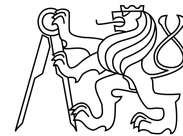
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	ídící systém a p eklada pracující v reálném ase pro pokro ilou ídící jednotku lineárního motoru
Student:	Bc. František Žemli ka
Vedoucí:	Ing. Mat j Bartík
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2017/18

Pokyny pro vypracování

Seznamte se s realizovaným systémem ídící jednotky, komunika ním protokolem a podporovanými instrukcemi. Navrhn te aplikaci (aplikace), která bude umož ovat vytvá ení uživatelských program pro ídící jednotku v grafickém prost edí a bude komunikovat s ídící jednotkou. Navrhn te možnost, jak pohyb motoru vizualizovat/simulovat. Tato aplikace by m la umožnit ukládání a na ítání program lineárního motoru pro ú ely opakovaných experimentálních m ení.

Funk ní požadavky:
- Tvorba program pro obsluhu motoru na základ p edvypo ítané trajektorie nebo možnost pohyb naprogramovat z jednotlivých krok .
- Ukládání, na ítání, editování a spojování existujících program .
- Vizualizace pohybu/stavu motoru a/nebo možnost program simulovat.

Nefunk ní požadavky:
- Aplikace bude komunikovat s ídící jednotkou po sériové lince.

Seznam odborné literatury

Dodá vedoucí práce.

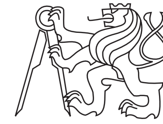
L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 24. února 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství



Diplomová práce

Řídící systém a překladač pracující v reálném čase pro pokročilou řídící jednotku lineárního motoru

Bc. František Žemlička

Vedoucí práce: Ing. Matěj Bartík

9. května 2016

Poděkování

Děkuji Ing. Bartíkovi za vedení této práce a mé kamarádce Evě za její korekturu. A především děkuji své rodině za podporu během let studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Řitce dne 9. května 2016 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2016 František Žemlička. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Žemlička, František. *Řídící systém a překladač pracující v reálném čase pro pokročilou řídící jednotku lineárního motoru.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato práce se zabývá analýzou chování lineárního motoru EZ Limo EZC6E030M–C a jeho nové řídící jednotky na FPGA čipu. Dále popisuje návrh, implementaci a testování desktopové ovládací aplikace, která umožňuje s jednotkou komunikovat a zadávat pro ní programy v grafickém prostředí.

Klíčová slova Lineární motor, řídící jednotka, sériový port, C++, Qt

Abstract

This thesis analyzes the behavior of the linear motor EZ Limo EZC6E030M–C and its new controller implemented on FPGA chip. It also describes the design, implementation and testing of desktop control application that lets user communicate with unit and input its programs in graphical environment.

Keywords Linear motor, control unit, serial port, C++, Qt

Obsah

Odkaz na tuto práci viii

Úvod **1**

1 Analýza **3**
1.1 Cíl práce 3
1.2 Stav v době zahájení práce 3
1.3 Funkční požadavky 3
1.3.1 Základní požadavky 3
1.3.2 Zadávání a editace modelů 4
1.3.3 Zobrazení stavových informací a průběhu 4
1.3.4 Nastavení řídící jednotky 4
1.3.5 Lineární motor EZ Limo EZC6E030M–C 4
1.4 Řídící jednotka 5
1.4.1 Obecný popis 5
1.4.2 Komunikační rozhraní 5
1.4.3 Instrukční sada 7
1.4.3.1 Obecný popis 7
1.4.3.2 Instrukce MOVE a WAIT 7
1.4.3.3 Prioritní instrukce 8
1.4.4 Instrukční cyklus 8
1.4.4.1 Časování instrukcí 8
1.4.5 Odezvy a události generované jednotkou 10

2 Návrh **13**
2.1 Rozbor funkčních požadavků vzhledem k vlastnostem motoru a řídící jednotky 13
2.1.1 Zadávání a editace průběhů 13
2.1.2 Zobrazení stavových informací 14
2.1.3 Reakce na události 14

2.1.4 Nastavení parametrů jednotky 16
2.1.5 Odesílání instrukcí 16
2.2 Výběr programovacího jazyka a knihoven 17
2.2.1 Java a její knihovny 17
2.2.2 C# a jeho knihovny 18
2.2.3 C++ a jeho knihovny 18
2.2.3.1 Boost 18
2.2.3.2 wxWidgets 19
2.2.3.3 Qt 19
2.2.4 Závěr 19
2.3 Architektura aplikace 19
2.3.1 Model 20
2.3.1.1 Communicator 20
2.3.1.2 Program 20
2.3.1.3 EngineModel 21
2.3.1.4 ModelHolder 21
2.3.2 View 21
2.4 Návrh uživatelského rozhraní 23
2.4.1 Hlavní okno aplikace 23
2.4.2 Dialogy 24
2.4.3 Rozdělení akcí 25

3 Realizace **27**
3.1 Správa paměti 27
3.2 Komunikace mezi komponentami, smyčka událostí 28
3.2.1 Smyčka událostí 28
3.2.2 Signály a sloty 28
3.2.3 Použití v aplikaci 28
3.3 Formát vstupních dat 29
3.4 Generování a editace průběhů 29
3.4.1 Sinusoida 30
3.4.2 Pila 30
3.4.3 Lineární funkce 31
3.4.4 Ruční zadávání instrukcí 31
3.4.5 Validace průběhů 32
3.4.6 Spojování a opakování průběhů 33
3.5 Překlad průběhů na instrukce 33
3.5.1 Spouštění programů 37
3.6 Komunikace s řídící jednotkou 38
3.6.1 Knihovna QSerialPort 38
3.6.2 Communicator 40
3.6.3 Komunikační stavy 40
3.6.4 Hlavní stavy 40
3.6.5 Přechodné stavy 41

3.7 View 42
3.7.1 Hlavní okno 42
3.7.2 PlotWidget 43
3.7.3 Informace o běžícícm průběhu 43
3.7.4 Odhad rychlosti 44

4 Testování **45**
4.1 Test překladače a editoru 45
4.2 Softwarová simulace 45
4.2.1 Popis simulátoru 46
4.3 Test s hardwarem 47
4.4 Výsledky testů 48
4.4.1 Zjištěné problémy 48
4.4.2 Změny implementace na základě testů 48

Závěr **49**

Literatura **51**

A Instrukční sada řídící jednotky **53**

B Odpovědi vysílané řídící jednotkou **57**

C Seznam použitých zkratek **59**

D Instalační příručka **61**

E Obsah přiloženého CD **63**

Seznam obrázků

1.1 Lineární motor EZ Limo EZC6E030M–C 5
1.2 Převodník USB - UART 6
1.3 Instrukční cyklus řídící jednotky [1] 10

2.1 Cyklus průběhu programu s událostí NEXT_256 15
2.2 Ideální plnění instrukční fronty 17
2.3 Podrobný diagram ModelHolderu 22
2.4 Diagram hlavních komponent 23
2.5 Návrh GUI hlavního okna aplikace 24
2.6 Návrh GUI generátoru průběhů 25

3.1 Hierarchie vlastnictví objektů 28
3.2 Ukázka průběhu sinusoidy 30
3.3 Ukázka průběhu pily 31
3.4 Ukázka lineárního průběhu 32
3.5 Spojení dvou průběhů s opakováním 35
3.6 Diagram procesu spuštění programu 39
3.7 Diagram stavů komunikace 41
3.8 Použití QDockWidgetu 43

4.1 Schéma činnosti simulátoru 46
4.2 Testování aplikace se simulátorem 47

Seznam tabulek

1.1 Parametry motoru 4
1.2 Parametry UART 6
1.3 Odezvy instrukcí generované řídící jednotkou 11
1.4 Struktura odezvy na instrukci STATUS_REQ 11
1.5 Odezvy událostí generované řídící jednotkou 12

A.1 Instrukční sada řídící jednotky. 55

B.1 Odezvy instrukcí generované řídící jednotkou 57
B.2 Odezvy událostí generované řídící jednotkou 58

Úvod

Tato práce vznikla z popudu Ing. Matěje Bartíka, aby doplnila chybějící softwarovou část jím vytvořené řídící jednotky lineárního motoru. Ta vznikla pro Laboratoř biomechaniky člověka Ústavu mechaniky biomechaniky a mechatroniky Fakulty strojní Českého vysokého učení technického v Praze jako náhrada řídící jednotky ESMC–C2 lineárního motoru EZ Limo EZC6E030M–C. Ten v laboratoři slouží k experimentálním měřením mechanických vlastností biologických a syntetických cévních náhrad.

