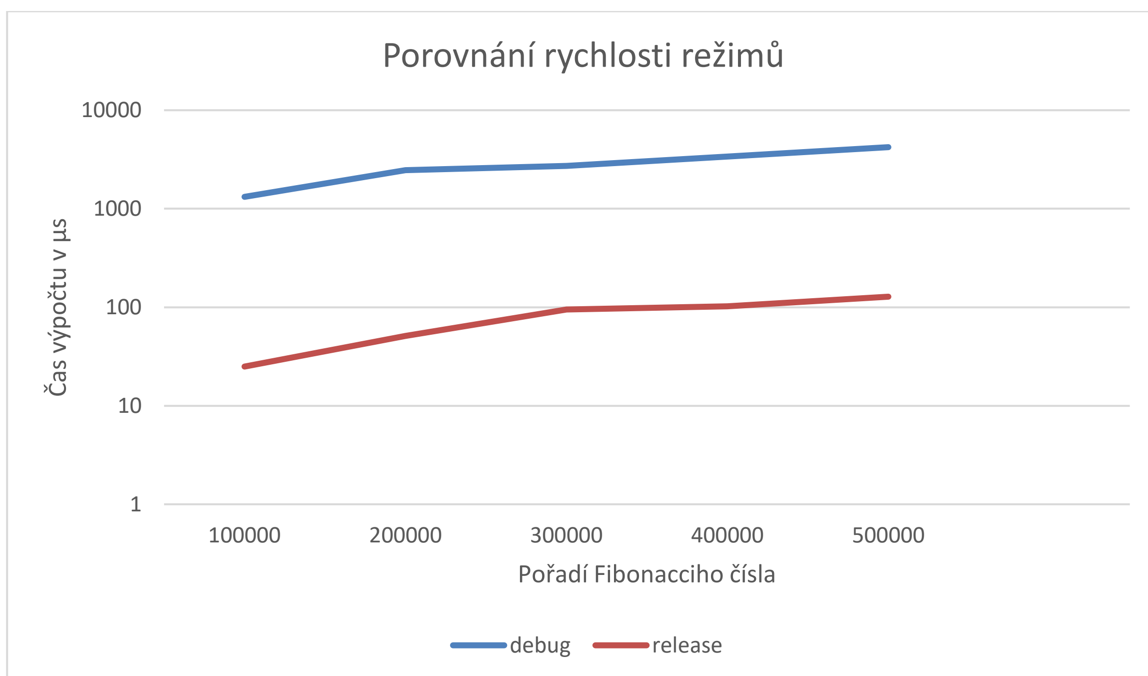
Program lze zkompilovat a pustit příkazem `cargo run`, případně pouze zkompilovat příkazem `cargo build`. Jako výstup příkazu je několik řádků:

1. Compiling …
    - Tento řádek obsahuje, že nástroj Cargo kompiluje projekt, jeho název, verzi a jeho lokaci na disku.
2. Finished …
    - Řádek s informací, že projekt byl úspěšně kompilován a jak dlouho trvalo projekt kompilovat
3. Running …
    - Informace o tom, jaký zkompilovaný soubor byl spuštěn
4. Hello, World!
    - Řádek se samotným výstupem programu

Samotný spustitelný soubor se bude nacházet v adresáři `target/debug`, a bude mít stejný název, jako je název projektu.

Při kompilaci Rust projektu pomocí nástroje *Cargo* lze zvolit mezi dvěma režimy, *debug* a *release*. Výchozí režim, ve kterém nástroj operuje, je *debug*. Tento režim je určen pro vývoj a ladění programu. Kompilace programu je rychlejší, ale výsledný program obsahuje ladící informace, není optimalizovaný pro výkon a je větší.

Režim *release* se aktivuje pomocí přidáním přepínače `--release` a používá se pro finální verze programu. Tento režim vytvoří optimalizovaný program, který urychlí jeho běh a sníží jeho výpočetní náročnost. Výsledkem je menší a rychlejší binární soubor. Tento režim je ideální pro pozdější vytváření programů pro řídicí kostku EV3, Výsledný spustitelný soubor poté bude uložen do adresáře `target/release`.



Graf 3: Porovnání rychlosti programu kompilovaného v režimech debug a release

Na předchozím grafu s logaritmickou osou *Y* je zobrazen rozdíl rychlosti programu při výpočtu Fibonacciho čísel pro různé vstupní hodnoty. Zatímco program zkompilovaný v režimu *debug* dosahuje času v tisících mikrosekund (tedy milisekund), *release* verze programu je výrazně rychlejší a vypočítá výsledky desítky krát rychleji.

2.3 Proměnné

Práce s proměnnými je jedním ze základních prvků každého programovacího jazyka. V Rustu je však způsob jejich použití mírně odlišný oproti jiným jazykům. V Rustu jsou ve výchozím nastavení *neměnné*, což znamená, že jejich hodnota nemůže být po přiřazení změněna. Tento přístup pomáhá předcházet chybám v kódu, zejména v rozsáhlejších aplikacích.

Pro tuto kapitolu vytvořím nový projekt se jménem *promenne* (v anglickém jazyce *variables*). Před vytvořením projektu se musím ujistit, že se nacházím v pracovním adresáři, a ne v některém jiném projektu. Poté ho mohu vytvořit příkazem `cargo new promenne`.

Poté v adresáři nového projektu otevřu soubor `src/main.rs`, nahradím jeho obsah následujícím kódem a soubor uložím:

```
fn main() {
    let x = 12;
    println!("Hodnota x je: {x}");
}
```

V kódu jsou dvě nové věci. Jako první je definice proměnné, která se nachází na druhém řádku. V Rustu se definují proměnné pomocí klíčového slova `let`, které říká kompilátoru, že vytváříme novou proměnnou. V tomto případě na druhém řádku je definována proměnná `x` a je jí přiřazena hodnota 12. Pravidla pro pojmenování proměnných jsou stejná, jako pro projekty.

Druhá nová věc, která se nachází na třetím řádku, je *interpolace*. Ta umožňuje Rustu automaticky nahradit `{x}` aktuální hodnotou proměnné `x`, v tomto případě 12.

Alternativně by šlo příkaz napsat následujícím způsobem. Obě varianty fungují stejně, ale zápis pomocí `{x}` je v případě krátkých názvů proměnných přehlednější:

```
println!("Hodnota x je: {}", x);
```

2.3.1 Proměnlivost proměnných

Rust standardně nastavuje proměnné jako neměnné, jejich hodnotu tedy nelze po přiřazení změnit. Je to dáno základní myšlenkou jazyka Rustu, tedy bezpečné práce s pamětí, což zvyšuje bezpečnost programu a snižuje pravděpodobnost nečekaných chyb. Mohlo by se stát, že v komplexním projektu by do proměnné byla omylem uložena jiná hodnota, která může způsobit nesprávnou funkčnost programu.

Pokud je potřeba, aby bylo možné hodnotu přepsat, musíme použít klíčové slovo `mut`. To kompilátoru říká, že proměnná není pevně daná a její hodnota se může v průběhu programu měnit.

Pro ukázku poslouží následující kód:

```
fn main() {
    let mut x = 12;
    println!("Hodnota x je: {x}");
    x = 15;
    println!("Hodnota x je: {x}");
}
```

V definici proměnné `x` klíčové slovo `mut` říká, že proměnnou lze v budoucnu přepsat. Zároveň jí bude přiřazena hodnota 12.

Pokud budu později hodnotu proměnné změnit, provedu to příkazem ve formátu `proměnná = hodnota;`.

Po spuštění programu v příkazovém řádku se program úspěšně spustí, a v jeho výpisu je vidět, že se hodnota proměnné skutečně přepíše. Bez označení `mut` by druhé přiřazení způsobilo chybu překladu. Překladač by ohlásil, že nelze přiřadit novou hodnotu do neměnné proměnné.

Chybová hláška bude mít tuto podobu:

```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src\main.rs:4:5
  |
2 |     let x = 12;
  |         - first assignment to `x`
3 |     println!("Hodnota x je: {x}");
4 |     x = 15;
  |     ^^^^^^ cannot assign twice to immutable variable
  |
help: consider making this binding mutable
  |
2 |     let mut x = 12;
  |         +++
```

Součástí chybové hlášky je i návrh na řešení, a to přidání klíčového slova `mut`.

2.3.2 Rozsah a stínování

Každá proměnná v rámci programu má definovaný svůj *rozsah platnosti* (anglicky *scope*)[24)], který určuje, kde je možné tuto proměnou používat. Rozsah je obvykle omezen blokem kódu. Pokud je proměnná definována uvnitř bloku, nelze ji použít mimo něj.

Specifická vlastnost je stínování (anglicky *shadowing*) [24)]. Stínování nastává tehdy, když je ve stejném rozsahu vytvořena nová proměnná se stejným názvem, jako již existující. V takovém případě dojde k „překrytí“ původní proměnné novou. Původní proměnná zůstává

nedotčena, ale v daném rozsahu ji nelze použít. Místo ní se pracuje právě s nově definovanou proměnou.

Následující příklad ukazuje jednoduché stínování nejen hodnoty, ale i datového typu:

```
fn main() {
    let x = 5;
    println!("Původní hodnota x: {x}");

    {
        let x = x + 1;
        println!("Hodnota x uvnitř vnořeného bloku: {x}");
    }
    println!("Hodnota v nadřazeném bloku zůstane: {x}");

    let x = "text";
    println!("Nová hodnota jiného typu: {x}");
}
```

Díky stínování je možné nejen měnit hodnotu proměnné bez použití `mut`, ale dokonce i její datový typ. To je v Rustu vlastnost, která umožňuje zachovat neměnnost proměnných, a přitom upravovat jejich hodnotu. Je důležité poznamenat, že při stínování se původní hodnota neaktualizuje, ale vzniká zcela nová proměnná.

2.3.3 Konstanty

Zvláštním typem proměnných jsou konstanty[25)], jejichž hodnota je pevně daná a nelze ji během běhu programu změnit. Konstanty jsou definovány pomocí klíčového slova `const`. Hodnota konstanty a její datový typ musí být definovány v době kompilace. To znamená, že ji nelze přiřadit hodnotu výpočtu, který by byl znám až po spuštění programu.

Dalším důležitým rozdílem je, že u konstant nelze použít klíčové slovo `mut`. Každý pokus o označení konstanty jako proměnlivé skončí chybou při kompilaci. Rust tím zajišťuje, že hodnota konstanty zůstane skutečně neměnná a nemůže být omylem přepsána jinou částí programu.

Konstanty jsou nejčastěji používány pro uchování fixních hodnot, jako jsou například koeficienty, neměnné parametry aplikace nebo matematických konstant. Konstanty se zapisují s použitím velkých písmen.

```
const PI: f32 = 3.1415196;
```

Na rozdíl od proměnných mají konstanty širší rozsah platnosti – lze je používat v celém programu, pokud jsou definovány na globální úrovni (mimo funkci `main`).

2.4 Datové typy

Každá hodnota v Rustu má přiřazený datový typ[26)]. Ten určuje, jaká data mohou být v proměnné uložena a jak s nimi lze pracovat. Rust je *staticky typovaný jazyk*, což znamená, že datový typ proměnných musí být známý již při kompilaci programu. Hlavní výhodou je, že překladač dokáže včas upozornit na chyby, které by vedly k nečekanému chování programu.

Rust často dokáže odvodit, jaký typ chceme použít na základě hodnoty. Proto v předchozích příkladech nebylo nutné určit u proměnných jejich datových typ. Každopádně je vhodné u proměnných datové typy uvádět. Datový typ konstant musí být uveden vždy.

Rust podporuje několik datových typů, od jednoduchých čísel až po složitější struktury, jako jsou pole.

Celá čísla (anglicky *integer*) jsou čísla bez desetinné složky. Tento datový typ má dvě skupiny: *signed* a *unsigned* (česky „se znaménkem“ a „bez znaménka“). Tyto typy určují, zda číslo může být záporné. Jinými slovy, zda může být číslo i záporné (v oboru matematiky *celé číslo*), nebo jestli číslo bude vždy kladné (v oboru matematiky *přirozené číslo*).

Následující tabulka ukazuje vestavěné celočíselné typy. K deklaraci proměnné lze použít kteroukoli z těchto variant.

Bitová délka	**Signed** (se znaménkem)	**Unsigned** (bez znaménka)
8 bitů	i8	u8
16 bitů	i16	u16
32 bitů	i32	u32
64 bitů	i64	u64
128 bitů	i128	u128

Tabulka 3: Typy celočíselného datového typu[26)]

Rozsah hodnot pro každou délku čísla se znaménkem je od $-(2^{n-1})$ do $2^{n-1}-1$ včetně, kde `n` je bitová délka, které varianta používá. Například do datového typu `i16` lze uložit hodnoty v rozsahu $-(2^{15})$ do $2^{15}-1$, což jsou čísla od -32 768 až 32 767.

Varianty bez znaménka mohou ukládat čísla v rozsahu `0` do $2^{n}-1$. Například do typ `u8` lze uložit čísla v rozsahu od `0` do 2^{8}, neboli 0 až 255.

```
let a: i16 = -500;
let b: u8 = 200;
```

Rust také podporuje **čísla s desetinnou čárkou**. Rust podporuje dvě varianty tohoto typu, a to `f32` a `f64`, které mají velikost 32bitů a 64 bitů. Typ `f32` zabírá méně paměti, je rychlejší na méně výkonných zařízení, ale má malou přesnost – přibližně 6-7 číslic za desetinnou čárkou. Typ `f64` má vyšší přesnost, přibližně 15 číslic za desetinnou čárkou, a je výchozím typem pro desetinná čísla.

```
let vyska: f32 = 1.75;
let hmotnost = 68.5;   // výchozí typ je f64
```

Pro **logické hodnoty** je použit typ `bool`. Tento typ může nabýt pouze dvou hodnot, `true` (pravda) a `false` (nepravda). Tento datový typ bývá velmi často použit v podmínkách.

```
let odeslano = true;
let doruceno = false;
```

Datový typ `char` je schopen uložit jednotlivé **znaky**, přičemž znaky se zapisují do jednoduchých uvozovek. Rust používá kódování *Unicode*[27)], takže kromě znaků anglické abecedy lze do něj uložit znaky s diakritikou, řecká písmena, emotikony apod.

```
let znak: char = 'Ř';
let srdce = '❤';
```

Textové řetězce lze v Rustu ukládat pomocí typu `String`[28)], který je vhodný pro většinu situací, kdy se pracuje s textem. Tento datový typ také používá kódování *Unicode*.

```
let jmeno = String::from("Petr");
let prijmeni = String::from("Novák");
println!("{jmeno} {prijmeni}");
```

Pole slouží k uchovávání pevného počtu hodnot stejného typu. Délka pole je známa už při překladu programu a nelze ji měnit. Ke konkrétním prvkům pole se přistupuje pomocí indexu. První prvek má index 0. Pro získání hodnoty délky pole, tedy počet uložených prvků.

