ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Softwarové PLC

Diplomová práce

Vladimír Kloz

Vedoucí práce: *Ing. František Vacek*

Praha, 2003

Prohlášení

Prohlašuji, že jsem diplomovou práci včetně doprovodného programového vybavení vypracoval samostatně s použitím uvedené literatury a konzultací s vedoucím diplomové práce Ing. Františkem Vackem.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb. , o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne: 13.ledna 2004 *Vladimír Kloz*

Poděkování

Děkuji panu Ing. Františku Vackovi za cenné rady a připomínky, které mi poskytl během konzultací při zpracování diplomové práce.

Vladimír Kloz

Abstrakt

Cílem této diplomové práce je vytvořit softwarové PLC s vlastnostmi definovanými v normě IEC-1131-3, které bude fungovat pod operačním systémem Real-Time Linux. Celý systém je vytvořen jako preprocesor ze zvoleného programovacího jazyka pro PLC (zvolen byl jazyk Instruction List) do jazyka C++.

Výsledkem překladu programu pro PLC je modul zaveditelný do oblasti kernelu operačního systému Real-Time Linux, který prezentuje program pro PLC.

Součástí práce je také vytvoření základní podpory pro přístup z PLC programu na zařízení digitálních vstupů/výstupů (implementovány jsou např. ovladače pro paralelní port, ovladač PCI karty Labcard PCI-1750 atd.).

Abstract

The intention of this thesis to realize software PLC with features defined in the norm IEC-1131-3, which will run under Real-Time Linux operating system. The system is designed as preprocesor from selected programming language for PLC (selected language was Instruction List) into C++ language.

The result of PLC program compilation is Real-Time Linux kernel module, that is representing software PLC.

The part of this thesis is also creation of basic support for access to digital inputs/outputs from PLC program (implemented are drivers for parallel port, PCI card PLX-1750 etc.).

Obsah

Úvod

Cílem této práce je vytvořit takzvané *Softwarové PLC*. Pod pojmem *Softwarové PLC* si v našem případě představujeme softwarový emulátor programové kódu určeného pro PLC. Tento emulátor by měl být vytvořen pod operačním systémem RT-Linux na procesorové platformě Intel, měl by podporovat vybraný programovací jazyk pro PLC a komunikovat s PCI kartou digitálních vstupů/výstupů. PLC by mělo splňovat vlastnosti definované normou IEC-1131[5].

PLC je zkratka pro *programovatelný logický automat*. Jedná se o zařízení určené zejména pro řídící a regulační techniku. V podstatě je to mikroprocesorem osazený systém obvykle umožňující rozšíření o další moduly (např. moduly digitálních vstupů/výstu-pů, A/D a D/A převodníky, moduly pro průmyslové sběrnice atd.). Ve většině případů lze tyto systémy programovat pomocí třech základních programovacích jazyků: IL (instruction list - obdoba assembleru, ST (structured text - obdoba jazyka Pascal) a tzv. žebříčkových diagramů. Instrukční soubor je velmi jednoduchý se zaměřením na řízení a regulaci. Programovací jazyky pro PLC jsou detailně popsány v [5].

Operační systém Linux je volně šiřitelný systém Unix dostupný včetně zdrojových kódů. Vlastní Linux je pouze jádro operačního systému. Operační systém je šířen v distribucích[1], které kromě jádra operačního systému obsahují také další utility a programy. Na základě klasického jádra (kernelu) byl následně vytvořen tzv. RT-Linux. Jedná se o rozšíření kernelu o vlastnosti hard real-time operačního systému, přičemž jsou zachovány předchozí výhody normálního jádra a případné zvolené distribuce ve smyslu existence a použitelnosti ovladačů zařízení a utilit. Vlastní real-time úkoly poté běží s vyšší prioritou než linuxové jádro jako real-time vlákna v paměťovém prostoru jádra.

V části 1 je uveden rozbor řešení a volba následné implementace. Kapitola 2 se v krátkosti zabývá historií a vlastnostmi operačního systému RT-Linux a možnostmi jazyka C++ při využití v RT-Linuxu. Kapitola 3 se zabývá způsobem zpracování zdrojového programu pro PLC pomocí konečného stavového automatu. Kapitoly 4, 5, 6 a 7 se věnují zpracování jednotlivých částí zdrojového kódu. Kapitola 2.1 popisuje způsob implementace a podpory v operačním systému RT-Linux, včetně vytváření ovladačů zařízení pro využití v softwarovém PLC. Kapitola 9 popisuje vlastní utilitu plc2cpp a její využití při tvorbě softwarového PLC. Příloha B uvádí celkový přehled vlastností vytvořeného softwarového PLC v porovnání s normou IEC-1131-3. V příloze A je uve-

[1] Např. RedHat Linux, Debian, Slackware atp.

den krátký návod na zprovoznění RT-Linuxu. Příloha C se krátce zabývá vytvořeným ukázkovým PLC programem pro řízení otáček stejnosměrného motorku.

Kapitola 1

Návrh řešení

Norma IEC-1131-3 zabývající se problematikou programovacích jazyků pro PLC definuje základní programovací jazyky uvedené v tab.1.1. Jazyky IL, ST a LD mají tu výhodu, že jsou mezi sebou navzájem převoditelné, pro vlastní kompilaci programu stačí implementace libovolného z nich a následné vytvoření konverzní utility z ostatních programovacích jazyků.

Název jazyka	**Popis**
IL	Instruction List - jednoduchý programovací jazyk, obdoba assembleru.
ST	Structured Text - pokročilejší programovací jazyk, obdoba jazyka Pascal či C.
LD	Ladder diagram - grafický nástroj pro programování PLC, automat se v tomto případě programuje pomocí reléového schematu.

Tabulka 1.1: Přehled programovacích jazyků pro PLC dle IEC-1131-3.

Pro zápis programového kódu PLC byl zvolen jazyk IL s podporou standardních datových typů, číselných typů (pro celá i desetinná čísla) a datového typu pro uložení času[1]. Z bloků pro zápis vlastního PLC programu byly vybrány bloky uvedené v tab.1.2.

Dále se budeme zabývat analýzou možných řešení a následnou volbou vlastní implementace.

Celkově lze problém řešit buď jako interpret zdrojového kódu pro PLC nebo jako překladač PLC kódu do jiného programovacího jazyka, ze kterého bude následně zkompilována binární podoba programu, který odpovídá programu PLC.

1. **Interpret PLC kódu** - výhodou tohoto řešení je jednodušší použití z hlediska uživatele. Odpadá totiž nutnost po každé změně programu provádět jeho rekompilaci.

[1]Tento datový typ je třeba zejména kvůli definici spouštění PLC programů v konfiguračním bloku.

Název bloku	**Popis**
FUNCTION	Blok typu funkce. Jedná se o blok umožňující volat kód PLC programu s alespoň jedním vstupním parametrem a návratovou hodnotou, která se po návratu z volání uloží jako aktuální pracovní hodnota na zásobníku. Při každém volání jsou opětovně inicializovány všechny proměnné.
FUNCTION_BLOCK	Funkční blok. Jedná se o blok, který může mít libovolný počet vstupních, výstupních a vstupně-výstupních parametrů. Při použití funkčního bloku se nejprve deklaruje jeho instance jako lokální proměnná. Vnitřní stav instance funkčního bloku zůstává zachován po dobu existence proměnné, kterou je tato instance reprezentována.
PROGRAM	Programový blok. Jedná se o obdobu bloku typu FUNCTION_BLOCK s tím rozdílem, že existuje pouze jedna instance tohoto bloku, která je platná po celou dobu běhu programu. V programu není třeba deklarovat proměnnou, kterou je reprezentována používaná instance.
CONFIGURATION	Konfigurační blok. Hlavní blok definující globální proměnné a obsahující definici spouštění jednotlivých bloků typu program. Tento blok nelze využít pro zápis programového kódu v žádném z programovacích jazyků.

Tabulka 1.2: Programové bloky zvolené pro implementaci.

Hlavní nevýhodou řešení je velká náročnost na interpret z hlediska jeho stability a odolnosti vůči chybám v PLC programu. Interpret by totiž v tomto případě musel fungovat v rámci kernel-space RT-Linuxu.

2. **Překladač PLC kódu** - výhododou tohoto řešení je zejména možnost vytvořit pouze preprocesor do jiného jazyka a pro vlastní generování binární podoby PLC programu využít tento programovací jazyk.

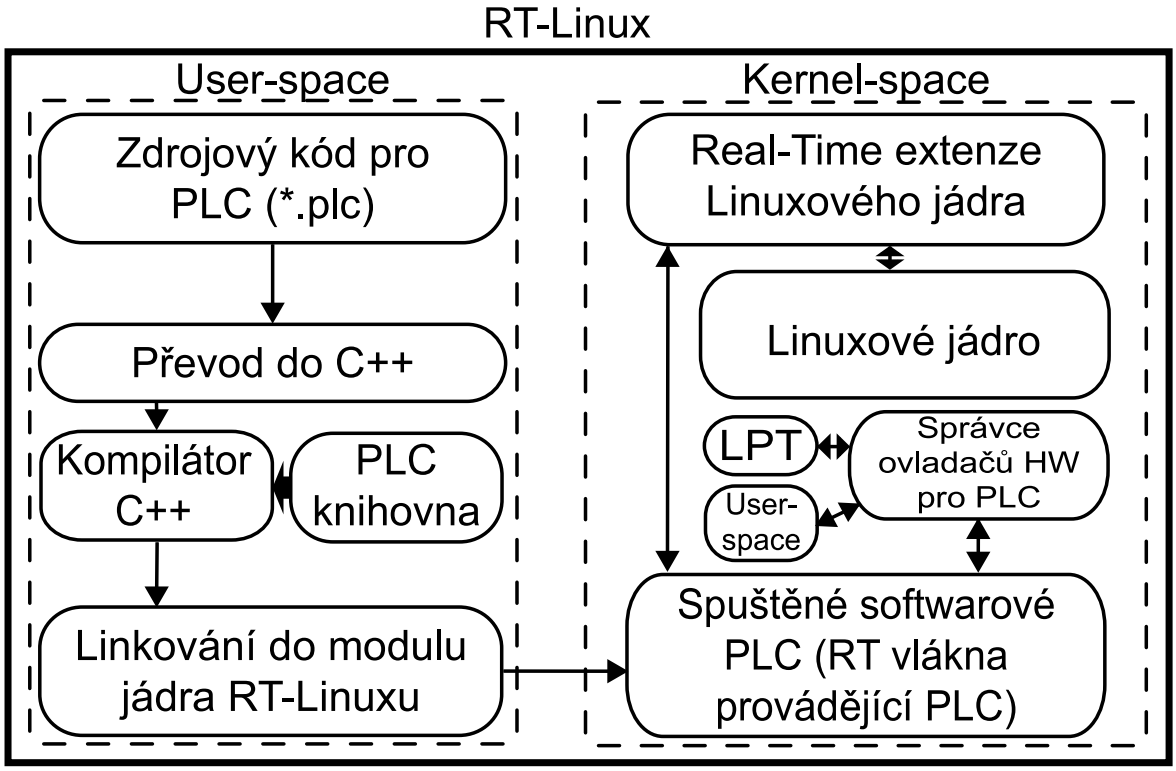
   Nevýhodou tohoto řešení je nutnost po každé změně PLC programu tento program znovu kompilovat.

Pro implementaci softwarového PLC zvolíme variantu č.2. Důvodem je možnost

využít již existující kompilátor jazyka C nebo C++, který je standardně využíván pro programování částí Linuxového a RT-Linuxového jádra.

Po analýze [5] je zřejmé, že způsob zápisu programu pro PLC lze nejlépe reprezentovat kódem v jazyce C++ (podobnost funkčních bloků s objekty, možnost definovat velké množství funkcí pomocí přetížených operátorů v podpůrné knihovně atp.).

Zvolené řešení odpovídá funkčnímu diagramu na obr.1.1. Cílem diplomové práce tedy bude vytvořit konvertor programu pro PLC do jazyka C++. Vygenerovaný kód bude následně přeložen do modulu jádra běžícího v RT-Linuxu. Pro zjednodušení konverze bude ještě třeba vytvořit podpůrnou knihovnu použitelnou v oblasti jádra, která bude využita při překladu C++ kódu.



Obrázek 1.1: Funkční diagram softwarového PLC

Z obrázku je patrné, že softwarové PLC bude rozděleno na dvě hlavní části, přičemž překlad programu bude spouštěn v user-space RT-Linuxu a vlastní softwarové PLC poběží jako real-time vlákna RT-Linuxu v rámci kernel-space. Krátký rozbor jednotlivých vlastností je uveden v částech 1.1 a 1.2.

Bližší informace o RT-Linuxu a některých použitých termínech z předchozí části lze nalézt v kapitole 2.1, případně v manuálových stránkách RT-Linuxu.

1.1 Funkční část pro user-space

User-space, nebo-li kontext ve kterém běží uživatelské programy bude využit pro běh preprocesoru (konvertoru) zdrojového kódu. Tento konvertor by měl splňovat následující požadavky:

- Umožnit předzpracování kódu ve zvoleném jazyce pro PLC do jazyka C++ v takovém tvaru, aby bylo možné tento kód zkompilovat do modulu běžícího v RT-Linuxu.
- Možnost tvorby a následného využití knihoven funkcí a funkčních bloků pro opakované využití.

1.2 Funkční část pro kernel-space

Kernel-space, neboli kontext jádra operačního systému RT-Linux bude využit pro vlastní běh vytvořeného softwarového PLC. Vygenerovaný kód v C++ bude zkompilován jako modul jádra RT-Linuxu. Pro kompilaci bude třeba vytvořit podpůrnou knihovnu, která bude mít následující vlastnosti:

- Řešení emulace aktuální hodnoty zásobníku (tzv. current result), kde je uložena hodnota poslední provedené operace.
- Emulace adresování přístupu do paměti ve stylu PLC včetně přístupu na externí zařízení dle adresy.
- Možnost deklarace proměnných ve stylu PLC, tj. včetně případné specifikace absolutní adresy, na které se proměnná nachází.
- Možnost deklarovat parametry funkcí, funkčních bloků a programových bloků tak, aby bylo možné je přiřazovat dle syntaxe PLC.
- Možnost modulárně vytvářet ovladače zařízení a tyto ovladače následně registrovat na konkrétní adresy vstupů/výstupů softwarového PLC.

Kapitola 2

RT-Linux

Tato kapitola uvádí v části 2.1 základní informace týkající se RT-Linuxové modifikace jádra a v části 2.2 diskutuje možnosti a omezení využití jazyka C++ v rámci RT-Linuxu.

2.1 Princip činnosti RT-Linuxu

V této části bude v krátkosti popsán princip činnosti RT-Linuxu. Informace zde uvedené jsou získány zejména z [9], případně z [1].

2.1.1 Definice hard real-time systému

Tzv. hard real-time systém je systém, ve kterém je zaručeno časově přesné spouštění jednotlivých úloh. Každá úloha má definovánu svoji prioritu, se kterou je spouštěna a periodu spouštění.

Perioda spouštění je v podstatě tzv. dead-line, tj. nejpozdější čas, do kterého musí být daná real-time úloha dokončena, aby mohla být opět spuštěna.

Priorita úlohy definuje "důležitost" úlohy z hlediska plánovače procesů.

2.1.2 Popis RT-Linuxu

RT-Linux je hard real-time operační systém určený pro řízení robotů a technologických procesů, časově náročná měření a jiné úlohy náročné na čas.

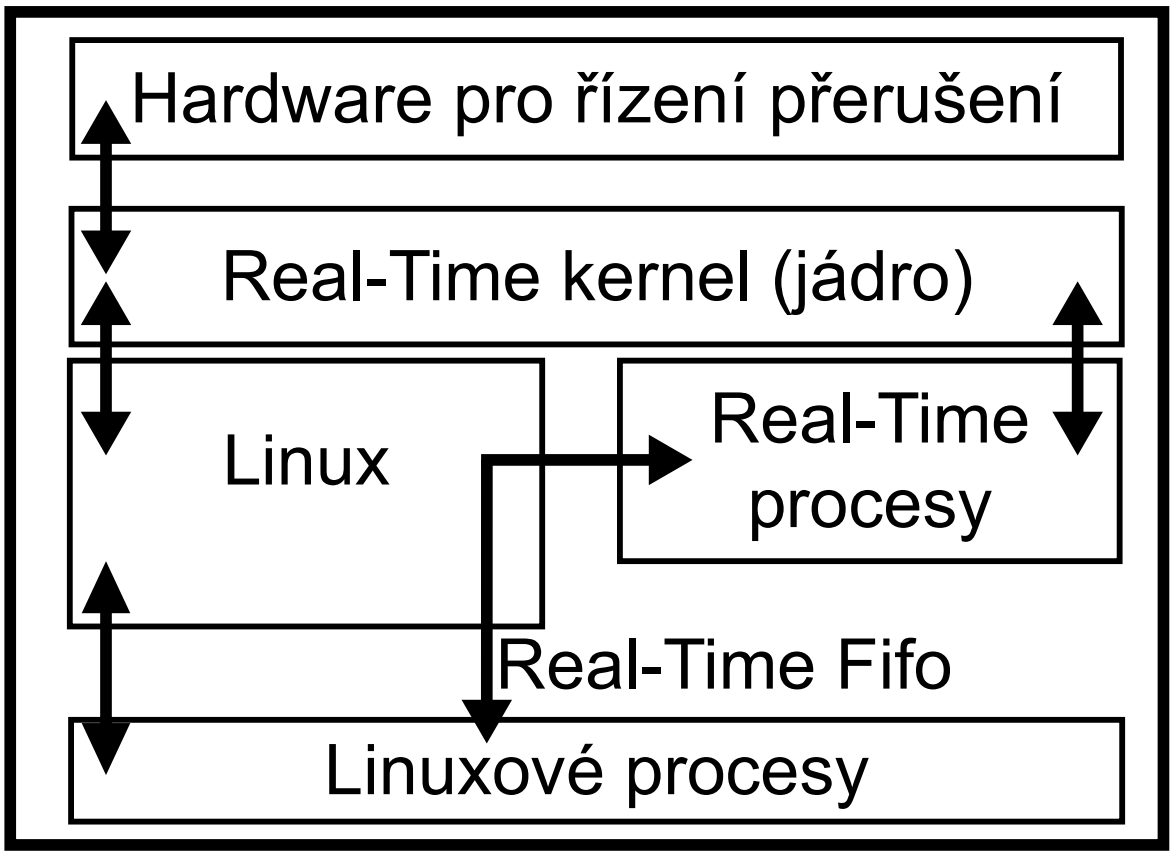
Dle náročnosti vykonávané úlohy a množství spouštěných vláken lze dosáhnout periody spouštění v řádu mikrosekund.

