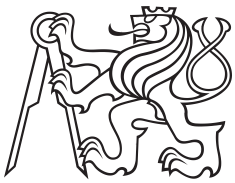
Diplomová práce



České
vysoké
učení technické
v Praze

F2

Fakulta strojní
Ústav přístojové a řídící techniky

Teach-in algoritmus pro PLC

Bc. Stanislav Linhart

Vedoucí: Mgr. Ing. Jakub Jura, Ph.D
Obor: Automatizační a přístrojová technika
Studijní program: Automatizace a průmyslová informatika
Srpen 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Linhart** Jméno: **Stanislav** Osobní číslo: **466567**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav přístrojové a řídící techniky**

Studijní program: **Automatizační a přístrojová technika**

Specializace: **Automatizace a průmyslová informatika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Teach-in algoritmus pro PLC

Název diplomové práce anglicky:

Teach-in algorithm for PLC

Pokyny pro vypracování:

1) Proveďte rešerši metod programování sekvenčních úloh
2) Prozkoumejte možnosti objektově orientovaného přístupu k programování v jazyce ST pro PLC
3) Prozkoumejte možnosti svobodných software a otevřených hw řešeních v průmyslových řídicích systémech
4) Vyberte vhodnou metodu/metody pro Teach-in funkci
5) Implementujte zvolenou metodu pro sekvenční řízení a v ní implementujte funkci Teach-in
6) Pro systém Teach-in vytvořte vhodnou vizualizaci

Seznam doporučené literatury:

[1] J. Jura and Martinásková. M., “Funkce teach-in implementována v plc,”Automatizace, vol. 48/9, pp. 522–525, 2005
[2] Mervis manuál: https://www.unipi.technology/cs/mervis-p56
[3] PLC Unipi https://www.unipi.technology/cs
[4] MySCADA https://www.myscada.org/cs/

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Mgr. Jakub Jura, Ph.D., U12110.3

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **30.04.2021** Termín odevzdání diplomové práce: **24.08.2021**

Platnost zadání diplomové práce: _____

Ing. Mgr. Jakub Jura, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Michael Valášek, DrSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Poděkování

Chtěl bych poděkovat vedoucímu práce doktoru Jakubu Jurovi za kvalitní vedení a rady při zpracování této práce.

Dále bych chtěl poděkovat své rodině a nejbližším přátelům, kteří mě podporovali při mém studiu.

V poslední řadě bych chtěl poděkovat za pět let kvalitního studia na Fakultě strojní ČVUT v Praze.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

V Praze, 20. srpna 2021

Abstrakt

Práce se zabývá integrací teach-in algoritmu pomocí objektově orientovaného programování v jazyce structured text (IEC 61131-3). Pro řízení používá programovatelný logický kontrolér UniPI Axon, který poskytuje otevřenou strukturu softwaru PLC.

V teoretické části práce jsou popsány různé způsoby sekvenčního řízení, dále jsou zde zmíněny možnosti objektově orientovaného programování v jazycích IEC 61131-3, zejména však jazyku ST. Práce se také zabývá otevřenou strukturou v programovatelných logických automatech a možnostmi, které nám takový automat poskytuje.

Klíčová slova: teach-in, učení, pneumatický manipulátor, OOP, ST, otevřený software, PLC, IEC 61131-3

Vedoucí: Mgr. Ing. Jakub Jura, Ph.D
12110 - Department of Instrumentation and Control Engineering,
Technická 4,
Praha 6

Abstract

The thesis deals with the integration of the teach-in algorithm using object-oriented programming in the structured text language (IEC 61131-3). For control, it uses a programmable logic controller UniPI Axon, which provides an open structure of PLC software.

The theoretical part of the thesis describes various methods of sequential control, there are also mentioned the possibilities of object-oriented programming in languages IEC 61131-3, but especially the language ST. The work also deals with the open structure in programmable logic controllers and the possibilities that such a controller provides us.

Keywords: teach-in, pneumatic manipulator, OOP, ST, open source, open structure, PLC, IEC 61131-3

Title translation: Teach-in algorithm for PLC

Obsah

Obrázky

Tabulky

Kapitola 1

Úvod

V této diplomové práci se zabývám implementací teach-in algoritmu pro učební pomůcku umístěnou v laboratoři programovatelných automatů. Jedná se o pneumatický manipulátor, který byl naposledy upravován v bakalářské práci z roku 2011, jejíž autorem byl Ing. Radek Orlita [5]. Jeho cílem práce bylo vytvořit dokumentaci robota a demonstrovat na něm možnosti práce s produkty od společnosti Siemens. Firma Siemens má pravděpodobně největší podíl na evropském trhu s programovatelnými logickými automaty (Programmable Logic Controller - PLC) a poskytuje širokou podporu pro výrobní podniky, proto jsou to nejen na středních strojírenských školách často jediná PLC, s kterými se žáci a studenti setkávají.

Je však také důležité studentům ukázat, že pro PLC nemusí být podmínkou, aby zařízení a vývojové prostředí byly od jednoho výrobce. Proto se tato práce zabývá otevřenými softwarovými platformami pro programovatelné logické automaty. To znamená, že výrobce automatu poskytuje nástroje, které zprostředkovávají přístup k hardwarovým zdrojům a tím umožňují být nezávislý na jedné softwarové platformě. Navíc tyto softwary jsou často tzv. open source, tedy autor poskytuje zdrojový kód s dokumentací a licenci umožňující používání a upravování a tím umožňuje další nezávislý vývoj. Dále těmito nástroji mohou být například ovladače a rozhraní pro ovládání vstupů a výstupů.

V prostředí softwarového vývoje obecně je již několik let standardem používání objektově orientovaného paradigmatu, který poskytuje přehlednost kódu a jeho snadnou rozšířitelnost. V průmyslovém standardu *IEC 61131-3* se však stále využívá programování zejména strukturované. Toto paradigma

má sice nižší požadavky na abstrakci, ale výrazně zvyšuje chybovost kódu. V případě některých jazyků pro PLC se ale objevují prvky objektového paradigmatu. Proto se tato práce těmito novými prvky a možnostmi jejich dalšího rozvoje zabývá.

V tomto duchu je vytvářen i kód pro učící se algoritmus: teach-in. Samotný teach-in algoritmus umožňuje pro zmíněný pneumatický manipulátor vytvořit libovolnou sekvenci na základě učení postupných pohybů uživatelem v manuálním režimu. Za tímto účelem je potřeba naprogramovat formální sekvenční automat tak, aby bylo možné měnit jeho konkrétní stavy v procesu učení. Z tohoto důvodu jsou v první části práce popsány různé metody řešení sekvenčních úloh.

Po volbě metody řešení sekvenční úlohy a návrhu teach-in algoritmu je implementováno takové řízení, které bude snadno rozšířitelné a přehledné. Poslední částí této diplomové práce je vytvoření grafického rozhraní pro ovládání pneumatického manipulátoru, které bude zajišťovat snadné řízení uživatelem systému.

Část I

Teoretická část práce

Kapitola 2

Sekvenční řízení

Logické řízení dělíme na kombinační a sekvenční. Kombinační řízení pracuje tak, že výstupy přímo reagují na vstupy bez ohledu na předchozí stav. Oproti tomu sekvenční řízení zohledňuje pořadí stavů. V automatizaci se převážně setkáváme se sekvenčním typem úloh, plyne to už z fyzikální podstaty systémů. [6]

Samotný algoritmus *teach-in*, kterým se ve své práci zabývám, je vlastně aplikovaní algoritmu sekvenční logické funkce takovým způsobem, že uživatel zařízení si tuto funkci sám nadefinuje bez potřeby znát vnitřní fungování zařízení a nutnosti znát programovací jazyk, v kterém je tento algoritmus napsaný. Algoritmus *teach-in* potom vytváří přechodové funkce a nastavuje výstupy hodnoty pro každý stav za nás.

2.1 Konečný automat

Konečný automat je systém, který se může vyskytovat pouze v konečném množství stavů. Automat reaguje změnou stavu na konečné množství vnějších vstupů. Konečný automat je deterministický, což znamená, že aktuální stav a vnější vstupy systému jednoznačně určují, jaký stav bude následující. [7]

Konečný automat bývá modelem reálného systému, například algoritmu, nebo elektrického zařízení. Do takového systému mohou vstupovat vnější

podněty a ovlivňovat ho. Tyto podněty nazýváme vstupní symboly, díky kterým se stane, že konečný automat přejde z jednoho stavu do následujícího. Následující stav může být ale i tím samým jako stav současný.

Popis konečného automatu se tedy skládá z konečné množiny stavů, konečné množiny symbolů a přechodových funkcí, které budou definované v každém stavu a pro všechny vstupní symboly. [8]

2.1.1 Definice konečného stavového automatu

Deterministický konečný automat je uspořádaná pětice

$$A = (Q, X, \delta, q_0, F), \tag{2.1}$$

kde

- Q je konečná neprázdná množina stavů,
- X je konečná neprázdná množina vstupních symbolů (abeceda)
- $\delta : Q \times X \to Q$ je přechodová funkce,
- $q_0 \in Q$ je počáteční stav a
- $F \subset Q$ je množina koncových stavů (cílová množina) [7]

2.2 Mealyho a Mooreův sekvenční stroj

Obecný konečný automat nebere v úvahu působení na okolní prostředí, tedy neobsahuje žádné výstupní signály. Pokud jsou tyto signály do definice konečného automatu doplněny, získáme sekvenční stroj. Oproti konečnému automatu není u sekvenčního stroje definován počáteční stav a množina koncových stavů. [7]

Mezi Mealyho a Moorovo automatem je rozdíl ve výstupní funkci takový, že v případě Maelyho automatu je výstupní symbol závislý pouze na aktuálním stavu. V případě Moorova automatu je výstupní symbol definován nejen stavem automatu, ale i množinou vstupních symbolů. Důsledkem toho je, že Mooreův sekvenční stroj bude mít vždy o jedničku vyšší délku výstupního řetězce než vstupního. [8]

2.2.1 Definice Mealyho sekvenčního stroje

Konečným Maelyho sekvenčním strojem nazýváme pětici

$$A = (Q, X, Y, \delta, \lambda), \tag{2.2}$$

kde

- Q je konečná neprázdná množina stavů,
- X je konečná neprázdná množina vstupních symbolů (abeceda)
- Y je konečná neprázdná množina výstupních symbolů,
- $\delta : Q \times X \to Q$ je přechodová funkce a
- $\lambda : Q \to Y$ je výstupní funkce. [7]

2.2.2 Definice Moorova sekvenčního stroje

Konečným Mooreovým sekvenčním strojem nazýváme pětici

$$A = (Q, X, Y, \delta, \mu), \tag{2.3}$$

kde

- Q je konečná neprázdná množina stavů,
- X je konečná neprázdná množina vstupních symbolů (abeceda)
- Y je konečná neprázdná množina výstupních symbolů,
- $\delta : Q \times X \to Q$ je přechodová funkce a
- $\mu : Q \times X \to Y$ je výstupní funkce. [7]

2.3 Taktovací řetězce

Taktovací řetězce jsou druh sekvenčního řízení a vychází převážně z reléového řízení. Pracují na principu samodržného zapojení relé. Ke spuštění dalšího

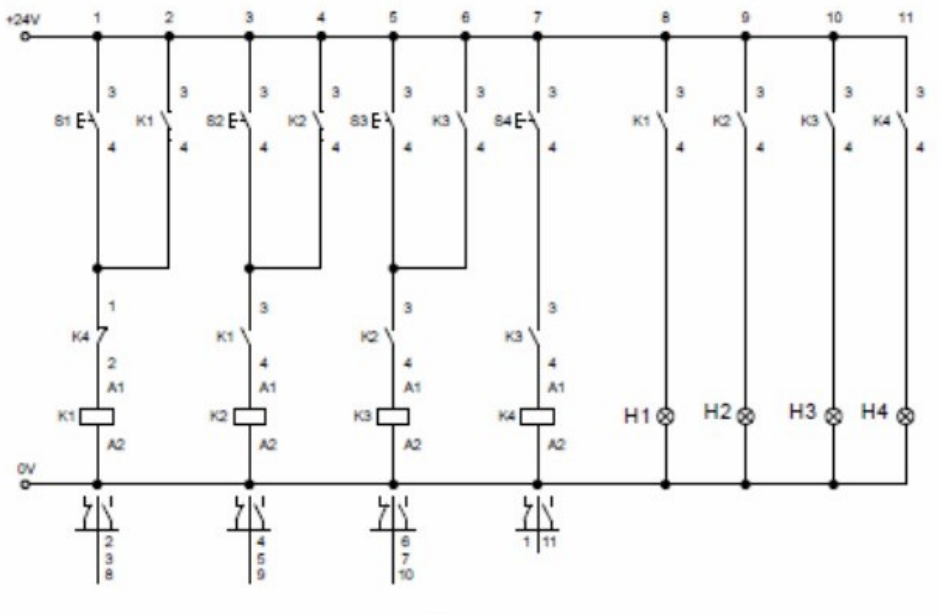
obvodu musí být splněna přechodová funkce a musí být aktivní poslední obvod v pořadí. Tedy předchozí obvod dává signál následujícímu, že je na řadě vyhodnotit přechodovou funkci. Pro názornost popisu taktovacích řetězců použiji v této podkapitole jejich nejčastější aplikaci a právě v reléovém řízení. To lze v současné době nalézt pouze u starších strojů, v nových aplikacích se díky dostupnosti *PLC* již nepoužívá. Taktovací řetězce se dají jednoduše aplikovat programovacím jazykem Ledder Diagram dle normy *IEC 61131-3* a tedy se dá konstatovat, že se v softwarové podobě používají i dnes.

Taktovací obvod pro ovládání akčních členů se dá sestavit pouze ze spínačů a relé. V případě elektropneumatických obvodů bude takovým akčním členem ventil řízený elektromagnetem. Taktovací řetězce se skládají ze dvou částí, z části řídicí a výkonové. V logické části se nachází samotný taktovací řetězec a ve výkonové části jsou asociace na akční členy. Známe dva typy taktovacích řetězců, stojící a mazající. Liší se tím, v jaké ze dvou částí obvodu se nacházejí logické podmínky pro sepnutí akčního členu. [1]

2.3.1 Stojící taktovací řetězec

Na obrázku 2.1 vidíte stojící taktovací řetězec aplikovaný v reléovém řízení. V levé části obvodu se nachází stojící taktovací řetězec a v pravé části výkonová část s akčními členy, kterými jsou v tomto případě žárovky.

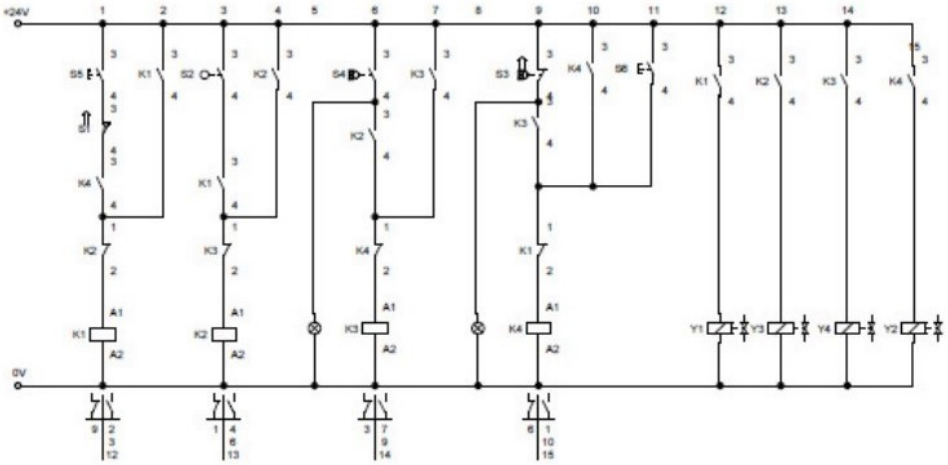
Stojící taktovací řetězec pracuje tak, že jednotlivá samodržně zapojená relé jsou aktivována v pořadí od prvního do posledního, kdy každé následující relé je aktivováno obvodem předchozího stavu po splnění přechodové funkce. Tedy nedochází k deaktivaci klopných obvodů během cyklu, všechna relé se deaktivují koncovou podmínkou, která deaktivuje první obvod a tím i všechny následující. [1]



Obrázek 2.1: Stojící taktovací řetězec [1]

2.3.2 Mazající taktovací řetězec

Mazající taktovací řetězec pracuje tím způsobem, že stejně jako u stojícího řetězce je následující řetězec aktivován po splnění přechodové podmínky a aktivním stavem předchozího klopného obvodu. Hlavním rozdílem je, že předchozí obvod je deaktivován právě aktivním klopným obvodem. Tedy jsou vždy aktivní maximálně dva obvody současně a to jen ve chvíli přechodu. Ale pokud taktovací řetězec vyčkává na splnění přechodové funkce, pak je aktivní vždy jeden. Jelikož první klopný obvod je závislý na aktivaci posledního obvodu, musí dojít k aktivaci mazajícího taktovacího řetězce pomocí další podmínky, aby byl obvod spuštěn. [1]



Obrázek 2.2: Mazající taktovací řetězec [1]

2.4 CASE-IF

Jedná se o sekvenční způsob řízení pracující tak, že máme konečné množství stavů, kde každý stav je rozlišen pomocí identifikátoru. Tento způsob řízení se snadno implementuje pomocí konstrukce CASE. Ta se implementuje pomocí rozdělení části programu na stavy a každý stav bude obsahovat námi definovaný kód, který chceme v daném stavu vykonávat. Volba stavu probíhá pomocí selektoru, který má uloženou hodnotu identifikátoru zvoleného stavu. Aktuální stav se bude cyklicky vykonávat, dokud nebude změněna hodnota selektoru. Ten bývá většinou celé číslo nebo textový řetězec.

Z jednoho stavu může přejít tento automat do libovolného stavu pomocí přechodové funkce. Přechodové funkce jsou implementovány pomocí konstrukce IF. Pokud dojde ke splnění logické podmínky, změní se selektor a tím i současný stav na libovolný další. Naprogramované pořadí stavů je závislé na hodnotě, která se v konstrukci IF do selektoru zapíše. CASE-IF tedy pracuje tak, že do každého stavu přidáváme konečné množství konstrukcí IF, které vyhodnocují přechodovou funkci, a je-li splněna, přepíšou hodnotu selektor, a tím změní stav automatu.

Kód 2.1: Příklad kódu CASE-IF v jazyce ST

```
1 FUNCTION_BLOCK
2     VAR
3         stav : INT;
4     END_VAR
5     CASE stav OF
6     0:
```

```
7           //kod aktualniho stavu
8           IF <Podminka prechodu> THEN
9               stav := 1;
10          END_IF;
11      1:
12          //kod aktualniho stavu
13          IF <Podminka prechodu> THEN
14              stav := 2;
15          END_IF;
16      2:
17          //kod aktualniho stavu
18          IF <Podminka prechodu> THEN
19              stav := 0;
20          END_IF;
21      END_CASE;
22  END_FUNCTION_BLOCk
```

Kapitola 3

Svobodný software pro programovatelné logické kontroléry

V této kapitole zmiňuji možnosti otevřeného softwaru v řízení průmyslových procesů programovatelnými logickými automaty (PLC). Automaty s otevřeným softwarem jsou novým zajímavým směrem pro průmysl nebo řízení provozu budov. Mohou poskytovat širší spektrum využití, zejména pak provozu více softwarových platforem najednou.

3.1 Open source software

Programy se nepíšou způsobem, jakým jim rozumí počítač, ale jsou psány programovacím jazykem, který zajišťuje abstrakci mezi instrukcemi stroje a chápáním člověka. Programovací jazyk může být textový nebo grafický, převážně se ale používá textový. Zdrojové soubory, které obsahují veškeré informace nutné pro kompilaci programu, se nazývají zdrojový kód. Program, který napíšeme ve formě zdrojového kódu a zkompilujeme do podoby strojového kódu, lze jen těžko dekompilovat zpět do původní podoby.

Jde tedy zejména o otázku, zda-li chceme aby naše práce byla volně šiřitelná v podobě zdrojového kódu, nebo nikoli. Pokud zvolíme možnost open source, neboli otevřeného softwaru, dáváme k dispozici náš zdrojový kód a tedy i naši práci k dalšímu použití. Neznamená to jen technickou schopnost získat zdrojový kód, ale také licenci určující práva, která uživatel kódu získá. V

některých definicích otevřeného zdroje jsou také připojeny konotace, jako jsou bezplatný software nebo software vyvinutý komunitou místo komerční společnosti. Tyto definice však nejsou správné, protože software s otevřeným zdrojovým kódem nemusí být bezplatný a také některé společnosti změnily politiku svého softwaru na software s otevřeným zdrojovým kódem. [4]

Mezi hlavní výhody patří ta, že od tvůrce softwaru dostáváme svolení si s jeho výtvorem nakládat tak, jak uznáme za vhodné. Tedy upravovat kód, opravovat chyby, použít jeho část a aplikovat ji pro jiný software. Jedna z dalších velkých výhod otevřeného softwaru vychází zejména z povahy, jak tento software převážně vzniká. Často je vyvíjen komunitou programátorů, která se stará o vývoj daného softwaru i bez nároku na mzdu. Díky tomu, že je volně k dispozici zdrojový kód a je kolem daného projektu komunita programátorů, dochází u takového softwaru k rychlejšímu odhalování chyb a jejich oprav. Z toho vyplývá také otázka bezpečnosti, která je díky rychlému odhalení chyb vyšší než u uzavřeného softwaru.

Nevýhodou je však to, že pokud je software poskytován společně se zdrojovým kódem, není takový software sám o sobě chráněný proti kopírování a tedy je nesnadné ho distribuovat za účelem zisku. Peníze na vývoj otevřeného softwaru často pocházejí z darů od uživatelů softwaru, ať už jednotlivců nebo firem. Také se používá zpoplatněná podpora jako služba, z které jdou peníze na vývoj softwaru. [4]

3.2 PLC jako otevřená platforma

Z předchozí podkapitoly vychází otevřená platforma, která spojuje vlastnosti open source softwaru a hardwaru. Výrobce hardwaru nám tak dává možnosti vytvářet vlastní software pro řízení daného zařízení, nebo použít jiný software třetích stran pro ovládání daného zařízení. Někdy dokonce poskytuje výrobce elektrická schémata svého zařízení.

Z hlediska programovatelných logických kontroléru je tato praxe poměrně nová. Konvenční PLC poskytuje výrobce většinou se softwarem k programování, který umožňuje programovat PLC pomocí normy *IEC 61131-3*. Typickým příkladem je firma Siemens, která je dominantní na evropském trhu výrobců programovatelných logických automatů. Jejich systémy jsou oblíbené zejména tím, že za nimi stojí podpora globální korporace. Firma Siemens má několik produktových řad PLC a k nim je uživatel nucen si pořídit softwarovou licenci pro vývojové prostředí. Uživatel nemá právo zvolit

si runtime podle svého uvážení a je tedy odkázán pouze na výrobce. [9]

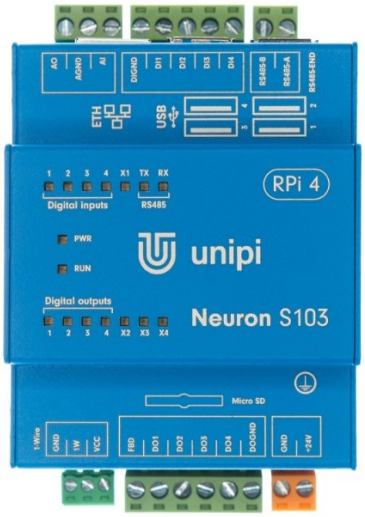
Uvnitř programovatelných logických automatů, které fungují jako otevřená platforma, najdeme operační systém Linux, do kterého se můžeme přihlásit pomocí SSH (Secure Shell - Zabezpečený vzdálený příkazový řádek). To umožňuje instalovat libovolný software jako runtime třetích stran, SCADA (Supervisory Control And Data Acquisition - Software pro operátorské grafické rozhraní v průmyslu) servery, či databáze pro ukládání dat. Díky tomu lze taková zařízení považovat jako univerzální hardware, který si může uživatel následně specializovat pro řízení inteligentního domu, průmyslového procesu, nebo samostatného stroje. [9]

Dalé zde vzniká možnost vyvinout vlastní software za pomoci libovolného programovacího jazyka. To nám dává velkou výhodu oproti jazykům dle *IEC 61131-3*, a to velké množství knihoven a již hotových softwarů, které můžeme použít od veřejné komunity. Mohlo by také dojít ke vzniku frameworků, které by práci usnadnily a byly by zaměřené na programování PLC. [9]

3.3 Zařízení UniPi

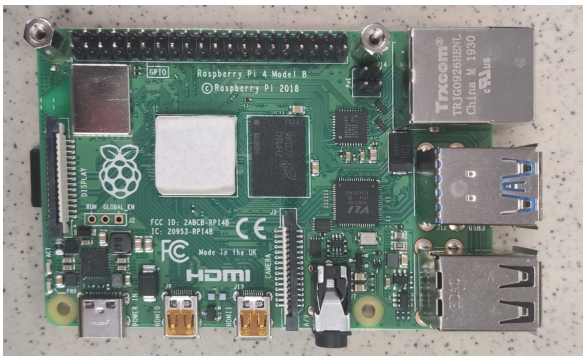
Společnost Faster CZ s.r.o., která vyvíjí logické kontroléry Unipi, používá politiku otevřené platformy. Jednotky v sobě obsahují procesory řady STM32, které ovládají vstupy a výstupy kontroléru, a hlavní SBC (Single Board Computer - Jednodeskový počítač), který s nimi komunikuje po sběrnici. V současnosti společnost nabízí tři řady kontrolérů a to Patron, Axon a Neuron. V případě Neuronu, je uvnitř použitý SBC Raspberry Pi. V ostatních případech jde o jiné SBC moduly. Všechny ale pracují s operačním systémem Linux založeném na Debian. To oproti konvenčním PLC umožňuje větší spektrum využití. [9]

3.3.1 Produktová řada Neuron



Obrázek 3.1: Neuron S103 (Raspberry Pi 4) [2]

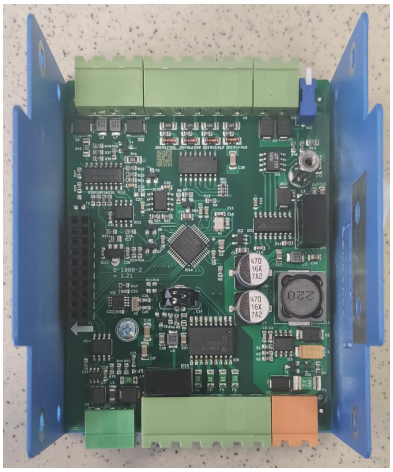
Jedná se o řadu programovatelných logických kontrolérů určených pro měření a regulaci, automatizaci, či monitoring. Vyznačují se tím, že pracují s jednodeskovým počítačem Raspberry Pi. Používají převážně jeho starší verzi 3B+, ale u některých modelů má možnost si zákazník zvolit současnou verzi 4B. [10]



Obrázek 3.2: SBC Raspaberry Pi 4B

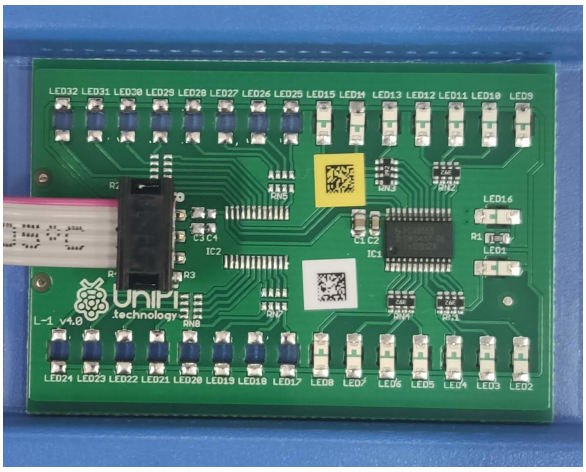
Na obrázku 3.2 je vyfocen počítač Raspberry pi 4B demontovaný přímo z Unipi Neuron. Tento počítač disponuje procesorem Broadcom BCM2711, jedná se o čtyř jádrový 64-bitový procesor typu ARMv8. Procesory Arm jsou využívány pro svou nízkou spotřebu energie zejména v mobilních zařízeních. Patří do skupiny RISC (Reduced Instuction Set Computer - Redukovaná instrukční sada). Znamená to, že je menší počet instrukcí procesoru, protože procesor má mikroinstrukce hardwarově implementovány. Tedy fungování

softwaru vyžaduje méně instrukcí a je tedy výkonnější než u procesorů typu CISC (Complex Instruction Set Computer - Komplexní instrukční sada). Dále tento SBC disponuje gigabitovým Ethernet portem, čtyřmi porty USB (2x USB 3.0 a 2x USB 2.0), dvěma micro-HDMI porty a dalšími konektory. Dále se zde nachází 40-pinový GPIO konektor, který je podstatný z hlediska fungování produktové řady Neuron, protože za pomocí tohoto konektoru komunikuje s procesory na modulech vstupů a výstupů. [10]



