VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY
INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS

SOFTWARE PRO TVORBU KONFIGURAČNÍCH SOUBORŮ ROBOTŮ ABB
SOFTWARE FOR THE CREATION OF THE ABB ROBOTS CONFIGURATION FILES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **František Badal**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Jakub Bražina**
SUPERVISOR

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání bakalářské práce

Ústav: Ústav mechaniky těles, mechatroniky a biomechaniky

Student: **František Badal**

Studijní program: Aplikované vědy v inženýrství

Studijní obor: Mechatronika

Vedoucí práce: **Ing. Jakub Bražina**

Akademický rok: 2021/22

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma bakalářské práce:

Software pro tvorbu konfiguračních souborů robotů ABB

Stručná charakteristika problematiky úkolu:

V rámci této práce se student seznámí s konfiguračními soubory pro průmyslové roboty ABB. Tyto soubory obsahují různé systémové informace, kde nejdůležitější je popis signálů. Tyto signály slouží ke vzájemné komunikaci PLC a robotu a je nutné je ručně přidat do řídícího systému robotu a PLC. Signály bývají ve formátu tabulky (.xlsx) ve kterém je možno je nahrát do PLC, ale také do řady simulačních softwarů. Konfigurační soubory jsou ve formátu textového dokumentu (.config) a za účelem zjednodušení reálného zprovoznění je přínosné je vygenerovat na základě dostupných tabulek se signály. Proto je hlavním cílem této práce návrh softwaru, který tabulky se signály načte a vygeneruje požadované konfigurační soubory, které jsou posléze nahrány do robotu. Navržený software usnadní nastavení signálů řídícího systému robotu a jeho následnou komunikaci s PLC, případně dalšími platformami. Software bude využíván ve výuce problematiky virtuálního zprovoznění robotických výrobních systémů.

Cíle bakalářské práce:

Rešerše dané problematiky.

Analýza konfiguračních souborů.

Návrh softwaru pro tvorbu konfiguračních souborů.

Test funkčnosti vygenerovaných konfiguračních souborů na řídícím systému robotu ABB.

Seznam doporučené literatury:

SICILIANO, B. a O. KHATIB. Springer handbook of robotics. Berlin: Springer, c2008. ISBN978-3-50-23957-4.

KOLÍBAL, Z. a kol.: Roboty a robotizované výrobní technologie. VUTIUM Brno, 2016, ISBN 978---214-4828-5.

OLSSON, Mikael. C# 7 Quick Syntax Reference: A Pocket Guide to the Language, APIs, and Library.

Finland: Apress, c2018. ISBN 978-1-4842-3816-5.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2021/22

V Brně, dne

L. S.

prof. Ing. Jindřich Petruška, CSc.	doc. Ing. Jaroslav Katolický, Ph.D.
ředitel ústavu	děkan fakulty

Abstrakt

Tato bakalářská práce se v teoretické části zabývá programovatelnými logickými automaty, základy nejznámějších průmyslových sběrnic a konfiguračních souborů a dává přehled o principech objektově orientovaného programování. V praktické části se pak tato práce zabývá problematikou návrhu programu pro ulehčení generování konfiguračního souboru pro průmyslové roboty ABB, a to hlavně z hlediska zjednodušení tvoření a vkládání signálů pro komunikaci mezi jednotlivými roboty a PLC. Celý program je napsán v programovacím jazyce C Sharp za užití objektově orientovaného přístupu ve vývojovém prostředí Visual Studio.

Abstract

This bachelor's thesis is divided to two parts. The theoretical part acquaints the reader with programmable logic controllers, the basics of the most well-known industrial buses and configuration files and gains an overview of the principles of object-oriented programming. In the practical part, this work deals with the design of a program to facilitate the generation of a configuration file for ABB industrial robots, mainly in terms of simplifying the generation and insertion of signals for communication between individual robots and PLCs. The whole program is written in the C Sharp programming language using an object-oriented approach in the Visual Studio development environment.

Klíčová slova

PLC, Průmyslová sběrnice, Profinet, Konfigurační soubor, EIO signál, Programování, OOP, Objektově orientované programování, C Shrap, Visual Studio

Keywords

PLC,Industrial bus, Profinet, Config file, EIO signal, Programming, OOP, Object oriented programming, C Sharp, Visual Studio

Bibliografická citace

BADAL, František. *Software pro tvorbu konfiguračních souborů robotů ABB.* Brno, 2022. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/132991. Bakalářská práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav mechaniky těles, mechatroniky a biomechaniky. Vedoucí práce Jakub Bražina.

Čestně prohlašuji, že jsem bakalářskou práci *Software pro tvorbu konfiguračních souborů robotů ABB* vypracoval samostatně pod vedením Ing. Jakuba Bražiny, a s použitím materiálů uvedených v seznamu zdrojů.

František Badal

Děkuji svému vedoucímu Ing. Jakubovi Bražinovi za věnovaný čas, technické rady, trpělivost a svědomité vedení mé práce.

František Badal

Obsah

1. Úvod

Průmyslové odvětví se stále posouvá vpřed, a především ve výrobě se čím dál více prosazují prvky, které zjednodušují a zrychlují výrobu, nebo celý průmyslový proces. K tomu se používá různých optimalizačních metod ale především i automatizace. Automatizace ve výrobě mimo jiné spoléhá na PLC tedy programovatelné logické automaty, které mohou řídit jak jednotlivé roboty, senzory, aktuátory atd. ve výrobě, tak i celý výrobní proces.

K tomu, aby se celý proces dal řídit a kontrolovat se mohou používat další nadřazené systémy, který celý proces monitorují. Tyto procesy musí mezi sebou nějakým způsobem komunikovat. Tato komunikace probíhá skrz různé průmyslové sběrnice, a na základě signálů z těchto sběrnic a z různých subsystémů z části výroby nebo přímo z jednotlivých robotů, senzorů atd. S příchodem průmyslu 4.0 je pravděpodobné, že počet těchto signálů, na základě kterých se budou řídit celé procesy se bude zvyšovat.

Cílem bakalářské práce je pak vyvinout program, který usnadní nastavení těchto signálů, jelikož se tyto signály v originálním softwaru tvoří velmi neefektivně, a tedy například rozjetí většího počtu PLC jednotek může být časově velmi náročné. Tyto signály se nachází v konfiguračním souboru, který pak lze nahrát do PLC.

V teoretické části si čtenář může udělat představu o tom, jak PLC funguje, jaké jsou používané sběrnice v průmyslu, co si představit pod pojmem konfigurační soubor, nebo si udělat základní představu o programovacích jazycích a objektově orientovaném přístupu v programování.

V praktické části je pak popsáno, jak vyvinutý program funguje a jaké jsou jeho části.

2. Teoretická část

2.1 PLC

PLC neboli programovatelný logický automat je druh počítače určený pro průmyslové využití. Tento počítač pak disponuje jak hardwarovou, tak softwarovou výbavou, aby mohli plnit různé úkoly, především ve výrobě, ale v dnešní době i v jiných odvětvích než jen průmyslu, jako například ve zdravotnictví, komunikační infrastruktuře nebo energetice.

Existují dva základní druhy. Kompaktní PLC je jedno jediné zařízení obsahuje vše, co zákazník potřebuje, modulární PLC se skládá ze samostatných modulů a lze tak jednoduše rozšiřovat úkony, které má vykonávat [1].

Obrázek 1: Ukázka PLC od společnosti ABB

2.1.1 Historie

Hlavní důvod vzniku PLC počítačů byla hlavně snaha o optimalizování a zautomatizování procesů v průmyslu a výrobě. V době před těmito počítači byly stěžejním řízením procesů hlavně elektromagnetické relé obvody, které ovládali různé ventily, motory, tlačítka atd. Tyto relé obvody většinou zabírali celé místnosti a jejich údržba byla náročná, zvlášť pokud nastala porucha v jednom relé, na kterém byly závislé další operace v procesu.

Oprava takového obvodu byla náročná, zdlouhavá a tím pádem i finančně nákladná. První PLC počítače se začali objevovat v pozdních 60. letech 20. století a vzhledem k náhradě obvodů s relé prvky bylo programování těchto počítačů založeno na booleanské algebře [2].

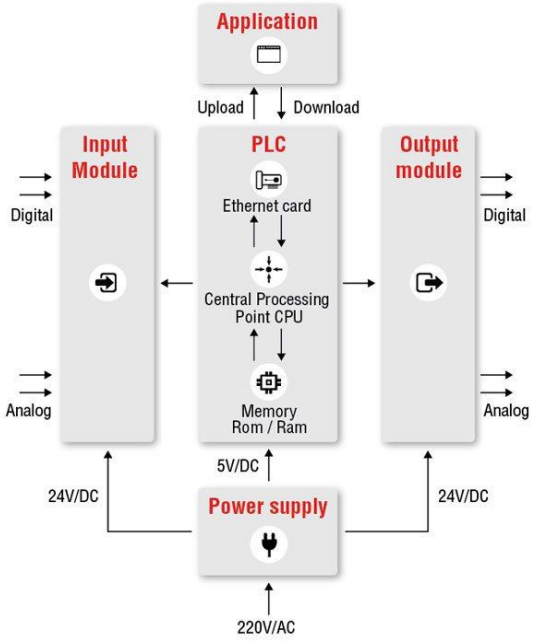
Postupem času se s větším výpočetním výkonem a pamětí začali objevovat jiné formáty pro programování PLC, jako například funkční diagramy, strukturovaný text nebo grafické ledder digramy. V dnešní době se těmto počítačům věnují hlavně velké firmy, jako například Siemens, Allen-Bradley nebo ABB.

2.1.2 Architektura PLC

PLC počítač jako takový lze rozdělit do několika subsystémů. Konkrétně do vstupních modulů, výstupních modulů, Centrální procesorové jednotky (CPU) a paměti ROM (Read-Only Memory) a RAM (Random Acces Memory). Vstupní a výstupní moduly se mohou dále třídit na analogové a digitální vstupy/výstupy, které se podle potřeby nadefinují. Toto záleží již na konkrétní aplikaci, ať už se jedná o sběr dat ze senzorů nebo výstupu signálů, které ovládají např. motory, ventily atd. CPU a paměti ROM a RAM pak mají stejný úkol jako v normálních počítačích.

Ve většině aplikací je několik PLC počítačů pod centrální aplikací, která řídí a monitoruje buď jen část nebo celý proces. Centrální aplikace pak může běžet například na serveru, a pomocí různých sběrnic úkolovat konkrétní PLC počítače. Tato „topologie“ se ale samozřejmě může měnit podle potřeb a možností v konečném individuálním procesu.

Pro dodatečnou interakci nebo monitorování stavu PLC se využívá i Human Machine Interface neboli zkráceně HMI. Většinou se jedná o dotykový display kompatibilní s PLC a jak název napovídá, jde o zařízení, které je uzpůsobeno tak, aby s ním interagoval člověk, a mohl tak do procesu PLC zasáhnout, popřípadě zkontrolovat chyby atd [3].



Obrázek 2: Architektura PLC, převzato z [3]

2.2 Průmyslové sběrnice

Průmyslová sběrnice se dá považovat za jakousi páteř komunikace, ať už ve výrobní lince nebo v jiném automatizovaném procesu. Tyto sběrnice pak díky svým standardům definují jak, kdy a s kým můžou jednotlivá zařízení komunikovat, ať už se jedná o server na lince, který řídí celý proces, nebo jednotlivé PLC počítače, která úkolují

jednotlivé roboty, manipulátory, motory atd. S příchodem průmyslu 4.0 je pravděpodobné, že se nároky na tyto sběrnice budou zvyšovat, a to hlavně z hlediska počtu připojených zařízení, nebo rychlosti přenosu dat.