Výhodou RT-Linuxu je, že rozšiřuje možnosti použití klasického unixového systému o hard real-time vlastnosti při zachování původní funkčnosti systému.

2.1.3 Princip činnosti

Vlastní systém je implementován jednak jako patch[1] klasického Linuxového jádra, který zajišťuje obsluhu přerušení před vlastním Linuxovým kernelem. Druhá část jsou dynamicky nahrávané moduly, které po svém zavedení zajišťují vlastní hard real-time funkcionalitu (plánování procesů, poskytnutí synchronizačních primitiv, možnosti komunikace s user-space částí linuxu atd.).

Jak vlastní RT-Linuxový systém funguje je dobře patrné z obr.2.1.



Obrázek 2.1: Blokové schema funcionality RT-Linuxové extenze.

Celý systém funguje následovně:

- Po zavedení RT-Linuxových modulů jsou všechna přerušení odchytávána RT-Linuxem.
- Pokud je vyvoláno přerušení, je zpracováno RT-Linuxem, v případě, že je přerušení využíváno některou real-time úlohou, je vyvolán handler přerušení této úlohy.

[1]Patchem je obvykle nazývána soubor, obsahující úpravu či oprava zdrojového kódu nějakého programu. Aplikací patche na zdrojové soubory původního programu je pak získána nová, upravená verze.

- Pokud existují naplánované real-time úlohy čekající na volný čas procesoru, pak jsou spuštěny (v závislosti na prioritě).
- Pokud není ve frontě žádná real-time úloha a procesor je volný, je předáno řízení původnímu linuxovému jádru.
- Původní linuxové jádro využívá přerušení, která jsou emulována RT-Linuxem[2].

2.1.4 Využití matematického koprocesoru

V případě, že real-time vlákno chce využívat matematický koprocesor (dále jen FPU), je nutno tuto skutečnost oznámit plánovači procesů[3]. Plánovač následně zajistí při přepínání takového vlákna uložení stavu registrů FPU a jejich případné obnovení ve chvíli, kdy vlákno pokračuje ve svém běhu.

Pokud plánovač procesů není informován, že vlákno využívá FPU může docházet k navracení chybných výsledků výpočtu. K tomu může dojít z následujících důvodů:

1. FPU v jeden okamžik využívá program z user-space a zároveň real-time vlákno.
2. FPU v jeden okamžik využívají alespoň dvě real-time vlákna.
3. FPU je v jeden okamžik požadováno několika real-time vlákny i programy z user-space.

2.2 RT-Linux a C++

V RT-Linuxu je možno využívat programovacího jazyka C++ pro tvorbu real-time vláken. Použití však má jistá omezení, která vyplývají již přímo z principu, jakým systém funguje.

Následující části se postupně zabývají jednotlivými konstrukcemi jazyka C++ a jejich použitím v RT-Linuxu. Část 2.2.1 diskutuje použití dynamické alokace paměti, část 2.2.2 krátce diskutuje použití exceptions[4] a část 2.2.3 diskutuje použití templates(šablon).

2.2.1 Dynamická alokace paměti

Problém použití dynamické alokace paměti lze rozdělit na část, která se týká alokování paměti během inicializace real-time modulu a problematiku dynmické alokace paměti během vlastního běhu real-time procesu (tj. alokace paměti přímo z běžícího RT vlákna).

[2] V podstatě lze činnost původního linuxového jádra přirovnat ke zvláštnímu real-time vláknu RT-Linuxu, běžícímu s nejnižší real-time prioritou.

[3] To je možné zajistit buď informací při zakládání real-time vlákna nebo nastavením stejného parametru za běhu real-time vlákna.

[4] Tj. vyjímek pro ošetření chybových stavů při běhu programu

Oba tyto problémy lze řešit za pomoci přetížení C++ operátoru new a delete, případně operátorů new[] a delete[] pro alokace a uvolňování paměti dynamických polí.

V případě, že potřebujeme dynamicky alokovat paměť během inicializace, případně deinicializace modulu, stačí přetížit standardní C++ operátory uvedené výše na volání linuxového jádra kalloc(), případně kfree(). Toto přetížení je již standardně provedeno v souboru rtl_cpp.h, který je součástí distribuce RT-Linux.

Za předpokladu, že je třeba dynamicky alokovat paměť za běhu real-time vlákna, je třeba si vytvořit vlastní správu dynamické paměti. Důvodem je zejména možnost vzniku kolize při volání standardních systémových volání.

Vlastní dynamická správa paměti by měla fungovat následujícím způsobem:

1. V okamžiku inicializace RT modulu naalokuje paměť z oblasti jádra pomocí volání kmalloc(). Velikost takto alokované paměti by měla odpovídat alespoň maximální potřebné velikosti paměti, kterou budeme chtít dynamicky alokovat. Bloky této paměti musí být alokován ve fyzické paměti počítače.

2. Přetíží operátory new, new[], delete, delete[] tak, aby volaly funkce zajišťující alokaci a dealokaci paměti v rámci paměťového bloku naalokovaného během zavádění RT modulu (viz. krok 1).

3. Během deaktivace modulu provede uvolnění paměťového bloku, který byl získán z paměťové oblasti linuxového jádra v okamžiku zavedení modulu do paměti.

2.2.2 Ošetření chyb - exceptions

Vyjímky, nebo také exceptions, nelze bohužel v paměťovém prostoru linuxového jádra a tedy ani v oblasti, která je využívána RT rozšířením využít. Hlavním důvodem je, že linker optimalizuje kód pro volání vyjímek. Proto by během linkování modulu jádra musely být známy všechny moduly, které se budou v paměti nacházet.

Dalším důvodem je, že linuxové jádro je psána čistě v jazyce C a již během jeho kompilace jsou vyjímky zakázany pomocí parametru překladače.

2.2.3 Použití C++ templates

Templates jako vlastnost jazyka C++ lze bez obtíží využít, neboť během kompilace je takový kód nahrazen skutečným datovým typem včetně všech kontrol během kompilace.

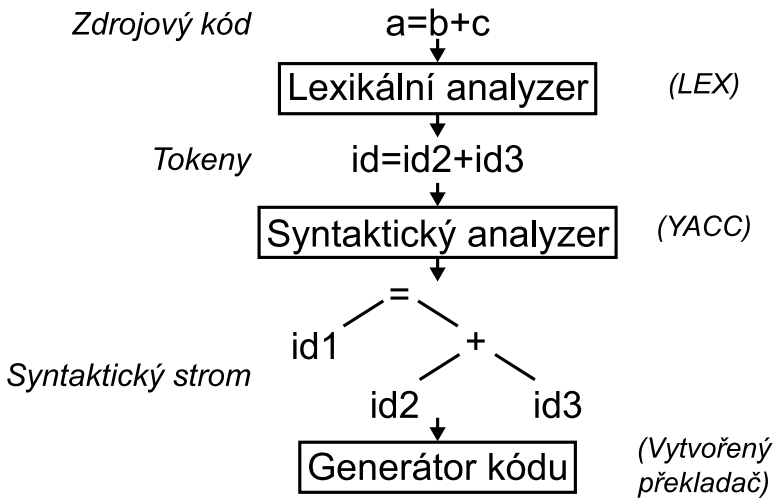
Bohužel nelze jednoduše využít tříd implementovaných v knihovně STL[5]. Důvodem jsou dynamické alokace paměti, které jsou v těchto třídách využívany. Omezení z toho plynoucí jsou uvedena v části 2.2.1. Určitou možností, jak této knihovny využít by zřejmě byla vlastní rekompilace ze zdrojových kódů se správně přetíženými operátory pro správu paměti.

[5]Standard Template Library.

Kapitola 3

Lexikální analýza zdrojového kódu

V této části se budeme zabývat vlastním zpracováním zdrojového kódu v jazyce IL, tedy vytvořením kompilátoru (překladače). Dle [7] lze překladač reprezentovat pomocí konečného stavového automatu. V UNIXových systémech jsou k dispozici programy LEX a YACC, které umožňují takový stavový automat pro zpracování zdrojového kódu vytvořit na základě zápisu pravidel v příslušném tvaru. Část 3.1 této kapitoly se zabývá popisem programu LEX, který se využívá pro vytvoření lexikálního analyzeru (viz. obr.3.1). Část 3.2 se zabývá popisem jazyka pomocí pravidel zadaných programu YACC (na obr.3.1 to odpovídá syntaktickému analyzeru). Část 3.3 popisuje princip, jakým oba výše uvedené programy spolupracují při tvorbě překladače.



Obrázek 3.1: Kompilační sekvence.

3.1 Lexikální analýza zdrojového kódu

Pro lexikální analýzu zdrojového kódu bude využit program LEX. Informace, které jsou uvedeny v této části jsou čerpány z [8] a [4]. Program LEX slouží k vytvoření parseru zdrojového kódu a převodu klíčových slov na tzv. tokeny. Zároveň umožňuje vytvořit stavový automat definující možnou následnost různých klíčových slov ve zdrojovém kódu.

Vytvořený stavový automat může pracovat dvěma způsoby:

1. Stavový automat při přechodu do dalšího stavu předchozí stav zapomene.
2. Stavový automat ukládá stavy na zásobník. Změna stavu je možná dvěma směry:
   - Přechod do nového stavu, původní stav je uložen na zásobník.
   - Z aktuálního stavu se přechází do stavu uloženého na zásobníku, aktuální stav je zapomenut.

3.1.1 Struktura zdrojového kódu pro LEX

Vstupní soubor pro LEX je rozdělen do třech částí tak, jak je patrno z ukázky 3.1. Jednotlivé části jsou od sebe odděleny pomocí značek %%.

```
... definice maker a vlastností analyzátoru ...
%%
... definice pravidel ...
%%
... funkce v jazyce C/C++ ...
```

Ukázka 3.1: Struktura vstupního souboru pro LEX.

První část obsahuje makra pro zjednodušení definičních regulárních výrazů pro vyhledávání klíčových slov, parametry parseru a případně zdrojový kód v jazyce C/C++. V tab.3.1 je uveden přehled klíčových slov, které mohou být v první části obsaženy.

Druhá část vstupního souboru obsahuje pravidla ve tvaru regulárních výrazů pro zpracování zdrojového kódu a definici akcí v jazyce C/C++, které pravidlu odpovídají. Pravidlo může být aktivní pouze v definovaných stavech, akce může obsahovat přechod do stavu nového nebo do stavu uloženého na zásobníku. Struktura pravidel je patrná z ukázky 3.2.

První část pravidla je nepovinná a definuje stavy, ve kterých je pravidlo platné, případně stavy, ve kterých platné není. Zde záleží na tom, jak je definován stav v první části vstupního souboru.

Definice	**Popis**
Název reg. výraz	Předdefinované regulární výrazy, které lze použít dále pro zjednodušení definičních pravidel pro vyhledávání tokenů ve zdrojovém kódu.
%s	Definice stavů, ve kterých se může nacházet vygenerovaný stavový automat. Na jednom řádku se může nacházet deklarace více stavů jejichž jména jsou oddělena mezerou.
%{ ... %}	Ohraničení zdrojového kódu v jazyce C/C++, který bude vložen tak, jak je napsán, na začátek vygenerovaného zdrojového kódu parseru.
%option ...	Parametry vygenerovaného parseru, pokud obsahuje klíčové slovo stack, jsou stavy parseru ukládány na zásobník.

Tabulka 3.1: Přehled klíčových slov použitelných v první části definičního souboru pro LEX.

```
[<stav1[,stav2[,...]]>]reg.výraz {
akce v jazyce C;
}
```

Ukázka 3.2: Struktura pravidel pro analýzu zdrojového kódu.

Druhá část je regulární výraz, který definuje, jak dané klíčové slovo, konstanta či jiná součást zdrojového kódu je zapsána. Základní struktura regulárních výrazů je popsána v tab. 3.2, podrobnější informace o regulárních výrazech lze získat např. v [6]. V případě, že zdrojový text obsahuje část, která odpovídá příslušnému regulárnímu výrazu, je tento text uložen do proměnné YYTEXT a lze ho použít v části pro zápis příslušné akce, která se vyvolá pro daný výraz.

V části pro zápis akce lze zapsat libovolnou akci nebo nepsat nic, pokud se má vše, co odpovídá danému regulárnímu výrazu ignorovat. V případě, že chceme spolupracovat s programem YACC, může akce vypadat např. jako v ukázce 3.3.

Ukázka 3.4 uvádí celý vstupní soubor pro LEX obsahující konkrétní pravidlo, včetně akce v jazyce C. Pravidlo detekuje token jako celočíselnou konstantu ve zdrojovém kódu, pokud se stavový automat nachází ve stavu *any_value*.

Regulární výraz	**Popis**
LD	Regulární výraz je slovo LD.
^LD	Regulární výraz je slovo LD na začátku řádku.
LD$	Regulární výraz je slovo LD na konci řádku.
LD/([ ]+)	Regulární výraz je slovo LD následované libovolným počtem mezer nebo tabulátorů, část za znakem / není uložena v proměnné YYTEXT a je ponechána v bufferu pro další zpracování.
([0-9]+)	Regulární výraz je číslo skládající se z libovolného počtu číslic.
{SPACE}	Regulární výraz odpovídá definici z prvního bloku zdrojového souboru s názvem SPACE.

Tabulka 3.2: Struktura regulárních výrazů v definici pravidel pro LEX.

```
// pokud se právě nacházím ve stavu var_declaration
if (YY_START == var_declaration)
{
   // vyzvedni aktuální stav ze zásobníku (dojde k přesunu
   // do stavu předchozího)
   yy_pop_state();
   // a ulož na zásobník stav var_data_type
   yy_push_state(var_data_type);
   // ulož hodnotu nalezeného regulárního výrazu do proměnné
   // pro YACC
   yylval.string = strdup(yytext);
   // a vrať informaci, že jsi nalezl datový typ
   // proměnné
   return TOK_VAR_DATATYPE;
}
// nejsem v bloku deklarací proměnných,
// ale očekávám deklaraci datového typu
// funkce
yylval.string = strdup(yytext);
return TOK_FUNCTION_DATATYPE;
}
```

Ukázka 3.3: Zápis akce pro nalezený regulární výraz ze zdrojového kódu.

```
%{
#include <stdio.h>
// odkaz na počítadlo řádků
extern int iLineCount;
%}
// použij zásobník při změně stavu
%option stack
// definice povolených stavů
%s var_init
%%
// pokud jsi ve stavu, kdy je očekávána inicializace
// proměnné, povol detekci celého čísla.
<var_init>[-+]?([0-9]+) {
   yylval.number = atoi(yytext); // hodnotu vrať jako číslo
   return VAL_INTEGER;
}
// pokud jde o konec řádku, zvyš počítadlo a jinak
// nic nedělej
\n   ++iLineCount;
%%
```

Ukázka 3.4: Ukázka definice pravidla pro program LEX.

3.2 Popis gramatiky jazyka

Pro popis gramatiky jazyka lze využít systém YACC, který umožňuje ve spolupráci s programem LEX definovat velmi jednoduše pravidla , jak daný programovací jazyk vypadá, tj. jaké umožňuje vytvářet programátorské konstrukce a zápisy.

Informace zde uvedené jsou čerpány z [8] a [4].

3.2.1 Struktura zdrojového souboru pro YACC

Struktura zdrojového souboru pro systém YACC je opět rozdělena do třech částí tak, jak je patrno z obrázku 3.5. Jednotlivé části jsou od sebe odděleny pomocí značek %%.

První část obsahuje definici tokenů pro vyhledávání (z těchto definic jsou poté vygenerovány unikátní identifikátory jejichž symbolické názvy odpovídají názvům tokenů), prioritu operátorů apod. V tab.3.3 je uveden přehled základních příkazů, které mohou být v této části obsaženy.

Část pro definici pravidel obsahuje již přímo jednotlivá pravidla, která odpovídají konstrukcím, které lze v jazyce vytvořit. Každé takové pravidlo má strukturu dle obr.3.6.

Význam jednotlivých prvků je následující:

| Definuje, že následující část pravidla je nepovinná.

```
... definice ...
%%
... definice pravidel pro gramatiku jazyka ...
%%
... funkce v jazyce C ...
```

Ukázka 3.5: Struktura vstupního souboru pro program YACC.

Definice	**Popis**
%token jméno	Definice tokenu s názvem jméno
%{...%}	Blok obsahující kód v jazyce C/C++, který bude po zpracování vložen na začátek vygenerovaného kódu.
%union{int iData;char *pData;}	Návratová hodnota yylval je typu union a je určena některým datovým typem uvedeným v unionu.
%token <iData> TOK_INTEGER	Definuje token s datovým typem iData (dle definice v unionu).
%left ’,’	Definice priority operátoru.
%right ’,’	Definice priority operátoru.

Tabulka 3.3: Základní příkazy použitelné v bloku definic vstupního souboru pro YACC.

```
nazev:
     [|]
     nazev2
     [{
     ... programový kód v jazyce C/C++ ...
     }]
     [[|] nazev3]
     ;
```

Ukázka 3.6: Struktura pravidel pro definici jazyka v programu YACC.

nazev Název pravidla (bez diakritiky) oddělený dvojtečkou od jeho definice.

nazev2 Název tokenu definovaného v části definic na začástku vstupního souboru pro

YACC nebo název jiného pravidla. Pro vytváření rekurzivních pravidel může být stejný i jako nazev.