Práce s polem je zobrazena v následujícím ústřižku kódu:

```
let cisla = [1, 2, 3, 4, 5];
let druhy = cisla[1];       // vrátí hodnotu 2
let delka = pole.len();     // vrátí hodnotu 5
```

K-tice, na rozdíl od pole, umožňují spojit více hodnot různých datových typů do jedné proměnné. Délka k-tice musí být známa již při kompilaci, stejně jako typy jednotlivých prvků. Přístup ke konkrétním prvkům se provádí pomocí tečkové notace a číselného indexu.

```
let osoba = ("Pavel", 25, true);
let jmeno = osoba.0;
let vek = osoba.1;
let muz = osoba.2;
```

2.5 Komentáře

Komentáře jsou velmi důležitou součástí každého programovacího jazyka. Pomáhají programátorům zpřehlednit a vysvětli, jaké je funkce části kódu nebo proč je takovým způsobem napsán. Tím se mnohonásobně ulehčí oprava staršího napsaného kódu a zároveň usnadní spolupráci více osob na jednom projektu.

Rust převzal syntaxi komentářů z programovacího jazyka *C*[29)]. Jedním ze dvou základních typů komentářů jsou *jednořádkové komentáře*, které začínají dvěma lomítky, tedy `//`. Všechny znaky, které se v řádku nacházejí za těmito lomítky se stanou součástí komentáře

a budou pro kompilátor neviditelné. Tento typ komentářů se nejčastěji používá pro krátké poznámky.

Druhým základním typem jsou *víceřádkové komentáře*, které začínají znaky `/*` a končí znaky `*/`. Pomocí nich lze zapsat komentář přes několik řádků. Také lze použít pro dočasné „vypnutí“ části kódu, aby se během vývoje programu nemusel mazat a poté znovu napsat. Oba typy komentářů jsou znázorněny v následujícím kódu:

```
// Toto je jednoduchý komentář
fn main() {
    /*  Toto je tělo víceřádkového komentáře
      let rok = 2015;
      println!("Rust byl vydán v roce {rok}");
     */
    println!("RUST!");
}
```

Speciálním typem komentářů jsou takzvané *dokumentační komentáře*. Lze je poznat podle tří lomítek `///` nebo podle `//!`. Tyto komentáře umožňují automatické generování dokumentace pomocí nástroje *rustdoc*[30)], který z nich vytváří přehlednou HTML[31)] dokumentaci. Ačkoli v této práci nebudou použity, uvádím je zde, aby čtenáře nepřekvapily v případě, že na ně narazí v dokumentaci jazyka Rust pro LEGO Mindstorms.

2.6 Funkce

Funkce v jazyce Rust, ale i v ostatních programovacích jazycích, přestavují základní strukturu programu. Umožňují rozdělit složitý kód na menší, přehlednější a znovupoužitelné části. Pomocí funkcí se tedy zvyšuje čitelnost a udržovatelnost zdrojového kódu.

Pro tuto kapitolu vytvořím nový Rust projekt s názvem *funkce* a obsah souboru *main.rs* přepíši následovně:

```
fn main() {
    let vysledek = soucet(6, 11);
    println!("Soucet je: {}", vysledek);
}
```

```
fn soucet(a: i32, b: i32) -> i32 {
    a + b
}
```

Deklaraci základní funkce `main` je popsána v kapitole 2.2 Syntaxe, proto začnu popisem funkce `soucet`. Nová část v deklaraci je „`->  i32`“, což znamená že funkce bude vracet hodnotu, jejíž návratový datový typ se definuje pomocí `->` následovaný seznamem parametrů (v tomto případě `i32`).

Tato funkce přijímá dvě celočíselné hodnoty a vrací jejich součet. V těle funkce chybí středník, protože v Rustu platí, že poslední výraz v těle funkce bez středníku je automaticky považován za návratovou hodnotu. Alternativní zápis pro návratovou hodnotu je pomocí příkazu return. Tento způsob používá většina programovacích jazyků a zápis funkce by vypadal následovně:

```
fn soucet(a: i32, b: i32) -> i32 {
    return a + b;
}
```

Teď zpátky k funkci `main` a definici proměnné `vysledek`. Funkce se volají pomocí jejich názvu a předáním argumentů v závorkách. V tomto případě program volá funkci `soucet(6,11)`, jejíž návratová hodnota bude 17, která se uloží do proměnné `vysledek`. Tento postup si lze představit v těchto krocích:

1. `let vysledek = soucet(6, 11);`
    - Program zavolá funkci s parametry
2. `let vysledek = 17;`
    - Funkce vrátí hodnotu 17. Ta je poté přiřazena proměnné `vysledek`

Funkce lze deklarovat i bez návratové hodnoty. Tento typ je ekvivalent příkazu *void* v jiných programovacích jazycích. Tento typ funkce se používá například pro práci s proměnnými nebo pro výpis textu.

Ukázka funkce bez návratové hodnoty:

```
fn main() {
    let x = 5;
    pricti_jednicku(x);
}
fn pricti_jednicku(a: i32) {
    println!("O cislo vetsi: {}", a + 1);
}
```

2.7 Řízení toku programu

Řízení toku programu (anglicky *control* flow)[32)] je jednou ze základních součástí programovacích jazyků. Umožňuje programátorovi rozhodnout, jestli se určité bloky kódu mají vykonat, případně kolikrát se mají opakovat. V Rustu existuje několik způsobů pro řízení toku programu, ale v této práci budou použity základní: podmínky a cykly. Jejich způsob zápisu jsou podobné většině programovacích jazyků.

Pro úplnost je vhodné zmínit, že Rust nabízí i další pokročilé konstrukce, jako je například porovnávání vzorů nebo nekonečný cyklus s možností přerušení. Tyto konstrukce bývají používány ve složitějších případech, ale pro rámec této práce budou stačit následující uvedené konstrukce.

2.7.1 Podmínky

Pomocí podmínek lze spustit části kódu pouze v případě, když bude splněna zadaná logická podmínka. Nejčastěji se používá konstrukce `if` (česky *pokud*). Zápis její nejjednodušší varianty vypadá následovně: `if <hodnota> {…}`. Obsah podmínky se spustí pouze, když obsah podmínky má hodnotu větší nebo rovno jedné. Podmínka se tedy spustí v případě, že `hodnota` má logickou hodnotu `true`, která nejčastěji vznikne porovnáním dvou proměnných.

Tato konstrukce bývá často doplněná případnou větví `else` (česky „jinak“). Tento druh podmínky, známý jako *if-else*, je znázorněn v následujícím kódu.

```
fn main() {
     let cislo = 12;
     let limit = 7;

     if cislo > limit {      // podmínka porovnává hodnoty proměnných
           // pokud je podmínka pravdivá (cislo je větší než limit),
           // spustí se tento blok kódu a blok else se přeskočí
           println!("Číslo {cislo} je větší než limit {limit}.");
     } else {
           // pokud není podmínka pravdivá, blok if se
           // přeskočí a spustí se pouze blok else
           println!("Číslo {cislo} je menší než limit {limit}.");
     }
}
```

Během psaní programu však nastávají případy, kdy je potřeba použít více podmínek. Pro tyto případy se používá *else-if*. Lze říci, jde o několik podmínek postupně za sebou. Pokud není splněna podmínka `if`, kontroluje se splnění následující podmínky bloku `else if`. V podmínce může být hned několik `else if`, ale způsob spouštění bloků kódu funguje podobným způsobem, jako u `if else`.

Následující program má tři možné případy, které mohou nastat:

```
fn main() {
    let cislo = 12;

    if cislo > 0 {
        println!("Cislo {cislo} je kladné.");
    } else if cislo < 0 {
        println!("Cislo {cislo} je záporné.");
    } else{
        println!("Cislo {cislo} je nula.");
    }
}
```

2.7.2 Cykly

Cyklus (anglicky *loop*) je blok kódu, který se opakuje, dokud není splněna podmínka, nebo není ukončen příkazem `break`[33)]. Tento oddíl bude zaměřen na cykly `while` a `for`.

Cyklus while

Cyklus `while` se provádí opakovaně do té doby, dokud je splněná jeho logická podmínka. Jakmile podmínka bude vyhodnocena jako nepravdivá, cyklus se ukončí a běh programu bude pokračovat dál. Cyklus jde znázornit následujícím příkladem kódu:

```
fn main() {
    let mut x = 0;

    while x <5 {
        println!("Hodnota x: {x}");
        x = x + 1;
    }

    println!("Cyklus skončil s hodnotou x: {x}");
}
```

Abych upřesnil výpis programu, tak v posledním opakování smyčku byla hodnota `x` 4. Tato hodnota splnila požadovanou podmínku, cyklus proběhl a k `x` přičetl jedničku. Na začátku dalšího cyklu již neplatila podmínka `x <5`, proto se cyklus přerušil a program pokračoval dál.

Cyklus for

Tento cyklus se často používá k procházení prvků v poli nebo k iteraci v daném rozsahu čísel. Pro procházení všech prvků pole je preferovaný následující způsob zápisu, protože programátor nemusí kontrolovat platnost indexů jako v jiných jazycích, čímž se snižuje riziko chyb přístupu k paměti mimo rozsah pole.

```
fn main() {
    let pole = [10, 40, 30, 20, 50];

    for prvek in pole {
        println!("Aktuální prvek je: {prvek}");
```

```
    }
}
```

V tomto případě se proměnná `prvek` při každé iteraci naplní hodnotou jednoho prvku z pole. Díky tomu není nutné používat indexy.

Kromě procházení prvků v poli je možné tento cyklus procházení rozsahu čísel, který se obvykle vyjadřuje pomocí dvou teček. Výchozí syntaxi představuje konstrukce `start..end`, která vytvoří posloupnost od `start` do `end - 1`.

```
fn main() {
    for i in 0..5 {
        println!("i = {i}");
    }
}
```

2.8 Struktury

Struktury v jazyce Rust slouží k definici vlastních datových typů. Pomocí struktur lze sdružit více hodnot do jednoho celku. Tento mechanismus je podobný *objektům* v jiných jazycích. Používání struktur bude klíčové pro programování a ovládání robotické stavebnice.

Struktura se definuje pomocí klíčového slova `struct`, za kterým následuje název struktury. Názvy struktur začínají velkým písmenem. Poté v těle struktury lze definovat názvy a typy dat, kterým se říká *pole*. Následující ústřižek kódu zobrazuje strukturu, která ukládá velmi základní informace o osobě.

```
struct Osoba {
    jmeno: String,
    vek: u8,
}
```

Pro použití struktury lze vytvořit *instanci* této struktury zadáním konkrétních hodnot pro každé z polí. Instance se tvoří tak, že se uvede název struktury a poté se přidají složené závorky obsahující páry *klíč: hodnota*, kde klíč je název pole a hodnota jsou data, která budou do těchto polí uložena. Hodnoty nemusí být k polím přiřazena ve stejném pořadí, jako ve kterém byla pole deklarována ve struktuře.

Pokud bude v budoucnu zapotřebí změnit hodnoty ve struktuře, je nutné proměnou osoba deklarovat jako proměnlivou pomocí `mut`. K samostatným vlastnostem struktury lze přistoupit pomocí tečkové notace, jako u K-tic.

```
let mut osoba = Osoba {
    jmeno: String::from("Jan"),
    vek: 25,
};
println!("{}, je mu {} let", osoba.jmeno, osoba.vek);
```

Ke každé struktuře lze implementovat *metoda* pomocí bloku `impl`. Všechny metody struktury se zapisují uvnitř tohoto bloku.

Metody jsou podobné funkcím: jsou deklarovány pomocí klíčového slova `fn` a názvu, mohou mít parametry, návratovou hodnotu a obsahují nějaký kód. Na rozdíl od funkcí jsou metody jsou definovány v kontextu struktury a jejich prvním parametrem je vždy `self`, který představuje instanci struktury, na kterou je metoda volána.

```
impl Osoba {
    fn pozdrav(&self) {
        println!("Ahoj, jmenuji se {}.", self.jmeno);
    }
}
```

V tomto případě je metoda `pozdrav` navázána na strukturu `Osoba`. Metody se volají pomocí tečkové notace.

```
osoba.pozdrav();
```

V ostatních programovacích jazycích lze vytvářet nové objekty (v Rustu struktury) pomocí konstruktoru a klíčového slova *new*, čímž odpadá nutnost přiřazovat ručně hodnoty ke každé vlastnosti. Rust tento konstruktor nemá, ale běžným postupem je definovat tzv. přidruženou funkci (anglicky *associated function*), obvykle pojmenovanou `new`, která vytvoří instanci dané struktury. Tato funkce je definovaná v bloku `impl` a funguje na stejném principu jako konstruktor.

```
impl Osoba {
    fn new(jmeno: &str, vek: u8) -> Self {
        Self {
            jmeno: String::from(jmeno),
            vek,
        }
    }
}
```

Poté lze k vlastnostem struktury přiřadit hodnoty zavoláním této funkce, jejíž parametry budou hodnoty vlastností.

```
let tomas = Osoba::nova("Tomas", 28);
```

V tradičních objektově orientovaných programovacích jazycích, jako je například Java, se používají *gettery* a *settery*[34)] pro zpřístupnění nebo změnu vnitřního stavu objektu. Rust tento přístup nevyžaduje, protože umožňuje přímý přístup k veřejným polím struktury. Přesto je vhodné definovat metody ve formátu `get_nazev_metody()` nebo `set_nazev_metody()`, pokud bude v budoucnu zapotřebí například kontroly dat při změně hodnoty. Dodržování tohoto pojmenování navíc zvyšuje čitelnost a konzistenci kódu.