Většina sběrnic nějakým způsobem implementuje referenční model ISO/OSI, který je považován za jednotný standard, jak by měly jednotlivé zařízení komunikovat mezi sebou.

2.2.1 Profibus

Profibus (PROcess Field BUS) je průmyslová sběrnice, jejíž první verze se objevila v roce 1989, je specifikována evropskou normou EN 50170 a je přizpůsobena pro všechny oblasti automatizace jako výrobní linky, procesní automatizace, ale i pro automatizaci v domácnosti. Tato sběrnice je jednou z nejrozšířenějších v Evropě.

Profibus je založen na otevřeném komunikačním protokolu ISO/OSI, kde využívá první, druhou a sedmou vrstvu, tj. fyzickou, linkovou a aplikační.
Fyzická vrstva je nejnižší vrstva v ISO/OSI modelu, a je zodpovědná za fyzický přenos dat mezi zařízeními, a to konkrétně pomocí seriové komunikace pomoci standartu RS-485, nebo pomocí optického vlákna [4].

2.2.2 Profinet

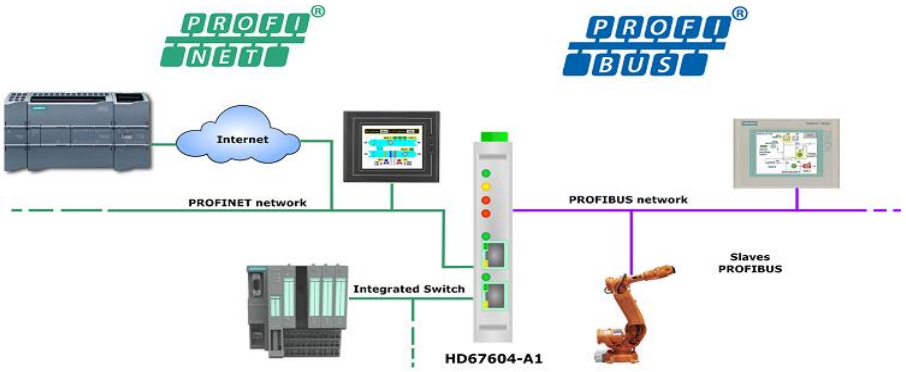
Profinet je další průmyslovou sběrnicí, která je otevřeným standardem. Původně vychází ze standartu ethernet, a vznikla především kvůli větší potřebě zapojit do průmyslu víc automatizačních a komunikačních prvků. Využívá 1,2,3,4 a 7 vrstvu modelu ISO/OSI a je specifikována evropským standardem EN 50173 [4] [5].

Díky tomu profinet umožňuje nejen relativně rychlou komunikaci ale i lepší integraci výrobních systémů jako celku, a to hlavně nadřazených systémů (PC,server,cloud…), které řídí celkový proces, ať už se jedná o výrobu nebo jakýkoliv jiný složitější automatizační systém.

2.2.3 Profibus vs Profinet

Jelikož jsou obě sběrnice využívané především v průmyslových linkách, v praxi se používají obě, a navzájem se mohou doplňovat. Zatímco Profinet, díky svým vysokým propustným rychlostem a lepší integrací do komunikační sítě se může využít hlavně pro nadřazené systémy výroby, Profibus pak může plnit komunikaci mezi PLC či embedded kontroléry a roboty ve výrobě.

Lze očekávat, že s nástupem průmyslu 4.0 bude sběrnice Profibus postupně mizet a budou ji nahrazovat novějšími, a to hlavně kvůli nízké rychlosti přenosu dat v porovnání s ostatními, jelikož se dá čekat, že právě rychlost a objem přenosu dat bude hlavní ideou právě průmyslu 4.0.



Obrázek 3: Možná kombinace sběrnicí Profinet a Profibus, převzato z [14]

2.2.4 DeviceNet

DeviceNet je průmyslová sběrnice využívána především v USA. Poskytuje komunikaci mezi velkým množstvím průmyslových systémů, ať už se jedná o PLC, tak i Embedded PC a ostatní zařízení. DeviceNet spravuje sdružení více než 700 firem ODVA, které se podílí na implementaci této sběrnice do svých výrobků. DeviceNet využívá 1,2 a 5-7 vrstvu ISO/OSI modelu, kde na druhé (linkové) vrstvě využívá standartu CAN (Control Area Network), který byl vyvinut pro automobilový průmysl. Na 5 a vyšší vrstvě využívá standartu CPI (Common Industrial Protocol), což je objektově orientovaný protokol, který dává možnost programovat jednotlivé procesy [6].

2.2.5 CAN

Sběrnice CAN není úplně typickou průmyslovou sběrnicí, jelikož vznikla hlavně pro potřeby automobilového průmyslu. Ovšem její vlastnosti jako například relativně vysoká rychlost komunikace, dobrá bezpečnost, snadná rozšiřitelnost a nízká cena se tato sběrnice začíná používat i v obecném odvětví průmyslu [7]. Jedinou možnou nevýhodou je, že tuto sběrnici některé PLC nemusí podporovat, a poté je potřeba zajistit kompatibilitu dalším zařízením.

2.3 Konfigurační soubory

Konfigurační soubory jsou soubory, které mohou ukládat několik různých informací. Většinou se jedná o prvotní nastavení jednotlivých systémů, ale mohou obsahovat i nové informace, třeba takové, které si uživatel navolí sám podle jeho potřeb. Tyto soubory se nachází na všech počítačích ať už na desktop zařízení, serverech, průmyslových počítačů jako PLC nebo robotických manipulátorech.

Formát těchto souborů závisí čistě výrobci, nebo systému, který se používá. V Unixových systémech to jsou soubory s příponami jako .cnf, .conf, .cfg nebo .ini. U systému Windows to mohou být soubory s příponou .sys [8]. Tyto soubory jsou textové soubory a jejich struktura čistě závisí opět na výrobci nebo na používaném systému.

Ve většině náročnějších aplikací (např. webové aplikace, hry…), kde je potřeba uchovávat větší množství dat i se strukturou se používají serializační formáty jako

JSON, XML, YAML [8]. Struktura těchto formátů je předem definovaná a musí dodržovat daný standard, který každý z těchto formátů má.

app.json

```
{
  "pages": [
    "pages/index/index",
    "pages/logs/logs"
  ],
  "window": {
    "backgroundTextStyle": "light",
    "navigationBarBackgroundColor": "#fff",
    "navigationBarTitleText": "WeChat",
    "navigationBarTextStyle": "black"
  },
  "sitemapLocation": "sitemap.json"
}
```

Obrázek 4 - Ukázka konfiguračního souboru ve formátu JSON, převzato z [15]

2.4 Programování desktopových aplikací

Pro programování pro desktop aplikace se může využít několik typů programovacích jazyků. Jejich zvolení závisí v prvé řadě na operačním systému, pro který se program vyvíjí. Dalším kritériem může být rychlost programu a taky osobní preference programátora, popřípadě kompromis.

2.4.1 Historie

První programy pro desktop aplikace vznikaly jako konzolové aplikace, bez žádné grafické vizualizace. Toto bylo zapříčiněno tím, že grafická reprezentace čehokoliv zabírá poměrně hodně paměti, jelikož si počítač musí pamatovat každý pixel, který vykresluje, což v dřívějších počítačích bylo zbytečné plýtvání. Tyto první programy se psali například v jazyku symbolických adres neboli Assembleru vyvinutého v 50. letech.

Následně se vyvíjeli první programovací jazyky jako například Fortran nebo Algol, určené především k vědeckým výpočtům. Dále se objevovali jazyky jako Pascal, C, C++, které se ve větší nebo menší míře používají dodnes [9]. S tím, jak více lidí vlastnilo počítače, začalo se objevovat i stále více grafických programů, a tedy i pro uživatele jednoduššího ovládání.

2.4.2 Rozdělení programovacích jazyků

Programovací jazyky lze rozdělit podle několika kritérií, zde je uvedeno pouze základní dělení na vyšší a nižší programovací jazyky, přičemž vyšší lze dále dělit na

interpretované a kompilované. Nelze jednoznačně říci, který je lepší, záleží na kritériích konkrétní aplikace.

2.4.2.1 Nižší programovací jazyky

Nižší programovací jazyky se vyznačují tím, že nemají téměř žádnou úroveň abstrakce, což se většinou projeví na přehlednosti psaného programu, a tudíž i daleko horší orientaci v kódu. Existují dvě generace těchto jazyků. První generace se vyznačuje psaním kódu přímo v instrukcích procesoru. Tento typ jazyků poskytují nejlepší "napojení" na hardware, neboť se programuje přímo na úrovni strojových instrukcí procesoru a kvůli tomu nejsou přenositelné na jiné typy procesorů [10]. V porovnáním s vyššími jazyky jsou ty nejrychlejší a nejméně náročné na paměť. Na vývoj běžných aplikací se nepoužívá a dnešní využití se prakticky nevyplácí.

Druhá generace již poskytuje zápis v lidštější formě, poskytuje klíčová slova, která alespoň trochu usnadňují psaní kódu, i když tyto slova jsou stále závislá na konkrétních procesorech. Typickým zástupcem je Assembler. Opět, na vývoj běžných aplikací se téměř nepoužívá, ale stále se mohou uplatnit tam, kde je požadovaný velmi rychlý a spolehlivý chod programu nebo je zapotřebí manipulovat přímo s hardwarem, jako jsou například ovladače hardwaru [10].

Občas se do nižších jazyků řadí i jazyk C, který je nezávislý na vývojové platformě, ale stále umožňuje „sahání" do hardwaru, například manipulaci s pamětí. Avšak pokud srovnáme psaní kódu v jazyce C a v Assembleru, jazyk C poskytuje daleko větší abstrakci.

```
L0:  MOV   R1, #a      ; Address of a
     MOV   R2, #b      ; in R1, of b in R2

L1:  LD    R3, (R1)    ; Inport bits in R3
     CMP   R3, #0      ; IF-condition
     BNE   L3          ;

L2:  MOV   R4, #1      ; IF-branch
     JMP   L4          ;

L3:  MOV   R4, #0      ; ELSE-branch

L4:  ST    (R2), R4    ;
     JMP   L1          ;
```

Obrázek 5 - Ukázka kódu v jazyce assembler, převzato z [16]

2.4.2.2 Vyšší programovací jazyky

Vyšší programovací jazyky jsou úrovní abstrakce daleko příznivější pro člověka, a to hlavně v psaní přehlednějšího kódu a díky tomu i v následné orientaci v kódu. To je dobré nejen pro autora programu ale i pro eliminaci chyb v programu [11]. Programování jako takové je již samotnému hardwaru velmi vzdálen, což je kompenzováno zase rychlostí programu, i když v dnešní době poměrně výkonných osobních počítačů toto nepředstavuje pro většinu aplikací zásadnější problém. Ve vyšších programovacích jazycích se často užívá objektivně orientovaného programování, popřípadě různých návrhových vzorů.