{ ... } Ohraničuje blok obsahující kód v jazyce C/C++, který bude vykonán v případě, že pravidlo bude obsaženo ve vstupním zdrojovém souboru.

; Označení konce bloku, který definuje pravidlo nazev.

Programový kód v jazyce C/C++ definuje akce, které budou při aplikaci pravidla vyvolány. V případě, že nalezený token má definován datový typ, lze k hodnotě předané z programu LEX přistupovat pomocí konstrukce $x, kde x je číslo definující pořadí tokenu v definici pravidla. V případě pravidla z obr.3.6 by pro token nazev2 s definovaným typem odpovídala hodnota $1. Do číslování pořadí se zahrnují i programové bloky uzavřené mezi {}.

Na obr.3.7 je uvedena krátká komentovaná ukázka definice několika jednoduchých pravidel.

3.3 Princip spolupráce programů LEX a YACC

Tato část popisuje spolupráci programů LEX a YACC při vytváření překladače navrženého jazyka. Princip činnosti je vysvětlen na obr.3.2 pomocí vytvářeného softwarového PLC (výsledkem bude program plc2cpp, který bude převádět kód ve zvoleném jazyce pro PLC do jazyka C/C++).

Princip spolupráce obou programů lze popsat následovně:

1. Nejprve vytvoříme lexikální analyzer pomocí programu LEX, který vyhledá ve zdrojovém textu tzv. tokeny (tj. definovaná klíčová slova).

2. Následně definujeme pomocí programu YACC syntaxi navrženého jazyka (tj. možné kombinace vyhledaných tokenů).

3. Zkompilujeme vstupní soubor pro program YACC. Výsledkem je hlavičkový soubor y.tab.h, který definuje pro jednotlivé názvy tokenů číselné hodnoty (makra) v jazyce C/C++ a soubor y.tab.c, který obsahuje vlastní syntaktický analyzátor jazyka.

4. Zkompilujeme vstupní soubor pro program LEX. Ten ve vygenerovaném kódu využívá soubor y.tab.h vygenerovaný v předchozím kroku. Výsledkem je soubor lex.yy.c, který obsahuje lexikální analyzer v jazyce C/C++.

5. Zkompilováním souborů vygenerovaných v předchozích kroků spolu s dalšími podpůrnými částmi získáme požadovaný kompilátor. V našem případě plc2cpp.

6. Zkompilovaný program z předchozích kroků je již vytvářeným kompilátorem, který lze využít ke kompilaci zdrojového kódu ve vytvářeném jazyce. V našem případě tedy zdrojový kód v implementovaným programovacím jazykem pro PLC.

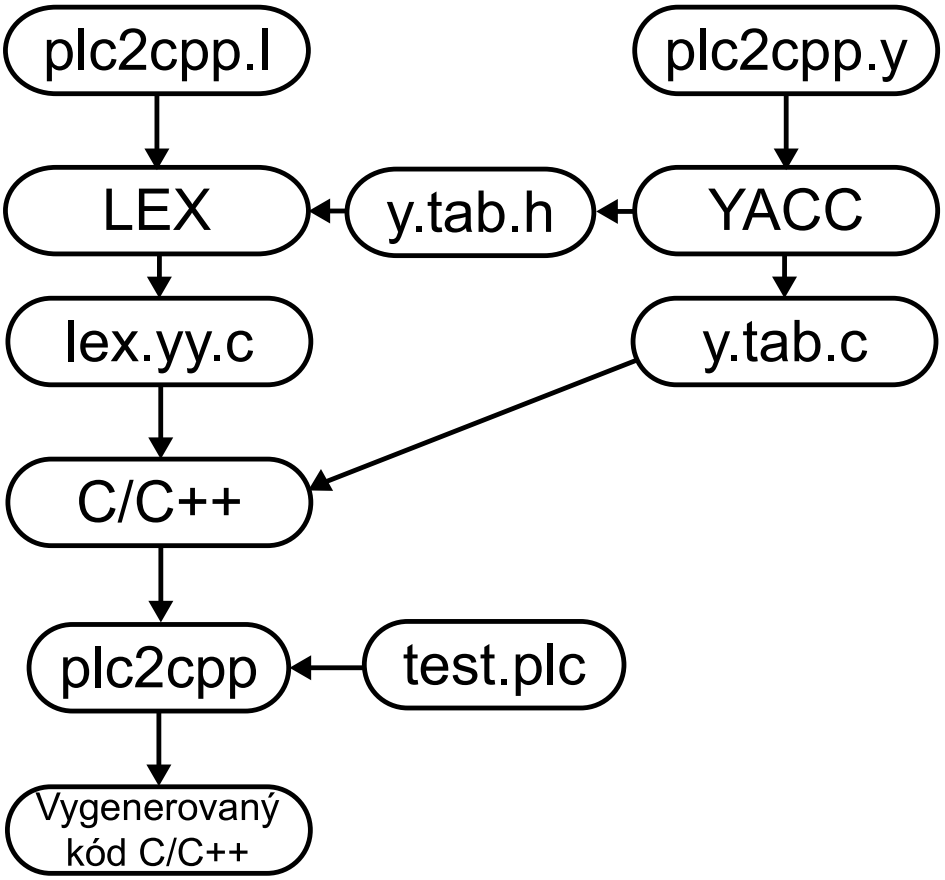
```
// Definuj token pro přiřazení hodnoty
%token TOK_SET
// Přiřazovaná hodnota je buď celé číslo
// nebo nulou ukončený řetězec
%union
{
   int number;
   char *string;
}
// Definuj token s určeným datovým typem pro návrat
// celočíselné hodnoty
%token <number> VAL_INT
// Definuj token s určeným datovým typem pro návrat
// nulou ukončeného řetězce
%token <string> VAL_STR

%%
// Definuj základní pravidlo pro detekci jednotlivých
// bloků ve vstupním souboru.
blocks:
   // Žádný z následujících bloků v pravidle není povinný
   |
   // Po nalezení bloku pro přiřazení celého čísla může následovat
   // další vstupní blok (rekurze))
   set_int blocks
   |
   // Po prvním bloku přiřazujícím řetězec již nenásleduje
   // žádný další blok
   set_string
   ;

set_int:
   TOK_SET VAL_INT
   {
      printf("Číselná hodnota: %d", $1);
   }
   ;

set_string:
   TOK_SET VAL_STR
   {
      printf("Řetězcová hodnota: %s", $1);
   }
   ;
%%
```

Ukázka 3.7: Ukázka definice pravidla ve zdrojovém kódu pro YACC.



Obrázek 3.2: Princip tvorby a funkce překladače jazyka definovaného pomocí programů LEX a YACC.

Zde uvedené vlastnosti a konstrukce obsahují pouze malou část možností, které programy YACC a LEX nabízejí. V podstatě shrnuje vlastnosti využité během konstrukce softwarového PLC. Podrobnější informace lze nalézt v uvedené literatuře.

Kapitola 4

Zpracování proměnných PLC programu

Tato část se zabývá zpracováním proměnných a parametrů funkcí, funkčních bloků a programových bloků. Část 4.1 nastiňuje princip zpracování proměnných v programu, část 4.2 se zabývá deklarací jednotlivých proměnných, část 4.3 rozebírá jednotlivé možné bloky proměnných a část 4.4 uvádí základní principy týkající se implementace a zpracování proměnných v softwarovém PLC.

4.1 Princip zpracování proměnných

Proměnné v PLC programu se zapisují v rámci bloků proměnných. Každý blok proměnných má jiný význam a ve svém těle obsahuje deklaraci jedné nebo více proměnných tak, jak je patrno z ukázky 4.1. Významem jednotlivých bloků proměnných se zabývají další části této kapitoly.

Princip zpracování proměnných v rámci PLC programu probíhá následujícím způsobem:

1. Detekuj blok deklarující proměnné (viz. část 4.2).
2. Najdi deklaraci proměnné a ukládej její parametry do kontejneru CContainerVariable:
   (a) Separuj název proměnné, převeď ho na malá písmena a vlož do kontejneru.
   (b) Detekuj, zda deklarace proměnné obsahuje specifikaci adresy, pokud ano, ulož adresu v kontejneru.
   (c) Identifikuj datový typ proměnné, pokud se jedná o funkční blok, sděl kontejneru požadavek na vložení hlavičkového souboru s příslušným funkčním blokem.
   (d) Pokud se jedná o standardní datový typ, detekuj případnou deklaraci inicializační hodnoty. V případě, že je proměnná inicializována, ulož inicializační hodnotu do kontejneru.

3. Existuje-li deklarace další proměnné, opakuj vše od bodu 2.

4. Detekuj konec bloku proměnných, uzavři kontejner proměnných a přidej ho do kontejneru obsahujícího jednotlivé kontejnery proměnných platných v aktuálním programovém bloku.

4.2 Deklarace proměnných

Jednotlivé proměnné se deklarují v několika typech bloků s různým významem. Jednotlivé druhy bloků proměnných jsou povoleny v různých kontextech PLC kódu. Proměnné se zapisují způsobem uvedeným v ukázce 4.1.

```
...
Začátek bloku proměnných
...
typ [AT adresa]: Název proměnné [:= hodnota];
...
Konec bloku proměnných
...
```

Ukázka 4.1: Deklarace proměnné

Každá proměnná musí obsahovat datový typ a název. Pokud je datový typ proměnné jeden ze základních datových typů uvedených v tab.4.1[1], může deklarace proměnné obsahovat ještě klíčové slovo AT následované adresou ve tvaru uvedeném v tab.4.3. Dále v tomto případě může deklarace obsahovat inicializační hodnotu proměnné oddělenou pomocí :=.

Jako datový typ může být použit také název funkčního bloku, zde je tedy třeba také zajistit, aby se vygeneroval řádek pro provedení include příslušného hlavičkového souboru obsahujícího deklaraci funkčního bloku. Celá deklarace je ukončena pomocí ;.

Ukázka 4.2 uvádí příklad deklarace několika proměnných tak,aby byl patrný konkrétní zápis proměnné v programu. Pro větší názornost jsou zapsány v rámci bloku lokálních proměnných.

4.2.1 Přímé adresy

V tab.4.2 je uvedeno několik ukázek přímých adres pro PLC. Adresa se vždy skládá z prefixu, který definuje typ adresy následovaný prefixem, definujícím velikost paměťového bloku, kterému adresa odpovídá.

[1] V tabulce jsou uvedeny pouze datové typy podporované ve vytvářeném softwarovém PLC.

Datový typ	Velikost	Poznámka
BOOL	1	
SINT	8	
INT	16	
DINT	32	
USINT	8	
UINT	16	
UDINT	32	
REAL		
TIME	64	Dostupný pouze jako konstanta určující periodu spouštění programu.
BYTE	8	
WORD	16	
DWORD	32	

Tabulka 4.1: Přehled základních datových typů použitelných v PLC.

```
VAR
   current: DINT;
   otacky: DINT := 0;
   rps AT %QB10: DINT := 0;
   sp AT %IB10: DINT;
   forward AT %Q0.0: BOOL := FALSE;
   backward AT %Q0.1: BOOL := FALSE;
END_VAR
```

Ukázka 4.2: Ukázka deklarace jednotlivých proměnných.

Adresa	Poznámka
%QX35, %Q35	Výstupní bit 37
%MX3.5	5 bit 3 bytu ve vnitřní paměti
%IW23	Vstupní slovo 23

Tabulka 4.2: Ukázky zápisu adres pro PLC.

Přehled jednotlivých prefixů pro adresaci paměťových míst PLC je uveden v tab.4.3.

Adresa	**Poznámka**
Typ	
Q	Adresa výstupu
I	Adresa vstupu
M	Adresa ve vnitřní paměti
Velikost	
X	Adresa specifikuje jeden bit
Nic	Adresa specifikuje jeden bit
B	Adresa specifikuje jeden byte (8bitů)
W	Adresa specifikuje jedno slovo (16bitů)
D	Adresa specifikuje dvojité slovo (32bitů)
L	Adresa specifikuje dlouhé slovo (64bitů)

Tabulka 4.3: Způsob zápisu přímých adres pro PLC.

4.3 Přehled bloků pro deklaraci proměnných

Následující část se zabývá významem a způsobem zápisu jednotlivých typů bloků proměnných. Informce zde uvedené jsou čerpány z [5].

Norma [5] definuje následující typy proměnných:

- **Globální proměnné** - používají se pro zápis proměnných dostupných v celém PLC programu.
- **Lokální proměnné** - používají se pro deklaraci lokálních proměnných v rámci funkce, funkčního bloku nebo bloku program.
- **Externí proměnné** - používají se k definici odkazu na globální proměnné v rámci bloků funkce, funkční blok nebo blok program.
- **Vstupní proměnné** - používají se pro deklaraci vstupních parametrů funkcí, funkčních bloků nebo bloků typu program.
- **Výstupní proměnné** - používají se pro deklaraci výstupních parametrů funkčních bloků a bloků typu program.
- **Vstupně/výstupní proměnné** - zastupují úlohu parametrů předávaných referencí v rámci bloků program a funkčních bloků.

4.3.1 Globální proměnné

Globální proměnné je možné deklarovat v rámci bloku CONFIGURATION, případně v rámci podbloku konfigurace RESOURCE. Globální proměnné jsou následně dostupné v celém PLC programu, přičemž z bloků typu FUNCTION, PROGRAM a

FUNCTION_BLOCK je třeba se na ně odkázat pomocí deklarace VAR_EXTERNAL (viz.4.3.2). Způsob deklarace bloku globálních proměnných je patrný z ukázky 4.3.

```
... nadřazený blok ...
   VAR_GLOBAL
      ...
      Deklarace jednotlivých proměnných;
      ...
   END_VAR
... pokračování nadřazeného bloku ...
```

Ukázka 4.3: Deklarace globálních proměnných v programu.

4.3.2 Externí proměnné

Blok externích proměnných se využívá k odkazu na globální proměnnou, kterou chceme využít v rámci bloků PROGRAM, FUNCTION a FUNCTION_BLOCK. V tomto bloku lze použít pouze deklaraci proměnné bez specifikace adresy, kde se nachází, ani deklarovat inicializační hodnotu - tu je možno specifikovat v rámci deklarace proměnné v bloku VAR_GLOBAL. Způsob deklarace bloku globálních proměnných je patrný z ukázky 4.4.

```
... nadřazený blok ...
   VAR_EXTERNAL
      ...
      Deklarace jednotlivých proměnných;
      ...
   END_VAR
... pokračování nadřazeného bloku ...
```

Ukázka 4.4: Deklarace externích proměnných v programu.

4.3.3 Lokální proměnné

Lokální proměnné lze použít pouze v rámci programových bloků PROGRAM, FUNCTION a FUNCTION_BLOCK. Proměnné deklarované jako lokální mají platnost pouze v programovém bloku, kde jsou deklarovány.

V případě funkčních bloků a bloků typu program si lokální proměnné uchovávají svoji hodnotu z předchozího volání po celou dobu existence instance funkčního bloku.

Způsob deklarace bloku lokálních proměnných je patrný z ukázky 4.5.

```
... nadřazený blok ...
   VAR
      ...
      Deklarace jednotlivých proměnných;
      ...
   END_VAR
... pokračování nadřazeného bloku ...
```

Ukázka 4.5: Deklarace lokálních proměnných v programu.

4.3.4 Vstupní proměnné

Bloky vstupních proměnných lze využívat v rámci bloků PROGRAM, FUNCTION a FUNCTION_BLOCK. V jejich deklaraci nelze specifikovat adresu a využívají se jako vstupní parametry výše uvedených bloků. Způsob deklarace bloku vstupních proměnných je patrný z ukázky 4.6.

```
... nadřazený blok ...
   VAR_INPUT
      ...
      Deklarace jednotlivých proměnných;
      ...
   END_VAR
... pokračování nadřazeného bloku ...
```

Ukázka 4.6: Deklarace vstupních proměnných v programu.

4.3.5 Výstupní proměnné

Výstupní proměnné lze použít pouze v rámci bloků PROGRAM a FUNCTION_BLOCK. V jejich deklaraci nelze specifikovat adresu a využívají se pro předávání hodnot z aktuálního programového bloku do bloku nadřazeného. Způsob deklarace bloku výstupních proměnných je patrný z ukázky 4.7.

```
... nadřazený blok ...
   VAR_OUTPUT
      ...
      Deklarace jednotlivých proměnných;
      ...
   END_VAR
... pokračování nadřazeného bloku ...
```

Ukázka 4.7: Deklarace výstupních proměnných v programu.

4.3.6 Vstupně/výstupní proměnné

Bloky vstupně/výstupních proměnných lze použít pouze v rámci bloků PROGRAM a FUNCTION_BLOCK. V deklaraci jednotlivých proměnných nelze specifikovat adresu a zastupují úlohu parametrů předávaných referencí. Tyto parametry lze využít jak pro předání hodnoty do volaného bloku, tak pro získání této hodnoty po ukončení volání příslušného bloku. Způsob deklarace bloku vstupně/výstupních proměnných je patrný z ukázky 4.8.

```
... nadřazený blok ...
   VAR_IN_OUT
      ...
      Deklarace jednotlivých proměnných;
      ...
   END_VAR
... pokračování nadřazeného bloku ...
```

Ukázka 4.8: Deklarace výstupních proměnných v programu.

4.4 Implementace proměnných

Implementace proměnných musí splňovat všechny požadavky, které vyplývají z předchozí části:

1. Spolupráce se všemi datovými typy zvolenými pro implementaci.
2. Možnost specifikovat absolutní adresu, na které se proměnná nachází.

3. Možnost specifikovat inicializační hodnotu proměnné.

4. Pro vstupně/výstupní proměnné umožnit předat hodnotu proměnné do bloku, který provedl volání příslušného funkčního nebo programového bloku.

Cílem tedy bude vytvořit příslušné knihovní třídy, které umožní akce definované v předchozím přehledu a vyplývající z předchozí část. Přehled navržených tříd a požadavky na jejich funkcionalitu jsou uvedeny v tab.4.4.

