Bakalářská práce

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra řídící techniky

# Nadstavba funkcionálního testování I/O PLC Siemens modulů. (Black-Box / Functional Approach)

Richard Rubáš

Vedoucí: Ing. Josef Kopečný
Obor: Kybernetika a robotika
Květen 2024

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Rubáš** Jméno: **Richard** Osobní číslo: **507799**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra řídicí techniky**

Studijní program: **Kybernetika a robotika**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Nadstavba funkcionálního testování I/O PLC siemens modulů. (Black-Box / Functional Approach)**

Název bakalářské práce anglicky:

**Functional testing of I/O PLC siemens modules extension. (Black-Box / Functional Approach)**

Pokyny pro vypracování:

V současnosti se k testování modulů používá softwarová emulace, která se zatím využívá převážně pro unit testování. Cílem bakalářské práce je tuto emulaci rozšířit, aby byla vhodná i pro funkcionální testování. V tomto prostředí je emulované F-CPU s Bus Interface a hardwarová vrstva modulu. Nad touto vrstvou je vrstva s Frameworkem a nad ní je již reálný module specific kód. Díky tomu je možné navodit a otestovat všechny stavy modulu. Tyto unit testy jsou psány v jazyce C++ za použití interních knihoven. Nevýhodou tohoto řešení je, že pro explorativní testování je toto prostředí příliš pracné na úpravu a pomalé. Proto by bylo vhodné mít nadstavbu nad tímto systémem a umožnit tak testovat/ladit modul. Výhodou tohoto řešení nebude nutnost neustále kompilovat kód pro ověření každé změny v prostředí, ale tato změna se projeví okamžitě.
Cílem práce je tedy vymyslet univerzální způsob, jak:
- Jednoduše naparametrizovat testovaný modul (xml/gui ...).
- Umožnit přechod mezi jednotlivými stavy modulu (F-CPU je ve start/stop).
- Dynamicky měnit prostředí tak aby bylo možné otestovat požadované chovaní modulu. (Short to P, Short to M, Wirebreak)
- Získávání diagnostiky z modulu (IO data, Alarmy, volitelně Datasets)
- Zasílání IO dat do modulu.

Volitelně:
- Vizualizovat chování modulu (IOData, Diagnostika)

Seznam doporučené literatury:

[1] A. Shvets, Dive Into Design Patterns, 2018 by Refactoring.Guru,
[2] N. M. Josuttis, The C++ Standard Library - A Tutorial and Reference, 2nd Edition, Addison Wesley Longman, 2012

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**Ing. Josef Kopečný Siemens, s.r.o. , Siemens Advanta Development**

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

Datum zadání bakalářské práce: **15.01.2024** Termín odevzdání bakalářské práce: **24.05.2024**

Platnost zadání bakalářské práce:
**do konce letního semestru 2024/2025**

| Ing. Josef Kopečný | prof. Ing. Michael Šebek, DrSc. | prof. Mgr. Petr Páta, Ph.D. |
|---|---|---|
| podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | podpis děkana(ky) |

 © VUT v Praze, Design: VUT v Praze, VIC

## III. PŘEVZETÍ ZADÁNÍ

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

.

Datum převzetí zadání

Podpis studenta


© VUT v Praze, Design: VUT v Praze, VIC

## Poděkování

Tímto bych chtěl poděkovat vedoucímu této práce Ing. Josefovi Kopečnému za vstřícný přístup při dotazech ohledně praktické i teoretické části. Doktoru Martinu Hlinovskému, za zprostředkování této práce v rámci fakulty.

## Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 24. května 2024

Podpis ................................

# Abstrakt

Tato práce se věnuje funkcionálnímu testování I/O modulů PLC Siemens. Jsou rozebrány základní metody testování softwaru a popsány *Fail-safe* I/O moduly, na které se práce zaměřuje. Dále je představena emulace hardwaru těchto modulů. Zbytek práce se zabývá návrhem aplikace, která umožňuje komunikaci s touto emulací a pohodlné testování PLC I/O modulů prostřednictvím grafického uživatelského rozhraní. Aplikace je napsána v jazycích C/C++ s využitím knihovny wxWidgets.



**Klíčová slova:** PLC, C/C++, Softwarové testování, Fail-safe


**Vedoucí:** Ing. Josef Kopečný
Siemens, s.r.o,
Siemensova 1,
Praha 13


# Abstract

This thesis focuses on the functional testing of Siemens PLC I/O modules. It discusses the basic methods of software testing and describes the *Fail-safe* I/O modules, which are the main focus of this work. Additionally, it introduces the software emulation of these modules. The rest of the thesis is dedicated to the design of an application that communicates with this emulation, allowing for convenient testing of PLC I/O modules through a graphical user interface. This application is developed in C/C++ using the wxWidgets library.



**Keywords:** PLC, C/C++, Software testing, Fail-safe


**Title translation:** Functional testing of I/O PLC Siemens modules extension. (Black-Box / Functional Approach)

# Obsah

**1 Úvod** 1

1.1 SIMATIC IO 200SP fail-safe moduly . . . . 1

1.2 Jazyková úprava textu . . . . 2

**2 Rozbor zadání** 3

2.1 Reálný F-IO modul . . . . 3

2.1.1 PROFINET a PROFIsafe komunikace . . . . 4

2.2 Struktura softwaru F-IO modulů 5

2.3 Testování softwaru . . . . 6

2.3.1 Přístupy k integračnímu testování . . . . 7

2.3.2 Black box a White box testování 8

2.4 Simulační prostředí . . . . 8

2.4.1 Výhody Simulačního prostředí 10

2.4.2 Nevýhody Simulačního Prostředí . . . . 10

2.5 Stanovení cílů . . . . 10

**3 Návrh řešení** 13

3.1 Řešení . . . . 13

3.1.1 Volba programovacího jazyka 14

3.1.2 Volba knihovny pro uživatelské grafické rozhraní . . . . 14

3.1.3 Volba ostatních knihoven . . . 14

**4 Komunikace** 17

4.1 Rozbor problému . . . . 17

4.2 Požadavky na komunikační rozhraní . . . . 18

4.3 Komunikace Windows-WSL . . . . 18

4.4 Komunikace Windows-Windows 20

4.5 Implementace zpráv . . . . 21

4.6 Implementace komunikačního rozhraní . . . . 22

**5 Parametrizace modulů** 25

5.1 Popis zařízení v síti PROFINET 25

5.1.1 Parametrizace v GSDML souboru . . . . 26

5.2 Parametrizace v uživatelském testovacím rozhraní . . . . 27

5.2.1 Vyhledávání modulů v GSDML souboru . . . . 28

**6 Model prostředí pro F-IO moduly** 29

6.1 Cyklus firmware F-IO modulu . . 29

6.2 Struktura modelu prostředí pro F-IO moduly . . . . 30

6.2.1 Modelování vstupních a výstupních kanálů . . . . 30

6.2.2 Náhrada IO dat . . . . 32

6.3 Implementace jednotlivých komponent . . . . 32

6.4 Tvorba modelu prostředí F-IO modulů . . . . 34

6.4.1 Propojení modelu prostředí a Virtuálního modulu . . . . . . . . . . . . 35

6.4.2 Vytváření modelu prostředí v testovacím rozhraní . . . . . . . . . . . . 36

**7 Ovládání a vizualizace simulace 39**

7.1 Schéma simulace . . . . . . . . . . . . . . 39

7.2 Řízení simulace . . . . . . . . . . . . . . . 40

7.2.1 Průběh cyklu firmware v simulaci . . . . . . . . . . . . . . . . . . . . . . 42

7.2.2 Implementace řízení simulace 43

7.3 Vizualizace simulace . . . . . . . . . . 44

7.3.1 Nastavení uživatelských vstupů 45

**8 Závěr 47**

**9 Seznam zkratek 49**

**Literatura 51**

# Obrázky

2.1 Schéma struktury F-IO modulu. Převzato z [1]. . . . . 4

2.2 Struktura komunikačního cyklu. Převzato z [2]. . . . . 4

2.3 Struktura IO dat pro F-IO moduly. 5

2.4 Struktura softwaru F-IO modulu. 6

2.5 Integrační testování. . . . . 8

2.6 Struktura Simulačního prostředí pro F-IO moduly. . . . . 9

2.7 Schéma použití Simulačního prostředí. . . . . 10

2.8 Schéma rozšíření Simulačního prostředí. . . . . 11

4.1 Schéma komunikace mezi testovacím prostředím a Virtuálním modulem. . . . . 18

4.2 Schéma komunikace pomocí souborů mezi procesy WSL-Windows. . . . . 20

4.3 Schéma komunikace pomocí TCP mezi procesy Windows-Windows. . 21

4.4 Struktura zpráv. . . . . 22

4.5 Struktura komunikačního rozhraní. . . . . 23

5.1 Parametrizace v programu TIA Portal. . . . . 26

5.2 Schéma parametrizace v GSDML souboru. . . . . 26

5.3 Volba parametrizace v testovacím rozhraní. . . . . 27

5.4 Vyhledávání modulu v testovacím prostředí. . . . . 28

6.1 Cyklus firmware F-IO modulu. . 29

6.2 Schéma a model jednoduchého výstupního kanálu. . . . . 31

6.3 Schéma a model výstupního kanálu s paralelními spínači. . . . . 31

6.4 Schéma a model kanálu se simulovaným zkratem. . . . . 32

6.5 Kompozice komponent modelu prostředí pro F-IO modul. . . . . 33

6.6 Schéma XML souboru pro tvorbu uživatelského prostředí. . . . . 34

6.7 Zápis jednoduchého modelu prostředí v XML souboru. . . . . 35

6.8 Schéma propojení modelu prostředí a Virtuálního modulu. . . . . 36

6.9 Editor prostředí pro F-IO moduly. 37

6.10 Kompozice komponent modelu prostředí pro F-IO Modul. . . . . 37

7.1 Schéma simulace. . . . . 39

7.2 Ovládací panel. . . . . 40

7.3 Stavový automat simulace. . . . . 41

7.4 Sekvenční diagram simulace cyklu firmware. . . . . 42

7.5 Znázornění zasílání zpráv mezi Virtuálním modulem a testovacím rozhraním. . . . . . . . . . . . . . . . . . . . . . 43
7.6 Schéma vzájemné kontroly mikrokontrolérů. . . . . . . . . . . . . . . . . 43
7.7 Struktura bloku Řízení simulace. 44
7.8 Vizualizace průběhu simulace v GUI. . . . . . . . . . . . . . . . . . . . . . . . . . 44
7.9 XML schéma pro definování uživatelského vstupu. . . . . . . . . . . . . 45
7.10 Editor uživatelského vstupu. . . 46
7.11 Vizualizace komponent s uživatelským vstupem. . . . . . . . . . 46

# Tabulky

# Kapitola 1

# Úvod

Programovatelné logické automaty (PLC), jsou dnes nejrozšířenější technologií pro řízení průmyslových procesů. Typicky se PLC dělí na dvě části *Central Processing Unit* (CPU) a vstupně/výstupní (IO) periferie [3]. PLC systémy se využívají v široké škále aplikací, od řízení plně autonomních výrobních linek přes ovládání zařízení, které interagují s lidmi, až po kritické systémy jako je řízení jaderných elektráren. Různá prostředí, kladou odlišné požadavky na PLC systémy. Jedním z příkladů těchto požadavků jsou stupně integrity bezpečnosti (SIL), které se řadí do čtyř úrovní. Čím vyšší stupeň tím vyšší nároky na bezpečnost. Tato práce se zabývá testováním IO periferií z řady "SIMATIC IO 200SP Fail-safe", které mohou být využity v aplikacích vyžadujících SIL3/PL "e", nebo nižší.

## 1.1 SIMATIC IO 200SP fail-safe moduly

*Fail-safe IO* (F-IO) moduly se dále dělí na čtyři základní typy.

- Výstupní digitální modul (F-DQ).
- Vstupní digitální modul (F-DI).
- Výstupní analogový modul (F-AQ).
- Vstupní analogový modul (F-AI).

Tyto moduly se v základních funkcích příliš neliší od těch, které nejsou označeny jako "fail-safe". U vstupních modulů spočívá jejich úloha ve čtení dat z externích senzorů, zatímco výstupní moduly se zaměřují na ovládání aktuátorů. Nicméně, na rozdíl od standardních modulů, F-IO moduly provádějí širokou škálu testů a měření, díky nimž jsou schopny rozpoznat interní poruchy, jako jsou například nefunkční spínače, nebo problémy spojené s

připojenými senzory či aktuátory. V momentě, kdy je zaznamenána jakákoli závada, je klíčové, aby se modul dokázal přepnout do stavu, který minimalizuje riziko pro lidské zdraví, životní prostředí nebo majetek. Pro dosažení klasifikace SIL3/PL "e" musí být modul vyvinut a testován podle předepsaných standardů IEC61508 a ISO13849. Nakonec je kompletní dokumentace předložena externí certifikační autoritě, která po revizi a při splnění všech stanovených kritérií udělí modulu požadovanou certifikaci.

## 1.2 Jazyková úprava textu

Tato práce je psaná v českém jazyce, proto jsem se všechny obrázky a schémata snažil popisovat českými popisky. Výjimkou jsou diagramy tříd zdrojového kódu, které jsem nechal v jazyce anglickém, jelikož celý zdrojový kód je taktéž psán v jazyce anglickém. Další výjimkou jsou specifické názvy, které nemají český ekvivalent, ty jsou v obrázcích a schématech psané v uvozovkách. V textu jsou anglické výrazy psané kurzívou (*example of English text.*). Dále se v textu vyskytují názvy tříd, metod a další útržky z programovacích jazyků. Ty jsou psané v následujícím formátu `functionExample()`. Také se v textu nachází množství zkratek, jejichž úplný seznam se nachází v kapitole 9.

# Kapitola 2

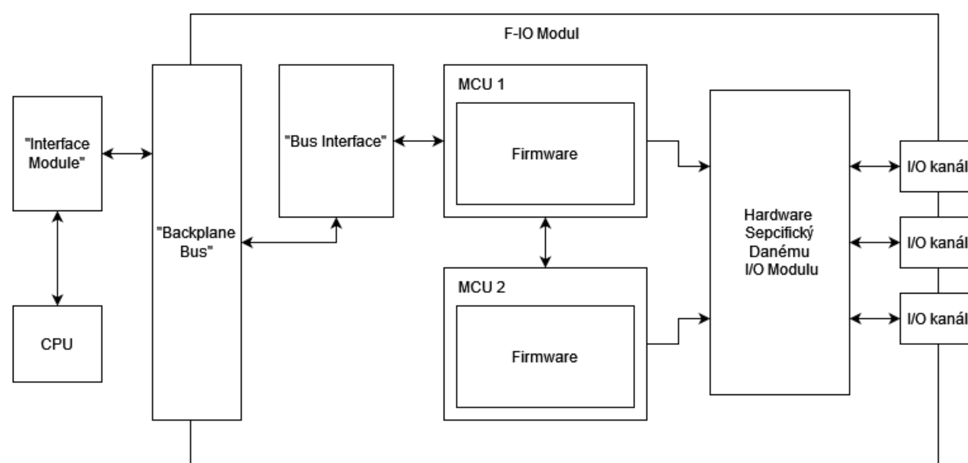
# Rozbor zadání

Tato kapitola se snaží přiblížit jak fungují F-IO moduly. Z jakých částí se skládá hardware a jak je strukturován software. Dále se věnuje základním metodám testování softwaru, představuje Simulační prostředí pro F-IO moduly a jak toto prostředí pomáhá při testování. Na závěr jsou stanoveny cíle práce, která má Simulační prostředí rozšířit a umožnit jeho nové použití.