V dnešní době se pro vývoj desktopových, popřípadě webových aplikací používají právě vyšší programovací jazyky. Mezi ně můžeme řadit například C#, Java, PHP, Python, C++, Swift.

```
 7
 8import weather
 9
10#
11# Categorize temperatures as "cool" or "warm"
12# for Blacksburg, VA. Temperatures are in
13# Fahrenheit degrees.
14#
15
16temperatures = weather.get_forecasts("Blacksburg, VA")
17cool = 0
18warm = 0
19for temp in temperatures:
20    if temp < 70:            # cool is below 70
21        cool = cool + 1
22    elif temp >= 85:         # warm is at least 85
23        warm = warm + 1
24print(cool)
25print(warm)
26
```

Obrázek 6 - Ukázka kódu v jazyce Python, převzato z [17]

2.4.2.3 Kompilované a interpretované vyšší programovací jazyky

Kompilační programovací jazyky jsou takové jazyky, jejichž kód se musí nejprve celý zkompilovat, tj. přeložit do strojového jazyka. Teprve poté mohou být spuštěny. Výhody tohoto procesu jsou hlavně ty, že se může okamžitě odhalit chyba v syntaxi kódu, což dělá celkově program spolehlivější, a díky již přeloženému kódu do strojového jazyka také rychlejším. Hlavní nevýhoda kompilačních programů je ten, že při malé změně kódu se musí celý program opět zkompilovat, což v případě velmi rozsáhlých programů může být nezanedbatelné. Mezi kompilované programovací jazyky se řadí například C, C++, Go nebo Pascal.

Interpretované jazyky oproti kompilovaným se nemusí jako celek překládat do strojového jazyka, a nevzniká tedy samotný program, který by byl sám od sebe spustitelný. Místo toho se program spouští pomocí interpreta, který vykonává program řádek po řádku a syntax kódu se kontroluje až při čtení. Výhody takových jazyků je ta, že není potřeba kompilovat celý program a navíc, pokud je samotný interpret multiplatformní, pak se i každý program stává multiplatformním. Ovšem kvůli absenci kompilátoru se případná syntaktická chyba objeví až za chodu, což se bez dostatečného testování může projevit jako fatální chyba. Mezi další nevýhody patří i menší rychlost, což je dáno právě interpretem, který kód překládá řádek po řádku v „real-time" [12].

Avšak většina dnes používaných jazyků lze řadit mezi tzv. "Intermediate" programovací jazyky. Ty se nejprve přeloží do přechodného jazyka, známého jako bytecode. Ten pak mohou zpracovat interpreti nebo být zkompilován na různých platformách a program tedy může být přenositelný a nezávislý na platformě. Mezi takové jazyky patří např. Java, C Sharp nebo Python [23].

2.5 Objektově orientované programování – OOP

Objektově orientované programování je způsob programování, který poskytuje abstrakci kódu, ulehčuje čitelnost, psaní, testování, a tedy i následnou údržbu a rozšiřitelnost kódu. Programátor tvoří v kódu třídy, což je jakýsi vzor, který říká, co má mít objekt za vlastnosti, metody, jaké má mít proměnné atd. Podle těchto tříd se následně generují objekty (tzv. instance tříd), které mají již konkrétní vlastnosti.

Tyto třídy reprezentují nějaký menší celek programu, měli by splňovat základní principy OOP, a mají na starost menší celky funkčnosti celého programu. Díky tomuto se může také urychlit vývoj dalších programů, jelikož se v kódu tyto objekty mohou použít opakovaně, a tedy splňují funkci jakýchsi "stavebních bloků". Tyto stavební bloky je pak možno „svázat" dohromady pomocí tzv. knihoven.

Knihovny v programování pak většinou poskytují již vyřešený problém, a programátor pak pouze pracuje s „black boxem", který mu poskytuje kompletní řešení k nějaké složitější operaci. Knihovny tedy urychlují vývoj – nemusí se znovu vynalézat kolo.

V praxi se pak může využívat OOP společně s návrhovými vzory, což jsou dané postupy při řešení problémů v návrhu programu tak, aby zajistili lepší čitelnost kódu, a tedy i udržitelnost kódu, popřípadě jeho další rozšiřitelnost.

2.5.1 Principy OOP

Za základní principy objektově orientovaného programování, které by jednotlivé třídy měli splňovat můžeme považovat následující vlastnosti.

2.5.1.1 Zapouzdření

Pokud má nějaká třída (objekt) plnit nějakou funkci, například vracet obsah textového souboru, pak samotné čtení souboru může představovat několik úkonů, které plní jednotlivé funkce nebo metody uvnitř třídy.

Tyto funkce a metody ale nezbytně nemusí být přístupné pro ostatní třídy s kterými může třída komunikovat. Místo toho může poskytovat jen jednu funkci, která celý proces spustí a výsledek pak vrátí. Pro jinou třídu (popřípadě pro programátora) se tedy jeví jako jakýsi "black box", který i třeba na základě vstupních argumentů vrátí požadovaný obsah.

2.5.1.2 Dědičnost

Pokud se dá považovat jedna třída (objekt) jako nadřazená jiné třídě (objektu) např. objekt vůz je nadřazený objektu auto, dodávka atd. pak je velmi pravděpodobné, že tyto objekty (třídy) podřazené budou potřebovat používat stejné vlastnosti, metody, funkce atd.

Pak tyto podřazené třídy mohou dědit právě vlastnosti, funkce a metody z třídy nadřazené. Pokud je pak potřebná změna například nějaké metody, pak se tato změna

projeví u všech tříd, které dědí tyto metody. Tímto se zrychluje a zjednodušuje psaní kódu a minimalizuje se počet chyb.

2.5.1.3 Polymorfismus

Velmi souvisí s dědičností. Polymorfismus nám říká, že každá třída, která dědí metody nebo funkce může každou tuto metodu či funkci implementovat jinak. Například, pokud třída, od které se dědí je reprezentace zaměstnance a obsahuje metodu, která zajišťuje že zaměstnanec má pracovat, pak třída reprezentující účetní bude mít jistě v metodě pro práci jiné úkony než třeba uklízečka, nebo dělník.

2.6 Programovací jazyk C Sharp a Visual Studio

C# (C Sharp), je vyšší programovací, objektově orientovaný jazyk vyvinutý společností Microsoft. Je určený především pro programování desktopových ale i serverových, i mobilních aplikací. C Sharp patří do ekosystému .NET, což je multiplatformní, open-source technologie pro vývoj různých aplikací.

C Sharp vznikl v roce 2000, aby poskytl jednoduchý, moderní a objektově orientovaný jazyk. Narozdíl od jazyka jako např. C++ disponuje tzv. garbage collectorem, což znamená, že si sám hlídá využití paměti, což je další komfort pro vývojáře, a zjišťuje tak i další eliminaci chyb v programu [13].

Microsoft nabízí vývojové prostředí Visual Studio, což je kompletní řešení pro vývoj programů. Obsahuje jak kompilátor, tak debugger ale i jednoduché prostředí pro návrh grafického rozhraní programu tzv. GUI (Graphical User Interface), kde jsou předdefinované prvky, které se očekávají v každé grafické aplikaci, jako jsou tlačítka, popisky, textové boxy atd.

Do prostředí Visual Studia navíc lze importovat prostředí třetích stran. Například firma Beckhoff vyvíjí software pro programování PLC počítačů s názvem Twincat 3, který běží přímo ve vývojovém prostředí Visual Studia.

2.6.1 C Sharp v rámci OOP

Aby bylo možné psát objektově orientovanou aplikaci v programovacím jazyku C Sharp, je nutné znát několik pojmů a klíčových slov, které se během programování mohou použít.

2.6.1.1 Přístupy k metodám a funkcím

Toto téma souvisí silně se zapouzdřením. Na základě třech klíčových slov tedy můžeme omezovat viditelnost funkce a metody pro další třídy. Toto zlepšuje jak přehlednost, tak údržbu kódu.

Pokud chceme například, aby nějaká metoda byla viditelná pouze uvnitř třídy (tzn. nebude ji moci volat ani potomek co od třídy dědí, ale ani žádná jiná) použijeme klíčové slovo „private“ před samotnou deklarací této metody (obdobně u funkcí).

```
private bool isTopicExist(string topic,List<string> lines)
{
    bool ret = false;
    foreach(string line in lines)
    {
        if (line.Contains(topic))
        {
            ret = true;
        }
    }
    return ret;
}
```

Obrázek 7 - Ukázka použití klíčového slova "private"

Jestli má metodu používat jak nadřazená třída ale i její potomek, pak se použije před deklarací klíčové slovo „protected“. Tím se zajistí, že funkce či metoda je viditelná jen v samotné třídě a ve třídách které z ní dědí.

Posledním klíčovým slovem je „public“. Přidáním tohoto slova před deklaraci zpřístupníme onu metodu či funkci všem třídám, bez ohledu na to, zdali jsou potomky či ne.

Tyto klíčová slova lze použít i pro proměnné, avšak přistupovat přímo k proměnným z vnější je špatně. K tomuto slouží tzv. „getter“ a „setter“, což jsou jakési proměnné, které ale dokážou předávat („getter“) nebo přepsat („setter““) svoje hodnoty. Těmto proměnným se říká vlastnosti a deklarují se přidáním klíčových slov „get“ nebo „set“ za název vlastnosti (proměnné) ve složených závorkách, protože po těchto klíčových slovech může následovat i další kód, který bude výstup nebo vstup vlastnosti nějakým způsobem zpracovávat.

```
public int Month
{
    get => _month;
    set
    {
        if ((value > 0) && (value < 13))
        {
            _month = value;
        }
    }
}
```

Obrázek 8 - Ukázka použití "get" a "set" v jazyce C Sharp, převzato z [20]

2.6.1.2 Vytvoření třídy

Pro základní vytvoření třídy se použije klíčové slovo „class“. Následuje uzavření do závorek. Třídu je možno deklarovat téměř kdekoliv v kódu, avšak pro lepší přehlednost se zpravidla deklaruje v samostatném souboru.

```
class ErrorDataModel
{
    Počet odkazů: 13
    public bool isError { get; set; }
    Počet odkazů: 13
    public string errorMessage { get; set; }

}
```

Obrázek 9 - Ukázka vytvoření třídy v jazyce C Sharp

2.6.1.2 Abstraktní třídy

Abstraktní třídu lze chápat jako nějakou obecnou nadřazenou kategorii, která svazuje ostatní třídy, a to tak, že může předepisovat jaké vlastnosti, ale i funkce a metody musí mít, avšak potomci třídy mohou obsahovat i další a odlišné vlastnosti, funkce atd. Hlavní je ale to, že abstraktní třídu jako takovou nelze vytvořit (její instanci). Instance se tvoří až z podřadných tříd, které dědí nadřazenou abstraktní třídu.

Příkladem abstraktní třídy může být třída „Vozidlo“. Řekněme, že tato třída bude implementovat metodu „ZapniSvětla()“. Z této abstraktní třídy pak budou dědit třídy „Auto“, „Autobus“, „Motorka“ atd. Pokud následně budeme chtít zapnout světla všem vozidlům, nemusíme procházet jednotlivé objekty reprezentující třídy podřazené, ale metodu můžeme zavolat na všechny objekty „Vozidlo“.