Jednoduchý program pro používání struktury Osoba může vypadat následovně:

```
struct Osoba{
    jmeno: String,
    vek: u32,
}
impl Osoba{
    fn new(jmeno: String, vek: u32) -> Self {
        Self { jmeno, vek }
    }
    fn get_jmeno(&self) -> &String {
        &self.jmeno
    }
    fn get_vek(&self) -> u32 {
        self.vek
    }
```

```
    fn set_vek(&mut self, vek: u32) {
        self.vek = vek;
    }
    fn pozdrav(&self) {
        println!("Jsem {} a je mi {} let!", self.jmeno, self.vek);
    }
}

fn main(){
    let mut osoba = Osoba::new("Vladislav".to_string(), 45);
    let mut osoba2 = Osoba::new("Karel".to_string(), 50);
    println!("Jméno: {}", osoba.get_jmeno());
    println!("Věk: {}", osoba.get_vek());
    osoba2.set_vek(51);
    osoba2.pozdrav();
}
```

2.9 Moduly

Rust používá systém modulů k rozdělení funkcí, struktur a dalších prvků do přehledných částí. Tyto moduly jsou součástí knihoven, které lze do projektu přidat. Díky modulům lze jednoduše zpřístupnit (tzv. *importovat*) pouze ty části kódu, které program aktuálně využívá. Tím se sníží velikost a výpočetní náročnost výsledného programu.

Každý modul představuje logickou skupinu souvisejících funkcí nebo struktur. Moduly mohou být vnořené a používá se zápis pomocí dvojteček. V této práci bude nejvíce využíváno funkcí `std::thread::sleep`[35)]a struktury `std::time::Duration`[36)]ze standartní knihovny `std`[37)], která je součástí každé instalace Rustu. Tyto moduly slouží k pozastavení programu na určitý čas a k definování, jak dlouho má být program pozastaven.

Pokud je potřeba využít tyto prvky, doporučuje se jejich explicitní zavedení pomocí klíčového slova `use`. Následující zápis umožní použití funkce `sleep` a `Duration`, aniž by bylo nutné opakovaně udávat celou cestu.

```
use std::thread::sleep;
use std::time::Duration;
fn main(){
    println!("Zpodění jedné sekundy...");
    sleep(Duration::from_secs(1));
    println!("... hotovo");
}
```

Alternativně by šlo použít následující zápis bez použití `use`, ale je delší, při častém používání velmi nepraktický a zhoršuje čitelnost zdrojového kódu:

```
fn main(){
    println!("Zpodění jedné sekundy...");
    std::thread::sleep(std::time::Duration::from_secs(1));
    println!("... hotovo");
}
```

V této práci budou používány externí moduly pro ovládání prvků robotické stavebnice a vestavěné moduly pro pozastavení programu. Bez pozastavení by některé programy pracovaly rychleji, než by bylo jejich činnost možné postřehnout.

3 Linux pro LEGO Mindstorms EV3

Stavebnice LEGO Mindstorms EV3 je poslední a nejvýkonnější z generace robotických stavebnice od společnosti *LEGO*. Tato populární robotická stavebnice se používá jako pomůcka pro výuku algoritmického myšlení a programování. Hlavní ovládací kostka této stavebnice využívá základní proprietární firmware od společnosti LEGO, který podporuje jednodušší programovací jazyky vhodné pro začátečníky, ale pro složitější úlohy to může být značně limitující faktor. Alternativní možností je použití operačního systému *ev3dev*, který je založen na operačním systému Linux.

Linux je open-source operační systém, který je široce využíván nejen na běžných počítačích, ale také na mobilních telefonech, nebo ve vestavěných systémech, jako jsou například síťové routery a zařízení pro průmyslovou automatizaci. Hlavními důvody, proč je tento operační systém používán, je jeho stabilita, časté aktualizace zvyšující bezpečnost systému a možnost jeho detailní konfigurace.

Tento systém odemyká nové možnosti robotické stavebnice, jako je využití pokročilých programovacích jazyků, nebo možnost připojení nezávislých zařízení pomocí kabelu i bezdrátových sítí Wi-Fi a Bluetooth. Kromě toho systém umožňuje použití více periferií, než je podporováno ze strany společnosti LEGO[38)].

Systém ev3dev se instaluje na paměťovou kartu. Pokud bude paměťová karta z robota vyjmuta, kostka bude používat původní firmware.

3.1 Porovnání s originálním firmwarem

Použití linuxového operačního systému přináší hned několik výhod. Mezi hlavní výhody patří otevřenost systému. Ta nejen rozšiřuje přístup k ovládání řídící kostky, ale také umožňuje instalaci dodatečného software pro přidání dalších funkcí. Navíc umožňuje nadšeným členům komunity systém stále vyvíjet, přestože společnost LEGO přestala tuto stavebnici prodávat v červnu roku 2021[39)] a ukončila podporu koncem roku 2022. Jako ukázkový příklad člena komunity lze uvést elektrotechnickou fakultu ČVUT[40)], která vyvinula stavebnici Brian[41)] , která má sloužit jako náhrada za již dosluhující stavebnici Mindstorms EV3.

Další, již zmiňovaná výhoda, je podpora množství pokročilých programovacích jazyků. Řídicí kostka s originálním firmware jde programovat v podstatě jen vizuálních programovacích jazyků. Oproti tomu linuxový systém *ev3dev* plně podporuje hned několik programovacích jazyků[42)].

Výhodou je i lepší řízení toku programu a kontrola výkonu. Díky plnému přístupu k jádru systému lze optimalizovat využití procesoru a operační paměti, což zvýší výkon ve složitých programech.

Mezi nevýhody linuxového systému patří vyšší složitost nastavení systému a jeho používání. Originální firmware nabízí příjemnější uživatelský zážitek, protože spustit a naprogramovat stavebnici lze hned po prvním rozbalení z krabice.

Další nevýhodou je, že oproti originálnímu firmware je systém *ev3dev* náročnější na hardware řídicí kostky. Již při startu systému je tento rozdíl poznat tak, že se systému trvá načíst až dvě minuty. Tato nevýhoda se dá vyřešit použitím rychlých a nenáročných programovacích jazyků, jako je Rust nebo C, nebo případným přetaktováním procesoru řídicí kostky. Druhá možnost může zkrátit živnost řídicí kostky, ale přesto popíši, jak lze řídicí kostku přetaktovat. Tuto možnost doporučuji pouze v případě, že výchozí výkon kostky nestačí.

Změnou také bude zobrazení stavu baterie. V původním firmwaru je pomocí ikony baterie s čárkami, zatímco ev3dev udává stav pomocí velikosti napětí. Tento ukazatel se nachází v levém horním rohu displeje. Baterie nebo akumulátor jsou plně nabité, pokud napětí přesáhne 8 voltů. Pokud napětí klesne na úroveň 5 voltů, kostka se sama vypne.

3.2 Instalace

Nejprve je nutné stáhnout komprimovaný *.zip*[43)] soubor s obrazem nejnovější verze operačního systému ev3dev, který se nachází na oficiálním GitHub repozitáři[44)].

Operační systém lze nainstalovat na *microSD* nebo *microSDHC* kartu, ke které bude potřeba i adaptér, pomocí kterého je možné připojit kartu k počítači. Tato karta by dle doporučení[45)] měla mít kapacitu 2 až 32 GB. Použití paměťové karty s větší kapacitou se nedoporučuje.

Nyní bude možné nahrát systémový obraz na paměťovou kartu. Tento proces se nazývá *vypalování* nebo *flashování* (z anglického *flash*, vypálit). Tímto procesem se extrahují

systémové oddíly a soubory ze staženého souboru a zapíšou se na paměťovou kartu takovým způsobem, aby bylo možné z paměťové karty zpustit operační systém. Běžně používaný program pro vypalování obrazů je *Balena Etcher*[46)], který je dostupný pro všechny běžně používané operační systémy.

Postup pro vypálení obrazu je následující:

1. Stažení a nainstalování programu z oficiálních stránek[46)] pro čtenářův operační systém
2. V prvním poli programu vyberte stažený *.zip* soubor[43)], obsahují obraz pro ev3dev
    a. Program umí sám extrahovat soubory ze *.zip* archivu, ale doporučuji nejdříve archiv rozbalit a nahrát samotný *.img*[47)] soubor.
3. V druhém poli vyberte paměťovou kartu. Ujistěte se, že na paměťové kartě nejsou uloženy žádné soubory, které Vám budou chybět. Tento proces **smaže všechna uložená data** na paměťovém zařízení.
4. Po ujištění, že se obraz vypálí na správnou paměťovou kartu, pro zahájení vypálení stačí kliknout na tlačítko "Flash!".
5. Vypalování zabere i několik desítek minut, záleží na rychlosti disku a paměťové karty. Po dokončení vyjměte paměťovou kartu z počítače.

Pro spuštění linuxového systému na řídicí kostce EV3 stačí do vypnuté kostky z boku vložit paměťovou kartu a zapnout kostku. Po krátké chvíli by se měl začít načítat operační systém ev3dev. Načtení systému se dá poznat podle nápisu "EV3DEV" v horní části obrazovky.

Je dobré upozornit, že prvního spouštění systému bude trvat i několikrát déle, protože systém během prvního spuštění bude dokončovat instalaci a vytvářet konfigurační soubory.

3.3 Nastavení

Po úspěšném spuštění systému ev3dev z paměťové karty je nastavení systému, aby se k řídicí kostce šlo pomocí počítače připojit a nahrávat na ni hotové programy. Popíši způsob propojení s počítačem způsobem připojení řídicí kostky do společné sítě pomocí Wi-Fi a její ovládání pomocí *SSH* protokolu[48)]. Tento způsob není jediný[49)], ale je nejjednodušší na nastavení a na praktické používání.

Menší nevýhodou je, že je zapotřebí vlastnit vhodný USB Wi-Fi adaptér. Na stránkách systému *ev3dev* je uvedeno několik doporučených adaptérů, ale není nutné přesnou shodu. Například já budu používat adaptér *TP-Link TL-WN725N*[50] , který používá čip *RTL8188EUS*[51] od společnosti *Realtek*. Osobně jsem vyzkoušel i adaptér *TL-WN722N v2/v3*[52], který používá stejný čip a fungoval. Pokud tedy bude použit jiný adaptér se stejným čipem, pravděpodobně bude fungovat.

Také bych doporučil použití spíše menšího adaptéru oproti adaptéru s velkou externí anténou. Malé adaptéry oproti větším sice mohou mít mírně zhoršený příjem signálu, ale na druhou stranu se dají velmi jednoduše zakomponovat do konstrukce robota.

Po připojení Wi-Fi adaptéru je možné se připojit k bezdrátové síti pomocí nabídky “**Wireless and Network**” → “**Wi-Fi**”. Ujistěte se, že je zaškrtnuté políčko “Powered”, jinak nebude Wi-Fi zapnuté a nebude se možné k síti připojit.

Poté se ovládací kostka najde viditelné Wi-Fi sítě a stačí už jen zvolit požadovanou síť ze seznamu, zadat heslo a připojit se do sítě. Doporučím zapnout automatické připojení k síti pomocí nabídky “**Wireless and Network**” → “**All Network Connections**” → **<název sítě>** a zaškrtnutím políčka “Connect automatically”.

Po úspěšném připojení do sítě se v levém horním rohu obrazovky zobrazí IPv4 adresa[53], která byla kostce udělena. Pokud se žádná adresa nezobrazí, je nutné zkontrolovat připojení k síti nebo na kostce dodatečně ručně nakonfigurovat síť. Nyní je možné navázat spojení mezi počítačem a kostkou pomocí této IP adresy. Nejčastěji se k pro vzdálené ovládání po sítí používá protokol SSH protokol, který umožňuje bezpečné a šifrované spojení.

Pro připojení na kostku z počítače otevřete na počítači příkazový řádek a napište příkaz `ssh robot@<ip>,` kde `<ip>` nahradíte IP adresou kostky. Při prvním připojení bude chtít počítač potvrdit, jestli se opravdu chcete připojit k tomuto zařízení, protože ho počítač ještě nezná. Stačí napsat *yes* a stisknout klávesu *Enter*. Poté pro dokončení připojení bude zapotřebí heslo kostky, které je v základu **maker**. Během psaní se nebude heslo zobrazovat, jedná se bezpečnostní prvek, který neukazuje délku hesla, čímž zabraňuje potenciální odhadnutí hesla podle počtu znaků. Uživatelský účet, na který se lze připojit, má jméno **robot** a heslo **maker**.

3.3.1 Základní příkazy pro ovládání systému ev3dev

Po úspěšném připojení bude uživatel přesměrován do příkazové řádky systému ev3dev na řídicí kostce. Všechny příkazy, které budou do příkazové řádky se tedy nebudou spouštět na počítači, ale na ovládací kostce EV3.

```
PS C:\Users\novak> ssh robot@192.168.1.106
The authenticity of host '192.168.1.106 (192.168.1.106)' can't be established.
ED25519 key fingerprint is SHA256:RvV/Cl7n2PrfwKbvHrVzQBYBl/xlY1ix38oXIUqVV0Q.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '192.168.1.106' (ED25519) to the list of known hosts.
(robot@192.168.1.106) Password:
Linux ev3dev 4.14.117-ev3dev-2.3.5-ev3 #1 PREEMPT Sat Mar 7 12:54:39 CST 2020 armv5tejl
             _____     _
   _____   _|___ /  __| | _____   __
  / _ \ \ / / |_ \ / _` |/ _ \ \ / /
 |  __/\ V / ___) | (_| |  __/\ V /
  \___| \_/ |____/ \__,_|\___| \_/

Debian stretch on LEGO MINDSTORMS EV3!
Last login: Fri Apr  4 13:41:24 2025 from 192.168.1.10
robot@ev3dev:~$
```

Obrázek 2: Připojení pomocí protokolu SSH

Operační systém ev3dev založený na systému Debian Linux, proto podporuje stejné příkazy, jako běžné linuxové distribuce. Níže budou zmíněny jen základní příkazy, které jsou pro práci se stavebnicí nejdůležitější. Detailní popis práce s Linuxem by byl mimo rozsah této práce, proto doporučuji získat informace pomocí jiných zdrojů[54)]. Zároveň doporučuji najít si podrobnější informace o následujících příkazech55).

Příkaz cd

Příkaz `cd` slouží k přesunu uživatele mezi adresáři. Například příkazem `cd test` se uživatel přesune do adresáře s názvem „test“. Pokud je potřeba se přesunout do nadřazeného adresáře, lze to provést příkazem `cd ..`.

Příkaz ls

Tento příkaz vypíše obsah aktuálního adresáře. Pokud je zapotřebí zobrazit více informací o souborech, lze to provést příkazem `ls -l`.

Příkaz mkdir

Příkaz na vytváření nových adresářů. Pomocí příkazu `mkdir ev3_rust` se vytvoří adresář s názvem „ev3_rust“.

Příkaz rm

Příkaz pro mazání souborů je `rm`. Například pro smazaní souboru „test“ slouží příkaz `rm test`. Pro smazání celého adresáře, včetně jeho obsahu, se používá příkaz `rm -r adresar`.