## 2.1 Reálný F-IO modul

F-IO moduly jsou příkladem vestavěných systémů, tedy skládají se z hardwaru a softwaru. Hardware typického vestavěného systému se skládá ze dvou částí. Výpočetní jednotky, která řídí a spravuje funkce systému. Výpočetní jednotka může být mikroprocesor (MPU), mikrokontrolér (MCU) nebo hradlové pole (FPGA). Druhou částí jsou periferie systému, to jsou například senzory, aktuátory, komunikační rozhraní nebo napájení.

Hardware F-IO modulů, zobrazený na obrázku 2.1, je navrhován, aby splnil *Performance Level* ($PL_r$) kategorie 4, který udává norma ISO 13849-1. Pro dosažení $PL_r$ kategorie 4 se výpočetní jednotka musí skládá ze dvou mikrokontrolérů, které se navzájem kontrolují a v případě zaznamenání odchylky nastaví modul do bezpečného stavu. Komunikační rozhraní se skládá s bloků *Bus Interface* a *Backplane Bus*, které zprostředkovávají komunikaci s *Interface Module*. *Interface Module* následně komunikuje s CPU pomocí PROFINET a PROFIsafe technologie. Poslední částí je hardware specifický modulu. F-DI a F-DQ moduly se budou lišit pouze v této části.

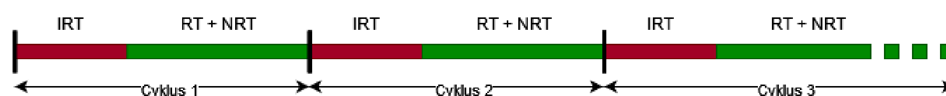



**Obrázek 2.1:** Schéma struktury F-IO modulu. Převzato z [1].

### 2.1.1 PROFINET a PROFIsafe komunikace

PROFINET je komunikační systém standardizovaný podle standardu IEC 61158, založený na průmyslovém Ethernetu. PROFINET využívá *switche* (přepínače) jako prvky síťové infrastruktury a přiřazuje prioritu jednotlivým zprávám. Díky tomuto přístupu jsou odstraněny kolize a je dosažena téměř deterministická komunikace.

Zprávy jsou řazeny do tří kategorií *Isochronous Real Time* (IRT), *Real Time* (RT) a *Non Real Time* (NRT). Komunikace probíhá v cyklech, kde každý cyklus je rozdělen na dvě části. Část pro IRT komunikaci, označovaná jako "červený interval" a část pro RT a NRT komunikaci, označovaná jako "zelený interval" [2][ss. 31-32]. Zobrazeno na obrázku 2.2.

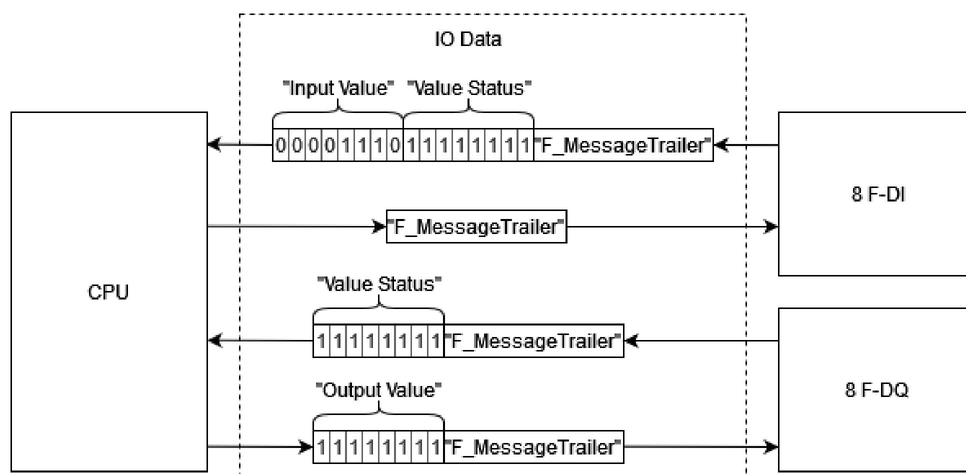



**Obrázek 2.2:** Struktura komunikačního cyklu. Převzato z [2].

Nejvyšší prioritu mají IRT zprávy. Komunikace těchto zpráv je dopředu naplánovaná, čímž je zaručeno doručení s přesností na mikrosekundy. Využívají se zejména pro řízení pohyblivých částí. Tyto zprávy nejsou podporovány F-IO moduly z řady ET 200SP.

Druhou nejvyšší prioritu mají RT zprávy. Tyto zprávy F-IO moduly využívají k přenosu tzv "IO dat" a "alarmů". Alarmy předávají informaci o chybách.

IO data se využívají pro přenos informací ohledně stavu jednotlivých kanálů, jak je zobrazeno na obrázku 2.3. V případě F-DQ modulů, CPU posílá do modulů jeden byte dat (*Output value*), kde každý bit udává jestli má být daný kanál sepnut. Dále F-DQ i F-DI moduly posílají do CPU tzv *Value status*, který reportuje jestli jsou jednotlivé kanály v pořádku. *Value status* "1" říká, že je kanál v pořádku. Naopak "0" říká, že je na kanále detekována chyba. F-DI moduly navíc posílají tzv *Input Value*, který reportuje stav jednotlivých kanálů, ve smyslu detekce logické "1" nebo "0" [4][s. 77]. Na konci každá zpráva IO dat obsahuje tzv *F_MessageTrailler*, což je redundantní informace zajišťující integritu posílaných zpráv. Tato informace je součástí PROFIsafe.




**Obrázek 2.3:** Struktura IO dat pro F-IO moduly.

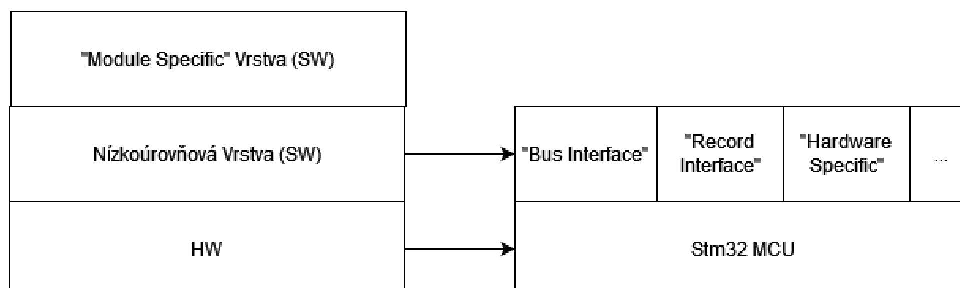
Nejnižší prioritu mají NRT zprávy, které se využívají pro vyčítání diagnostiky nebo pro posílání parametrizace do modulu.

PROFIsafe je softwarová nadstavba PROFINET, která přidává bezpečnostní funkcionality. Zajišťuje komunikaci, která splňuje standardy SIL3/PL "e". Je to nepovinná nadstavba, tedy ne všechny zařízení v PROFINET síti jí umožňují. Ale lze nakonfigurovat PROFINET sít, kde část zařízení bude využívat PROFINET s nadstavbou PROFIsafe a zbytek bude používat čistě jen PROFINET [5]. F-IO moduly umožňují PROFIsafe komunikaci.

## 2.2 Struktura softwaru F-IO modulů

Software F-IO modulů lze rozdělit do dvou vrstev, jak je vidět na obrázku 2.4. První vrstvou je nízkoúrovňová vrstva, která je umístěna přímo nad hardwarem. Tato vrstva je univerzální pro všechny F-IO moduly z řady ET 200SP a zajišťuje zápis a čtení registrů, konfiguraci periferií mikrokontroléru

a poskytuje abstrakci pro vyšší vrstvu. Díky tomu vyšší vrstva je nezávislá na hardwaru, na kterém je spuštěna. Vyšší vrstva, nazývaná *Module Specific* (specifická pro modul), je pro každý modul implementovaná odlišně a udává specifické funkcionality jednotlivých modulů.




**Obrázek 2.4:** Struktura softwaru F-IO modulu.

Nízkoúrovňová vrstva se skládá z mnoha částí, přičemž každá se zaměřuje na specifický segment hardwaru. Pro tuto práci jsou důležité pouze tři části.

- *Bus Interface* - Zajišťuje komunikaci s CPU.
- *Record Interface* - Stará se o obsluhu Analogově-digitálních převodníků (ADC).
- *Hardware Specific* - Řídí vyčítání vstupních pinů a ovládání výstupních pinů mikrokontroléru.

Jak tyto části spolu souvisí, lze vysvětlit na příkladu spínání výstupu DQ modulu. Proces spínání začíná v CPU, které odesílá zprávu o sepnutí (IO data) směrem k modulu. Zprávu přijme *Bus Interface*, který jí následně předá *Module Specific*. *Module Specific* pak instruuje část *Hardware Specific* k aktivaci spínačů. Tento proces kontroluje *Record Interface*, který pomocí ADC zaznamenává napětí za spínači a tak kontroluje jestli opravdu došlo k sepnutí. Po zaznamenání hodnot *Modul Specific* tyto informace zpracuje a prostřednictvím *Bus Interface* odesílá informace o stavu kanálu k CPU.

## 2.3 Testování softwaru

Testování softwaru je rozsáhlé a široké téma, a proto se zaměřím specificky na testování vestavěných systémů, do kterých spadají i F-IO moduly. V diplomové práci [1], která se věnuje testování F-IO modulů, je testování vestavěných systémů rozděleno do tří hlavních částí:

***Unit Tests* (Testy jednotek)**
Tyto testy ověřují základní funkčnost jednotlivých komponent programu.

Jejich cílem je zaručit, že každá část programu funguje podle specifikací.

**Softwarové integrační testy**
Tato fáze testování se zaměřuje na ověření správné spolupráce a komunikace mezi softwarovými komponentami. Hlavním cílem je zajistit integritu celého softwarového systému, aby fungoval správně jako sjednocený celek.

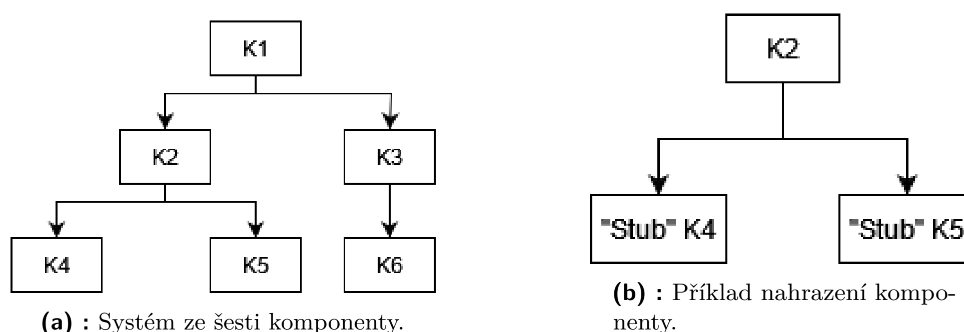
**Hardware/softwarové integrační testy**
Tyto testy jsou zásadní pro vestavěné systémy. Na rozdíl od softwarových integračních testů, které se obvykle provádějí na alternativním hardwaru (např. osobních počítačích), jsou tyto testy realizovány přímo na cílovém hardwaru. Cílem je ověřit, že software je plně funkční i na cílovém hardwaru.

### 2.3.1 Přístupy k integračnímu testování

Při integračním testování systému lze postupovat dvěma způsoby: *Bottom-up* (Zdola nahoru) nebo přístupem *Top-down* (Shora dolů). Často se využívá kombinace obou přístupů. Tyto metody vysvětlím na obrázku 2.5a, který zobrazuje systém sestávající ze šesti komponent (K1-K6).

**Zdola nahoru**
Tato metoda začíná testováním spodních komponent (K4-K6), které nejsou závislé na ostatních komponentách systému. Následně se postupuje směrem k vyšším komponentám (K2, K3 a nakonec K1), přičemž pro testování těchto vyšších komponent se používají již otestované nižší komponenty. [6].

**Shora dolů**
Při testování shora dolů se začíná u nejvyšších komponent systému (K1) a postupuje se k nižším (K2, K3 a dále K4-K6). Tento přístup může narazit na problém, kdy nižší komponenty v době testování neexistují nebo nejsou k dispozici pro testování. V takovém případě se pro tyto komponenty vytvoří tzv *stub*, což je jednoduchá implementace komponent, která obvykle disponuje jen základním rozhraním pro komunikaci s ostatními částmi systému [6], [1].




**(a) :** Systém ze šesti komponenty.

**(b) :** Příklad nahrazení komponenty.

**Obrázek 2.5:** Integrační testování.

### 2.3.2 Black box a White box testování

Další klasifikace testů rozlišuje metody *White box* (Bílá skříňka) a *Black box* (Černá skříňka).

**Bílá skříňka**

Je způsob testování, při kterém je známá vnitřní struktura nebo zdrojový kód testovaného systému. Testuje se při něm správný tok dat v systému. Pokrytí instrukcí, kolik instrukcí kódu bylo při testování zavoláno. Nebo pokrytí rozhodování, kolik se prošlo různých větví programu [7]. U F-IO modulů je požadováno 100% pokrytí rozhodovacích větví, což znamená, že každá řádka kódu musí být během testování zavolána alespoň jednou. Tímto typem testování se tato práce nezaobírá.

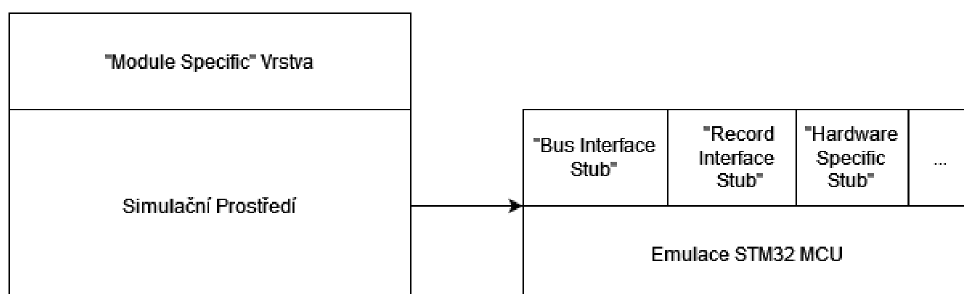
**Černá skříňka**

Někdy označované jako funkcionální testování, je způsob testování, při kterém není známá vnitřní struktura systému. Tedy pracuje se pouze s uživatelským rozhraním, zadají se vstupy a zkontrolují se výstupy, zatímco interní fungování systému zůstává skryto [7]. Pro F-IO moduly existuje dokument *Safety Requirement Specification* (SRS), který specifikuje jak se má daný modul chovat. Testování černé skříňky pro F-IO moduly ověřuje jestli se modul chová podle SRS.