Jednotka ESMC–C2 dodávaná výrobcem se vyznačuje nedostatečnou softwarovou podporou, která neumožuňuje zadávat delší pohyby a využít kompletní funkcionalitu motoru. Nakonec se podařilo naimplenetovat novou řídící jednotku na bázi FPGA čipu, která netrpí funkčními nedostatky jednotky ESMC-C2.

Součástí návrhu nové jednotky měl být i ovládací software, který by umožňoval pokročilé ovládání jednotky, sledování jejího stavu a zadávaní programů. K jeho implementaci bohužel nedošlo.

Tato práce se zabývá vytvořením chybějící ovládací aplikace spolu s překladačem, který umožní zadávat programy v podobě diskrétních reprezentací matematických funkcí pohybu motoru. V první kapitole jsou popsány parametry fungování řídící jednotky a motoru EZC6E030M–C. Druhá kapitola popisuje analýzu softwarových požadavků a návrh obslužného software, jeho architekturu a uživatelské rozhraní. Třetí kapitola se zabývá samotnou implementací, popisuje chování aplikace a použité algoritmy. Poslední, čtvrtá kapitola, se zabývá testováním jak pomocí softwarové simulace, tak s připojeným hardwarem.

Kapitola 1

Analýza

1.1 Cíl práce

Cíle této práce jsou analýza funkcí řídící jednotky lineárního motoru a vytvoření obslužné desktopové aplikace, která pro tuto jednotku umožní vytvářet uživatelské programy v grafickém prostředí. Aplikace bude komunikovat s řídící jednotkou, umožní uživateli její ovládání a poskytne zpětnou vazbu o jejím chování. Přesný výčet požadavků je popsán v části 1.3.

1.2 Stav v době zahájení práce

V době zahájení práce byla k dispozici hotová řídící jednotka implementovaná na FPGA čipu. Dále existovala demonstrační aplikace, která umožňuje generovat a spouštět pilovité průběhy. Jakékoliv složitější průběhy nebo použití pokročilých instrukcí je třeba zadávat ručně přímo ve strojovém kódu instrukční sady a odesílat je pomocí externích programů pro komunikaci po sériové lince, kterou jednotka používá.

1.3 Funkční požadavky

1.3.1 Základní požadavky

- Uživatel bude moci v rámci aplikace zadávat programy pro řídící jednotku, spouštět je a ovládat tak lineární motor.
- Programy budou zadávány v podobě funkcí představujících závislost polohy motoru na čase a překládány na instrukce z instrukční sady řídící jednotky.
- Aplikace bude komunikovat s řídící jednotkou pomocí sériové linky.

1.3.2 Zadávání a editace modelů

- Jednotlivé funkce průběhu budou načítány v podobě diskrétních hodnot z textového souboru.
- U každé funkce půjde nastavit počet opakování a změnit její parametry - dobu trvání, frekvenci, amplitudu.
- Aplikace bude obsahovat validaci, která uživatele upozorní, pokud jím zadaný průběh překročí maximální hodnotu amplitudy, případně rovnou zabrání jeho spuštění.
- Funkce bude možné řadit za sebe a spojovat do větších celků.
- Uživatel bude moci vytářet jednoduché průběhy přímo v aplikaci, spojovat je se svými vlastními průběhy a ukládat je.

1.3.3 Zobrazení stavových informací a průběhu

- Uživatel bude moci v aplikaci zobrazit grafy průběhů jednotlivých funkcí.
- V případě, že bude z aplikace spuštěn program v řídící jednotce, uvidí uživatel průběh vykonávání tohoto programu v okně aplikace.
- Aplikace zobrazí informace o stavu řídící jednotky a motoru, odhadovanou rychlost, stav magnetické brzdy a limity pohybu.

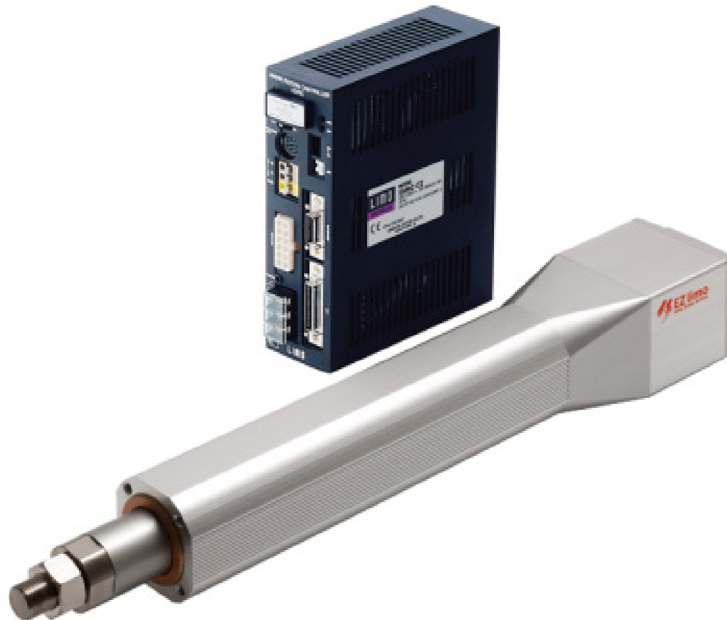
1.3.4 Nastavení řídící jednotky

- Aplikace umožní nastavovat parametry řídící jednotky, hranice pohybu motoru a ovládání magnetické brzdy.

1.3.5 Lineární motor EZ Limo EZC6E030M–C

Délka pohyblivé části	300 mm
Rozlišení	0,01 mm
Přesnost při opakovaném návratu na stejnou pozici	± 0.02 mm
Maximální rychlost	300 mm/s
Maximální síla tahu	400 N
Maximální síla tlačení	500 N
Magnetická brzda	Ano

Tabulka 1.1: Parametry lineárního motoru EZ Limo EZC6E030M–C [2]



Obrázek 1.1: Lineární motor EZ Limo EZC6E030M–C

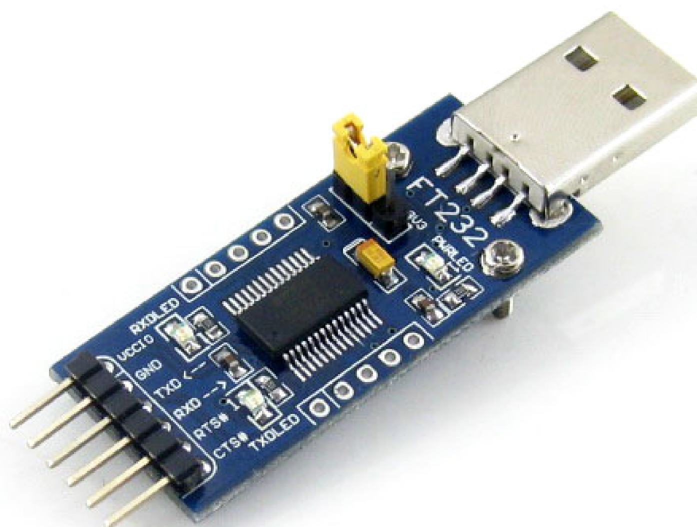
1.4 Řídící jednotka

1.4.1 Obecný popis

Řídící jednotka je navržena na bázi programovatelného hradlového pole FPGA. Jde o univerzální logický obvod, jehož chování lze naprogramovat. Používá vnitřní oscilátor s taktem 50 MHz. Jednotka nemá žádnou vlastní paměť, pouze několik registrů a instrukční frontu, která pojme 512 instrukcí o šířce 72 bitů [1].