Příkaz chmod

Tento příkaz slouží ke změně oprávnění souboru. Pokud se například při spuštění programu zobrazí chyba „Permission denied“, je nutné nastavit soubor jako spustitelný příkazem `chmod +x program`.

3.3.2 Zvýšení taktu procesoru

Řídicí kostka využívá procesor TI Sitara AM1808[56)], který je v základu systémem taktován na frekvenci 300 MHz. Tato frekvence pro běžné použití stačí, ale u komplexních programů může být výkon nedostačující. Jelikož je systém ev3dev založený na open-source, je možné frekvenci procesoru zvýšit, čemuž se říká přetaktování (anglicky *overclocking*).

Přetaktováním se zvýší pracovní frekvence procesoru, což může zrychlit načítání systému, spouštění programu i jeho samotný běh. Toto zvýšení výkonu ale přináší určitá rizika. Vyšší frekvence znamená větší tepelnou zátěž a může vést ke snížení životnosti procesoru nebo ke ztrátě stability systému. Z tohoto důvodu silně doporučuji pouze bezpečné přetaktování na frekvenci 375 MHz, která nevyžaduje zvýšení napájecího napětí, a zároveň je stabilní oficiální frekvencí procesoru. Už pouze toto zvýšení by mělo zajistit teoretické zrychlení procesoru a jeho výpočetní rychlosti o 25 %.

Změna frekvence se provádí úpravou konfiguračního souboru, který se nachází v adresáři `/boot/flash` a má název `uEnv.txt`. Pro navýšení frekvence je nutné otevřít tento soubor na najít řádek s textem `#cpufreq=375`. Tento řádek je ve výchozím nastavení zakomentovaný znakem `#`, takže pro aktivaci parametru pro nastavení frekvence je nutně tento znak odstranit. Výsledná podoba řádku by tedy bude `cpufreq=375`. Po úpravě je nutné soubor uložit a kostku restartovat, aby se nastavení nové frekvence projevilo.

Toto upravení souboru lze po připojení ke kostce pomocí SSH provést příkazem `sudo sed -i s/#cpufreq/cpufreq/g /boot/flash/uEnv.txt`. Příkaz bude vyžadovat potvrzení zadáním hesla. Alternativně lze tento soubor upravit po vyjmutí microSD karty a vložení do počítače. Tento soubor by se měl nacházet na FAT oddílu paměťové karty.

4 Programování stavebnice a vzorové programy

V této kapitole je popsáno, jak upravit Rust projekty, aby je bylo možné spustit na řídicí kostce stavebnice EV3. Dále jsou v kapitole uvedeny jednoduché příklady programů napsaných v jazyku Rust, pomocí kterých lze ovládat robotické prvky stavebnice. V kapitole také bude několika případech uvedeny čísla součástek stavebnice EV3 i NXT, čímž bude ujasněno, o jaké součástky se jedná.

4.1 Nastavení projektu pro ev3dev

Pro programování robotické stavebnice LEGO Mindstorms EV3 v jazyce Rust je nutné připravit Rust projekt tak, aby bylo možné spustit kompilovaný program na ovládací kostce.

Tyto úpravy projektu budou trochu složitější, ale jsou nutné pro spuštění programů na ovládací kostce. Ta totiž používá architekturu procesoru typu ARM, přesněji ARM9[56)], který není kompatibilní s programy vytvořenými pro architekturu x86-64[57)], které používají běžné počítače. A jelikož program bude kompilován na počítači, je nutné nastavit projekt pro tzv. křížovou kompilaci[58)] (anglicky *cross-compilation*), kdy nástroj *cargo* zkompiluje program pro přímou kompatibilitu s procesorem v ovládací kostce.

Nejdříve je nutný vytvořit běžný Rust projekt pomocí nástroje *Cargo*. V pracovním adresáři tedy vytvořím projekt příkazem `cargo new ev3`, a přepnu se do něj příkazem `cd ev3`. Nyní je nutné nastavit cílovou platformu pro kompilaci programu. Cílová platforma tedy bude architektura ARM s linuxovým systémem a do projektu ji lze přidat příkazem `rustup target add armv5te-unknown-linux-musleabi`. Tento příkaz stáhne a nainstaluje do projektu podporu pro kompilaci pro kostku EV3, přičemž tento proces bude trvat jen několik vteřin.

V dalším kroku je zapotřebí vytvořit v kořenovém adresáři adresář `.cargo` a v něm soubor `config.toml` s následujícím obsahem:

```
[build]
target = "armv5te-unknown-linux-musleabi"

[target.armv5te-unknown-linux-musleabi]
linker = "rust-lld"
```

Tento konfigurační soubor a jeho obsah zajistí, že kompilovaný program bude kompatibilní s architekturou procesoru EV3 kostky.

Nyní je možné využívat pouze vestavěné funkce jazyku Rust a není možné ovládat prvky stavebnice, jako jsou motory a senzory. Proto je nutné v posledním kroku přidání závislostí, aby Rust používal knihovnu vytvořenou pro stavebnici EV3, jinak by program nevěděl, jak má prvky stavebnice ovládat. Pro přidání je zapotřebí do souboru *Cargo.toml* přidat řádek `ev3dev-lang-rust = "0.15.0"`. Před samotným přidáním je vhodné ověřit, zda se jedná aktuální verzi knihovny na crates.io[59)]. Níže je uvedený příklad výsledné podoby souboru *Cargo.toml*:

```
[package]
name = "ev3"
version = "0.1.0"
edition = "2024"

[dependencies]
ev3dev-lang-rust = "0.15.0"
```

Po správném nastavení projektu je možné projekt zkompilovat pomocí příkazu `cargo build --release`. Pokud během kompilace nastane chyba, je nutné ji před pokračováním vyřešit. Výsledný spustitelný soubor, jehož název bude stejný, jako název projektu, se uloží do složky `target/armv5te-unknown-linux-musleabi/release/`.

Tento soubor je nyní nutné nahrát do řídicí kostky EV3. Nejjednodušší způsob nahrání je pomocí nástroje *scp*[60)]. Tento nástroj umí přenášet soubory mezi zařízení přes síť, k čemuž využívá protokol SSH. Pomocí příkazu `scp target/armv5te-unknown-linux-musleabi/release/ovladani_ev3 robot@<ip>:~` se program zkopíruje z počítače do domovské složky uživatele `robot` na kostce EV3 (je

nutné nahradit <*ip*> IP adresou kostky). Pro samotné zahájení kopírování bude zapotřebí zadání hesla **maker**.

Pro spuštění programu na kostce je nutné se k ní připojit pomocí SSH. Doporučuji si pro toto spojení otevřít nové terminálové okno, které bude sloužit pouze ovládání robota, a to původní používat jen k sestavování Rust projektů. Tím se zpřehlední, který příkazový řádek patří čemu, a usnadní to práci.

Poté lze spustit program příkazem `./ev3`. Pokud se program nespustí a zobrazí se chybová hláška „Permission denied“, program nemá uděleno oprávnění spustit se. To se dá jednoduše opravit příkazem `chmod +x ev3`, který soubor nastaví jako spustitelný.

Pokud se program spustil a vypsal „Hello world!“, je vše nastaveno správně a lze postoupit k programování samotné stavebnice.

Jelikož bude SSH připojení využíváno opakovaně, doporučuji nahrát na zařízení EV3 veřejný SSH klíč počítače. Tímto krokem se výrazně usnadní práce, protože odpadne nutnost zadávat heslo při každém připojení nebo při kopírování souborů pomocí nástroje *scp*.

Na operačních systémech Linux a macOS lze vytvořit a nahrát nové klíče na kostku EV3 těmito dvěma příkazy zadaných do terminálu:

```
ssh-keygen -t ed25519 -C "ev3dev"
ssh-copy-id robot@<ip>
```

Na systémech Windows 10 a novějších, lze použít podobné příkazy v prostředí PowerShell[61]):

```
ssh-keygen -t ed25519 -C "ev3dev"
type $env:USERPROFILE\.ssh\id_ed25519.pub | ssh robot@ev3dev.local "mkdir -p ~/.ssh && cat >> ~/.ssh/authorized_keys"
```

4.2 Vstupně-výstupní prvky kostky

V tomto oddíle budou popsány jednotlivé vstupní a výstupní prvky a způsoby jejich programování. Tyto prvky umožňují základní komunikaci mezi uživatelem a robotickou stavebnicí. Mezi tyto prvky patří LED diody, vestavěný reproduktor, tlačítka a obrazovka. Všechny tyto prvky lze ovládat za použití knihovny `ev3dev-lang-rust`, která poskytuje

vhodné metody pro jejich využití, které značně ulehčí práci se samotným ovládáním prvků i programováním.

Kromě knihovny pro EV3 bude v ukázkách často použita knihovna `std`, což je standardní knihovna[37)] jazyka Rust, která obsahuje běžně používané funkce a nástroje. Nejčastěji z ní budou použity moduly, funkce a struktury pro pozastavení běhu programu, aby bylo možné postřehnout výstupy programu, jako například změny barvy LED diod nebo zobrazeného textu na displeji.

4.2.1 LED signalizace

Okolo prostředního tlačítka jsou umístěny dvě LED diody, levá a pravá, které lze použít pro signalizaci aktuálního stavu programu. Knihovna pro EV3 umožňuje nezávislé nastavení barvy každé z nich. Diody umí svítit v červené a zelené barvě, včetně barevné kombinace těchto dvou barev.

Následující program postupně přepíná barvu obou barevných diod, přičemž jsou použity předdefinované konstanty reprezentující různé barevné kombinace. Zobrazení každé barvy je nastaveno na dobu jedné sekundy. Zároveň program vypíše hodnoty barevné kombinace diod na výstup.

Je možné nastavení vlastní barevný odstín LED diody pomocí metody `led.set_left_color((R, G))`, kde argumentem je dvojice hodnot, jejíž prvky je intenzita červené a zelené barevné složky. Obě hodnoty jsou datového typu `u8`, tedy celé číslo v rozsahu 0 až 255.

```
use ev3dev_lang_rust::Led;
use ev3dev_lang_rust::Ev3Result;
use std::{thread::sleep, time::Duration};

fn main() -> Ev3Result<()> {
    let led = Led::new()?;

    let colors = [
        Led::COLOR_RED, Led::COLOR_ORANGE,
        Led::COLOR_GREEN, Led::COLOR_AMBER,
        Led::COLOR_YELLOW, Led::COLOR_OFF,
```

```
    ];

    for color in colors {
        println!("Změna barvy LED na (R, B): {:?}", color);
        led.set_left_color(color)?;
        led.set_right_color(color)?;
        sleep(Duration::from_secs(1));
    }
    Ok(())
}
```

Program začíná importem potřebných modulů a definuje hlavní funkci `main`, která vrací typ `Ev3Result`. Tento návratový typ je použit proto, že volané funkce mohou skončit chybou – například pokud není dostupné potřebné zařízení nebo pokud dojde k chybě při zápisu do souborového systému. Rust umožňuje pomocí operátoru `?` tyto chyby automaticky předávat bez nutnosti explicitního zpracování, což ulehčuje práci s programováním.

Výraz `Ok(())` na konci funkce `main` slouží k označení, že program úspěšně skončil bez chyby. Vzhledem k tomu, že funkce vrací typ `Result`, je nutné při úspěšném průběhu vrátit hodnotu `Ok`, a protože `main` nevrací žádný konkrétní výsledek, použije se prázdný typ `()` (tzv. jednotkový typ). Tento zápis je běžnou konvencí v Rustu pro funkce, které mohou skončit chybou.

4.2.2 Zvukový výstup

Pomocí modulu `sound` lze přes vestavěný reproduktor přehrávat zvukové tóny, zvukové soubory a převádět text na syntetizovanou řeč (*text-to-speech*)[62)]. Reproduktor tak může sloužit například k signalizaci dokončení úlohy nebo upozornění na chybu. Následující ukázka demonstruje způsob pro přehrávání tónu, převod textu na řeč z textu a přehrání sekvence tónů.

```
use ev3dev_lang_rust::Ev3Result;
use ev3dev_lang_rust::sound;
```

```
fn main() -> Ev3Result<()> {
    sound::tone(440.0, 300)?.wait()?;
    sound::speak("Rust on EV3")?.wait()?;
    sound::tone_sequence(&[
        (400.0, 300, 50),
        (500.0, 150, 50),
        (600.0, 150, 50),
        (700.0, 150, 50),
    ])?.wait()?;
    Ok(())
}
```

První zvukový výstup zajišťuje funkce `tone`, která přehraje jednoduchý tón o frekvenci 440 Hz (odpovídající tónu komorní A)[63)] po dobu 300 milisekund. Metoda `wait` následně zajistí, že program bude čekat na dokončení přehrávání.

Ve druhém kroku se využívá funkce `speak`, která převádí textový řetězec "Rust on EV3" na syntetizovanou řeč. Výstup opět čeká na dokončení přehrávání pomocí `wait`. Tato funkce aktuálně umí syntetizovat pouze anglický jazyk.

Ve třetím kroku program využívá funkci `tone_sequence`, která přehraje sekvenci čtyř tónů. Každý tón této sekvence je prvkem K-tice a má tři parametry: frekvenci tónu v Hz, délku tónu v milisekundách a délku pauzy po jeho odehrání.

4.2.3 Tlačítka jako vstup

Na přední straně kostky se nachází šest tlačítek, přičemž nedoporučuji k ovládání programu používat tlačítko „Zpět". Tyto tlačítka fungují jako klávesy na klávesnici[64)] a tlačítkem „zpět" se ukončuje spuštěný program. Pokud se program spustí skrz SSH připojení, tak k žádnému problému nedojde. Když ale bude program spuštěný pomocí rozhraní na kostce, přenastavením chování tlačítka může znemožnit ukončení programu.

Ostatní tlačítka lze využít například pro přepínání režimů programu, pozastavení a spuštění činnosti nebo jiným vstupům od uživatele.

Kromě jednoduché detekce jednoho stisknutí lze vytvářet i složitější kombinace, jako stisk více tlačítek, nebo počet stisknutí tlačítka za určitý časový úsek. Pro detekci stisknutí tlačítek lze použít následující kód[65)]:

```
extern crate ev3dev_lang_rust;

use ev3dev_lang_rust::{Button, Ev3Result};

fn main() -> Ev3Result<()> {
    let button = Button::new()?;

    loop {
        button.process();

        println!(
            "{}, {}, {}, {}, {}, {}",
            button.is_up(),
            button.is_down(),
            button.is_left(),
            button.is_right(),
            button.is_enter(),
            button.is_backspace(),
        );
        println!("{:?}", button.get_pressed_buttons());

        std::thread::sleep(std::time::Duration::from_secs(1));
    }
}
```

4.2.4 Práce s obrazovkou

Na řídicí kostce stavebnice je umístěn černobílý LCD displej. Tento displej lze použít například pro výpis informací o stavu programu a zobrazení dynamických dat, jako například výstup ze senzoru. Zobrazení na displej je omezeno velikostí 178x128 pixelů, proto je vhodné výpis zjednodušit. Nejjednodušší forma výpisu textu na obrazovku je použitím

`println!()`. Pro zobrazení textu na displeji kostky je nutné spustit program přímo na kostce, a ne skrz SSH protokol.