## 2.4 Simulační prostředí

V rámci diplomové práce [1] bylo vyvinuto Simulační prostředí určené pro F-IO moduly. Toto prostředí poskytuje možnost softwarového integračního testování části softwaru *Module Specific*, která není závislá na fyzickém hardwaru. Jak je zobrazeno na obrázku 2.6, prostředí se dělí na dvě hlavní komponenty. První část představuje softwarovou emulaci mikrokontroléru STM32. Druhou část tvoří upravené komponenty nižších vrstev softwaru, které v produkčním

prostředí interagují s reálným hardwarem.

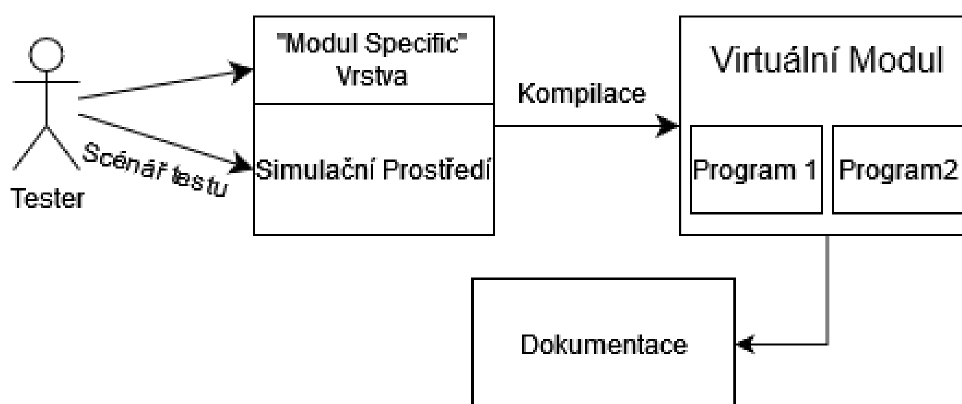



**Obrázek 2.6:** Struktura Simulačního prostředí pro F-IO moduly.

Funkce Simulačního prostředí je ilustrována na obrázku 2.7. Typický proces vytváření testu v tomto prostředí zahrnuje několik kroků. Nejprve je nutné propojit *Module Specific* část se Simulačním prostředím. Tento krok vyžaduje značné úsilí, naštěstí je potřeba jej provést jen jednou pro každý modul. Po propojení těchto částí, může tester začít s tvorbou a psaním testovacích scénářů.

Jednotlivé testy mohou simulovat různé situace, jako jsou chyby v komunikaci mezi CPU a modulem, reakce na zkrat, nebo chování při nízkém napětí. Návrh testovacího scénáře je relativně jednoduché, ale sepsaní a zprovoznění testu je podstatně náročnější. Napsání testů vyžaduje hluboké porozumění Simulačnímu prostředí, včetně specifikace parametrů modulu, definice IO data, nastavení registrů ADC a vstupních pinů a reakce na změny výstupních pinů. Výsledkem je, že vytvoření jednoho testu může vyžadovat i 1000 řádků kódu.

Po dokončení psaní testu následuje kompilace, která obvykle trvá 10-15 minut. Tento proces vygeneruje tzv Virtuální modul, který se skládá ze dvou programů, z nichž každý reprezentuje jeden mikrokontrolér. Spuštěním těchto programů se zahájí test, jehož výsledky jsou následně uloženy do dokumentace.




**Obrázek 2.7:** Schéma použití Simulačního prostředí.

### 2.4.1 Výhody Simulačního prostředí

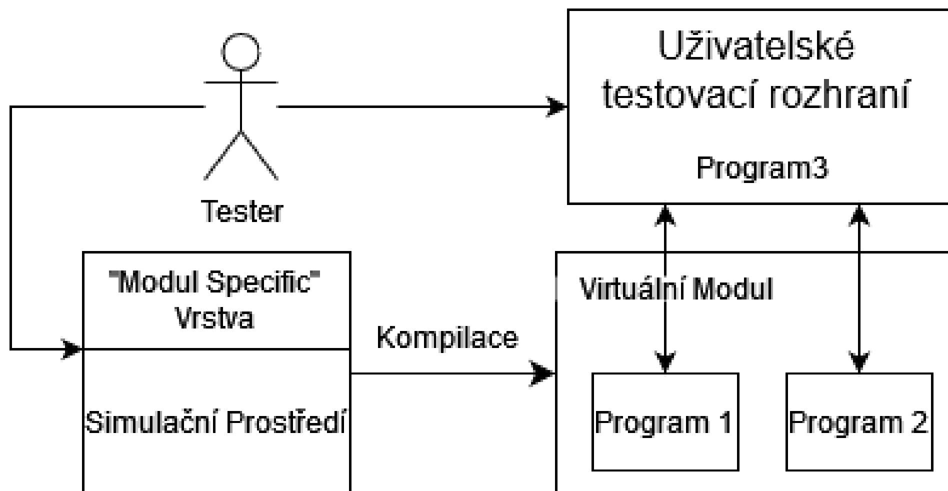
1. Není závislé na reálném hardwaru, který v době testování nemusí existovat. Testy jsou spustitelné na osobním počítači.
2. Možnost otestovat i takové scénáře, které by se špatně navozovaly s reálným hardwarem.

### 2.4.2 Nevýhody Simulačního Prostředí

1. Pracné a časově náročné vytváření testů.
2. Nutnost znalosti Simulačního prostředí.
3. Nevhodné pro explorativní testování. Například tester vytvoří test. Po vyhodnocení výsledků testu ho napadne další test, který se od předešlého liší pouze v jednom řádku. Což je jednoduchá úprava, ale musí počkat 15 min na kompilaci.

## 2.5 Stanovení cílů

Cílem je rozšířit Simulační prostředí a tím odstranit jeho nedostatky. Rozšíření by mělo umožnit nové použití Simulačního prostředí, jak je ilustrováno na obrázku 2.8. Tester by stále musel propojit *Module Specific* část se Simulačním prostředím, ale pak by stačila pouze jedna kompilace. Výsledný zkompilovaný Virtuální modul bude následně schopen komunikovat s uživatelským rozhraním, což umožní navrhování a spouštění testovacích scénářů přímo z tohoto rozhraní.




**Obrázek 2.8:** Schéma rozšíření Simulačního prostředí.

Jednotlivé dílčí úkoly jsou následující. Vymyslet jak:

- Zasílat IO data do modulu.
- Zasílat diagnostiku z modulu (IO data, alarmy, volitelně datasety).
- Jak jednoduše parametrizovat modul.
- Jak modelovat periferie/prostředí modulu, aby bylo možno navodit různé testovací scénáře (zkrat do P, zkrat do M, přepětí, ...).
- Řídit simulaci z uživatelského rozhraní.

Volitelně:

- Pokusit se vhodně vizualizovat průběh testu.

# Kapitola 3

# Návrh řešení

V této kapitole přiblížím, jak bude výsledné řešení vypadat a z jakých částí se bude skládat. Dále rozeberu výběr programovacího jazyka a volbu nástrojů a knihoven.

## 3.1 Řešení

Abych splnil požadavky zmíněné v kapitole 2.5, rozhodl jsem se že vytvořím aplikaci, která bude sloužit jako uživatelské testovací rozhraní a bude schopna komunikovat s Virtuálním modulem. Problém je že Simulační prostředí neumožňuje komunikaci s externími procesy. Tedy mé řešení se bude skládat z rozšíření Simulačního prostředí o komunikaci s externími procesy a návrhu testovacího prostředí.

Aplikace pro uživatelské rozhraní se skládá ze čtyř hlavních komponent, každá z těchto komponent bude popsána ve vlastní kapitole.

1. Komunikační rozhraní, kapitola 4.
2. Parametrizace F-IO modulu, kapitola 5.
3. Modelu prostředí pro F-IO moduly, kapitola 6. (Jedná se o jednoduchý softwarový model, který dokáže simulovat hodnoty AD převodníků a vstupní/výstupní piny mikrokontroléru.)
4. Ovládání a vizualizace simulace, kapitola 7.

### 3.1.1 Volba programovacího jazyka

Vzhledem k tomu, že celé Simulační prostředí i produkční kód pro F-IO moduly jsou napsány v jazycích C/C++, je výběr těchto jazyků pro rozšíření Simulačního prostředí jasnou volbou.

Při výběru jazyka pro vývoj aplikace jsem zvažoval několik možností, včetně Pythonu, C#, C++ a Javy. Vzhledem k mé osobní zkušenosti a znalostem jsem z možností vyloučil C# a Javu, s nimiž nemám praktické zkušenosti. Volba nakonec padla na C++, což je jazyk, který je již používán pro vývoj F-IO modulů. Tento výběr navíc usnadňuje konzultace s mým vedoucím, který má s C++ bohaté zkušenosti.

### 3.1.2 Volba knihovny pro uživatelské grafické rozhraní

Vzhledem k požadavku na uživatelskou přívětivost testovacího rozhraní, včetně volitelného zobrazení dat, jsem se rozhodl pro implementaci grafického uživatelského rozhraní (GUI). Pro efektivní a jednoduchý vývoj GUI je vhodné využít některou z dostupných knihoven nebo frameworků. Pro jazyk C++ se jako nejrozšířenější možnosti nabízeli knihovny *Qt* [8] a *wxWidgets* [9], které obě poskytují širokou škálu nástrojů pro vytvoření GUI. Přestože jsou obě knihovny dostatečně flexibilní a mohou pokrýt potřeby mé relativně jednoduché aplikace, rozhodl jsem se pro *wxWidgets* jelikož ta má, na rozdíl od *Qt*, jednoduchou licenci *"wxWindows Library Licence"* [10], která umožňuje i komerční použití.

### 3.1.3 Volba ostatních knihoven

Za účelem jednoduchosti a spolehlivosti aplikace bylo po konzultaci s vedoucím práce rozhodnuto dávat přednost *C++ Standard Library*, kdekoliv je to možné. Tato knihovna nabízí definice základních datových typů, jako jsou dynamické pole, spojové seznamy, binární stromy, a zároveň implementuje různé algoritmy pro práci s těmito datovými strukturami včetně vyhledávání prvků a třídění. Kromě toho poskytuje podporu pro vícevláknové programování a další užitečné nástroje [11].

Při vývoji aplikace jsme však narazili na specifické problémy, které nebylo možné efektivně řešit pouze s využitím *C++ Standard Library* nebo *wxWidgets*. Jeden z problémů se týkal práce se soubory ve formátu *Extensible Markup Language* (XML). Tento problém byl vyřešen využitím knihovny *RapidXml* [12], což je jednoduchá knihovna napsaná v C++, umožňující snadnou práci s XML soubory. Další problém spočíval ve vizualizaci atributovaných grafů. Pro

tuto úlohu byla zvolena knihovna *Graphviz* [13], která umožňuje vizualizaci nejrůznějších grafů. Dále jsem využil aplikační programové rozhraní (API) *Windows Sockets 2*, které umožňuje síťovou komunikaci [14].

# Kapitola 4

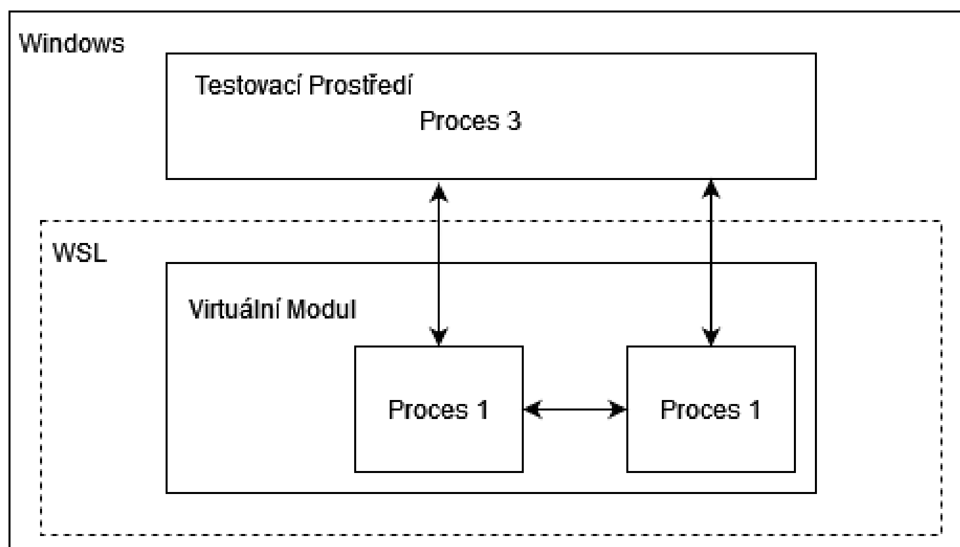
# Komunikace

V této kapitole popisuji, jakým způsobem probíhá komunikace mezi testovacím prostředím a Virtuálním modulem, jaké možnosti tato komunikace nabízí a detailně rozeberu její architekturu.

## 4.1 Rozbor problému

Testovací prostředí a Virtuální modul se skládají ze tří procesů, jak je zobrazeno na obrázku 4.1. Testovací prostředí tvoří jeden samostatný proces, který je vždy spuštěn v operačním systému Windows. Virtuální modul naopak obsahuje dva procesy, které mohou být spuštěny buď v systému Windows nebo v rámci *Windows Subsystem for Linux* (WSL). Existují dvě verze, WSL a novější WSL2, kde WSL2 má podstatně více funkcí. Virtuální modul je spouštěn ve starší verzi. WSL je funkce operačního systému Windows, umožňující spuštění linuxového prostředí na zařízeních s Windows bez potřeby virtualizace linuxového prostředí nebo konfigurace zařízení do režimu *dual boot* [15]. Volba platformy pro Virtuální modul, buď Windows nebo WSL, závisí na verzi Simulačního prostředí.

V praxi se využívají obě možnosti, proto je nezbytné navrhnout komunikaci jak mezi procesy spuštěné v systému Windows a WSL, tak mezi procesy spuštěné oba v systému Windows.




**Obrázek 4.1:** Schéma komunikace mezi testovacím prostředím a Virtuálním modulem.

## 4.2 Požadavky na komunikační rozhraní

Komunikační rozhraní musí splňovat dva základní požadavky:

- Nesmí blokovat běh hlavního programu. Tento požadavek je kritický zejména při použití v GUI, kde by bylo nežádoucí, aby se celý program zasekl při čekání na příchozí zprávu nebo při odesílání zprávy.
- Musí být schopno posílat zprávy různých délek.