1.4.2 Komunikační rozhraní

Řídící jednotka komunikuje s počítačem pomocí rozhraní sériové linky UART. Je to jeden z nejstarších a nejjednodušších protokolů, díky čemuž je využíván v mnoha zařízeních a stal se průmyslovým standardem. Funguje na bázi jednoduchého převodníku, který převádí jednotlivé bajty na sekvenci bitů, kterou následně vysílá po lince [3]. Na druhém konci této linky se nachází druhý UART, který přijaté bity dekóduje. V minulosti bylo toto rozhraní přítomno ve většině osobních počítačů v podobě sériového portu RS-232, který je nyní většinou nahrazen rozhraním USB. Existují ale převodníky mezi UART a USB viz obrázek 1.2.



Obrázek 1.2: Převodník USB - UART

Celá komunikace je izochronní[1] a používá start/stop bity k oddělení jednotlivých bloků dat. Parametry sériové linky implementované v řídící jednotce jsou vypsány v tabulce 1.2.

Nevýhodou tohoto rozhraní je nízká přenosová rychlost, proto se příliš nehodí pro velké objemy dat. Na druhou stranu – díky své jednoduchosti je podporován všemi operačními systémy a existuje pro něj mnoho knihoven v nejrozšířenějších programovacích jazycích. Navíc vzhledem k reakční době motoru je i tato rychlost dostatečná.

Přenosová rychlost	115 200 baudů[2]
Počet bitů	8
Počet stop bitů	1
Parita	Ne
Řízení toku dat	Ne

Tabulka 1.2: Parametry pro přenos dat přes rozhraní UART [1]

[1] Každý bit je reprezentován signálem konstantní délky.

1.4.3 Instrukční sada

1.4.3.1 Obecný popis

Všechny instrukce mají pevnou délku slova 72 bitů, kde prvních 8 bitů je určeno pro operační znak instrukce a zbylých 64 bitů je rozděleno na 2×32 bitů pro reprezentaci celočíselných operandů instrukcí [1].

Kompletní soupis všech instrukcí včetně jejich operačních kódů a popisu se nachází v příloze A. Zde jsou uvedeny pouze ty nejdůležitější.

1.4.3.2 Instrukce MOVE a WAIT

Toto jsou dvě nejdůležitější instrukce, které ovládají pohyb motoru. Obě dvě instrukce jsou závislé na na fyzických parametrech řídící jednotky, respektive samotného motoru. Konkrétně na délce kroku motoru a hodinovém taktu řídící jednotky.

MOVE zajišťuje pohyb motoru dopředu nebo dozadu a má dva operandy:

- Operand č.1 - interval mezi dvěma kroky motoru v hodinových taktech řídící jednotky.
- Operand č.2 - počet kroků motoru.

Řídící jednotka pracuje na frekvenci 50 MHz. Pokud se má jednotka pohybovat rychlostí například 1000 kroků/s, znamená to zapsat do operandu č.1 hodnotu $(50 * 10^6)/10^3 = 50000$.

Podle tabulky 1.1 má motor délku kroku 0,01 mm. Proto pokud má urazit například 10 mm, znamená to 1000 kroků, které se zapíší jako hodnota 2. operandu. Informace o směru pohybu je dána znaménkem operandu. Kladná hodnota znamená pohyb vpřed, záporná návrat do výchozí pozice. Motor má délku pohyblivé části 300 mm, to znamená 30 000 kroků. Je tedy vidět, že 32 bitů bohatě stačí na zadání celého pohybu.

WAIT je instrukce s jedním operandem, který označuje délku čekání v taktech. Maximální hodnota operandu je 2^{32}, délka jednoho taktu řídící jednotky je 20 ns. To znamená, že jednou instrukcí `WAIT` lze čekat $20 * 2^{32}$ ns $= 85,899$ s. To dostačuje při každém rozumném použití.

[2] Jednotka modulační rychlosti. Pro jednoduché protokoly bez zvláštního kódování odpovídá 1 Bd = 1 bit/s.

1.4.3.3 Prioritní instrukce

Instrukce `RESUME`, `HALT`, `FLUSH` a `RESETCMD` jsou instrukce sloužící k řízení činnosti jednotky. Jsou zpracovány prioritně mimo instrukční frontu a generují 1 bajtovou odpověď, která je odesílána sériovým portem ovládacímu počítači.

RESUME slouží k zahájení nebo obnovení činnosti jednotky. Po jejím zpracování se začnou vykonávat instrukce ve frontě, pokud není prázdná.

HALT zastaví činnost jednotky. Nevykonané instrukce zůstávají ve frontě.

FLUSH stejně jako `HALT` zastavuje činnost jednotky, ale navíc vyprázdní instrukční frontu, takže může být případně nahrán jiný program.

RESETCMD restartuje jednotku, která se tak dostane do výchozího stavu po zapnutí.

1.4.4 Instrukční cyklus

Instrukční cyklus řídící jednotky má čtyři fáze [1]:

- `IDLE` - klidový stav, prázdná instrukční fronta,
- `LOAD` - načítání instrukce z instrukční fronty,
- `DECODE` - dekódování instrukce, na základě, operačního kódu přejde jednotka do příslušného stavu,
- `EXECUTE` vykonání instrukce.

Stavový automat znázorňující jednotlivé fáze a přechody je vyznačen na obrázku 1.3. Fáze `EXECUTE` má několik větví (typů instrukcí), které jsou označeny rámečkem.

1.4.4.1 Časování instrukcí

Jak je vidět na obrázku 1.3, jednotlivé instrukce mohou mít rozdílnou délku fáze `EXECUTE`. Počet taktů potřebných k jejich vykonání se liší v závislosti na prioritě a generované odezvě.

MOVE a WAIT Tyto dvě instrukce mají nejdelší dobu provádění, protože jsou přímo závislé na činnosti motoru. Doba jejich provádění je určena těmito vzorci [1]:

- `MOVE`: $Time = (2 \times Prescaler + 1) \times Distance + 6 + 2 + 3 \quad [Cykly]$
- `WAIT`: $Time = (2 \times Timer + 1) + 6 + 2 + 3 \quad [Cykly]$

Provádění instrukce `WAIT` je prakticky totožné jako u `MOVE`, pouze parametr vzdálenosti je roven 1 a signály pro pohyb motoru jsou potlačeny. Proměnné *Prescaler, Distance a Timer* odpovídají jednotlivým operandům.

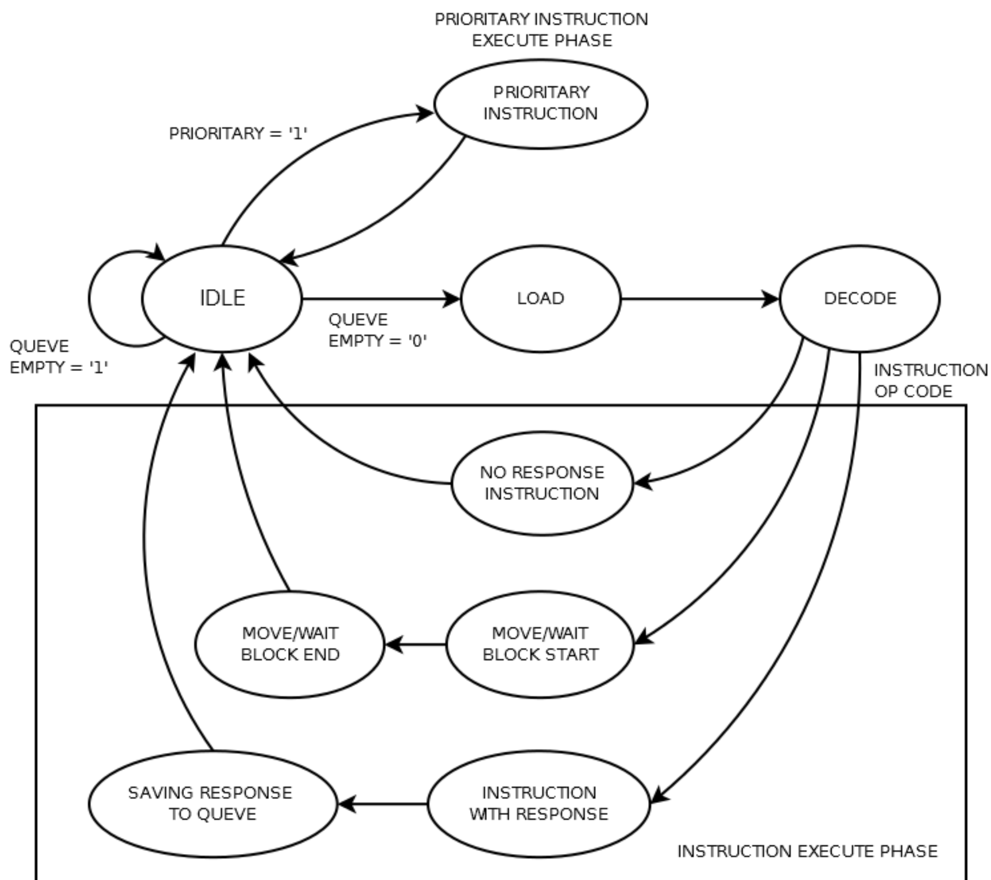
Instrukce bez odezvy Instrukce které negenerují odezvu mají dobu vykonání čtyři takty, což odpovídá jednotlivým fázím instrukčního cyklu. Mezi tyto instrukce patří `SET_POS`, `SET_TIME_*`, `SET_LIMIT_*`, `DISEN_LIMIT_*`, `EN_LIMIT_*`[3].