Obrázek 3.3: Modul pro vstupy a komunikační sběrnice uvnitř Unipi Neuron

Uprostřed desky na obrázku 3.3 vidíte mikrokontrolér STM32, který rovněž patří do skupiny ARM, vyrábí je společnost STMicroelectronics. Tento modul neslouží jen jako vstupní a výstupní zařízení, ale hlavně se na něm nachází napájecí obvod celého programovatelného logického kontroléru. Díky těmto modulům, může Unipi vytvářet různé kombinace vstupů a výstupů. [10]

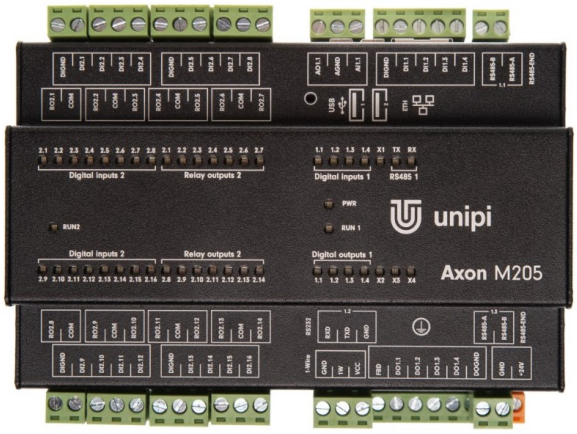


Obrázek 3.4: Deska s indikačními LED diodami rozhraní

Kromě modulů pro vstupy a výstupy se na vrchní straně šasi nachází deska,

kterou vidíte na obrázku 3.4, s indikačními LED diodami. [10]

3.3.2 Produktová řada Axon

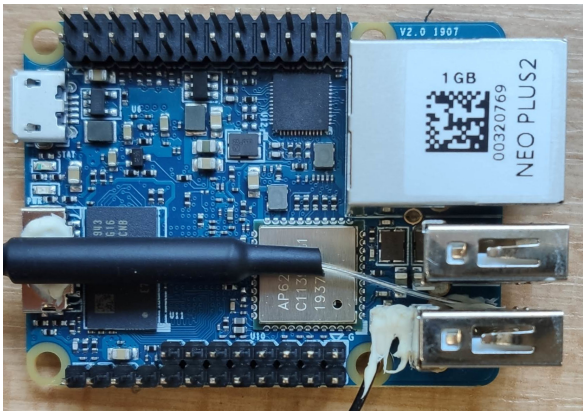


Obrázek 3.5: Unipi Axon M205 [3]

Na rozdíl od řady Neuron, je řada Axon řízena pomocí konkurenčního SBC NanoPi NEO Plus2, které vidíte na obrázku 3.6. Tento modul je výrazně levnější než konkurenční Raspberry Pi, a také je mnohem menší. Disponuje procesorem Allwinner H5 typu ARM s jádrem Cortex-A53, což je stejné jádro jako u procesorů Broadcom BCM2837B0, které jsou použity v jednodeskovém počítači Raspberry Pi 3B+. Má rovněž stejné množství paměti RAM (Random Access Memory) a to 1 GB. Oproti Raspberry Pi, pro které potřebujete externí paměťovou kartu, používá NanoPI Neo Plus2 paměť typu eMMC (embedded Multi Media Card - Vestavěná paměťová karta). Na modulu je také slot pro externí paměťovou kartu. Krom jiného řídícího modulu používá obdobný systém modulárních desek vstupů a výstupů jako u produktové řady Neuron.

3.4 Mervis

Při zakoupení kteréhokoli PLC UniPi dostáváte automaticky licenci k runtime systému Mervis. Ten pracuje na API vrstvě PLC. Jedná se systém vyvíjený společností ENERGOCENTRUM PLUS s.r.o. Je to komplexní řešení určené primárně do průmyslu, který používa standard IEC 61131-3. Podporuje dva programovací jazyky. Prvním je jednodušší jazyk FBD (function block



Obrázek 3.6: Řídící modul uvnitř produktové řady Axon

diagram) také nazývaný FUPLA, jedná se o grafický programovací jazyk, který uživatelům systému bez zkušenosti programování umožňuje snazší používání a redukci chyb. [11]

3.5 Alternativní způsoby řízení poskytované pro zařízení Unipi

Na rozdíl od konvenčních výrobců programovatelných logických kontrolérů nabízí UniPi cesty, jak vyvíjet vlastní software pro jejich PLC a nebo jej používat tak, že algoritmy řízení vstupů a výstupů pracují na jiném zařízení, a tak řídí vstupy a výstupy vzdáleně.

3.5.1 Ovladač typu SysFS

Tento typ ovladače je využíván v systémech Linux od verze jádra 2.6. Jelikož Linux striktně odděluje paměť uživatelskou a jádra kernelu, je zde prostředník v podobně objektů zvaných *kobjects*. Z hlediska programování se jedná o strukturu složek a souborů exportovanou kernelem, s kterými programátor pracuje jako se standardními soubory. [12] těchto souborech jsou zapsány informace o vstupech a výstupech UniPi PLC. Nejsou to soubory v pravém slova smyslu, jen prohlížeč souborů operačního systému je tak reprezentuje. Díky tomu, že nám UniPi tento ovladač poskytuje a dokumentaci k němu, je možné vytvářet vlastní runtime pro řízení vstupů a výstupů PLC a stává se z něj otevřené zařízení, s kterým jsme jako uživatelé zařízení zcela nezávislí

na softwaru. Díky tomu můžeme vytvářet vlastní software nebo používat software třetích stran.

3.5.2 Webové API

Další možnost ovládání vstupů a výstupu, kterou UniPi poskytuje,+ je vzdálené řízení pomocí http protokolu. Jelikož UniPi poskytuje ke svému zařízení REST API (Reprtesentational State Transfer - způsob čtení, editace a mazaní daz ze serveru, Application Programming Interface - rozhraní pro programování aplikací), dává tím možnost doslova vytvářet internet věcí, kdy vstupy a výstupy zařízení jsou čteny a řízeny pomocí HTML příkazů. Díky této architektuře *klient-server* je možné provozovat řídící software na libovolném hardwaru v síti. Podporuje REST, Bulk JSON, REST JSON, SOAP, WebSocket a JSON-RPC. To umožňuje implementovat kontroléry do jiné aplikace nezávisle na tom, jestli je aplikace na stejném hardwaru. [13]

Řízení průmyslového procesu z jiného serveru pomocí webového API ve smyslu, že každý krok procesu je řízen na jiném hardwaru než na PLC, není vhodný z hlediska bezpečnosti řízeného stroje. Webové API ale nemusí být používáno jen tímto způsobem. Pokud by software běžící na stejném hardwaru používal API ke komunikaci se vstupy a výstupy, jednalo by se o zajímavé řešení se snadnou implementací do softwarů třetích stran nebo námi napsaných. Webové API je také zajímavé řešení pro komunikaci SCADA systémů s PLC. [13]

REST API

REST API je v současné době velmi oblíbenou architekturou rozhraní, které pro svou práci používá protokol HTTP. Je orientovaný na práci s daty a s těmi pracuje pomocí čtyř základních metod HTTP protokolu, kterými jsou GET, POST, PUT a DELETE. [14]

GET je metoda používána k získávání dat ze serveru. Nelze pomocí ní data měnit, ale pouze číst. Metoda GET primárně získává data z adresního řádku.[14]

POST je druhou nejpoužívanější metodou, která slouží zejména k ukládání dat na server například data formuláře. Na rozdíl od metody GET jsou data

předávána v obsahu zprávy nikoli v adresním řádku.[14]

PUT je metoda sloužící k aktualizaci dat konkrétní entity.[14]

DELETE je metoda k odstranění entity z báze dat.[14]

Kapitola 4

Objektově orientované programování u programovatelných logických kontrolérů

Objektově orientované programování je již několik let současným standardem v programování většiny softwarů. Mezi tři nejpopulárnější programovací jazyky s současné době dle [15] patří jazyk Python, Java a JavaScript. Jazyk Java je z těchto tří jazyků jediným čistě objektově orientovaným jazykem. Jazyk Python a JavaScript jsou objektově orientované skriptovací jazyky.

OOP nabízí rozšířené možnosti použití PLC a snadnější znovupoužitelnost softwaru díky předem připraveným balíkům knihoven, které by mohl poskytovatel vývojového prostředí nabízet. OOP by tak mohlo přinést do programování PLC větší efektivitu a přehlednost.

4.1 Od strojového kódu k objektům

Procesor je velká sada logických obvodů složených zejména z tranzistorů. Skládá se z aritmeticko-logické jednotky (ALU), registrů, dekodérů instrukcí a dalších částí. Dekodér instrukcí provádí volbu logických obvodů ALU, neboli řídí činnost procesoru, tedy volí operace jako sčítání, násobení, dělení, logický součin, součet, negace a jiné. Tyto operace dekodér rozlišuje pomocí instrukcí. [16]

Programování ve strojovém kódu znamená psát přímo binární kód bez

jakékoliv abstrakce takovým způsobem, jakému procesor bezprostředně rozumí. To čemu rozumí stroj, je ale pro člověka špatně pochopitelné. Velmi rychle se objevil první způsob, jak nahradit strojový kód prvním programovacím jazykem. [17]

4.1.1 Nestrukturované paradigma

Pro alespoň nejnižší stupeň abstrakce bylo nutné vytvořit první programovací jazyk, díky kterému bylo možné nahradit špatně čitelný strojový kód. Proto vznikl jazyk symbolických adres známý jako Assembler. Zjednodušeně jde o jednoduchý překlad slov na binární čísla s velmi jednoduchou gramatikou. Tímto způsobem programování je však velmi neefektivní vytvářet rozsáhlejší programy, protože je zde náchylnost k chybám. Program je procesorem vykonáván v tom pořadí, jak jsou instrukce zapsány. Jsou zde i příkazy pro skok na jiné místo v paměti programu, ale toto paradigma postrádá strukturu a znovupoužitelnost takového kódu je velmi nízká až prakticky žádná. Určitou podobnost k Assembleru nalezneme i v programovacím jazyk Instruction List dle normy *IEC 61131-3*. [17]

Kód 4.1: Příklad programu v jazyku Assembler pro blikání LED pro procesor 8051 [18]

```
ORG 00H       ; Assembly Starts from 0000H.
; Main Program
START:  MOV P1, #0XFF   ; Move 11111111 to PORT1.
    CALL WAIT   ; Call WAIT
    MOV A, P1   ; Move P1 value to ACC
    CPL A     ; Complement ACC
    MOV P1, A   ; Move ACC value to P1
    CALL WAIT   ; Call WAIT
    SJMP START  ; Jump to START
WAIT:   MOV R2, #10   ; Load Register R2 with 10 (0x0A)
WAIT1:  MOV R3, #200    ; Load Register R3 with 10 (0xC8)
WAIT2:  MOV R4, #200    ; Load Register R4 with 10 (0xC8)
    DJNZ R4, $  ; Decrement R4 till it is 0. Stay there if not
    0.
    DJNZ R3, WAIT2  ; Decrement R3 till it is 0. Jump to WAIT2
    if not 0.
    DJNZ R2, WAIT1  ; Decrement R2 till it is 0. Jump to WAIT1
    if not 0.
    RET     ; Retu rn to Main Program
END         ; End Assembly
```

Kód 4.2: Příklad programu v jazyku instruction list dle normy IEC 61131-3 [19]

```
LD     TRUE          (*load TRUE in the accumulator*)
ANDN   BOOL1         (*execute AND with the negated value of the
     BOOL1 variable*)
JMPC   label         (*if the result was TRUE, then jump to the
    label "label"*)
```

```
4 LDN     BOOL2          (*save the negated value of *)
5 ST      ERG            (*BOOL2 in ERG*)
6 label:
7
8 LD      BOOL2          (*save the value of *)
9 ST      ERG            (*BOOL2 in ERG*)
```

4.1.2 Strukturované programování

Další stupeň abstrakce je vložení programu do celků, které bude možné znovu používat a dát programu strukturu. U jazyků jako byl Fortran se používalo nejdříve rozdělení programu na procedury a funkce. Procedury byly podprogramy, které pouze provedly požadovanou činnost, a funkce sloužily ke spočítání konkrétní hodnoty. Důvodem k tomuto rozdělení bylo, že návrat z funkce zpět do programu byl pomalejší, jelikož bylo potřeba řešit předání návratové hodnoty. [20]

S příchodem jazyku C se jeho autoři rozhodli, že i procedura se dá považovat za funkci, která nevrací žádnou hodnotu. K rozdělení programu používáme funkcionální dekompozici, tedy program rozdělíme na podproblémy a pro každý podproblém vytvoříme funkci, kterou budeme v programu volat. Jde tím docílit určité znovupoužitelnosti funkcí, ale není možné funkce modifikovat. Je možné ovlivňovat chování funkcí pomocí parametrů, to je ale s jejich velkým množstvím nepřehledné. Další možností úpravy chování programu jsou globální proměnné. U těch je ale velké riziko toho, že k nim v celém programu mají všechny funkce přístup a hrozilo by, že si programátor neuvědomí následky přepsání proměnné v jiné funkci. Taková chyba se velmi špatně dohledává. Typickými příklady strukturovaných programovacích jazyků jsou jazyk C/C++ a jazyk Pascal, na jehož základě se vytvořil jazyk Structured text pro textové strukturované programování PLC. [20, 17]

Kód 4.3: Příklad strukturovaného programování v jazyce ST

```
1  FUNCTION_BLOCK vstupy_do_pole
2    VAR_OUTPUT
3      pole_vstupu : ARRAY [0..7] OF BOOL;(* add out variables here
       *)
4    END_VAR
5  (*function block body*)
6    pole_vstupu[0] := hw.controler_robot_vzadu;
7    pole_vstupu[1] := hw.controler_robot_vpredu;
8    pole_vstupu[2] := hw.controler_robot_dole;
9    pole_vstupu[3] := hw.controler_robot_nahore;
10   pole_vstupu[4] := hw.controler_ruka_zasunuta;
11   pole_vstupu[5] := hw.controler_ruka_vysunuta;
12   pole_vstupu[6] := hw.controler_ruka_svisle;
13   pole_vstupu[7] := hw.controler_ruka_vodorovne;
```

```
14 END_FUNCTION_BLOCK
```

4.1.3 Objektově orientované programování

Jedná se o současné paradigma používané v moderních vyšších programovacích jazycích. Jeho zásadní rozdíl oproti strukturovanému programování je, že zde nerozdělujeme program na podproblémy, ale dělíme jej do celků, které se dají přenést do objektů reálného světa. Zjednodušeně se dá říct, že v strukturovaném programování základní jednotky programů reprezentují činnosti, tedy slovesa, ale v objektově orientovaném programování jsou základní jednotkou objekty, které budou pojmenovány podstatnými jmény. Vlastní činnosti, které nazýváme metodami, budou mít tyto objekty definované až uvnitř sebe.

Objekty kromě metod obsahují atributy, které uchovávají vlastností objektů. V OOP neexistují žádné globální proměnné, které by nepatřily ke konkrétnímu objektu. Objekty spolu totiž nekomunikují pomocí globálních proměnných, ale předávají si informace formou dotazovaní. [17]

Při programování nepředepisujeme jednotlivé objekty, nýbrž návody na jejich výrobu, které se nazývají třídy. Třídy jsou předpisy objektů, v kterých deklarovat, jaké atributy bude mít daný objekt a definujeme jeho metody. Objektově orientované programování má tři základní pilíře, a to zapouzdření, dědičnost a polymorfismus. [17]

Zapouzdření

Zapouzdřeni znamená, že vnitřní fungování objektu nemusí být pro jeho použití známo. Pro práci s objekty používáme rozhraní objektu, které nám jeho tvůrce připravil. Jedna z hlavních výhod je, že se tak předchází chybám a na programu může snadno pracovat více lidí. Z toho také vyplývá, že uvnitř objektu se nacházejí metody, které nejsou k dispozici pro uživatele objektu, ale pouze pro objekt samotný. [17]

Dědičnost

Dědění mezi třídami znamená, že jedna třída může získat metody a atributy z jiné třídy. To výrazně napomáhá k znovupoužitelnosti kódu. To umožňuje vytvářet nové struktury na základě předchozích. Příkladem může být třída nazvaná zaměstnanec, která bude definovat společné atributy a metody pro potomky, kteří budou rozšiřovat tuto třídu o další vlastnosti. Například třída *manager* bude dědit třídu zaměstnanec stejně tak jí bude dědit třída *skladník*. Objekt třídy *manager* bude mít jiné metody než objekt třídy *skladník*. V předkovi těchto tříd budou tedy definované vlastnosti jako jméno a příjmení pro všechny potomky této třídy a potomci budou rozšiřovat metody a atributy, které jsou specifické pro danou třídu. Dále se dědičnost dá použít tak, že již hotovou třídu, kterou získáme například z knihovny, rozšíříme o vlastní metody a atributy. Dědičnost můžeme použít i u konstrukce interface.[17]

Polymorfismus

Polymorfismus je třetím pilířem objektově orientovaného programování. Dává nám možnost přepisovat chování metod. Příkladem může být interface *Útvar*, který obsahuje metodu *obsah*. Potomci tříd, které implementují tento interface, poté definují chování metody *obsah*, tedy například třída *Čtverec* a třída *Kruh* budou obě implementovat metodu *obsah*, ale objekty třídy *Čtverec* a *Kruh* budou každý provádět různý výpočet. [17]

4.2 Objektově orientované programování v jazycích pro PLC

Structured Text je jazyk standardu *IEC 61131-3*, který, jak už název sám napovídá, používá strukturované paradigma. Byl poprvé přidán do této normy aktualizací v roce 1993 a jedná se o hojně využívaný textový jazyk při programování programovatelných logických automatů. Ve třetí aktualizaci jazyka v roce 2012 bylo provedeno několik úprav ve směru objektově orientované filozofie programování.

Objektově orientované programování v jazyku ST má několik výhod oproti strukturovanému. Hlavní výhodou je přehlednost kódu, která je díky řízení

funkčních bloků pomocí volání metod mnohem lepší. Oproti tomu ve strukturovaném programování vzniká takzvaný špagetový kód. Díky této přehlednosti a zapouzdření jednotlivých funkčních bloků (tříd) je program velmi snadno rozšířitelný. [21]

4.2.1 Funkční bloky v režimu OOP jazyku ST

Třída je základní stavební jednotkou při objektově orientovaném programování. Objekty reálného světa dokáže člověk zařadit dle vlastností a činností do určité skupiny, tyto skupiny nazýváme třídy. Pokud si objektově orientované programování pro pochopení přiblížíme reálnému světu jako továrnu na automobily, tak objekt je konkrétní automobil, který si zákazník vybral, a třída je návrh onoho modelu automobilu. V jazyku ST jsou nahrazeny třídy funkčními bloky. [20]

V jazyku ST mají funkční bloky dva režimy. V standardním strukturovaném režimu funkčního bloku probíhá ovládání z vnějšku pomocí vstupních proměnných a ty přímo vstupují do algoritmu daného funkčního bloku. Poté nám blok vrátí výstupní proměnné. Jedná se o jakýsi mezistupeň v strukturovaném a objektovém programování, protože v čistě strukturovaném programování nemáme možnost vytvářet vlastní datové typy. Ale v jazyku ST se dá definice funkčního bloku označit za uživatelem vytvořený datový typ. Oproti třídám, které se dají také nazvat datovým typem, v tomto režimu funkční blok neobsahuje metody.

V případě objektového režimu má funkční blok pouze metody, které jiné funkční bloky mohou z vnějšku volat. Tedy se neřídí chování instance funkčního bloku pomocí vstupních proměnných, ale pomocí volání metod, které však mohou vstupní proměnné požadovat.[22]

4.2.2 Deklarace atributů

Atributy jsou vlastnosti, které v sobě třída nebo funkční blok uchovává. V případě jazyka Java se atributy definují v těle třídy. Mohou to být buď datové typy primitivní nebo objektové. V případě Javy nám vývojářský balíček (JDK) poskytuje již předem připravené třídy například pro pole nebo řetězce.

Kód 4.4: Příklad deklarace atributů v jazyku JAVA [23]

```
1 public class Main {
```

```
  int x = 5;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

Jelikož je jazyk ST založen na jazyku Pascal, je deklarace proměnných umístěna vždy v konstrukci s párovými klíčovými slovy VAR a END_VAR. Samozřejmě jsou zde i další typy párových znaků pro deklaraci proměnných, ale ty jsou primárně určeny pro strukturovaný režim použití funkčních bloků. Párový znak VAR_INPUT a END_VAR se ale používá také v objektovém režimu pro deklaraci vstupních proměnných metod.

Kód 4.5: Příklad deklarace atributů v jazyku ST

```
FUNCTION_BLOCK Sequence
  VAR
    transferMatrix : ARRAY [0..100,0..8] OF BOOL;
    outputMatrix : ARRAY [0..101,0..6] OF BOOL;
    enSensorMatrix : ARRAY [0..100] OF BOOL;
    waitingTimeMatrix : ARRAY [0..100] OF TIME;
    lastState : int;
  END_VAR
END_FUNCTION_BLOCK
```

4.2.3 Metody

V objektově orientovaném programování klademe důraz na zapouzdření třídy. To znamená, že uživatel třídy nepotřebuje znát vnitřní fungování objektu, ale pracuje jen s přidělenými metodami, kterým se říká rozhraní třídy. Požívání metod místo vstupních proměnných pro vyvolání nebo změnu chování je mnohem přehlednější a také se zvyšuje efektivita. V jazyku ST se metoda definuje pomocí párového klíčového slova METHOD a END_METHOD. Pokud chceme, aby metoda vracela nějakou hodnotu, nejdříve napíšeme klíčové slovo METHOD, dále název metody a poté dvojtečku, za kterou napíšeme typ návratové proměnné. Návratovou proměnnou vracíme přiřazením hodnoty, kdy na levé straně napíšeme název metody, viz příklad kódu 4.6.

Kód 4.6: Příklad metody s návratovou hodnotou TIME v jazyku ST

```
  METHOD getTime : TIME
      VAR_INPUT
        step : int;
        sequenceID : Int;
      END_VAR
```

```
6       getTime := sequenceArray[sequenceID].getWaitingTime(step
    := step);
7   END_METHOD
```

Hlavní metoda a blok programu

V objektově orientovaném jazyce vždy musí existovat určitá část kódu, která se spustí jako první. Bez toho by překladač jazyka nevěděl, kde má program začít. V případě čistě objektově orientovaného jazyka Java je toto zajištěno pomocí metody *Main*, jak vidíte v příkladu kódu 4.7. [20]

Kód 4.7: Příklad hlavní metody v Javě [23]

```
1 public class Main {
2   public static void main(String[] args) {
3     System.out.println("Hello World");
4   }
5 }
```

Oproti tomu jazyk ST hlavní metodu nahrazuje blokem *Program*, v kterém se žádná metoda nacházet nesmí. Jazyk ST totiž umožňuje primárně režim strukturovaný a objektová struktura je spíše rozšířením, proto je v tomto případě podobný skriptovacím jazykům, jak je vidět v ukázce kódu 4.8. [24]

Kód 4.8: Příklad bloku programu v jazyku ST [24]

```
1  PROGRAM main
2      VAR
3          counter1 : CCounter;
4          counter2 : CCounter := (m_iUpperLimit := 15000,
       m_iLowerLimit := -10);
5          current_value : int;
6      END_VAR
7
8      counter1.SetDirection(bCountUp := true);
9
10     counter1.Count();
11
12     current_value := counter1.GetCount();
13
14     counter2.Count();
15 END_PROGRAM
```

4.2.4 Dědičnost v jazyku ST

Dědičnost je základní vlastnost objektově orientovaného programování. Díky této základní vlastnosti je u programovacích jazyků možné snadno rozšiřovat již vytvořené třídy z knihoven nebo tvořit stromovou strukturu. Dědičnost funkčních bloků se v jazyku ST provádí pomocí příkazu *EXTENDS*, jak vidíte na řádku 11 v příkladu 4.9. Dědění podporují i konstrukce rozhraní.[24]

Virtuální metody

Tento typ metod se v jiných jazycích nazývá abstraktní, ale jejich funkce je stejná. Používá se, když voláme stejnou metodu, ale chceme, aby se chování lišilo na základě potomka třídy. Například voláme funkci objem, ale výpočet se liší na základě toho, jestli je potomek koule anebo je krychle. Vytváří se tak, že v předkovi funkčního bloku definujeme metodu a v potomkovi vytvoříme metodu se stejným názvem, ale přidáme do ní algoritmus. Obě metody musí mít ve své definici klíčové slovo *VIRTUAL*. Metoda v předkovi nemusí být prázdná. Pokud se v této metodě předka nachází kód, je považován za výchozí. Příklad metody předka je na řádku 3 až 7 a k přepsání metody předka dojde na řádku 13 až 17 kódu 4.9. [24]

Přístup k předkovi

Někdy je potřeba uvnitř potomka volat metodu, která se vyskytuje v předkovi funkčního bloku. Pro volání metod předka potomkem nebo použití jeho atributů se používá klíčové slovo *SUPER*. V příkladu 4.9 se toto klíčové slovo vyskytuje na řádku 21, kde metoda *SuperDescription* volá metodu *Description*. [24]

Přístup k vlastním metodám a atributům

Pro přístup k atributu třídy nebo metody je možné použít klíčové slovo *THIS*. Toto klíčové slovo není potřeba používat, jelikož překladač automaticky vybere identifikátor z nejbližší úrovně. Příklad použití klíčového slova *THIS* se nachází na řádku 28 příkladu kódu 4.9. [24]

Kód 4.9: Příklad v jazyku ST s využitím dědičnosti a virtuálních metod [24]

```
FUNCTION_BLOCK Base

    METHOD VIRTUAL Description : STRING

        Description := "Base";

    END_METHOD

END_FUNCTION_BLOCK

FUNCTION_BLOCK Descendant EXTENDS Base

    METHOD VIRTUAL Description : STRING

        Description := "Descendant";

    END_METHOD

    METHOD SuperDescription : STRING

                SuperDescription := SUPER.Description();

    END_METHOD

    METHOD ThisDescription : STRING


        ThisDescription := THIS.Description();

    END_METHOD

END_FUNCTION_BLOCK

PROGRAM this_super_prg
    VAR
        fb : Descendant;

        desc, this_desc, super_desc : STRING;
    END_VAR

    desc := fb.Description();                // returns "
    Descendant"
    this_desc := fb.ThisDescription();        // returns "
    Descendant"
    super_desc := fb.SuperDescription();    // returns "Base"

END_PROGRAM
```

4.2.5 Namespace

Namespace je konstrukce, pomocí které můžeme uspořádávat třídy, objekty a proměnné do skupin. Můžeme mít díky této konstrukci například dvě stejně pojmenované konstrukce, ale pokud jsou v různých *namespace*, jsou jednoznačně identifikovatelné. Možným použitím může být vytváření knihoven a fremeworků. V jazyce Java se podobná konstrukce nazývá *package*. [24, 20]

Kód 4.10: Příklad v jazyku ST s namespace

```
NAMESPACE hardware
  VAR_GLOBAL
    plc : PLC;
    sensorA1 : BOOL;
    sensorB1 : BOOL;
    motorFOutput : BOOL;
    motorGOutput : BOOL;
  END_VAR
    FUNCTION_BLOCK
    //...
    END_FUNCTION_BLOCK
END_NAMESPACE
```

4.2.6 Rozhraní

Interface neboli rozhraní umožňuje více oddělit programátora od vnitřního fungování funkčních bloků (tříd). Velkou výhodou rozhraní oproti virtuálním funkčním blokům je, že každá třída může implementovat až mnoho rozhraní, avšak při dědění může dědit maximálně z jedné třídy. Neobsahují proměnné, pouze definované prázdné metody se stejným názvem jako metody funkčních bloků. Jedná se vlastně o funkční blok (třídu) složenou jen z virtuálních (abstraktních) metod. Pro implementování rozhraní do funkčního bloku se v jazyku ST používá klíčové slovo *IMPLEMENTS*. [24, 20, 23]

Kód 4.11: Příklad použití INTERFACE v jazyku ST

```
INTERFACE AutomatInterface
  METHOD getTime : TIME
  VAR_INPUT
      step : int;
      sequenceID : INT;
  END_VAR
    END_METHOD
END_INTERFACE

FUNCTION_BLOCK Automat IMPLEMENTS AutomatInterface
    METHOD getTime : TIME
      VAR_INPUT
```

```
13          step : int;
14          sequenceID : Int;
15        END_VAR
16        getTime := sequenceArray[sequenceID].getWaitingTime(step
      := step);
17    END_METHOD
18  END_FUNCTION_BLOCK
```

4.3 Jaké prvky OOP v jazyku ST chybí

I když je OOP do určité míry v jazyku ST implementované, jsou zde určité nedostatky oproti moderním programovacím jazykům. Jelikož ST není přímo překládán do strojového kódu ale do jazyků jako je například C/C++ nebo Java, vznikají tak rozdíly v tom, co jazyk nabízí. Například IDE (Integrated Development Envitorment - vývojové prostředí) Mervis překládá ST do C/C++, ale jsou i jiní vývojáři runtimů, kteří překládají ST do Javy. To má za následek to, že jejich implementace jazyka ST obsahuje například modifikátory přístupu.

4.3.1 Konstruktor

Konstruktor je speciální metoda, která se volá při vytvoření nové instance třídy. [20] V případě například Mervisu není konstruktor k dispozici ve formě metody definovatelné programátorem. Inicializace instance třídy probíhá tak, že v bloku proměnných můžeme při inicializaci objektu zadávat vstupní proměnné. Nelze však provádět inicializační kód jako u jazyku Java, která umožňuje například vyhodnotit vstupy a výsledky zapsat do atributů objektu. [24]

Kód 4.12: Příklad konstruktoru v jazyce JAVA [23]

```
1  public class Main {
2    int modelYear;
3    String modelName;
4
5    public Main(int year, String name) {
6      modelYear = year;
7      modelName = name;
8    }
9
10   public static void main(String[] args) {
11     Main myCar = new Main(1969, "Mustang");
12     System.out.println(myCar.modelYear + " " + myCar.modelName);
```

```
  }
}
```

4.3.2 Přetěžování metod a operátorů

Přetěžování metod slouží k přepisování metod. V praxi to znamená, že překladač zvolí metodu podle parametrů, které programátor použije. Přetěžování operátorů je definování chování při použití operátoru jako například plus, potom díky tomu můžeme sčítat objekty. Ani jednu z těchto funkcí jazyk ST nepodporuje. [24]

Kód 4.13: Příklad přetěžování metod v jazyce JAVA [25]

```
public class Sum {

    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y)
    {
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }

    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Část II

Praktická část práce

Kapitola 5

Úvod do praktické části práce

V této kapitole se věnuji tomu, jak jsem postupoval při volbě typu automatu a následně jeho Teach-In algoritmu. Pro řízení je třeba také zvolit programovatelný logický kontrolér, který bude řídit pneumatický manipulátor, a software, pomocí kterého tento kontrolér budu programovat. Dále se věnuji volbě struktury programování a zmiňuji pyramidu automatizace, podle které budu strukturovat vlastní dokumentaci.