## 4.3 Komunikace Windows-WSL

Jelikož Virtuální modul se skládá ze dvou procesů, komunikace mezi procesy spuštěnými ve WSL již byla implementována. Tato komunikace však má blokující charakter; například při odesílání dvou zpráv je nutné počkat s odesláním druhé zprávy, dokud druhý proces nezpracuje první zprávu. Ve Virtuálním modulu se komunikace mezi oběma procesy primárně využívá k synchronizaci, nikoli k přenosu velkých objemů dat, a proto je stávající implementace dostačující. Tato implementace využívá ke komunikaci .txt soubory. K tomu se využívá část *C++ Standard Library*, deklarovaná v hlavičkovém souboru `<fstream>`, který obsahuje základní třídy pro práci se soubory [16][ss. 791-801].

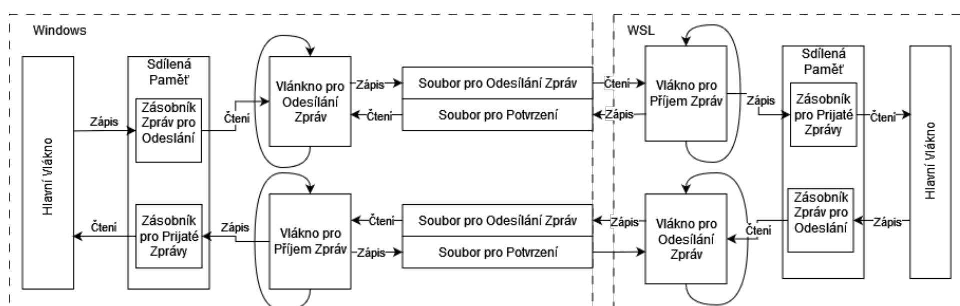
Jelikož stávající způsob komunikace není optimální, snažil jsem se najít alternativní řešení, které by umožnilo komunikaci mezi procesy ve WSL a Windows. Jediné řešení, které jsem našel a které by mělo být kompatibilní i s WSL verze 1, je použití tzv *Windows/WSL Interop with AF_UNIX*. Tato metoda umožňuje vytváření tzv "unixových soket" z procesu běžícího v systému Windows [17], [18].

V článku [18] je sice uveden ukázkový zdrojový kód spolu s diskuzí o možných problémech, které mohou nastat při vytváření unixových soket. Přesto se mi nepodařilo z procesu spuštěného ve WSL připojit k procesu spuštěnému ve Windows a tuto metodu jsem nevyužil.

Ve výsledném řešení jsem použil podobný přístup, jaký je implementován ve Virtuálním modulu, tedy s využitím tříd v hlavičkovém souboru `<fstream>`. Aby čtení i zápis zpráv neprobíhal v hlavním vláknu programu, využil jsem možnosti programování více vláknových aplikací knihovny *C++ Standard Library*. Tato knihovna (ve verzi pro C++11) umožňuje dva základní přístupy vytvoření nového vlákna programu. Pomocí funkce `async()` [16][ss. 945-946] a nebo pomocí třídy `std::thread`, což je sice složitější přístup, ale dává uživatelům více kontroly nad vytvořeným vláknem [16][ss. 979-981]. Proto jsem se rozhodl použít tuto třídu.

Při programování vícevláknových aplikací je zásadní dbát na správný přístup ke sdíleným zdrojům, aby nedošlo k situaci, kdy více vláken zároveň přistupuje ke stejné oblasti v paměti. K řízení přístupu ke sdílené paměti se využívají různé synchronizační mechanizmy, například objekty pro vzájemné vyloučení, známé jako mutexy. V mé práci jsem převážně využil dvě třídy `std::recursive_mutex` a `std::lock_guard`. Třída `std::recursive_mutex` umožňuje vícenásobné uzamknutí stejného mutexu v rámci jednoho vlákna. Tedy v rámci jednoho vlákna lze tentýž mutex zamknout vícekrát bez nutnosti čekání na jeho uvolnění. Třída `std::lock_guard` slouží k uzamknutí a následnému automatickému uvolnění mutexu. čímž eliminuje riziko, že by vlákno bylo ukončeno bez uvolnění zamknutého mutex[16][ss. 982-1000].

Výsledná implementace komunikace mezi procesy WSL-Windows je zobrazena na obrázku 4.2.




**Obrázek 4.2:** Schéma komunikace pomocí souborů mezi procesy WSL-Windows.

Hlavní myšlenka této implementace spočívá ve vytvoření nových vláken programu, která jsou zodpovědná za zápis a čtení zpráv. Tyto vlákna jsem implementoval jako smyčky, které periodicky kontrolují příchod nových zpráv, nebo přítomnost nové zprávy v zásobníku zpráv pro odeslání.

Toto řešení není optimální, především kvůli periodické kontrole, která zbytečně plýtvá výkonem procesoru. Pro odesílání zpráv by bylo možné využít mechanizmu *Condition Variables*, který umožňuje uspání jednoho vlákna a jeho následné probuzení z druhého vlákna *C++ Standard Library* [16][ss. 1003-1009]. Avšak nepodařilo se mi najít způsob monitorování změny souboru, což vedlo k závěru, že implementace *Condition Variables* by významně nezlepšila chod komunikace.

## 4.4 Komunikace Windows-Windows

Pro komunikaci mezi procesy běžícími ve Windows, jsem využil protokol *Transmission Control Protocol* (TCP), což je součást sady protokolů TCP/IP zajišťující spolehlivou a plně duplexní komunikaci. Zároveň se jedná o protokol se spojovanou komunikací, tedy před zahájením komunikace je nutné vytvořit spojení mezi serverem a klientem [19][ss. 45-56]. V mé implementaci slouží Testovací prostředí jako server a oba procesy ve Virtuálním modulu jako klienti.

Pro komunikaci mezi procesy spuštěnými na stejném zařízení, lze využít tzv *localhost* IP adresu. To je adresa 127.0.0.1 (IPv4) nebo ::1 (IPv6). Pakety poslané na tuto adresu se nedostanou do sítě, ale jsou pouze zpracovány v rámci zařízení. Také se tento druh komunikace označuje jako *loopback* [20].

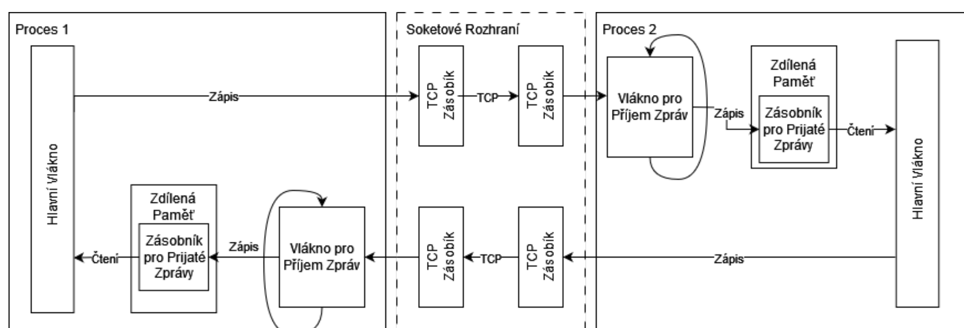
Pro jazyky C/C++ existuje *Windows Sockets 2* API. Toto API je umožňuje vývoj síťových aplikací pro operační systém Windows pomocí soketového rozhraní [14]. Já jsem využil jeho část, deklarovanou v hlavičkových souborech

`<winsock2.h>` a `<ws2tcpip.h>`, která mimo jiné umožňuje komunikaci pomocí protokolu TCP.

K odesílání zpráv jsem využil funkce `send()`, která přímo neodesílá data, ale pouze je zapíše do zásobníku pro odeslání. Tedy pokud není zásobník plný má neblokující charakter, pokud je zásobník plný funkce čeká dokud není v zásobníku místo.

K příjímání zpráv jsem využil funkce `recv()`, tato zpráva čte data ze zásobníku pro přijaté zprávy. Tedy většinou má blokující charakter, pokud je zásobník prázdný funkce čeká na naplnění. Velikosti zásobníku se dají zjistit pomocí funkce `getsockopt()` a změnit pomocí `setsockopt()`. Všechny zmíněné funkce jsou deklarované v hlavičkovém souboru `<winsock.h>` [21].

Díky převážně neblokujícímu charakteru posílání zpráv jsem implementoval komunikaci mezi procesy Windows-Windows podle schématu znázorněného na obrázku 4.3.



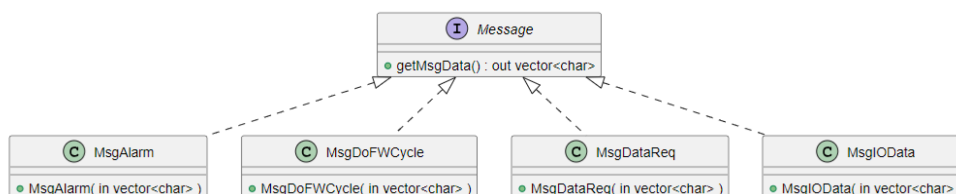

**Obrázek 4.3:** Schéma komunikace pomocí TCP mezi procesy Windows-Windows.

V tomto řešení žádné vlákno nic periodicky nekontroluje a neplýtvá tak výkonem procesoru. Vlákna pro příjem zpráv jsou většinu času neaktivní a jen čekají na přijaté data, následně data zpracují uloží do sdílené paměti. Vlákno dedikované zápisu není potřeba, jelikož operace poslání dat většinou probíhá jako zápis do zásobníku bez většího zdržení.

## 4.5 Implementace zpráv

Komunikační rozhraní musí být schopno posílat více typů zpráv. Momentálně existuje přes 10 typů zpráv. Z tohoto důvodu jsem se rozhodl implementovat univerzální rozhraní `Message`, které deklaruje jedinou metodu `getMsgData() : out vector<char>`. Tato metoda serializuje zprávu do posloupnosti bajtů, což zaručuje jednotné odesílání všech typů zpráv a usnadňuje

rozšíření systému o nové typy zpráv. Stačí nový typ zprávy implementovat podle rozhraní `Message`, a bude možné jej odesílat. Kromě toho musí všechny třídy, implementující rozhraní `Message`, obsahovat konstruktor, který umožňuje rekonstrukci zprávy z posloupnosti bajtů. Schéma zpráv je zobrazeno na obrázku 4.4.



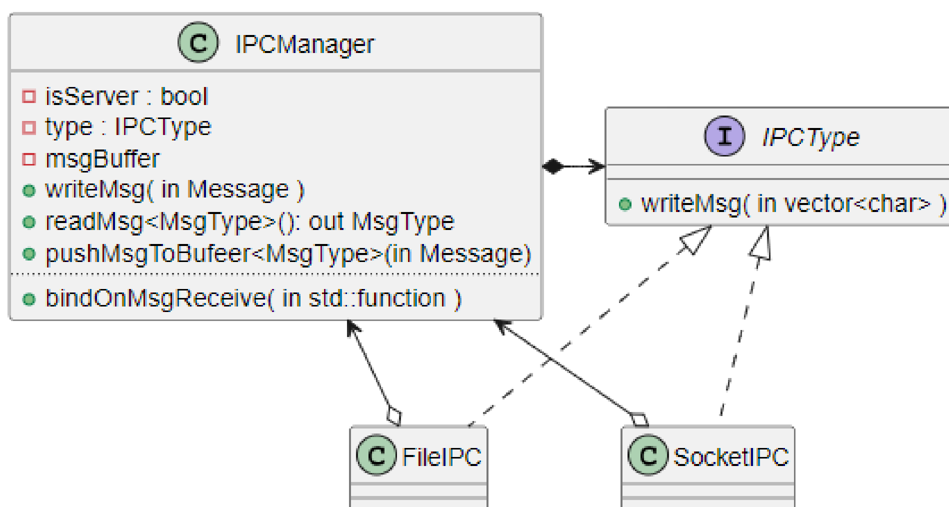

**Obrázek 4.4:** Struktura zpráv.

## 4.6 Implementace komunikačního rozhraní

Jelikož existují dva způsoby komunikace, tak jsem vytvořil třídu `IPCManager`, která je schopna obou způsobů komunikace. Zároveň se navenek tváří stejně pro oba typy komunikace. Tedy klienti využívající tuto třídu neznají způsob, jakým je zpráva posílaná. Toho je docíleno strukturou zobrazenou na obrázku 4.5. Kde rozhraní `IPCType`, deklaruje metodu pro posílání zpráv. Následně jeho realizace `FileIPC` pro komunikaci pomocí souborů (sekce 4.3) a `SocketIPC` pro komunikaci pomocí soketů (sekce 4.4), implementují tuto metodu různými způsoby. Zároveň tyto třídy různě implementují čtení zpráv, ale doručené zprávy ukládají stejným způsobem pomocí funkce `pushMsgToBuffer<MsgType>( in vector<char> )`.

Tato implementace umožňuje snadné rozšíření o nové typy komunikace. Především, kdyby se v budoucnu podařilo vymyslet komunikaci WSL-Windows smysluplněji, tak nebude problém s implementací do stávajícího řešení.

Další důležitou funkcionalitu přináší metoda `bindOnMsgReceive(in std::function)`. Díky které si klient může zvolit, co se má stát při obdržení zprávy, a nemusí se cyklicky kontrolovat, zdali nepřišla zpráva.




**Obrázek 4.5:** Struktura komunikačního rozhraní.

# Kapitola 5

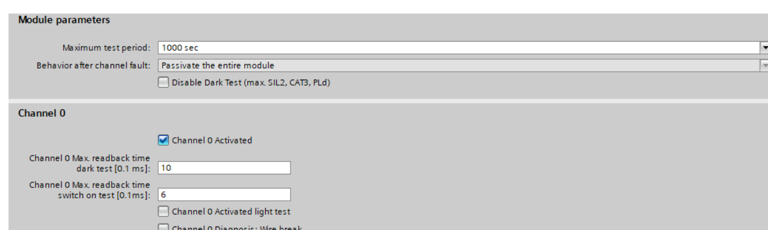
# Parametrizace modulů

Parametrizace modulu představuje zásadní součást testovacího procesu, jelikož je nutné ji definovat v rámci každého testu. Dokonce spousta testů se liší pouze v parametrizaci. To znamená, že jednoduché a pohodlné zadání parametrizace ušetří spoustu času při testování.

## 5.1 Popis zařízení v síti PROFINET

Zařízení, která jsou schopna komunikovat pomocí technologií PROFINET nebo PROFIBUS, využívají k popisu jejich specifikací soubory *General Station Description* (GSD). Tyto soubory obsahují identifikační číslo zařízení, jeho komunikační možnosti, diagnostiku a také možnosti parametrizace. Existují dva druhy GSD souborů. Pro zařízení v PROFIBUS síti existuje formát *GSD Language*, což ASCII soubor popisující zařízení pomocí klíčových slov. Pro zařízení v PROFINET existuje formát *GSD Markup Language* (GSDML), který je založený na formátu XML [22], [23].