V jazyce C Sharp se abstraktní třída předepisuje klíčovým slovem „abstract“. Toto klíčové slovo musí být také před jakoukoliv proměnou a také funkcí či metodě, která se neimplementuje. Potomci, kteří pak implementují jednotlivé metody a funkce pak musí před názvem této metody nebo funkce použít klíčové slovo „override“.

```
abstract class Checker
{
    Počet odkazů: 8
    public abstract ErrorDataModel check();
}
```

Obrázek 10 - Ukázka abstraktní třídy

2.6.1.2 Použití dědičnosti

Dědičnost se v jazyce C Sharp deklaruje pomoci přidáním znaku „:“ za název u deklarace třídy, po které následuje název třídy, ze které se má dědit. Podle moderních postupů by se v rámci přehlednosti nemělo dědit z více jak jedné třídy. To si také C Sharp ohlídá a pokud potomek dědí z více tříd, zahlásí C Sharp chybu. Jiný případ je pro tzv. rozhraní či interface, kde je povolena vícenásobná dědičnost.

```
class CfgSYSSIG_OUT : CfgRecord
{
    Počet odkazů: 6
    public override string TypeOfRecord { get; } = "SYSSIG_OUT";

    Počet odkazů: 6
    public override string[] parametersNeeded { get; set; } = { "Status", "Signal" };
    Počet odkazů: 12
    public override Dictionary<string, string> parametersInCfg { get; }
    Počet odkazů: 8
    public override string rawLine { get; set; }
    Počet odkazů: 6
    public override XRecord XlsRecordTwin { get; set; }
    Počet odkazů: 1
    public CfgSYSSIG_OUT(string line)...
}
```

Obrázek 11 - Ukázka použití dědičnosti v jazyce C Sharp

2.6.1.2 Rozhraní (Interface)

Rozhraní (anglicky Interface) předepisuje třídě, co má vlastně být schopna vykonávat za akce. Tyto akce jsou reprezentovány metodami.

Například, pokud bychom chtěli udělat třídu, která by měla reprezentovat dopravní prostředek, jistě musí umět přinejmenším vykonat akci „jeď“ a „brzdi“. Tyto dvě akce můžeme tedy přidat do rozhraní, které pak bude zděděno třídou dopravního prostředku a podle něj také implementováno – jistě bude jinak akce „brzdi“ implementována v autě a jinak ve vlaku.

Klíčové slovo pro vytvoření rozhraní v jazyce C Sharp je „interface“. Poté následuje seznam metod, které má interface obsahovat. Třída pak zdědí tento interface.

```
interface IFile
{
    void ReadFile();
}

interface IBinaryFile
{
    void OpenBinaryFile();
    void ReadFile();
}

class FileInfo : IFile, IBinaryFile
{
    void IFile.ReadFile()
    {
        Console.WriteLine("Reading Text File");
    }
```

Obrázek 12 - Ukázka použití rozhraní v jazyce C Sharp, převzato z [21]

3. Praktická část

Praktická část se věnuje jak seznámení se s jednotlivými programy využívané pro konkrétní PLC, tak hlavně návrhu a dokumentaci samotného programu který je věcí praktické části této práce. Program slouží hlavně pro ulehčení práce s nastavováním nových signálů pro roboty ve výuce.

3.1 Robot Studio 2020

Program Robot Studio od firmy ABB je především nástroj pro návrh virtuálního dvojčete reálné výrobní linky. Je tak možné testovat tuto linku ještě před samotným spuštěním. Toto může vést k odladění chyb, a tedy i k snížení ceny konečné reálné linky, popřípadě i její zlepšení. Po uvedení reálné linky může tato virtuální linka sloužit jako real-time monitorovací systém linky. Také je možno tento program využít jako platformu pro programování těchto robotů a PLC, zvlášť od firmy ABB. V této práci ale program slouží především jako prostředním pro nahrání vygenerovaného konfiguračního souboru.

3.2 Siemens TIA Portal

Jeden z nejpoužívanějších vývojových prostředí pro PLC je TIA Portal od firmy Siemens. Toto vývojové prostředí nabízí široké spektrum možností. Prakticky je možné naprogramovat celou výrobní linku pouze za pomoci tohoto nástroje, včetně konfigurace, nastavení bezpečnosti až po finální testování a odlaďování chyb.

V rámci této práce se TIA Portal používá právě pro nastavování signálů pro jednotlivé roboty. Avšak nastavování těchto signálů tak, aby si odpovídali jak v prostředí TIA Portalu a potom i v robotu je velmi zdlouhavá činnost. Právě proto se pracuje jen s exportem těchto signálů do tabulky ve formátu .xlsx, ze které se poté vygeneruje konfigurační soubor.

3.3 Konfigurační soubor ABB

Soubory, které se používají pro konfiguraci robotů ABB je celá řada a každý má jiný účel. Jedná se o textové soubory ve formátu .cfg popisující komunikaci, bezpečnostní funkce (safety), konfiguraci pohybových os a další. Pro tuto práci je důležitý konfigurační soubor pro vstupní a výstupní signály který se jmenuje EIO.cfg.

Tento soubor se skládá z několika částí, které popisují jednotlivá nastavení. Jedná se například o systémové signály, které mohou poskytovat informace o konkrétním stavu motorů, pozici atd. Dále pak nastavení sběrnice, v tomto případě Profinet, tedy informace o jménu, popisu, IP adresy stroje. Jednotlivé sekce jsou odděleny pomocí

znaku „#“ s vynechaným řádkem a následně názvem např. PROFINET_INTERNAL_DEVICE.

Tato práce, a především program ale pracuje s nastavením vedený pod názvem EIO_SIGNAL. Tato část nastavení popisuje logickou softwarovou reprezentaci vstupních a výstupních signálů které mohou být buď přímo na zařízení, které je připojeno do industriální sítě, ve které je i ovládaný systém, nebo také mohou sloužit pro simulaci v programu Robot Studio zmíněného výše. [18]

```
-Name "RB1_PN_Q_CMD_MCV_StartMachining" -SignalType "DO"\
-Device "PN_Internal_Device" -DeviceMap "72"

-Name "RB1_PN_I_Backup" -SignalType "DI" -Device "PN_Internal_Device"\
-DeviceMap "0"
```

Obrázek 13 - Ukázka popisu signálu EIO v konfiguračním souboru

Jak lze na obr. 13 vidět, popis signálu se skládá z několika parametrů jako jsou např. Name (jméno signálu), SignalType, který může nabývat DI/D0 (Digital Input – digitální vstup, Digital Output – digitální výstup), respektive AI/AO (Analog Input – analogový vstup, Analog Output – analogový výstup) nebo GI/GO (Group Input/Output – „skupinový“ vstup/výstup). Dále i Device, což je zařízení, přes které se signály posílají (v této práci se jedná především o PN_Internal_Device – což říká že se využije interní zařízení pro komunikaci přes sběrnici Profinet) a nakonec DeviceMap, což popisuje, který bit signál používá v paměti vstupně/výstupního zařízení [18].

3.4 Popis programu

3.4.1 Úvod

Celý program je napsaný v jazyce C Sharp a je „stavěn“ pomocí objektově orientovaného přístupu hlavně z důvodu jednodušší čitelnosti kódu a možné budoucí rozšiřitelnosti. Jako vývojové prostředí bylo zvoleno Visual Studio. Hlavním cílem tohoto programu je ulehčit práci se signály pro jednotlivé roboty, pomocí nichž komunikují s PLC.

Konkrétně se jedná o přepsání signálů z excelovské tabulky, která vznikne exportováním signálů ve vývojovém prostředí pro PLC - TIA Portal, do konfiguračního souboru, který již bude využívat robot a o kterém se tato práce zmiňuje výše. Tento soubor lze do robota nahrát pomocí aplikace RobotStudio.

3.4.2 Použité knihovny třetích stran

Ačkoliv výchozí knihovny dodávané společností Microsoft ve vývojovém prostředí Visual Studio poskytuje přímo knihovnu pro práci s Excelovskými soubory, funguje pouze pokud je na cílové stanici nainstalován program MS Excel. Kvůli tomu při vývoji aplikace byla použita open source knihovna Excel Data Reader, která umožňuje jednoduchou práci se čtením a zápisu do souborů typu excel (v tomto případě souborů

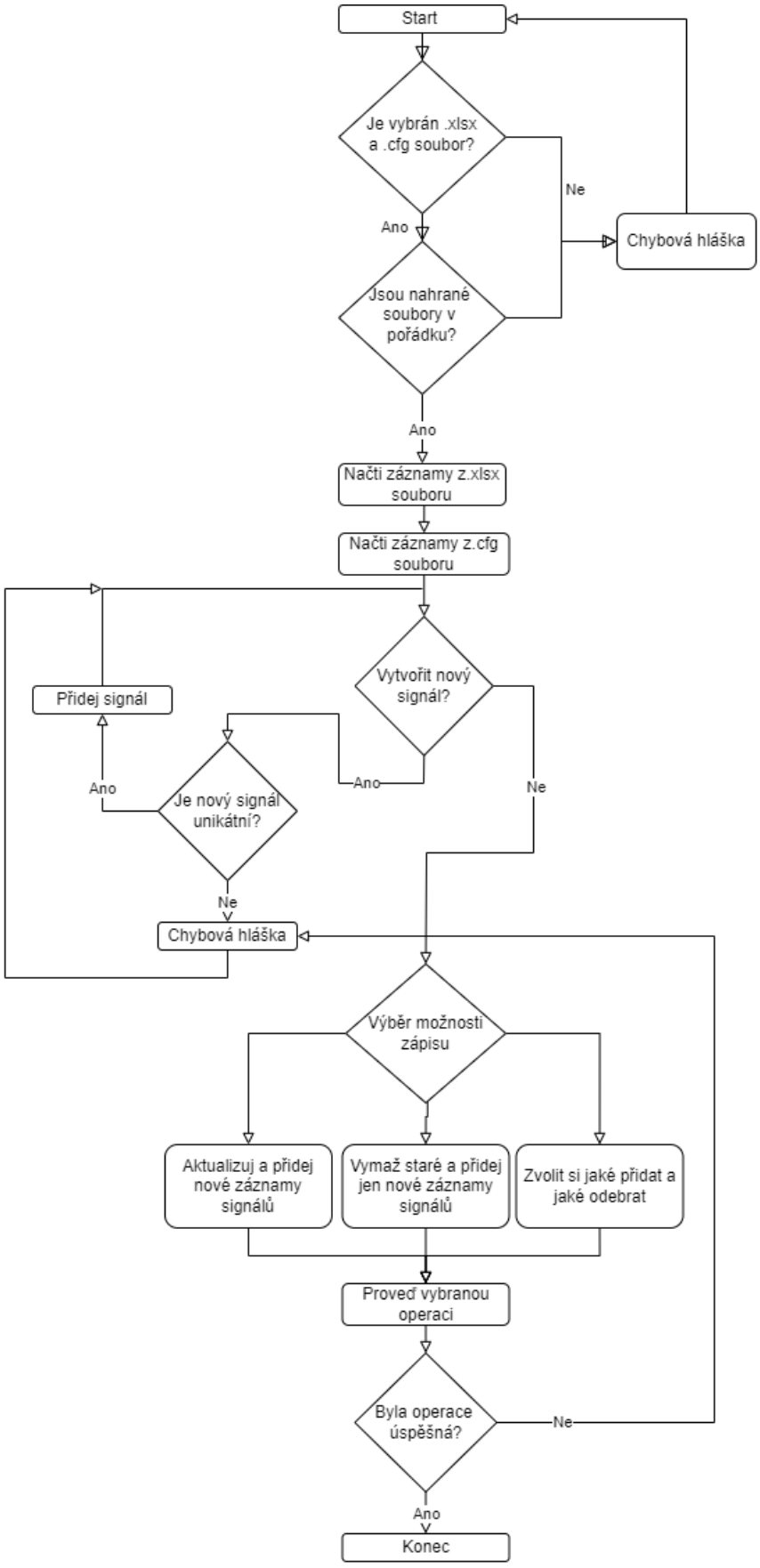
typu .xlsx) [19]. Velkou výhodou je pak to, že není nutno mít přímo nainstalovaný program MS Excel.