5.1 Volba typu automatu

Prvním algoritmem, který musím zvolit, abych mohl navrhnout jeho učení, je automat. Tedy algoritmus toho, jak bude pneumatický manipulátor vykonávat sekvenci. Základním požadavkem je taková sekvence, kterou je možné měnit za běhu programu, aby mohlo fungovat učení. Není možné generovat kód, proto musím sekvenci definovat pomocí proměnných. Podmínky přechodu a výstupní hodnoty pro daný stav automatu musí být vepsány v proměnných nikoli v kódu. Musím tedy zvolit takové sekvenční řízení, které toto bude umožňovat.

Z hlediska povahy úlohy, kde výstupní hodnoty nezáleží na aktuálním vstupu, ale pouze závisí na aktuálním stavu, volím pro svou práci Mealyho sekvenční stroj. Ten aplikuji pomocí dvojrozměrných polí tak, že sekvenční stroj se bude skládat z translačního pole a pole výstupní funkce, kde každý řádek značí jeden stav automatu a pokud souhlasí řádek translačního pole se

vstupy manipulátoru, pak se posune do nového stavu. Tedy posune se o jeden řádek, dokud nedojde do posledního stavu a nevrátí se opět na začátek.

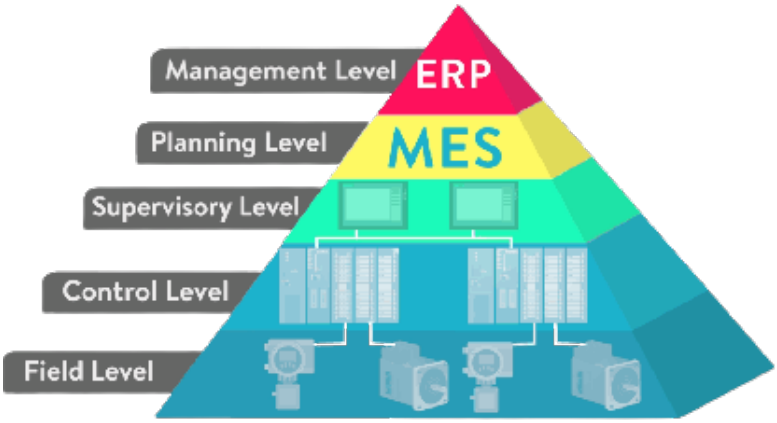
5.2 Návrh Teach-in algoritmu

Teach-in algoritmus je způsob "programování", kdy je zaznamenáván průběh používání systému za účelem následného opakování například odkrokováním sekvence, kterou chceme stroj naučit. Cílem této práce je navrhnout Teach-in algoritmus a aplikovat jej. Proto jsem si stanovil, co by měl můj algoritmus splňovat, a to, že by měl umožňovat naučit více než jednu sekvenci. Dále jsem si stanovil, že by měl umět vyčkávat v každé poloze nastavenou dobu. Měl by být také snadno rozšířitelný a znovupoužitelný.

Algoritmus tedy bude vycházet ze samotného automatu tím způsobem, že bude měnit translační pole a pole výstupní funkce, které používá automat. Tedy uživatel odkrokuje sekvenci takovým způsobem, že nastaví manipulátor do požadované polohy a uloží tuto polohu jako jeden stav do sekvence. Tedy samotný Teach-in algoritmus je prosté uložení hodnot do proměnných.

5.3 Struktura technické dokumentace

Struktura průmyslové automatizace se dělí na pět vrstev podle pyramidy automatizace, kterou vidíte na obrázku 5.1. Tento model je základním rozdělením výroby v průmyslových podnicích. Nejnižší vrstvou této pyramidy je polní vrstva (field level), do které řádíme všechny senzory a aktuátory ve výrobě. Jsou to jednotlivé pohony, kontrolky, senzory tlaku, teploty, vlhkosti a mnoho dalších. Účelem této vrstvy je převod fyzikálních hodnot na elektrické signály a naopak. [4]



Obrázek 5.1: Pyramida automatizace [4]

Další vrstvou je řídící vrstva (control level), která zahrnuje programovatelné logické automaty. Ty převádí elektrické signály na proměnné a řídí jednotlivé procesy ve výrobě. Tedy dávají elektrické signály aktuátorům, převádějí analogové signály na elektrické pomocí ADC převodníků a získávají hodnoty z digitálních vstupů a zapisují hodnoty na reléové nebo digitální výstupy. Obsahují také některé sběrnice typicky používané v průmyslu. [4]

Třetí vrstvou je dispečerská vrstva (supervisory level). Do této vrstvy řadíme operátorské panely, SCADA servery a velíny výrobních provozů. Ty slouží jako uživatelské rozhraní pro řízení procesů a ovládají chování kontrolní vrstvy. [4]

Čtvrtou vrstvou je vrstva plánování výroby (planning level), což jsou výrobní informační systémy. V této vrstvě se odehrává správa výrobních postupů, plánování a rozvrhování výroby, řízení a vyhodnocení výroby, správa prostojů, řízení procesu kvality, správa materiálu, sběr dat nebo výkonnostní analýzy. [26]

Poslední vrstvou je vrstva manažerská (management). Ta zahrnuje systémy plánování podnikového rozvoje (ERP systémy). V tomto systému probíhá správa samotného podniku, jako jsou personální kapacity, faktury, plány rozvoje podniku a výsledky hospodaření. [4]

Pro přehlednost jsem svou dokumentaci práce, která zahrnuje první tři vrstvy této pyramidy automatizace, rozdělil na tyto vrstvy a postupně je dokumentuji od nejnižší vrstvy až k dispečerské vrstvě. Nejdříve se zabývám použitým hardwarem, tedy polní vrstvou. Dále dokumentuji můj program pro PLC, který je součástí řídící vrstvy, a grafické rozhraní SCADA, to patří

do vrstvy operátorské. [4]

5.4 Úpravy na polní vrstvě pneumatického manipulátoru

V rámci této diplomové práce nebylo úkolem pneumatický manipulátor sestrojit ani ho nějak upravovat z hlediska polní vrstvy. Pneumatický manipulátor byl zachován v původním stavu, pouze došlo k mnoha opravám zejména na rozvodech stlačeného vzduchu, protože hadice byly popraskané. K největším únikům docházelo u plastových částí šroubení tvaru L pro přívod stlačeného vzduchu. Zde vlivem únavy materiálu došlo k popraskání téměř všech nástrčných šroubeních a k značným únikům vzduchu, proto byla tato šroubení vyměněna za hliníková přímá nástrčná šroubení.

5.5 Výměna programovatelného logického automatu

Původně byl pneumatický manipulátor řízen pomocí PLC *SIMATIC LOGO!* od firmy *SIEMENS*. Toto PLC je omezeno na práci s proprietárním softwarem a jedná se o uzavřené zařízení. Cílem této práce je se zaměřit na PLC s otevřeným softwarem a možností s nimi pracovat na pokročilejší úrovni. Proto byl vybrán PLC značky *Unipi*, které umožňuje přístup do operačního systému programovatelného logického automatu. Rovněž toto PLC nabízí možnost použití libovolného softwaru, který můžu použít pro řízení, nebo možnost naprogramování si vlastního.

Nejdříve byla práce programována pro Unipi Neuron M203, který je řízen SBC Raspberry Pi 3B+. Posléze bylo nahrazeno PLC Unipi Axon M205, které používá alternativní SBC NanoPi NEO Plus2.

5.6 Volba struktury programování

Při řešení diplomové práce jsem se nejdříve pokusil naprogramovat PLC pomocí strukturovaného programování v ST a FBD (Funcition Block Diagram - Diagram funkčních bloků). Funkční blok pro Teach-in algoritmus, který jsem vytvořil, měl v podstatě za úkol zapisovat do globálních proměnných typu pole za určitých vstupní proměnných, které povolily například zapsání vstupních a výstupních hodnot do globálních proměnných. Toto řešení se postupem času začalo zdát jako nevhodné, protože implementace nové funkce a oprava chyb zabrala obrovské množství času.

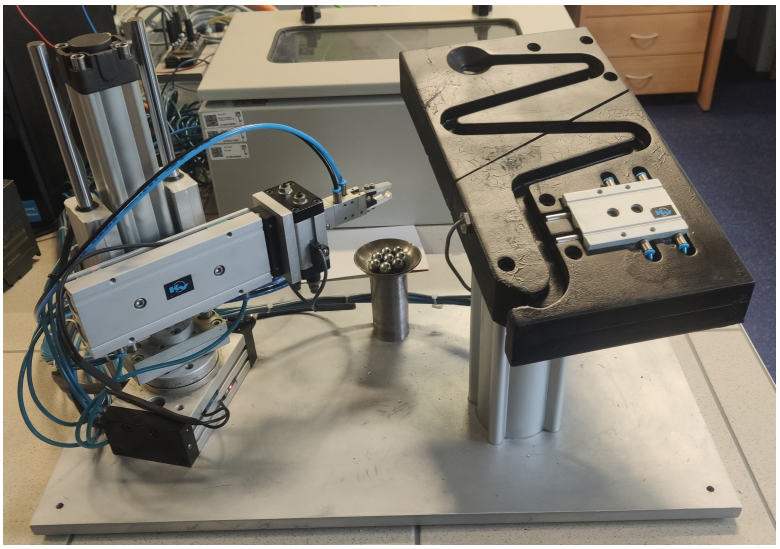
Velký problém nastával při předávání polí vstupních a výstupních hodnot. Pokud došlo ke změně velikosti pole, bylo potřeba v programu předefinovat pole u vstupních a výstupních proměnných a to v každém dotčeném funkčním bloku. Z toho vyplývá, že pokud celý program mezi jednotlivými funkčními bloky předává vstupní a výstupní proměnné PLC pomocí dvou polí, je toto řešení značně nepraktické pro opravy chyb a rozšiřování programu.

Problémem strukturovaného programování je, že program je velice často nepřehledný a při implementování nových funkcí je potřeba přemýšlet mnohdy nad celým programem než nad jeho malou částí. To značně ztěžuje vývoj složitějších řídících systémů.

Následné rozhodnutí pro přepracování celého programu do objektově orientovaného programování byl krok správným směrem. Počet funkčních bloků jsem tím snížil na polovinu a program se stal velmi přehledným a snadno rozšířitelným.

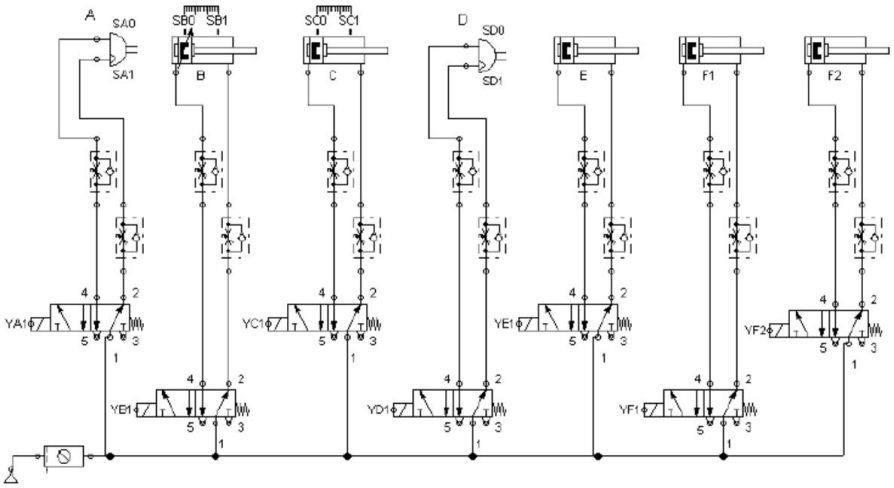
Kapitola 6

Pneumatický manipulátor



Obrázek 6.1: Pneumatický manipulátor

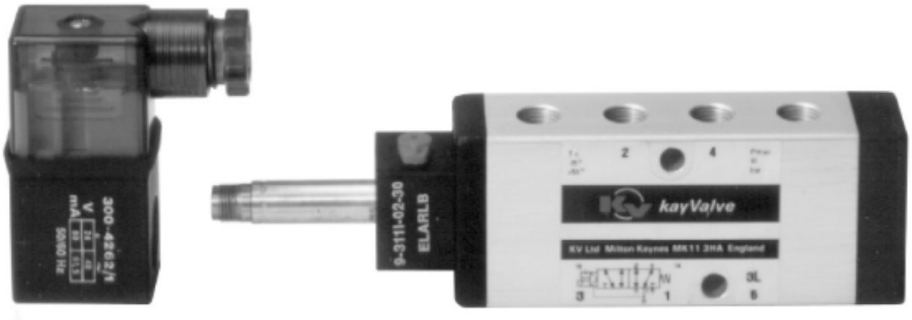
Manipulátor, pro který jsem v této práci připravil řízení, se sestává ze sedmi pneumatických pohonů. Součástí upravované učební pomůcky není pouze manipulátor, ale i dráha pro kuličky se dvěma pneumatickými závorami. Před první závorou ve směru pohybu kuličky se nachází senzor přiblížení, který je realizovaný pomocí indukčního senzoru.



Obrázek 6.2: Elektropneumatický obvod pneumatického manipulátoru [5]

6.1 Ventily

K ovládání pneumatického manipulátoru slouží ventilový terminál se sedmi monostabilními rozvaděči. Jsou umístěny přímo v hlavní skříni pneumatického manipulátoru a řídí jednotlivé motory. Na obrázku 6.3 vidíte použitý rozvaděč.



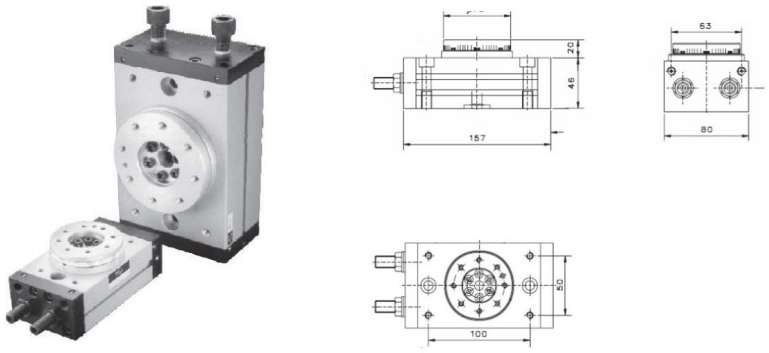
Obrázek 6.3: Monostabilní rozvaděč [5]

Výkres	Pohon	Zapojení	Symbolická proměnná
YA1	A	RO 2.7	hardware.motorAOutput
YB1	B	RO 2.5	hardware.motorBOutput
YC1	C	RO 2.4	hardware.motorCOutput
YD1	D	RO 2.6	hardware.motorDOutput
YE1	E	RO 2.3	hardware.motorEOutput
YF1	F1	RO 2.2	hardware.motorF1Output
YF2	F2	RO 2.1	hardware.motorF2Output

Tabulka 6.1: Zapojení rozvaděčů na výstupy PLC a jejich proměnné v Mervisu

6.2 Motor A

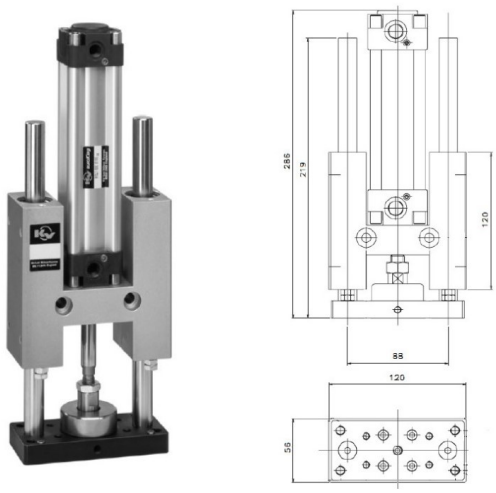
Motor A je dvojčinný otočný servopohon s magnetickým pístem a možností vymezení úhlu rotace manipulátoru. Rozsah úhlů je 0° až 190° a jejich vymezení probíhá pomocí šroubů umístěných na těle stolu. Je upevněn přímo k základně celého modelu a tvoří tak první osu manipulátoru. Krajní polohy jsou detekovány pomocí indukčních senzorů.



Obrázek 6.4: Motor A [5]

6.3 Motor B

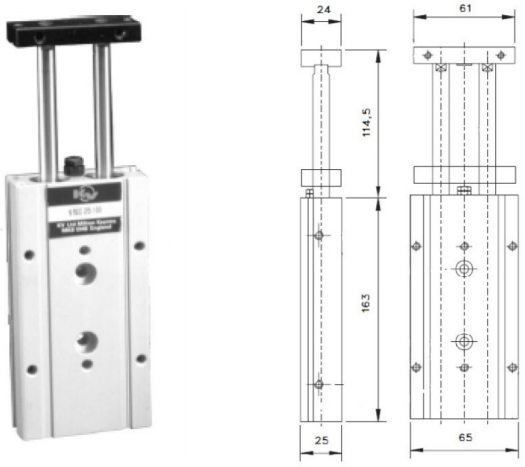
Sestava válce s vodící jednotkou motoru B tvoří kostru celého pneumatického manipulátoru. Zajišťuje pohyb manipulátoru ve směru první osy, tedy zdvihá celý komplet jeho paže vzhůru. Koncové polohy tohoto motoru jsou snímány pomocí indukčních senzorů.



Obrázek 6.5: Motor B [5]

6.4 Motor C

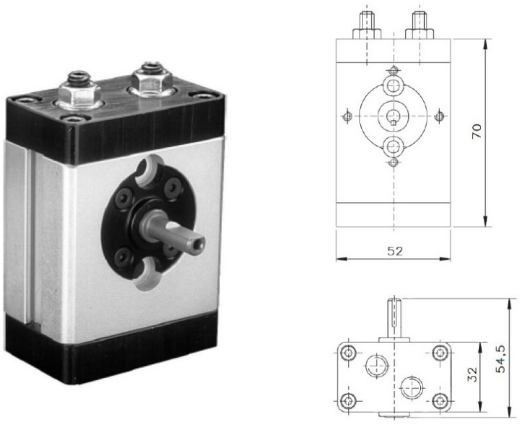
Motor C je dvojčinný pneumatický motor s dvěma magnetickými písty. Je připevněn kolmo na motor B a tvoří tak samotné rameno manipulátoru. Zajišťuje pohyb chapače vpřed a vzad, tedy prodloužení ruky. Jeho koncové polohy jsou rovněž snímány indukčními senzory.



Obrázek 6.6: Motor C [5]

6.5 Motor D

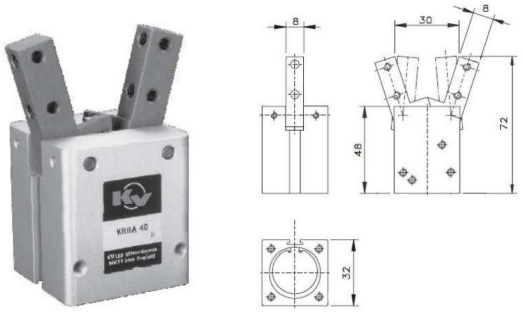
Motor D je dvojčinný rotační servopohon s magnetickým pístem, který je připevněn k motoru C. Slouží k rotaci chapače, tvoří tak druhou manipulátoru. Motor má dvě krajní polohy, vertikální a horizontální, a otáčí se v úhlu 90°.



Obrázek 6.7: Motor D [5]

6.6 Motor E

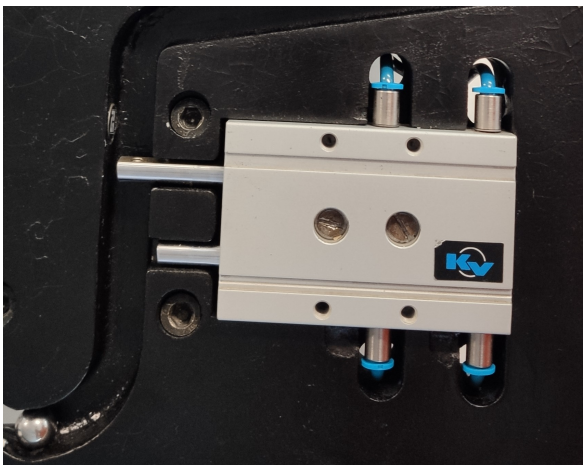
Motor E je dvojčinné úhlové chapadlo s magnetickým pístem, slouží k uchopování kuliček. Tento motor sám o sobě kuličku nepřenese, potřebuje pro to tvarově specializované kleště, které jsou připevněné k chapadlu.



Obrázek 6.8: Motor E [5]

6.7 Motory F1 a F2

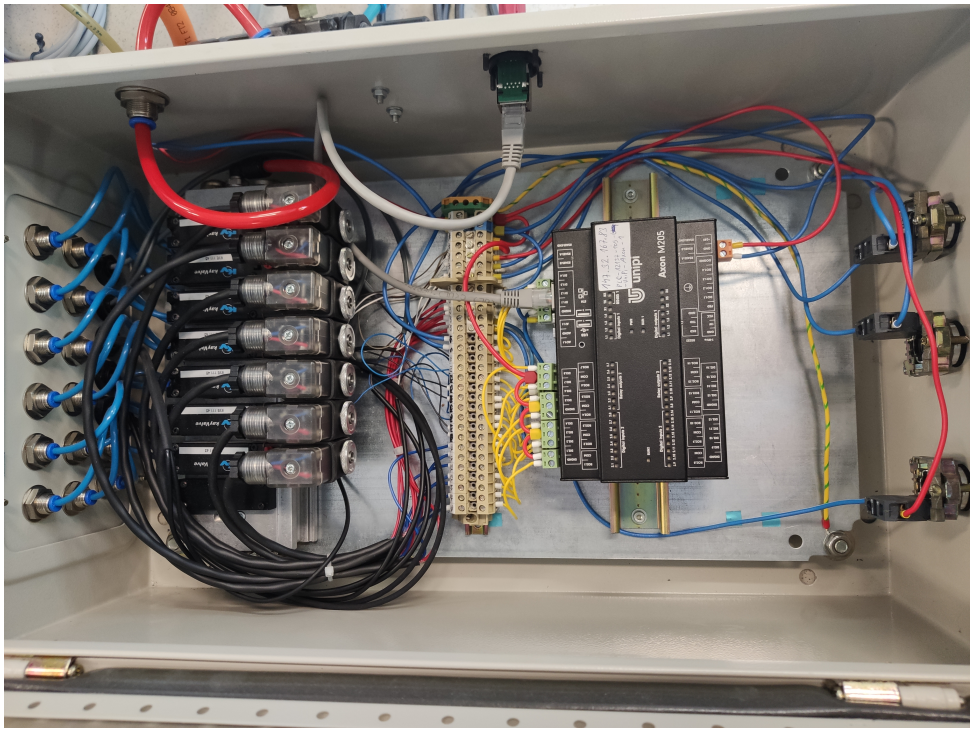
Jedná se o pneumatický oddělovač kuliček. Jeho použití je problematické z toho důvodu, že se snadno může stát, že pístnice vystřelí kuličku mimo kuličkovou dráhu. Aplikace tohoto oddělovače není úplně vhodná, bylo by potřeba celý oddělovač doplnit o plechové plíšky na jeho pístnicích a z tohoto důvodu posunout celý oddělovač. To by znamenalo vyfrézovat nové otvory do kuličkové dráhy. Pneumatický oddělovač může uživatel manipulátoru použít jako pneumatickou závoru. Úprava kuličkové dráhy není předmětem této diplomové práce, která se zabývá programováním a implementací Teach-in algoritmu.



Obrázek 6.9: Motory F1 a F2

6.8 Elektrické zapojení

Zapojení jednotlivých vstupů a výstupů jsem uvedl ve výkresu elektrického zapojení v příloze E. Na obrázku 6.10 je fotografie vnitřku rozvodné skříně, v které jsem nahradil PLC a zapojil k terminálu.