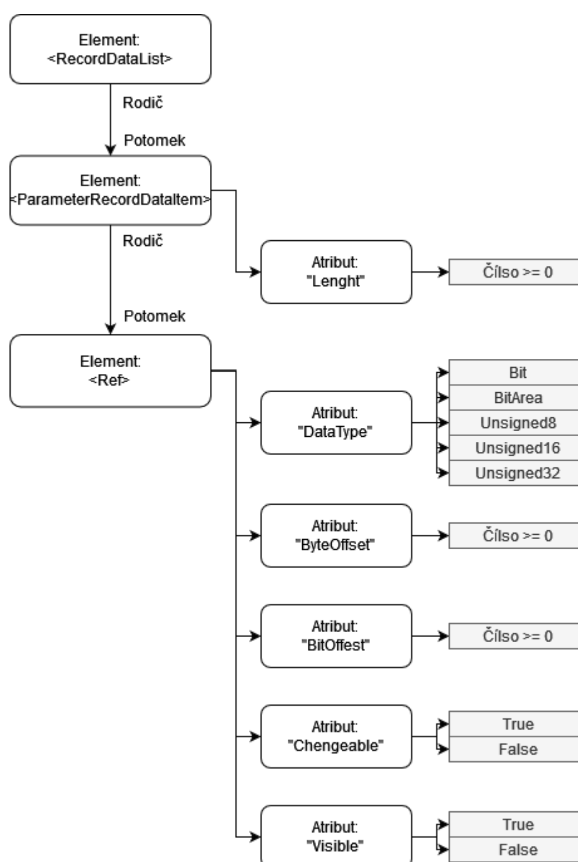
Konfigurační nástroje využívají tyto soubory, aby umožnily uživatelům konfigurovat sítě PROFIBUS a PROFINET. Příkladem takového nástroje je program *Totally Integrated Automation Portal* (TIA Portal), který se používá k programování Siemens PLC systémů. Tento program mimo jiné umožňuje parametrizaci F-IO modulů prostřednictvím jednoduchého grafického rozhraní, zobrazeného na obrázku 5.1.




**Obrázek 5.1:** Parametrizace v programu TIA Portal.

### 5.1.1 Parametrizace v GSDML souboru

F-IO moduly využívají ke svému popisu soubory ve formátu GSDML. Parametrizace modulů je v GSDML popsána v elementech `<RecordDataList>`, jejichž neúplná struktura je zobrazena na obrázku 5.2. Podrobnější popis je dostupný na odkazu [24].



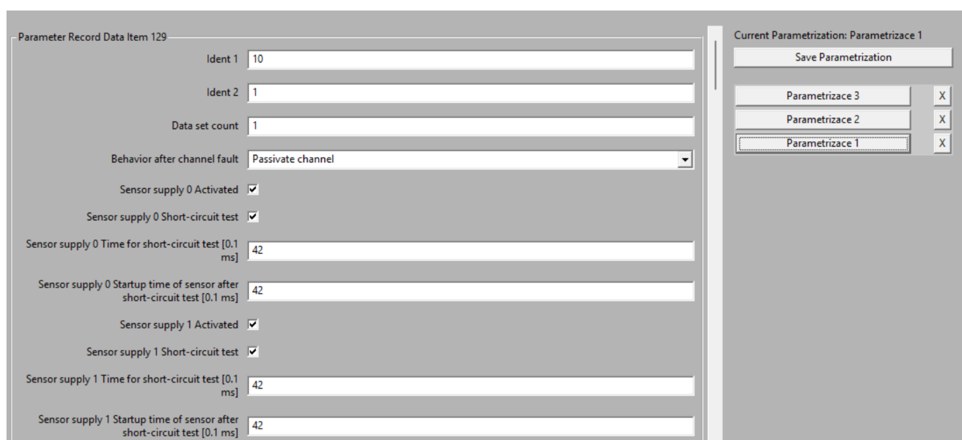

**Obrázek 5.2:** Schéma parametrizace v GSDML souboru.

`<RecordDataList>` obsahuje různý počet elementu `<ParameterRecordDataItem>`. Tyto elementy pak reprezentují pakety dat, které posílá CPU do IO periferií jako parametrizaci. Délku paketu v bytech udává atribut `"Length"`. Význam jednotlivým bitům v paketu udávají elementy `<Ref>`. Každý tento element představuje jeden parametr modulu. Jeho pozici a délku v paketu představují atributy `"DataType"`, `"ByteOffset"` a `"BitOffset"`. Dalšími vlastností parametru je jeho viditelnost a možnost změny v konfiguračních nástrojích. To určují atributy `"Chengeable"` a `"Visible"`.

## 5.2 Parametrizace v uživatelském testovacím rozhraní

Jelikož součástí zadání této bakalářské práce (sekce 2.5) je umožnit parametrizaci modulu, vytvořil jsem rozhraní vizuálně podobné tomu v programu TIA Portal. Mnou vytvořené rozhraní, zobrazené na obrázku 5.3, navíc umožňuje zobrazení a úpravu parametrů, které jsou označeny jako neměnitelné nebo neviditelné. Toto rozšíření umožňuje experimentování i s neplatnými parametrizacemi a testování reakcí modulu na takové nastavení.

Další funkcionalitou testovacího rozhraní je možnost ukládání parametrizací. Uživatelé tak mohou předem připravit různé parametrizace a následně jednoduše spouštět testy, které se liší pouze v parametrizaci.




**Obrázek 5.3:** Volba parametrizace v testovacím rozhraní.

### 5.2.1 Vyhledávání modulů v GSDML souboru

Pro všechny moduly z řady SIMATIC ET 200SP existuje jeden GSDML soubor, který obsahuje informace o všech modulech z této řady. Přibližně se jedná o 500 modulů. Jeden modul může být započten víckrát, jelikož každá verze firmware se počítá samostatně. Abych zjednodušil vyhledávání modulu implementoval jsem jednoduché vyhledávání, zobrazené na obrázku 5.4, které umožňuje hledání testovaného modulu podle názvu nebo čísla modulu.

F-DI 8

Module ID: F-DI 8x24VDC HF, BP
Module Num: 0x01004D40 0x00000108

Module ID: F-DI 8x24VDC HF2, BP
Module Num: 0x01004D42 0x00000108

Module ID: F-DI 8x24VDC HF2, BP V2
Module Num: 0x01004D43 0x00000108

Module ID: F-DI 8x24VDC HF2, BP V3
Module Num: 0x01004D45 0x00000108

Module ID: F-DI 8x24VDC HF2, XP
Module Num: 0x01004D42 0x00000308

**Obrázek 5.4:** Vyhledávání modulu v testovacím prostředí.

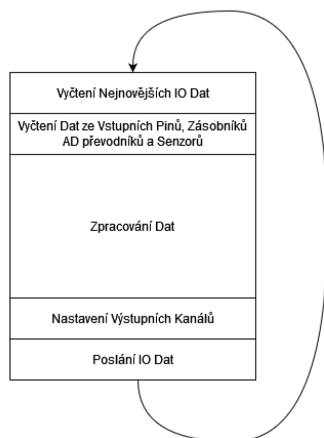
# Kapitola 6

# Model prostředí pro F-IO moduly

V této kapitole nastíním jak probíhá cyklus firmware v reálném F-IO modulu, jaké jsou rozdíly ve Virtuálním modulu a co je vlastně myšleno "modelem prostředí pr F-IO moduly".

## 6.1 Cyklus firmware F-IO modulu

Firmware F-IO modulů běží cyklicky v hlavní smyčce. Povrchní ilustrace je zobrazena na obrázku 6.1, ale na představení hlavních částí cyklu stačí.




**Obrázek 6.1:** Cyklus firmware F-IO modulu.

Cyklus začíná načtením nejnovějších IO dat. Následuje vyčtení vstupních pinů mikrokontroléru a načtení zásobníků hodnot AD převodníků. Následně probíhá zpracování a vyhodnocení dat. Na základě předchozího kroku se

nastaví výstupní kanály (pokud jimi modul disponuje). Na závěr se aktualizují IO data a případně se pošle alarm.

Cyklus ve Virtuálním modulu probíhá obdobě jen se mu musí některá data podvrhnout/nahradit. Jelikož když je spuštěn Virtuální modul, tak neexistuje žádné CPU, které by ho řídilo. Tím pádem se musí PROFINET komunikace nějak nahradit. Dále se také musejí nahradit data AD převodníků. Tato náhrada byla doposud řešena zadefinováním potřebných dat před kompilací Virtuálního modulu. Tedy uživatel zadefinoval nějaký scénář, kdy a jaké zprávy Virtuální modul obdrží a také jaké hodnoty se budou plnit do zásobníků AD převodníků. Následně proběhla kompilace a spuštění testu. Toto řešení má ovšem nevýhodu, že při změně scénáře je znova nutno čekat na kompilaci.

Tato práce rozšiřuje stávající řešení o novou možnost substituce části PROFIsafe komunikace a plnění zásobníků AD převodníků. Místo aby všechna data byla definována před kompilací Virtuálního modulu, generují se za běhu v testovacím prostředí. Tedy během simulace Virtuální modul posílá žádost o data do testovacího prostředí. Kde se data na základě tzv "modelu prostředí pro F-IO moduly" spočítají a následně se pošlou do Virtuálního modulu. Zároveň model prostředí slouží k nastavení IO dat.

## 6.2 Struktura modelu prostředí pro F-IO moduly

Model prostředí se skládá z komponent, jejichž propojením se dají modelovat výstupní i vstupní kanály. Celkem existuje 5 funkčních komponent.

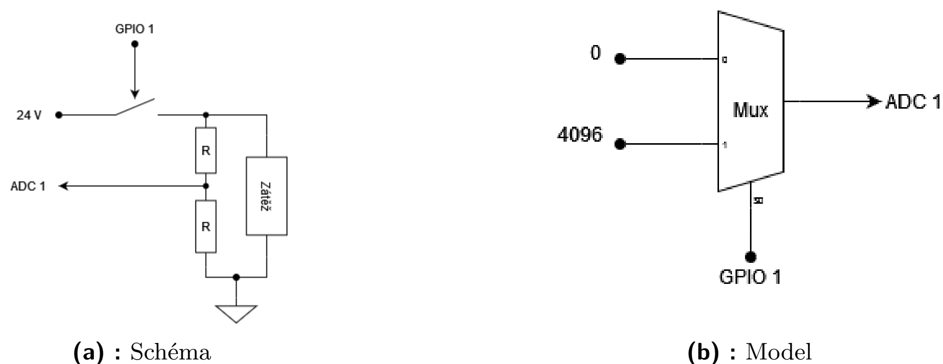
- "Konstanta" - Tato komponenta generuje konstantní hodnotu. Slouží jako náhrada určité hodnoty napětí.
- "GPIO" - Komponenta, která si pamatuje změny výstupního pinu Virtuálního modulu. Díky tomu dokáže simulovat výstupní pin.
- "Multiplexor" - Funguje jako klasický multiplexor, tedy na základě řídícího vstupu přepojí jeden ze vstupů na výstup. Slouží jako náhrada spínačů.
- "Výběr maxima" - Tato komponenta nastaví na výstup hodnotu nejvyššího vstupního signálu. Slouží k náhradě uzlů.
- "Uživatelský vstup" - Výstup této komponenty si může uživatel nastavit sám, slouží k simulování nestandardních podmínek.

### 6.2.1 Modelování vstupních a výstupních kanálů

Vezměme v potaz následující příklad, jednoduchého výstupního kanálu DQ modulu zobrazeného na obrázku 6.2a (ADC je připojeno přes odporový dělič

jelikož modul spíná 24 V, ale ADC mohou mít na vstupu maximálně 3,3 V.). V tomto případě můžou nastat dva případy. Spínač je sepnut a na ADC je přivedeno napětí okolo 3 V, spínač není sepnut a je na ADC je přivedeno 0 V.

Tento kanál se dá modelovat pomocí dvou komponent "Konstanta", které reprezentují stav napětí při sepnutém a rozepnutém spínači. (Hodnoty 0 a 4095 jsou mezní hodnoty, jelikož se STM32 mikrokontroler disponuje 12 bitovými AD převodníky, což odpovídá rozsahu [0, 4095]). Dále jedné komponenty "Multiplexor" fungující jako spínač a komponenty "GPIO", která řídí komponentu "Multiplexor". Celý model je znázorněn na obrázku 6.2b.

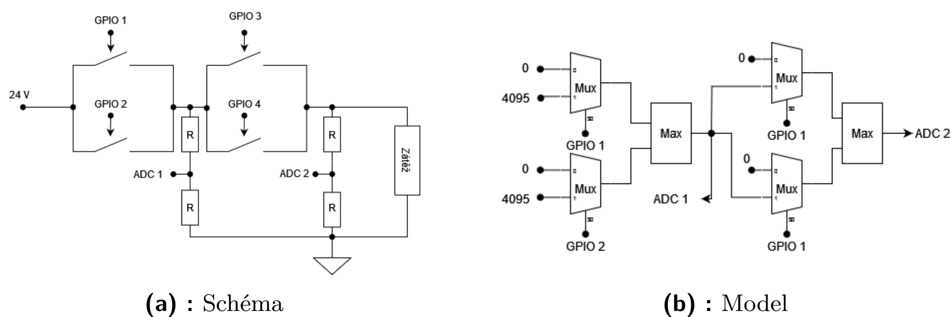



**(a) :** Schéma

**(b) :** Model

**Obrázek 6.2:** Schéma a model jednoduchého výstupního kanálu.

Toto byl úplně nejjednodušší příklad, většinou jsou spínače na kanálu paralelně, sériově nebo oběma způsoby zdvojeny jak je ukázáno na obrázku 6.3a. V tomto případě je nutné model rozšířit o komponentu "Výběr maxima", která dokáže nahradit uzly.



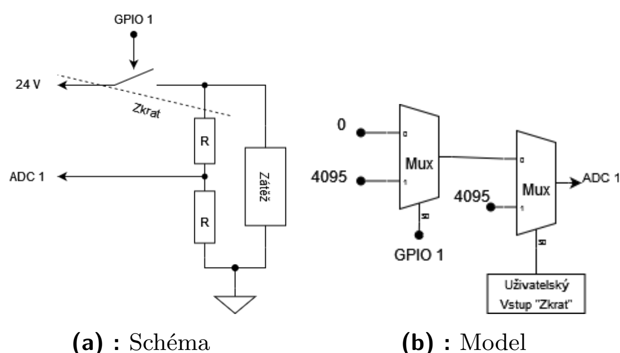

**(a) :** Schéma

**(b) :** Model

**Obrázek 6.3:** Schéma a model výstupního kanálu s paralelními spínači.

Předešlé příklady ukázaly jak navrhnout funkční kanál. K testování je potřeba simulovat chyby. Příkladem chyby je zkrat do 24 V, zobrazen na obrázku 6.4a. Toto lze simulovat komponentou "Uživatelský vstup", jak je znázorněno na obrázku 6.4b. V tomto případě pokud je nastavena komponenta "Uživatelský vstup" na hodnotu 0, nic se neděje a kanál funguje bez problémů.