3.4.3 Základní popis fungování aplikace

Celá aplikace se skládá ze dvou formulářů. V prvním, ve kterém si uživatel vybere oba soubory, tedy vyexportovaný .xlsx soubor z aplikace TIA Portal a také již existující konfigurační .cfg soubor z aplikace RobotStudio. Po nahrání těchto souborů se jednotlivé signály zobrazí jako grafické záznamy, se kterými lze pracovat, pokud si uživatel zvolí v pravém horním rohu mód „Select individualy“, kde si zvolí jen ty signály, které chce přidat. Také je možno si zvolit další dva módy, buď přidat nové a aktualizovat stávající záznamy (volba „Append new xls and update existing cfg signals“), nebo odebrat staré a přidat jen nové (volba „Delete old cfg and add new xls signals“).

Druhý formulář se objeví po kliknutí na tlačítku „Create new signal“, kde uživatel může narychlo přidat nový signál. Po vybrání módu zápisu stačí spustit přepsání souboru pomocí tlačítka „Create Cfg File“ dole vpravo. Krom toho lze v souboru App.config nastavit aplikaci, jak má dané signály pojmenovat v .cfg souboru pod parametrem „DeviceName“, jelikož se nutně nemusí jednat stále jen o hodnotu „PN_Internal_Device“. Dále lze nastavit i jak bude v excelovském souboru uvedeno, zdali se jedná o vstup či výstup, a taky jaký (pokud jestli) komentář následuje v excelovském souboru pro hodnotu, která má sloužit jako parametr „DeviceMap“. Také tu lze nastavit, jestli má program při vytváření signálů rozlišovat, jestli se můžou hodnoty parametru „DeviceMap“ překrývat, tedy jestli se bude rozlišovat mezi pamětí vstupní a výstupní, či nikoliv.

Následující vývojový diagram popisuje základní funkčnost programu.



Obrázek 14 - Základní diagram fungování aplikace

3.4.4 Popis jednotlivých tříd

Následující text obsahuje popis jednotlivých tříd, které byly vytvořeny, neobsahuje tedy popis těch tříd, které Visual Studio již obsahuje a jejich dokumentaci lze dohledat na oficiálních stránkách vývojářů vývojového prostředí Visual Studio.

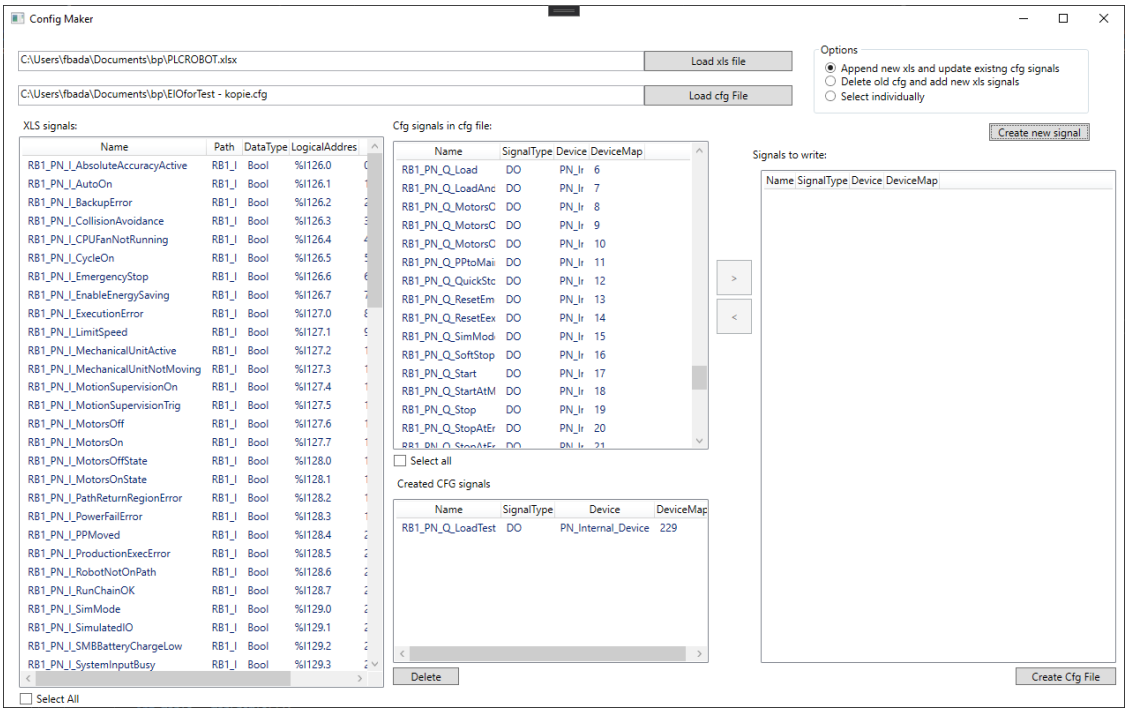
V zájmu zachování principů objektově orientovaného programování jsou některé třídy tzv. abstraktní. Tyto abstraktní třídy obsahují jen jakýsi popis a souvisí s polymorfismem, který je řešený v teoretické části. V praxi to potom znamená že je tato třída jen jakýsi návod, co určitě musí třída obsahovat – jaké funkce, metody, avšak v nich nemusí být uvedena konkrétní implementace. Implementace (tedy „tělo“ funkce který již obsahuje konkrétní algoritmus) se může objevit až u tříd, které z této abstraktní třídy bude dědit. V tomto programu se jedná o třídy „Checker.cs“ a „CfgRecord.cs“.

Abstraktní třída „Checker.cs“ předepisuje pouze, že děděná třída má obsahovat pouze metodu, která bude „něco“ kontrolovat. Proto se tu nabízí i abstraktní třídu nahradit rozhraním (také spíše známo jako Interface), avšak jelikož by toto rozhraní bylo využito právě v této následující abstraktní třídě, byl tento krok přeskočen.

Další abstraktní třída „CfgRecord.cs“ kromě předpisů proměnných obsahuje i implementaci funkce („mapFromCfg()“), která slouží pro „vytáhnutí“ a uložení dat z .cfg souboru. Jelikož z této třídy dědí třídy, určené pro držení dat, byla by implementace i v děděných třídách stejná.

3.4.4.1 MainWindow.xaml.cs

Celý program po spuštění načítá tuto hlavní třídu, která je zodpovědná za veškerou logiku programu. Taky se tato třída automaticky generuje s grafickým rozhraním ve vývojovém prostředí Visual Studio. Tato třída dědí základní třídy z Visual Studia, aby měla přístup například k vykreslování grafických oken. Podobná třída bude nezbytná u každé podobné formulářové aplikace, která je postavená na platformě .NET. Jak již předpona .xaml napovídá, jednotlivé grafické prvky jsou reprezentovány pomocí XAML tagů. Následně tato třída obsahuje i funkce, které se volají po interakci s uživatelským rozhraním.



Obrázek 15 - Ukázka grafického rozhraní programu

3.4.4.2 Loader.cs

Třída „Loader.cs“ je zodpovědná za kontrolu jak .xlsx tak .cfg souborů společně s voláním uživatelsky přívětivých komponent z balíčku Visual Studio pro snadnější výběr souborů přímo na místním zařízení. Po výběru souboru třída poskytuje cestu ke zvolenému souboru a také volá další kontrolní třídy, které pokud zjistí v souboru chybu (např. že je soubor poškozený), poskytne tyto chybová hlášení k dalšímu zpracování.

3.4.4.3 CfgReader.cs

Stará se o nahrání obsahu .cfg souboru a jeho následného roztřídění do dalších data modelů, aby je bylo možno později porovnávat se záznamy z .xlsx souboru. Stěžejní funkcí této třídy je funkce „mapCfgFile()“. Ta připraví potřebné záznamy signálů ke zpracování do dalších tříd, které slouží jako datové modely, aby je bylo možné později porovnávat (a tedy popřípadě je nějakým způsobem například aktualizovat) a celkově, aby s těmito záznamy bylo jednodušší pracovat v programu. To dělá jednoduše tak, že rozezná každý signál (v .cfg souboru začíná znakem “-“) a poté ho přiřadí do textového řetězce. Funkce pak tyto textové řetězce ukládá do listu, který pak vrací. List je datová struktura podobná poli, ale přináší řadu výhod, jako například dynamické přidávání, mazání atd.

3.4.4.4 XReader.cs

Tato třída má stejný úkol jako třída CfgReader.cs, ovšem pro .xlsx (případně pro .xls) záznamy. Jedná se o uložení záznamů budoucích signálů, na základě kterých se bude konfigurační soubor .cfg aktualizovat či měnit. Právě zde je využita knihovna třetí strany Excel Data Reader. O toto se stará funkce „Read()“ která jako návratovou hodnotu používá opět list, ve kterých jsou tyto záznamy uloženy.

3.4.4.5 ErrorDataModel.cs

Třída slouží jako datový model pro uchovávání případných chybových hlášek, které by se mohli vyskytnout v průběhu vykonávání různých částí programu. Třída také obsahuje informaci, zdali vůbec k chybě došlo.

3.4.4.6 CfgRecord.cs

Jak již bylo zmíněno výše, jedná se o abstraktní třídu, která poskytuje základní „kostru“ pro další třídy, které z ní dědí. Kromě předpisů, které proměnné má dědící třída obsahovat, obsahuje tato abstraktní třída i konkrétní implementaci funkce „mapFromCfg()“ která vezme jednotlivé textové řetězce z datové struktury list, kterou vrátí funkce „mapCfgFile()“ ve třídě CfgReader.cs. Na základě toho, jaké parametry signálu chceme rozdělit (v budoucnu se mohou lišit) funkce tyto parametry rozdělí a vrátí hodnotu parametru.

Pro toto rozdělování je použito vyhledávání pomocí regulárních výrazů. Regulární výrazy jsou nástroj, pomocí kterého lze textový řetězec filtrovat, ale i vyhledávat ověřovat atd. Fungují na základě různých kvantifikátorů, které udávají, co se má v textovém řetězci vyskytovat, kolikrát se má vyskytovat, jaká má být „syntaxe“ textu atd.

3.4.4.7 Xrecord.cs

Třída Xrecord.cs je hlavně datový model pro záznamy z excelovského souboru. Krom tohoto ale i obsahuje funkci „toCfgString()“, která konvertuje tyto záznamy do formátu konfiguračního .cfg souboru ve formátu textového řetězce který funkce vrátí. Tento textový řetězec je pak připraven pro vložení do konfiguračního souboru.

3.4.4.8 CfgEIO_SIGNAL.cs

Jedná se o třídu, která slouží jako datový model EIO signálů z konfiguračního souboru. Dědí z abstraktní třídy CfgRecord. Třída definuje, jaké parametry si má uchovávat. Třída si také uchovává v proměnné „XlsRecordTwin“ identický záznam pro tentýž signál ale ve formátu excelovského souboru. Tato proměnná je užívána, pokud si uživatel ručně vybírá, které signály chce do souboru přidat či odebrat.

3.4.4.9 EIO_Signal_ViewDataModel.cs

Třída slouží jako datový model, která se používá dále ve třídě „ListViewManager.cs“. Se samotnou logikou v programu nemá nic společného a slouží jen pro jednodušší nahrávání do komponenty ListView právě přes výše zmiňovanou třídu.

3.4.4.10 ListViewManager.cs

Třída dělá nezbytné operace k tomu, aby mohli být vidět záznamy pro uživatele v komponentně ListView, která poskytuje tabulkový přehled jednotlivých signálů z obou souborů, což umožňuje uživateli přidávat či odebírat jen signály, které potřebuje.