Obrázek 6.10: Rozvodná skříň

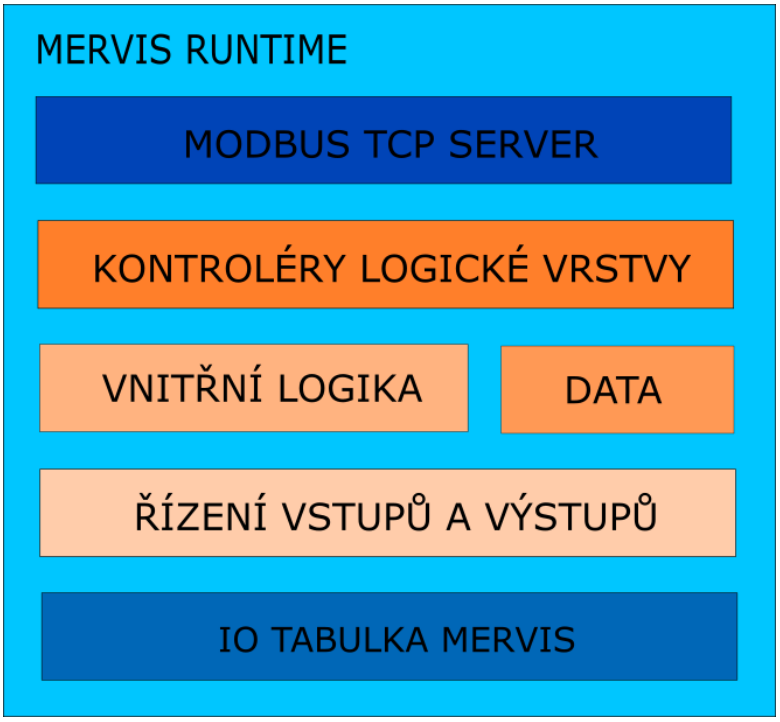
Kapitola 7

Architektura programu a integrace vstupů a výstupů do programu

Program pro logický automat s runtimem Mervis, který jsem vytvořil, je rozdělen do tří vrstev. Toto uspořádání velmi zjednodušuje orientaci v programu a zjednodušuje jeho rozšířitelnost. Na obrázku 7.1 jsou vyznačeny modře již předem připravené části runtimu Mervis a oranžově jsou vrstvy, které jsem implementoval. Diagram na obrázku 7.1 značí jednotlivé vrstvy od nejnižší vrstvy hardwarových vstupů a výstupů, kterou Mervis reprezentuje jako tabulku vstupů v výstupů, až po tu nejvyšší vrstvu, kterou je komunikační vrstva pro řízení celého programu, v mém případě jsem zvolil Modbus TCP. Jedná se sice o starší protokol, ale není vázán žádnou licencí a je zcela zdarma. Jeho nevýhodou je nezašifrovanost komunikace, to ale řeší Mervis sám pomocí SSL (Secure Socket Layer). Takže je možné tuto komunikaci mít i zašifrovanou, pokud se jedná o Modbus TCP nikoli Modbus RTU.

Z hlediska architektury mého programu jsem rozdělil program na čtyři části. V nejnižší části mého programu se nachází vrstva, která zajišťuje to, že veškeré vstupní a výstupní proměnné programu jsou přístupné z jedné instance funkčního bloku. To umožňuje velice snadno přidávat nové vstupy a výstupy do programu.

Další vrstvou je ta nejdůležitější vrstva, a to vrstva logická. V níž jsou definované veškeré funkce pneumatického manipulátoru, jako jsou ochrana proti kolizi, nebo Teach-in algoritmus. Jsou zde definované metody pro ovládání automatu nebo přímé ovládání pohonů. Důležitou součástí programu je datová část, kterou jsem na obrázku 7.1 zařadil do stejné úrovně jako logickou vrstvu, protože tato část nekomunikuje s ostatními vrstvami. Jedná se o



Obrázek 7.1: Vrstvy mého softwaru Mervis Runtime

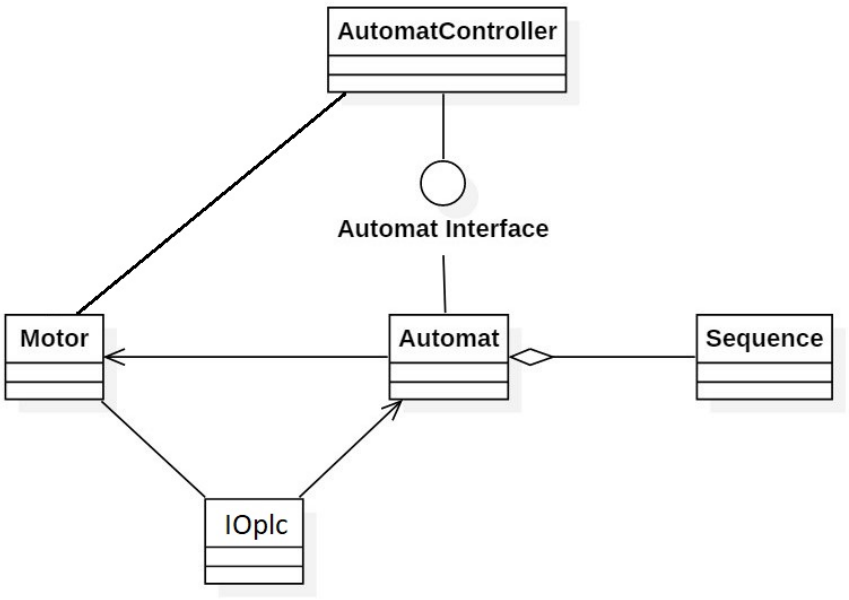
funkční blok sekvence, který v sobě ukládá všechna data pro danou sekvenci.

Poslední vrstvou mého programu je vrstva řídící, která v sobě obsahuje funkční bloky, které volají metody logické vrstvy na základě proměnných komunikačního protokolu Modbus TCP. Toto oddělení řídící vrstvy programu velmi zefektivňuje rozšíření programu o další funkce.

7.1 Integrace vstupů a výstupů do programu

Obecně při programování většiny PLC pomocí jiných jazyků *IEC 61131-3* nebo také ST ale v strukturovaném režimu, by program používal přímo proměnné namapované na tabulku vstupů a výstupů. V případě objektově orientovaného přístupu je používaní globálních proměnných nežádoucí. Jelikož ve své práci používám objektově orientované programování v jazyku ST ve vývojovém prostředí *Mervis*, které tento přístup podporuje, je také důležité propojení vstupů a výstupů PLC do programu.

Vytvořil jsem proto funkční blok, který v sobě obsahuje dvě pole, jedno je



Obrázek 7.2: Zjednodušený diagram funkčních bloků programu pro PLC

čistě pro výstupní a druhé pro vstupní hodnoty. Tyto pole se skládají pouze s booleovských hodnot a jsou klíčové pro běh celého programu. S použitím návrhového vzoru jedináček (singleton) přistupují ostatní vrstvy programu k proměnným vstupních a výstupních hodnot pomocí volání metod typu *get* a *set* funkčního bloku *IOplc*.

Tento způsob má oproti napojení proměnných přímo v logické vrstvě několik výhod. První výhodou je obrovská přehlednost a jednoduchost programu. Další výhodou je snadná rozšířitelnost programu. Pokud chceme, aby se program rozšířil o další proměnné, stačí jen rozšířit příslušné pole a nově vytvořené pozice propojit s proměnnými v namespace *hardware*. Z tohoto pole následně bude program číst hodnotu pomocí metod z logické vrstvy programu.

7.2 Mapování proměnných k hardwarovým vstupům a výstupům

Mervis nepodporuje mapování hardwarových vstupů a výstupů na proměnné instancí objektů, proto jsem vytvořil namespace *hardware*, kde jsou definované globální proměnné pro namapování vstupů a výstupů PLC. V tabulce 7.1 jsem zapsal pro každý výstup jeho označení na svorkovnici, dále označení v

Pohon	Zapojení	Symbolická proměnná	Index pole výstupů
A	RO 2.07	hardware.motorAOutput	0
B	RO 2.05	hardware.motorBOutput	1
C	RO 2.04	hardware.motorCOutput	2
D	RO 2.06	hardware.motorDOutput	3
E	RO 2.03	hardware.motorEOutput	4
F1	RO 2.02	hardware.motorFOutput	5
F2	RO 2.01	hardware.motorGOutput	6

Tabulka 7.1: Symbolická jména výstupů PLC

Sezor	Zapojení	Symbolická proměnná	Index pole vstupů
A0	DI 2.02	hardware.sensorA0	0
A1	DI 2.01	hardware.sensorA1	1
B0	DI 2.04	hardware.sensorB0	2
B1	DI 2.03	hardware.sensorB1	3
C0	DI 2.06	hardware.sensorC0	4
C1	DI 2.05	hardware.sensorC1	5
D0	DI 2.08	hardware.sensorD0	6
D1	DI 2.07	hardware.sensorD1	7
S	DI 1.02	hardware.sensorBall	8

Tabulka 7.2: Symbolická jména vstupů PLC

IO tabulce v Mervisu a jeho symbolickou proměnnou, která je definovaná v namespace *hardware*, a pozici v poli výstupů v instanci funkčního bloku *IOplc*. Tabulka 7.2 je obdobnou tabulkou, s tím rozdílem, že se v ní nacházejí vstupy PLC.

7.3 Funkční blok IOplc

Název metody	Parametry	Typ	Návratový typ
read	-	-	-
write	-	-	-
getInput	valueArrPosition	INT	BOOL
getOutput	valueArrPosition	INT	BOOL
setOutput	valueArrPosition input	INT BOOL	-

Tabulka 7.3: Rozhraní funkčního bloku IOplc

7.3.1 read

Přečte vstupy z vstupní tabulky PLC a zapíše je do pole vstupů.

7.3.2 write

Zapíše výstupy z pole výstupních hodnot do výstupní tabulky PLC.

7.3.3 getInput : BOOL

Vrací hodnotu z pole vstupů.

- **valueArrPosition : INT** - Pozice hodnoty v poli vstupů.
- **getOutput : BOOL** - Vrací hodnotu z pole výstupů.
- **valueArrPosition : INT** - Pozice hodnoty v poli výstupů.

7.3.4 setOutput

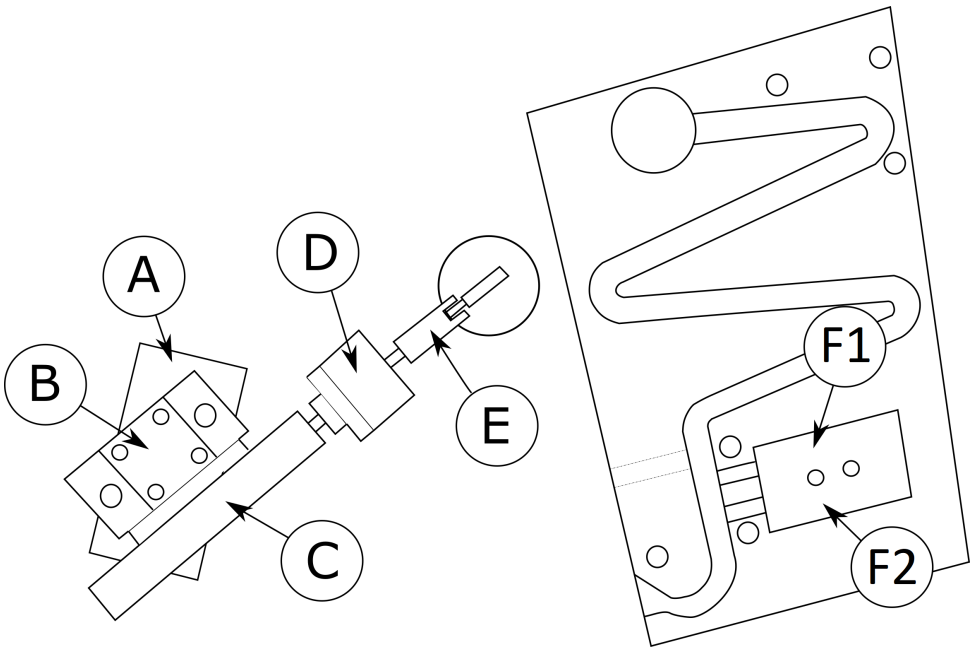
Zapsání hodnoty do pole výstupů.

- **valueArrPosition : INT** - Pozice hodnoty v poli výstupů.
- **input : BOOL** - Zapisovaná hodnota do pole výstupů.

Kapitola 8

Řízení pohonů

Tato vrstva zajišťuje vnitřní logiku fungování pneumatického manipulátoru. Jsou v ní obsaženy funkční bloky pro řízení pohybu motorů a automat daného systému.



Obrázek 8.1: Schéma pohonu s označením pohonů

8.1 Antikolizní systém

Antikolizní systém zajišťuje omezení pohybů manipulátoru, aby nedošlo ke kolizi manipulátoru s kuličkovou dráhou. Celkem jsou na manipulátoru 4 pohony, které mohou způsobit kolizi. Jsou to pohony A až D, dle schématu 8.1. Z tohoto důvodu jsem navrhl algoritmus, který zajišťuje, že pohony se kolizního pohybu nedopustí. Nejdříve je nutné pro tyto pohony zavést logické funkce, které definují kolizní pohyby pneumatických pohonů.

Algoritmus pracuje tak, že každá instance funkčního bloku motoru obsahuje implementované logické funkce, které povolují a zamezují pohyb pro daný pohon v závislosti na vstupech ze senzorů koncových poloh motoru A až D. Tyto funkce jsou implementovány tak, že uvnitř funkčního bloku motoru se nachází pole obsahující logickou tabulku všech možných stavů vstupů. Druhé pole v sobě udržuje uchovává informaci, pro které řádky je pohyb povolen. Jelikož jsou všechny stavy ve výchozím stavu povolené, stačí jen definovat zakázané řádky (stavy). Před vykonáním pohybu motoru se řídí algoritmus pouze povolenými řádky a porovnává je s aktuálními vstupy PLC, pokud se neshoduje žádná z povolených kombinací, pohyb se nevykoná. Díky tomu, že automat ověřuje pouze povolené kombinace vstupů nikoli zakázané, zamezí se možnosti vykonávat jiný pohyb, když je automat v mezistavu. V tomto případě nemá motor ani jeden koncový senzor aktivní a vstupy se nebudou shodovat s žádným řádkem.

8.1.1 Chybné stavy pneumatického manipulátoru

V tabulce 8.1 jsou vypsány stavy jednotlivých pohonů, kdy hodnota v tabulce symbolizuje polohu 0 a 1. V pravé části tabulky jsem označil stavy pneumatického manipulátoru, které nejsou možné z hlediska konstrukce. Pro zdůraznění jsem zde označil chybné stavy písmenem E. Jedná se tedy o stavy, kterých nelze žádným pohybem dosáhnout, a tedy je budu dále v následujících tabulkách takto označovat.

A	B	C	D	CHYBA
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	E
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	E
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Tabulka 8.1: Chybné stavy pneumatického manipulátoru

8.1.2 Odvození funkce povolení pohybu pohonu A

Pro pneumatický pohon A jsem z pozorování manipulátoru v tabulce 8.2 označil kombinace vstupů z koncových senzorů pneumatických pohonů. V tabulce 8.2 jsem vyznačil logickou hodnou 1 pro pohyby motoru A z polohy A0 do polohy A1, které nejsou kolizními.

B0	B1	C0	C1	D0	D1	A0 → A1
1	0	1	0	1	0	1
1	0	1	0	0	1	1
1	0	0	1	1	0	0
1	0	0	1	0	1	0
0	1	1	0	1	0	1
0	1	1	0	0	1	1
0	1	0	1	1	0	E
0	1	0	1	0	1	1

Tabulka 8.2: Povolené kombinace stavů vstupů pro pohyb motoru A z A0 do A1

Z tabulky 8.2 je patrné, že pohyb z A0 do A1 není povolen ve dvou stavech. Prvním případ, kdy nesmí k takovému pohybu dojít je, když se chapač

nachází pod kuličkovou dráhou. Druhý případ je zcela stejný, liší se jen ve stavu pohonu D (natočení chapače). Kdyby k pohybu A0 na A1 došlo v těchto dvou případech, tak by manipulátor kolidoval s kuličkovou dráhou. Z tabulky jsem 8.2 odvodil logickou funkci 8.1.

$$A1 = YA1 \cdot \overline{(B0 \cdot \overline{B1} \cdot \overline{C0} \cdot C1 \cdot D0 \cdot \overline{D1} + B0 \cdot \overline{B1} \cdot \overline{C0} \cdot C1 \cdot \overline{D0} \cdot D1)} \tag{8.1}$$

B0	B1	C0	C1	D0	D1	A1 → A0
1	0	1	0	1	0	1
1	0	1	0	0	1	1
1	0	0	1	1	0	0
1	0	0	1	0	1	E
0	1	1	0	1	0	1
0	1	1	0	0	1	1
0	1	0	1	1	0	0
0	1	0	1	0	1	1

Tabulka 8.3: Povolené kombinace stavů vstupů pro pohyb motoru A z A1 do A0

Obdobně jako v předchozím případě jsem odvodil logickou funkci pro povolení opačného směru pohybu pohonu A, tedy směru ze stavu A1 do stavu A0. Z tabulky 8.3 vyplývají opět dva stavy, v kterých není možný tento pohyb motoru. Prvním stavem je stav, kdy se chapač nachází v drážce pro sbírání kuliček z kuličkové dráhy. Druhým stavem je případ, kdy je chapač přímo nad touto drážkou a zároveň je chapač ve vertikální poloze.

$$A0 = \overline{YA1} \cdot \overline{(B0 \cdot \overline{B1} \cdot \overline{C0} \cdot C1 \cdot D0 \cdot \overline{D1} + \overline{B0} \cdot B1 \cdot \overline{C0} \cdot C1 \cdot D0 \cdot \overline{D1})} \tag{8.2}$$

8.1.3 Odvození funkce povolení pohybu pohonu B

Pneumatický pohon B zajišťuje zdvih manipulátoru směrem nahoru a dolů. Pro odvození funkcí pro zamezení kolizního pohybu pohonu jsem zapsal všechny kombinace v vstupů koncových senzorů do tabulky 8.4, z které je patrné, že pohyb pohonu B ze stavu B0 na stav B1 je zakázaný ve dvou

stavech. Když se chapač nachází pod kuličkovou dráhou, změna stavu pohonu do této polohy by měla za následek kolizi. Dva stavy jsou zde proto, že nezáleží na natočení chapače.

A0	A1	C0	C1	D0	D1	B0 → B1
1	0	1	0	1	0	1
1	0	1	0	0	1	1
1	0	0	1	1	0	0
1	0	0	1	0	1	0
0	1	1	0	1	0	1
0	1	1	0	0	1	1
0	1	0	1	1	0	1
0	1	0	1	0	1	E

Tabulka 8.4: Povolené kombinace stavů vstupů pro pohyb motoru B z B0 do B1

Z tabulky 8.4 jsem stav pohonu B1 vyjádřil rovnicí 8.3.

$$B1 = YB1 \cdot \overline{(A0 \cdot \overline{A1} \cdot \overline{C0} \cdot C1 \cdot D0 \cdot \overline{D1} + \overline{A0} \cdot A1 \cdot \overline{C0} \cdot C1 \cdot D0 \cdot \overline{D1})} \tag{8.3}$$

A0	A1	C0	C1	D0	D1	B1 → B0
1	0	1	0	1	0	1
1	0	1	0	0	1	1
1	0	0	1	1	0	E
1	0	0	1	0	1	0
0	1	1	0	1	0	1
0	1	1	0	0	1	1
0	1	0	1	1	0	1
0	1	0	1	0	1	0

Tabulka 8.5: Povolené kombinace stavů vstupů pro pohyb motoru B z B1 do B0

Obdobně jako v předchozím případě je nutno zamezit koliznímu pohybu motoru také v opačném směru. Zakázanými stavy stavy jsou v tomto případě dva stavy. Prvním stavem je, když se chapač nachází nad kuličkovou dráhou v místě, kde upouští kuličky. Druhým je, když se nachází nad drážkou pro sbírání kuliček a chapač je v horizontální poloze. Z tabulky jsem vyjádřil rovnici 8.4 vyjadřující stav pohonu B0.

$$B0 = \overline{YB1} \cdot \overline{(A0 \cdot \overline{A1} \cdot \overline{C0} \cdot C1 \cdot \overline{D0} \cdot D1 + \overline{A0} \cdot A1 \cdot \overline{C0} \cdot C1 \cdot \overline{D0} \cdot D1)} \tag{8.4}$$

8.1.4 Odvození funkce povolení pohybu pohonu C

Motor C zajišťuje pohyb chapače vpřed a vzad. Má dvě krajní polohy a to C0 a C1. V tabulce 8.6 jsem uvedl kombinace vstupů ze senzorů a označil zakázané stavy jako logickou 0. Zakázané stavy manipulátoru, kdy se nesmí motor C pohnout do stavu C1, jsou dva. Prvním je pokud se manipulátor nachází v horní poloze, motor A je ve stavu A0 a chapač je ve vertikální poloze. Pak by došlo ke kolizi chapače s kuličkou dráhou. Další stav je, když se nachází chapač před drážkou pro sběr kuliček a je v horizontální poloze.

A0	A1	B0	B1	D0	D1	C0 → C1
1	0	1	0	1	0	1
1	0	1	0	0	1	1
1	0	0	1	1	0	0
1	0	0	1	0	1	1
0	1	1	0	1	0	1
0	1	1	0	0	1	0
0	1	0	1	1	0	1
0	1	0	1	0	1	1

*

Tabulka 8.6: Povolené kombinace stavů vstupů pro pohyb motoru C z C0 do C1

Z tabulky 8.6 jsem získal rovnici 8.5, která vyjadřuje stav C1 motoru C na vstupech ze senzorů.

$$C1 = YC1 \cdot \overline{(A0 \cdot \overline{A1} \cdot \overline{B0} \cdot B1 \cdot D0 \cdot \overline{D1} + \overline{A0} \cdot A1 \cdot B0 \cdot \overline{B1} \cdot \overline{D0} \cdot D1)} \tag{8.5}$$

Dle tabulky 8.7 vidíte, že motor C nemá žádný zakázaný stav v tomto směru.

A0	A1	B0	B1	D0	D1	C1 → C0
1	0	1	0	1	0	1
1	0	1	0	0	1	1
1	0	0	1	1	0	1
1	0	0	1	0	1	E
0	1	1	0	1	0	1
0	1	1	0	0	1	E
0	1	0	1	1	0	1
0	1	0	1	0	1	1

Tabulka 8.7: Povolené kombinace stavů vstupů pro pohyb motoru C z C1 do C0

8.1.5 Odvození funkce povolení pohybu pohonu D

V tabulce 8.8 vidíte závislost stavů vstupních hodnot ze senzorů poloh motorů pro pohyb motoru D ze stavu D0 do stavu D1. Takový pohyb motoru je zakázaný ve stavu, kdy se chapač nachází uvnitř drážky pro sběr kuliček.

A0	A1	B0	B1	C0	C1	D0 → D1
1	0	1	0	1	0	1
1	0	1	0	0	1	1
1	0	0	1	1	0	1
1	0	0	1	0	1	E
0	1	1	0	1	0	1
0	1	1	0	0	1	0
0	1	0	1	1	0	1
0	1	0	1	0	1	1

Tabulka 8.8: Povolené kombinace stavů vstupů pro pohyb motoru D z D0 do D1

Z tabulky 8.8 jsem získal rovnici 8.6 pro stav pohonu D1.

$$D1 = YD1 \cdot \overline{(\overline{A0} \cdot A1 \cdot B0 \cdot \overline{B1} \cdot \overline{C0} \cdot C1)} \tag{8.6}$$

A0	A1	B0	B1	C0	C1	D1 → D0
1	0	1	0	1	0	1
1	0	1	0	0	1	1
1	0	0	1	1	0	1
1	0	0	1	0	1	0
0	1	1	0	1	0	1
0	1	1	0	0	1	E
0	1	0	1	1	0	1
0	1	0	1	0	1	1

Tabulka 8.9: Povolené kombinace stavů vstupů pro pohyb motoru D z D1 do D0

Obdobně jako v předchozím případě je nutno zamezit koliznímu pohybu motoru také v opačném směru. V opačném směru má pohon zakázaný pohyb ve stavu, kdy se chapač nachází v poloze pro vypouštění kuliček do kuličkové dráhy, jak vidíte v tabulce 8.9. Z této tabulky rovněž vychází rovnice 8.7 pro stav D0 pro motor D.

$$D0 = \overline{Y\,D1} \cdot \overline{(A0 \cdot \overline{A1} \cdot \overline{B0} \cdot B1 \cdot \overline{D0} \cdot D1)} \tag{8.7}$$

8.2 Funkční blok: Motor

Název metody	Parametry	Typ	Návratový typ
setAnticollision	motorID	INT	-
	sensor1	BOOL	
	sensor0	BOOL	
	PK0 - PK15	BOOL	
isPosition1	-	-	BOOL
isPosition0	-	-	BOOL
isMovementSafe	-	-	BOOL
goTo1	disableAnticollision	BOOL	-
goTo0	disableAnticollision	BOOL	-

Tabulka 8.10: Motor - rozhraní funčního bloku

8.2.1 setAnticollision

- **motorID** - Pneumatický manipulátor pracuje s poli vstupů a výstupů, který dále zpracovává objekt třídy PLC. Tímto číslem se nastavuje pozice v poli, která odpovídá výstupu PLC pro ovládání ventilu daného pohonu. Tuto hodnotu je potřeba nastavit i u pohonů bez koncových senzorů.
- **sensor1** - Hodnota vstupního senzoru koncové polohy pohonu v poloze 1. Výchozí hodnota False, proto není třeba u pohonů bez senzorů koncových poloh tento vstup využívat.
- **sensor0** - Hodnota vstupního senzoru koncové polohy pohonu v poloze 0. Výchozí hodnota False, proto není třeba u pohonů bez senzorů koncových poloh tento vstup využívat.
- **PK0 - PK15** - Slouží k definování zakázaných stavů dle tabulky 8.11, která definuje veškeré kombinace stavů motorů robotického manipulátoru. Pokud se motor bude nacházet v zakázané kombinaci, nezmění svůj stav. Všechny kombinace jsou ve výchozím stavu povolené, pokud chceme kombinaci zakázat, je třeba při volání této metody zapsat hodnotu *False* do příslušné proměnné. Značení proměnných je PK#, kde za # dosadíme číslo řádku dle tabulky 8.11, jenž odpovídá stavu robotického

manipulátoru, při kterém nesmí dojít ke změně polohy motoru. Příkladem postupu volby *PK* může být, když se robot nachází pod kuličkovou dráhou, v tomto případě je třeba zakázat pohyb motoru A z polohy 0 do polohy 1. V tabulce nalezneme všechny stavy, kdy motor nesmí změnit svojí polohu, tedy pokud řešíme motor A, tak zakázaní pohybu z polohy 0 do polohy 1 bude odpovídat v tabulce těm řádkům, které mají v sloupci A číslo 0. Dále platí, že motor když je pod kuličkovou dráhou je motor B v poloze 0 a motor C v poloze 1. Avšak kolizi neodvrátí ani jedna z poloh motoru D, proto zvolíme oba řádky. Zvolená čísla řádků 2 a 3 doplníme o název proměnné *PK* a při volání metody zapíšeme hodnotu tímto způsobem: *PK2 := False, PK3 := False*

A	B	C	D	PK
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Tabulka 8.11: Tabulka pro volbu PK dle zakázaných pohybů

8.2.2 isPosition1

Vrací booleovskou hodnotu senzoru koncové polohy 1, je-li hodnota *True*, motor je v této koncové poloze.

8.2.3 isPosition0

Vrací booleovskou hodnotu senzoru koncové polohy 0, je-li hodnota *True*, motor je v této koncové poloze.

8.2.4 isMovementSafe

Vrací booleovskou hodnotu, jestli změna polohy motoru nezpůsobí kolizi, je-li hodnota *True*, pohyb je bezkolizní.

8.2.5 goTo1

Změní polohu motoru z polohy 0 do polohy 1, pokud objekt vyhodnotí, že se nejedná o kolizní stav, nebo pokud je antikolizní funkce vypnutá.

- **disableAnticollision** Defaultně nastavená na False. Nastavením hodnoty na True se vypne antikolizní funkce.

8.2.6 goTo0

Změní polohu motoru z polohy 0 do polohy 1, pokud objekt vyhodnotí, že se nejedná o kolizní stav, nebo pokud je antikolizní funkce vypnutá.