Pro složitější výpisy lze použít modul `screen` z knihovny `ev3dev-lang-rust`, pomocí kterého lze na obrazovku vykreslit i obrazce. Nejprve je zapotřebí vytvořit instanci struktury `Screen`, pomocí které půjde ovládat obsah na obrazovce. Pomocí metody `clear()` struktury `Screen` se obsah obrazovky vymaže a metoda `update()` provede aktualizaci displeje a vykreslí obsah struktury `Screen`. Tento přístup je ale složitější, má specifické použití a je zapotřebí do projektu přidat další závislosti. Proto doporučuji pro použití postupovat podle knihovny[66)].

4.3 Ovládání motorů

Motory patří mezi jedny ze základních prvků každého robota. Umožňují robotovi pohyb a manipulaci s objekty. Stavebnice Mindstorms EV3 obsahuje výkonné krokové motory, které umožňují přesné řízení rychlosti i směru otáčení. K dispozici jsou dva typy motorů, a to velký motor (součástka 45502) a střední motor (součástka 45503), Knihovna `ev3dev-lang-rust` umožňuje ovládání i velkého motoru ze starší stavebnice NXT (součástka 53787).

Knihovna nabízí pro ovládání motorů tři struktury. Struktura `LargeMotor` je určena pro velké motory ze stavebnic EV3 a NXT. Pomocí struktury `MediumMotor` lze ovládat pouze střední motor. Struktura `TachoMotor` je hlavní strukturou, pomocí které lze ovládat všechny zmíněné motory. Všechny tyto struktury spolu sdílejí stejné metody.

Následující kód ukazuje, jakým způsobem mohou být motory ovládány. V kódu jsou použity všechny tři zmíněné struktury.

```
extern crate ev3dev_lang_rust;
use   ev3dev_lang_rust::motors::{LargeMotor,   MediumMotor,   MotorPort,
TachoMotor};
use ev3dev_lang_rust::Ev3Result;

fn main() -> Ev3Result<()> {

    let motor_a = LargeMotor::get(MotorPort::OutA)?;
    let motor_b = TachoMotor::get(MotorPort::OutB)?;
```

```
    let motor_c = MediumMotor::get(MotorPort::OutC)?;

    motor_c.set_speed_sp(300)?;
    motor_c.run_forever()?;

    motor_a.set_speed_sp(250)?;
    motor_a.set_time_sp(2500)?;
    motor_a.run_timed(None)?;
    motor_a.wait_until_not_moving(None);

    motor_b.set_speed_sp(200)?;
    motor_b.run_to_rel_pos(Some(-720))?;
    motor_b.wait_until_not_moving(None);

    motor_a.run_timed(None)?;
    motor_a.wait_until_not_moving(None);
    motor_c.stop()?;
    Ok(())
}
```

Metodou `set_speed_sp()` se motoru nastaví rychlost otáčení. Parametrem je celé číslo v rozsahu -1000 až 1000. Při záporné hodnotě rychlosti se motor bude otáčet v opačném směru.

Pro otočení motorem o určitý úhel lze použít metodu `run_to_rel_pos()`. Velikost vstupní hodnoty odpovídá velikosti datového typu *i32*. Při použití této metody má na směr otáčení motoru vliv znaménko u velikosti úhlu.

Metoda `run_forever()` spustí otáčení motorem na dobu neurčitou, dokud nedostane motor nějakou jinou instrukci nebo pokyn `stop()` pro zastavení činnosti. Pokud motoru nebude řečeno jinak, bude se otáčet i po skončení programu.

Pomocí metody `set_time_sp()` lze nastavit dobu v milisekundách, po jakou se bude motor otáčet. Následné spuštění otáčení se provádí metodou `run_timed()`. Tato metoda se kombinuje s metodou `wait_until_not_moving(None)`, která pozastaví běh programu, dokud se motor nepřestane otáčet.

Motorům také lze nastavit způsob zastavení. Výchozím nastavení je tzv. „coast“, kdy se motor po vykonání činnosti volně dotáčí setrvačností. V některých případech může být vhodnější, aby se motor ihned zastavil. Tato vlastnost lze nastavit metodou `set_stop_action()`, přičemž možné parametry jsou následující:

- "coast" – volné dojetí motoru
- "brake" – aktivní brzdění, které urychleně zastaví motor
- "hold" – motor se zastaví a působí proti vnější síle, která by s motorem chtěla otočit

Správné zvolení druhu bržd̆ění závisí na situaci. Například pro přesné ovládání jízdy je vhodné použít vlastnost *brake*, která zkracuje brzdnou dráhu robota. V některých případech ale můžou robotovi při rychlém zastavení podklouznou kola. Tím robot začne ztrácet přesnost, která je od něj vyžadována.

4.4 Získávání dat ze senzorů

Senzory umožňují robotovi získávat data o okolním prostředí a tím mu umožní základní orientaci v prostoru. Mezi základní senzory patří spínače, světelný senzor, ultrazvukový senzor a gyroskop.

Nejjednodušším typem senzorů jsou **dotykové spínače**. Jejich výstupem je logická hodnota *pravda* v sepnutém stavu, a *nepravda* při rozepnutém stavu. Nejčastěji se tyto spínače používají pro detekci nárazu robota do zdi nebo jako spouštěč nějaké funkce. Knihovna umí pracovat se spínači ze sady EV3 (součástka 95648) i NXT (součástka 53793).

```
extern crate ev3dev_lang_rust;
use ev3dev_lang_rust::sensors::TouchSensor;
use ev3dev_lang_rust::Ev3Result;
use std::thread::sleep;
use std::time::Duration;

fn main() -> Ev3Result<()> {

    let tlacitko = TouchSensor::find()?;
    loop{
        println!("tlacitko: {:?}",  tlacitko.get_pressed_state());
```

```
        sleep(Duration::from_millis(250));
    }
}
```

Metoda `TouchSensor::find()` najde a použije první nalezený sensor a funguje v případě, když je ke kostce připojený jeden senzor tohoto typu. Pro použití většího počtu senzorů je nutné použít strukturu `SensorPort`, kterou lze přidat pomocí `use ev3dev_lang_rust::sensors::SensorPort;`. Poté lze nastavit senzoru samotné číslo vstupu pomocí `TouchSensor::get(SensorPort::In1)?;`, kde `In1` bude nahrazeno vstupem, do kterého je senzor zapojený.

Dalším typem je **světelný senzor**, který umí měřit intenzitu odraženého světla a barvu. Knihovna uvádí, že umí pracovat se senzorem ze stavebnice EV3 (součástka 95650).

```
extern crate ev3dev_lang_rust;

use ev3dev_lang_rust::sensors::ColorSensor;
use ev3dev_lang_rust::Ev3Result;
use std::thread::sleep;
use std::time::Duration;

fn main() -> Ev3Result<()> {
    let cs = ColorSensor::find()?;

    // Měření intenzity odraženého světla
    cs.set_mode_col_reflect()?;

    loop {
        println!("REF: {:?}", cs.get_color()?);
        sleep(Duration::from_secs(1));
    }
}
```

Nastavování čísla vstupu senzoru a použití více senzorů najednou je stejné, jako u dotykových tlačítek. V režimu rozpoznávání barev je návratnou hodnotou číslo 0 až 7, které reprezentuje barvu.

Návratová hodnota	0	1	2	3	4	5	6	7
Barva	Žádná barva	Černá	Modrá	Zelená	Žlutá	Červená	Bílá	Hnědá

Tabulka 4: Číselné hodnoty reprezentující barvy[67)]

Senzor navrátí hodnoty v rozsahu 0 až 100, pokud je v režimu měření intenzity odraženého světla nebo intenzity okolního světla.

V režimu měření intenzity barevných složek světla je návratovou hodnotou funkce `get_rgb()` trojice čísel v rozsahu 0 až 1020. Pokud je zapotřebí získat intenzitu pouze jedné zvolené barevné složky, jde to provést pomocí metod `get_red()`, `get_green()` a `get_blue()` pro červenou, zelenou a modrou složku.

V následující tabulce je uvedeno, do jakých režimů lze senzor nastavit, jak se nastaví a jakou metodou lze ze senzoru získat data.

Režim	Nastavení režimu	Získání dat
Měření intenzity odraženého světla	set_mode_col_reflect()?	get_color()?
Měření intenzity okolního světla	set_mode_col_ambient()?	get_color()?
Rozpoznání barvy	set_mode_col_color()?	get_color()?
Měření intenzity jednotlivých barevných složek světla	set_mode_col_rgb_raw()?	get_rgb()?

Tabulka 5: Režimy barevného senzoru

Ultrazvukový senzor se používá k detekci překážek a měření vzdálenosti až na 255 cm. Funguje na principu vysílání vysokofrekvenčních tónů, které se odrazí od vzdáleného objektu. Robot změří čas, za jaký se odražená vlna vrátí, a podle něj vypočítá vzdálenost objektu. V programu lze nastavit, zda bude robot udávat vzdálenost v centimetrech nebo palcích. Podle dokumentace knihovna podporuje senzor ze stavebnice EV3 (součástka 95652). O senzoru ze sady NXT se knihovna nezmiňuje.

```
extern crate ev3dev_lang_rust;
use ev3dev_lang_rust::Ev3Result;
use ev3dev_lang_rust::sensors::UltrasonicSensor;
use std::thread::sleep;
use std::time::Duration;

fn main() -> Ev3Result<()> {

    let us = UltrasonicSensor::find()?;
    us.set_mode_us_dist_cm()?;

    // Nastavení měření vzdálenosti v palcích
    // us.set_mode_us_dist_inch()?;

    // Poslouchání cizího ulztrazvuku
    // us.set_mode_us_listen()?;
    loop {
        println!("{:?}", us.get_distance()?);
        sleep(Duration::from_secs(1));
    }
}
```

Předchozí program nastavuje režim měření na metrickou soustavu pomocí metody `set_mode_us_dist_cm()`, jejíž návratovou hodnotou je vzdálenost v milimetrech. Ultrazvukový senzor lze přepnout do imperiální soustavy pro měření vzdálenosti v palcích pomocí metody `set_mode_us_dist_in()`.

Tento senzor má také režim, ve kterém pouze poslouchá vysokofrekvenční tóny a žádné nevysílá. Tím robot dokáže zjistit, jestli nějaký jiný robot v jeho blízkosti používá ultrazvukový senzor. Tento režim lze nastavit pomocí metody `set_mode_us_listen()`, přičemž návratovým typem je číslo 0 a 1. Pokud senzor zachytí vysílání cizího senzoru, návratová hodnota bude 1.

Gyroskop (součástka 99380), je poslední základní senzor, který bude v této práci zmíněn. Ten je klíčový pro dvoukolé balancující roboty nebo pro přesnou kalibraci otáčení robota. Tento senzor umí měřit otáčení kolem jedné osy, která je vyobrazena na těle senzoru.

Senzor lze nastavit do režimu měření úhlového náklonu a režimu měření úhlové rychlosti. Režim měření úhlového náklonu pracuje v rozsahu -32768 až 32767 stupňů. Režim měření úhlové rychlosti pracuje v rozsahu -440 až 440 stupňů za sekundu.

Při používání gyroskopu je důležité provést jeho kalibraci. Běžným jevem je *drift*[68)], tedy samovolná změna naměřených hodnot, i když se senzor nepohybuje. Kalibrace je nejvíce efektivní, pokud je gyroskop v době kalibrace v klidu.

```
extern crate ev3dev_lang_rust;
use ev3dev_lang_rust::sensors::GyroSensor;
use ev3dev_lang_rust::Ev3Result;
use std::thread::sleep;
use std::time::Duration;

fn main() -> Ev3Result<()> {
    let gs = GyroSensor::find()?;

    gs.set_mode_gyro_cal()?;
    gs.set_mode_gyro_ang()?;

    loop{
        println!("{:?}", gs.get_angle()?);
        sleep(Duration::from_millis(100));
    }
```

V ukázce je senzor kalibrován metodou `set_mode_gyro_cal()` a poté je nastaven na režim měření úhlu metodou `mode_gyro_ang()`. Hodnota je získána metodou `get_angle()`. Pro měření úhlové rychlosti je nutné nastavit režim metodou `set_mode_gyro_rate()` a následně získat návratovou hodnotu metodou `get_rotational_speed()`.

5 Úlohy a možné řešení

V této kapitole jsou uvedeny praktické úlohy, které lze realizovat s robotickou stavebnicí LEGO Mindstorms EV3. Úlohy jsou navrženy tak, aby odpovídaly úrovni středního odborného vzdělávání a současně reflektovaly současné požadavky rámcového vzdělávacího programu pro obor 18-20-M/01 Informační technologie [69].

Úlohy jsou navrženy tak, aby studentům poskytly možnost procvičit si dovednosti, které souvisí se základním programováním a tvorbou algoritmů. Studenti se zároveň učí pracovat s periferními prvky, propojit programové a konstrukční řešení robota a řešit úkoly, které vyžadují analytický přistup i logické řešení.

Ne vždy to skutečnosti umožňují, ale studenti, popřípadě skupiny studentů, mohou být podle kvality provedení úlohy bodováni. Tyto body mohou být následně sčítány a studenti s nejvyšším počtem bodů mohou být osloveni, jestli by se nechtěli účastnit soutěží.

Přesné znění zadání úloh není pevně stanoveno, protože se může lišit v závislosti na konkrétních podmínkách dané školy, například na technickém vybavení (typ a množství dostupných součástek) nebo na vstupních znalostech a dovednostech žáků. Z tohoto důvodu jsou v textu uváděna modelová řešení, která slouží jako inspirace a mohou být dále upravena podle aktuální situace a konkrétních potřeb výuky.

U každé úlohy je na závěr uveden přehled klíčových a odborných kompetencí, které daná aktivita podporuje v souladu s požadavky RVP pro obor Informační technologie.

Není nutné, aby studenti pro řešení každé úlohy museli sestavovat nového robota. Většinu úloh lze vyřešit pomocí univerzální konstrukce, které bude mírně upravena pro specifickou úlohu.