Instrukce s odezvou Instrukce, které generují odezvu, vysílanou po sériové lince, mají dobu vykonání závislou na velikosti odezvy. Zápis 1 bajtu do výstupní fronty zabere právě jeden takt. Doba jejich vykonání tedy je:

$$Time = 4 + delka_odezvy \quad [Cykly]$$

Prioritní instrukce Vzhledem k tomu, že instrukce `RESUME`, `HALT` a `FLUSH` se vykonávají mimo pořadí, závisí doba jejich provedení na právě probíhající instrukci. Její vykonání nelze přerušit, instrukce jsou atomické. Výjimkou jsou instrukce `MOVE` a `WAIT`. Ty lze přerušit během stavu MOVE_RUNNING, tedy když je vykonáván samotný pohyb motoru.

[3]Instrukce s hvězdičkou mají více verzí. Viz dodatek A



Obrázek 1.3: Instrukční cyklus řídící jednotky [1]

1.4.5 Odezvy a události generované jednotkou

Řídící jednotka odesílá zprávy po sériové lince jako odezvu na vykonání některých instrukcí a také jako informaci o událostech, ke kterým dochází během jejího běhu. Struktura odezvy instrukcí je popsána v tabulkách 1.3 a 1.4, události pak v tabulce 1.5.

Odezvy instrukcí generované řídící jednotkou			
Instrukce generující odezvu	Binární reprezentace odezvy – hlavička	Délka odezvy	Význam předávané hodnoty
HALT	1101 0001	1 B	Není
FLUSH	1101 0011	1 B	Není
RESUME	1101 0010	1 B	Není
READ_POS	0100 0101	5 B	Pozice motoru
READ_R_POS	0001 0101	5 B	Pozice získaná z kvadraturního enkodéru
READ_TIME	0010 0101	5 B	Času jednotky
BRAKE_EN	11110110	1 B	Není
BRAKE_DIS	11110111	1 B	Není
STATUS_REQ	1010 0100	4 B	Stavové informace, struktura odpovědi – tabulka 1.4
ECHO	1100 0011	1 B	Není
RELAY1_DOWN	1110 0000	1 B	Není
RELAY1_UP	1110 0001	1 B	Není
RELAY2_DOWN	1110 0010	1 B	Není
RELAY2_UP	1110 0011	1 B	Není
ALARM_CLEAR	0101 1010	1 B	Není

Tabulka 1.3: Odezvy instrukcí generované řídící jednotkou

Struktura odezvy na instrukci STATUS_REQ				
Bit(y)	39 – 32	31 – 27	26	25
Význam	Hlavička odpovědi	0000	Magnetická brzda	Limita 1
Bit(y)	24	23 – 19	18 – 8	7 – 0
Význam	Limita 2	00000	Počet instrukcí v instrukční frontě	0000 0000

Tabulka 1.4: Struktura odezvy na instrukci STATUS_REQ

Odezvy generované událostí řídící jednotky		
Událost generující odezvu	Binární reprezentace odezvy – hlavička	Délka odezvy
WATCHDOG	1101 0001	1 B
QUEVE_OVERLOADED	1101 0011	1 B
NEXT_256	1101 0010	1 B
REACHED_L1	0100 0101	1 B
REACHED_L2	0001 0101	1 B
BRAKE_MANUAL	0010 0101	1 B
ALARM	11110110	1

Tabulka 1.5: Odezvy událostí generované řídící jednotkou

Kapitola 2

Návrh

Tato kapitola popisuje návrh celé aplikace. Nejprve rozebere jednotlivé funkční požadavky vzhledem k fungování motoru. Následuje popis a srovnání vhodných programovacích jazyků a dostupných aplikačních knihoven a výběr jedné z nich pro použití v implementační části. Poslední částí kapitoly je návrh architektury a uživatelského rozhraní aplikace.

2.1 Rozbor funkčních požadavků vzhledem k vlastnostem motoru a řídící jednotky

Jak bylo popsáno v kapitole 1.4, řídící jednotka pracuje s taktem 50 MHz a zpracovává instrukce přijaté pomocí sériové linky. Instrukce jsou uloženy ve frontě s kapacitou 512 instrukcí. Instrukční sada jednotky obsahuje instrukce pro ovládání pohybu motoru, nastavování a čtení parametrů, zastavení a spuštění.

Jednotka ovládá lineární motor s maximální rychlostí pohybu 300 mm/s a dosažitelnou vzdáleností 300 mm. Během svého fungování jednotka vysílá zpět po sériové lince odezvy jako informace o některých vykonaných instrukcích nebo vnějších událostech.

Dostupné instrukce a uvedené parametry motoru a jednotky je třeba zohlednit při analýze funkčních požadavků a zahrnout do výsledné aplikace.

2.1.1 Zadávání a editace průběhů

Funkcionalita řídící jednotky nemá na zadávání průběhů žádný vliv. Jediné, co je třeba brát v potaz, jsou parametry motoru. Zadávané průběhy nemohou překročit maximální rychlost motoru a jeho rozsah. Proto bude třeba vytvořit mechanismus validace, který zabrání spuštění takových průběhů a ideálně upozorní uživatele na problémová místa.

2.1.2 Zobrazení stavových informací

Mezi požadavky na výslednou aplikaci patří mimo jiné i zobrazení průběhu právě spuštěného programu a stavových informací. Řídící jednotka má několik instrukcí, které umožňují takové informace získat, například `STATUS_REQUEST`, nebo `READ_R_POS`. Pomocí těchto a dalších instrukcí by tedy mělo být možné zjistit stav jednotky. Nevýhodou je, že tyto instrukce nemají v systému žádnou prioritu, a proto zřejmě nepůjde zjišťovat tyto informace kdykoliv za běhu, ale přinejlepším se zpožděním v závislosti na délce čekajících instrukcí `MOVE/WAIT`.

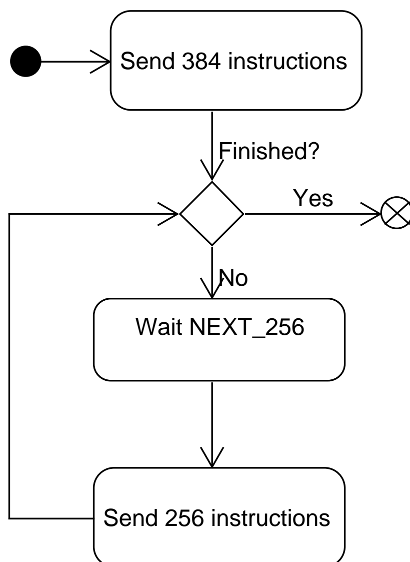
Dalším problémem je, že jednotka neodesílá žádné informace o vykonávání instrukcí `MOVE` a `WAIT`, a tedy ani o pohybu motoru. Jeho polohu lze sice zjistit pomocí `READ_R_POS`, to by ale znamenalo vložit takovou instrukci za každou `MOVE` a zároveň zajistit, aby jeden `MOVE` netrval příliš dlouho. Navíc podle [1], je signál pro zjištění pozice zpožděn o více než 10 ms. `READ_R_POS` se tedy hodí spíše k ověření dosažení očekávané koncové polohy motoru. Hodnoty polohy a rychlosti zobrazené v aplikaci budou tedy pouze lokálně vypočítané odhady. Ten může být navíc mírně zpožděn o čas odeslání prvního bloku instrukcí, nicméně toto zpoždění bude maximálně okolo 250 ms[4].