Třída	**Funkcionalita**
CVariable	Template třída implementující vlastnosti proměnných pro všechny základní datové typy z tab.4.1.
CPropertyIn	Template třída zapouzdřující vlastnosti proměnných deklarovaných v rámci bloku VAR_INPUT.
CPropertyOut	Template třída zapouzdřující vlastnosti proměnných deklarovaných v rámci bloku VAR_OUTPUT.
CPropertyInOut	Template třída zapouzdřující vlastnosti proměnných deklarovaných v rámci bloku VAR_IN_OUT, včetně případného předání hodnoty proměnné do bloku, který provedl volání příslušného funkčního nebo programového bloku.

Tabulka 4.4: Přehled tříd pro implementaci proměnných v softwarovém PLC.

4.4.1 Parser zdrojového kódu

V rámci parseru zdrojového kódu pro PLC budou v generovaném C++ kódu využity objekty z tab.4.4. Pro vlastní zpracování kódu bude vytvořen kontejner CContainerVariable pro udržování informací o aktuální zpracovávané proměnné (informace o datovém typu, názvu, adrese, inicializační hodnotě atp.). Dále bude vytvořena template třída CContainerContainer, která bude sloužit ke sdružování kontejnerů patřících do jednoho bloku PLC programu.

4.4.2 Vygenerovaný zdrojový kód

Vygenerovaný zdrojový kód v C++ je rozdělen do následujících částí:

- V případě, že se nejedná o globální proměnné:

1. Povinná část deklarace vlastní proměnné v private nebo public části deklarace objektu, který reprezentuje daný blok z programu.
2. Nepovinná část v konstruktoru třídy zajišťující inicializaci proměnné. Generováno pouze v případě, že proměnná má definovánu inicializační hodnotu.
3. Pro výstupní a vstupně-výstupní proměnné je ještě generován kód resetující proměnnou pro další použití.

- V případě, že se jedná o globální proměnnou, je vygenerována deklarace proměnné typu CVariable s příslušným datovým typem a případnou inicializační hodnotou proměnné.

Po zpracování zdrojového kódu je pro lokální proměnnou vygenerován kód odpovídající ukázce 4.9. Ve vygenerovaném kódu je využita template třída CVariable.

```
// deklarace třídy,                          // konstruktor třídy
// hlavičkový soubor                         // counter AT %QB3: INT := 3;
// counter AT %QB3: INT := 3;                init :: init(void)
class init                                   {
{                                            #line 3 "test.plc"
...                                             m_counter = CVariable<INT> ((INT) 3, "QB3");
private:                                     }
#line 3 "test.plc"
   CVariable<INT> m_counter;
...
};
```

Ukázka 4.9: Ukázka kódu vygenerovaného pro lokální proměnné.

Kód generovaný pro parametry funkcí, funkčních bloků nebo programů je patrný z ukázky 4.10. Ve vygenerovaném kódu je využita jedna z template tříd CPropertyIn, CPropertyOut nebo CPropertyInOut dle typu parametru pro přístup k parametru z místa, odkud je provedeno volání a template třída CVariable, která je přiřazena k objektu a je využívána k přístupu k parametru uvnitř programu.

```
// deklarace třídy,
// hlavičkový soubor
// vstup: INT;
class init                          // soubor s implementací třídy
{                                   // konstruktor třídy
private:                            // vstup: INT;
#line 6 "test.plc"                  init :: init(void)
   CVariable<INT> m_vstup;          {
   ...                              #line 6 "test.plc"
public:                                vstup = CPropertyIn<INT> (&m_vstup);
#line 6 "test.plc"                  }
   CPropertyIn<INT> vstup;
   ...
};
```

Ukázka 4.10: Ukázka kódu vygenerovaného pro parametry funkcí, funkčních bloků nebo programů.

Kapitola 5

Zpracování jazyka IL

Tato část uvádí přehled jednotlivých instrukcí jazyka IL, zabývá se jejich zpracováním a vykonáváním v softwarovém PLC. Informace zde uvedené jsou čerpány z [5].

Jednotlivé instrukce jazyka IL můžeme rozdělit dle funkčního významu do následujících kategorií:

1. **Přiřazovací instrukce** - instrukce pro nastavení aktuální hodnoty zásobníku, proměnných nebo adres.
2. **Logické instrukce** - instrukce umožňující logické operace mezi proměnnými.
3. **Aritmetické instrukce** - instrukce pro násobení, dělení, sčítání atd.
4. **Porovnávací instrukce** - instrukce sloužící k porovnání argumentu a aktuálního stavu na zásobníku.
5. **Instrukce řízení toku programu** - instrukce pro podmíněné skoky, volání funkcí a funkčních bloků.

Každé z kategorií uvedených výše je věnována samostatná část v této kapitole popisující krátce vlastnosti a specifika jednotlivých částí.

Dle [5] by měl aktuální datový typ na zásobníku odpovídat datovéu typu operandu. Vzhledem k využití vlastností jazyka C++ jsou zajištěny implicitní konverze mezi většinou dostupných datových typů. Pokud požadovaná konverze neexistuje, je chyba hlášena v okamžiku kompilace vygenerovaného C++ souboru.

Dle [5] lze logické, aritmetické a porovnánací instrukce řetězit pomocí závorek, přičemž pravá závorka má vždy význam vyhodnocení výrazu od levé závorky. Ukázka 5.1 uvádí takové volání. Tato vlastnost není v implementovaném softwarovém PLC dostupná, neboť každý výraz lze nahradit takovou posloupností operací, která závorkování zcela nahradí.

```
MUL ( input1
SUB input2
)
(* Výsledek je interpretován jako:      *)
(* result := result * (input1 - input2) *)
```

Ukázka 5.1: Použití závorkovaných výrazů v instrukcích.

5.1 Přiřazovací instrukce

Tato část se zabývá zpracováním instrukcí, které umožňují přiřazovat hodnoty proměnných a adres. Jednotlivé instrukce, které do této části spadají a jejich významje uveden v tab.5.1.

Instrukce	Popis
LD	Nastaví aktuální hodnotu na zásobníku. Argumentem může být proměnná, adresa nebo konstanta.
LDN	Nastaví aktuální hodnotu na zásobníku, přičemž nejprve provede negaci hodnoty. Argumentem může být proměnná, adresa nebo konstanta.
ST	Uloží aktuální hodnotu ze zásobníku do požadované proměnné nebo na požadovanou adresu.
STN	Uloží aktuální hodnotu ze zásobníku do požadované proměnné nebo na požadovanou adresu, přičemž nejprve provede negaci hodnoty.
S	Nastaví obsah boolean proměnné nebo adresy v argumentu na 1, provádí se pouze v případě, že aktuální stav zásobníku je TRUE.
R	Nastaví obsah boolean proměnné nebo adresy v argumentu na 0, provádí se pouze v případě, že aktuální stav zásobníku je TRUE.

Tabulka 5.1: Přehled přiřazovacích instrukcí jazyka IL.

Vygenerovaný kód pro vybrané instrukce je patrný z ukázky 5.2.

Instrukce, které nastavují aktuální hodnotu zásobníku (ve vygenerovaném kódu proměnná result, definovaná jako instance třídy CResult) používají přetížený konstruktor objektu pro příslušný datový typ.

```
{  // LD 10
   result = 10;
}
{  // ST vstup
   result.GetValue(m_vstup);
}
{ // STN %QB3
   CVariable<BYTE> tmp0 = CVariable<BYTE>("QB3", 0);
   result.GetValueNeg(tmp0);
}
   // S %QX1.1
if (result == true) {
   CVariable<BIT> tmp0 = CVariable<BIT>("QX1.1", 0);
   tmp0 = 1;
}
```

Ukázka 5.2: Generovaný C++ kód pro přiřazovací instrukce jazyka IL.

5.2 Logické instrukce

Tato část se zabývá zpracováním logických instrukcí jazyka IL. Přehled jednotlivých logických instrukcí je uveden v tab.5.2.

Všechny instrukce spadající do této části používají jako první argument aktuální hodnotu zásobníku a jako druhý argument (operand instrukce) mohou mít buď číselnou konstantu, proměnnou (případně parametr funkce/funkčního bloku) nebo přímou adresu. Výsledek provedené operace je vždy uložen na zásobník, jako jeho aktuální hodnota (ve vygenerovaném kódu proměnná result).

Jednotlivé operace jsou implementovány jako přetížený operátor template třídy CResult z podpůrné PLC knihovny.

Ukázka 5.3 uvádí příklad C++ kódu, který je vygenerován pro logické instrukce.

5.3 Aritmetické instrukce

Tato část se zabývá zpracováním aritmetických instrukcí jazyka IL. Přehled jednotlivých instrukcí je uveden v tab.5.3. Základní sada aritmetických instrukcí jazyka IL obsahuje instrukce pro sčítání, odčítání, násobení a dělení všech základních datových typů. Jako první operand příslušné aritmetické operace je vždy použita aktuální hodnota zásobníku, jako druhý operand je použit parametr instrukce.

Výsledek aritmetické operace je vždy uložen zpět na zásobník jako jeho aktuální

Instrukce	Popis
AND	Na zásobník je uložen výsledek logické operace (Zásobník AND Operand), operand musí mít stejný datový typ jako aktuální hodnota zásobníku.
OR	Na zásobník je uložen výsledek logické operace (Zásobník OR Operand), operand musí mít stejný datový typ jako aktuální hodnota zásobníku.
XOR	Na zásobník je uložen výsledek logické operace (Zásobník XOR Operand), operand musí mít stejný datový typ jako aktuální hodnota zásobníku.
ANDN	Na zásobník je uložen výsledek logické operace (Zásobník AND NOT Operand), operand musí mít stejný datový typ jako aktuální hodnota zásobníku.
ORN	Na zásobník je uložen výsledek logické operace (Zásobník OR NOT Operand), operand musí mít stejný datový typ jako aktuální hodnota zásobníku.
XORN	Na zásobník je uložen výsledek logické operace (Zásobník XOR NOT Operand), operand musí mít stejný datový typ jako aktuální hodnota zásobníku.

Tabulka 5.2: Přehled logických instrukcí jazyka IL.

```
{  //  AND 2#0101_0101
   result = result & 85;
}
{  //  XORN 8#07_02
   result = result ^ ~(450);
}
```

Ukázka 5.3: Generovaný C++ kód pro logické instrukce jazyka IL.

hodnota.

Instrukce DIV (dělení) uloží v případě dělení nulou na zásobník jako aktuální hodnotu 0 a informace o chybě je vypsána jako hlášení do logu jádra operačního systému.

Ukázka 5.4 uvádí C++ kód, který je pro tyto instrukce generován.

Instrukce	Popis
ADD	Na zásobník je uložen výsledek aritmetické operace (Zásobník + Operand), operand musí mít stejný datový typ jako aktuální hodnota zásobníku.
SUB	Na zásobník je uložen výsledek aritmetické operace (Zásobník - Operand), operand musí mít stejný datový typ jako aktuální hodnota zásobníku.
MUL	Na zásobník je uložen výsledek aritmetické operace (Zásobník * Operand), operand musí mít stejný datový typ jako aktuální hodnota zásobníku.
DIV	Na zásobník je uložen výsledek aritmetické operace (Zásobník / Operand), operand musí mít stejný datový typ jako aktuální hodnota zásobníku.

Tabulka 5.3: Přehled aritmetických instrukcí jazyka IL.

```
{ // ADD 16#0F
   result = result + 15;
}
{  // MUL %IW4
   CVariable<WORD> tmp0 = CVariable<WORD>("IW4", 0);
   result = result * tmp0;
}
```

Ukázka 5.4: Generovaný C++ kód pro aritmetické instrukce jazyka IL.

5.4 Porovnávací instrukce

Tato část se zabývá zpracováním instrukcí jazyka IL pro porovnávání aktuálního stavu zásobníku a operandu. Přehled jednotlivých instrukcí je uveden v tab.5.2.

Všechny instrukce modifikují aktuální hodnotu zásobníku na logickou hodnotu TRUE nebo FALSE dle výsledku porovnání.

Ukázka 5.5 uvádí kód, který odpovídá vybraným porovnávacím instrukcím jazyka IL.

5.5 Instrukce řízení toku programu

Tato část se zabývá přehledem a zpracováním instrukcí pro řízení toku programu. Jedná se o instrukce umožňující provádět skoky v rámci programu, volat jiné programové

Instrukce	Popis
GT	Na zásobník je uloženo TRUE nebo FALSE dle výsledku porovnání (Zásobník > Operand).
GE	Na zásobník je uloženo TRUE nebo FALSE dle výsledku porovnání (Zásobník >= Operand).
EQ	Na zásobník je uloženo TRUE nebo FALSE dle výsledku porovnání (Zásobník = Operand).
NE	Na zásobník je uloženo TRUE nebo FALSE dle výsledku porovnání (Zásobník ≠ Operand).
LE	Na zásobník je uloženo TRUE nebo FALSE dle výsledku porovnání (Zásobník <= Operand).
LT	Na zásobník je uloženo TRUE nebo FALSE dle výsledku porovnání (Zásobník < Operand).

Tabulka 5.4: Přehled porovnávacích instrukcí jazyka IL.

```
{  //GT counter
   result = (result > m_counter);
}
{  //  NE 10
   result = (result != 10);
}
```

Ukázka 5.5: Generovaný C++ kód pro porovnávací instrukce jazyka IL.

bloku a provést návrat z volaného bloku. Přehled instrukcí je uveden v tab.5.5.

Kód generovaný pro jednotlivé instrukce z tab.5.5 je uveden v ukázce 5.6. Jak je patrno, instrukce JMP, JMPN a JMPC jsou nahrazeny v jazyce C++ konstrukcí goto LABEL;, přičemž návěští v PLC programu je převedeno do programu v jazyce C++. Instrukce RET, RETC a RETN jsou nahrazeny C++ voláním return;.

Složitější situace nastává v případě použití instrukcí CAL, CALC a CALN, které umožňují volat instance funkčních bloků, neboť je třeba zajistit také předání hodnot parametrů. Dle [5] lze hodnoty parametrů pro funkční bloky předávat 3 různými způsoby, tak jak je definováno v tab.5.6.

Vytvořený software podporuje možnosti č. 1 a 2. V případě referenčních argumentů funkčních bloků, které nejsou specifikovány, lze takový funkční blok volat s tím, že se použije implicitní hodnota parametru, přičemž případnou navracenou hodnotu již nelze z programu získat.

Ukázka 5.6 uvádí kód vygenerovaný při volání instance funkčního bloku.

Instrukce	Popis
JMP	Skok na návěští definované v rámci programového bloku.
CAL	Volání funkčního bloku.
RET	Návrat z volaného programového bloku.
JMPN	Podmíněný skok na návěští v rámci programového bloku v případě, že aktuální stav zásobníku je FALSE.
CALN	Podmíněné volání funkčního bloku v případě, že je aktuální stav zásobníku FALSE.
RETN	Podmíněný návrat z volaného programového bloku v případě, že aktuální stav zásobníku je FALSE.
JMPC	Podmíněný skok na návěští v rámci programového bloku v případě, že aktuální stav zásobníku je TRUE.
CALC	Podmíněné volání funkčního bloku v případě, že je aktuální stav zásobníku TRUE.
RETC	Podmíněný návrat z volaného programového bloku v případě, že aktuální stav zásobníku je TRUE.

Tabulka 5.5: Přehled instrukcí pro řízení toku programu jazyka IL.

Číslo	Popis/ukázka
1	CAL se vstupními parametry: CAL C10(CU := %IX10, PV := 15)
2	CAL + LD a ST pro nastavení vstupů: LD 15 ST C10.PV LD %IX10 ST C10.CU CAL C10
3	Použití vstupních operátorů: LD 15 PV C10 LD %IX10 CU C10

Tabulka 5.6: Možnosti předávání parametrů při volání funkčních bloků.

5.5.1 Volání funkcí

Funkce lze volat tak, že na místo instrukce se zapíše název funkce, přičemž jako první argument funkce je předána aktuální hodnota zásobníku. Případné další hodnoty parametrů lze zadat oddělené čárkou v rámci argumentu instrukce. Návratová hodnota funkce je uložena jako aktuální hodnota zásobníku. Ukázka 5.7 uvádí PLC

```
{  // CAL controler(P :=  2)
   g_controler.p = 2;
   g_controler.Function();
}
   // RETC
if (result != 0)
   RETURN();
{  // JMP ahoj
   goto ahoj;
}
```

Ukázka 5.6: Generovaný kód pro vybrané instrukce řízení toku programu jazyka IL.

kód pro volání funkce.

```
LD      10    (* Volání funkce MAX(X, Y) *)
MAX     20    (* Aktální hodnota zásobníku je 20 *)
```

Ukázka 5.7: Ukázka volání funkce z jazyka IL.

Ukázka 5.8 uvádí kód vygenerovaný během volání funkce. Nejprve je deklarována instance objektu, který funkci reprezentuje, následně je vyplněn první parametr funkce aktuální hodnotou zásobníku a poté jsou vyplněny případné další parametry. Po volání funkce je nastavena nová aktuální hodnota zásobníku.

```
{  // MIN 33
   MIN f_function;
   result.GetValue(f_function.parm0);
   f_function.parm1 = 33;
   f_function.Function();
   result = f_function.GetResult();
}
```

Ukázka 5.8: Generovaný C++ kód při volání funkce v jazyce IL.

Kapitola 6

Zpracování programových bloků

Tato část se zabývá zpracováním základních bloků, které se používají pro zápis programu.

Dle [5] jsou definovány tři typy programových bloků:

1. **Funkční bloky** - umožňují využít vstupní parametry, výstupní parametry a parametry vstupně-výstupní. Při použití funkčního bloku je třeba nejprve definovat jeho instanci. Vnitřní proměnné jsou platné po dobu platnosti instance.

2. **Programové bloky** - obdoba funkčních bloků. Liší se tím, že existuje pouze jedna globálně platná instance jejíž vnitřní proměnné jsou platné po celou dobu běhu programu.

3. **Funkce** - podporují pouze vstupní parametry, vždy používají návratovou hodnotu. Vnitřní proměnné jsou platné pouze po dobu jednoho volání funkce.

Jednotlivé bloky, jejich význam a použití jsou popsány v následujících částech této kapitoly. Veškeré uvedené informace jsou získány z [5].