5.1 Motorka

Jednou z nejjednodušších úloh, která může sloužit jako úvodní aktivita při práci s robotickou stavebnicí, je sestavení a naprogramování motorky. Sice se nejedná o typickou robotickou úlohu, je to vhodná začátečnická úloha, a navíc má didaktický význam. Student je nucen se zamyslet nad konstrukcí robota a uvědomit si, že existuje několik způsobů řízení směru jízdy – nejčastěji konstrukce typu *auto* a *tank*. Zatímco tank se dokáže otáčet na místě díky

dvěma nezávislým motorům, konstrukce typu auto vyžaduje řízení jedné hnací osy a druhé osy, která zajišťuje změnu směru jízdy.

Konstrukce motorky se bude skládat z jednoho velkého motoru zajištující pohon robota. Přední kolo motorky bude upevněno do vidlice napojené na střední motor, který tak bude představovat řídítka. Jednodušší řešení konstrukce bude takové, že velký motor bude pohánět zadní kolo motorky. Ke každé straně motoru bude připojeno jedno kolo, čímž se zvýší stabilita motorky.

Samotné ovládání robota probíhá poměrně jednoduše: velký motor bude motorku pohánět vpřed a střední motor bude otáčet řídítky, čímž bude udávat směr jízdy.

Tato úloha je vhodná zejména jako první krok při seznamování nejen s praktickým ovládání robota pomocí jazyka Rust, ale i se samotnými robotickými prvky stavebnice. Díky své jednoduchosti tak umožní pochopení základních principů navrhování konstrukce robota a jeho následné naprogramování.

Z programové stránky úlohy se řešení týká pouze ovládání motorů a jejich správné synchronizace. Hlavní výzvou bude správné načasování a rychlost otáčení motorů, protože při rychlé jízdě a prudkém zatočení se motorka může převrhnout. Proto doporučuji motorku nejdříve naprogramovat na pomalou jízdu s jemným zatáčením, a poté postupně zvyšovat rychlost obou motorů.

Kvalitu konstrukce robota a přesnost nastavení jeho ovládání se poté může ověřit například jízdou robota po dráze, která připomíná číslo osm. Touto činností se otestuje stabilita konstrukce během jízdy a její symetričnost – jestli se robot během zatáčení nenaklání na jednu stranu více než na stranu druhou. Následně lze hodnotit kvalitu naprogramování podle rychlosti a plynulosti jízdy robota.

5.1.1 Rozvoj kompetencí

Úloha podporuje rozvoj klíčových kompetencí:

- Kompetence k učení:
    - student si vytváří základní pracovní návyky a způsoby získávání poznatků při práci s technologií.
- Kompetence k řešení problémů:

    - navrhování různých variant řízení motorky (řízení směru jízdy – auto vs. tank, stabilita při zatáčení apod.).
- Digitální kompetence:
    - programování a ovládání robotického zařízení (motory, synchronizace, řízení pohybu).

, a odborných kompetencí:

- Programovat a vyvíjet uživatelská, databázová a webová řešení, konkrétně:
    - algoritmizace úloh a tvorba aplikací (v úloze jde o ovládání a synchronizaci motorů);
    - testování a ověřování kvality programů (např. ladění jízdy po osmičce).
- Navrhovat, sestavovat a udržovat hardware, jelikož:
    - student sestavuje konstrukci robota z reálných prvků stavebnice a analyzuje stabilitu.
- Pracovat se základním programovým vybavením:
    - ovládání motorů a jejich parametrizace (rychlost, směr, rozsah, bezpečnost).

5.2 Sledování černé čáry

Sledování a jízda po černé čáře je jedna nejznámějších a nejčastěji zadávaných robotických úloh, která vyžaduje přesné řízení robota a jeho orientaci na základě dat ze senzorů. Cílem je naprogramovat robota tak, aby dokázal sledovat trasu tvořenou černou čárou na světlém pozadí, aniž by z této trasy sjel.

Tento druh úlohy má navíc jednoduchou přípravu. Primitivní dráhu lze vytvořit nalepením černé elektrikářské pásky na světlou podlahu nebo lavici. Pokud bude zapotřebí vytvořit kvalitnější dráhu, lze ji vytisknout na velkoformátové tiskárně. Tuto přesnější dráhu lze vytvořit na počítači, kde lze nastavit přesný poloměr křivky zatáček.

Úloha má jasně měřitelný výstup. Robot buď trasu následuje, nebo z ní sjede. Následně lze hodnotit plynulost jízdy a kvalitu zpracování konstrukce robota. Tato úloha je primárně zaměřena na ladění programu, kdy student musí upravovat parametry rychlosti motorů na základně dat z barevného senzoru. Tím si student ověřuje účinnost programu a učí se hledat optimální hodnoty parametrů.

Pro splnění této a všech následujících úloh, pokud nebude uvedeno jinak, bude nejjednodušším řešením postavení vozítka, jehož konstrukce se bude podobat tanku. Jízda tedy zajištěna pomocí dvou nezávisle řízených motorů. Jízda vpřed a vzad bude realizovaná tak, že oba motory se budou otáčet stejnou rychlostí a zatáčení vozítka bude tvořenou rozdílem rychlostí. Toto řešení umožní otáčení vozítka na místě, pokud se motory budou otáčet navzájem opačnou rychlostí.

Aby byla zajištěna stabilita vozítka, je zapotřebí přidat třetí opěrný bod, který bude mít kontakt se zemí. Jednoduché a účinné řešení je přidělání všesměrové kuličky včetně jejího držáku (součástky 99948 a 65453) na opačnou stranu robota, než jsou umístěny hnací kola.

Barevný senzor, pomocí kterého bude robot detekovat černou čáru, je umístěn mezi hnacími koly tak, aby mířil směrem dolů. Ovládání motorů je poté řízeno jednoduchou podmínkou, ve které se budou porovnávat data ze senzoru. Toto řešení je podobné ostatním řešením úloh pro sledování černé čáry[70)].

Důležité je zmínit, že robot ve skutečnosti nebude sledovat černou čáru, ale pojede po jednom z jejích okrajů, tedy na hranici černé čáry a světlého podkladu. Díky tomu lze řídit směr jízdy pomocí jednoho senzoru. Toto jednoduché řízení využívá metodu „cik-cak“[71)], která funguje na principu jízdy ze strany na stranu. Pokud se senzor bude nacházet nad černou čárou, robot pojede směrem od ní. Naopak, pokud bude senzor nad světlým pozadím, robot se bude otáčet na druhou stranu.

Pro vylepšení výsledku sledování čáry je vhodným postupem před samotnou jízdou robota kalibrovat barevný senzor. Světelné podmínky se mezi jednotlivými jízdami mohou měnit, proto je důležité senzorem naměřit hodnoty černé čáry a světlého pozadí. Pomocí těchto hodnot lze spočítat průměrnou hodnotu, která by měla odpovídat hodnotě přechodu mezi čárou a podkladem.

Tato hodnota robotovi usnadní rozhodování, jestli právě snímá čáru, nebo ne. Pomocí této průměrné hodnoty lze navíc zvýšit plynulost jízdy robota přidáním více podmínek. Pokud se bude momentálně naměřená hodnota barveného senzoru jen o trochu lišit od průměrné hodnoty, robot může zatáčet pozvolně. Pokud se naměřená hodnota bude lišit více, robot bude zatáčet silněji.

5.2.1 Rozvoj kompetencí

Tato úloha přispívá k rozvoji následujících odborných kompetencí:

- Navrhovat a vytvářet algoritmy a programy pro řízení jednoduchých zařízení.
- Používat vhodné senzory a snímače, interpretovat jejich výstupy a využívat je při řízení.
- Realizovat a testovat jednoduché řídicí algoritmy v konkrétním hardwarovém prostředí.
- Vyhodnocovat chování programu v reálném čase a upravovat jeho parametry dle naměřených hodnot.
- Propojovat softwarové a hardwarové komponenty do funkčního celku, který se chová předvídatelně.
- Aplikovat postupy při řešení technických úloh, pracovat metodicky a vyhodnocovat úspěšnost řešení.

5.3 Navigace pomocí barevných značek

Tato úloha přímo navazuje na úlohu **sledování čáry** a je jejím pokračováním. Robot bude sledovat černou čáru, která bude obohacena o barevné kódy, což budou krátké barevné úseky. Ty lze vytvořit například nalepením kousku barevné elektrikářské izolační pásky přes již existující černou dráhu.

Robot pomocí barevného senzoru bude tyto kódy rozpoznávat a podle nich vykonávat určité akce. Každá barva tohoto kódu bude zastupovat určitou instrukci, například:

- Červená: zastav na tři vteřiny
- Modrá: otoč se o 360 stupňů
- Zelená: zahraj náhodný tón

Tuto úlohu lze řešit například nastavením barevného senzoru do režimu měření barevných složek světla. Pokud budou naměřené hodnoty barevných složek poměrně vyrovnané, robot na žádný barevný kód nenarazil a může pokračovat ve sledování černé čáry. Pokud však začne být hodnota jedné barevné složky značně vyšší než ostatní, robot narazil na barevný

kód. Následně se v programu spustí funkce, která vykoná správnou instrukci pro daný barevný kód.

Pokročilou verzí této úlohy může být reagování na barevné sekvence, které zvýší počet instrukcí, které můžou navrhnout studenti. Tyto sekvence se mohou skládat například ze čtyř barevných kódu, přičemž začátek sekvence určuje barevný kód a konec sekvence černá čára. V této verzi mohou nastávat i situace, kdy některé barevné sekvence budou neplatné. Zároveň lze vytvořit sekvence, které budou mít rozdílný význam, pokud je robot naskenuje při jízdě popředu nebo pozpátku.

Řešení pokročilé verze by bylo podobné se základní verzí. Robot by však načítal naskenované barevné kódy do pole a pokud by rozpoznal barevnou sekvenci, provedl by instrukci.

Inspirací pro tuto úlohu byl princip programování robota Ozobot[72)], který využívá podobnou formu příkazů pro řízení robota.

Tato úloha podporuje následující odborné a klíčové kompetence dle RVP SOV:

- Programovat a vyvíjet uživatelská řešení: návrh algoritmu, zpracování dat ze senzoru, reakce robota na podněty.
- Testovat a ověřovat kvalitu programů: kontrola, zda robot správně rozpoznává barvy a adekvátně reaguje.
- Pracovat s aplikačním programovým vybavením a hardwarem: čtení senzorických dat a ovládání motorů dle vstupů.

Z klíčových kompetencí se žák učí:

- Řešit problémy samostatně i ve spolupráci – testování reakcí na různé barvy.
- Využívat digitální technologie kreativně a efektivně – vytváření vlastního systému interpretace barev.
- Zodpovědně přistupovat k plnění úkolů – program je citlivý na detaily, takže je důležitá pečlivá implementace i testování.

5.4 Autonomní vozidlo

Tato úloha částečně navazuje na kapitolu o sledování čáry a je rozdělena na *základní* a *rozšířenou* variantu. Cílem úlohy je vytvořit robota, který je schopen detekovat překážky a následně na ně reagovat. Robot se při jízdě bude vyhýbat překážkám buď náhodně, nebo je v pokročilejší variantě může objíždět.

Základní varianta této úlohy je vhodná do většího prostoru, jako je celá učebna nebo chodba. Situace, ve které se bude v prostoru pohybovat více robotů současně umožní studentům inspirovat se ostatními řešeními a vylepšovat svého vlastního robota.

Překážky lze detekovat pomocí kontaktu robota s překážkou s použitím dotykových spínačů nebo bezkontaktně pomocí ultrazvukového senzoru. Pokud bude k dispozici pouze jeden ultrazvukový senzor, doporučuji použít dotykový spínač, jinak by nebylo možné realizovat rozšířenou verzi.

Robot může nadále využívat stejnou konstrukci tanku, jako v předchozích úlohách. Pro detekci překážek je však zapotřebí přidat na předek robota vhodný senzor. Pokud tímto senzorem bude dotykový senzor, jeho detekční část musí být nejpřednější částí robota, jinak by nebylo možné detekovat náraz do překážky.

Po detekci překážky se robot zastaví a náhodně zvolí směr otočení, čímž se pokusí překážce vyhnout. Tento postup je jednoduchý na programování a seznámení se s generátorem náhodných čísel. Studenti následně mohou nastavit limit úhlu otáčení. Tak mohou sledovat, jaké rozmezí úhlů má nejlepší výsledky pro vyhýbání se překážkám v současném prostoru.

Rozšířená varianta navazuje na dříve řešenou úlohu sledování čáry. Robot bude mít stejnou konstrukci, jako doposud (včetně barevného senzoru), a navíc bude mít na jedné straně umístěný ultrazvukový senzor.

V této variantě bude robot sledovat černou čáru, ale na čáře budou umístěny překážky, které musí robot objet. Při kontaktu s překážkou tedy robot zastaví a otočí se 90° takovým způsobem, aby ultrazvukový senzor na straně robota směřoval na překážku, a následně se podél ní začne pohybovat. Ke sledování překážky lze použít obdobný algoritmus použitý

pro sledování čáry, ale místo barvy bude robot používat vzdálenost od překážky. Po objetí překážky robot znovu najde černou čáru a bude po ní pokračovat.

Obě varianty úlohy mají jasně měřitelný výstup – robot s umí překážce vyhnout, nebo se na překážce zasekne. Při hodnocení lze zohlednit kvalitu konstrukce, přesnost detekce překážky a kvalitu i funkčnost programového řešení.

V rozšířené variantě úlohy se studenti učí naprogramovat robota tak, aby reagoval na více podnětů z jeho okolí. Studenti se tak učí, jak robotovi zadat sekvenci příkazů ve správném pořadí, a tím vytvářet složitější algoritmy.

5.4.1 Rozvíjení kompetencí

Tato úloha přispívá k rozvoji následujících odborných kompetencí:

- Navrhovat a vytvářet algoritmy a programy pro řízení jednoduchých zařízení.
- Implementovat logiku rozhodování na základě údajů ze senzorů.
- Ovládat periferní zařízení (motory, senzory) pomocí vhodných knihoven.
- Navrhnout chování autonomního systému s ohledem na předvídatelnost
- Vyhodnocovat reakce robota a přizpůsobovat program aktuálním podmínkám
- Kombinovat základní algoritmy s praktickým návrhem konstrukce robota.

5.5 Orientace v bludišti

Tento druh úloh je zaměřen na schopnost robota pohybovat se v neznámém bludišti a nalézt cestu k cíli, případně splnit specifický úkol. Důraz je kladen nejen na vhodné konstrukční řešení, ale především na vytvoření efektivní strategie. V tomto oddíle je popsáno několik forem provedení bludišť a tři různé přístupy k řešení těchto úloh.