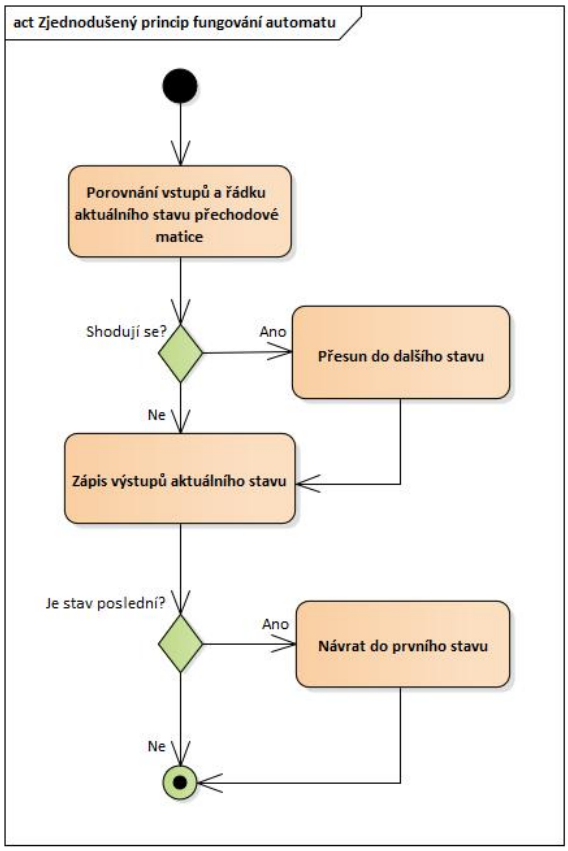
- **disableAnticollision** Defaultně nastavená na False. Nastavením hodnoty na True se vypne antikolizní funkce.

Kapitola 9

Automatické řízení a Teach-in

Automat díky použitému OOP je snadněji implementovatelný než pomocí strukturovaného programování. Pro uživatele třídy jde vlastně o jednu metodu, která se volá při každém cyklu PLC.

Automat pracuje na principu porovnání hodnot vstupů s řádkem v přechodovém poli a zápisu výstupních hodnot stejného řádku pole výstupů. Na obrázku 9.2 jsem velmi zjednodušeně znázornil tok algoritmu tohoto automatu. Nejdříve automat porovnává řádek pole přechodu s polem vstupů. Pokud se řádek shoduje, dojde k posunu do dalšího stavu. Pokud se řádek neshoduje, V každém cyklu program bez ohledu na splnění podmínky přechodu zapisuje hodnoty z pole výstupů na výstupy PLC. Výstupy nejsou nikdy zapisovány na přímo ale pomocí instancí jednotlivých motorů, které zamezují kolizi. Pokud tedy automat dojde do stavu, jehož hodnota se rovná uloženému číslu posledního stavu, vrátí se automat do prvního kroku.



Obrázek 9.1: Zjednodušený diagram fungování maticového automatu

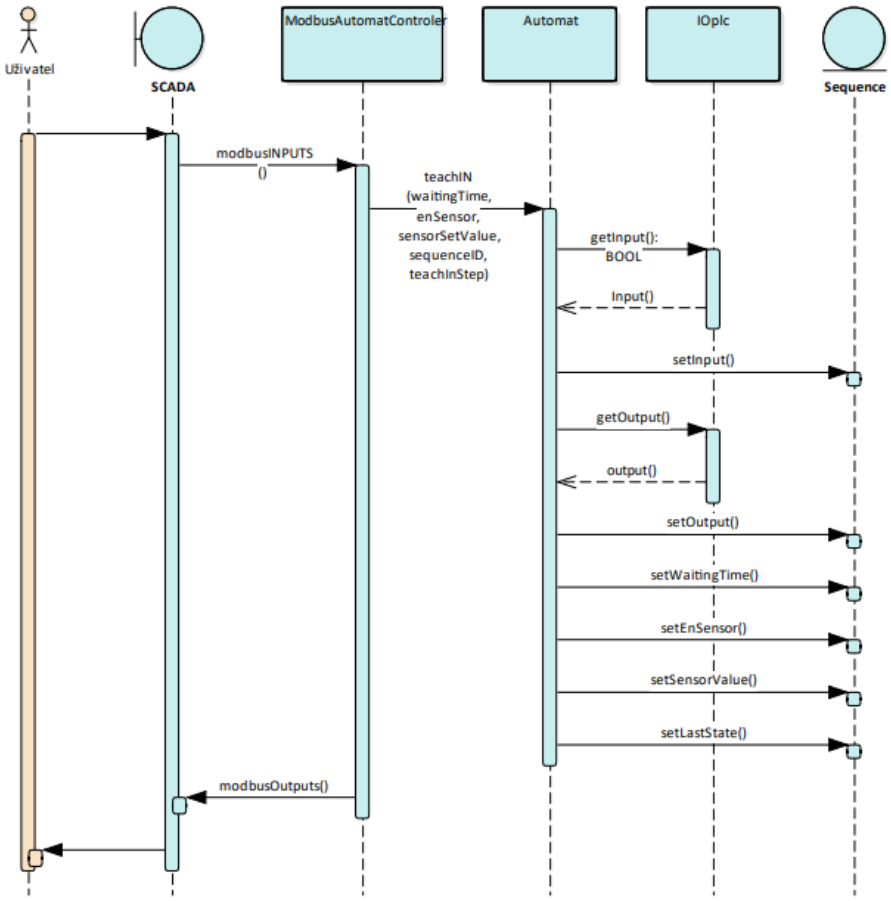
9.1 Teach-In

Teach-in algoritmus slouží k změně sekvenčního řízení bez změny kódu programovatelného logického automatu. Proto je nutné ho implementovat tak, že přechodová funkce a výstupní hodnoty jsou založeny na proměnných nikoli na pevném naprogramování sekvence. Tyto proměnné v sobě ukládají informaci o požadovaných hodnotách, s kterými automat porovnává informace. Taková implementace by v jiných jazycích dle *IEC 61131-3* než v jazyku ST, který jako jediný umožňuje vytvářet pole a podporuje objektově orientovaný přístup, byla podstatně složitější.

Uživateli systému jsem připravil uživatelské rozhraní, díky kterému lze snadno sekvence učit. Postup je takový, že uživatel postupně odkrokuje sekvenci pomocí manuálního ovládání a každý krok uloží. Při ukládání sekvence může uživatel přidávat další podmínky přechodu, kterými jsou hodnota

senzoru přítomnosti kuličky a vyčkávací čas.

Uložení sekvence probíhá následujícím způsobem. Tím, že uživatel stiskne tlačítko pro uložení kroku v grafickém rozhraní *SCADA*, dojde ke změně booleovské hodnoty v *Modbus* zprávě. Tato hodnota je namapovaná na proměnnou programu, jejíž hodnota *True* je podmínkou pro *IF* konstrukci ve funkčním bloku *ModbusAutomatController*. Takto je spuštěna metoda *teachStep* instance *Automat*. Metoda samotná je velmi jednoduchá, pomocí cyklů *for* totiž uloží všechny proměnné aktuálních vstupů a výstupů do polí, která jsou atributy objektů *Sequence*. Tedy se dá říct, že můj Teach-in algoritmus je tak jednoduchý, že se jedná pouze o uložení aktuálních hodnot pomocí zavolání jediné metody.



Obrázek 9.2: Sekvenční diagram funkce Teach-in

9.1.1 Funkce výchozí polohy

Jedna z metod, kterou rozhraní funkčního bloku *Automat* poskytuje, slouží k vyvolání pohybu do výchozí polohy. Pracuje tak, že je tato metoda volána dokud není automat ve výchozí poloze. Uvnitř této metody se nachází volání metody *motorGoMinus()* pro všechny instance pohonů. Díky antikoliznímu systému se při každém volání metody vykonají pouze povolené pohyby.

Automat aplikuje jakýsi jednoduchý algoritmus nejkratší cesty. Mapu, po které cestuje, tvoří povolené pohyby na základě aktuálních hodnot senzorů, ale pohybů vykonává i několik zároveň, pokud mu to mapa umožňuje.

9.1.2 Funkce vyčkávání

Vyčkávání slouží k tomu, že až se splní podmínky vstupů k přechodu do dalšího stavu, spustí se standardizovaný funkční blok *TON*. Po uběhnutí času pokračuje automat do dalšího stavu. Čas se dá nastavit různý pro každý krok a libovolnou sekvenci.

Protože je obtížnější umísťovat standardní časové funkční bloky do objektové struktury programu, nacházejí se tyto bloky v samostatných blocích programů a automat s nimi komunikuje pomocí přepisování a čtení proměnných.

9.2 Funkční blok: Automat

Název metody	Parametry	Typ	Návratový typ
resetAutomat	-		-
teachIN	waitingTime sensorSetVal teachSequence enSensor teachInStep	TIME BOOL INT BOOL INT	-
getCycleCounter	-		INT
getTime	sequenceID step	INT INT	TIME
run	sequenceID	INT	-
goToStart	-		BOOL
delSequence	sequenceID		-

Tabulka 9.1: Automat - Rozhraní funkčního bloku

9.2.1 resetAutomat

Provede vynulování čítače cyklů, dále vrátí automat do výchozího stavu.

9.2.2 teachIN

Metoda Teach-in algoritmu, provádí uložení stavu a přechodu pro jeden stav sekvence zvolené pomocí proměnné *teachSequence*. Provádí zápis všech vstupů a výstupů pro aktuální stav a následujících vstupních proměnných metody.

- **sequenceID : INT** - Volí sekvenci, do které chceme hodnoty uložit. Počet sekvencí je stanoven na 11, tedy hodnoty tohoto vstupu mohou být 0 až 10. Jedná se o identifikátor sekvence v programu.
- **teachInStep : INT** - Volí aktuální číslo kroku sekvence, který chceme sekvenci naučit. Sekvence se potom bude vykonávat postupně od kroku 0 do posledního uloženého kroku sekvence. Pro fungování sekvence je třeba, aby nebyly vynechané žádné kroky mezi prvním a posledním krokem, automat by se poté při vykonávání sekvence u tohoto kroku zastavil a vrátil do základní polohy. Maximální počet kroků sekvence je 100 kroků.

- **waitingTime : TIME** - Vyčkávací čas pro vyčkávání v poloze. Vyčkávací čas se spustí až po splnění přechodové podmínky do dalšího kroku, tedy zpozdí přechod do dalšího kroku při vykonávání sekvence.
- **enSensor : BOOL** - V daném kroku přidá do podmínky přechodu hodnotu senzoru. Pokud je v daném kroku a sekvenci nastavený na *False*, nebude brát v potaz hodnotu senzoru pro přechod. V opačném případě se do podmínky přechodu přidá hodnota senzoru.
- **sensorSetValue : BOOL** - Nastaví požadovanou hodnotu pro přechod do dalšího stavu. Pokud je hodnota *True*, bude pro přechod do dalšího stavu třeba, aby senzor byl v tuto chvíli aktivní. V opačném případě musí být neaktivní, pokud by hodnota byla nastavena na *False*, tak by automat čekal, dokud senzor nebude neaktivní.

9.2.3 getCycleCounter : INT

Vrací hodnotu čítače dokončených cyklů automatu.

9.2.4 getTime : TIME

Vrací hodnotu uloženého vyčkávacího času ve zvoleném kroku a sekvenci.

- **sequenceID : INT** - Volí sekvenci, z které chceme hodnotu získat. Počet sekvencí je stanoven na 11, tedy hodnoty tohoto vstupu mohou být 0 až 10. Jedná se o identifikátor sekvence v programu.
- **step : INT** - Volí číslo kroku sekvence, z kterého chceme získat hodnotu vyčkávacího času.

9.2.5 run

Voláním metody v programu je v provozu automat. Zavoláním metody provede automat jedno vyhodnocení aktuální přechodové funkce, pokud je přechodová funkce splněna, automat se posune do dalšího stavu. Dále se při zavolání metody vždy zapíšou aktuální výstupní hodnoty ze sekvence na výstupy PLC.

- **sequenceID : INT** - Volí sekvenci, z kterou chceme, aby automat použil. Počet sekvencí je stanoven na 11, tedy hodnoty tohoto vstupu mohou být 0 až 10. Jedná se o identifikátor sekvence v programu.

9.2.6 goToStart : BOOL

Vrací automaticky robotický manipulátor do výchozí polohy. Není možné řešit rekurzí, proto metoda vrací hodnotu *True*, pokud ještě není ve výchozí poloze, tuto hodnotu je třeba použít k opětovnému zavolání metody.

9.2.7 delSequence

Uvede zvolenou sekvenci do výchozího stavu, sekvence jsou typu proměnných *RETAIN*, což znamená, že nebudou restartem PLC uvedený do výchozích hodnot. Proto je třeba provést přepsání hodnot na nulové hodnoty.

- **sequenceID : INT** - Volí sekvenci, z kterou chceme, aby automat smazal z paměti. Počet sekvencí je stanoven na 11, tedy hodnoty tohoto vstupu mohou být 0 až 10. Jedná se o identifikátor sekvence v programu.

Kapitola 10

Ukládání sekvencí

Tato podvrstva slouží k ukládání dat o naučených sekvencích. Jelikož *Mervis* neumožňuje vytvářet databáze nebo jiné pokročilejší datové struktury, vyřešil jsem ukládání dat pomocí globálních RETAIN proměnných. Tento typ totiž ukládá *Mervis* do flash paměti PLC.

10.1 Sekvence

Díky objektové struktuře obsahuje automat pole několika instancí třídy *Sequence*. Tyto instance mají v sobě definovanou přechodovou funkci, dále výstupní hodnoty a vyčkávací čas pro každý stav. Všechny tyto hodnoty jsou uloženy v polích různých rozměrů. Všechny řádky polí reprezentují jeden stav.

Prvním polem je dvourozměrné pole přechodové funkce. Na řádku tohoto pole jsou uloženy hodnoty vstupů aktuálního stavu. Pokud automat vyhodnotí, že vstupy PLC se rovnají řádku tohoto pole, posune se automat do dalšího stavu. Dalším důležitým dvourozměrným polem je pole obsahující výstupní hodnoty. Automat cyklicky zapisuje hodnoty na výstupy dle aktuálního stavu a cyklicky porovnává přechodovou funkci.

10.2 Funkční blok: Sequence

Název metody	Parametry	Typ	Návratový typ
isLastState	step	INT	BOOL
setLastState	step	INT	-
getWaitingTime	step	INT	TIME
setWaitingTime	step value	INT TIME	-
getTransferValue	step valueArrPosition	INT INT	BOOL
getOutputValue	step valueArrPosition	INT INT	BOOL
setTransferValue	value step valueArrPosition	BOOL INT INT	-
setOutputValue	value step valueArrPosition	BOOL INT INT	-
deleteData	-		-
getSensorEnabled	step	INT	BOOL
enSensor	step value	INT BOOL	BOOL

Tabulka 10.1: Sequence - rozhraní funčního bloku

10.2.1 isLastState : BOOL

Vrací hodnotu *True*, pokud je *stav* posledním stavem sekvence.

- **stav : INT** - Stav stavového automatu. Aktuální hodnota kroku.

10.2.2 setLastState

Nastaví poslední stav sekvence dle hodnoty *stav*

- **stav : INT** - Stav stavového automatu. Aktuální hodnota kroku.

10.2.3 getWaitingTime : TIME

Vrací hodnotu nastaveného vyčkávacího času z pole času, kde řádek matice je roven hodnotě *stav*.

- **stav : INT** - Stav stavového automatu. Aktuální hodnota kroku.

10.2.4 setWaitingTime

Nastaví hodnotu *value* vyčkávacího času do pole času, kde řádek matice je roven hodnotě *stav*.

- **stav : INT** - Stav stavového automatu. Aktuální hodnota kroku.
- **value : TIME** - Hodnota vyčkávacího času

10.2.5 getTransferValue : BOOL

Vrací hodnotu z pole přechodu, kde *stav* je řádek pole a *valueArrPosition* je sloupec.

- **stav : INT** - Stav stavového automatu. Aktuální hodnota kroku.
- **valueArrPosition : INT** - Číslo sloupce pole přechodu. Pořadí hodnot v řádku pole přechodu odpovídá poli vstupních hodnot ve funkčním bloku *PLC*.

10.2.6 setTransferValue

Nastaví hodnotu *value* do pole přechodu, kde *stav* je řádek pole a *valueArrPosition* je sloupec.

- **stav : INT** - Stav stavového automatu. Aktuální hodnota kroku.
- **value : BOOL** - Nastavovaná hodnota, kterou chceme do pole přechodu uložit.
- **valueArrPosition : INT** - Číslo sloupce pole přechodu. Pořadí hodnot v řádku pole přechodu odpovídá poli vstupních hodnot ve funkčním bloku *PLC*.

10.2.7 getOutputValue : BOOL

Varcí hodnotu z pole výstupů, kde *stav* je řádek pole a *valueArrPosition* je sloupec.

- **stav : INT** - Stav stavového automatu. Aktuální hodnota kroku.
- **valueArrPosition : INT** - Číslo sloupce pole výstupů. Pořadí hodnot v řádku pole přechodu odpovídá poli výstupních hodnot ve funkčním bloku *PLC*.

10.2.8 setOutputValue

Nastaví hodnotu *value* do pole výstupů, kde *stav* je řádek pole a *valueArrPosition* je sloupec.

- **stav : INT** - Stav stavového automatu. Aktuální hodnota kroku.
- **value : BOOL** - Nastavovaná hodnota, kterou chceme do pole výstupu uložit.
- **valueArrPosition : INT** - Číslo sloupce pole výstupů. Pořadí hodnot v řádku pole přechodu odpovídá poli výstupních hodnot ve funkčním bloku *PLC*.

10.2.9 deleteData

Vymaže nastavené hodnoty všech polí instance Sekvence.

10.2.10 getSensorEnabled

Vrací hodnotu z pole senzoru, kde je nastaveno, kdy má automat brát hodnotu pro senzor v poli přechodu v potaz.

- **stav : INT** - Stav stavového automatu. Aktuální hodnota kroku.

10.2.11 enSensor

Nastaví hodnotu *value* do pole senzorů, pokud je hodnota v poli nastavena na *True*, pak bude hodnotu senzoru funkční blok automat v daném *stavu* brát v potaz.

- **stav : INT** - Stav stavového automatu. Aktuální hodnota kroku.
- **value : BOOL** - Nastavovaná hodnota, kterou chceme do pole senzoru uložit.

Kapitola 11

Provozní stavy manipulátoru

Tato část programu zajišťuje propojení mezi protokolem *Modbus TCP* a logickou vrstvou programu. Jejím hlavním účelem je řídit chod programu na základě proměnných z tohoto protokolu. Proměnné protokolu Modbus je třeba parsovat, protože neexistuje již připravená mezivrstva, která by převedla komunikaci z protokolu *Modbus* na volání metod objektů, například jako u http protokolu ve frameworku *Spring*, kde velmi jednoduše pomocí Java anotací přiřadíme metody k http požadavkům.

Z tohoto důvodu jsem vytvořil funkční blok kontroléru. Ten má v sobě nadefinované proměnné pro komunikaci prostřednictvím protokolu *Modbus TCP*. Funkční blok *ModbusAutomatKontroler* slouží k řízení logické vrstvy. Jeho hlavní funkcí je převod jednoduchých proměnných typu *BOOL* na volání metod rozhraní automatu a instancí funkčního bloku Motor. Do těchto metod poté vstupují proměnné typu *BOOL*, *INT* a *LREAL*, které se zapisují jako parametry metod. Tabulku, jak jsou namapovány jednotlivé proměnné v Modbus TCP protokolu, naleznete v příloze.

11.1 Stavový stroj

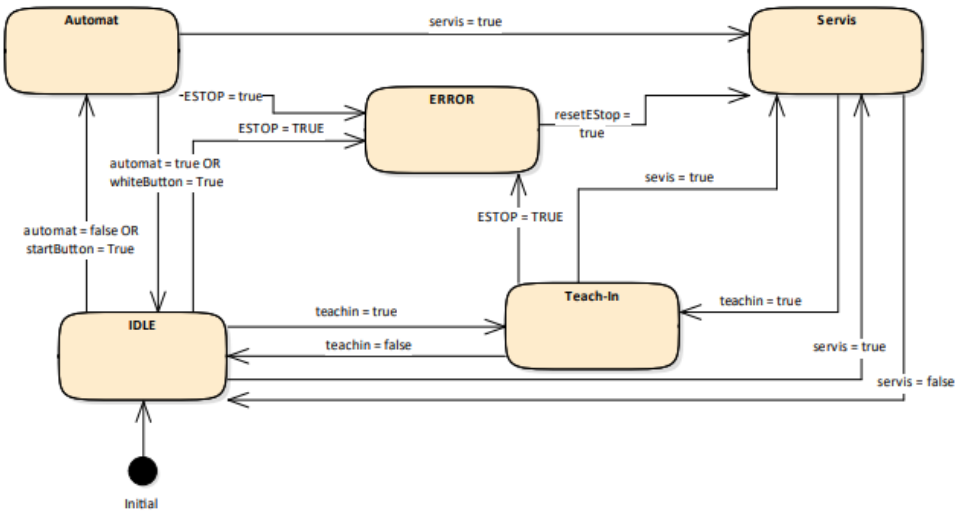
Pneumatický manipulátor je řízen pomocí stavového stroje typu CASE-IF. Používá pět stavů a to *IDLE*, *AUTOMAT*, *TEACH-IN*, *SERVIS* a *ERROR*. K Přechodu mezi stavy slouží proměnné, které jsem vypsal s jejich funkcí v tabulce 11.1. S těmito proměnnými musíme pracovat, abychom se dostali k

funkcím automatu.

Proměnná	Popis funkce
teachin : BOOL	Přepíná do stavu Teach-In
automat : BOOL	Přepíná do stavu Automat
selector : INT	Informace o aktuálním stavu
ESTOP : BOOL	Přepíná do stavu Error
resetEStop : BOOL	Přepíná ze stavu Error do IDLE

Tabulka 11.1: Řídící proměnné stavového automatu

Na obrázku 11.1 vidíte stavový diagram mého řídícího programu pneumatického manipulátoru. Díky použitému stavovému stroji není možné například používat zároveň smazání sekvence a mít spuštěný automat, nebo ovládat motory manuálně a zároveň mít spuštěný automat. Je to základní prvek zabránění chyb, tedy jasně definujeme, které funkce se v daném stavu smí používat.



Obrázek 11.1: Diagram stavového stroje pneumatického manipulátoru

11.1.1 Stav: IDLE

Proměnná	Popis
manMotorInt : INT	Označuje ID motoru (1-7)
manMotorPlus : BOOL	Pohyb motoru (manMotorInt) z pozice 0 na 1
manMotorMinus : BOOL	Pohyb motoru (manMotorInt) z pozice 1 na 0
stepSequence : BOOL	Spustí krokování
cisloKrokuTeach : INT	Řídí stav automatu (0 až 100)
intSekvence : INT	Volí sekvenci (0 až 11)
reset : BOOL	Uvede automat do výchozího stavu

Tabulka 11.2: Řídící proměnné ve stavu IDLE

Ve stavu IDLE je možno provádět tři činnosti na pneumatickém manipulátoru. První z funkcí je manuální řízení pohonů. Pro každý pohyb je nutné používat právě dvě proměnné. První hodnotou, kterou musí řízení znát je, který motor chceme ovládat. Děje se tak pomocí proměnné *manMotorInt*. Ta může nabývat hodnot, dle rozsahu typu proměnné INT. Motory jsou však označeny čísly 1 až 7, pořadí číslování odpovídá pořadí motorů v tabulce 7.1 na straně 51.

Druhou možností, kterou je možné ovládat v režimu IDLE je krokování sekvencí. Pomocí proměnné *stepSequence* aktivujeme funkci krokování. Znamená to, že se budou zapisovat výstupy PLC, dle výstupní funkce zvolené sekvence. Zároveň dochází k zápisu nastavení tohoto kroku do automatu. Při přechodu do stavu Automat, bude automat pokračovat od této nastavené sekvence. Poslední funkcí v tomto stavu je možnost resetování automatu pomocí proměnné *reset*. Ta vynuluje čítač cyklů, nastaví automat na stav 0 a spustí sekvenci pro navrácení robota do výchozí polohy.

11.1.2 Stav: AUTOMAT

Pokud přejde stavový stroj do stavu Automat, pak jedinou povolenou činností je vykonávání zvolené sekvence, kterou je možné měnit pouze v režimu IDLE. Pro spuštěni sekvence není třeba dalších proměnných. Jakmile se totiž stavový stroj nachází ve stavu automat, pak je sekvence právě vykonávána. Důvodem opuštění toho stavu zpět do stavu IDLE může být například dokončení nastaveného počtu cyklů, pozastavení automatu nebo chyba pohonu.

11.1.3 Stav: TEACH-IN

Proměnná	Popis
teach : BOOL	Provede uložení hodnot do sekvence
teachSekvence : INT	Volí číslo sekvence (0 - 10)
cisloKrokuTeach : INT	Číslo stavu zapisovaného stavu
sensorSetValue : BOOL	Učená hodnota senzoru
newTIME : int	Nastavovaná hodnota uloženého času
enableSensor : BOOL	Zapne vyhodnocování senzoru
deleteSequence : BOOL	Smaže jednu sekvenci

Tabulka 11.3: Řídící proměnné ve stavu TEACH-IN

Ve stavu Teach-In se vykonává učící algoritmus. Jeho řídící proměněné jsem vypsal do tabulky 11.3. Jsou zde možné dvě činnosti, a to naučit a smazat sekvenci. Pro naučení jednoho kroku sekvence je potřeba změnit hodnotu proměnně *teach* na hodnotu *True*. Krok, do kterého v sekvenci zapisujeme, nastavuje proměnná *cisloKrokuTeach*. Ten uložíme pomocí proměnné *teachSekvence*. V rámci metody, která se hodnotou *True* na proměnné *teachSekvence* zavolá, dojde také k zapsání hodnot *enableSensor* a *sensorSetValue*. První z těchto proměnných nastavuje, zda má automat v daném kroku hodnotu senzoru vyhodnocovat, a druhá požadovanou hodnotu senzoru, která splní přechodovou funkci.

Druhou funkcí, kterou je možné v tomto stavu spustit, je automatické smazaní celé zvolené sekvence. Tato funkce je důležitá zejména při začátku učení nové sekvence, vymaže totiž veškeré hodnoty ze sekvence. Tuto funkci doporučuji využít před přepsáním celé sekvence.

Převod typu TIME na INT

Tento funkční blok v sobě také obsahuje převod proměnných typu TIME do typu LREAL a naopak. Tento převod je zapotřebí, protože Modbus nepodporuje typ proměnné TIME. Použil jsem k tomu funkci *to_int*, která převádí libovolný datový typ na INT, a funkci *mul_time_lreal*, která násobí hodnotu typu LREAL časovým úsekem. Ten je nastavený- na hodnotu 100 ms.

11.1.4 Stav: SERVIS

Proměnná	Popis
manMotorInt : INT	Označuje ID motoru (1-7)
manMotorPlus : BOOL	Pohyb motoru (manMotorInt) z 0 na 1
manMotorMinus : BOOL	Pohyb motoru (manMotorInt) z 1 na 0
resetAllMotors : BOOL	Provede vynulování chyb všech motorů
disableAnticollision : BOOL	Vypne antikolizní systém
setWatchdogTime : BOOL	Uloží všechny nastavené časy watchdogů
setWatchdogTimeA : LREAL	Watchdog čas morotu A
setWatchdogTimeB : LREAL	Watchdog čas morotu B
setWatchdogTimeC : LREAL	Watchdog čas morotu C
setWatchdogTimeD : LREAL	Watchdog čas morotu D
deleteAllSequences : BOOL	Smaže všechny sekvence

Tabulka 11.4: Řídící proměnné ve stavu SERVIS

Stejně tak jako ve stavu *IDLE* je možné pomocí proměnných *mamMotorInt*, *manMotorPlus* a *manMotorMinus* manuálně ovládat motory. Ve stavu *SERVIS* je možné navíc vypnout antikolizní funkci všech motorů. Deaktivace této funkce se provádí pomocí hodnoty *True* na proměnné *disableAnticollision*. Tím se nejen umožní motory pohybovat zakázanými směry, ale umožní i to pohyb motorem, který watchdog motorů zablokoval. Blokování motorů je možné resetovat pomocí proměnné *resetAllMotors*.