V případe nastavení na hodnotu 1 nastává zkrat do 24 V.




**(a)** : Schéma　　**(b)** : Model

**Obrázek 6.4:** Schéma a model kanálu se simulovaným zkratem.

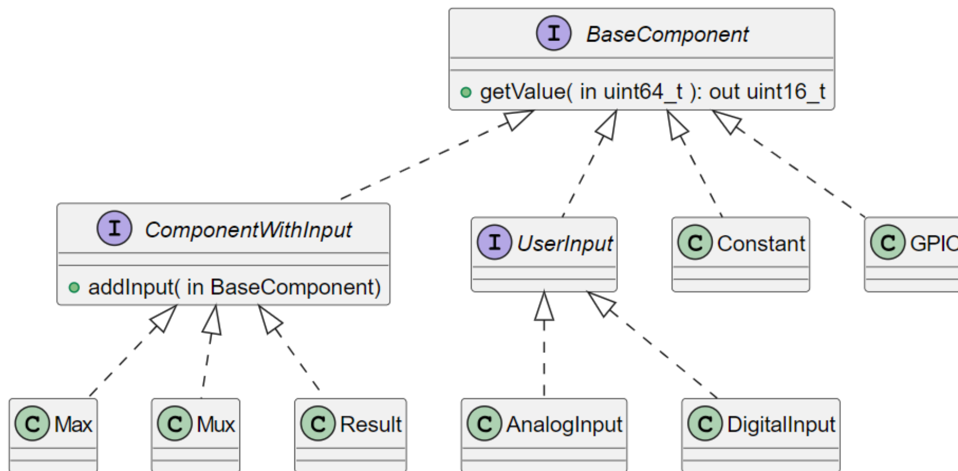
### 6.2.2 Náhrada IO dat

Model prostředí umožňuje nastavení části IO dat. Jedná se o část IO dat, kterou v realitě posílá CPU do modulu. Tato část IO dat, pro výstupní moduly, mimo jiné obsahuje informaci o nastavení výstupních kanálů ("zapnout"/"vypnout"). Pro digitálí moduly F-DQ, se jedná o jeden bajt dat. Alespoň pro F-DQ, které mají 8 nebo méně kanálů.

Tento bit se dá nastavit pomocí komponenty, která má `"id" = "IOData"`. Tedy pokud bude existovat komponenta s tímto `id` budou její hodnoty použity pro generování tohoto bajtu IO dat. Tato komponenta může být libovolného typu "Konstanta", "Uživatelský vstup" nebo jakýkoliv jiný.

## 6.3 Implementace jednotlivých komponent

K vytvoření modelu prostředí, představeného v předešlé sekci, je potřeba implementovat jednotlivé komponenty. Ve výsledku jsem vytvořil 7 tříd reprezentujících jednotlivé komponenty ( `Max` , `Mux` , `Result` , `AnalogInput` , `DigitalInput` , `Constant` , `GPIO` ).




**Obrázek 6.5:** Kompozice komponent modelu prostředí pro F-IO modul.

Pro implementaci komponent jsem zvolil návrhový vzor "Kompozice" [25][ss. 177-184]. Diagram zobrazující kompozici všech komponent je zobrazen na obrázku 6.5. Kde všechny komponenty kopírují jedeno společné rozhraní `BaseComponent`, které deklaruje jednu metodu.

```
getValue(in uint64_t): out uint16_t
```

Tato metoda slouží k určení hodnoty komponenty v daný čas. Vstupní proměnná, reprezentována datovým typem `uint64_t`, značí čas v mikrosekundách. Výstupní hodnota reprezentována datovým typem `uint16_t` reprezentuje hodnotu na výstupu, což je dostatečně velká hodnota, aby pokryla rozsah AD převodníků i IO Dat.

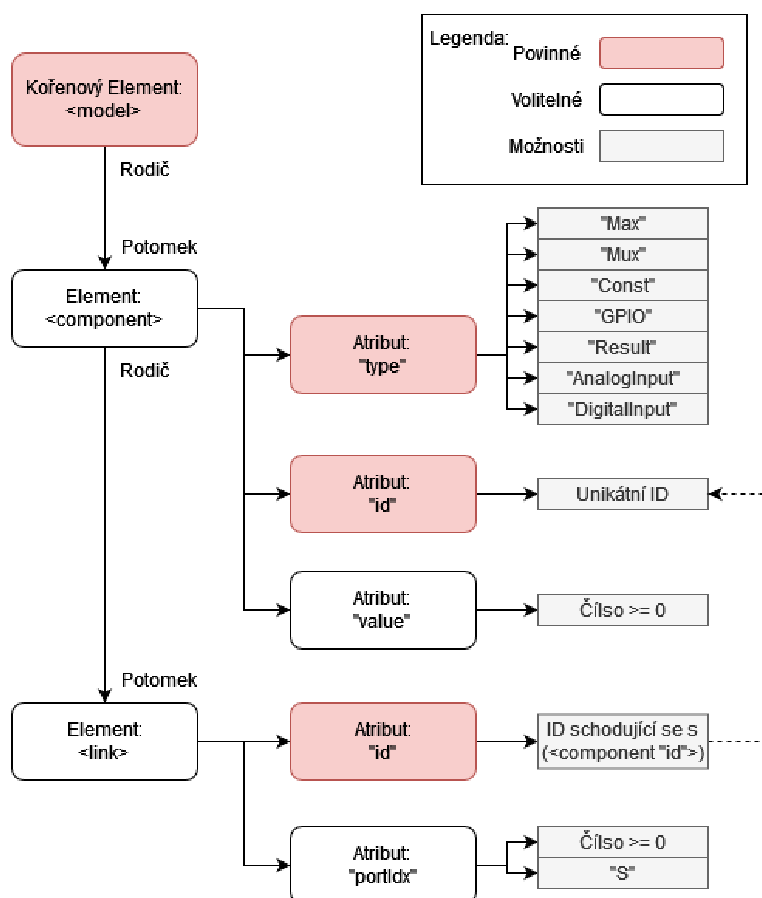
Jednotlivé komponenty implementují metodu `getValue()` různě třeba komponenta `Constant`, vždy vrátí přednastavenou hodnotu. Komponenta `GPIO` vrátí "1" nebo "0" v závislosti na tom jestli je příslušný pin aktivní.

Další skupinou komponent jsou komponenty které navíc implementují rozhraní `ComponentWithInput`, tedy to jsou komponenty které mají ostatní komponenty jako vstup. Komponenta `Max` vrátí nejvyšší hodnotu napojených komponent. Komponenta `Mux` vrátí hodnotu komponenty, na kterou ukazuje řídicí komponenta.

Poslední skupinou jsou komponenty implementující rozhraní `UserInput`, tyto komponenty vracejí hodnotu, kterou si nastaví uživatel. Jak se tyto komponenty nastavují je rozebráno v sekci 7.3.1.

## 6.4 Tvorba modelu prostředí F-IO modulů

K tvorbě modelu prostředí jsem navrhl schéma XML souboru popsaného na obrázku 6.6.

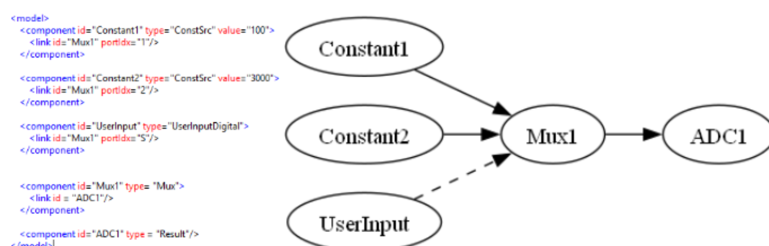



**Obrázek 6.6:** Schéma XML souboru pro tvorbu uživatelského prostředí.

Základem je kořenový element `<model>`, který obsahuje libovolný počet elementů `<component>`. Tyto elementy reprezentují jednotlivé komponenty, představené v předchozí sekci 6.3. Tyto elementy musejí obsahovat atribut `"type"`, který určuje typ komponenty, a atribut `"id"`, což je unikátní identifikátor komponenty sloužící k spojování komponent a identifikaci komponenty při simulaci. Dále je zde volitelný atribut `"value"`, k definování hodnoty komponenty `Constant`.

K spojování komponent slouží element `<link>`, který obsahuje atribut `"id"`, určující na jakou komponentu je daná komponenta napojena. Dále může obsahovat atribut `"portIdx"`, který slouží k upřesnění napojení na komponentu `Mux`. Buď je reprezentován číslem nebo písmenem "S", které

říklá že daná komponenta je napojená jako řídící. Příklad jednoduchého modelu napsaného jako XML je zobrazen na obrázku 6.7.



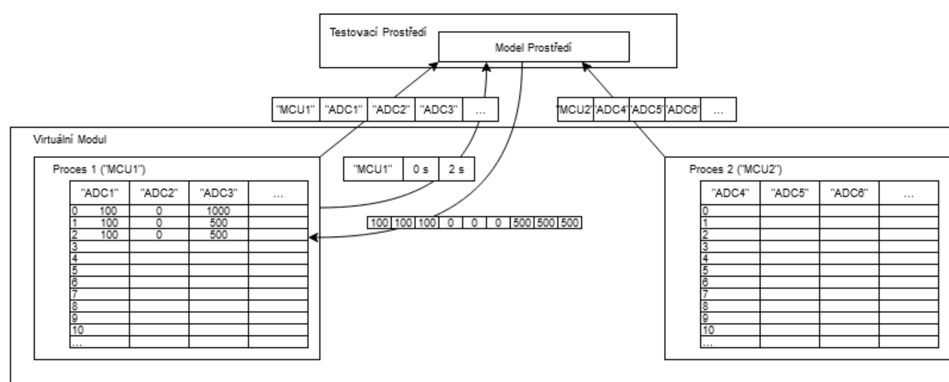

**Obrázek 6.7:** Zápis jednoduchého modelu prostředí v XML souboru.

### 6.4.1 Propojení modelu prostředí a Virtuálního modulu

Když Virtuální modul zažádá o data AD převodníků, podle čeho model prostředí pozná jaké komponenty má použít pro generování dat? K tomuto účelu slouží atribut `"id"`, kterým povinně disponuje každá komponenta.

To jaké `"id"` se budou používat určuje Virtuální modul. Uživatel testovacího prostředí se musí přizpůsobit a používat stejné `"id"`, jako Virtuální modul. Ve Virtuálním modulu pro každý AD převodník existuje zásobník, kde se ukládají jeho hodnoty. Půlka zásobníků se nachází v jednom procesu a půlka v druhém.

Modelová situace je znázorněna na obrázku 6.8. Po vytvoření komunikace, oba procesy Virtuálního modulu (každý reprezentuje jedno MCU) pošlou zprávu s informací, jaké `"id"` se mají používat pro výpočet hodnot AD převodníků pro daný proces. Taková zpráva obsahuje id procesu ("MCU1" nebo "MCU2") a následně list `"id"` daných komponent. Po této inicializační zprávě Virtuální modul posílá požadavky na hodnoty AD převodníků. Zpráva ve tvaru: "MCU1 potřebuje hodnoty AD převodníků v intervalu [0 s, 3 s].". Už nespecifikuje `"id"` jednotlivých komponent jelikož testovací prostředí už ví, jaké komponenty má použít pro vygenerování hodnot.



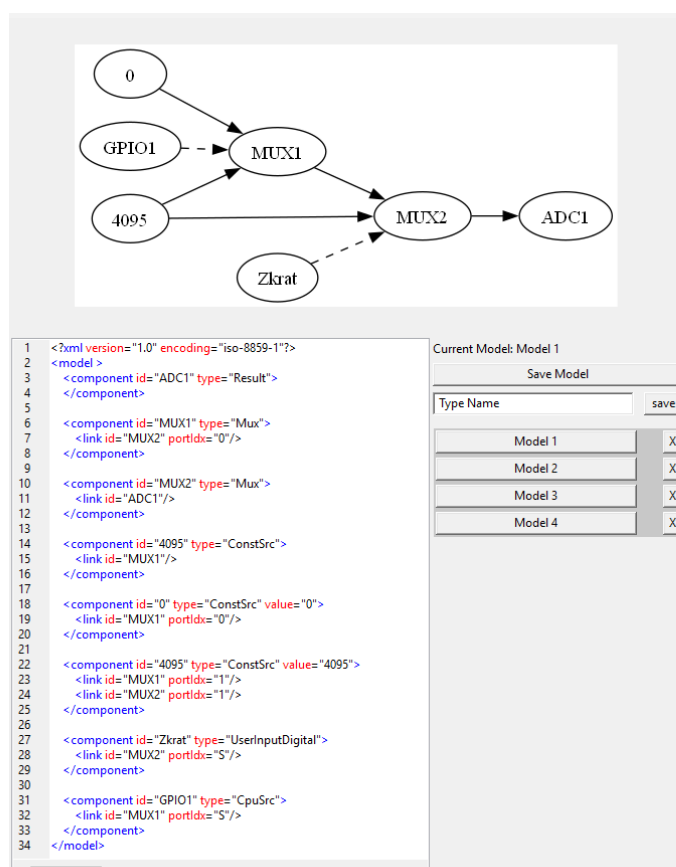

**Obrázek 6.8:** Schéma propojení modelu prostředí a Virtuálního modulu.

Identifikace komponent, které reprezentují výstupní piny, probíhá podobně. Jen při změně hodnoty pinu se odešle zpráva, která obsahuje `"id"` komponenty, čas změny a hodnota na kterou se pin změnil ("0", "1"). Hodnota je redundantní informace, jelikož při změně by se hodnota měla změnit na opačnou hodnotu. Ale pro případ, kdy by dvakrát za sebou přišla zpráva ohledně změny na stejnou hodnotu se posílá i hodnota.

### 6.4.2 Vytváření modelu prostředí v testovacím rozhraní

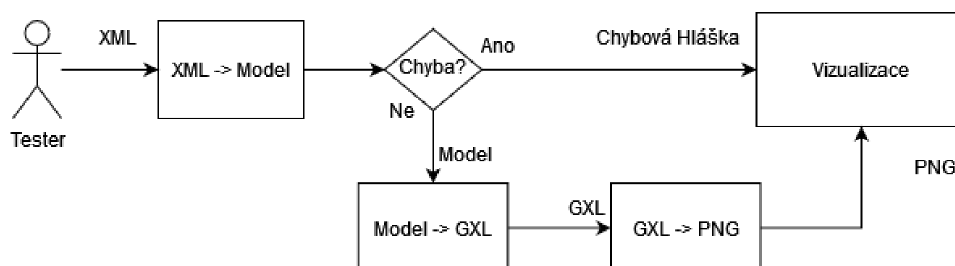
V uživatelském testovacím rozhraní jsem vytvořil jednoduchý editor modelu prostředí. Tento editor, zobrazený na obrázku 6.9, se skládá ze tří částí. Tyto části jsou textový editor, vizualizace a panel pro ukládání a přepínání mezi jednotlivými modely.