Kód generovaný pro jednotlivé druhy programových bloků odpovídá vždy vygenerované třídě v jazyce C++. Název vygenerované třídy je vždy rozšířen o prefix, který zajišťuje, že je možné shodně pojmenovat funkci, funkční blok i blok typu program. Konverzní program vždy podle kontextu, ve kterém je název využit použije správný prefix a tím i správný blok. Pro každý z uvedených bloků je vždy vygenerován hlavičkový soubor a soubor s implementací třídy. Hlavičkový soubor je následně automaticky includován do generovaného kódu v případě, že je příslušný blok v programu využit.

6.1 Funkční bloky

Funkční bloky svou podstatou odpovídají objektům jazyka C++. Každý funkční blok předtím, než je možné ho volat musí mít deklarovánu svoji instanci pomocí proměnné s datovým typem odpovídajícím názvu funkčního bloku (viz. ukázka 6.1).

Jinými slovy, každý funkční blok může mít více instancí, které jsou reprezentovány různými proměnnými (buď globálními nebo lokálními). Zánikem proměnné zaniká také instance příslušného funkčního bloku.

```
...
VAR
   ...
   pid1: PID; (* deklarace instance FB pro PID regulátor *)
   ...
END_VAR
...
CAL pid1
...
```

Ukázka 6.1: Deklarace instance funkčního bloku a jeho následné volání.

Princip zápisu kódu pro definici těla funkčního bloku je patrný z ukázky 6.2. Deklarace jednotlivých bloků proměnných je nepovinná. Význam a využití jednotlivých bloků proměnných je detailně popsán v části 4.

Blok lokálních proměnných má nasledující význam:

- Může obsahovat také deklarace instancí dalších funkčních bloků.
- Všechny lokální proměnné si udržují svoji hodnotu po celou dobu existence instance funkčního bloku.

Norma [5] umožňuje předávat v parametrech funkčního bloku také odkazy na instanci jiného funkčního bloku. Tato možnost není ve vytvořeném softwarovém PLC dostupná. V některých případech by totiž bylo nutné zajistit kopírovací konstruktor objektu, kterým je funkční blok reprezentován. Problematické by též bylo chování v případě přiřazení instance funkčního bloku deklarované v rámci jiného bloku programu do výstupní nebo vstupně/výstupní proměnné.

Ukázka 6.4 obsahuje vygenerovaný kód pro funkční blok z ukázky 6.3. Z ukázky je patrné, že vygenerovaný kód je rozdělen na hlavičkový soubor s definicí třídy reprezentující funkční blok a zdrojový soubor s implementací této třídy.

Název třídy, která funkční blok reprezentuje je ve vygenerovaném kódu rozšířen o prefix "plcfb_". To umožňuje deklarovat v rámci programu různé typy bloků se stejným názvem.

```
FUNCTION_BLOCK name
   VAR_INPUT
      ... deklarace jednotlivých vstupních proměnných ...
   END_VAR
   VAR_OUTPUT
      ... deklarace jednotlivých výstupních proměnných ...
   END_VAR
   VAR_IN_OUT
      ... deklarace jednotlivých vstupně/výstupních proměnných ...
   END_VAR
   VAR
      ... deklarace jednotlivých lokálních proměnných ...
   END_VAR
   ...
   kód programu ve zvoleném jazyce (v našem případě IL)
   ...
END_FUNCTION_BLOCK
```

Ukázka 6.2: Deklarace funkčního bloku.

```
FUNCTION_BLOCK ukazka
   VAR_INPUT
      x: INT;
   END_VAR
   VAR_OUTPUT
      y: INT;
   END_VAR
   VAR
      state: INT := 1;
   END_VAR

   LD  state
   ADD 1
   ST  state
   MUL x
   ST  y
END_FUNCTION_BLOCK
```

Ukázka 6.3: Vstupní soubor pro ukázku vygenerovaného kódu funkčního bloku.

```
// deklarace třídy,
// hlavičkový soubor
#include <plc.h>

class plcfb_ukazka
{
   private:
#line 3 "fb.plc"
      CVariable<INT> m_x;
#line 6 "fb.plc"
      CVariable<INT> m_y;
#line 9 "fb.plc"
      CVariable<INT> m_state;
   public:
#line 3 "fb.plc"
      CPropertyIn<INT> x;
#line 6 "fb.plc"
      CPropertyOut<INT> y;
      plcfb_ukazka(void);
      virtual ~plcfb_ukazka(void);
      void Function(void);
      void ExitCall(void);
};
```

```
// soubor s implementací třídy
#include "ukazk.h"
#include "plcfb_ukazka.h"

#define RETURN() {ExitCall();return;}

plcfb_ukazka :: plcfb_ukazka(void)
{
#line 3 "fb.plc"
   x = CPropertyIn<INT> (&m_x);
#line 6 "fb.plc"
   y = CPropertyOut<INT> (&m_y);
#line 9 "fb.plc"
   m_state = CVariable<INT> ((INT) 1);
}
void plcfb_ukazka :: ExitCall(void)
{
}
plcfb_ukazka :: ~plcfb_ukazka(void)
{
}
void plcfb_ukazka :: Function(void)
{
   CResult result;
#line 12 "fb.plc"
   {
      result = m_state;
   }
#line 13 "fb.plc"
   {
      result = result + 1;
   }
#line 14 "fb.plc"
   {
      result.GetValue(m_state);
   }
#line 15 "fb.plc"
   {
      result = result * m_x;
   }
#line 16 "fb.plc"
   {
      result.GetValue(m_y);
   }
   RETURN();
}
```

Ukázka 6.4: Vygenerovaný kód funkčního bloku z ukázky 6.3.

6.2 Programové bloky

Programové bloky jsou obdobou funkčních bloků popsaných v předchozí části. Hlavní rozdíl spočívá v tom, že tento typ bloků má automaticky generovánu svoji unikátní instanci, která je platná po celou dobu běhu PLC. Tato instance samozřejmě uchovává hodnoty lokálních proměnných.

Programové bloky jsou určeny zejména pro spouštění jednotlivých vláken PLC programu, případně pro jednorázové spouštění např. při náběžné hraně signálu. Nicméně je lze volat i přímo v programu obdobně jako funkční bloky, přičemž jako parametr instrukce CAL se použije přímo název programového bloku.

Struktura zápisu programového bloku je patrná z ukázky 6.5 a je obdobná jako struktura funkčního bloku.

```
PROGRAM name
   VAR_INPUT
      ... deklarace jednotlivých vstupních proměnných ...
   END_VAR
   VAR_OUTPUT
      ... deklarace jednotlivých výstupních proměnných ...
   END_VAR
   VAR_IN_OUT
      ... deklarace jednotlivých vstupně/výstupních proměnných ...
   END_VAR
   VAR
      ... deklarace jednotlivých lokálních proměnných ...
   END_VAR
   ...
   kód programu ve zvoleném jazyce (v našem případě IL)
   ...
END_PROGRAM
```

Ukázka 6.5: Deklarace programového bloku.

Vygenerovaný kód odpovídá kódu funkčního bloku z ukázky 6.3, pouze prefix v názvu třídy je nahrazen "plcp_" a v hlavičkovém souboru je nadefinována globální proměnná deklarující instanci bloku pro využití v generovaném kódu.

6.3 Funkce

Funkce jsou nejjednodušší jednotky pro zápis programu. Umožňují předávat pouze vstupní parametry[1] přičemž jako první parametr funkce je předána aktuální hodnota zásobníku. Funkce musí předávat návratovou hodnotu, která se stane aktuální hodnotou zásobníku.

Způsob deklarace funkce je patrný z ukázky 6.6. První řádek deklarace definuje kromě názvu funkce také datový typ návratové hodnoty[2].

```
FUNCTION name:DATATYPE
   VAR_INPUT
      ... deklarace jednotlivých vstupních proměnných ...
   END_VAR
   VAR
      ... deklarace jednotlivých lokálních proměnných ...
   END_VAR
   ...
   kód programu ve zvoleném jazyce (v našem případě IL)
   ...
END_FUNCTION
```

Ukázka 6.6: Deklarace funkce.

Návratová hodnota funkce se nastaví tak, že se uloží hodnota do proměnné s názvem shodným s názvem funkce (viz. ukázka 6.7).

Každé funkci deklarované v kódu softwarového PLC odpovídá vygenerovaná třída v jazyce C++. Volání funkce následně spočívá ve vytvoření jednorázové instance této třídy. Tato instance třídy zanikne po ukončení volání a předání návratové hodnoty do aktuální hodnoty zásobníku. Jednorázovost instance třídy reprezentující funkci je zajištěna vytvořením speciálního name-space v jazyce C++[3].

Ukázka 6.9 uvádí vygenerovaný kód pro funkci z ukázky 6.8. Jak je patrno, princip generování kódu je shodný s generováním kódu pro předchozí programové bloky. Název vygenerované třídy je rozšířen o prefix "plcf_".

[1] Ze způsobu předávání hodnot parametrů vyplývá, že každá funkce musí mít alespoň jeden vstupní parametr.

[2] Jedná se o položku DATATYPE, která je oddělena od názvu funkce pomocí ":".

[3] Namespace zajišťuje platnost instancí všech tříd a je ohraničen pomocí {}

```
FUNCION exp:INT
   VAR_INPUT
      int Value;
   END_VAR
   LD Value
   ...
   ST exp (* Aktuální hodnotu zásobníku vložím do návratové *)
          (* hodnoty funkce *)
   ...
END_FUNCTION
```

Ukázka 6.7: Způsob nastavení návratové hodnoty funkce.

```
FUNCTION min: DINT
   VAR_INPUT
      val1: DINT;
      val2: DINT;
   END_VAR

   LD  val1
   ST  min
   LT  val2
   RETC

   LD  val2
   ST  min
END_FUNCTION
```

Ukázka 6.8: Vstupní kód pro ukázku generovaného kódu funkce.

```
// deklarace třídy,
// hlavičkový soubor
#include <plc.h>
class plcf_min
{
   private:
      CVariable<DINT> m_min;
#line 3 "function.plc"
      CVariable<DINT> m_val1;
#line 4 "function.plc"
      CVariable<DINT> m_val2;
   public:
#line 3 "function.plc"
      CPropertyIn<DINT> parm0;
#line 4 "function.plc"
      CPropertyIn<DINT> parm1;

      plcf_min(void);
      virtual ~plcf_min(void);
      void Function(void);
      const CVariable<DINT>&
            GetResult(void);
};
```

```
// Soubor s implementací třídy
#define RETURN() return;
plcf_min :: plcf_min(void)
{
#line 3 "function.plc"
   parm0 = CPropertyIn<DINT> (&m_val1);
#line 4 "function.plc"
   parm1 = CPropertyIn<DINT> (&m_val2);
}
plcf_min :: ~plcf_min(void)
{
}
const CVariable<DINT> &plcf_min ::
         GetResult(void)
{
   return(this->m_min);
}
void plcf_min :: Function(void)
{
   CResult result;
#line 7 "function.plc"
   {
      result = m_val1;
   }
#line 8 "function.plc"
   {
      result.GetValue(m_min);
   }
#line 9 "function.plc"
   {
      result = (result < m_val2);
   }
#line 10 "function.plc"
   if (result != 0)
      RETURN();
#line 12 "function.plc"
   {
      result = m_val2;
   }
#line 13 "function.plc"
   {
      result.GetValue(m_min);
   }
}
```

Ukázka 6.9: Vygenerovaný kód funkce z ukázky 6.8.

Kapitola 7

Konfigurace softwarového PLC

Tato část se zabývá speciálním blokem CONFIGURATION definovaným v [5]. Tento blok slouží:

- K deklaraci globálních proměnných dostupných v programu.
- K definici úloh (definují parametry pro spouštění bloků typu program).
- K přiřazení úloh k jednotlivým blokům typu program.

Ukázka 7.1 uvádí strukturu bloku typu CONFIGURATION se základními podbloky, které jsou vybrány pro implementaci v softwarovém PLC. Norma [5] definuje ještě další podbloky, které nejsou pro běh softwarového PLC třeba.

Blok s globálními proměnnými obsahuje deklarace globálních proměnných tak, jak je popsáno v kapitole 4. Proměnné mohou být buď standardních datových typů, nebo se může jedna o globální deklarace funkčních bloků.

Blokem RESOURCE se podrobněji zabývá následující část 7.1.

Blok CONFIGURATION v generovaném kódu má tedy za úkol definovat podklady pro vytvoření funkcí potřebných pro spuštění a ukončení softwarového PLC (zavedení kernel modulu softwarového PLC do paměti a jeho vyjmutí).

7.1 Struktura bloku RESOURCE

Tato část se zabývá významem a obsahem bloku RESOURCE. Informace zde uvedené jsou čerpány z [5].

Jak je patrno z ukázky 7.1, může i tento blok obsahovat deklaraci globálních proměnných stejně jako přímo blok CONFIGURATION.

Po případné deklaraci globálních proměnných již následuje definice úloh (TASK) a přiřazení těchto úloh jednotlivým blokům typu program. To zajistí spuštění softwarového PLC.

```
CONFIGURATION name
   VAR_GLOBAL
      ...
      variable
      ...
   END_VAR
   RESOURCE xx ON yy
      VAR_GLOBAL
         ...
         variable
         ...
      END_VAR
      ...
      TASK TaskName(INTERVAL:=T#0.1ms);
      ...
      ...
      PROGRAM p1 WITH TaskName: ProgramName();
      ...
   END_RESOURCE
END_CONFIGURATION
```

Ukázka 7.1: Struktura bloku CONFIGURATION.

7.1.1 Struktura bloků TASK

Bloky TASK definují vlastnosti pro spouštění bloků PROGRAM. Jedná se tedy o parametry, dle kterých budou následně vytvořeny real-time vlákna v rámci RT-Linuxového jádra. Blokům tudíž přímo neodpovídá žádný generovaný kód.

V ukázce 7.2 je uveden přehled všech parametrů, které se mohou v definici TASK bloku vyskytnout.

```
TASK name([INTERVAL := T#50ms][, PRIORITY := 2][, SINGLE := %Q2.1]);
```

Ukázka 7.2: Struktura bloku TASK a jeho zápis.

Tab.7.1 obsahuje popis jednotlivých parametrů bloku a jejich význam v různých kontextech.

Parametr	Popis
name	Název bloku, použitý v okamžiku přiřazení TASK bloku typu PROGRAM.
INTERVAL	Definuje interval, se kterým je spouštěno vlákno, které má přiřazeno tento TASK.
PRIORITY	Priorita vlákna, kterému je TASK přiřazen. Hodnota v rozsahu $0-50$, 0 odpovídá nejvyšší prioritě, 50 odpovídá nejvyšší prioritě.
SINGLE	Pokud je hodnota parametru INTERVAL různá od 0, pak definuje, že blok typu PROGRAM má být spouštěn periodicky pouze v případě, že hodnota vstupu SINGLE je log.0. Jestliže INTERVAL je 0, pak blok je spouštěn pouze v případě, že na vstupu SINGLE byla detekována náběžná hrana. Hodnota vstupu může být reprezentována buď přímo adresou nebo globální proměnnou.

Tabulka 7.1: Význam a popis parametrů bloku TASK.

7.1.2 Struktura bloků PROGRAM

Bloky typu PROGRAM v rámci deklarace RESOURCE slouží k přiřazení bloků TASK k jednotlivým programovým blokům PROGRAM[1]. Výsledkem je pro všechny bloky typu PROGRAM, které jsou zde uvedeny, vygenerování samostatného real-time vlákna, které dle parametrů uvedených v bloku TASK spouští příslušný blok PROGRAM.

Pokud v deklaraci chybí přiřazení bloku TASK, měl by být blok dle [5] spouštěn trvale s nejnižší možnou prioritou vždy, když má procesor volný čas. Podmínka týkající se spouštění vždy, když má procesor volný čas musí být pro vytvářené softwarové PLC modifikována.

V případě, že by totiž vždy existovalo nějaké RT-Linuxové vlákno, které má běžet a tudíž by nezůstal žádný volný procesorový čas, nedostalo by se ke slovu původní Linuxové jádra. To by vedlo k "zatuhnutí" celého systému. Proto jsou i tyto úlohy spouštěny periodicky s tím, že periodu spouštění je možné zvolit z příkazového řádku.

Deklarace přiřazení TASK k bloku PROGRAM je patrna z ukázky 7.3. Význam jednotlivých parametrů je uveden v tab.7.2.

```
PROGRAM Name [WITH TaskName]: ProgramName([Parm1 := value1[, ...]]);
```

Ukázka 7.3: Přiřazení definovaného TASK k bloku typu PROGRAM v rámci bloku RESOURCE.

[1] Ve významu bloku pro zápis programu z kapitoly 6.

Název	**Popis**
Name	Unikátní název definovaného vlákna.
TaskName	Název bloku TASK, jehož parametry se použijí při definici vlákna.
ProgramName	Název bloku typu PROGRAM, který bude spouštěn s parametry TASK.
Parm1	Pokud má blok parametry, obsahuje čárkou oddělené parametry s přiřazenou hodnotou. Hodnota může být buď konstanta, adresa v paměti PLC nebo globální proměnná.

Tabulka 7.2: Popis a význam parametrů při definici spouštění programu.