Tyto bludiště mohou mít různé formy provedení, jako je například bludiště vytvořené z černé čáry na světlém podkladu, bludiště vytvořené pomocí vysokých zdí, nebo kombinované řešení, kdy robot přejíždí z jedné formy bludiště do druhé. Kombinovaná forma bludiště byla použita pro soutěžní úlohu Pathfinder[73)] v Robosoutěži roku 2016. Tyto formy budou vyžadovat podobnou konstrukci robota včetně obdobného programového řešení.

Prvním přístupem k projetí bludištěm je použití základního algoritmu, který byl použit v předhoczích úlohách pro sledování čáry a překážky. Tento algoritmus však může fungovat pouze v jednoduchých bludištích, ve kterých nejsou žádné smyčky a křižovatky. Robot by se mohl zaseknout na sledování izolovaného úseku, který nemůže opustit. Robot se v takovém případě tzv. *zacyklí*.

Druhým přístupem je řešení zmíněných křižovatek. Proto je nutné, aby byl robot konstruovaný tak, aby uměl rozpoznat, jestli se nachází na křižovatce. V případě, že se robot nachází v bludišti vytvořeném pomocí čar, robot bude muset používat alespoň dva barevné senzory – jeden pro sledování čáry a další pro detekci odboček. Jestliže naopak bude bludiště tvořené stěnami, lze použít podobnou konstrukci, jako v rozšířené variantě Autonomního vozidla**,** přičemž senzory budou umístěny vpředu a na boku. Následně se robot na každé křižovatce náhodně rozhodne, jakým směrem se vydá.

Řešení pomocí druhého přístupu je sice funkční, ale není efektivní a je časově náročné. **Třetí přístup** zvyšuje efektivitu průjezdu, ale vyžaduje poměrně komplexní řešení. Tento přístup vyžaduje, aby robot znal svoji polohu v bludišti a během jízdy zaznamenával svoji trasu.

Robot si tedy bude ukládat informace o ujeté vzdálenosti, o projetých křižovatkách a směru, kterým se na křižovatkách vydal. Největší výzvou tohoto přístupu je přesnost. Pokud si robot bude zaznamenávat uraženou vzdálenost pomocí počtu otáček motoru, tak v těchto záznamech bude započítaný i pohyb za strany na stranu. Tento pohyb je vedlejším účinkem algoritmu pro jízdu podle sledovaného objektu.

Tato nepřesnost se může zdát jako zanedbatelná, ale časem se rychle nasčítá. Tento problém lze vyřešit přidáním gyroskopického senzoru. Robot poté bude k ostatním datům zaznamenávat i rychlost náklonu. Poté lze do algoritmu přidat funkci, která bude pomocí dat z gyroskopického senzoru kompenzovat zmíněnou nepřesnost.

Studenti následně mohou být hodnoceni

5.5.1 Rozvoj kompetencí

Tato úloha přispívá k rozvoji odborných kompetencí:

- Navrhovat a vyvíjet algoritmy pro řízení chování robota v dynamickém prostředí.

- Testovat a ladit programy pro různé typy bludišť, optimalizovat strategie pohybu.
- Pracovat se senzory (barevné, vzdálenostní, gyroskopické), analyzovat vstupní data a reagovat na jejich změnu.

, a klíčových kompetencí:

- Řešit problémy samostatně i ve spolupráci – navrhovat, testovat a porovnávat různé strategie.
- Efektivně využívat digitální technologie – zaznamenávání a vyhodnocování dat, interní reprezentace mapy.
- Zodpovědně přistupovat k plnění úkolů – přesnost, robustnost a odolnost vůči chybám jsou klíčové faktory při tvorbě algoritmu.

5.6 Zpětnovazební regulátory

Zpětnovazební regulátory jsou základním stavebním prvkem komplexních řídících systémů, jejichž cílem je dosáhnout požadovaného chování řízení vyhodnocením odchylky mezi požadovanou a skutečnou hodnotou. Na základě této informace se vytváří regulační zásah, jehož cílem je redukovat danou odchylku a tím stabilizovat chování systému. Zpětná vazba je tedy nástroj, který zajišťuje přesnost a adaptivitu systémů.

Termín „systém“ představuje abstraktní pojem pro jakýkoli celek, který lze ovládat. V tomto kontextu lze za systém považovat sestaveného robota.

Velmi často používaným typem zpětnovazebního regulátoru je tzv. *PID* regulátor[74)]. Název regulátor vznikl spojením názvu jeho tří složek – **P**roporcionální, **I**ntegrační a **D**erivační. *Proporcionální složka* regulátoru okamžitě reaguje na vzniklou odchylku a je přímo úměrná její velikosti. *Integrační část* se zaměřuje na celkovou velikost odchylky za časový úsek, čímž umožní kompenzaci odchylek, které není proporcionální složka schopná odstranit. *Derivační člen* regulátoru reaguje na rychlost změny chyby, a tak tlumí náhlé výkyvy a zajišťuje hladší chování.

Význam PID regulace spočívá v její univerzálnosti a jednoduchosti nejen v průmyslovém prostředí, ale i v pedagogickém prostředí pro programování robotických sad. Tyto sady umožňují studentům ověřit fungování regulačních systémů například na robotovi sledujícím

černou čáru. Pokud je například robot sledující černou čáru řízen pouze proporcionálně, může docházet k častému kmitání a vyjíždění z trasy. Přidáním integrační složky by robot přesněji sledoval čáru a přidání derivační složky přispívá k plynulosti jízdy potlačením náhlých změn směru jízdy.

Zavedení zpětnovazebních regulátorů do výuky klade důraz nejen na teoretické porozumění, ale také na schopnost ladit parametry a analyzovat výstupní chování systému. Hodnoty konstant P, I a D jsou často řešeny experimentálním laděním, protože každý systém má jiné vlastnosti a vyžaduje rozdílné hodnoty parametrů. Jako výchozí bod pro ladění může sloužit například simulace nebo Ziegler-Nicholsova metoda, která poskytne přibližné nastavení jednotlivých složek.

Zpětnovazební regulátory tedy nejsou pouze teoretickým konceptem, ale reálným nástrojem, se kterým se studenti odborného vzdělávání velmi pravděpodobně setkají.

Z hlediska odborného vzdělávání má zvládnutí problematiky zpětnovazebních regulátorů důležité přínosy.

5.6.1 Rozvoj kompetencí

Studenti si pomocí praktických úloh rozvíjejí zejména následující kompetence:

- analýza technického problému a návrh vhodného řešení prostřednictvím regulačního algoritmu,
- praktické dovednosti v programování včetně práce s proměnnými, podmínkami a cykly při tvorbě regulační smyčky,
- aplikace matematických a fyzikálních poznatků při ladění parametrů PID regulace,
- porozumění principům měření, zpracování signálu a řízení systémů v reálném čase,
- schopnost interpretace chování systému na základě dat ze senzorů a výstupní odezvy,
- ladění a optimalizace algoritmů v různých prostředích
- spolupráce v týmu při návrhu, testování a vyhodnocování výsledků projektu,
- schopnost prezentace výsledků své práce v kontextu technické dokumentace či soutěžního projektu.

5.7 Dvoukolové balancující vozítko

Úloha dvoukolého balancujícího vozítka je praktickým příkladem úlohy, která se bez zpětnovazební regulace neobejde. Konstrukce robota bude založena na principu samovyrovnávacích zařízení, které nejvíce popularizovala společnost Segway[75)]. Cílem je udržet vozítko stabilní ve vertikální poloze motory, které budou upravovat náklon vozítka na základě dat z gyroskopického senzoru.

Z pohledu řízení se jedná o nestabilní systém, který bez perfektní regulace téměř okamžitě spadne. Tato úloha je tedy ideální pro praktické využití PID regulátoru. Studenti tak mohou vidět přímý dopad každé složky na chování robota.

Zásadní podmínkou pro úspěšné řešení úlohy je, aby studenti znali PID regulaci nejen po teoretické stránce, ale uměli ji aplikovat a najít správné hodnoty regulačních složek.

Po úspěšné stabilizaci robota ve vertikální lze úlohu rozšiřovat, čímž se zároveň zjistí, jak kvalitně je nastavený regulátor. První rozšíření, jízdu vpřed a vzad, by robot měl zvládnout vcelku snadno. Druhé rozšíření může být jízda ve tvaru čtverce a kruhu – rozšíření testuje nejen přímou jízdu, ale i natáčení robota. Následující rozšíření úlohy může být například autonomní jízda, sledování černé čáry či orientace v bludišti.

5.7.1 Rozvoj kompetencí

- Úloha podporuje rozvoj následujících kompetencí:
- Testovat a ladit programy, optimalizovat strategie pohybu
- Pracovat se senzory, analyzovat vstupní data a reagovat na jejich změnu
- Zodpovědně přistupovat k plnění úkolů – přesnost, robustnost a odolnost vůči chybám jsou klíčové faktory při tvorbě algoritmu.
- Implementovat logiku rozhodování na základě údajů ze senzorů.
- Navrhnout chování autonomního systému s ohledem na předvídatelnost
- Vyhodnocovat reakce robota a přizpůsobovat program aktuálním podmínkám

6 Závěr

Tato bakalářská práce se zaměřila na využití programovacího jazyka Rust pro řízení robotické stavebnice LEGO® Mindstorms EV3. Cílem bylo nejen ověřit technickou proveditelnost programování pomocí Rustu, ale i zhodnotit jeho přínos z hlediska výuky v rámci středního odborného vzdělávání.

V úvodních kapitolách byly popsány základní principy jazyka Rust, včetně instalace prostředí a způsobu práce se syntaxí a strukturou kódu. Následně byl popsán proces nasazení alternativního operačního systému ev3dev na řídicí jednotku EV3 a propojení s vývojovým prostředím. Tato část práce poskytla nezbytný teoretický základ pro praktickou část.

V praktické části byly vytvořeny ukázkové programy demonstrující ovládání jednotlivých robotických prvků stavebnice, jako jsou motory, senzory, obrazovka nebo zvukové výstupy. Na základě těchto programů byly sestaveny modelové úlohy, které mají za cíl rozvíjet algoritmické myšlení a technické dovednosti studentů.

Tyto úlohy byly ověřeny v praxi ve spolupráci přátel ze zájmového kroužku robotiky. Během ověřování se ukázalo, že úlohy jsou technicky proveditelné a reálně využitelné. Zároveň bylo zjištěno, že úspěšnost řešení závisí na úrovni znalostí jednotlivých studentů, což poukazuje na potřebu rozdílného přístupu při výuce a možnost úpravy zadání podle schopností konkrétní skupiny.

Na základě získaných zkušeností lze konstatovat, že cíle stanovené v úvodu práce byly naplněny – podařilo se ověřit technickou proveditelnost použití Rustu při programování stavebnice EV3 i jeho potenciál pro vzdělávací účely. V kombinaci s otevřeností platformy ev3dev se jedná o prostředí vhodné jak pro školní výuku, tak pro volnočasové aktivity. Do budoucna by bylo možné uvažovat o rozšíření navržených úloh o pokročilejší algoritmy nebo o vytvoření strukturovaného výukového materiálu dostupného širší komunitě pedagogů a studentů.Zároveň se nabízí i možnost rozšíření o ovládání dalších periferií, které nejsou standardně určeny pro řídicí jednotku EV3, či jednotku pomocí těchto periferií řídit.

Seznam použitých informačních zdrojů

1) LEGO Education. *LEGO MINDSTORMS Education EV3 Core Set (45544)* [online]. [cit. 2025-04-12]. Dostupné z: https://education.lego.com/en-us/products/lego-mindstorms-education-ev3-core-set/5003400/
2) How Rust went from a side project to the world's most-loved programming language. *MIT Technology Review* [online]. 2023, 2023-02-14 [cit. 2025-03-15]. Dostupné z: https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/
3) Mozilla releases Rust 0.1, the language that will eventually usurp Firefox's C++. *ExtremeTech* [online]. 2012, 2012-01-24 [cit. 2025-03-15]. Dostupné z: https://www.extremetech.com/internet/115207-mozilla-releases-rust-0-1-the-language-that-will-eventually-usurp-firefoxs-c
4) About. *Servo* [online]. [cit. 2025-03-15]. Dostupné z: https://servo.org/about/
5) Announcing Rust 1.0. *Rust Blog* [online]. 2015, 2015-03-15 [cit. 2025-03-15]. Dostupné z: 0 https://blog.rust-lang.org/2015/05/15/Rust-1.0.html
6) The Rust Foundation. *Rust Foundation* [online]. [cit. 2025-03-15]. Dostupné z: https://rustfoundation.org/
7) Bevy Engine. *Bevy* [online]. [cit. 2025-03-15]. Dostupné z: https://bevyengine.org/
8) Omarabid/rust-companies. *GitHub* [online]. 2025-04-05 [cit. 2025-04-14]. Dostupné z: https://github.com/omarabid/rust-companies
9) MACHADO, André. Rust in the Linux Kernel: Controversy and a Safer Future. In: *Substack* [online]. 2025 [cit. 2025-04-08]. Dostupné z: https://machaddr.substack.com/p/rust-in-the-linux-kernel-controversy
10) ARORA, Himanshu. Buffer Overflow Attack Explained with a C Program Example. In: *The Geek Stuff* [online]. 2013, 2013-06-04 [cit. 2025-04-08]. Dostupné z: https://www.thegeekstuff.com/2013/06/buffer-overflow/
11) Appendix: Glossary. In: *The Cargo Book* [online]. [cit. 2025-03-08]. Dostupné z: https://doc.rust-lang.org/cargo/appendix/glossary.html#package

12) C Programming For Beginners. In: *GeeksForGeeks* [online]. 2024-08-07 [cit. 2025-03-09]. Dostupné z: https://www.geeksforgeeks.org/c-programming-for-beginners-a-20-day-curriculum/

13) SHAH, Sharvin. The Ultimate Guide to Python: How to Go From Beginner to Pro. In: *FreeCodecamp* [online]. [cit. 2025-03-09]. Dostupné z: https://www.freecodecamp.org/news/the-ultimate-guide-to-python-from-beginner-to-intermediate-to-pro/

14) BLAHO, RNDr. Andej, PhD.; doc. RNDr. Ľubomír SALANCI a PhD.; Mgr. Václav ŠIMANDL, PH.D. Programování v jazyce Python pro střední školy. In: *Informatické myšlení* [online]. [cit. 2025-03-10]. Dostupné z: https://www.imysleni.cz/ucebnice/zaklady-programovani-v-jazyce-python-pro-stredni-skoly/