To dělá pomocí funkce „makeAndBindGridView()“ vytvořením instance třídy „GridView“, která vytvoří jakýsi vzor pro zobrazení dat. Tato instance se pak předá komponentě ListView, která podle tohoto ví, jak budou data řazena. Samotné nahrávání se pak provádí v „hlavní“ třídě „MainWindow.xaml.cs“.

Třída také obsahuje funkci „createEioSignalView()“, která připraví záznamy ze souboru .cfg tak, aby byly akceptovatelné pro komponentu ListView. Pro záznamy ze souboru .xlsx to není potřeba, neboť komponenta ListView již akceptuje data z třídy „XRecord.cs“.

3.4.4.11 Checker.cs

Abstraktní třída, která pouze předepisuje, že děděná třída musí obsahovat funkci „check()“ vracející třídu „ErrorDataModel.cs“. Děděné třídy si ji poté implementují podle sebe, v závislosti na tom, co mají kontrolovat. Z této třídy dědí třídy „XFileChecker.cs“, „XFormatChecker.cs“, „CfgChecker.cs“. Tato třída je pak nezbytná pro práci s kontrolováním souborů společně s třídou „MainChecker.cs“, která pak kontrolu provádí.

3.4.4.12 MainChecker.cs

Třída se stará o spouštění konkrétních kontrol v dalších třídách a to pomocí tzv. delegátu, což je typ, který umí odkazovat na jednotlivé funkce či metody [22]. Tato třída je tedy jakýsi prostředník mezi jednotlivými kontrolami. Díky delegátům dokáže zavolat všechny funkce, na které má odkaz a případně vrátit chybovou hlášku. Tento přístup byl zvolen zvlášť z důvodu možného rozšíření aplikace, jelikož tímto způsobem je další integrace kontrol jednodušší a vyžaduje jen minimální změny v kódu.

3.4.4.13 XFormatChecker.cs

Třída dědí z abstraktní třídy „Checker.cs“ a implementuje funkci „check()“, která kontroluje, jak název napovídá, správnost formátu, tedy jestli je nahrán soubor .xls nebo .xlsx. Pokud ne, funkce vrátí instanci třídy která „ErrorDataModel.cs“ která obsahuje chybovou hlášku. Kontrola probíhá jednoduše pomocí funkce „GetExtension()“ dostupné z tzv. jmenného prostoru (více známé anglické namespace) „Systém.IO“.

3.4.4.14 XFileChecker.cs

Jelikož se jedná o další třídu, která něco kontroluje, opět dědí z třídy „Checker.cs“. Tato třída ověřuje, jestli lze soubor otevřít, tedy jestli není soubor nějakým způsobem poškozen. Funkce „check()“ tu implementuje funkce z knihovny ExcelDataReader,které se snaží otevřít soubor Tyto funkce jsou „obaleny“ standartním zpracováním chyb v jazyce C# a to pomocí bloku „try-catch“. Kód v tomto bloku je pak zpuštěn a v případě že se nedokončí z jakéhokoliv důvodu, vrátí se chybová hláška. Tato chybová hláška je pak zachycena a zpracována tak, aby původní funkce „check()“ vrátila instanci třídy „ErrorDataModel.cs“ s konkrétní hláškou.

3.4.4.15 CfgFileChecker.cs

Třída je obdobou třídy „XFileChecker.cs“. Také kontroluje, zdali lze se souborem vůbec pracovat, a tedy i nahrát záznamy signálů ze souboru .cfg. Kontrola probíhá prakticky identicky jako ve výše zmíněné třídě, avšak v bloku „try-catch“ se nachází funkce dostupná z jmenného prostoru „Systém.IO“, která daný soubor otevře, přečte všechny řádky souboru, a nakonec jej i zavře. Pokud se během tohoto někde objeví chyba, funkce „check()“ opět předá instanci třídy „ErrorDataModel.cs“ s příslušným hlášením.

3.4.4.16 CfgFormatChecker.cs

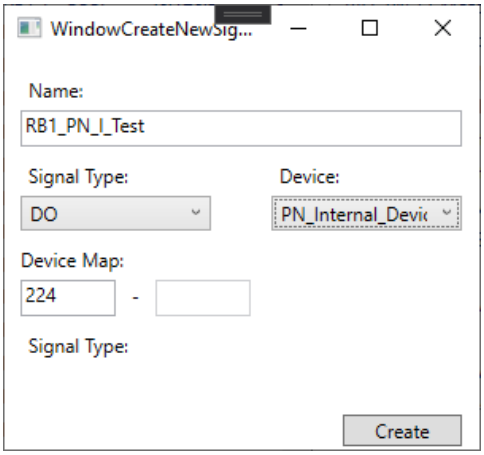
Třída plní stejnou funkci jako třída „XformatChecker.cs“. Také k tomu používá stejné funkce, ovšem jen s úpravou pro kontrolu .cfg přípony.

3.4.4.17 CfgWriter.cs

Tato třída je zodpovědná za finální zápis signálů do .cfg souboru. Třídě se předají všechna potřebná data a také jaký má být „mód“ zápisu. Na základě toho třída buď pouze vloží nové záznamy a aktualizuje stávající, smaže všechny staré potřebné signály a nahraje nové, nebo nahraje jen ty, které si uživatel předtím sám zvolil.

3.4.4.18 WindowCreateNewSignal.xaml.cs

Jedná se o třídu zodpovědnou za logiku druhého formuláře, který zajišťuje přidání nového signálu. Ještě před objevením formuláře se předají všechny nahrané záznamy z předchozího formuláře, které se dále zpracují do jednoho listu, kvůli následné kontrole, o kterou se však stará jiná třída. Po vyplnění údajů ve formuláři se nejprve zkontroluje, zdali uživatel opravdu zadal vše potřebné, pokud ne, upozorní ho chybová hláška. Po této kontrole se vytvoří nový signál, který je předám třídě „DeviceMapValidator.cs“, která zkontroluje, zdali nový signál nepoužívá parametry, které jsou již použity, jako jméno signálu či parametr „DeviceMap“. Pokud kontrola proběhne v pořádku, signál se předá zpět do hlavního formuláře do nového listu s vytvořenými signály. Pak se se signálem pracuje standartně jak s dalšími, s výjimkou toho, že se tyto vytvořené signály přidají vždy, z logiky věci.



Obrázek 16 - Formulář pro přidání nového signálu

3.4.4.19 DeviceMapValidator.cs

Jedná se o třídu, která dědí abstraktní třídu „Checker.cs“ a kontroluje nově vytvořený signál přímo v programu s těmi, které jsou již nahrány, aby nedošlo ke kolizi parametrů. Instance třídy již dostane list všech signálů v podobě cfg formátu a pak si informace důležité pro kontrolu nahraje do nového listu, který udržuje třídy sloužící jako jednoduchý datový model s názvem „DeviceMapInfo.cs“. Poté kontroluje dané parametry a v případě, že zjistí duplicitní hodnoty, vrátí pomocí zděděné funkce „check()“ instanci třídy „ErrorDataModel.cs“ která obsahuje chybovou hlášku. Třída se také volá znovu ve třídě „CfgWriter.cs“, kvůli tomu, že uživatel má možnost přidat signál i bez .xlsx souboru, avšak pokud tento soubor později přidá, není jisté zdali nedojde ke kolizi parametrů.

3.4.4.20 DeviceMapInfo.cs

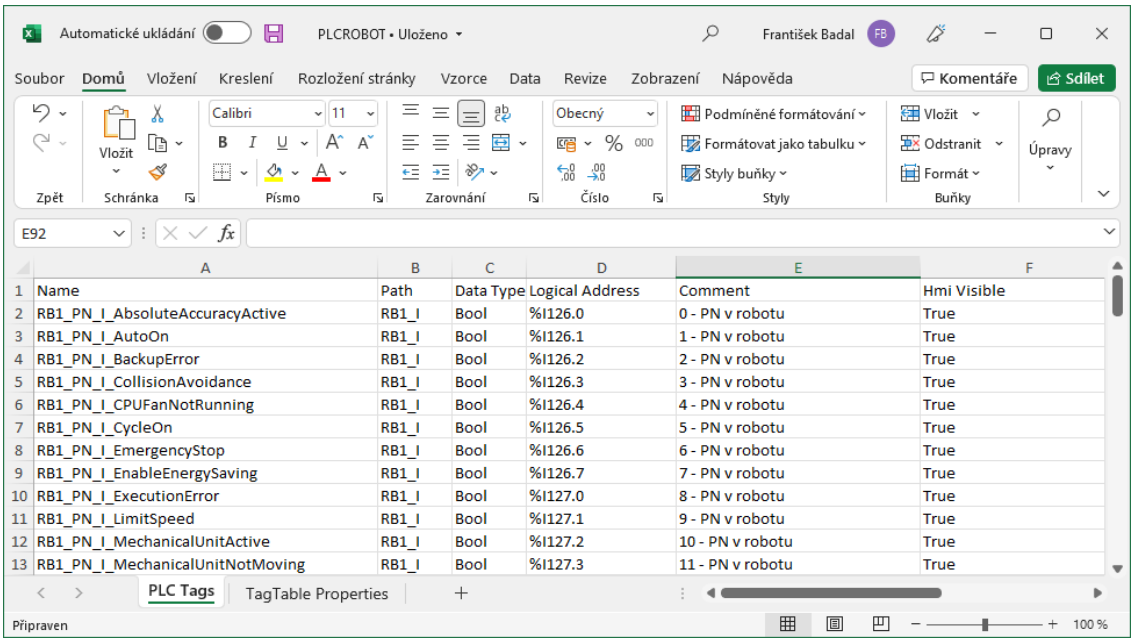
Třída slouží jen jako jednoduchý datový model pro signály, který udržuje potřebné údaje pro snazší ověření případných duplicit v uživatelem vytvořeném signálu. Je užíván třídou „DeviceMapValidator.cs“.

3.4.4.21 App.config

Tento soubor obsahuje konfiguraci parametrů aplikace, které se mohou měnit. Jedná se „DeviceName“, který říká, k jakému zařízení má nový signál přiřadit. Dále parametr „IsInputString“ obsahuje textový řetězec, který má program cíleně hledat ve jméně signálu, jestli tento signál má považovat za vstupní. Výstupní signál pak je označen automaticky ten, který tento řetězec neobsahuje. Parametr „DeviceNameComment“ je pomocný parametr, na základě kterého se ořezají další nepotřebná data ve sloupci „Comment“ v .xlsx souboru. Poslední parametr „InputOutputMemoryDifferent“ slouží k tomu, jestli má aplikace u vytvořeného signálu rozlišovat mezi vstupním a výstupním signálem u parametru „DeviceMap“ v .cfg souboru či nikoliv, tedy jestli se tyto čísla signálů mohou krýt. Pokud ano, nastaví se řetězcem „true“.