Ukázka 7.5 uvádí vygenerovaný kód v C++, který odpovídá definici z ukázky 7.4. Je z ní patrné, že každé vytvořené real-time vlákno je označeno pro plánovač procesů jako využívající FPU. Důvodem je možnost využití datového typu real kdekoliv v rámci PLC programu. Z toho plyne nutnost zajistit bezkonfliktní využití FPU v rámci celého operačního systému.

```
PROGRAM init
   VAR_INPUT
      stav: INT;
   END_VAR
END_PROGRAM

CONFIGURATION test
   RESOURCE xx ON yy
      TASK pok1(INTERVAL:=T#20ms, SINGLE:=%QX1.0);
      PROGRAM p1 WITH pok1: init(stav := %QB0);
   END_RESOURCE
END_CONFIGURATION
```

Ukázka 7.4: Definice spouštění PLC vláken.

```
#include <rtl_cpp.h>
CMemory<PLC_MEMORY_SIZE> g_KernelPLCMemory;
static void *PCallp1(void *pParms)
{
   struct sched_param p;

   p.sched_priority = 50;
   pthread_setschedparam(pthread_self(), SCHED_FIFO, &p);
   pthread_make_periodic_np(pthread_self(), gethrtime(), 20000000);

   while(1)
   {
      pthread_wait_np();
      if (CVariable<BOOL>("QX1.0", 0) != 0)
      {
         gprog_plcp_init.stav = CVariable<BIT>("QB0");
         gprog_plcp_init.Function();
      }
   }
   return NULL;
}

#define THREADS_COUNT 1
static pthread_t threads[THREADS_COUNT];
extern "C" {
int init_module(void)
{
   int iRetv = 0, iCount = 0;
   pthread_attr_t attr;

   __do_global_ctors_aux();
   pthread_attr_init(&attr);
   pthread_attr_setfp_np(&attr);
   iRetv += pthread_create(&threads[iCount++], attr, PCallp1, NULL);

   return iRetv;
}
void cleanup_module(void)
{
   for (int i = 0; i < THREADS_COUNT; i++)
   {
      pthread_delete_np(threads[i]);
   }
   __do_global_dtors_aux();
}
}
MODULE_LICENSE("GPL");
```

Ukázka 7.5: Vygenerovaný kód odpovídající ukázce 7.4.

Kapitola 8

Podpora pro RT-Linux

Tato kapitola se zabývá vytvořenou podporou pro RT-Linux. V části 8.1 je popsána podpůrná knihovna potřebná pro kompilaci modulů softwarového PLC, část 8.2 se zabývá tvorbou ovladačů pro vstupní a výstupní zařízení. Část 8.3 se zabývá potřebnou podporou přímo v jádře RT-Linuxu.

Obr.8.1 uvádí pro úplnost schematický náčrt celého vytvořeného softwarového PLC.

Šipky v obrázku naznačují, jakým způsobem probíhá vnitřní komunikace mezi jednotlivými částmi softwarového PLC.

8.1 Knihovna

Podpůrná knihovna umožňující kompilaci kódu vygenerovaného pomocí konverzního programu plc2cpp se skládá z několika částí, které spolu navzájem souvisejí. Jednotlivé části, které jsou implementovány v rámci podpůrné knihovny, jsou na obr.8.1 označeny v rámečcích s šedým podkladem.

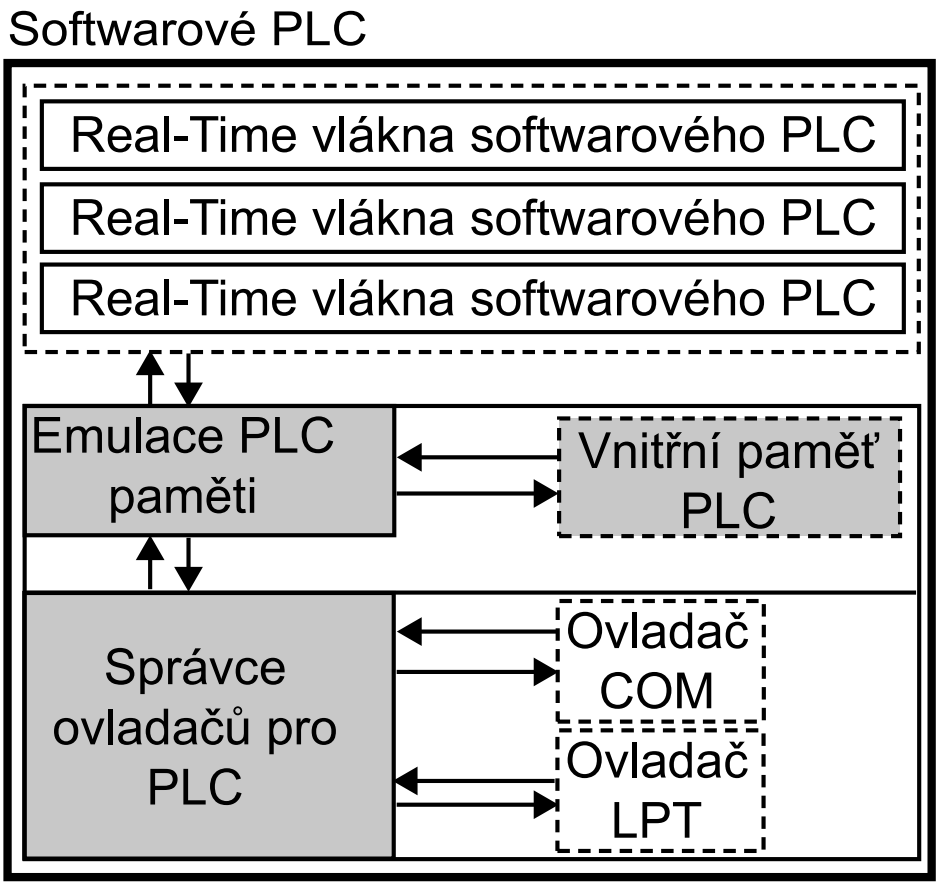
Tab.8.1 uvádí přehled jednotlivých tříd, které podpůrná knihovna implementuje. Ke každé třídě je zároveň uveden její krátký popis a význam.

8.1.1 Emulace paměti PLC

Emulace přístupu do paměti pomocí adres tak, jak je definováno v [5] implementuje třída CMemory[1]. Ta pro přístup do paměti umožňuje adresaci dle normy včetně přístupu na digitální vstupy/výstupy. V případě, že se jedná o adresu v interní PLC paměti, je přístup zpracován přímo třídou CMemory.

Pokud se jedná o přístup do paměti odpovídající adresaci digitálních vstupů/výstupů, je řízení předáno ovladači definovanému třídou CPLCDrivers. Podrobnější popis lze nalézt v části 8.2.

[1] Implementace v souboru cmemory.h



Obrázek 8.1: Schema vytvořeného softwarového PLC.

8.1.2 Podpora proměnných PLC

Podpora proměnných je implementována v template tříde CVariable[2]. Tato template třída implementuje následující operace:

- Specifikovat datový typ proměnné tak, jak je definováno v [5].
- Specifikovat inicializační hodnotu proměnné při deklaraci.
- Specifikovat adresu v paměťovém prostoru emulovaného PLC. Tato adresa samozřejmě může být i adresa vstupu nebo výstupu, je-li k této adrese zaregistrován některý ovladač ve správci zařízení (viz. dále).

[2]Implementace v souboru cvariable.h

Název třídy	Popis
CVariable	Třída využívaná k deklaraci proměnných ve vygenerovaném C++ kódu. Podporuje adresaci PLC paměti, pokud má proměnná tuto adresu v deklaraci specifikovánu.
CPropertyIn	Třída používaná ve vygenerovaném C++ kódu pro reprezentaci vstupních parametrů funkcí, funkčních bloků a programových bloků.
CPropertyOut	Třída využívaná ve vygenerovaném C++ kódu pro reprezentaci výstupních parametrů funkčních bloků a programových bloků.
CPropertyIO	Třída využívaná ve vygenerovaném C++ kódu pro reprezentaci vstupně-výstupních parametrů funkčních bloků a programových bloků.
CMemory	Třída, která implementuje emulaci PLC paměti, zapouzdřuje funkcionalitu pro přístup k adresám vnitřní paměti, vstupům a výstupům. Pro přístup k vstupům a výstupům využívá instanci třídy CDeviceManager.
CResult	Třída reprezentující aktuální hodnotu zásobníku.
CDeviceDriver	Virtuální třída využívaná pro tvorbu ovladačů vstupů a výstupů.
CDeviceManager	Třída spravující instance jednotlivých ovladačů a zajišťující mapování adresy vstupu/výstupu na příslušný ovladač zařízení.
CMutex	Třída zapouzdřující funkcionalitu mutexů pro zamykání kritických částí ovladačů PLC zařízení.
CSpinLock	Třída zapouzdřující funkcionalitu mutexů pro zamykání kritických částí ovladačů PLC zařízení (spinlock je funkčně podobný mutexu. Rozdíl je v tom, že po uzamčení oblasti pomocí spinlocku je zakázáno přerušení a tudíž prováděný kód nemůže být přerušen schedulerem).

Tabulka 8.1: Přehled tříd implementovaných v podpůrné knihovně pro PLC.

8.1.3 Podpora parametrů pro volání programových bloků

Podpora pro předávání parametrů programovým bloků je implementována v template třídách CPropertyIn, CPropertyOut a CPropertyIO. Při deklaraci parametru se definuje typ parametru (vstupní, výstupní, vstupně-výstupní) a dále se parametru přiřazuje

proměnná reprezentovaná třídou CVariable[3], která se používá v rámci vygenerovaného C++ kódu v programovacím jazyce PLC pro přístup k hodnotě parametru.

8.1.4 Emulace aktuální hodnoty zásobníku

Aktuální hodnota zásobníku, nebo též tzv. current-result je reprezentována třídou CResult[4]. Tato třída má následující vlastnosti:

- Uchovává výsledek poslední provedené operace včetně informace o jejím datovém typu.
- Implementuje operátory pro matematické a porovnávací operace.
- Umožňuje nastavit aktuální hodnotu zásobníku na hodnotu PLC proměnné(template třída CVariable).
- Umožňuje nastavit aktuální hodnotu zásobníku na hodnotu PLC parametru funkčního nebo programového bloku.

8.2 Ovladače karet digitálních vstupů/výstupů

Přístup na hardwarové karty pro digitální vstupy/výstupy je rozdělen na dvě úrovně (viz. obr.8.1). Nejvyšší úroveň je reprezentována třídou CDeviceManager, která je volána z objektu CMemory (objekt pro emulaci paměti PLC).

Tato třída udržuje seznam aktivních ovladačů karet pro PLC, přičemž zajišťuje předání volání ovladači příslušného hardwarového zařízení a zpracování požadavku. Vlastní seznam ovladačů je vytvořen jako pole statické velikosti obsahující odkazy na jednotlivé registrované ovladače[5]. Při vyhledávání ovladače odpovídajícímu zadané adrese se cyklicky prochází toto pole a ovladač je zde vyhledán. Pokud by bylo použito větší množství aktivních ovladačů, pak by bylo možno vytvořit seznam ovladačů jako vyhledávací strom pro zvýšení rychlosti vyhledání zařízení, které odpovídá požadované adrese. Vzhledem k tomu, že lze předpokládat za normálních okolností pouze jednotky použitých ovladačů, lze použité řešení považovat za dostatečně efektivní.

Následně je v paměťovém kontextu linuxového jádra k dispozici instance tohoto objektu s názvem g_KernelPLCDrivers, která je využívána vytvořeným softwarovým PLC i zaregistrovanými ovladači. Tato instance je vytvořena zavedením modulu plc_devicemanager.o do linuxového jádra a musí být dostupná před nahráním:

- Vlastního modulu, který reprezentuje emulované PLC.

[3]Obě template třídy jsou deklarovány se stejným parametrem šablony (stejným vnitřním datovým typem).

[4]Implementace v souboru cresult.h

[5]Je tedy třeba během překladu modulů odhadnout, kolik ovladačů bude maximálně použito a velikost pole správně nastavit.

- Jakéhokoliv používaného ovladače.

Modul lze zavést do paměti jedním z následujících způsobů[6]:

- V případě, že je softwarové PLC nainstalováno a modul je dostupný ve standardním adresáři modulů jádra:

```
]# modprobe plc_devicemanager
```

- V případě, že se modul nachází v jiném adresáři[7]:

```
]# insmod cesta/plc_devicemanager.o
```

8.2.1 Tvorba ovladače I/O karty

Tvorbu ovladače I/O zařízení lze rozdělit na dvě části:

1. Nejprve je třeba vytvořit odvozenou třídu od třídy CDeviceDriver, která implementuje virtuální metody třídy z ukázky 8.1 volané při přístupu na zařízení. Popis virtuálních metod je uveden v tab.8.2.

2. Poté je třeba vytvořit modul jádra RT-Linuxu, který během svého zavádění do paměti provede registraci instance třídy vytvořené v předchozím kroku do správce PLC zařízení. Registrací do správce zařízení ovladač oznámí, které adresní rozsahy vstupů a výstupů obsluhuje a mají mu tedy být předány. Registraci lze provést metodou RegisterDriver() základního objektu CDeviceDriver.

Název metody	
ReadAddress()	Virtuální metoda volaná v případě, že je třeba ze zařízení přečíst aktuální hodnotu zadané adresy.
SetAddress()	Virtuální metoda volaná v případě, že je třeba z programu nastavit hodnotu některé adresy, která odpovídá tomuto ovladači.
DriverInfo()	Metoda vracející název a popis ovladače. Je využívána při zobrazování informací týkajících se ovladače.

Tabulka 8.2: Popis virtuálních metod definovaných ve třídě CDeviceDriver.

Jako příklad pro tvorbu ovladače lze využít komentované zdrojové kódy vytvořených ovladačů, které jsou popsány níže.

Každý z následujících ovladačů umožňuje při zavádění modulu specifikovat také PLC adresy, na kterých bude ovladač dostupný. Přehled je k dispozici v tab.8.3.

Ovladač lze zavést do paměti pomocí následujícího příkazu:

[6]Případně lze zavedení modulu zajistit např. startovacím skriptem při spouštění operačního systému.

[7]Část cesta v umístění modulu může být absolutní nebo relativní cesta k modulu.

```
class CDeviceDriver
{
...
public:
   virtual int ReadAddress(void *Value,
                           int iValueSize,
                           int iInputAddr,
                           int iByteAddress,
                           int iBitAddress
   ) = 0;
   virtual int SetAddress(void *Value,
                          int iValueSize,
                          int iInputAddr,
                          int iByteAddress,
                          int iBitAddress
   ) = 0;
   virtual const char* DriverInfo(void) = 0;
...
};
```

Ukázka 8.1: Virtuální metody pro implementaci ovladače.

Parametr	**Popis**
iladdr	Parametr specifikující základní adresu, na které bude ovladač dostupný při adresování vstupů. Pokud je tedy např. hodnota iladdr=10, bude první adresa zařízení dostupná jako %IB10. Pokud parametr není specifikován, je zařízení adresováno od %IB0.
oladdr	Parametr specifikující základní adresu, na které bude ovladač dostupný při adresování výstupů. Pokud je tedy např. hodnota oladdr=7, bude první adresa zařízení dostupná jako %QB7. Pokud parametr není specifikován, je zařízení adresováno od %QB0.

Tabulka 8.3: Přehled společných parametrů pro moduly ovladačů.

```
]# insmod plc_lpt.o [iladdr=2] [oladdr=3]
```

Parametry modulu jsou nepovinné, předpokládá se, že je již zaveden modul správce ovladačů *plc_devicemanager.o*. Název modulu *plc_lpt.o* je třeba samozřejmě nahradit

příslušným názvem modulu, který chceme nahrát do paměti.

Ovladač PCI karty Advantech

Ovladač dodané karty Advantech PCI LabCard 1750 je vytvořen velice jednoduše. Ovladač si interně udržuje informace o všech aktuálních hodnotách výstupů. Tyto hodnoty jsou pak ovladačem navráceny v případě, že uživatel z programu čte hodnotu výstupu. Pokud uživatel zapíše na výstup novou hodnotu, je interní informace aktualizována a na výstupy je okamžitě zapsána požadovaná hodnota. V případě, že uživatel čte hodnoty vstupů, je aktuální hodnota získána při každém požadavku.

Ovladač by bylo možno upravit tak, že by vzorkoval hodnoty pomocí vlastního realtime vlákna maximální přípustnou frekvencí a uživatel by vždy získal pouze aktuální navzorkovanou hodnotu.

Ovladač paralelního portu

Ovladač standardního paralelního portu je vytvořen s ohledem na úlohu řízení otáček motorku. Popis a význam jednotlivých adres je uveden v tab.8.4.

Adresa	Velikost	Význam
0x00	1byte	Výstupní adresa, při zápisu odpovídá zápisu na adresu 0x378 (zápisu na základní adresu paralelního portu).
0x00	1byte	Vstupní adresa, při čtení odpovídá čtení adresy 0x379 (čtení základní adresy paralelního portu + 0x1).
0x01.0	1bit	Vstupní adresa, v případě, že obsahuje log.1, bylo vyvoláno přerušení od paralelního portu. Po přečtení hodnoty je automaticky nastaveno zpět na log.0.

Tabulka 8.4: Význam jednotlivých adres ovladače paralelního portu.

Úloha řízení motorku tento ovladač využívá k vytvoření PID regulátoru pro regulaci otáček pomocí PID regulátoru.

Název modulu ovladače paralelního portu je plc_lpt.o.

Ovladač pro komunikaci s user-space Linuxu

Tento ovladač slouží ke komunikaci s user-space Linuxu. Pomocí vstupních adres lze nastavovat hodnoty PLC adres z user-space části Linuxu, pomocí adres výstupních lze získávat z user-space části hodnoty proměnných PLC.

Pro komunikaci mezi kernel-space a user-space je využiváno jedno RT-Linux FIFO pro zápis proměnných do PLC, jedno pro čtení proměnných z PLC a jedno pro nastavování parametrů ovladače.

Ovladač poskytuje 64 adres pro vstupy a 64 adres pro výstupy. Velikost paměti je nastavována při kompilaci modulu pomocí makra PLC_USPACE_MEMORY.

Tab.8.5 popisuje, k čemu slouží jednotlivá real-time FIFO používaná ovladačem.