2.1.3 Reakce na události

Jak už bylo uvedeno v kapitole 1.4.5, kromě očekávaných odpovědí od odeslaných instrukcí, generuje jednotka také události, na které je třeba adekvátním způsobem reagovat. Některé z nich bude možné obsloužit automaticky, jiné signalizují potenciální problém a budou vyžadovat upozornění obsluhy.

NEXT_256 Toto je asi nejdůležitější událost, která signalizuje vykonání 256 `MOVE/WAIT` instrukcí. Ve chvíli jejího přijetí bude nezbytné, aby měla obslužná aplikace připravený další instrukční blok a neprodleně ho odeslala řídící jednotce nebo signalizovala ukončení programu. Z toho důvodu bude `NEXT_256` základem celého komunikačního protokolu. Jeho stručný návrh je vidět na diagramu 2.1.

[4]Odeslání prvního bloku instrukcí/přenosová rychlost $\frac{384*72b}{115200b} = 0,24 = 240ms$



Obrázek 2.1: Cyklus průběhu programu s událostí NEXT_256

REACHED L1 a L2 Tyto dvě události značí překročení povoleného rozsahu pohybu motoru stanoveného obsluhou aplikace. To znamená, že bude nutné nějakým, zřejmě vizuálním způsobem obsluhu upozornit, aby mohla reagovat.

QUEUE_OVERFLOW Přetečení fronty nastane, pokud bude odeslán například úvodní blok instrukcí do neprázdné fronty, nebo při špatné implementaci obsluhy události `NEXT_256`. Tato událost by neměla nastat, pokud bude dobře implementován komunikační algoritmus a aplikace bude odesílat správný počet instrukcí.

BRAKE_MANUAL Magnetickou brzdu lze spustit i ručně pomocí tlačítka řídící jednotky, což vyvolá tuto událost, indikující že motor nemůže pokračovat v běhu. V aplikaci proto bude neustále zobrazen stav magnetické brzdy a na jeho změny bude uživatel upozorněn.

ALARM Tato událost značí nemožnost vykonání zadaného příkazu motorem. Nastává například při pohybu mimo rozměrové hranice motoru nebo překročení jeho maximální rychlosti. Motor i řídící jednotka se v tom případě zastaví. Aplikace by měla informovat uživatele a následně odblokovat řídící jednotku instrukcemi `ALARM_CLEAR` a `FLUSH`.

2.1.4 Nastavení parametrů jednotky

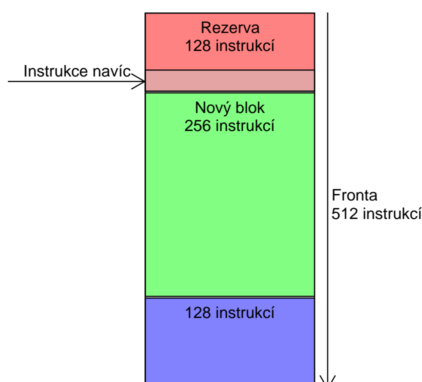
Řídící jednotka obsahuje instrukce pro zapnutí magnetické brzdy, času jednotky a nastavení limitů pro pohyb motoru, při jejichž dosažení upozorní uživatele odesláním události. Uživatelské rozhraní bude mít ovládací prvky, pomocí kterých se budou tyto parametry nastavovat. Při jejich použití aplikace vygeneruje a odešle příslušné instrukce.

Problémem je ale chybějící odezva některých instrukcí. Zatímco instrukce pro ovládání magnetické brzdy `BRAKE_EN` a `BRAKE_DIS` odezvu mají a není tím pádem problém promítnout informaci o stavu brzdy v uživatelském rozhraní, změny a zapínání limitů nebo času jsou bez odezvy. Tento nedostatek se bude kompenzovat následným odesláním instrukce `STATUS_REQUEST`, která zajistí potvrzení nového stavu. Přesný popis této komunikace je popsán v části 3.7.

Tyto změny nebude možné provádět při běhu programu, který vykonává pohyb motoru. Použité instrukce totiž nemají prioritu a proto by nebylo možné zajistit jejich provedení okamžitě, ale až se zpožděním daným délkou `MOVE` instrukcí.

2.1.5 Odesílání instrukcí

Podle [1] je doporučeným postupem odeslat při spuštění programu úvodní blok 384 instrukcí a následně reagovat odesíláním dalších bloků při přijetí události `NEXT_256`. Tím pádem je zajištěno, že jednotka bude mít stále dostatek instrukcí a zároveň nedojde k přetečení instrukční fronty. Navíc jednotka informuje pouze o vykonání `MOVE` a `WAIT`, 128 volných míst v instrukční frontě tak lze použít pro ostatní instrukce, přičemž se předpokládá, že těchto instrukcí bude vzhledem k jejich účelu minimum.



Obrázek 2.2: Ideální plnění instrukční fronty

2.2 Výběr programovacího jazyka a knihoven

Programovací jazyk je jedním z faktorů, které významně ovlivní následnou implementaci. Při jeho výběru hrálo roli několik kritérií, jako je dostupnost knihoven, kvalita dokumentace a podpora různých platforem, především Windows a případně i Linuxu. V potaz byly také brány osobní zkušenosti a znalosti uvedených knihoven.

S ohledem na výše uvedená kritéria a další funkční požadavky uvedené v kapitole 1.3, se nakonec rozhodování zúžilo na tři jazyky – C++, Javu a C#. Všechny tři patří do skupiny nejrozšířenějších jazyků [4] s velkým množstvím knihoven. Srovnání se zaměřuje hlavně na podporu GUI, vícevláknového zpracování a práce se sériovým portem, případně na dostupnost těchto knihoven.

2.2.1 Java a její knihovny

Podpora sériového portu Java nemá ve své standardní knihovně podporu pro práci se sériovým portem, takže je nutné využít externích knihoven. Jednou z možností je použití Java Communications API, rozšíření Javy vydané přímo Oraclem. Tato knihovna bohužel nepodporuje Windows [5], což jde proti záměru, aby aplikace byla multiplatformní. Kromě této knihovny existují i další, pod opensource licencí nebo placené.

Podpora vláken Java podporuje vícevláknové programování přímo prostřednictvím tříd v základní knihovně.

Podpora GUI Existují dvě základní knihovny pro tvorbu GUI desktopových aplikací. Starší AWT a novější Swing. Obě dvě jsou součástí základních knihoven Javy.

2.2.2 C# a jeho knihovny

C# je objektově orientovaný jazyk podobný Javě a C++. Je postaven nad frameworkem .NET a podobně jako Java je překládán do bajtkódu a spouštěn pomocí virtuálního stroje jménem CLR.

Podpora sériového portu Součástí .NETu je i C# implementace knihovny SerialPort [6], s jejíž pomocí lze naimplementovat synchronní i asynchronní komunikaci.

Podpora vláken Stejně jako Java má C# podporu vláken implementovanou jako součást základní knihovny.

Podpora GUI .NET má dvě různé knihovny pro tvorbu GUI, které se liší ve svém pojetí. Novější WPF je postavené na XML podobném jazyku XAML, který umožňuje oddělit definici grafického rozhraní od funkcionality aplikace.