15) CORREIA, Danny. Fibonacci (Iterative). In: *Medium* [online]. 2019, 2019-10-07 [cit. 2025-03-10]. Dostupné z: https://medium.com/@danfcorreia/fibonacci-iterative-28b042a3eec

16) HANÁK, Drahomír. Lekce 14 - Eratosthenovo síto. In: *ITnetwork* [online]. [cit. 2025-03-10]. Dostupné z: https://www.itnetwork.cz/algoritmy/matematicke/algoritmus-eratosthenovo-sito

17) AMD Ryzen 5 3400G Specs. In: *TechPowerUp* [online]. [cit. 2025-03-13]. Dostupné z: https://www.techpowerup.com/cpu-specs/ryzen-5-3400g.c2204

18) Download Visual Studio Code. In: *Visual Studio Code* [online]. [cit. 2025-03-13]. Dostupné z: https://code.visualstudio.com/Download

19) Rust-analyzer. In: *Visual Studio Marketplace* [online]. [cit. 2025-03-13]. Dostupné z: https://marketplace.visualstudio.com/items?itemName=rust-lang.rust-analyzer

20) Co je to snake_case? In: *IT-Slovník* [online]. [cit. 2025-03-28]. Dostupné z: https://it-slovnik.cz/pojem/snake-case

21) MCKENZIE, Cameron. What is Kebab case? In: *TheServerSide* [online]. 2020 [cit. 2025-03-28]. Dostupné z: https://www.theserverside.com/definition/Kebab-case

22) Defining Modules to Control Scope and Privacy. In: *The Rust Programming Language* [online]. [cit. 2025-03-28]. Dostupné z: https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html

23) Macros. In: *The Rust Programming Language* [online]. [cit. 2025-03-30]. Dostupné z: https://doc.rust-lang.org/book/ch20-05-macros.html
24) Scope and Shadowing. In: *Rust By Example* [online]. [cit. 2025-03-25]. Dostupné z: https://doc.rust-lang.org/rust-by-example/variable_bindings/scope.html
25) Constants. In: *Rust By Example* [online]. [cit. 2025-03-26]. Dostupné z: https://doc.rust-lang.org/rust-by-example/custom_types/constants.html
26) Data Types. In: *The Rust Programming Language* [online]. [cit. 2025-03-24]. Dostupné z: https://doc.rust-lang.org/book/ch03-02-data-types.html
27) HEROUT, Pavel. Kódy Unicode a UTF-8. In: *Katedra informatiky a výpočetní techniky* [online]. 1999-06-14 [cit. 2025-03-25]. Dostupné z: https://www.kiv.zcu.cz/~herout/pruzkumy/unicode/unicode.html
28) Storing UTF-8 Encoded Text with Strings. In: *The Rust Programming Language* [online]. [cit. 2025-03-25]. Dostupné z: https://doc.rust-lang.org/book/ch08-02-strings.html#storing-utf-8-encoded-text-with-strings
29) C Comments. In: MICROSOFT. *Microsoft Learn* [online]. 2023-01-25 [cit. 2025-03-26]. Dostupné z: https://learn.microsoft.com/en-us/cpp/c-language/c-comments?view=msvc-170
30) What is rustdoc? In: *The rustdoc book* [online]. [cit. 2025-03-26]. Dostupné z: https://doc.rust-lang.org/rustdoc/index.html
31) HTML: HyperText Markup Language. In: MOZILLA CORPORATION. *MDN Web Doc* [online]. [cit. 2025-03-26]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTML
32) Control Flow. In: *The Rust Programming Language* [online]. [cit. 2025-04-01]. Dostupné z: https://doc.rust-lang.org/book/ch03-05-control-flow.html
33) Break. In: *The Rust Programming Language* [online]. [cit. 2025-04-02]. Dostupné z: https://doc.rust-lang.org/std/keyword.break.html
34) Getter and Setter in Java. In: TUTORIALS POINT. *Tutorials Point* [online]. [cit. 2025-04-03]. Dostupné z: https://www.tutorialspoint.com/getter-and-setter-in-java
35) Function sleep. In: *The Rust Programming Language* [online]. [cit. 2025-04-04]. Dostupné z: https://doc.rust-lang.org/std/thread/fn.sleep.html

36) Struct Duration. In: *Https://doc.rust-lang.org/std/time/struct.Duration.html* [online]. [cit. 2025-04-04]. Dostupné z: https://doc.rust-lang.org/std/time/struct.Duration.html

37) Std. In: *The Rust Programming Language* [online]. [cit. 2025-04-04]. Dostupné z: https://doc.rust-lang.org/std/index.htm

38) Projects. In: EV3DEV. *Ev3dev.org* [online]. [cit. 2025-03-20]. Dostupné z: https://www.ev3dev.org/projects/

39) MINDSTORMS Education EV3 Retirement FAQs. In: LEGO. *LEGO Education* [online]. 2021-03-02 [cit. 2025-03-12]. Dostupné z: https://community.legoeducation.com/blogs/36/95

40) O fakultě. In: *FEL ČVUT* [online]. [cit. 2025-03-12]. Dostupné z: https://fel.cvut.cz/cs/fakulta/o-fakulte

41) Stavebnice Brian. In: *Robosoutěž* [online]. [cit. 2025-03-12]. Dostupné z: https://robosoutez.fel.cvut.cz/stavebnice-brian

42) Programming Languages: Control ev3dev devices from code. In: *Ev3dev.org* [online]. [cit. 2025-03-12]. Dostupné z: https://www.ev3dev.org/docs/programming-languages/

43) ZIP: Control ev3dev devices from code. In: *File Format* [online]. [cit. 2025-03-08]. Dostupné z: https://docs.fileformat.com/compression/zip/

44) Releases: ev3dev/ev3dev. In: *GitHub* [online]. [cit. 2025-03-08]. Dostupné z: https://github.com/ev3dev/ev3dev/releases/

45) Getting Started with ev3dev: ev3dev/ev3dev. In: *Ev3dev.org* [online]. [cit. 2025-03-09]. Dostupné z: https://www.ev3dev.org/docs/getting-started/

46) BalenaEtcher: Flash OS images to SD cards & USB drives. In: BALENA. *BalenaEtcher* [online]. [cit. 2025-03-10]. Dostupné z: https://etcher.balena.io/

47) IMG: DIsk Image File Format. In: *File Format* [online]. [cit. 2025-03-11]. Dostupné z: https://docs.fileformat.com/disc-and-media/img/

48) YLONEN, T. The Secure Shell (SSH) Transport Layer Protocol. In: *RFC Editor* [online]. 2006 [cit. 2025-03-12]. Dostupné z: https://www.rfc-editor.org/rfc/rfc4253

49) Networking: Connecting ev3dev to the internet and other devices. In: *Ev3dev.org* [online]. [cit. 2025-03-12]. Dostupné z: https://www.ev3dev.org/docs/networking/

50) TL-WN725N: Bezdrátový nano USB adaptér N s rychlostí 150 Mbit/s. In: TP-LINK. *Tp-link* [online]. [cit. 2025-03-12]. Dostupné z: https://www.tp-link.com/cz/home-networking/adapter/tl-wn725n/

51) RTL8188EUS PDF Datasheet: IEEE 802.11b/g/n 1T1R WLAN. In: *DatasheetCafe* [online]. 2022, 2022-05-23 [cit. 2025-03-12]. Dostupné z: https://www.datasheetcafe.com/RTL8188EUS-pdf-24725/

52) TL-WN722N: Vysokovýkonný bezdrátový USB adaptér 150 Mbit/s. In: TP-LINK. *Tp-link* [online]. [cit. 2025-03-12]. Dostupné z: https://www.tp-link.com/cz/home-networking/adapter/tl-wn722n/

53) RFC 791: INTERNET PROTOCOL. In: DARPA. *RFC Editor* [online]. 1981 [cit. 2025-03-12]. Dostupné z: https://www.rfc-editor.org/rfc/rfc791

54) HARTINGER, David. Lekce 1: Úvod do operačního systému Linux. In: ITNETWORK. *ITnetwork* [online]. [cit. 2025-03-14]. Dostupné z: https://www.itnetwork.cz/linux/zaklady/uvod-do-operacniho-systemu-linux

55) KERRISK, Michael. Linux man pages online. In: *Man7* [online]. [cit. 2025-03-14]. Dostupné z: https://www.man7.org/linux/man-pages/

56) AM1808 Data sheet. In: TEXAS INSTRUMETNS. *Texas Instrumetns* [online]. [cit. 2025-03-18]. Dostupné z: https://www.ti.com/product/AM1808

57) WRIGHT, Gavin. What is x86-64? In: *TechTarget* [online]. [cit. 2025-03-19]. Dostupné z: https://www.techtarget.com/whatis/definition/x86-64

58) Difference Between Native Compiler and Cross Compiler. In: GEEKSFORGEEKS. *GeeksForGeeks* [online]. 2024-09-27 [cit. 2025-03-19]. Dostupné z: https://www.geeksforgeeks.org/difference-between-native-compiler-and-cross-compiler/

59) Ev3dev-lang-rust. In: *Crates.io* [online]. [cit. 2025-03-22]. Dostupné z: https://crates.io/crates/ev3dev-lang-rust/0.15.0

60) How to Securely Copy Files in Linux: scp Command. In: GEEKSFORGEEKS. *GeeksForGeeks* [online]. 2025-04-08 [cit. 2025-04-12]. Dostupné z: https://www.geeksforgeeks.org/scp-command-in-linux-with-examples/
61) Co je PowerShell?: scp Command. In: MICROSOFT. *Microsoft Learn* [online]. 2025-01-28 [cit. 2025-04-12]. Dostupné z: https://learn.microsoft.com/cs-cz/powershell/scripting/overview
62) LAUMANN, PHD, Felix. Text-to-Speech 101: The Ultimate Guide. In: *Medium* [online]. 2023-11-30 [cit. 2025-04-12]. Dostupné z: https://medium.com/neuralspace/text-to-speech-101-the-ultimate-guide-9a4b10e20fef
63) Vlastnosti zvuku: The Ultimate Guide. In: IKAP. *Elektronická učebnice* [online]. [cit. 2025-02-20]. Dostupné z: https://eluc.ikap.cz/lekce/vlastnosti-zvuku
64) LEGO MINDSTORMS EV3. In: DEV3DEV. *Ev3dev* [online]. [cit. 2025-02-20]. Dostupné z: https://docs.ev3dev.org/projects/lego-linux-drivers/en/ev3dev-stretch/ev3.html#buttons
65) pixix4. Ev3dev-lang-rust. In: *GitHub* [online]. [cit. 2025-04-09]. Dostupné z: https://github.com/pixix4/ev3dev-lang-rust/blob/master/examples/buttons.rs
66) pixix4. Ev3dev-lang-rust. In: *GitHub* [online]. [cit. 2025-04-10]. Dostupné z: https://github.com/pixix4/ev3dev-lang-rust/blob/master/src/screen.rs
67) BURFOOT, John. EV3 Sensors. In: *LEGO® Engineering* [online]. 2018, 2018-03-22 [cit. 2025-04-10]. Dostupné z: http://legoengineering.com/ev3-sensors/index.html
68) SESHAN, Sanjay a Arvind SESHAN. ADVANCED EV3 PROGRAMMING LESSON: Introduction to Gyro Sensor and Drift. In: *EV3Lessons* [online]. 2018, 2018-03-22 [cit. 2025-04-10]. Dostupné z: https://ev3lessons.com/en/ProgrammingLessons/advanced/GyroDrift.pdf
69) Rámcový vzdělávací program pro obor vzdělání: 18 – 20 – M/01 Informační technologie. In: MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY. *Jednotný metodický portál MŠMT* [online]. 2023-09-01 [cit. 2025-04-12]. Dostupné z: https://www.edu.cz/rvpsov/ciste/18-20-M01.pdf

70) PARKER, Dave. Line Follower. In: *NXT Programs* [online]. [cit. 2025-04-12]. Dostupné z: https://www.nxtprograms.com/line_follower/steps.html

71) PARKER, Dave. Line Follower: Designed for NXT 2.0. In: *NXT Programs* [online]. [cit. 2025-04-12]. Dostupné z: https://www.nxtprograms.com/NXT2/line_follower/steps.html

72) Color codes. In: *Ozobot* [online]. [cit. 2025-04-10]. Dostupné z: https://ozobot.com/create/color-codes/

73) Zadání soutěžní úlohy: Pathfinder. In: *Robosoutěž* [online]. [cit. 2025-04-12]. Dostupné z: https://robosoutez.fel.cvut.cz/zadani-soutezni-ulohy-pathfinder

74) SLUKA, J. A PID Controller For Lego Mindstorms Robots. In: *InPharmix* [online]. [cit. 2025-04-09]. Dostupné z: https://www.inpharmix.com/jps/PID_Controller_For_Lego_Mindstorms_Robots.html

ZIEGLER, J. G. a N. B. NICHOLS. Optimum Settings for Automatic Controllers. In: InPharmix [online]. 1942 [cit. 2025-04-09]. Dostupné z: https://web.archive.org/web/20170918055307/http://staff.guilan.ac.ir/staff/users/chaibakhsh/fckeditor_repo/file/documents/Optimum%20Settings%20for%20Automatic%20Controllers%20(Ziegler%20and%20Nichols,%201942).pdf

75) Optimum Settings for Automatic Controllers. In: SEGWAY. *Segway* [online]. [cit. 2025-04-10]. Dostupné z: https://store.segway.com/segway-ninebot-s-plus

Vyjádření k využití nástrojů umělé inteligence

Při tvorbě této práce bylo využito nástrojů umělé inteligence k opravě pravopisu a překladu cizojazyčných zdrojů.

Seznam obrázků

Obrázek 1: Struktura projektu[22] 19
Obrázek 2: Připojení pomocí protokolu SSH 43

Seznam tabulek

Tabulka 1: Časy výpočtu Fibonacciho čísel 14
Tabulka 2: Časy nalezení prvočísel za použití Eratosthenova síta 14
Tabulka 3: Typy celočíselného datového typu[26] 27
Tabulka 4: Číselné hodnoty reprezentující barvy[67] 56
Tabulka 5: Režimy barevného senzoru 56

Seznam grafů

Graf 1: Zobrazení časové náročnosti výpočtu Fibonacciho čísla, logaritmická osa Y....... 15
Graf 2: Zobrazení časové náročnosti nalezení prvočísla, logaritmická osa Y.................... 15
Graf 3: Porovnání rychlosti programu kompilovaného v režimech debug a release.......... 22