Dále je možné nastavit časy watchdogů motorů. Watchdog slouží k vyvolání chyby, pokud uběhne tento nastavený čas od změny výstupní hodnoty PLC po aktivaci senzoru požadované koncové polohy. Uložení hodnot se provádí pomocí proměnné *setWatchdogTime*.

11.1.5 Stav: ERROR

Do stavu *ERROR* se stavový stroj dostane, pokud je stisknuto tlačítko STOP na rozvodné skříni pneumatického manipulátor, nebo na obrazovce v grafickém rozhraní SCADA. Z tohoto stavu vede jediná možnost přechodu do servisního stavu. V tomto stavu se zablokují všechny funkce pneumatického manipulátoru.

11.2 Výstupní komunikační proměnné

V následující tabulce 11.5 jsem vypsal zbylé proměnné, které jsou výstupními proměnnými programu a jejich funkce je pouze informativní.

Proměnná	Popis
teachedA1 : BOOL	Hodnota přechodové funkce pro senzor A1
teachedA0 : BOOL	Hodnota přechodové funkce pro senzor A0
teachedB1 : BOOL	Hodnota přechodové funkce pro senzor B1
teachedB0 : BOOL	Hodnota přechodové funkce pro senzor B0
teachedC1 : BOOL	Hodnota přechodové funkce pro senzor C1
teachedC0 : BOOL	Hodnota přechodové funkce pro senzor C0
teachedD1 : BOOL	Hodnota přechodové funkce pro senzor D1
teachedD0 : BOOL	Hodnota přechodové funkce pro senzor D0
teachedSensorValue : BOOL	Hodnota přechodové funkce pro senzor kuličky
teachedOutA : BOOL	Hodnota výstupní funkce pro motor A
teachedOutB : BOOL	Hodnota výstupní funkce pro motor B
teachedOutC : BOOL	Hodnota výstupní funkce pro motor C
teachedOutD : BOOL	Hodnota výstupní funkce pro motor D
teachedOutE : BOOL	Hodnota výstupní funkce pro motor E
teachedOutF1 : BOOL	Hodnota výstupní funkce pro motor F1
teachedOutF2 : BOOL	Hodnota výstupní funkce pro motor F2
timeINT : lreal	Hodnota uloženého vyčkávacího času sekvence
actualState : INT	Číslo aktuálního stavu automatu
lastState : INT	Číslo posledního stavu sekvence
motorAsenzorplus : BOOL	Hodnota senzoru A1
motorAsenzorminus : BOOL	Hodnota senzoru A0
motorBsenzorplus : BOOL	Hodnota senzoru B1
motorBsenzorminus : BOOL	Hodnota senzoru B0
motorCsenzorplus : BOOL	Hodnota senzoru C1
motorCsenzorminus : BOOL	Hodnota senzoru C0
motorDsenzorplus : BOOL	Hodnota senzoru D1
motorDsenzorminus : BOOL	Hodnota senzoru D0
motorAerror : BOOL	Chyba motoru A
motorBerror : BOOL	Chyba motoru B
motorCerror : BOOL	Chyba motoru C
motorDerror : BOOL	Chyba motoru D
motorApovol : BOOL	Povolený pohyb motoru A
motorBpovol : BOOL	Povolený pohyb motoru B
motorCpovol : BOOL	Povolený pohyb motoru C
motorDpovol : BOOL	Povolený pohyb motoru D

Tabulka 11.5: Výstupní proměnné programu PLC

Kapitola 12

Grafické rozhraní - SCADA

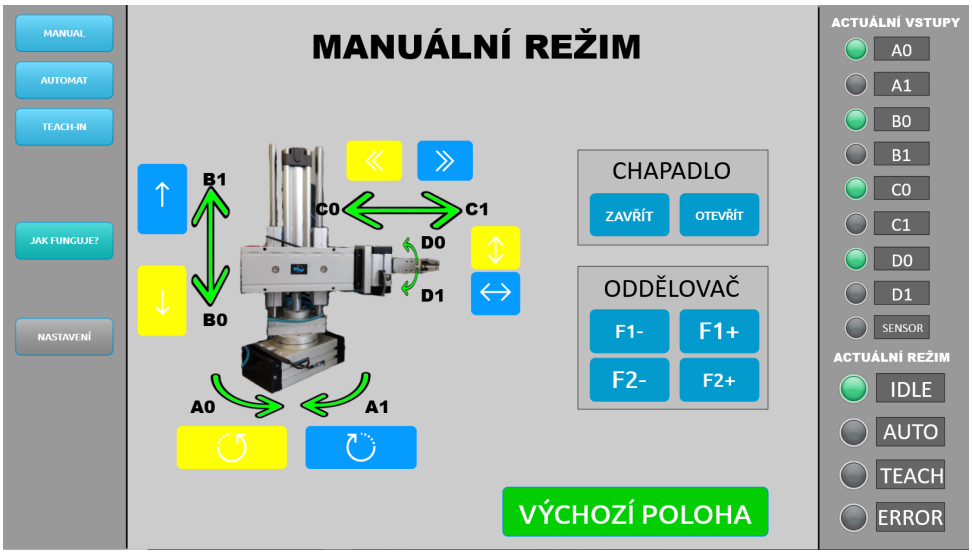
Pro grafické rozhraní pneumatického manipulátoru jsem zvolil software myS-CADA. Jedná se o plně webový SCADA systém, který lze velmi snadno vytvářet pomocí aplikace myDESIGNER. Základní funkcí SCADA systému je zajišťovat sběr dat z řídící vrstvy a zajišťovat dispečerské řízení. Základní stavební jednotkou SCADA systému je pohled. V těchto pohledech vytváříme grafické rozhraní pro uživatele. V případě průmyslových systémů jsou to například grafická znázornění výrobního procesu nebo jeho úseku.

12.1 Rozložení obrazovky

Rozložení grafického rozhraní pneumatického manipulátoru jsem rozdělil na tři sloupce. V levém sloupci se nachází hlavní menu pro přechod do konkrétního režimu řízení. Je to režim manuální, automatický a teach-in režim.

V prostřední části se nachází aktuální pohled a v levé části jsou informativní kontrolky. Tyto kontrolky zobrazují aktuální hodnoty vstupů PLC, konkrétně koncových senzorů a indukčního snímače před oddělovačem kuliček. V dolní části poté je indikován aktuální stav stavového stroje, který řídí pneumatický manipulátor.

12.2 Pohled: Manuální režim



Obrázek 12.1: SCADA - Pohled: Manuální režim

Na tomto pohledu je vlevo obrázek pneumatického manipulátoru se znázorněním směru pohybu. U každé šipky, která tento pohyb znázorňuje, se nachází označení koncové polohy. U každé z těchto šipek je tlačítko, které spouští pohyb v daném směru. Pokud je stav motoru aktivní, pak tlačítko, které spouští pohyb do tohoto stavu, má žlutou barvu. Tlačítko, které můžeme použít pro změnu polohy má vždy modrou barvu. A to bez ohledu na to, zda je pohyb kolizní či nikoli. V případě, že pohyb je nepovolený (tedy kolizní), vyskočí po stisknutí příslušného tlačítka hláška, která uživatele upozorní na to, že zvolil kolizní pohyb. Kolizní pohyby není na tomto pohledu možné vykonávat.

V pravé části pohledu se nacházejí tlačítka pro ovládání chapače a oddělovače kuliček. Všechny tři tyto pneumatické pohony nejsou osazeny snímači koncových poloh, i když to umožňují. Nejsou však pro řízení pneumatického manipulátoru potřeba, protože nemají vliv na kolizní funkce. Posledním tlačítkem na tomto pohledu je tlačítko výchozí poloha. To spustí sekvenci, kdy se manipulátor vrátí do výchozí polohy tak, jako na obrázku vlevo.

12.3 Pohled: Automatický režim

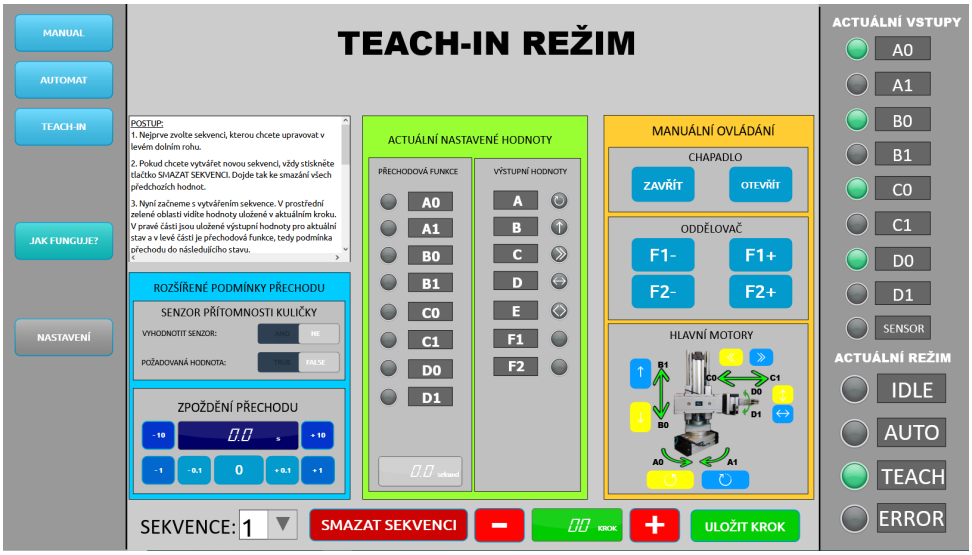
Obrázek 12.2: SCADA - Pohled: Automatický režim

V pohledu automatického režimu se pomocí rolovacího menu zvolí sekvence, kterou chceme, aby automat prováděl. Dále je možné nastavit požadovaný počet cyklů a krokovat sekvenci. To vše se smí odehrávat pouze pokud se stavový automat nachází ve stavu *IDLE*. Také se zde zobrazují informace o aktuálním kroku, v kterém se automat nachází. Dále se zde zobrazuje celkový počet kroků zvolené sekvence a počítadlo dokončených cyklů.

V dolní části nalezneme tři tlačítka. Zleva je tlačítko RESET, které vyresetuje celý automat, vynuluje počítač cyklů a vrátí manipulátor do výchozí polohy. Reset automatu je možný pouze v režimu *IDLE*. Dalším tlačítkem je START/PAUZA automatu. Když se toto tlačítko stiskne, stavový stroj se přepne do režimu automat. Opětovným stiskem se automat pozastaví a stroj se přepne do stavu *IDLE*. Také pokud dojde počítadlo cyklů do nastavené hodnoty, pak se automat sám vypne a stroj přejde do stavu *IDLE*.

Posledním tlačítkem je tlačítko E-STOP. To přepne celý stoj do stavu ERROR. Z tohoto stavu se dokáže stroj dostat pouze v případě, když se přihlásí uživatel s právy k pohledu SERVISNÍ REŽIM a zresetuje tento stav. Toto tlačítko má naprosto stejnou funkci jako tlačítko na rozvodné skříni manipulátoru.

12.4 Pohled: Teach-In režim



Obrázek 12.3: SCADA - Pohled: Teach-In režim

Jakmile uživatel přepne na pohled TEACH-IN, stavový stroj se do tohoto režimu rovněž přepne. Tedy vypne se automat nebo například SERVISNÍ REŽIM. Jedná se o nejsložitější pohled, a proto jsem uživateli do levého horního rohu umístil textový návod, jak tento režim používat.

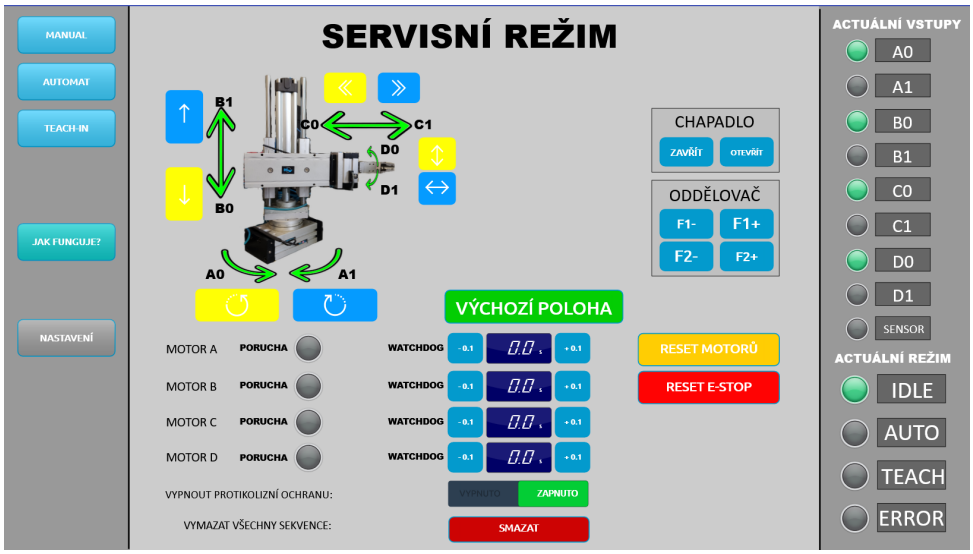
Nejdříve musí uživatel zvolit číslo sekvence, kterou bude učit chování manipulátoru. Pokud je sekvence obsazená, doporučuje se uživateli provést smazání sekvence. Uživatel rovněž může procházet zvolenou sekvenci nebo ji upravovat. Nastavené hodnoty pak vidí v prostřední části AKTUÁLNÍ NASTAVENÉ HODNOTY, tam se mu vždy zobrazí to, co uložil v daném kroku. Vpravo vidí, jaký stav budou mít jednotlivé pohony v daném kroku do té doby, než se splní PŘECHODOVÁ FUNKCE a automat přejde do dalšího kroku v pořadí.

Samotné učení probíhá tak, že uživatel nastaví polohu manipulátoru do takové polohy, v jaké ho chce mít a stiskne tlačítko ULOŽIT KROK. Tím se uloží aktuální stav pneumatického manipulátoru a automaticky se krok posune o jedničku vpřed. Uživatel opět zadá další konfiguraci motorů pomocí bloku MANUÁLNÍHO OVLÁDÁNÍ a opět uloží krok.

V každém kroku je možno rozšířit přechodovou funkci ZPOŽDĚNÍ PŘECHODU, jehož odpočet se vždy spustí po splnění přechodové funkce a zpozdí

tak posun automatu do dalšího kroku. Také se zde dá přidat podmínka indukčního snímače kuličky. Pro aktivaci této podmínky v přechodové funkci je třeba změnit přepínač VYHODNOTIT SENZOR do polohy ANO a poté nastavit požadovanou hodnotu vstupu ze senzoru. Pokud bude mít senzor právě hodnotu TRUE a automat požaduje, aby měl hodnotu FALSE, bude automat vyčkávat na správnou hodnotu.

12.5 Pohled: Servisní režim



Obrázek 12.4: SCADA - Pohled: Servisní režim

SERVISNÍ REŽIM slouží k ovládání pneumatického manipulátoru tak, že se dají vypnout ochranné funkce jako je antikolizní systém. V horní části se nachází stejná tlačítka jako v MANUÁLNÍ REŽIMU, ale v dolní části se nachází nastavení. Důležitým přepínačem je vypnutí antikolizní ochrany, které umožňuje pohybovat s motory bez ohledu na koncové senzory. Dále se na tomto pohledu nachází nastavení časů watchdogů pro každý motor, pokud to motor umožňuje. Hodnoty jsou uloženy ve flash paměti PLC, a tedy zůstanou nastaveny i po restartu stejně tak jako naučené sekvence. Ty se dají smazat všechny najednou tlačítkem SMAZAT.

Poslední dvě tlačítka jsou RESET MOTORŮ a RESET E-STOP. Prvním tlačítkem vymaže uživatel uvnitř motorů chybový stav. Pokud má motor chybový stav, ukáže se mu tato skutečnost na kontrolkách vlevo vedle názvu motoru. Stisknutím druhého tlačítka přepne uživatel manipulátor do stavu

SERVIS, a tedy ho odblokuje po stisknutí tlačítka ESTOP.

Kapitola 13

Závěr

Ve své diplomové práci jsem měl za úkol vytvořit řízení s možností Teach-in. To jsem měl aplikovat pro pneumatický manipulátor, který slouží jako učební pomůcka v laboratoři programovatelných automatů. Teach-in je způsob učení pomocí odkrokování sekvence uživatelem, proto bylo potřeba se nejdříve zaměřit na způsoby programování sekvenčních úloh.

Popsal jsem proto nejdříve fungování obecného deterministického konečného automatu. Dále jsem tento automat rozšířil o pojem stroj v podobě Mealyho a Moorova sekvenčního stroje. Zaměřil jsem také na taktovací řetězce, které byly aktuální spíše v minulosti, kdy nebyly programovatelné logické kontroléry tak dostupné. Používalo se často reléového řízení. Přesto taktovací řetězce naleznou uplatnění v softwarové podobě i dnes. Zejména při programování pomocí jazyka LD (Ledder Diagram), který je součástí normy *IEC 61131-3*. V případě jazyka ST (Structured Text) je více využívaný CASE-IF stavový stroj.

V práci jsem se zaměřil také na možnosti otevřeného softwaru z hlediska programovatelných logických automatů. Nejdříve jsem popsal pojem otevřeného softwaru, jeho výhody a také jeho nevýhody. Zaměřil jsem se zejména na dvě řady produktů značky *Unipi*. Jedná se o programovatelné logické automaty, které nabízejí uživateli otevřený operační systém *Linux* a přístup do něj dávají zcela k dispozici. Tímto způsobem nabízejí třetím stranám možnost vývoje vlastního softwaru, který bude automat řídit. Poskytují také ovladače pro řízení vstupů a výstupů jejich PLC. K dispozici jsou také další možnosti vzdáleného řízení PLC například pomocí REST API.

V rámci programování jazyky dle normy *IEC 61131-3* je standardem spíše strukturované paradigma nebo grafické programování. Aktualizací normy byla do jazyka ST přidána možnost objektově orientovaného programování. V této práci jsem prozkoumal tyto možnosti a porovnal je s jazykem Java. Toto paradigma jsem poté použil ve vlastním řešení řízení pneumatického manipulátoru této diplomové práce.

Algoritmus automatu, který jsem ve své práci vytvořil, implementuje Mealyho sekvenční stroj. Díky fuknci Teach-in je možné naučit automat až jedenácti sekvencí, které jsou uloženy v paměti PLC a nevymažou se po restartu zařízení. Každá sekvence má maximální délku sto kroků a umožňuje například časové zpoždění přechodu do dalšího stavu. Logika obsluhující jednotlivé motory v sobě obsahuje algoritmus pro zamezení kolize. Dalšími funkcemi, které jsem implementoval, je například automatický návrat manipulátoru do výchozí polohy nebo vyhodnocování poruch motorů.

Pro vzdálené řízení uživatelem jsem vytvořil grafické rozhraní pomocí softwaru mySCADA. To zajišťuje pohodlné ovládání uživatelem manipulátoru a také slouží k učení sekvence pomocí Teach-in algoritmu. SCADA systém pomocí Modbus TCP vzdáleně ovládá funkce manipulátoru a řídí stavy CASE-IF stavového stroje použitého pro řízení chodu manipulátoru. Svůj software jsem podrobil testování a manipulátor je plně funkční.

Přílohy

Příloha A

Literatura

[1] M. Martinásková, "Základy elektropneumatiky - cvičení," 2020. [Online]. Available: https://moodle-vyuka.cvut.cz/pluginfile.php/216753/mod_resource/content/2/EP_z%C3%A1kladn%C3%AD_cvi%C4%8Den%C3%AD_V11_bF.pdf

[2] "Unipi Neuron S103 (Raspberry Pi 4) | Unipi." [Online]. Available: https://www.unipi.technology/cs/unipi-neuron-s103-raspberry-pi-4-p369

[3] "Unipi Axon | Unipi." [Online]. Available: https://www.unipi.technology/products/unipi-axon-135?categoryId=13&categorySlug=unipi-axon

[4] "What is the Automation Pyramid? | RealPars," Jun. 2018. [Online]. Available: https://realpars.com/automation-pyramid/

[5] R. Orlita, "Inovace modulu robota," Bakalářská práce, ČVUT v Praze, Fakulta strojní, Praha, 2011.

[6] P. Zítek, J. Hlava, M. Hofreiter, České vysoké učení technické v Praze, and Strojní fakulta, *Automatické řízení*. Praha: ČVUT, 1999, oCLC: 320375703.

[7] M. Chytil, *Teorie automatů a formálních jazyků*, 1st ed. Praha: Státní pedagogické nakladatelství, 1978, no. Book, Whole.

[8] J. Kolář, České vysoké učení technické v Praze, and Elektrotechnická fakulta, *Teoretická informatika*. V Praze: České vysoké učení technické, 2009, oCLC: 436260595.

[9] "PLC programming (1): Starting with open source software | Unipi." [Online]. Available: https://www.unipi.technology/key_feature/plc-programming-1-starting-with-open-source-software-348

[10] "Unipi Patron | Unipi." [Online]. Available: https://www.unipi.technology/products/unipi-patron-374?categoryId=30&categorySlug=unipi-patron

[11] "Unipi.technology Knowledge Base," 2018. [Online]. Available: https://kb.unipi.technology/en:00-start?redirect=1

[12] "Sysfs - brána do jadra." [Online]. Available: http://www.abclinuxu.cz/clanky/system/sysfs-brana-do-jadra

[13] "API - Application Programming Interface | Unipi." [Online]. Available: https://www.unipi.technology/products/api-application-programming-interface-341?categoryId=6&categorySlug=software

[14] D. Hanák, "Stopařův průvodce REST API." [Online]. Available: https://www.itnetwork.cz/stoparuv-pruvodce-rest-api

[15] "PYPL PopularitY of Programming Language index." [Online]. Available: https://pypl.github.io/PYPL.html

[16] J. Chyský, České vysoké učení technické v Praze, and Strojní fakulta, *Vestavěné systémy I.* V Praze: České vysoké učení technické, 2010, oCLC: 693939181.

[17] D. Čápka, "Objektově orientované programování v Javě." [Online]. Available: https://help.codesys.com/api-content/2/codesys/3.5.13.0/en/_cds_obj_method/

[18] "8051 Microcontroller Assembly Language Programming," Nov. 2017. [Online]. Available: https://www.electronicshub.org/8051-microcontroller-assembly-language-programming/

[19] "Beckhoff Information System - English." [Online]. Available: https://infosys.beckhoff.com/english.php?content=../content/1033/tcplccontrol/html/TcPlcCtrl_Languages%20IL.htm

[20] R. Pecinovský, *Java 14: kompletní příručka jazyka*, 2020, oCLC: 1200258035.

[21] gcx11, "Lekce 1 - Úvod do objektově orientovaného programování v Pythonu." [Online]. Available: https://www.itnetwork.cz/python-tutorial-uvod-do-objektove-orientovaneho-programovani

[22] "The World of Mervis - Knowledge Base." [Online]. Available: https://kb.mervis.info/doku.php/en:mervis-ide:35-help:6-software_basic:7-oop

[23] "Java Variables." [Online]. Available: https://www.w3schools.com/java/java_variables.asp

[24] "The World of Mervis - Knowledge Base." [Online]. Available: https://kb.mervis.info/doku.php/en:mervis-ide:35-help:6-software_basic:7-oop

[25] "Overloading in Java - GeeksforGeeks." [Online]. Available: https://www.geeksforgeeks.org/overloading-in-java/

[26] S. User, "MES systém (Manufacturing Execution System)." [Online]. Available: http://www.mescenter.org/cz/clanky/5-co-je-to-mes-system

Příloha B

Seznam použitých zkratek

OOP	Objektově orientované programování - programovací paradigma
PLC	Programovatelný logický automat
ALU	Aritmeticko-logická jednotka - část procesoru
LD	Ledder diagram - grafický jazyk dle normy IEC 61131-3
ST	Structured text - textový jazyk dle normy IEC 61131-3
IL	Instruction list - textový jazyk dle normy IEC 61131-3
SSH	Secure Shell - zabezpečený způsob vzdáleného ovládání počítače pomocí příkazového řádku
IEC	International Electrotechnical Commission - komise, která vydává elektrotechnické normy, rovněž označení těchto norem
IDE	Integrated Development Environment - Vývojové prostředí softwaru
SBC	Single Board Computer - jednodeskový počítač
RAM	Random Acccess Memory - volativní paměť počítače
eMMC	embedded Multimedia Memory Card - vestavěná paměťová karta
RTU	Remote Terminal Unit - zařízení řízené mikroprocesorem, označení protokolu Modbus běžící na sériové lince
API	Aplication Programming Interface - rozhraní pro programování aplikací, používá se pro integraci
REST	Representational state transfer - softwarová architektura, která se typicky používá pro servery
RISC	Reduced Instruction Set Computer - označuje procesory s redukovanou instrukční sadou
CISC	Complex Instruction Set Computer - označuje procesory s širší instrukční sadou
JSON	JavaScript Object Notation - způsob přenosu objektů v rámci http protokolu
TCP	Transmission Control Protocol - protokol transportní vrstvy ISO/OSI modelu
SSL	Secure Sockets Layer - Zabezpečení komunikace mezi serverem
SOAP	Simple Object Access Protocol - Protokol výměny zpráv pomocí XML

Příloha C

Seznam použitého softwaru

- Mervis IDE 2.4.0
- myDESIGNER 8.14.2
- Enterprise Architect 14
- Inkscape 1.0.1
- SkyCAD Electrical

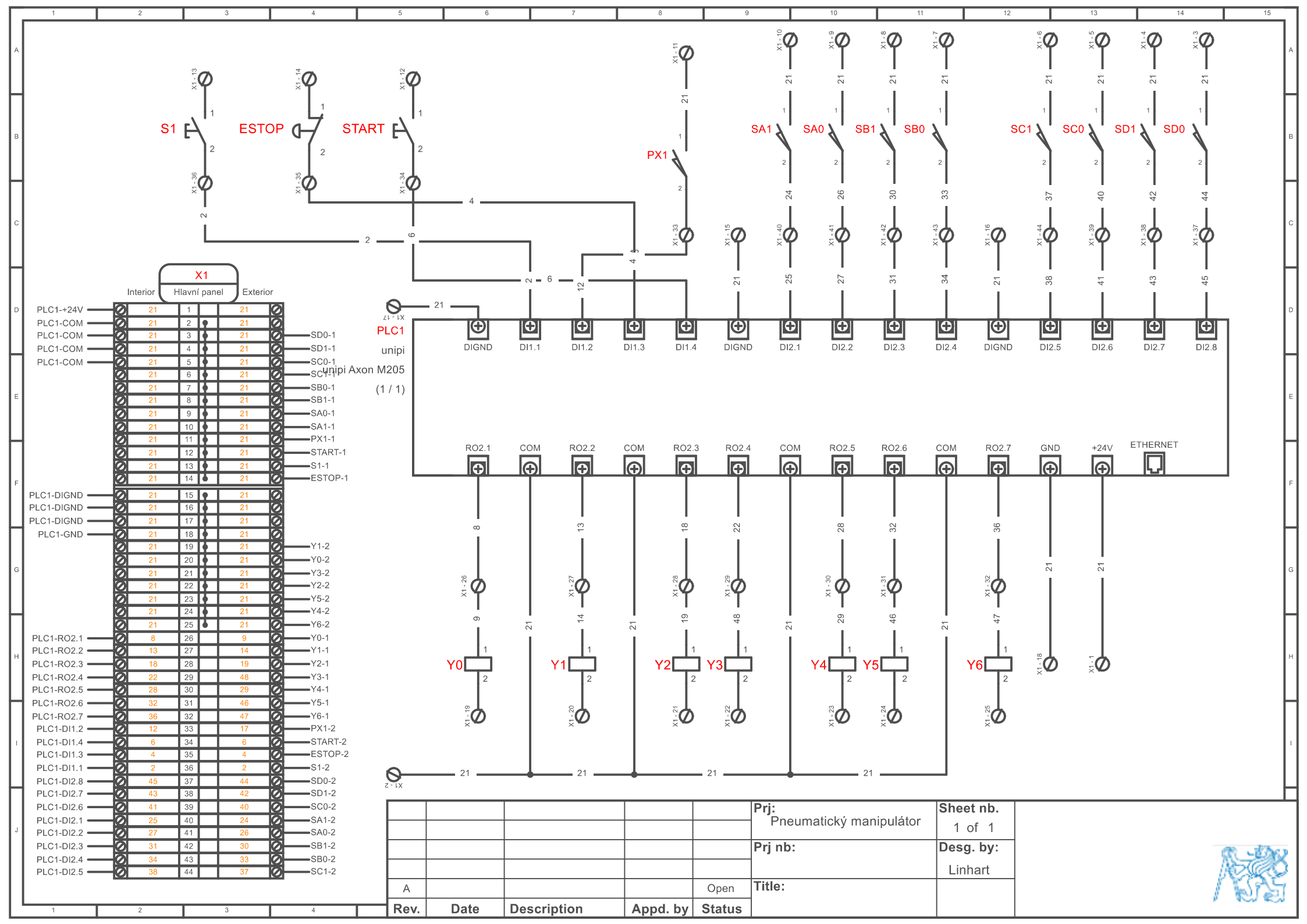
Příloha D

Obsah přiloženého CD

- Diplomová práce v elektronické podobě
- Zdrojový kód pro PLC - Projekt pro Mervis IDE
- Zdrojový kód pro GUI - Projekt pro myDESIGNER