2.2.3 C++ a jeho knihovny

U C++ je situace trochu složitější než u předchozích dvou jazyků. Předně, C++ je objektový jazyk vycházející z jazyka C, překládaný přímo do strojového kódu cílové platformy. To znamená, že v něm napsané programy musí být překládány zvlášť pro různé operační systémy a počítačové architektury. Nemá virtuální stroj, ale pouze minimální běhové prostředí, které nepodporuje garbage collection. Programátor tedy musí sám hlídat správnou alokaci paměti. Na druhou stranu nad ní má absolutní kontrolu. C++ má vlastní standardní knihovnu STL, která ale poněkud zaostává v abstrakci operací, které jsou závislé na operačním systému, jako je práce se souborovým systémem, nebo GUI. Z toho důvodu je třeba používat různé knihovny, API operačního systému nebo frameworky.

Abychom se vyhnuli případné nutnosti použítí WinAPI nebo různých unixových knihoven, což by znamenalo svázání s jedním operačním systémem, rozebereme možnosti několika velkých knihoven a frameworků, které by měly pokrýt všechny potřeby aplikace.

2.2.3.1 Boost

Boost je soubor knihoven mající za cíl doplnit chybějící funkcionalitu STL [7]. Je vydaný pod vlastní opensource licencí, která umožňuje volné použití pro ko-

merční i nekomerční účely [8]. Kromě jiných obsahuje třídy pro práci s vlákny a také sériovým portem. Nemá ale žádnou podporu grafiky.

2.2.3.2 wxWidgets

Multiplatformní knihovna pro tvorbu GUI aplikací. Umožňuje vytvářet grafické prostředí s nativním vzhledem na všech nejrozšířenějších operačních systémech. Je dostupná pod wxWindows licencí, která je založená na LGPL, ale nevyžaduje zveřejnění zdrojového kódu, takže se hodí i pro komerční aplikace [9]. Kromě tvorby GUI podporuje také práci s médii, sítí nebo vlákny. Bohužel nemá žádnou podporu práce se sériovým portem. Mohla by ale být vhodným doplňkem ke knihovně Boost.

2.2.3.3 Qt

Qt je spolu s wxWidgets asi nejrozšířenější knihovnou pro tvorbu grafického prostředí. Od wxWidgets se ale dost zásadně liší. Především nejde jen o grafickou knihovnu, ale ucelený framework pro tvorbu aplikací v různých jazycích, primárně v C++ a Pythonu. Funguje na několika platformách od Windows a Linuxu až po Android. Obsahuje knihovny pro práci s GUI, multimédii, vlákny, souborovým systémem a mnoho dalších. Má dokonce vlastní vývojové prostředí Qt Creator s GUI návrhářem. Je vyvíjen komerčně, ale umožňuje nekomerční použití pod LGPL licencí. Ve frameworku Qt je vytvořeno linuxové grafické prostředí KDE a jeho tvůrci dohlíží na to, že Qt zůstane opensource [10].

Nevýhodou Qt je, že pro implementaci některých svých principů rozšiřuje syntaxi C++. To sice zjednodušuje psaní programů, nicméně ty poté nelze přeložit běžným kompilátorem. Namísto toho je nutné nejprve použít generátor kódu zvaný Meta-Object Compiler (MOC), který z Qt specifických částí kódu vygeneruje čisté C++ přeložitelné kterýmkoliv překladačem [11].

2.2.4 Závěr

Pro implementaci byl nakonec vybrán jazyk C++ s frameworkem Qt. Důvodem byl především fakt, že framework obsahuje všechny potřebné knihovny a má dobře zpracovanou dokumentaci s množstvím návodů. Zároveň je multiplatformní s open source licencí pro nekomerční užití, což dostačuje pro naše použití. Svou roli také hrála předchozí zkušenosti s jeho použitím.

2.3 Architektura aplikace

Při návrhu aplikace byl použit architektonický vzor Model-View-Controller (MVC), respektive jeho Qt variantu zvanou Model/View. Koncept MVC spo-

čívá v rozdělení objektů do tří typů. Modelu, který reprezentuje data a vnitřní logiku aplikace, View který zajišťuje prezentaci dat v podobě uživatelského rozhraní a Controlleru který reaguje na uživatelské vstupy [12]. Qt tento koncept zjednodušuje sloučením Controlleru a View a výsledkem je architektura zvaná Model/View [13].

MVC většinou využívá implementace návrhového vzoru Observer k reagování na změny Modelu ve View. Ten spočívá v závislosti 1:N mezi objekty, kdy jeden objekt automaticky upozorňuje na změny N ostatních objektů [12]. Tato funkcionalita je zakomponována přímo v Qt jako *signály a sloty* [14] a je popsána dále v kapitole 3.2.

Na základě těchto konceptů bude aplikace rozdělena do dvou částí, Modelu a View, z nichž každá se bude skládat z několika dalších komponent.

2.3.1 Model

Model bude dále rozdělen do čtyř komponenet s různou úrovní provázanosti. Tyto komponenty mají každá na starost jednu logickou část aplikace a vzájemnou komunikací zajišťují její fungování. Tyto části se nazývají Communicator, ModelHolder, EngineModel a Program. V popisu každé z nich je uveden seznam činností, za které zodpovídají.

2.3.1.1 Communicator

Communicator má na starosti samotnou komunikaci s řídící jednotkou.

- Udržuje spojení přes sériový port.
- Odesílá instrukce spuštěného programu.
- Přijímá a obsluhuje události od řídící jednotky.
- Uchovává informaci o stavu vykonávání programu v jednotce (běží, zastaveno, pauza).

2.3.1.2 Program

Program reprezentuje sled instrukcí, které mají být vykonány řídící jednotkou.

- Volá kompilátor na jednotlivé modely z ModelHolderu a udržuje výsledný sled instrukcí.
- Na vyžádání od Communicatoru vytváří instrukční bloky k odeslání.
- Generuje instrukce pro servisní požadavky (Magnetická brzda apod.).
- Řídí zobrazení aktuálního průběhu ve View.

2.3.1.3 EngineModel

Je nejmenší část modelu, která udržuje informace o parametrech motoru a řídící jednotky.

- Definuje statické parametry jako je maximální rychlost a rozsah motoru nebo takt řídící jednotky, které předává kompilátoru nebo ModelHolderu při validaci modelů.
- Obsahuje informace o stavu magnetické brzdy a nastavených limitech motoru.

2.3.1.4 ModelHolder

Je složen z modelů jednotlivých průběhů a zajišťuje manipulaci s nimi.

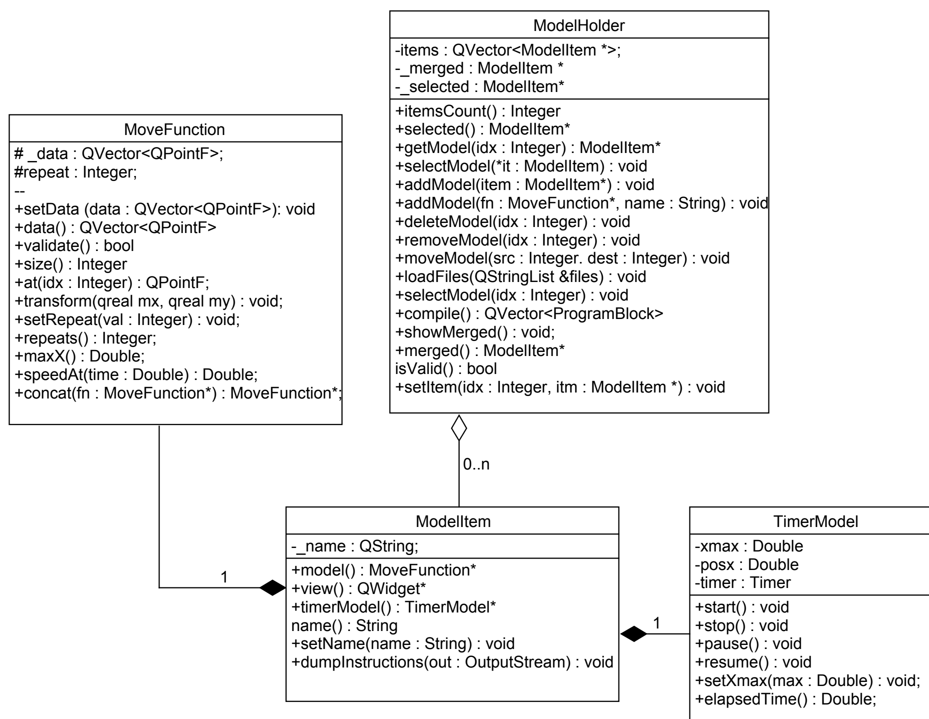
- Udržuje seznam modelů průběhů.
- Zajišťuje načítání a ukládání průběhů do souborů. Obsluhuje také mazání nebo změnu pořadí.
- Pro každý model drží jeho příslušný View.
- Umožňuje transformaci a spojování jednotlivých průběhů.
- Provádí validaci modelů před spuštěním.