Pro vytvoření textového editoru jsem využil již hotového řešení, které je implementováno v knihovně `wxWidgets` třídou `wxStyledTextCtrl` [26], což je implementace knihovny `Scintilla`. `Scintilla` je knihovna, která implementuje základní funkce úpravy textu [27]. Díky této třídě jen stačilo udělat vizuální úpravy jako zvýraznění klíčových slov a číslování řádků.




**Obrázek 6.9:** Editor prostředí pro F-IO moduly.

Jelikož XML soubor pro definici modelu prostředí může být docela složitý a nepřehledný. Tak jsem se rozhodl vytvořit jednoduchou vizualizaci modelu prostředí pomocí atributovaného grafu. K vizualizaci jsem využil knihovnu *Graphviz*, která dokáže generovat nejrůznější grafy. Problém je, že tato knihovna vyžaduje na vstupu kód v jazyce *DOT*, který slouží k popisu grafů [13]. Celý postup vizualizace je zobrazen na obrázku 6.10.




**Obrázek 6.10:** Kompozice komponent modelu prostředí pro F-IO Modul.

Uživatel nejprve vytvoří XML soubor, ve kterém definuje model prostředí. Z tohoto XML souboru se vytvoří model prostředí, tvořený jednotlivými

komponenty. Pokud se to nepovede zobrazí se chybová hláška. V případě úspěchu se model prostředí převede do jazyka *DOT*. Následně již lze využít knihovnu *Graphviz* a vygenerovat obrázek ve formátu *Portable Network Graphic* (PNG), který je nakonec zobrazen.

Tímto způsobem je možno si průběžně kontrolovat jak model vypadá a jestli jsou jednotlivé komponenty propojeny jak uživatel zamýšlel. Zároveň se průběžně kontrolují syntaktické chyby.

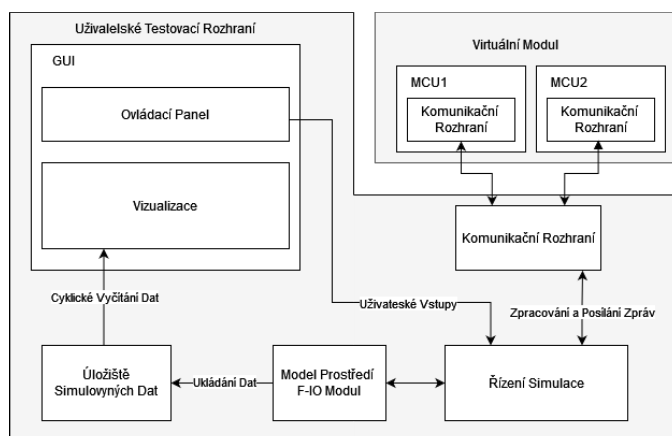
## Kapitola 7

# Ovládání a vizualizace simulace

V této kapitole vysvětlím jak probíhá simulace. Jak lze simulaci z uživatelského testovacího prostředí ovládat. Následně rozeberu jak je simulace vizualizována. Nakonec demonstruji základní využití simulačního prostředí na jednoduchém příkladu.

## 7.1 Schéma simulace

Schéma, které popisuje strukturu simulace je zobrazeno na obrázku 7.1. Struktura se skládá z pěti hlavních bloků.




**Obrázek 7.1:** Schéma simulace.

Jádrem simulace je blok **Řízení simulace**, který zpracovává a posílá zprávy Virtuálnímu Modulu. K tomu využívá blok **Komunikační rozhraní**,

které bylo představeno v kapitole 4.6. Zároveň reaguje na uživatelské vstupy, které uživatel zadává v bloku **Ovládací panel**. Blok **Řízení simulace** je poměrně obsáhlý a komplikovaný proto jsem se jeho podrobnému popisu rozhodl věnovat celou sekci 7.2.

Blok **Ovládací panel**, který je součástí GUI, umožňuje uživateli nastavení typu komunikace a periodu vzorkování AD převodníků (Všechny AD převodníky v rámci jednoho modulu mají stejnou periodu vzorkování, ale modul od modulu jsou odlišné.). Dále umožňuje uživateli spuštění, zastavení a restartování simulace.

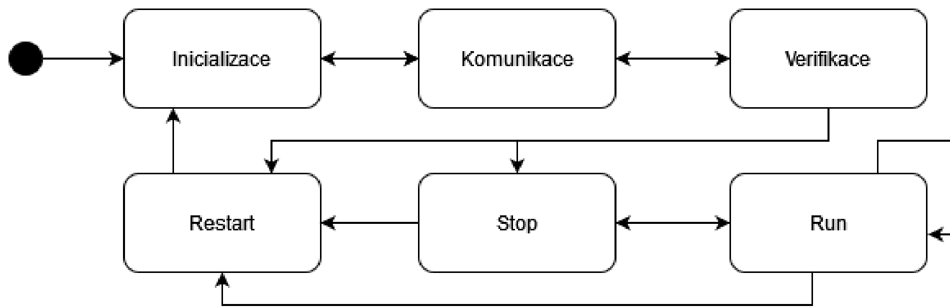
Blok **Model prostředí F-IO modulu**, je blok obsahující model prostředí představený v kapitole 6. Tento blok je zodpovědný za simulování IO dat a hodnot AD převodníků. Zároveň se veškeré simulované data ukládají v bloku **Úložiště simulovaných dat**.

Posledním blokem je **Vizualizace**, který cyklicky vyčítá nejnovější data z **Úložiště simulovaných dat** a následně data vizualizuje ve formě grafů. Tento blok je taktéž poměrně obsáhlý proto je mu věnována samostatná sekce 7.3.

## 7.2 Řízení simulace

Blok **Řízení simulace**, je implementován jako stavový automat, který je zobrazen na obrázku 7.3. Stavy se buď přepínají automaticky jak bude popsáno v následujícím odstavci nebo je přepíná uživatel v GUI, pomocí ovládacího panelu, který je zobrazen na obrázku 7.2.

**Obrázek 7.2:** Ovládací panel.




**Obrázek 7.3:** Stavový automat simulace.

**Inicializace**
: Je počáteční stav. V tomto stavu si uživatel může zvolit, parametrizaci, upravit model prostředí, zvolit periodu vzorkování AD převodníků a způsob komunikace. Následně pokud je vše nastaveno uživatel může stisknutím tlačítka *"Start"* přejít do stavu **Komunikace**.

**Komunikace**
: Tento stav se stará o inicializaci komunikace mezi Virtuálním modulem a testovacím rozhraním. Pokud je spojení úspěšně vytvořeno přechází se automaticky do stavu **Verifikace**. Uživatel také může stisknutím tlačítka *"Restart"* přejít do stavu **Inicializace**.

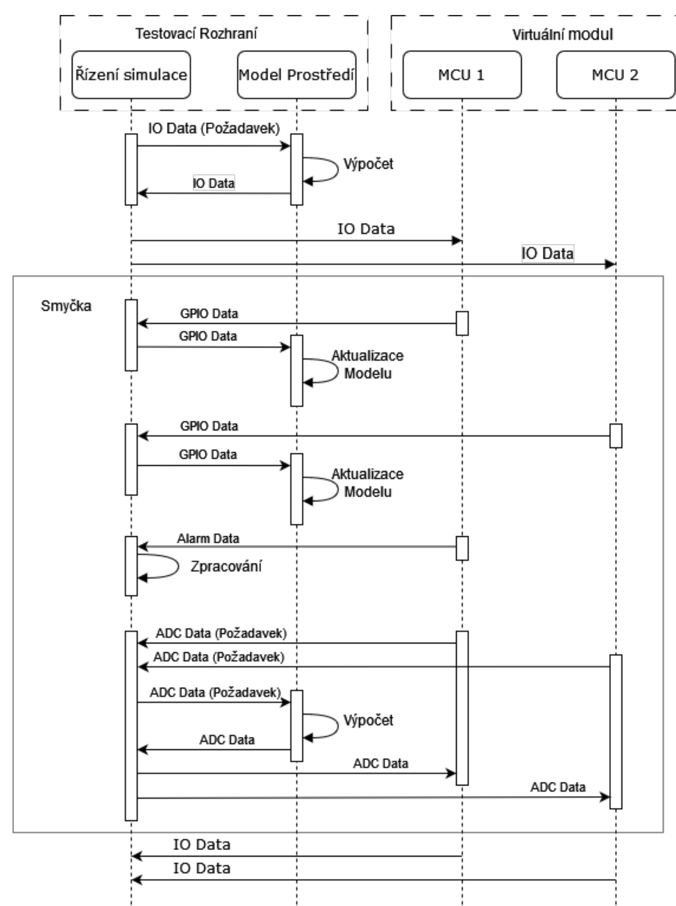
**Verifikace**
: Tento stav pošle parametrizaci do Virtuálního modulu. Dále se také ověří zdali model prostředí obsahuje potřebné komponenty. Pokud ověření proběhne úspěšně přejde se do stavu **Stop**. Pokud model prostředí neobsahuje potřebné komponenty, nebo uživatel zmáčkne tlačítko *"Restart"*, přechází se do stavu **Restart**.

**Restart**
: Tento stav pošle zprávu Virtuálnímu modulu, že došlo k restartování simulace. Virtuální modul se pak taky nastaví do výchozího stavu a čeká na novou parametrizaci. Následně se automaticky přejde do stavu **Inicializace**.

**Stop**
: V tomto stavu je simulace zastavena a nic se neděje. Uživatel buď může spustit simulaci, stisknutím tlačítka ">" nebo "≫". V obou případech se přejde do stavu **Run**, jen v případě ">" se odsimuluje jen jeden cyklus firmware a v případě *"≫"* se spustí automatický režim, ve kterém simulace poběží dokud uživatel nerestartuje simulaci, nebo nezmáčkne tlačítko "||".

**Run**
: Tento stav řídí samotný chod simulace. Simulace běží po jednotlivých cyklech firmware. (Jak probíhá reálný cyklus firmware je popsáno v sekci 6.1.) Průběh simulace jednoho firmware cyklu je popsán v následující podsekci 7.2.1. Pokud je nastavený automatický režim simulace, následuje znovu stav **Run**. Pokud ne, přechází se po dokončení cyklu do stavu **Stop**.

### 7.2.1 Průběh cyklu firmware v simulaci




**Obrázek 7.4:** Sekvenční diagram simulace cyklu firmware.

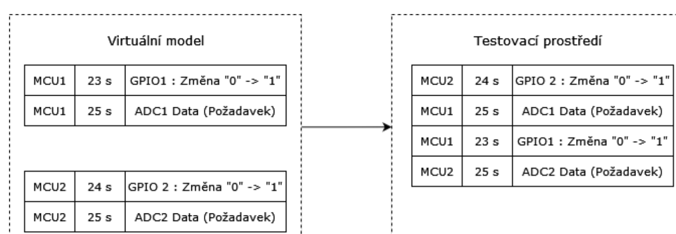
Celý firmware cyklus začíná zjištěním momentálních IO dat, která se následně zašlou do Virtuálního modulu. Při obdržení této zprávy Virtuální modul spouští další cyklus firmware.

Během průběhu cyklu Virtuální modul může zaslat zprávu o změně výstupního pinu. Na což blok **Řízení simulace** reaguje aktualizací dané komponenty v modelu prostředí. Taktéž může být zaslána zpráva o chybě (alarm). Pokud blok **Řízení simulace** zaznamená tuto zprávu, předá tuto informaci bloku **Vizualizace** a ten následně zobrazí příslušnou chybovou hlášku.

Taktéž si Virtuální modul může vyžádat data AD převodníků. Aby blok **Řízení simulace** dal pokyn pro výpočet hodnot AD převodníků musí obdržet požadavek od obou procesů. Po obdržení obou požadavků se vypočtou hodnoty AD převodníků a výsledky se zašlou zpět.

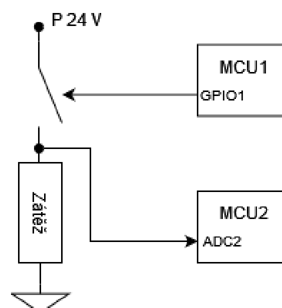
Zprávy o změně GIPO a požadavek na hodnoty AD převodníků můžou být poslány během jednoho cyklu firmware vícekrát. Virtuální modul hlásí konec cyklu, zasláním nových IO dat.

Proč se čeká na oba požadavky dat AD převodníků? Důvod je kvůli synchronizaci. Virtuální modul je sice synchronizován způsobem, že se oba procesy ptají na data AD převodníků současně. "Současně" je myšleno v čase simulace v realitě zprávy do testovacího rozhraní mohou dorazit zpřeházeně, jak je zobrazeno na obrázku 7.5.




**Obrázek 7.5:** Znázornění zasílání zpráv mezi Virtuálním modulem a testovacím rozhraním.

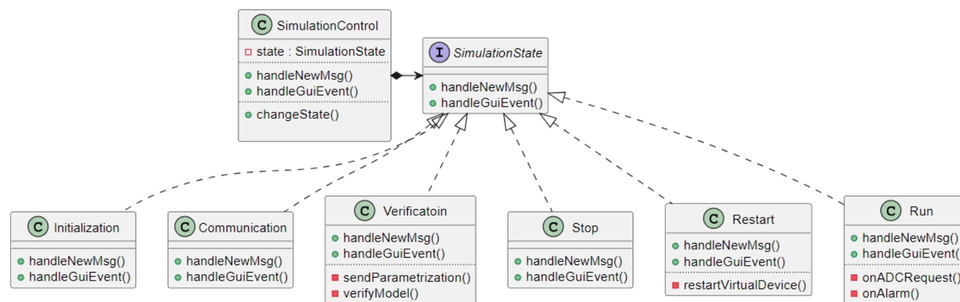
Kdyby se ale nečekalo na oba požadavky a zároveň by AD převodník příslušící jednomu mikrokontroléru byl za spínačem, který je řízen druhým mikrokontrolérem, obrázek 7.6. (Což se děje v rámci vzájemné kontroly mikrokontrolérů téměř vždy.) Tak by mohlo dojít k vypočtení chybných hodnot *"ADC2"*. Stačilo by aby zpráva o změně *"GPIO1"* přišla až po požadavku na výpočet hodnoty "ADC2".




**Obrázek 7.6:** Schéma vzájemné kontroly mikrokontrolérů.

### 7.2.2 Implementace řízení simulace

Jak, bylo zmíněno blok **Řízení Simulace**, je stavový automat. Proto jsem se k jeho implementaci rozhodl použít návrhový vzor "Stav" [25][ss. 352-360]. Má implementace tohoto návrhového je zobrazena na obrázku 7.7.