Index	**Název**	**Popis**
$FirstFIFO + 0$	Řídící FIFO	Slouží k naprogramování intervalu, se kterým je zasílán obsah vnitřní paměti ovladače. Pro zápis příkazu slouží struktura struct PLCUserSpaceControl deklarovaná v hlavičkovém souboru plc_uspace.h. Do položky struktury Command patří jedna z hodnot PLC_SEND_SET_TIME, PLC_SEND_STOP, PLC_RESET. Pokud je hodnota PLC_SEND_SET_TIME, patří do položky iTime časový interval v $[ms]$ s jakým budou zasílána data. Popis jednotlivých příkazů je popsán v hlavičkovém souboru.
$FirstFIFO + 1$	Vstupní FIFO	Slouží k zápisu dat odpovídajích jednotlivým vstupním adresám ovladače. Při zápisu musí být zasláno najednou hodnota všech vstupů.
$FirstFIFO + 2$	Výstupní FIFO	Slouží k získávání hodnot vstupů a výstupů ovladače (obsah vnitřní paměti ovladače). Struktura zasílaných dat je patrna z obr.8.2.

Tabulka 8.5: Popis jednotlivých RT FIFO využívaných ovladačem pro komunikaci s user-space.

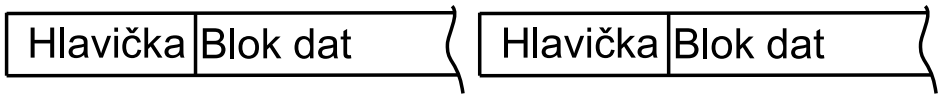
Ovladač je založen na následujícím principu:

- Po zavedení modulu ovladače (plc_uspacedriver.o) do paměti se zaregistruje ve správci PLC zařízení a otevře si 3 po sobě následující real-time fifo[8].
- Pokud chce uživatel získávat hodnoty proměnných ze softwarového PLC, otevře si řídící FIFO a naprogramuje si interval, s jakým má být do výstupního FIFO zasílán aktuální stav vnitřní paměti ovladače (adres).

[8]Pomocí parametru FirstFIFO při zavádění modulu lze specifikovat, jaké nejnižší FIFO se použije.

- Pokud chce uživatel nastavit některý ze vstupů ovladače, otevře si vstupní FIFO a zašle do něho nový stav všech vstupů. Tj. musí zapsat takové množství dat, které odpovídá velikosti vnitřní paměti ovladače[9].

Obr.8.2 ukazuje strukturu dat, která je zasílána ovladačem při čtení hodnot vstupů a výstupů ovladače. Hlavička nabývá vždy hodnoty "I"[10] nebo "O"[11]. Oba bloky jsou vždy zasílány za sebou. Pokud hlavička neobsahuje jednu z uvedených hodnot, pak zřejmě jsou data zasílána příliš rychle a proces v user-space Linuxu je nestíhá číst. Důsledkem je, že data v FIFO jsou zahazována. V tomto případě je vhodné provést reset ovladače (viz. popis řídícího FIFO).



Obrázek 8.2: Struktura bloku dat při čtení hodnot paměti ovladače.

Ovladač je určen zejména pro možnost vizualizace řídícího procesu. Takto je také využíván v rámci vzorové úlohy řízení otáček motorku.

Ovladač pro ladění programu

Ovladač pro ladění programu emuluje vstupy a výstupy pomocí vnitřní paměti, přičemž při změně hodnoty je vypsána na aktuální konzoli Linuxu nová hodnota nastavované proměnné včetně informace o adrese, na které se změna udála. Při čtení hodnoty je na konzoli vypsána informace o čtené adrese a hodnotě, která se na ní právě nachází.

Tento ovladač je vhodné využívat pouze v případě ladění pomalejších programů (nebo dočasně snížit frekvenci spouštění příslušného RT vlákna), neboť vzhledem k časové náročnosti výstupu na linuxovou konzoli může při častém přístupu dojít k nestabilitě celého systému.

8.3 Podpora C++ v RT-Linuxovém kernelu

Tato část souvisí s použitým překladačem GCC a jeho vlastnostmi využitelnými při překladu programu. V případě kompilace vygenerovaného C++ kódu jiným překladačem je možné, že se výčet potřebných symbolů bude lišit.

Pro překladač GCC je třeba mít v rámci jádra RT-Linuxu k dispozici modul rtl_cpp, který implementuje symboly pro některé funkce a operátory potřebné k běhu zkompilovaného kódu jazyka C++. Potřebné symboly se mohou lišit ze dvou důvodů:

1. Je použita odlišná verze překladače GCC.

[9] Hodnota makra PLC_USPACE_MEMORY při překladu ovladače.
[10] Pokud následuje blok s hodnotami vstupů.
[11] Pokud následuje blok s hodnotami výstupů.

2. Překladač GCC byl byl během své kompilace zkompilován s jinými parametry.

Součástí distribuce RT-Linuxu je verze modulu, která implementuje funkce *fputs(), __pure_virtual(), __cxa_atexit(), __this_fixmap_does_not_exist(), __assert_fail(), __isnan().*

V balíku RT-Linux Contrib je k dispozici novější verze modulu, která navíc oproti původní verzi implementuje operátory *new, new[], delete, delete[].* Tyto operátory jsou v podstatě pouze přesměrovány tak, aby využívaly volání standardního linuxového kernelu pro dynamické alokace a dealokace paměti. Z toho plyne omezení, že je lze využít pouze během inicializace a deinicializace modulu jádra. Dostupnost těchto operátorů je proto podmíněna existencí makra *__CPP_INIT__*. Navíc tato verze umožňuje linkování real-time modulů složených z více zdrojových souborů.

Na přiloženém CD je upravená verze původního modulu rtl_cpp z RT-Linux Contrib, která je vyzkoušena s kompilátory GCC 2.96 a GCC 3.2. Tato verze navíc implementuje symbol *__cxa_pure_virtual().*

Vzhledem k tomu, že kompilátor GCC lze kompilovat s mnoha různými parametry, může se stát, že v jiné verzi, případně verzi z jiné distribuce bude chybět nějaký další symbol. Obvykle se jedná o různé funkce se speciálním významem, které jsou v rámci normálních programů linkovány s defaultní funkcionalitou, lze je deklarovat podobně jako výše uvedené symboly.

Kapitola 9

Použití softwarového PLC

Tato kapitola se zabývá použitím vytvořeného software pro programování softwarového PLC. Nejprve je v části 9.1 uveden návod na zprovoznění programu, část 9.2 popisuje utilitu plc2cpp a její parametry na příkazovém řádku, část 9.3 popisuje možnost vytváření knihoven funkcí a funkčních bloků pro znovuvyužití již hotového kódu a část 9.4 obsahuje jednoduchý návod, jak vytvořit softwarové PLC.

9.1 Instalace softwarového PLC

V této části je uveden stručný návod ke zprovoznění softwarového PLC. Nejprve je však třeba mít k dispozici funkční instalaci RT-Linuxu. Krátký návod k jeho zprovoznění je uveden v příloze A.

V případě, že je RT-Linux úspěšně nainstalován, postup je následující:

1. Připravte si zdrojové kódy projektu. Získat je lze buď z přiloženého CD, nebo z domovské stránky projektu [2].

2. Rozbalíme zdrojový kód do pracovního adresáře[1]:

```
]# tar xzf SoftwarePLC-x.yy.tgz
```

3. Změníme aktuální adresář na adresář se zdrojovým kódem:

```
]# cd SoftwarePLC-x.yy
```

4. Spustíme konfigurační skript[2]:

```
]# ./configure
```

[1] Znaky x.yy znamenají verzi programu.

[2] V případě, že chcete změnit některé výchozí nastavení programu, lze přidat parametr –help pro nápovědu o dostupných parametrech.

5. Nyní již můžeme spustit kompilaci programu a podpůrných modulů jádra:

```
]# make dep
]# make
```

6. Pokud se kompilace zdařila, je můžeme přistoupit k instalaci programu:

```
]# make install
```

   Instalace spočívá v nakopírování modulů jádra se správcem ovladačů zařízení a jednotlivých ovladačů do systémového adresáře s moduly jádra, instalaci pomocného skriptu *mkplc.sh* pro vytváření softwarového PLC, instalaci preprocesoru *plc2cpp* a instalaci pomocných knihoven potřebných pro kompilaci PLC programu do modulu jádra pomocí C++.

V tuto chvíli máme tedy k dispozici vše potřebné pro vytváření uživatelských programů pro PLC.

9.2 Parametry příkazového řádku

Přehled a popis parametrů příkazového řádku utility plc2cpp je uveden v tab.9.1.

Parametr	**Popis**
-t directory	Adresář, do kterého se mají ukládat vygenerované zdrojové soubory v C++.
-s source	Název vstupního souboru v jazyce Instruction List.
-p project	Název projektu. Používá se pro název vygenerovaného modulu jádra, pro odvození názvu hlavičkového souboru s exportem globálních proměných atp.
-r time	Čas v $[ns]$ určující jak často bude detekována náběžná hrana signálu při spouštění úloh, které nejsou spouštěny periodicky.
-n time	Čas v $[ns]$ určující periodu spouštění úloh, které nemají přiřazen TASK.
-L libdirs	Další adresáře oddělené čárkou používané pro hledání knihoven.
-h	Zobrazí nápovědu k programu.
-l listfile	Do souboru listfile uloží seznam všech vygenerovaných souborů v C++ (včetně hlavičkových souborů).

Tabulka 9.1: Přehled parametrů příkazového řádku utility plc2cpp.

9.3 Vytváření knihoven

Vytváření knihoven je velmi jednoduché. Stačí vytvořit zdrojový soubor s jedním nebo více bloky pro zápis programu (funkce, funkční blok, blok typu program).

Knihovny se hledají v následujících adresářích:

- Pokud je při překladu plc2cpp definována hodnota makra PLC_LIBDIRS, je jeho hodnota předpokládána jako : oddělený seznam dalších adresářů, kde jsou hledány objekty z knihoven.
- Pokud je deklarována proměnná prostředí PLC_LIBDIR, je její hodnota předpokládána jako : oddělený seznam dalších adresářů, kde jsou hledány další objekty z knihoven.
- Další možností je parametr -L příkazového řádku(viz.tab.9.1), který může obsahovat čárkou oddělený seznam adresářů, kde se také mohou nacházet knihovní objekty.

V případě, že knihovna používá jinou knihovnu, lze ve zdrojovém kódu definovat závislost způsobem uvedeným v ukázce 9.1.

```
(* Include: min.plc,max.plc *)
(* Include: pid.plc *)
PROGRAM prg_control
...
```

Ukázka 9.1: Definice využití jiné knihovny ve zdrojovém kódu PLC.

Tato vlastnost není v [5] definována. Jedná se o rozšíření vytvořeného softwarového PLC, která je vytvořena za účelem komfortnějšího vytváření knihoven i programů v jazyce Instruction List.

V případě, že je nalezena direktiva *Include:* v komentáři, který začíná na novém řádku, jsou uvedené soubory zpracovány stejně, jako by byly součástí původního zdrojového kódu. Za direktivou může následovat jeden či více zdrojových souborů, jejichž jednotlivé názvy jsou odděleny čárkou.

9.4 Vytvoření softwarového PLC

V této části je uveden jednoduchý návod k vytvoření softwarového PLC za pomoci výše vytvořeného programu. Podmínkou pro správnou funkci je samozřejmě zprovozněné softwarové PLC a funkční RT-Linux.

Předpokládejme, že chceme vytvořit projekt, který se bude jmenovat *foo*, bude se nacházet v domovském adresáři uživatele v podadresáři s názvem *foo_plc* a bude se skládat ze dvou zdrojových souboru foo1.plc a foo2.plc uložených přímo v adresáři projektu.

Postup je tedy následující:

1. Založíme nový projekt

   ```
   tmp]# mkplc.sh --create foo --target-dir ~/foo_plc
   ```

2. Nyní přidáme zdrojové soubory do Makefile. Najdeme v adresáři */foo_plc* soubor Makefile. Otevřeme ho v oblíbeném textovém editoru a najdeme řádek:

   ```
   # FILES=deps/filename.plc
   ```

   Tento řádek změníme na:

   ```
   FILES=deps/foo1.plc deps/foo2.plc
   ```

3. Nyní vytvoříme soubory obsahující zdrojové kódy pro PLC v jazyce instruction list *foo1.plc* a *foo2.plc*

4. Chceme-li provést kompilaci programu zadáme na příkazovém řádku v adresáři projektu:

   ```
   foo_plc]# make
   ```

5. Pro spuštění vytvořeného softwarového PLC stačí zadat[3]:

   ```
   foo_plc]# make start
   ```

6. Pro zastavení softwarového PLC lze použít:

   ```
   foo]# make stop
   ```

Skript pro vytváření softwarového PLC je koncipován tak, aby byl co nejjednodušší. Předpokládá, že zdrojové kódy pro PLC se budou nacházet přímo v adresáři projektu, vygenerované soubory budou ukládány do podadresáře *sources* v adresáři projektu a vygenerovaný modul jádra se bude jmenovat *plc_projekt.o*. Dále je třeba, aby v některém zdrojovém souboru pro PLC byl definován blok typu *CONFIGURATION* s názvem

[3]Předpokládá se, že je spuštěn RT-Linux a zaveden modul *plc_devicemanager.o*, případně ovladače zařízení, které bude program využívat.

shodným s názvem projektu (v našem případě by to tedy byl blok *CONFIGURATION foo*.

V seznamu souborů, ze kterých se projekt skládá je třeba uvádět název souboru včetně podadresáře *deps*. Důvodem je, že v tomto podadresáři je uložen pomocný soubor, který obsahuje seznam dříve vygenerovaných souborů[4] a zároveň je využíván programem *make*, pro udržování závislostí v projektu.

Ukázka 9.2 uvádí základní kostru zdrojového kódu PLC. V kódu je definován jeden blok typu program využívající globální proměnnou *out*, konfigurační blok (s názvem stejným jako název projektu - tedy foo) a spouštění programového bloku *prg_rotace* s periodou 0.5s. Zároveň je v rámci komentáře na prvním řádku uvedeno použití knihovny *pid.plc*.

```
(* Include: pid.plc *)
PROGRAM prg_rotace
   VAR_EXTERNAL
      out: BYTE;
   END_VAR

   LD   out
   ...
END_PROGRAM

CONFIGURATION foo
VAR_GLOBAL
out AT %QB0: BYTE := 1;
END_VAR

RESOURCE xx ON yy
TASK rotaceTask(INTERVAL:=T#0.5s);

PROGRAM p1 WITH rotaceTask: prg_rotace();
END_RESOURCE
END_CONFIGURATION
```

Ukázka 9.2: Ukázka kostry PLC programu.

V případě složitějšího projektu lze samozřejmě vygenerovaný *Makefile* upravit ručně, nebo vytvořit zcela nový soubor odpovídající požadavkům.

[4]Tento seznam je používán pro smazání dříve vygenerovaných souborů v případě rekompilace.

Závěr

Úkolem diplomové práce bylo vytvořit pro operační systém RT-Linux softwarové vybavení umožňující spouštět program zapsaný v některém jazyce pro PLC, který je definovaném v normě IEC-1131-3.

Cíl práce se podařilo splnit. Bylo vytvořeno softwarové PLC s takovými základními vlastnostmi, jak je definováno v normě IEC-1131-3. Pro implementaci byl zvolen jazyk Instruction List. Funkcionalita vytvořeného PLC odpovídá požadavkům normy. Výběr vlastností byl proveden tak, aby bylo možno využít všechny instrukce, bloky pro zápis programu, proměnné reprezentované zástupným jménem i přímou adresou. Dále je možno využít všechny způsoby spouštění vláken PLC. Úmyslně nebyla implementována možnost využívat závorkování aritmetických operací. Dále se nepodařilo implementovat předávání funkčních bloků v parametrech funkcí, funkčních bloků a bloků typu program. V příloze B je uveden přehled vlastností definovaných v [5], které vytvořený systém podporuje, případně nepodporuje.

Vzhledem k potřebě vytvářet knihovny, je možné pomocí direktivy *Include:*, umístěné uvnitř komentáře definovat závislost zdrojového kódu na příslušné knihovně. A to jak v rámci koncového programu uživatele, tak v rámci kterékoliv knihovní funkce. Bližší informace o této vlastnosti lze nalézt v části 9.3.

Softwarové PLC bylo vytvořeno jako utilita (preprocesor) jazyka Instruction List do jazyka C++. Pro kompilaci do strojového kódu je využit standardní linuxový kompilátor. Pro kompilaci vygenerovaného C++ kódu je třeba pomocná knihovna, která je součástí softwarového PLC.

Vzhledem k volbě převádět PLC program na program v jazyce C++, byla zároveň ověřena možnost využít k tvorbě RT-Linuxových aplikací C++. Je však třeba se v programu držet omezení uvedených v části 2.2.

V rámci PLC programu je možné také využívat FPU pro matematické operace s pohyblivou řádovou čárkou. Toho je docíleno tím, že každé vygenerované vlákno má ve svých parametrech při zakládání uvedeno, že využívá FPU.

Součástí vytvořeného PLC je také návrh pro modulární tvorbu ovladačů, které umožňují z PLC programu přistupovat na externí zařízení. Modularita ovladačů umožňuje libovolně používané PLC konfigurovat dle dostupného hardware. V rámci projektu byly vytvořeny ovladače pro paralelní port, ovladač umožňující komunikaci mezi uživatelskou částí linuxu a softwarovým PLC s využitím pro visualizaci řízeného procesu. Dále byl vytvořen ovladač pro PCI kartu s digitálními vstupy/výstupy.

Funkčnost celého systému byla ověřena na PLC programu pro řízení otáček mo-

torku. Řídící program komplexně demonstruje vlastnosti a možnosti vytvořeného softwarového PLC. Využívá možnost spouštění úlohy na základě náběžné hrany (měření aktuálních otáček motorku), periodické spouštění programu pro regulaci otáček pomocí PID regulátoru. Zároveň je využit ovladač pro komunikaci s user-space RT-Linuxu.

Na vzorovém řídícím programu byly také otestovány rychlostní možnosti softwarového PLC. Na počítači Intel Pentium 3 700MHz bylo vyykoušeno cyklické spouštění vláken PLC programu s periodou kratší než 100[μs].

Vytvořené softwarové PLC je umístěno na internetové adrese [2] jako projekt pod licencí GPLv2. Na této adrese je též možné nalézt případně další úpravy a opravy programu.