Detailní pohled na tuto komponentu lze vidět na diagramu 2.3. Třída *ModelHolder* funguje jako kontejner pro objekty *ModelItem*. Každý z nich obsahuje objekt třídy *MoveFunction* jako reprezentaci funkce průběhu. Dále *TimerModel*, který slouží k výpočtu odhadu pozice motoru a odkaz na příslušný PlotWidget, komponentu View.

2.3.2 View

View bude složen z několika samostatných vizuálních komponent – widgetů. Každý widget bude zobrazovat informace a ovládací prvky odpovídající příslušným modelům. Všechny tyto widgety pak budou zobrazeny v rámci hlavního okna aplikace MainWindow.

LogWidget Zobrazuje v textové podobě informace o běhu aplikace. Zaznamenává detailní popis akcí a událostí Communicatoru a ModelHolderu. Také zaznamená a informuje uživatele, pokud se pokouší provést neplatnou akci, například otevřít neznámý soubor.



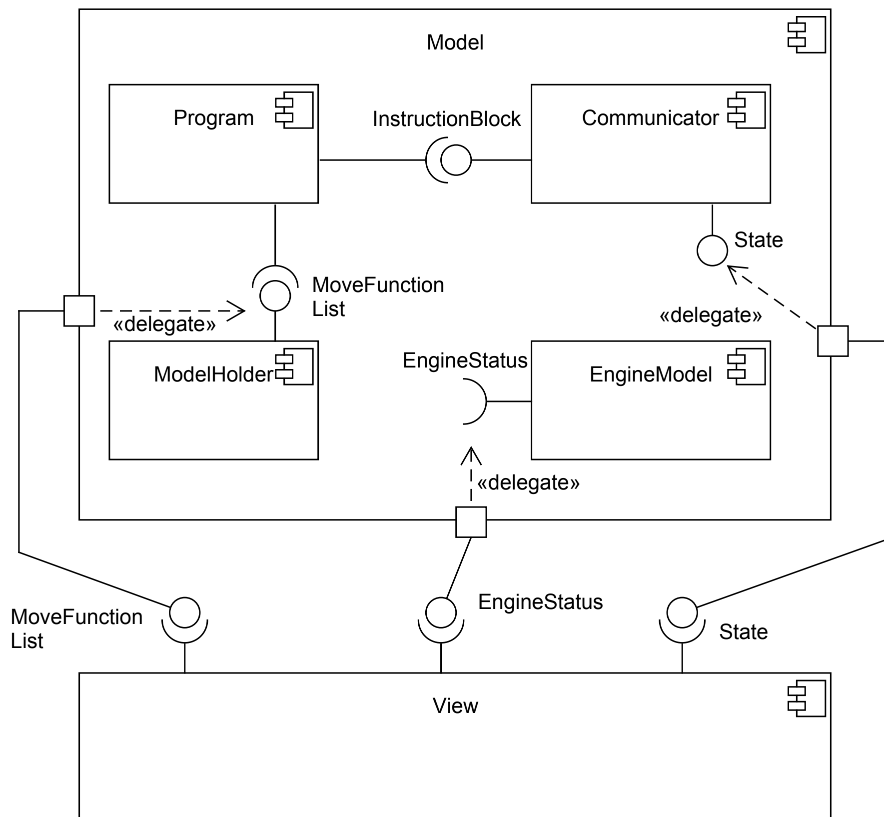
Obrázek 2.3: Podrobný diagram ModelHolderu

EngineStatusWidget Zobrazuje informace od EngineModelu – odhadovanou rychlost, stav magnetické brzdy, stav limit a port, na kterém je aplikace připojena. Jeho součástí jsou i ovládací prvky, které umožní nastavit jednotlivé parametry.

FileListWidget Zobrazuje seznam načtených modelů průběhů, spravované ModelHolderem. Zároveň uživateli zprostředkovává akce, které je možné s modely provádět, jejich načítání, mazání, úpravu pořadí a přepínání zobrazení v MainWidgetu.

MainWidget Tento widget by měl zabírat hlavní část okna. Slouží k zobrazení objektu třídy *PlotWidget*, grafu právě vybraného nebo spuštěného sloučeného průběhu.

Dialogy K editaci nebo generování modelů budou sloužit dialogy, které se zobrazí nezávisle na hlavním okně. Každý dialog obsahuje potřebné grafické prvky sloužící k editaci daného modelu. Data z jeho vstupů se předávají ModelHolderu ke zpracování.



Obrázek 2.4: Diagram hlavních komponent

MainWindow Slouží jako kontejner pro výše uvedené widgety. Stará se o jejich celkové rozložení a zprostředkovává jejich případnou interakci. Navíc ještě zobrazuje a zajišťuje obsluhu akcí hlavního menu aplikace.

2.4 Návrh uživatelského rozhraní

2.4.1 Hlavní okno aplikace

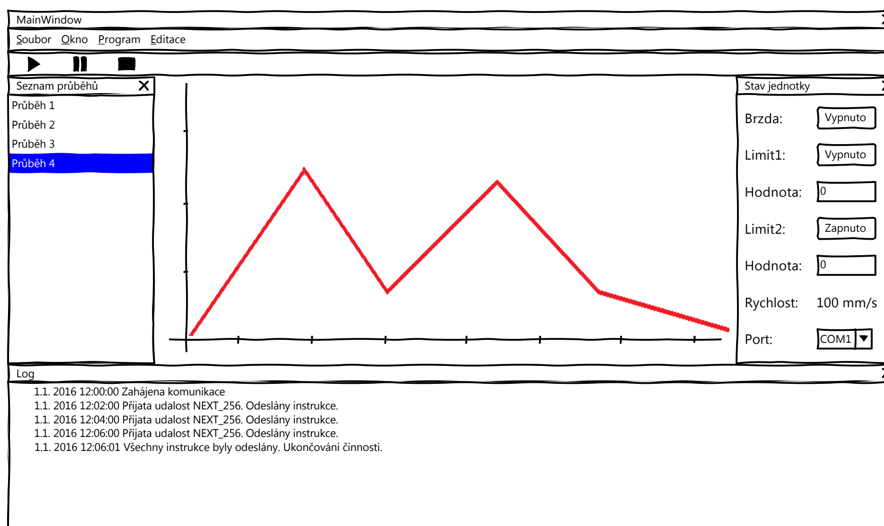
Hlavní okno aplikace bude zajištovat většinu interakce s uživatelem. Insporací pro návrh jeho uživatelského rozhraní byly především aplikace různých simulátorů a vývojových prostředí. Cílem bylo přiblížit se v rámci možností, rozložení a pojmenování prvků, na které jsou uživatelé těchto programů zvyklí.