3.5 Otestování funkčnosti vygenerovaného konfiguračního souboru

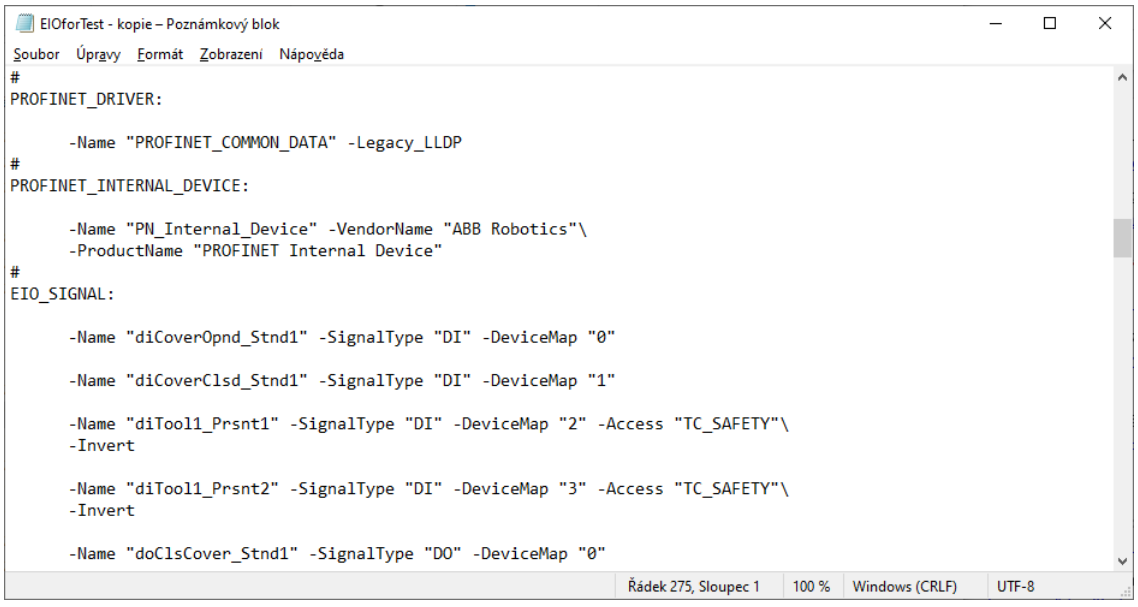
K otestování programu a vygenerovaného souboru byl použit již vygenerovaný .xlsx soubor z programu TIA Portal, který má několik signálů.



	A	B	C	D	E	F
1	Name	Path	Data Type	Logical Address	Comment	Hmi Visible
2	RB1_PN_I_AbsoluteAccuracyActive	RB1_I	Bool	%I126.0	0 - PN v robotu	True
3	RB1_PN_I_AutoOn	RB1_I	Bool	%I126.1	1 - PN v robotu	True
4	RB1_PN_I_BackupError	RB1_I	Bool	%I126.2	2 - PN v robotu	True
5	RB1_PN_I_CollisionAvoidance	RB1_I	Bool	%I126.3	3 - PN v robotu	True
6	RB1_PN_I_CPUFanNotRunning	RB1_I	Bool	%I126.4	4 - PN v robotu	True
7	RB1_PN_I_CycleOn	RB1_I	Bool	%I126.5	5 - PN v robotu	True
8	RB1_PN_I_EmergencyStop	RB1_I	Bool	%I126.6	6 - PN v robotu	True
9	RB1_PN_I_EnableEnergySaving	RB1_I	Bool	%I126.7	7 - PN v robotu	True
10	RB1_PN_I_ExecutionError	RB1_I	Bool	%I127.0	8 - PN v robotu	True
11	RB1_PN_I_LimitSpeed	RB1_I	Bool	%I127.1	9 - PN v robotu	True
12	RB1_PN_I_MechanicalUnitActive	RB1_I	Bool	%I127.2	10 - PN v robotu	True
13	RB1_PN_I_MechanicalUnitNotMoving	RB1_I	Bool	%I127.3	11 - PN v robotu	True

Obrázek 17 - Exportovaný excel soubor z TIA Portalu

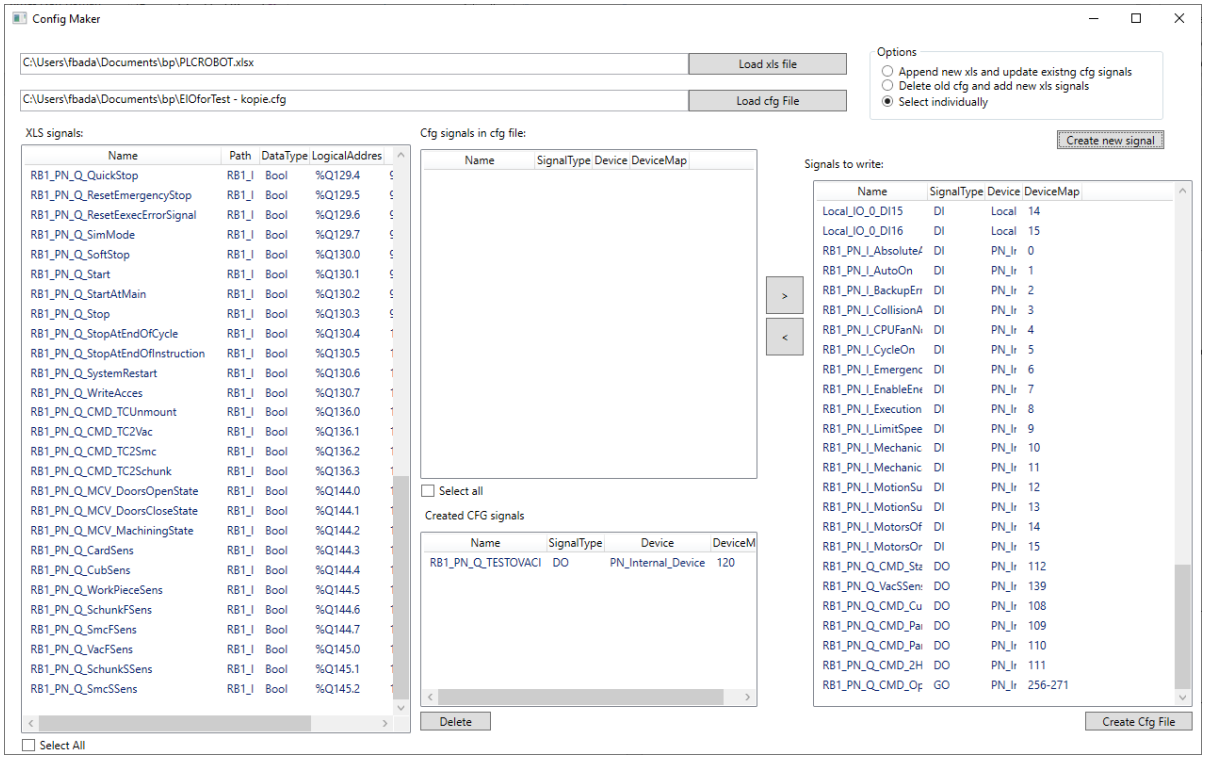
Konfigurační soubor .cfg je použit základní, exportovaný z Robot Studia s již nastavenými parametry, kromě právě signálů typu „PN_INTERNAL_DEVICE“, které se vygenerují automaticky pomocí programu.



```
#
PROFINET_DRIVER:

      -Name "PROFINET_COMMON_DATA" -Legacy_LLDP
#
PROFINET_INTERNAL_DEVICE:

      -Name "PN_Internal_Device" -VendorName "ABB Robotics"\
      -ProductName "PROFINET Internal Device"
#
EIO_SIGNAL:

      -Name "diCoverOpnd_Stnd1" -SignalType "DI" -DeviceMap "0"

      -Name "diCoverClsd_Stnd1" -SignalType "DI" -DeviceMap "1"

      -Name "diTool1_Prsnt1" -SignalType "DI" -DeviceMap "2" -Access "TC_SAFETY"\
      -Invert

      -Name "diTool1_Prsnt2" -SignalType "DI" -DeviceMap "3" -Access "TC_SAFETY"\
      -Invert

      -Name "doClsCover_Stnd1" -SignalType "DO" -DeviceMap "0"
```

Obrázek 18 - Základní .cfg soubor bez signálů z TIA Portal

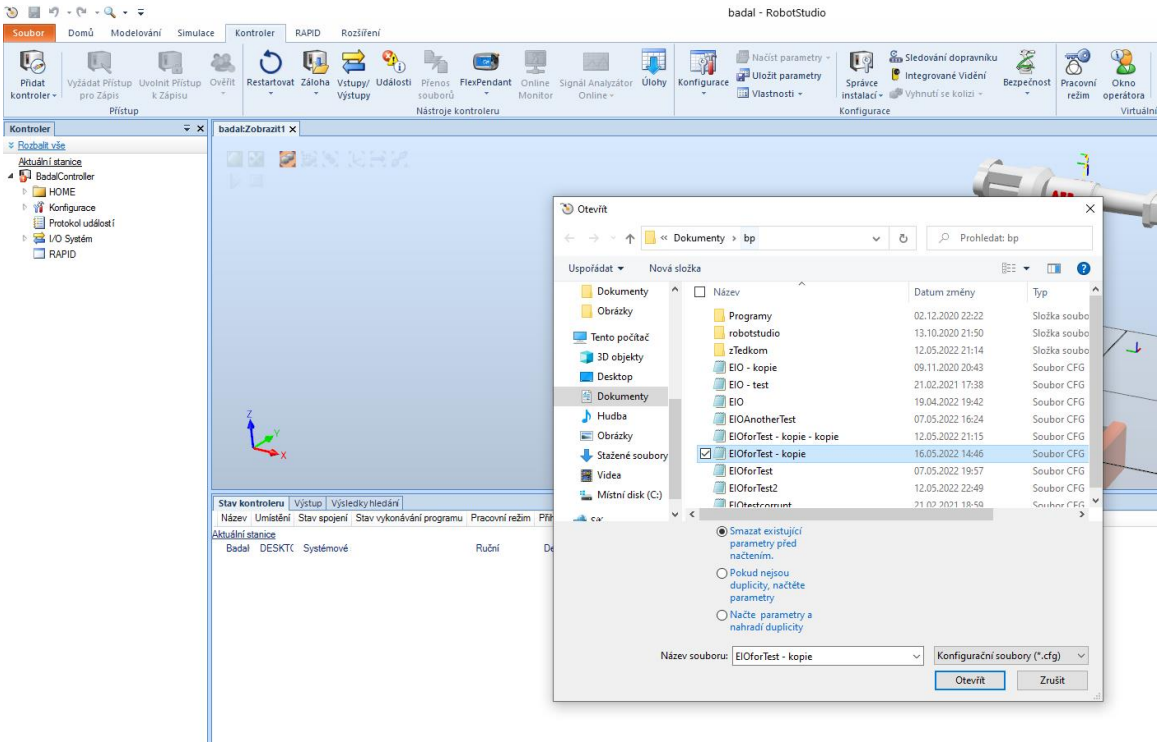
K testování byl vybrán mód výběru „Select individually“ a bylo tak vybráno několik signálů z excelovského souboru, které se mají přidat do souboru. Také byl vytvořen v programu testovací signál navíc, který se taktéž přidá do konečného vygenerovaného souboru.



Obrázek 19 - Použití konečného programu k vygenerování konfiguračního souboru

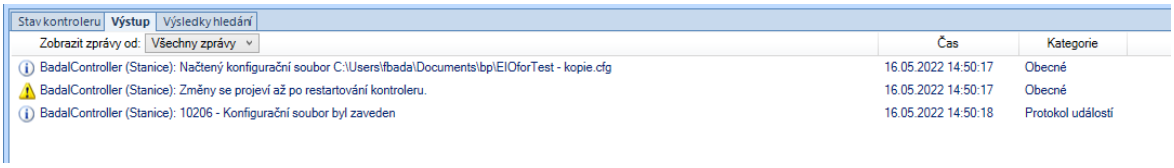
Následně se vybraný soubor .cfg přepsal již i s vybranými signály z TIA Portalu prostřednictvím excelovského souboru. Tento soubor se poté nahraje přímo do robotu pomocí programu Robot Studio.

Po spuštění a nahrání projektu v Robot Studio, je možné nahrát konfigurační soubor na záložce „Kontrolér“ a v sekci „konfigurace“ vybrat načíst parametry. Poté se v dialogovém okně zvolí vygenerovaný soubor a vybere se možnost „Smazat existující parametry před načtením“.