Příloha E

Výkres zapojení

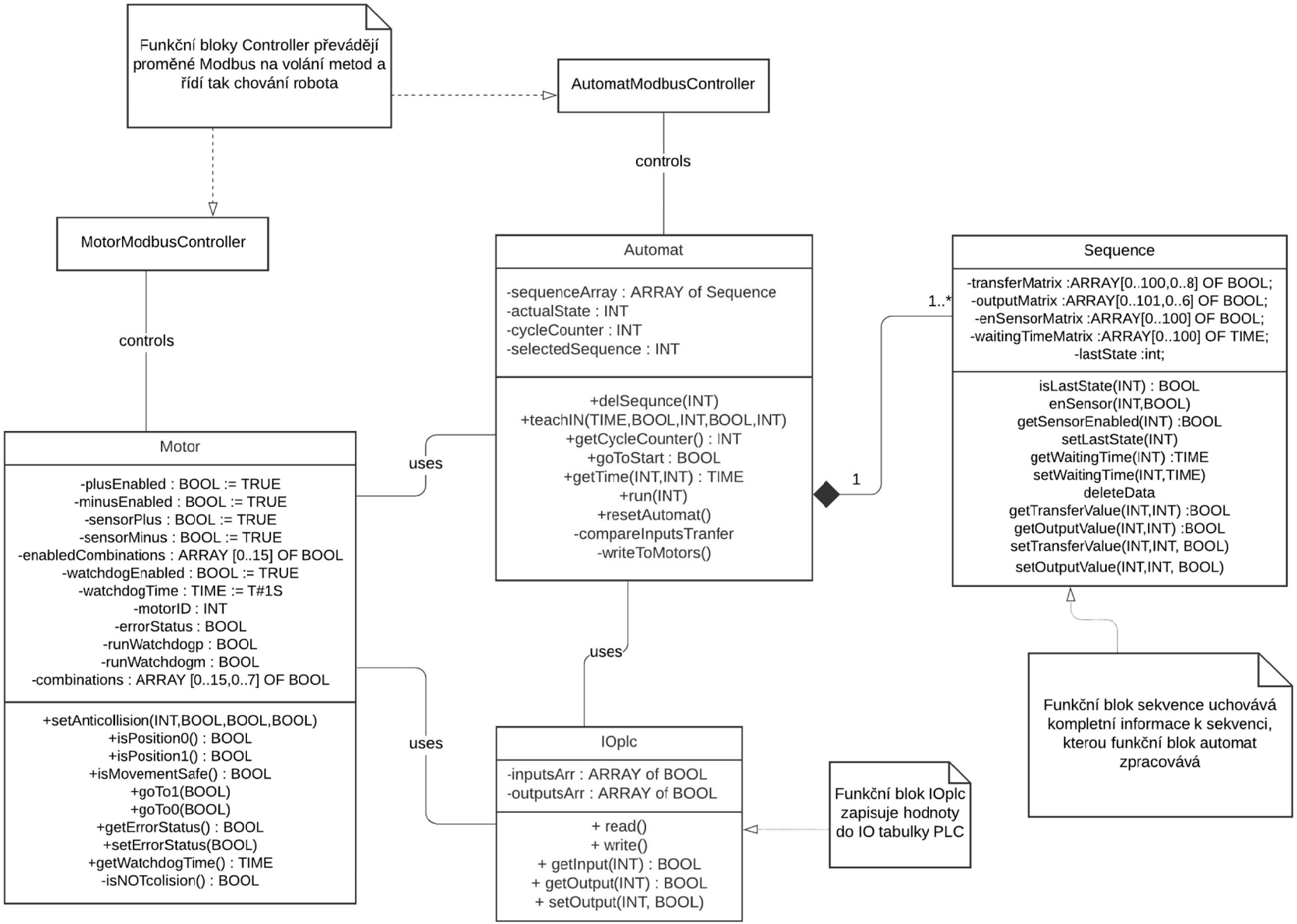


					Prj: Pneumatický manipulátor	Sheet nb. 1 of 1
					Prj nb:	Desg. by: Linhart
A				Open	Title:	
Rev.	Date	Description	Appd. by	Status		

Příloha F

Diagram tříd (Funkčních bloků)

Diagram funkčních bloků progamu PLC



Příloha G

Modbus Registry

ID	Typ registru	Napojená proměnná	Typ
0	Holding register	hlavniCyklus.modbusAutomat.$manMotorInt$	int
1	Holding register	hlavniCyklus.modbusAutomat.$intSekvence$	int
2	Holding register	hlavniCyklus.modbusAutomat.$cisloKrokuTeach$	int
3	Input register	hlavniCyklus.modbusAutomat.$cycleCounter$	int
4	Holding register	hlavniCyklus.modbusAutomat.$newTIME$	int
5	Holding register	hlavniCyklus.modbusAutomat.$timeINT$	lreal
6	Holding register	hlavniCyklus.modbusAutomat.$teachSekvence$	int
7	Holding register	hlavniCyklus.modbusAutomat.$newWatchdogTimeA$	lreal
8	Holding register	hlavniCyklus.modbusAutomat.$newWatchdogTimeB$	lreal
9	Holding register	hlavniCyklus.modbusAutomat.$newWatchdogTimeC$	lreal
10	Holding register	hlavniCyklus.modbusAutomat.$newWatchdogTimeD$	lreal
11	Input register	hlavniCyklus.modbusAutomat.$setWatchdogTimeA$	lreal
12	Input register	hlavniCyklus.modbusAutomat.$setWatchdogTimeB$	lreal
13	Input register	hlavniCyklus.modbusAutomat.$setWatchdogTimeC$	lreal
14	Input register	hlavniCyklus.modbusAutomat.$setWatchdogTimeD$	lreal
15	Input register	hlavniCyklus.modbusAutomat.$actualState$	int
16	Input register	hlavniCyklus.modbusAutomat.$lastState$	int
17	Holding register	hlavniCyklus.modbusAutomat.$setCycle$	int
18	Input register	hlavniCyklus.modbusAutomat.$selector$	int

Příloha H

Modbus diskrétní proměnné

ID	Typ registru	Napojená proměnná
0	Coil	hlavniCyklus.modbusAutomat.$manMotorPlus$
1	Coil	hlavniCyklus.modbusAutomat.$manMotorMinus$
2	Vstup	hlavniCyklus.modbusAutomat.$motorApovol$
4	Vstup	hlavniCyklus.modbusAutomat.$motorBpovol$
6	Vstup	hlavniCyklus.modbusAutomat.$motorCpovol$
8	Vstup	hlavniCyklus.modbusAutomat.$motorDpovol$
10	Vstup	hlavniCyklus.modbusAutomat.$motorAsenzorplus$
11	Vstup	hlavniCyklus.modbusAutomat.$motorAsenzorminus$
12	Vstup	hlavniCyklus.modbusAutomat.$motorBsenzorplus$
13	Vstup	hlavniCyklus.modbusAutomat.$motorBsenzorminus$
14	Vstup	hlavniCyklus.modbusAutomat.$motorCsenzorplus$
15	Vstup	hlavniCyklus.modbusAutomat.$motorCsenzorminus$
16	Vstup	hlavniCyklus.modbusAutomat.$motorDsenzorplus$
17	Vstup	hlavniCyklus.modbusAutomat.$motorDsenzorminus$
18	Coil	hlavniCyklus.modbusAutomat.$teach$
19	Coil	hlavniCyklus.modbusAutomat.$automat$
20	Coil	hlavniCyklus.modbusAutomat.$enableSensor$
21	Coil	hlavniCyklus.modbusAutomat.$reset$
22	Vstup	hlavniCyklus.modbusAutomat.$motorAerror$
23	Vstup	hlavniCyklus.modbusAutomat.$motorBerror$
24	Vstup	hlavniCyklus.modbusAutomat.$motorCerror$
25	Vstup	hlavniCyklus.modbusAutomat.$motorDerror$
26	Coil	hlavniCyklus.modbusAutomat.$goToStart$
27	Coil	hlavniCyklus.modbusAutomat.$sensorSetValue$
28	Coil	hlavniCyklus.modbusAutomat.$deleteSequence$
29	Vstup	hlavniCyklus.modbusAutomat.$ballSensor$
30	Coil	hlavniCyklus.modbusAutomat.$stepSequence$
31	Vstup	hlavniCyklus.modbusAutomat.$teachedA0$
32	Vstup	hlavniCyklus.modbusAutomat.$teachedA1$
33	Vstup	hlavniCyklus.modbusAutomat.$teachedB0$
34	Vstup	hlavniCyklus.modbusAutomat.$teachedB1$
35	Vstup	hlavniCyklus.modbusAutomat.$teachedC0$
36	Vstup	hlavniCyklus.modbusAutomat.$teachedC1$
37	Vstup	hlavniCyklus.modbusAutomat.$teachedD0$
38	Vstup	hlavniCyklus.modbusAutomat.$teachedD1$
39	Vstup	hlavniCyklus.modbusAutomat.$teachedSensorValue$
40	Vstup	hlavniCyklus.modbusAutomat.$teachedOutA$
41	Vstup	hlavniCyklus.modbusAutomat.$teachedOutB$
42	Vstup	hlavniCyklus.modbusAutomat.$teachedOutC$
43	Vstup	hlavniCyklus.modbusAutomat.$teachedOutD$
44	Vstup	hlavniCyklus.modbusAutomat.$teachedOutE$
45	Vstup	hlavniCyklus.modbusAutomat.$teachedOutF1$
46	Vstup	hlavniCyklus.modbusAutomat.$teachedOutF2$

47	Coil	hlavniCyklus.modbusAutomat.*disableAnticolision*
48	Coil	hlavniCyklus.modbusAutomat.*deleteAllSequences*
49	Coil	hlavniCyklus.modbusAutomat.*resetAllMotors*
50	Coil	hlavniCyklus.modbusAutomat.*setWatchdogTime*
51	Coil	hlavniCyklus.modbusAutomat.*ESTOP*
52	Coil	hlavniCyklus.modbusAutomat.*teachin*
53	Coil	hlavniCyklus.modbusAutomat.*resetEStop*
54	Coil	hlavniCyklus.modbusAutomat.*servis*