**Obrázek 7.7:** Struktura bloku Řízení simulace.

Třída `SimulationControl` implementuje tři veřejné metody. Metoda `handleGuiEvent()` slouží ke zpracování uživatelského vstupu (kliknutí na tlačítko). Metoda `handleNewMsg()` souží ke zpracování nové zprávy z Virtuálního modulu. Tato třída nedefinuje chování těchto metody, pouze deleguje práci na konkrétní stav. Každý stav je reprezentován svojí třídou, která implementuje rozhraní `SimulationState` a až tyto třídy definují co se stane při obdržení zprávy nebo uživatelského vstupu. Zároveň logika přepínání není koncentrovaná na jednom místě, ale je distribuována mezi jednotlivé stavy, kde si každý stav definuje možnosti přechodu do jiného stavu. Tento přístup se vyhýbá složitým podmínkám pro zpracování vstupů a přechodů mezi stavy. Navíc je relativně jednoduché ho rozšířit.

## 7.3 Vizualizace simulace

Vizualizace se skládá z grafů signálů jednotlivých komponent modelu prostředí, jak je zobrazeno na obrázku 7.8.

**Obrázek 7.8:** Vizualizace průběhu simulace v GUI.

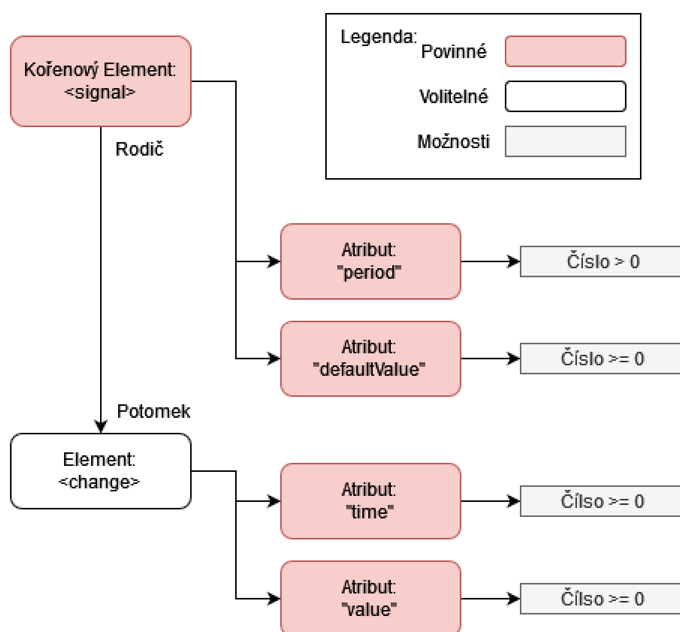
Každé komponentě přísluší jeden graf. Jelikož ne všechny grafy se vejdou na obrazovku, tak si uživatel si může v levé části zvolit, které komponenty budou zobrazeny. Zároveň ne všechny grafy jsou zajímavé, třeba grafy konstant jsou jen přímky, ale i přes to jsem je přidal do vizualizace. Alespoň si uživatel může zkontrolovat že se všechny komponenty chovají jak mají.

Pokud je simulace spouštěna po jednotlivých cyklech firmware vykreslování grafů probíhá po dokončení každého cyklu. Pokud je spuštěn automatický režim tento způsob by neobstál jelikož se může stát, že by se musely grafy vykreslit více než 200 krát za sekundu. Což běžné monitory nestíhají vykreslit a ani by se nestíhaly spočítat a vygenerovat nové grafy. V automatickém režimu jsem tedy omezil vykreslování grafů na frekvenci 40 Hz.

Další částí vizualizace je zobrazení historie chybových hlášek, které se nachází v pravé části. Tato historie uživatele informuje jaké chyby modul detekoval a kdy k nim došlo. Také lze nastavit přerušení simulace pokud se detekuje chybová hláška, následně uživatel může analyzovat signály jednotlivých komponent a jestli se modul zachoval podle specifikací.

### 7.3.1 Nastavení uživatelských vstupů

Již v sekci 6.3, byly představeny komponenty představující uživatelský vstup. Hodnoty na těchto komponentách může uživatel nastavit pomocí XML souboru, jehož schéma je zobrazeno na obrázku 7.9.

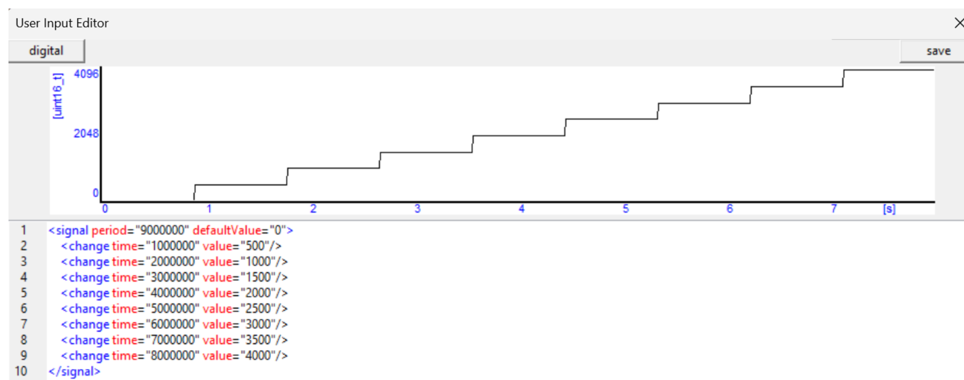



**Obrázek 7.9:** XML schéma pro definování uživatelského vstupu.

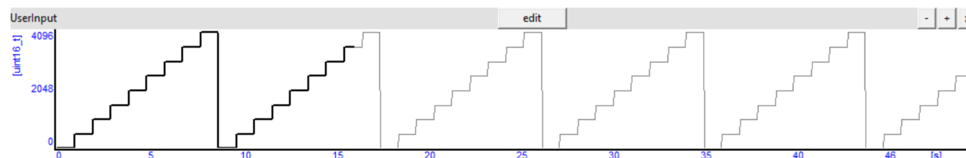
Základem je kořenový element `<signal>`, který povinně obsahuje atributy `"period"` a `"defaultValue"`. Dále může obsahovat libovolné množství elementů `<change>`. Atribut `"period"`, udává periodu signálu (v mikrosekundách). Atributem `"defaultValue"` udává hodnotu signálu na počátku. Element `<change>` udává změny v signálu. Obsahuje dva atributy a to jsou, `"time"` udávající, kdy došlo ke změně v mikrosekundách a `"value"` udávající hodnotu, na kterou se signál změnil.

Dále jsem vytvořil jednoduchý editor, zobrazený na obrázku 7.10. Tento editor umožňuje editovat XML soubor a zároveň znázorňuje jak vytvářený signál vypadá.

Dále komponenty uživatelského vstupu mají implementovanou vizualizaci předpovědi (Šedá čára na obrázku 7.11). Také lze měnit uživatelské vstupy za běhu simulace.




**Obrázek 7.10:** Editor uživatelského vstupu.




**Obrázek 7.11:** Vizualizace komponent s uživatelským vstupem.

# Závěr

Cílem práce bylo rozšít Simulační prostředí pro Siemens F-IO moduly, které se využívá převážně pro unit testování, aby bylo vhodné i pro explorativní testování. Toho bylo docíleno vytvořením aplikace, která slouží jako testovací uživatelské rozhraní. Tato aplikace splňuje cíle práce zmíněné v sekci 2.5, včetně volitelné vizualizace.

V prvních dvou kapitolách byly představeny F-IO moduly z řady ET 200SP a byla rozebrána problematika testování softwaru těchto modulů, včetně využití Simulačního prostředí při testování. Zbylé kapitoly byly věnovány tvorbě rozšiřující aplikace. Tato aplikace umožňuje výběr a parametrizaci testovaného modulu. Dále možnost vytvoření prostředí, díky kterému se dají simulovat různé chybové scénáře. Toto prostředí lze vytvářet přímo v aplikaci pomocí XML schématu v integrovaném editoru. Následně lze jednoduše spouštět simulaci a výsledky analyzovat pomocí vytvořené vizualizace.

V budoucnu by bylo vhodné vymyslet lepší řešení komunikace mezi Windows a WSL procesy. Je možné rozšířit model prostředí o nové komponenty, aby byla možnost vytváření sofistikovanějších testovacích scénářů. Největší potenciál pro rozšíření je však ve vizualizaci, kde by bylo hezké přidání možnosti procházet signály zpětně. Zároveň zadávání uživatelských vstupů by se dalo rozšířit o možnost nakreslení signálu, popřípadě zadání pomocí matematické funkce.

# Kapitola 9

# Seznam zkratek

| **Zkratka** | **Význam** |
|---|---|
| ADC | *Analog-to-Digital Converter* (Analogově-digitální převodník) Elektronická součástka pro převod spojitého signálu na diskerétní. |
| AI, F-AI | *Fail-save / Analog Input* (Bezpečnostní / Analogový vstup) V textu používáno jako označení analogového vstupního modulu. |
| API | Application Programming Interface (Rozhraní pro programování aplikací) |
| AQ, F-AQ | *Fail-safe / Analog Output* (Bezpečnostní / Analogový výstup) V textu používáno jako označení analogového výstupního modulu. |
| CPU | *Central Processing Unit* (Centrální procesorová jednotka) Řídicí část PLC systému. |
| DI, F-DI | *Fail-safe / Digital Input* (Bezpečnostní /Digitální vstup) V textu používáno jako označení digitálního vstupního modulu. |
| DQ, F-DQ | *Fail-safe / Digital Output* (Bezpečnostní / Digitální výstup) V textu používáno jako označení digitálního výstupního modulu. |
| FPGA | *Field-Programmable Gate Array* (Programovatelné hradlové pole) |
| GPIO | *General-Purpose Input/Output* (Univerzální vstup/výstup) |
| GSD | *General Station Description* Formát souboru s popisem zařízení, pro PROFIBUS technologii. |
| GSDML | *General Station Description Markup Language* Formát soubor s popisem zařízení, pro PROFINET technologii. |
| GUI | *Graphical User Interface* (Grafické uživatelské rozhraní) |
| IRT | *Isochronous Real Time* Zprávy v síti PROFINET z nejvyšší prioritou. Také označovány jako RTC3 *Real-time Class 3.* |
| MCU | *Microcontroller Unit* (Mikrokontrolér) |
| MPU | *Microprocessor Unit* (Mikroprocesor) |

| | |
|---|---|
| NRT | *Non-Real Time* Zprávy v síti PROFINET z nejnižší prioritou. |
| PLC | *Programmable Logic Controller* (Programovatelný logický automat) |
| PL / $PL_r$ | *Performance Level* |
| PNG | *Portable Network Graphics* Grafický formát |
| RT | *Real Time* Zprávy v síti PROFINET s vyšší prioritou. Také označovány jako RTC1 *Real-time Class 1*. |
| SIL | *Safety Integrity Level* (Úroveň integrity bezpečnosti) |
| STM32 | Rodina 32bitových mikrokontrolérů od STMicroelectronics |
| TCP | *Transmission Control Protocol* |
| TIA | *Totally Integrated Automation* Zkratka názvu aplikace pro programování Siemens PLC systémů. |
| WSL | *Windows Subsystem for Linux* (Windows Subsystém pro Linux) Součást systému Windows, která dokáže spustit linuxové prostředí na zařízení s Windows. |
| XML | *Extensible Markup Language* |

# Literatura


[1] M. Saric, "Simulation-based testing of failsafe industrial peripheral modules," Březen 2019.

[2] P. Burget, "Principy komunikace a diagnostika sítí profinet," *AUTOMA*, no. 5, 2013. [Online]. Dostupné z: https://www.automa.cz/cz/casopis-cislo/automa-2013_05/

[3] F. D. Petruzella, "Programmable logic controllers, fifth edition." 2017.

[4] Siemens AG, "SIMATIC functional manual S7-1500, ET 200MP, ET 200SP." [Online]. Dostupné z: https://support.industry.siemens.com/cs/document/59192926/simatic-s7-1500-et-200mp-et-200sp-et-200al-et-200pro-et-200eco-pn-diagnostics?dti=0&lc=en-CZ

[5] M. Bowne, "The difference between PROFINET and PROFIsafe." [Online]. Dostupné z: https://us.profinet.com/the-difference-between-profinet-and-profisafe/

[6] R. SM, "What is the windows subsystem for linux?" [Online]. Dostupné z: https://www.softwaretestingmaterial.com/integration-testing/?utm_content=cmp-true

[7] L. Žoltá, "Testovací techniky." [Online]. Dostupné z: http://lucie.zolta.cz/index.php/statnice-vsb/170-testovaci-techniky

[8] Společnost Qt, "Qt documentation." [Online]. Dostupné z: https://doc.qt.io/

[9] V. tým wxWidgets, "wxwidgets documentation." [Online]. Dostupné z: https://docs.wxwidgets.org/3.0/

[10] J. Smart, "The wxwindows library licence." [Online]. Dostupné z: https://opensource.org/license/wxwindows-php

[11] N. M. Josuttis, “The c++ standard library - a tutorial and reference, 2nd edition,” 2012.

[12] M. Kalicinski, “Rapidxml.” [Online]. Dostupné z: https://rapidxml.sourceforge.net/

[13] Graphviz Development Team, “Graphviz.” [Online]. Dostupné z: https://graphviz.org/

[14] S. White a spol., “Windows sockets 2.” [Online]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/winsock/windows-sockets-start-page-2

[15] C. Loewen a spol., “What is the windows subsystem for linux?” [Online]. Dostupné z: https://learn.microsoft.com/en-us/windows/wsl/about

[16] N. M. Josuttis, “The c++ standard library - a tutorial and reference, 2nd edition,” 2012.

[17] S. Cooley, “Af_unix comes to windows.” [Online]. Dostupné z: https://devblogs.microsoft.com/commandline/af_unix-comes-to-windows

[18] C. Wilhite, “Windows/wsl interop with af_unix.” [Online]. Dostupné z: https://devblogs.microsoft.com/commandline/windowswsl-interop-with-af_unix

[19] J. Novák, “Komunikace a distribuované systémy b3b38kds.” [Online]. Dostupné z: https://reference.opcfoundation.org/PNGSDGM/v100/docs/6.7

[20] R. H. Derrick Rountree, “Loopback address.” [Online]. Dostupné z: https://www.sciencedirect.com/topics/computer-science/loopbackaddress

[21] Společnost Microsoft Corp., “Winsock.h header.” [Online]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/api/winsock/

[22] PROFIBUS Nutzerorganisation e.V., “Gsdml / gsdx specification for profinet.” [Online]. Dostupné z: https://www.profibus.com/download/gsdml-gsdx-specification-for-profinet

[23] P. University, “Profinet gsd file basics.” [Online]. Dostupné z: https://profinetuniversity.com/profinet-basics/profinet-gsd-file-basics/

[24] P. University, “Opc foundation.” [Online]. Dostupné z: https://reference.opcfoundation.org/PNGSDGM/v100/docs/6.7

[25] A. Shvet, “Dive into design patter,” 2018.

[26] wxWidgets, “wxStyledTextCtrl class reference.” [Online]. Dostupné z: https://docs.wxwidgets.org/3.0/classwx_styled_text_ctrl.html

[27] N. Hodgson, “Scintilla documentation.” [Online]. Dostupné z: https://www.scintilla.org/ScintillaDoc.html