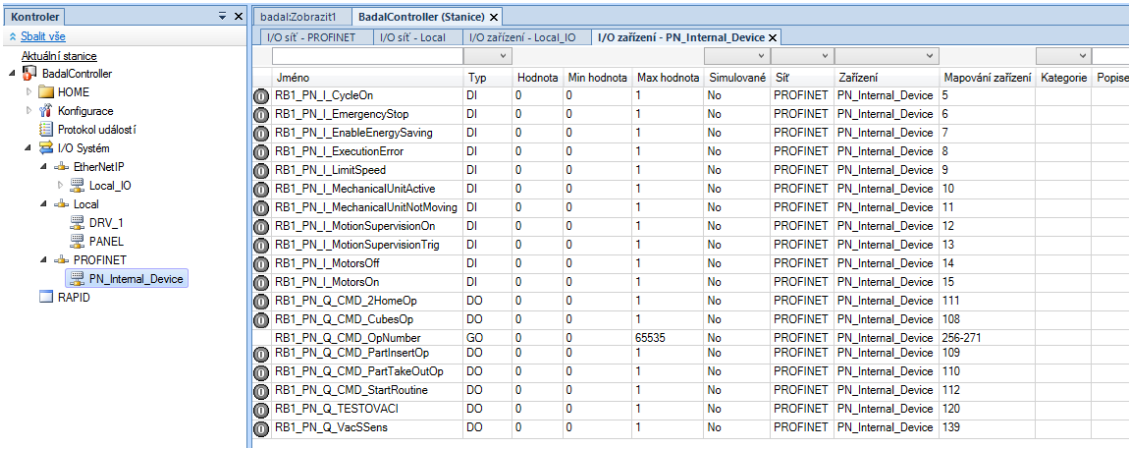
Obrázek 20 - Nahrání vygenerovaného souboru

Po potvrtení tlačítkem „Otevřít“ můžeme ve výstupu vidět, že se soubor s konfiguračními parametry zavedl.



Obrázek 21 - Log s informacemi o úspěšném zavedení souboru

Po restartu kontroléru pak také lze na následujícím obrázku vidět, že se ony signály opravdu přidali.



Jméno	Typ	Hodnota	Min hodnota	Max hodnota	Simulované	Síť	Zařízení	Mapování zařízení	Kategorie	Popise
RB1_PN_I_CycleOn	DI	0	0	1	No	PROFINET	PN_Internal_Device	5		
RB1_PN_I_EmergencyStop	DI	0	0	1	No	PROFINET	PN_Internal_Device	6		
RB1_PN_I_EnableEnergySaving	DI	0	0	1	No	PROFINET	PN_Internal_Device	7		
RB1_PN_I_ExecutionError	DI	0	0	1	No	PROFINET	PN_Internal_Device	8		
RB1_PN_I_LimitSpeed	DI	0	0	1	No	PROFINET	PN_Internal_Device	9		
RB1_PN_I_MechanicalUnitActive	DI	0	0	1	No	PROFINET	PN_Internal_Device	10		
RB1_PN_I_MechanicalUnitNotMoving	DI	0	0	1	No	PROFINET	PN_Internal_Device	11		
RB1_PN_I_MotionSupervisionOn	DI	0	0	1	No	PROFINET	PN_Internal_Device	12		
RB1_PN_I_MotionSupervisionTrig	DI	0	0	1	No	PROFINET	PN_Internal_Device	13		
RB1_PN_I_MotorsOff	DI	0	0	1	No	PROFINET	PN_Internal_Device	14		
RB1_PN_I_MotorsOn	DI	0	0	1	No	PROFINET	PN_Internal_Device	15		
RB1_PN_Q_CMD_2HomeOp	DO	0	0	1	No	PROFINET	PN_Internal_Device	111		
RB1_PN_Q_CMD_CubesOp	DO	0	0	1	No	PROFINET	PN_Internal_Device	108		
RB1_PN_Q_CMD_OpNumber	GO	0	0	65535	No	PROFINET	PN_Internal_Device	256-271		
RB1_PN_Q_CMD_PartInsertOp	DO	0	0	1	No	PROFINET	PN_Internal_Device	109		
RB1_PN_Q_CMD_PartTakeOutOp	DO	0	0	1	No	PROFINET	PN_Internal_Device	110		
RB1_PN_Q_CMD_StartRoutine	DO	0	0	1	No	PROFINET	PN_Internal_Device	112		
RB1_PN_Q_TESTOVACI	DO	0	0	1	No	PROFINET	PN_Internal_Device	120		
RB1_PN_Q_VacSSens	DO	0	0	1	No	PROFINET	PN_Internal_Device	139		

Obrázek 22 - Nové signály nahrané v robotu

4. Závěr

Hlavním cílem této práce byl návrh softwaru pro tvorbu konfiguračních souborů do ABB robotu, a to konkrétně tak, že přidá vyexportované signály z programu TIA Portal v excelovském formátu (.xlsx) do již existujícího konfiguračního souboru typu .cfg, a to tak, že převede tyto signály do formátu EIO Signálů, které jsou využity v robotech ABB. Tyto signály poté slouží ke komunikaci s robotem pomocí sběrnice Profinet.

K tomu byl vybrán programovací jazyk C Sharp v kombinaci s vývojovým prostředím Visual Studio. Program je psán objektově orientovaným přístupem, díky čemuž by mělo být další rozšiřování aplikace jednodušší a obecně kód i lépe čitelnější a mít lepší udržitelnost.

Program procházel vývojem, a to od jednoduchého přepsání signálů, tedy hlavní funkce, až do výběru několika módů zápisu, a nakonec i přidání vlastního vytvořeného signálu a jeho následnou kontrolu kvůli duplicitám v adresaci v paměti zařízení.

Následně byl tento program otestován a s ním taktéž vygenerovaný soubor, který se nahrál do simulované linky prostřednictvím programu Robot Studio.

5. Zdroje

[1] CO JE PLC NEBOLI PROGRAMOVATELNÝ LOGICKÝ AUTOMAT. *Dreamland PLC* [online]. [cit. 2021-8-1]. Dostupné z: https://dreamland-plc.cz/plc-programovatelny-logicky-automat/

[2] History of the PLC. *Library at AutomationDirect* [online]. [cit. 2021-8-1]. Dostupné z: https://library.automationdirect.com/history-of-the-plc/

[3] What is the definition of "PLC"? *Unitronics* [online]. [cit. 2021-8-1]. Dostupné z: https://www.unitronicsplc.com/what-is-plc-programmable-logic-controller/

[4] Průmyslové komunikační sítě. *Unitronics* [online]. 2013 [cit. 2021-8-1]. Dostupné z: https://www.elektroprumysl.cz/automatizace/prumyslove-komunikacni-site

[5] DRAHOŠ, Peter a Juraj GABRIEL. Komunikačný systém Profinet IO. *Automa* [online]. 2006 [cit. 2021-8-1]. Dostupné z: https://www.elektroprumysl.cz/automatizace/prumyslove-komunikacni-site

[6] VOJÁČEK, Antonín. Průmyslová sběrnice DeviceNet. *Automatizace.hw.cz* [online]. 2014 [cit. 2021-8-1]. Dostupné z: https://automatizace.hw.cz/prumyslove-sbernice-a-komunikace/prumyslova-sbernice-devicenet.html

[7] FANTON, Darek. Why All The Fuss About CAN Bus? *Onlogic blog* [online]. 2020 [cit. 2021-8-1]. Dostupné z: https://www.onlogic.com/company/io-hub/fuss-can-bus/

[8] BIGELOW, Stephen J. Why All The Fuss About CAN Bus? *SearchITOperations* [online]. 2018 [cit. 2021-8-1]. Dostupné z: https://searchitoperations.techtarget.com/definition/configuration-file

[9] KIDDER, T.J. *A Brief History of Programming Languages: Part I* [online]. 2019 [cit. 2021-8-2]. Dostupné z: https://www.learnacademy.org/blog/first-programming-language-use-microsoft-apple/

[10] KIDDER, T.J. What is a programming language? *Java T Point* [online]. [cit. 2021-8-2]. Dostupné z: https://www.javatpoint.com/classification-of-programming-languages

[11] High-level programming language. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2005, 2005 [cit. 2021-8-2]. Dostupné z: https://en.wikipedia.org/wiki/High-level_programming_language

[12] Interpreted vs Compiled Programming Languages: What's the Difference? *FreeCodeCamp* [online]. 2020 [cit. 2021-8-2]. Dostupné z: https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/

[13] WAGNER, Bill, David COULTER a Ben NEWCOMB. A tour of the C# language. *Microsoft Docs* [online]. 2021 [cit. 2021-8-2]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

[14] PROFINET / PROFIBUS - Converter. *ADFweb* [online]. [cit. 2021-8-2]. Dostupné z: https://www.adfweb.com/home/products/PROFINET_PROFIBUS.asp?frompg=nav12_12

[15] WeChat applet: JSON configuration file. *ProgrammerSought* [online]. [cit. 2021-8-2]. Dostupné z: https://www.programmersought.com/article/83635059207/

[16] SIEMERS, Christian. *Reconfigurable Microprocessor and Microcontroller–Architectures and Classification* [online]. 2004, , 2 [cit. 2021-8-2]. Dostupné z: https://www.researchgate.net/figure/Translation-of-the-source-code-from-figure-1-into-assembler-code_fig4_228941590

[17] *A Python programming environment* [online]. [cit. 2021-8-2]. Dostupné z: https://vt.instructure.com/courses/27918/pages/book-5-dot-2-a-python-programming-environment

[18] *Technical reference manual: System parameters* [online]. 19. 11. 2021 [cit. 2022-04-01]. Dostupné z: https://abb.sluzba.cz/Pages/Public/OmniCoreRoboticsDocumentationRW7/Controllers/RobotWare/System%20parameters/en/3HAC065041-001.pdf

[19] ExcelDataReader. *Github.com* [online]. 2. 5. 2019 [cit. 2022-05-01]. Dostupné z: https://github.com/ExcelDataReader/ExcelDataReader

[20] Použití vlastností: Průvodce programováním v C#. In: *Microsoft Docs* [online]. 6. 4. 2022 [cit. 2022-05-01]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/csharp/programming-guide/classes-and-structs/using-properties

[21] C# - Interface. In: *Tutorials Teacher* [online]. [cit. 2022-05-01]. Dostupné z: https://www.tutorialsteacher.com/csharp/csharp-interface

[22] Delegáti: Průvodce programováním v C#. *Microsoft Docs* [online]. 10. 5. 2022 [cit. 2022-05-11]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/csharp/programming-guide/delegates/

[23] Co je středně pokročilý jazyk?. *Netinbag.com* [online]. [cit. 2022-05-16]. Dostupné z: https://www.netinbag.com/cs/internet/what-is-an-intermediate-language.html

6. Seznam obrázků

7. Seznam zkratek

PLC	Programovatelný logický automat
HMI	Human machine interface
CAN	Controller Area Network
TIA Portal	Totally Integrated Automation Portal
OOP	Objektově orientované programování

8. Seznam příloh

P.1-ABBConfigVisualStuido – Kompletní řešení spustitelné jako projekt ve vývojovém prostředí Visual Studio

P.2-ABBConfig – Samostatný spustitelný vyexportovaný program

P.3-EIO – konfigurační soubor bez Profinet signálů k testování programu a k případnému nahrání do robota.

P.4-PLCROBOT – vyexportovaný excelovský soubor se signály z programu TIA Portal