Příloha I

Kód pro PLC

```
INTERFACE AutomatInterface
	METHOD delSequence
		VAR_INPUT
			sequenceID : int;
		END_VAR
	END_METHOD
	METHOD getLastState : INT
	VAR_INPUT
		sequenceID : INT;
	END_VAR
	END_METHOD
	METHOD teachIN
			VAR_INPUT
			waitingTime : TIME;
			sensorSetVal : BOOL;
			sequenceID : INT;
			enSensor : BOOL;
			teachInStep : int;
		END_VAR
	END_METHOD
	METHOD getCycleCounter : INT
	END_METHOD
	METHOD goToStart : BOOL
	END_METHOD
	METHOD getTime : TIME
	VAR_INPUT
			step : int;
			sequenceID : INT;
	END_VAR
	END_METHOD
	METHOD setActualState
	VAR_INPUT
		step : int;
	END_VAR
	END_METHOD
	METHOD run
	 VAR_INPUT
			sequenceID : INT;
		END_VAR
	END_METHOD
	METHOD resetAutomat
	END_METHOD
	METHOD writeToMotors
		VAR_INPUT
			seqID : INT;
			actualStep : INT;
		END_VAR
	END_METHOD
	METHOD getTeachedTransferValue : BOOL
		VAR_INPUT
			sequenceID : Int;
			step : int;
			valueArrPosition : int;
		END_VAR
	END_METHOD
	METHOD getTeachedOutputValue : BOOL
		VAR_INPUT
			sequenceID : Int;
			step : int;
			valueArrPosition : int;
```

```
		END_VAR
	END_METHOD

	METHOD getActualState : INT
	END_METHOD

END_INTERFACE

VAR_GLOBAL RETAIN //Flash variables
		sequenceArray : ARRAY [0..10] OF Sequence; //Sequence array, size can be changed for increase capacity
of sequences
END_VAR

FUNCTION_BLOCK Automat IMPLEMENTS AutomatInterface
	VAR
		actualState : int; //automat state number, critical value which controls automat step
		cycleCounter : INT; //stores how many cycles were finished by automat
		sequenceSelected : INT; //selects sequenceArray position > that means which sequence to use for
automat
		selector :INT;
	END_VAR
	METHOD resetAutomat
	(* Provede vynulování čítače cyklů, dále vrátí automat do výchozího stavu. *)
		actualState := 0;
		cycleCounter := 0;

	END_METHOD
	METHOD teachIN
	(*Metoda Teach-In algoritmu, provádí uložení stavu a přechodu pro jeden stav
	sekvence zvolené pomocí proměnné teachSequence.
	Provádí zápis všech vstupů a výstupů pro aktuální stav
	a následujících vstupních proměnných metody.*)
		VAR_INPUT
			waitingTime : TIME;
			sensorSetVal : BOOL;
			sequenceID : INT;
			enSensor : BOOL;
			teachInStep : int; //indicates number for saving actual row to sequence
		END_VAR
		VAR
			valueArrPosition : int;
			valueArrPosition2 : int;
		END_VAR
		IF sequenceID >= 0 and sequenceID < 101 and teachInStep >= 0 and teachInStep < 101 THEN
			FOR valueArrPosition := 0 TO 7 DO
				sequenceArray[sequenceID].setTransferValue(step :=teachInStep, valueArrPosition :=
valueArrPosition, value := hardware.IOplc.getInput(valueArrPosition := valueArrPosition));
			END_FOR;
			FOR valueArrPosition2 := 0 TO 6 DO
				sequenceArray[sequenceID].setOutputValue(step :=teachInStep, valueArrPosition :=
valueArrPosition2, value := hardware.IOplc.getOutput(valueArrPosition := valueArrPosition2));
			END_FOR;
			sequenceArray[sequenceID].setTransferValue(step :=teachInStep, valueArrPosition := 8,
value := sensorSetVal);
			sequenceArray[sequenceID].enSensor(value := enSensor, step := teachInStep);
			sequenceArray[sequenceID].setWaitingTime(step :=teachInStep, value := waitingTime);
			sequenceArray[sequenceID].setLastState(step := teachInStep);

		END_IF;
	END_METHOD
```

```
METHOD getLastState : INT
(*Vrací hodnotu nastaveného posledního stavu*)
VAR_INPUT
	sequenceID : INT;
END_VAR
	getLastState := sequenceArray[sequenceID].getLastState();
END_METHOD

METHOD getActualState : INT
(*Vrací hodnotu aktuálního stavu*)
	getActualState := actualState;
END_METHOD

METHOD setActualState
(*Ruční nastavení stavu automatu*)
VAR_INPUT
	step : int;
END_VAR
	actualState := step;
END_METHOD

METHOD run
(* Voláním metody v programu je v provozu automat.
Zavoláním metody provede automat jedno vyhodnocení aktuální přechodové
funkce, pokud je přechodová funkce splněna, automat se posune do dalšího
stavu. *)
	VAR_INPUT
		sequenceID : INT;
	END_VAR
	IF sequenceID >= 0 and sequenceID < 11 THEN
		sequenceSelected := sequenceID;
	END_IF;
	compareInputsTransfer()
	writeToMotors(seqID:=sequenceID,actualStep := actualState)
				//IF ac()tualState <> 0 THEN
	IF sequenceArray[sequenceSelected].isLastState(step := actualState) THEN
			actualState := 0;
			cycleCounter := cycleCounter + 1;
	END_IF;
END_METHOD

METHOD getCycleCounter : INT
	getCycleCounter := cycleCounter;
END_METHOD

METHOD compareInputsTransfer
(* Porovnává jestli přechodová funkce odpovídá aktuální vstupům a poté spouští časovač*)
	VAR
		valueArrPosition : int;
		radekStejny : BOOL := TRUE;
	END_VAR
		FOR valueArrPosition := 0 TO 7 DO
			IF hardware.IOplc.getInput(valueArrPosition := valueArrPosition) <> sequenceArray
[sequenceSelected].getTransferValue(step := actualState, valueArrPosition := valueArrPosition) THEN
				radekStejny := FALSE;
			END_IF;
		END_FOR;
		IF sequenceArray[sequenceSelected].getSensorEnabled(step := actualState) THEN
			IF hardware.IOplc.getInput(valueArrPosition := 8) <> sequenceArray
```

```
[sequenceSelected].getTransferValue(step := actualState, valueArrPosition := 8) THEN
                                        radekStejny := FALSE;
                                END_IF;
                        END_IF;
                        IF radekStejny THEN
                                timerAutomat.runTime    :=    getTime(sequenceID:=sequenceSelected,
step:=actualState);
                                timerAutomat.start := TRUE;
                                IF timerAutomat.finishFlag THEN
                                        actualState := actualState + 1;
                                        timerAutomat.start := FALSE;
                                END_IF;
                        END_IF;
        END_METHOD

        METHOD writeToMotors
        (* Zapíše hodnoty výstupů na výstupy v PLC, je zde CASE-IF automat,
        aby nedocházelo ke dvěma zápisům zároveň *)
                VAR_INPUT
                        seqID : INT;
                        actualStep : INT;
                END_VAR

                VAR
                        valueArrPosition : int;
                        i:int;

                END_VAR

                CASE selector OF
                0:
                        IF motors.A.isMovementSafe() THEN
                                IF    sequenceArray[sequenceSelected].getOutputValue(step    :=    actualStep,
valueArrPosition := 0) THEN
                                        motors.A.goTo1()
                                        IF motors.A.isPosition1() THEN
                                         selector := selector + 1;
                                        END_IF;
                                ELSE
                                        motors.A.goTo0()
                                        IF motors.A.isPosition0() THEN
                                         selector := selector + 1;
                                        END_IF;
                                END_IF;
                        ELSE
                                selector := selector + 1;
                        END_IF;
                1:
                        IF motors.B.isMovementSafe() THEN
                                IF    sequenceArray[sequenceSelected].getOutputValue(step    :=    actualStep,
valueArrPosition := 1) THEN
                                        motors.B.goTo1()
                                        IF motors.B.isPosition1() THEN
                                         selector := selector + 1;
                                        END_IF;
                                ELSE
                                        motors.B.goTo0()
                                        IF motors.B.isPosition0() THEN
                                         selector := selector + 1;
                                        END_IF;
```

```
                END_IF;
            ELSE
                selector := selector + 1;
            END_IF;
        2:
            IF motors.C.isMovementSafe() THEN
                IF sequenceArray[sequenceSelected].getOutputValue(step := actualStep,
valueArrPosition := 2) THEN
                    motors.C.goTo1()
                    IF motors.C.isPosition1() THEN
                     selector := selector + 1;
                    END_IF;
                ELSE
                    motors.C.goTo0()
                    IF motors.C.isPosition0() THEN
                     selector := selector + 1;
                    END_IF;
                END_IF;
            ELSE
                selector := selector + 1;
            END_IF;
        3:
            IF motors.D.isMovementSafe() THEN
                IF sequenceArray[sequenceSelected].getOutputValue(step := actualStep,
valueArrPosition := 3) THEN
                    motors.D.goTo1()
                    IF motors.D.isPosition1() THEN
                     selector := selector + 1;
                    END_IF;
                ELSE
                    motors.D.goTo0()
                    IF motors.D.isPosition0() THEN
                     selector := selector + 1;
                    END_IF;
                END_IF;
            ELSE
                selector := selector + 1;
            END_IF;
        4:
            IF sequenceArray[sequenceSelected].getOutputValue(step := actualStep, valueArrPosition := 4)
THEN
                motors.E.goTo1()
            ELSE
                motors.E.goTo0()
            END_IF;

            IF sequenceArray[sequenceSelected].getOutputValue(step := actualStep, valueArrPosition := 5)
THEN
                motors.F.goTo1()
            ELSE
                motors.F.goTo0()
            END_IF;

            IF sequenceArray[sequenceSelected].getOutputValue(step := actualStep, valueArrPosition := 6)
THEN
                motors.G.goTo1()
            ELSE
                motors.G.goTo0()
            END_IF;
            selector := 0;
```

```
		ELSE
			selector := 0;
			RETURN;
		END_CASE;
	END_METHOD
	METHOD getTime : TIME
	(* Vrací hodnotu vyčkávacího času uloženého v Sekvenci *)
			VAR_INPUT
				step : int;
				sequenceID : Int;
			END_VAR
			getTime := sequenceArray[sequenceID].getWaitingTime(step := step);
	END_METHOD

	METHOD delSequence
	(* Uvede zvolenou sekvenci do výchozího stavu, sekvence jsou typu proměnných
	RETAIN, což znamená, že NEBUDOU restartem PLC uvedený do výchozích hodnot.
	Proto je třeba provést přepsání hodnot na nulové hodnoty. *)
		VAR_INPUT
			sequenceID : int;
		END_VAR
			sequenceArray[sequenceID].deleteData();
	END_METHOD

	METHOD getTeachedTransferValue : BOOL
	(* Vrací hodnotu přechodové funkce uložené v Sekvenci *)
		VAR_INPUT
			sequenceID : Int;
			step : int;
			valueArrPosition : int;
		END_VAR
			getTeachedTransferValue := sequenceArray[sequenceID].getTransferValue(step := step,
valueArrPosition := valueArrPosition);
	END_METHOD

	METHOD getTeachedOutputValue : BOOL
	(* Vrací hodnotu výstupní hodnoty uložené v Sekvenci *)
		VAR_INPUT
			sequenceID : Int;
			step : int;
			valueArrPosition : int;
		END_VAR
			getTeachedOutputValue := sequenceArray[sequenceID].getOutputValue(step := step,
valueArrPosition := valueArrPosition);
	END_METHOD

	METHOD goToStart : BOOL
	(* Vrací automaticky robotický manipulátor do výchozí polohy.
	Není možné řešit rekurzí, proto metoda vrací hodnotu True
	pokud ještě není ve výchozí poloze, tuto hodnotu je třeba
	použít k opětovnému zavolání metody. *)
			motors.A.goTo0()
			motors.B.goTo0()
			motors.C.goTo0()
			motors.D.goTo0()
			motors.E.goTo0()
			motors.F.goTo0()
			motors.G.goTo0()
			IF motors.A.isPosition0() AND motors.B.isPosition0() AND motors.C.isPosition0() AND
```

```
motors.D.isPosition0() THEN
                                        goToStart := FALSE;
                                        RETURN;
                                ELSE
                                        goToStart := TRUE;
                                        RETURN;
                                END_IF;
        END_METHOD
END_FUNCTION_BLOCK
```

```
PROGRAM hlavniCyklus
VAR
	modbusAutomat: ModbusAutomatKotroler;
END_VAR
	//přečtení vstupů PLC objektem plc
	hardware.IOplc.read();
	//inicializace motororů
	modbusAutomat.initMotors();

	//spuštění hlavní metody kontroleru automatu
	modbusAutomat.read();

	//zapsání proměnných na výstupy PLC
	hardware.IOplc.write();
(*program's body*)
END_PROGRAM
```

```
NAMESPACE hardware
	VAR_GLOBAL
		IOplc : IOplc;
		sensorA1 : BOOL;
		sensorB1 : BOOL;
		sensorC1 : BOOL;
		sensorD1 : BOOL;
		sensorA0 : BOOL;
		sensorB0 : BOOL;
		sensorC0 : BOOL;
		sensorD0 : BOOL;
		sensorBall : BOOL;
		motorAOutput : BOOL;
		motorBOutput : BOOL;
		motorCOutput : BOOL;
		motorDOutput : BOOL;
		motorEOutput : BOOL;
		motorFOutput : BOOL;
		motorGOutput : Bool;
		startButton : BOOL;
		stopButton : BOOL;
		whiteButton : BOOL;
	END_VAR
END_NAMESPACE

FUNCTION_BLOCK IOplc
	VAR
		inputsArr : ARRAY [0..8] OF BOOL;
		outputsArr : ARRAY [0..6] OF BOOL;

	END_VAR

	METHOD read
	(*Přečte vstupy z vstupní tabulky PLC a zapíše je do pole vstupů. *)
		inputsArr[0] := hardware.sensorA0;
		inputsArr[1] := hardware.sensorA1;
		inputsArr[2] := hardware.sensorB0;
		inputsArr[3] := hardware.sensorB1;
		inputsArr[4] := hardware.sensorC0;
		inputsArr[5] := hardware.sensorC1;
		inputsArr[6] := hardware.sensorD0;
		inputsArr[7] := hardware.sensorD1;
		inputsArr[8] := hardware.sensorBall;
	END_METHOD

	METHOD write
	(*Zapíše výstupy z pole výstupních hodnot do výstupní tabulky PLC. *)
			hardware.motorAOutput := outputsArr[0];
			hardware.motorBOutput := outputsArr[1];
			hardware.motorCOutput := outputsArr[2];
			hardware.motorDOutput := outputsArr[3];
			hardware.motorEOutput := outputsArr[4];
			hardware.motorFOutput := outputsArr[5];
			hardware.motorGOutput := outputsArr[6];
	END_METHOD

	METHOD getInput : BOOL
	(*Vrací hodnotu z pole vstupů. *)
		VAR_INPUT
			valueArrPosition : INT;
```

```
        END_VAR
                getInput := inputsArr[valueArrPosition];
    END_METHOD

    METHOD getOutput : BOOL
    (* Vrací hodnotu z pole vstupů. *)
        VAR_INPUT
            valueArrPosition : INT;
        END_VAR
                getOutput := outputsArr[valueArrPosition];
    END_METHOD

    METHOD setOutput
    (* Zapsání hodnoty do pole výstupů. *)
        VAR_INPUT
            valueArrPosition : INT;
            input : BOOL;
        END_VAR
            outputsArr[valueArrPosition] := input;
    END_METHOD
END_FUNCTION_BLOCK
```

```
NAMESPACE motors
	VAR_GLOBAL
		A : Motor;
		B : Motor;
		C : Motor;
		D : Motor;
		E : Motor;
		F : Motor;
		G : Motor; //zvážit předělání motorů do pole
	END_VAR
END_NAMESPACE

FUNCTION_BLOCK ModbusAutomatKotroler
(*
EXTENDS //base type
IMPLEMENTS //interface type list
*)
	VAR
	//OVLÁDÁNÍ STAVŮ
		teachin : BOOL;
		automat : BOOL;
		selector : INT;
		ESTOP : BOOL;
		resetEStop : BOOL;
		servis : BOOL;

	//TEACH-IN
		teach : BOOL;
		teachSekvence : INT;
		cisloKrokuTeach : INT;
		sensorSetValue : BOOL;

		teachedA1 : BOOL;
		teachedA0 : BOOL;
		teachedB1 : BOOL;
		teachedB0 : BOOL;
		teachedC1 : BOOL;
		teachedC0 : BOOL;
		teachedD1 : BOOL;
		teachedD0 : BOOL;

		teachedSensorValue : BOOL;

		teachedOutA : BOOL;
		teachedOutB : BOOL;
		teachedOutC	: BOOL;
		teachedOutD : BOOL;
		teachedOutE : BOOL;
		teachedOutF1 : BOOL;
		teachedOutF2 : BOOL;

		timeINT : lreal;
	  newTIME : int;
		enableSensor : BOOL;

		stepSequence : BOOL;

		deleteAllSequences : BOOL;
		deleteSequence : BOOL;
```

```
//AUTOMAT
        intSekvence : INT;
        reset : BOOL;
        cycleCounter : INT;
        actualState : INT;
        lastState : INT;
        setCycle : INT;

        goToStart : bool;

        auto : REF_TO AutomatInterface;
        automatInstance : Automat;

//MANUAL
        manMotorInt : INT;
        manMotorPlus : BOOL;
        manMotorMinus : BOOL;

        motorApovol : BOOL;

        motorBpovol : BOOL;

        motorCpovol : BOOL;

        motorDpovol : BOOL;


        motorAsenzorplus : BOOL;
        motorAsenzorminus : BOOL;
        motorBsenzorplus : BOOL;
        motorBsenzorminus : BOOL;
        motorCsenzorplus : BOOL;
        motorCsenzorminus : BOOL;
        motorDsenzorplus : BOOL;
        motorDsenzorminus : BOOL;

        motorAerror : BOOL;
        motorBerror : BOOL;
        motorCerror : BOOL;
        motorDerror : BOOL;

        disableAnticolision : BOOL := FALSE;
        ballSensor : BOOL;

        resetAllMotors : Bool;

        setWatchdogTime : BOOL;

        setWatchdogTimeA : LREAL;
        setWatchdogTimeB : LREAL;
        setWatchdogTimeC : LREAL;
        setWatchdogTimeD : LREAL;

        newWatchdogTimeA : LREAL;
        newWatchdogTimeB : LREAL;
        newWatchdogTimeC : LREAL;
        newWatchdogTimeD : LREAL;
END_VAR

METHOD initMotors
```

```
                motors.A.setAnticollision(motorID := 0,
                                                                PK2 := False, PK3 := False, PK10 := False, PK14 :=
False,
                                                                sensor1         :=          hardware.IOplc.getInput
(valueArrPosition := 1),
                                                                sensor0         :=          hardware.IOplc.getInput
(valueArrPosition := 0)
                                                                ); //nastavení povolovacích funkcí, a senzorů
krajních pozic
                //inicializace motororu B
        motors.B.setAnticollision(motorID := 1,
                                                                PK2 := False, PK3 := False, PK11 := FALSE, PK6 :=
False, PK7 := False, PK15 := FALSE,
                                                                sensor1         :=          hardware.IOplc.getInput
(valueArrPosition := 3),
                                                                sensor0         :=          hardware.IOplc.getInput
(valueArrPosition := 2)
                                                                ); //nastavení povolovacích funkcí, a senzorů
krajních pozic
                //inicializace motororu C
        motors.C.setAnticollision(motorID := 2,
                                                                PK4 := False, PK9 := False, PK11 := False,
                                                                sensor1         :=          hardware.IOplc.getInput
(valueArrPosition := 5),
                                                                sensor0         :=          hardware.IOplc.getInput
(valueArrPosition := 4)
                                                                ); //nastavení povolovacích funkcí, a senzorů
krajních pozic
                //inicializace motororu D
        motors.D.setAnticollision(motorID := 3,
                                                                PK6 := False, PK10 := False, PK7 := FALSE,
PK11 := False,
                                                                sensor1         :=          hardware.IOplc.getInput
(valueArrPosition := 7),
                                                                sensor0         :=          hardware.IOplc.getInput
(valueArrPosition := 6)
                                                                ); //nastavení povolovacích funkcí, a senzorů
krajních pozic
                //inicializace motororu E
        motors.E.setAnticollision(motorID := 4);
                //inicializace motororu F
        motors.F.setAnticollision(motorID := 5); //nastavení reference na výstupní proměnné objektu motoru
                //inicializace motororu G
        motors.G.setAnticollision(motorID := 6); //nastavení reference na výstupní proměnné objektu motoru
        //Konec Inicializace motorů
        END_METHOD

        METHOD read
                VAR
                        toStart : BOOL;

                        i : INT;
                END_VAR

                auto := REF(automatInstance);

                motorAsenzorplus := motors.A.isPosition1();
                motorAsenzorminus := motors.A.isPosition0();
                motorBsenzorplus := motors.B.isPosition1();
                motorBsenzorminus := motors.B.isPosition0();
```

```
motorCsenzorplus := motors.C.isPosition1();
motorCsenzorminus := motors.C.isPosition0();
motorDsenzorplus := motors.D.isPosition1();
motorDsenzorminus := motors.D.isPosition0();

motorApovol := motors.A.isMovementSafe();
motorBpovol := motors.B.isMovementSafe();
motorCpovol := motors.C.isMovementSafe();
motorDpovol := motors.D.isMovementSafe();

motorAerror := motors.A.getErrorStatus();
motorBerror := motors.B.getErrorStatus();
motorCerror := motors.C.getErrorStatus();
motorDerror := motors.D.getErrorStatus();

ballSensor := hardware.IOplc.getInput(valueArrPosition := 8);

cycleCounter := auto.getCycleCounter();

lastState := auto.getLastState(intSekvence);
actualState := auto.getActualState();

teachedA0 := auto.getTeachedTransferValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 0);
teachedA1 := auto.getTeachedTransferValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 1);
teachedB0 := auto.getTeachedTransferValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 2);
teachedB1 := auto.getTeachedTransferValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 3);
teachedC0 := auto.getTeachedTransferValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 4);
teachedC1 := auto.getTeachedTransferValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 5);
teachedD0 := auto.getTeachedTransferValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 6);
teachedD1 := auto.getTeachedTransferValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 7);
teachedSensorValue := auto.getTeachedTransferValue(sequenceID := teachSekvence, step :=
cisloKrokuTeach, valueArrPosition := 8);

teachedOutA := auto.getTeachedOutputValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 0);
teachedOutB := auto.getTeachedOutputValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 1);
teachedOutC := auto.getTeachedOutputValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 2);
teachedOutD := auto.getTeachedOutputValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 3);
teachedOutE := auto.getTeachedOutputValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 4);
teachedOutF1 := auto.getTeachedOutputValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 5);
teachedOutF2 := auto.getTeachedOutputValue(sequenceID := teachSekvence, step := cisloKrokuTeach,
valueArrPosition := 6);

timeINT := TIMETOTALMILLISECONDS(in1 := auto.getTime(step := cisloKrokuTeach,
sequenceID:=teachSekvence));

setWatchdogTimeA := TIMETOTALMILLISECONDS(in1 := watchdogTimeA);
```

```
setWatchdogTimeB := TIMETOTALMILLISECONDS(in1 := watchdogTimeB);
setWatchdogTimeC := TIMETOTALMILLISECONDS(in1 := watchdogTimeC);
setWatchdogTimeD := TIMETOTALMILLISECONDS(in1 := watchdogTimeD);

CASE selector OF
	0: //IDLE

		IF stepSequence THEN
			auto.writeToMotors(actualStep:=cisloKrokuTeach,
seqID:=intSekvence);
			auto.setActualState(step := cisloKrokuTeach);
		END_IF;

		IF goToStart THEN
			goToStart := auto.goToStart();
		END_IF;

		IF reset THEN
			stepSequence := FALSE;
			auto.resetAutomat();
		END_IF;

	IF manMotorPlus THEN
		CASE manMotorInt OF
			1: motors.A.goTo1();
			2: motors.B.goTo1();
			3: motors.C.goTo1();
			4: motors.D.goTo1();
			5: motors.E.goTo1();
			6: motors.F.goTo1();
			7: motors.G.goTo1();
		END_CASE;
	END_IF;

	IF manMotorMinus THEN
		CASE manMotorInt OF
			1: motors.A.goTo0();
			2: motors.B.goTo0();
			3: motors.C.goTo0();
			4: motors.D.goTo0();
			5: motors.E.goTo0();
			6: motors.F.goTo0();
			7: motors.G.goTo0();
		END_CASE;
	END_IF;

	//TRANSFER FUNCTION

	IF automat or hardware.startButton THEN
		selector := 1;
		automat := TRUE;
	END_IF;

	IF teachin THEN
		selector := 2;
	END_IF;

	IF not hardware.stopButton THEN
		automat := FALSE;
		ESTOP := TRUE;
```

```
                selector := 10;
        END_IF;

        IF servis THEN
                selector := 3
        END_IF;

1: //Automat
                auto.run(sequenceID := intSekvence);

        //TRANSFER FUNCTION

        IF auto.getCycleCounter() >= setCycle AND setCycle <> 0 THEN
                automat := FALSE;
                selector := 0;
        END_IF;

        IF NOT automat OR hardware.whiteButton THEN
                selector := 0;
                automat := FALSE;
        END_IF;

        IF teachin THEN
                selector := 2;
                automat := FALSE;
        END_IF;

2: //Teach
        IF manMotorPlus THEN
                CASE manMotorInt OF
                        1: motors.A.goTo1(disableAnticollision := disableAnticolision);
                        2: motors.B.goTo1(disableAnticollision := disableAnticolision);
                        3: motors.C.goTo1(disableAnticollision := disableAnticolision);
                        4: motors.D.goTo1(disableAnticollision := disableAnticolision);
                        5: motors.E.goTo1();
                        6: motors.F.goTo1();
                        7: motors.G.goTo1();
                END_CASE;
        END_IF;

        IF manMotorMinus THEN
                CASE manMotorInt OF
                        1: motors.A.goTo0(disableAnticollision := disableAnticolision);
                        2: motors.B.goTo0(disableAnticollision := disableAnticolision);
                        3: motors.C.goTo0(disableAnticollision := disableAnticolision);
                        4: motors.D.goTo0(disableAnticollision := disableAnticolision);
                        5: motors.E.goTo0();
                        6: motors.F.goTo0();
                        7: motors.G.goTo0();
                END_CASE;
        END_IF;
                IF cisloKrokuTeach < 0 THEN
                        cisloKrokuTeach := 0;
                ELSIF cisloKrokuTeach > 100 THEN
                        cisloKrokuTeach := 100;
                END_IF;

                IF deleteSequence THEN
                        auto.delSequence(sequenceID := teachSekvence);
                END_IF;
```

```
                    IF newTIME < 0 THEN
                        newTIME := 0;
                    END_IF;

                    IF teach THEN
                        auto.teachIN(waitingTime := mul_time_int(in1 := T#100ms, in2 := newTIME), enSensor := enableSensor, sensorSetVal := sensorSetValue, sequenceID := teachSekvence, teachInStep := cisloKrokuTeach);
                        cisloKrokuTeach := cisloKrokuTeach + 1;
                        teach := False;
                    END_IF;

                //TRANSFER FUNCTION

                IF not teachin THEN
                    selector := 0;
                    teach := FALSE;
                END_IF;

                IF ESTOP THEN
                    selector := 10;
                    teach := FALSE;
                END_IF;

                IF servis THEN
                    selector := 3
                    teach := FALSE;
                END_IF;

            3: //Servis
                IF manMotorPlus THEN
                    CASE manMotorInt OF
                        1: motors.A.goTo1(disableAnticollision := disableAnticolision);
                        2: motors.B.goTo1(disableAnticollision := disableAnticolision);
                        3: motors.C.goTo1(disableAnticollision := disableAnticolision);
                        4: motors.D.goTo1(disableAnticollision := disableAnticolision);
                        5: motors.E.goTo1();
                        6: motors.F.goTo1();
                        7: motors.G.goTo1();
                    END_CASE;
                END_IF;

                IF manMotorMinus THEN
                    CASE manMotorInt OF
                        1: motors.A.goTo0(disableAnticollision := disableAnticolision);
                        2: motors.B.goTo0(disableAnticollision := disableAnticolision);
                        3: motors.C.goTo0(disableAnticollision := disableAnticolision);
                        4: motors.D.goTo0(disableAnticollision := disableAnticolision);
                        5: motors.E.goTo0();
                        6: motors.F.goTo0();
                        7: motors.G.goTo0();
                    END_CASE;
                END_IF;

                IF resetAllMotors THEN
                    motors.A.setErrorStatus(input := FALSE);
                    motors.B.setErrorStatus(input := FALSE);
                    motors.C.setErrorStatus(input := FALSE);
                    motors.D.setErrorStatus(input := FALSE);
```

```
                    resetAllMotors := FALSE;
                END_IF;

                IF setWatchdogTime THEN
                    watchdogTimeA:=mul_time_lreal(in1 := T#1ms, in2 := newWatchdogTimeA);
                    watchdogTimeB:=mul_time_lreal(in1 := T#1ms, in2 := newWatchdogTimeB);
                    watchdogTimeC:=mul_time_lreal(in1 := T#1ms, in2 := newWatchdogTimeC);
                    watchdogTimeD:=mul_time_lreal(in1 := T#1ms, in2 := newWatchdogTimeD);
                    setWatchdogTime := FALSE;
                END_IF;

                IF deleteAllSequences THEN

                        FOR i := 0 TO 10 DO

                            auto.delSequence(sequenceID := i);

                        END_FOR;

                        deleteAllSequences := FALSE;

                END_IF;

                //TRANSFER FUNKTION

                IF not servis THEN
                    selector := 0
                    servis := False;
                END_IF;

            10:


                //TRANSFER FUNCTION
                IF (*NOT (motors.A.getErrorStatus() OR motors.B.getErrorStatus() OR
motors.C.getErrorStatus() OR motors.D.getErrorStatus()) AND *) resetEStop THEN
                    resetEStop := FALSE;
                    ESTOP := FALSE;
                    selector := 3;
                END_IF;

        END_CASE;



        IF not hardware.stopButton THEN
            automat := FALSE;
            ESTOP := TRUE;
            selector := 10;
        END_IF;


    END_METHOD

END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK Motor
(*
EXTENDS //base type
IMPLEMENTS //interface type list
*)

VAR
		counter : INT;
		plusEnabled : BOOL := TRUE;
		minusEnabled : BOOL := TRUE;
		sensorPlus : BOOL := TRUE;
		sensorMinus : BOOL := TRUE;
		enabledCombinations : ARRAY [0..15] OF BOOL;
		watchdogEnabled : BOOL := TRUE;

		ID : INT;
		errorStatus : BOOL;
		runWatchdogp : BOOL;
		runWatchdogm : BOOL;
		combinations : ARRAY [0..15, 0..7] OF BOOL :=
		[1,0,   1,0,   1,0,   1,0, //0
		 1,0,   1,0,   1,0,   0,1, //1
		 1,0,   1,0,   0,1,   1,0, //2
		 1,0,   1,0,   0,1,   0,1, //3
		 1,0,   0,1,   1,0,   1,0, //4
		 1,0,   0,1,   1,0,   0,1, //5
		 1,0,   0,1,   0,1,   1,0, //6
		 1,0,   0,1,   0,1,   0,1, //7
		 0,1,   1,0,   1,0,   1,0, //8
		 0,1,   1,0,   1,0,   0,1, //9
		 0,1,   1,0,   0,1,   1,0, //10
		 0,1,   1,0,   0,1,   0,1, //11
		 0,1,   0,1,   1,0,   1,0, //12
		 0,1,   0,1,   1,0,   0,1, //13
		 0,1,   0,1,   0,1,   1,0, //14
		 0,1,   0,1,   0,1,   0,1  //15
		];
END_VAR

METHOD setAnticollision
(*Pneumatický manipulátor pracuje s poli vstupů a výstupů,
který dále zpracovává objekt třídy PLC. Tímto číslem se nastavuje pozice v poli,
která odpovídá výstupu PLC pro ovládání ventilu daného pohonu.
Tuto hodnotu je potřeba nastavit i u pohonů bez koncových senzorů.*)
	VAR_INPUT
		motorID : INT;
		sensor1 : BOOL := FALSE;
		sensor0 : BOOL := FALSE;

		PK0 : BOOL := TRUE;
		PK1 : BOOL := TRUE;
		PK2 : BOOL := TRUE;
		PK3 : BOOL := TRUE;
		PK4 : BOOL := TRUE;
		PK5 : BOOL := TRUE;
		PK6 : BOOL := TRUE;
		PK7 : BOOL := TRUE;
		PK8 : BOOL := TRUE;
		PK9 : BOOL := TRUE;
		PK10 : BOOL := TRUE;
```

```
                PK11 : BOOL := TRUE;
                PK12 : BOOL := TRUE;
                PK13 : BOOL := TRUE;
                PK14 : BOOL := TRUE;
                PK15 : BOOL := TRUE;

        END_VAR
        VAR
                i : int;
        END_VAR
                        ID := motorID;
                        sensorPlus := sensor1;
                        sensorMinus := sensor0;
                        enabledCombinations[0] := PK0;
                        enabledCombinations[1] := PK1;
                        enabledCombinations[2] := PK2;
                        enabledCombinations[3] := PK3;
                        enabledCombinations[4] := PK4;
                        enabledCombinations[5] := PK5;
                        enabledCombinations[6] := PK6;
                        enabledCombinations[7] := PK7;
                        enabledCombinations[8] := PK8;
                        enabledCombinations[9] := PK9;
                        enabledCombinations[10] := PK10;
                        enabledCombinations[11] := PK11;
                        enabledCombinations[12] := PK12;
                        enabledCombinations[13] := PK13;
                        enabledCombinations[14] := PK14;
                        enabledCombinations[15] := PK15;

END_METHOD

METHOD _isNOTcolision : BOOL
(*Vyhodnocení protikolizního systému*)
        VAR
                i : int;
                k : int;
                counter : int := 0 ;
        END_VAR

        FOR i := 0 TO 15 DO
        counter := 0
                IF enabledCombinations[i] THEN
                        FOR k := 0 TO 7 DO
                                IF combinations[i, k] = hardware.IOplc.getInput(valueArrPosition := k) THEN
                                        counter := counter + 1;
                                END_IF;
                        END_FOR;
                        IF counter = 8 THEN
                                _isNOTcolision := TRUE;
                                RETURN;
                        END_IF;
                END_IF;
        END_FOR;
        _isNOTcolision := FALSE;
END_METHOD

METHOD isPosition0 : BOOL
(*Vrací booleovskou hodnotu senzoru koncové polohy 0,
je-li hodnota True, motor je v této koncové poloze.*)
```

```
	isPosition0 := sensorMinus;
END_METHOD

METHOD isPosition1 : BOOL
(*Vrací booleovskou hodnotu senzoru koncové polohy 1,
je-li hodnota True, motor je v této koncové poloze.
*)
	isPosition1 := sensorPlus;
END_METHOD

METHOD isMovementSafe : BOOL
(*Vrací booleovskou hodnotu, jestli změna polohy motoru nezpůsobí kolizi,
je-li hodnota True, pohyb je bezkolizní*)
	isMovementSafe := _isNOTcolision();
END_METHOD

METHOD goTo1
(*Provede pohyb motoru z 0 do 1*)
	VAR_INPUT
		disableAnticollision : BOOL := False;
	END_VAR
	IF _isNOTcolision() and not disableAnticollision and not errorStatus THEN
		hardware.IOplc.setOutput(valueArrPosition := ID, input:=TRUE);
		runWatchdogp := TRUE;
		//pocitadlo(reset := sensorPlus);
	ELSIF disableAnticollision THEN
		hardware.IOplc.setOutput(valueArrPosition := ID, input:=TRUE);
		runWatchdogp := TRUE;
		//pocitadlo(reset := sensorPlus);
	END_IF;

END_METHOD

METHOD goTo0
(*Provede pohyb motoru z 1 do 0*)
	VAR_INPUT
		disableAnticollision : BOOL := FALSE;
	END_VAR

	IF _isNOTcolision() and not disableAnticollision and not errorStatus THEN
		hardware.IOplc.setOutput(valueArrPosition := ID, input:=FALSE);
		runWatchdogm := TRUE;

	ELSIF disableAnticollision THEN
		hardware.IOplc.setOutput(valueArrPosition := ID, input:=FALSE);
		runWatchdogm := TRUE;

	END_IF;
END_METHOD


METHOD getRunWatchdogP : BOOL
(*Vrací řídící hodnotu watchdogu*)
	getRunWatchdogP := runWatchdogp;
END_METHOD

METHOD resetRunWatchdogP
(*Reseteuje řídící hodnotu watchdogu*)
	IF sensorPlus THEN
		runWatchdogp := FALSE;
```

```
        END_IF;

END_METHOD

METHOD getRunWatchdogM : BOOL
(*Vrací řídící hodnotu watchdogu*)
        getRunWatchdogM := runWatchdogm;
END_METHOD

METHOD resetRunWatchdogM
(*Reseteuje řídící hodnotu watchdogu*)
        IF sensorPlus THEN
                runWatchdogm := FALSE;
        END_IF;

END_METHOD

METHOD getErrorStatus : BOOL
(*Vrací hodnotu bool, zda je motor v chybném stavu*)
        getErrorStatus := errorStatus;
END_METHOD

METHOD setErrorStatus
(*Nastaví hodnotu, zda je motor v chybném stavu*)
VAR_INPUT
        input : BOOL;
END_VAR

        runWatchdogm := False;
        runWatchdogp := False;
        errorStatus := input;

END_METHOD

(*function block body*)
END_FUNCTION_BLOCK
```

```
FUNCTION_BLOCK Sequence
(*
EXTENDS //base type
IMPLEMENTS //interface type list
*)
	VAR
		transferMatrix : ARRAY [0..100,0..8] OF BOOL;
		outputMatrix : ARRAY [0..101,0..6] OF BOOL;
		enSensorMatrix : ARRAY [0..100] OF BOOL;
		waitingTimeMatrix : ARRAY [0..100] OF TIME;
		lastState : int;
	END_VAR
		METHOD isLastState : BOOL
		(*Vrací hodnotu True, pokud je stav posledním stavem sekvence.*)
			VAR_INPUT
				step : int;
			END_VAR
				IF lastState = step - 1 THEN
					isLastState := TRUE;
				ELSE
					isLastState := FALSE;
				END_IF;
		END_METHOD

		METHOD getLastState : INT
		(*Vrací INT posledního stavu.*)
			getLastState := lastState;
		END_METHOD

		METHOD enSensor
		(*Nastaví hodnotu do matice enSensorMatrix*)
			VAR_INPUT
				step : INT;
				value : BOOL;
			END_VAR
			enSensorMatrix[step] := value;
		END_METHOD

		METHOD getSensorEnabled : BOOL
		(*Vrací hodnotu z matice enSensorMatrix*)
			VAR_INPUT
				step : INT;
			END_VAR
				getSensorEnabled := enSensorMatrix[step];
		END_METHOD

		METHOD setLastState
		(*Nastaví poslední stav sekvence dle hodnoty \textit{stav}.*)
			VAR_INPUT
				step : int;
			END_VAR
				lastState := step;
		END_METHOD
		METHOD getWaitingTime : TIME
		(*Vrací hodnotu času pro vyčkávání*)
			VAR_INPUT
				step : int;
			END_VAR
				getWaitingTime := waitingTimeMatrix[step];
		END_METHOD
```

```
		METHOD setWaitingTime
		(*Nastaví hodnotu času pro vyčkávání*)
			VAR_INPUT
				step : int;
				value : TIME;
			END_VAR
				waitingTimeMatrix[step] := value;
		END_METHOD
		METHOD deleteData
		(*Smaže celou sekvenci*)
		VAR
			i : int;
			j : int;
			k : int;
			l : int;
			t : int;
		END_VAR
			FOR  i := 0 TO 100 DO
				FOR  j := 0 TO 8 DO
					transferMatrix[i, j] := False;
				END_FOR;
			END_FOR;
			FOR  i := 0 TO 101 DO
				FOR  j := 0 TO 6 DO
					outputMatrix[i, j] := False;
				END_FOR;
			END_FOR;
			FOR  t := 0 TO 100 DO
					waitingTimeMatrix[t] := T#0s;
					enSensorMatrix[t] := FALSE;
			END_FOR;
		END_METHOD
//vrací přechodový bod
		METHOD getTransferValue : BOOL
			VAR_INPUT
				step : int;
				valueArrPosition : int;
			END_VAR
				getTransferValue := transferMatrix[step, valueArrPosition];
		END_METHOD
//vraci vystupni bod
		METHOD getOutputValue : BOOL
			VAR_INPUT
				step : int;
				valueArrPosition : int;
			END_VAR
				getOutputValue := outputMatrix[step, valueArrPosition];
		END_METHOD
//nastaví přechodový bod
		METHOD setTransferValue
			VAR_INPUT
				value : BOOL;
				step : int;
				valueArrPosition : int;
			END_VAR
				transferMatrix[step, valueArrPosition] := value;
		END_METHOD
//nastaví výstupní bod
		METHOD setOutputValue
			VAR_INPUT
```

```
                value : BOOL;
                step : int;
                valueArrPosition : int;
            END_VAR
                outputMatrix[step, valueArrPosition] := value;
        END_METHOD
END_FUNCTION_BLOCK
```

```
PROGRAM timerAutomat
	VAR
		timer : TON; (* add local variables here *)
	END_VAR

	VAR_INPUT

		start : BOOL;

		runTime : TIME;
	END_VAR

	VAR_OUTPUT
		finishFlag : BOOL;
	END_VAR

	timer(IN := start, PT := runTime);
	finishFlag := timer.Q;

END_PROGRAM
```

```
VAR_GLOBAL RETAIN
	watchdogTimeA : TIME := T#1s;
	watchdogTimeB : TIME := T#1s;
	watchdogTimeC : TIME := T#1s;
	watchdogTimeD : TIME := T#1s;
END_VAR

PROGRAM watchdog

	VAR
		wtonAp : TON;
		wtonBp : TON;
		wtonCp : TON;
		wtonDp : TON;

		wtonAm : TON;
		wtonBm : TON;
		wtonCm : TON;
		wtonDm : TON;

	END_VAR


		wtonAp(IN := not motors.A.isPosition1() and motors.A.getRunWatchdogP() and not motors.A.getErrorStatus(), PT := watchdogTimeA);

		motors.A.resetRunWatchdogP();

		wtonBp(IN := not motors.B.isPosition1() and motors.B.getRunWatchdogP() and not motors.B.getErrorStatus(), PT := watchdogTimeB);

		motors.B.resetRunWatchdogP();

		wtonCp(IN := not motors.C.isPosition1() and motors.C.getRunWatchdogP() and not motors.C.getErrorStatus(), PT := watchdogTimeC);

		motors.C.resetRunWatchdogP();

		wtonDp(IN := not motors.D.isPosition1() and motors.D.getRunWatchdogP() and not motors.D.getErrorStatus(), PT := watchdogTimeD);

		motors.D.resetRunWatchdogP();

		wtonAm(IN := not motors.A.isPosition0() and motors.A.getRunWatchdogM() and not motors.A.getErrorStatus(), PT := watchdogTimeA);

		motors.A.resetRunWatchdogM();

		wtonBm(IN := not motors.B.isPosition0() and motors.B.getRunWatchdogM() and not motors.B.getErrorStatus(), PT := watchdogTimeB);

		motors.B.resetRunWatchdogM();

		wtonCm(IN := not motors.C.isPosition0() and motors.C.getRunWatchdogM() and not motors.C.getErrorStatus(), PT := watchdogTimeC);

		motors.C.resetRunWatchdogM();

		wtonDm(IN := not motors.D.isPosition0() and motors.D.getRunWatchdogM() and not motors.D.getErrorStatus(), PT := watchdogTimeD);
```

```
		motors.D.resetRunWatchdogM();

		IF wtonAp.Q or wtonAm.Q THEN
			motors.A.setErrorStatus(input := True);
		END_IF;

		IF wtonBp.Q or wtonBm.Q THEN
			motors.B.setErrorStatus(input := True);
		END_IF;

		IF  wtonCp.Q or wtonCm.Q THEN
			motors.C.setErrorStatus(input := True);
		END_IF;

		IF wtonDp.Q or wtonDm.Q THEN
			motors.D.setErrorStatus(input := True);
		END_IF;

END_PROGRAM
```