Dodatek A

Instalace RT-Linuxu

Tato část v krátkosti uvádí návod k instalaci RT-Linuxu. Předpokládají se alespoň částečné znalosti potřebné pro správu Linuxu a instalaci uživatelsky kompilovaného jádra. Pro podrobnější informace je třeba si prostudovat příslušnou dokumentaci na níže uvedených adresách.

K instalaci je třeba z adresy http://www.rtlinux.org stáhnout distribuci RT-Linuxu. Součástí staženého balíku je i tzv. patch[1] pro vybrané verze původního linuxového jádra. Ten provede v původním zdrojovém kódu linuxu změny potřebné k běhu RT-Linuxu.

Nyní je možné si z adresy ftp://ftp.linux.org stáhnout zdrojový kód linuxového jádra. Je třeba vybrat takovou verzi, která je podporována RT-Linuxem (tj. existuje patch na tuto verzi linuxového jádra).

Máme-li k dispozici oba balíky popsané v předchozích odstavcích, můžeme přistoupit k vlastní instalaci:

- Budeme předpokládat, že všechny operace budeme provádět na cestě /usr/src:

  ```
  ]# cd /usr/src
  ```

- Nyní rozbalíme stažené linuxové jádro (pokud se stažený soubor jmenuje linux-2.4.20.tar.gz):

  ```
  ]# tar xzf linux-2.4.20.tar.gz
  ```

- Rozbalíme stažené zdrojové kódy RT-Linuxu (pokud se stažený soubor jmenuje rtlinux-3.2-pre1.tar.gz):

  ```
  ]# tar xzf rtlinux-3.2-pre1.tar.gz
  ```

- Nyní vytvoříme symblický link na nové zdrojové soubory jádra (tím zajistíme použití správných hlavičkových souborů během kompilace):

[1] Tj. soubor jehož aplikací na původní linuxové jádro získáme jádro s podporou real-time extenze.

```
]# rm -f /usr/src/linux
]# ln -s /usr/src/linux-2.4.20 /usr/src/linux
```

- Nyní najdeme v adresáři *rtlinux-3.2-pre1/patches* patch na použitou verzi jádra.
- Nyní aplikujeme patch (za předpokladu, že jsme našli soubor *kernel_patch-2.4.20-rtl3.2-pre1*):

```
]# cd linux-2.4.20
]# patch -p1 < ../rtlinux-3.2-pre1/patches/kernel_patch-
   2.4.20-rtl3.2-pre1
```

- Provedeme konfiguraci linuxového jádra (je třeba vybrat všechny potřebné ovladače a podporu modulů[2])

```
]# make menuconfig
```

- Nyní již můžeme provést kompilaci jádra a nainstalovat nové moduly:

```
]# make bzImage modules modules_install
```

- Zbývá ještě nakonfigurovat RT-Linuxovou extenzi a nainstalovat její moduly:

```
]# cd ../rtlinux-3.2-pre1
]# make menuconfig
]# make dep
]# make all
]# make install
```

Nyní již máme k dispozici zkompilovanou verzi linuxového jádra s podporou real-time extenze a moduly RT-Linuxu, které po zavedení do jádra linuxu aktivují real-time rozšíření (a tedy také RT-Linux). Zavedení těchto modulů lze provést např. pomocí příkazu[3]:

```
/etc/rc.d/rtlinux start
```

Další informace lze nalézt kromě internetu také v souborech README, případně INSTALL, které se nacházejí v adresářích s Linuxem, případně RT-Linuxem.

[2]Podpora modulů je třeba pro kompilaci RT-Linuxových modulů.

[3]Cesta ke spouštěcímu skriptu se může lišit v závislosti na parametrech zvolených při kompilaci RT-Linuxu a na použité distribuci linuxu

Dodatek B

Kompatibilita s normou IEC-1131-3

Níže uvedená tabulka obsahuje přehled podporovaných vlastností softwarového PLC vzhledem k požadovaným vlastnostem, které jsou uváděny v [5]. V tabulce je vždy uváděno číslo tabulky z [5], číslo vlastnosti, krátká charakteristika (obvykle pro celou skupinu vlastností), informace o tom, zda je vlastnost podporována a případně poznámku k implementaci.

Některé vlastnosti, které nesouvisí přímo s programovacím jazykem Instruction List, nejsou níže uvedeny.

Tab.-č.	**Popis**	**Podporováno**	**Poznámka**
1-1	Znakové sady		Podporovány ASCII znaky, diakritika podporována v rámci komentářů.
1-2	Malá písmena	Ano	Podporována v rámci názvů proměnných, funkcí a funkčních bloků. V těchto případech je rozdíl v malých a velkých písmenech ignorován.
1-3a		Ano	
1-3b		Ne	
1-4a		Ne	
1-4b		Ne	
1-5a		Ne	Programování pomocí žebříčkových diagramů není k dispozici.
1-5b			
1-6a	Závorkové operace	Ne	Viz. poznámka v kapitole 5.

Tab.-č.	Popis	Podporováno	Poznámka
1-6b	Závorkové operace	Ne	
2-1		Ano	
2-2		Ano	
2-3		Ano	
3-1	Komentáře	Ano	Na samostatném řádku i na řádku s kódem programu. Zároveň využito k definici závislosti na knihovnách (viz. kapitola 9).
4-1		Ano	
4-2		Ano	
4-3		Ano	
4-4		Ano	
4-5		Ano	
4-6		Ano	
4-7		Ano	
4-8		Ano	
5-1	Řetězcové konstanty	Ne	
6-1	Kombinace znaků v řetězcích	Ne	
6-2		Ne	
6-3		Ne	
6-4		Ne	
6-5		Ne	
6-6		Ne	
6-7		Ne	
6-8		Ne	
7-1a	Časové konstanty	Ano	
7-1b		Ano	
7-2a		Ano	
7-2b		Ano	
8-1	Konstanty pro datum a denní čas	Ne	Příslušné datové typy nejsou podporovány.
8-2		Ne	
8-3		Ne	
8-4		Ne	
8-5		Ne	
8-6		Ne	

Tab.-č.	Popis	Podporováno	Poznámka
10-1	Elementární datové typy	Ano	
10-2		Ano	
10-3		Ano	
10-4		Ano	
10-5		Ano	
10-6		Ano	
10-7		Ano	
10-8		Ano	
10-9		Ano	
10-10		Ano	
10-11		Ne	
10-12		Ano	
10-13		Ne	
10-14		Ne	
10-15		Ne	
10-16		Ne	
10-17		Ano	
10-18		Ano	
10-19		Ano	
10-20		Ano	
12-1	Odvozené uživatelské datové typy	Ne	
12-2		Ne	
12-3		Ne	
12-4		Ne	
12-5		Ne	
14-1	Inicializace uživatelských datových typů	Ne	
14-2		Ne	
14-3		Ne	
14-4		Ne	
14-5		Ne	
14-6		Ne	
15-1	Přímá adresace	Ano	
15-2		Ano	
15-3		Ano	
15-4		Ano	
15-5		Ano	
15-6		Ano	

Tab.-č.	Popis	Podporováno	Poznámka
15-7		Ano	
15-8		Ano	
15-9		Ano	
16-1	Bloky proměnných	Ano	
16-2		Ano	
16-3		Ano	
16-4		Ano	
16-5		Ano	
16-6		Ano	
16-7		Ne	
16-8		Ano	Klíčové slovo je ignorováno, původní význam je ztracen.
16-9		Ano	
16-10		Ano	
17-1	Deklarace proměnných	Ano	
17-2		Ano	Studený restart není možné běžnými prostředky v RT-Linuxu zpracovat. Klíčové slovo *RETAIN* je ignorováno, proměnné si po restartu nezachovávají svoji hodnotu.
17-3		Ano	
17-4		Ne	Pole nejsou podporována.
17-5		Ano	
17-6		Ne	
17-7		Ne	
17-8		Ne	
18-1	Inicializace proměnných	Ano	
18-2		Ano	Viz. poznámka o RETAIN proměnných.
18-3		Ano	
18-4		Ne	

Tab.-č.	Popis	Podporováno	Poznámka
18-5		Ano	S vyjímkou řetězcových proměnných.
18-6		Ne	
18-7		Ano	
18-8		Ne	
18-9		Ano	
33-1	Možnosti deklarace funkčních bloků	Ano	Klíčové slovo RETAIN je ignorováno.
33-2		Ano	Klíčové slovo RETAIN je ignorováno.
33-3		Ano	Klíčové slovo RETAIN je ignorováno.
33-4a		Ano	
33-4b		Ne	Grafické programovací jazyky nejsou podporovány.
33-5a		Ne	
33-5b		Ne	
33-6a		Ne	
33-6b		Ne	
33-7a		Ne	
33-7b		Ne	
33-8a		Ne	
33-8b		Ne	
33-9a		Ne	
33-9b		Ne	
49-1	Možnosti deklarace bloků CONFIGURATION a RESOURCE	Ano	
49-2		Ano	
49-3		Ano	
49-4		Ano	
49-5a		Ano	
49-5b		Ano	Spouštění je ve skutečnosti nahrazeno periodickým spouštěním. Bližší informace lze nalézt v kapitole 7.
49-6a		Ano	
49-6b		Ne	

Tab.-č.	Popis	Podporováno	Poznámka
49-6c		Ano	
49-7		Ne	Deklaraci lze nahradit pomocnou proměnnou.
49-8a		Ano	
49-8b		Ano	
49-9a		Ano	
49-9b		Ano	
49-10a		Ne	
49-10b		Ne	
49-10c		Ne	
49-10d		Ne	
49-10e		Ne	
49-10f		Ne	
50-1a	Parametry deklarace bloků TASK	Ano	
50-1b		Ano	
50-2a		Ne	Grafické jazyky nejsou podporovány.
50-2b		Ne	
50-3a		Ano	
50-3b		Ne	
50-4a		Ne	
50-4b		Ne	
50-5a		Ne	
50-5b		Ano	
51-1		Ano	
52-1	Instrukce jazyka IL	Ano	
52-2		Ano	
52-3		Ano	
52-4		Ano	
52-5		Ano	
52-6		Ano	
52-7		Ano	
52-8		Ano	
52-9		Ano	
52-10		Ano	
52-11		Ano	
52-12		Ano	
52-13		Ano	
52-14		Ano	

Tab.-č.	Popis	Podporováno	Poznámka
52-15		Ano	
52-16		Ano	
52-17		Ano	
52-18		Ano	
52-19		Ano	
52-20		Ano	
52-21		Ne	Řetězení operací pomocí závorek není podporováno.
53-1	Volání funkčních bloků	Ano	
53-2		Ano	
53-3		Ne	
54-4		Ne	Funkční bloky jako parametry nejsou podporovány.
54-5		Ne	
54-6		Ne	
54-7		Ne	
54-8		Ne	
54-9		Ne	
54-10		Ne	
54-11		Ne	
54-12		Ne	
54-13		Ne	

Dodatek C

Řízení motorku pomocí softwarového PLC

V této části je krátce popsána vzorová aplikace, která ukazuje, jak lze softwarové PLC využít. Jedná se o řízení motorku, který je připojen k PC pomocí paralelního portu.

Aplikace se skládá ze dvou nezávislých částí:

1. Vlastní řídící program napsaný v jazyce Instruction List pro PLC. Ten pomocí PID regulátoru řídí motorek, který je připojen k počítači pomocí paralelního portu.

2. Jednoduchá ukázková vizualizace pro X-Windows naprogramovaná v jazyce C++, zobrazující aktuální otáčky motorku a umožňující nastavit požadované otáčky motorku.

C.1 Řídící program motorku

Řídící program motorku je koncipován jako program pro PLC v jazyce Instruction List a jako takový je zpracován vytvořenou utilitou *plc2cpp*. Následně je zkompilován do modulu RT-Linuxového kernelu. Rychlost otáčení je řízena generátorem PWM.

Snímání směru a rychlosti otáčení i vlastní řízení probíhá pomocí paralelního portu (je tedy využíván vytvořený ovladač paralelního portu).

Velikost otáček (žádaná veličina) se nastavuje z user-space s využitím příslušného ovladače. Lze ji tedy dynamicky měnit za běhu programu. Stejně tak jsou předávány aktuální měřené otáčky do user-space RT-Linuxu.

Program se skládá ze tří periodicky spouštěných programů s náledujícím významem:

1. Snímání počtu impulzů vygenerovaných inkrementálním čidlem umístěným na hřídelce motorku. Z generovaných impulsů je zároveň určován směr otáčení.

2. Pravidelný převod počtu vygenerovaných impulsů na otáčky za sekundu. Hodnota otáček je kladná pro jeden směr a záporná pro směr druhý.

3. Vlastní výpočet řídících veličin. Skládá se z vyhodnocení regulační odchylky, výpočtu PID regulátoru a generování PWM.

Celý řídící program lze nalézt v podadresáři *plc/motorek* v archivu se softwarovým PLC.

C.2 Vizualizační program

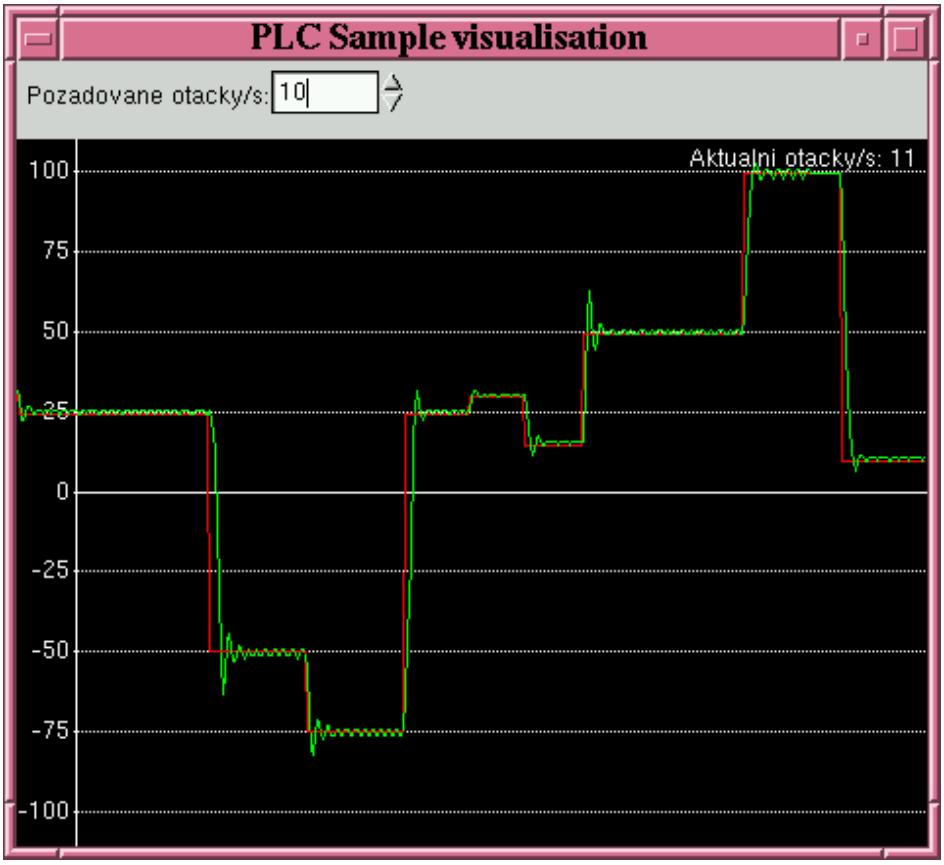
Vizualizační program je vytvořen jako jednoduchá aplikace pro X-Windows. Ke své činnosti využívá objektovou aplikační knihovnu wxWindows (viz. adresa [3]).

Program demonstruje možnosti využití PLC, přičemž má tyto vlastnosti:

- Umožňuje nastavit požadované otáčky motorku (tj. s využitím ovladače pro komunikaci PLC s user-space RT-Linuxu zapíše tuto hodnotu do paměťového prostoru softwarového PLC[1]).
- Graficky zobrazuje aktuální otáčky motorku tak jak je měří softwarové PLC (tj. tyto otáčky jsou čteny z paměťového prostoru softwarového PLC). Dále zobrazuje aktuální požadovanou hodnotu otáček, tak jak ji nastavil uživatel. Vzhled aplikace je patrný z obr.C.1.

Vizualizační program je dostupný v podadresáři *visualisation* softwarového PLC.

[1] Z pohledu RT-Linuxu se jedná o paměťový prostor linuxového kernelu.



Obrázek C.1: Ukázka jednoduchého vizualizačního rozhraní.

Dodatek D

Obsah přiloženého CD

Součástí diplomové práce je CD, jehož obsah a struktura jsou patrny z tab.D.1.

Adresář	**Obsah adresáře**
dokument	Text diplomové práce ve formátu PDF a Postscript.
linux	Zdrojové kódy linuxového jádra s aplikovaným patchem RT-Linuxu verze 3.2-pre1. Jedná se o verzi, na které bylo PLC vyvíjeno a testováno.
rtlinux	Stažené balíky pro RT-Linux a RT-Linux Contrib.
plc	Obsahuje vytvořené softwarové PLC.

Tabulka D.1: Obsah a struktura přiloženého CD.

Literatura

[1] *Domovské stránky projektu RT-Linux* [online]. <http://www.rtlinux.org>.

[2] *Domovské stránky projektu Softwarové PLC pro RT-Linux* [online]. <http://sourceforge.net/projects/softwareplc>.

[3] *Domovské stránky projektu wxWindows* [online]. <http://www.wxwindows.org>.

[4] Charles Donnelly and Stallman Richard. *Bison*. Free Software Foundation, 1995.

[5] In *IEC-1131-3 - Programmable controlers - Part3*, Březen 1993.

[6] A.M. Kuchling. *Regular Expression HOWTO* [online]. <http://www.amk.ca/python/howto/regex>.

[7] Melichar, Češka, Ježek, and Richta. *Konstrukce překladačů 1,2*. Vydavatelství ČVUT, 1999.

[8] Thomas Niemann. *A Compact Guide To Lex&Yacc*. ePaper Press.

[9] Victor Yodaiken, Duben 1999. *The RTLinux Manifesto* [online]. <http://www.fsmlabs.com/developers/white_papers/rtmanifesto.pdf>.