Hlavní okno lze rozdělit do pěti částí. V každé z těchto částí se nachází jeden z widgetů. V dolní části okna bude umístěn *LogWidget* s textovým zá-

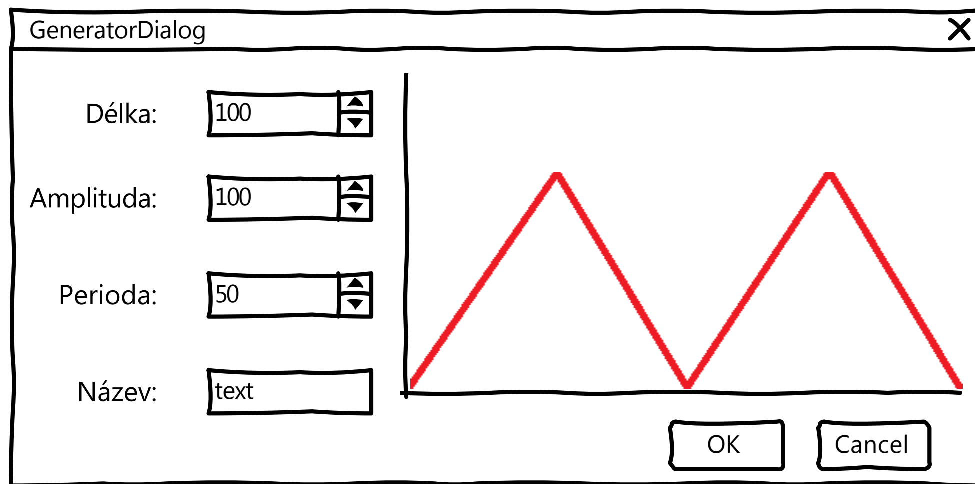
znamem událostí, vlevo seznam jednotlivých prběhů *FileListWidget* a vpravo *EngineStatusWidget* s informacemi o jednotce. Centrální část okna zabírá *MainWidget* s aktuálně vybraným průběhem. V horní části hlavního okna bude menu a nástrojová lišta s nejdůležitějšími akcemi v podobě ikon. Wireframe tohoto rozložení lze vidět na obrázku 2.5. Velikost všech widgetů bude nastavitelná podle potřeb uživatele a všechny kromě centrálního widgetu půjdou skrýt.

2.4.2 Dialogy

Vzhledem k tomu, že ruční kreslení průběhu by pro uživatele bylo poměrně zdlouhavé a zbytečně složité, bude možné v aplikaci generovat několik základních průběhů zadáním jejich parametrů. K tomuto účelu budou sloužit dialogová okna s formulářem, podobná tomu na obrázku 2.6. V levé části budou vstupní pole sloužící k zadání parametrů pro generátor průběhu. Vpravo se zobrazí náhled aktuálně generovaného průběhu, stejně jako v hlavním okně.



Obrázek 2.5: Návrh GUI hlavního okna aplikace



Obrázek 2.6: Návrh GUI generátoru průběhů

2.4.3 Rozdělení akcí

Akce, které může uživatel provádět, jsou podle svého určení rozděleny mezi menu hlavního okna a jednotlivé widgety.

MainWindow Menu volby zobrazené v horní liště okna.

- Soubor

  Otevřít

  Uložit vše

  Konec

- Okno

  Zobrazit/skrýt *LogWidget*

  Zobrazit/skrýt *FileListWidget*

  Zobrazit/skrýt *EngineStatusWidget*

- Program

  Start

  Stop

  Pauza

  Pokračovat

- Editace

    Editovat parametry průběhu

    Generovat průběh

FileListWidget kontextové menu v seznamu průběhů.

- Uložit
- Přejmenovat
- Smazat
- Dump instrukcí

Kapitola 3

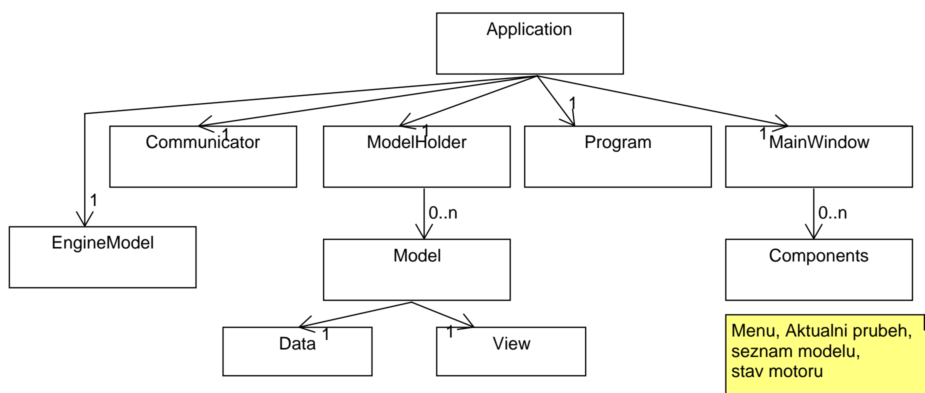
Realizace

3.1 Správa paměti

Jak už bylo řečeno, C++ nechává správu paměti na programátorovi a správnou alokaci a dealokaci objektů na haldě je nutné dělat manuálně. Qt používá pro správu paměti techniku zvanou objektový strom. Ten spočívá ve vytvoření stromové struktury objektů ve vztahu rodič-potomek. Základem je třída QObject, od které dědí všechny ostatní třídy a která poskytuje rozhraní pro udržování seznamu potomků. Strom se poté sám stará o správnou dealokaci objektů. Stačí smazat rodičovský prvek a ten následně rekurzivně smaže všechny své potomky. Zároveň je bezpečné smazat potomka dřív než rodiče. Vazba ve stromu je obousměrná a smazání potomka způsobí jeho odebrání ze seznamu v rodiči [15].

Při implementaci bylo využito tohoto mechanismu uspořádáním vlastnictví objektů do stromové hierarchie podle schématu 3.1. Základní objekt Application vlastní všechny komponenty, které existují po celou dobu běhu aplikace, takže není třeba starat se o jejich životnost. Výjimkou jsou objekty typu *ModelItem*. Ty lze během průběhu přidávat a zase odebírat, a je úkolem ModelHolderu, aby je udržoval. Z praktických důvodů zahrnuje každý *ModelItem* kromě datové definice průběhu i příslušný View, tak aby jeho správu bylo možné provádět spolu s daty. Pouze u právě zobrazeného průběhu je vlastnictví jeho View předáno konteineru MainWindow. Ten pak hlídá jeho navrácení modelu, pokud má dojít ke smazání.

Všechny ostatní vazby mezi objekty potřebné pro vzájemnou komunikaci jsou předávány pomocí ukazatelů, přičemž je ošetřeno buďto z principu daného výše popsanou hierarchií, nebo explicitní kontrolou, že ukazují na platný objekt.



Obrázek 3.1: Hierarchie vlastnictví objektů

3.2 Komunikace mezi komponentami, smyčka událostí

3.2.1 Smyčka událostí

Qt je, podobně jako mnoho jiných grafických frameworků, řízen smyčkou událostí. Události jako uživatelské akce, kliknutí myší nebo požadavek na překreslení okna jsou ukládány do fronty. Odtud je funkce implementující smyčku čte a vykonává je [16].

3.2.2 Signály a sloty

Qt používá koncept signálů a slotů pro komunikaci mezi objekty. Objekty definují signály, které emitují pokud chtějí informovat ostatní objekty o změně svého stavu. Zároveň definují sloty – funkce, které lze se signály propojit. Jde o Qt verzi implementace už zmíněného návrhového vzoru *observer* a callback funkcí. Generátor *MOC* vygeneruje před samotnou kompilací mapovací tabulku, která k signálům zajistí volání příslušných slotů [14]. Toto propojení má jednu výhodu v tom, že pokud objekt vysílající sigál, patří do jiného vlákna než objekt připojeného slotu, nedojde k zavolání funkce slotu rovnou, ale přes událost zpracovávanou smyčkou událostí cílového vlákna. Tím odpadá nutnost řešit synchronizaci mezi vlákny a zamykání – volání je asynchronní [17].

3.2.3 Použití v aplikaci

Aplikace využívá těchto konceptů pro komunikaci mezi jednotlivými komponentami. Ovládací prvky GUI emitují signály při uživatelských akcích jako

je kliknutí na tlačítko nebo změna hodnoty ve vstupním poli. Signály jsou napojeny na sloty příslušných modelů a umožňují tak měnit jejich stav. Zároveň všechny komponenty definují sloty *Start, Stop, Pause, Resume*, přes které reagují na změnu stavu vykonávání programu v řídící jednotce.

Komponenta *Communicator* má vlastní vlákno se smyčkou událostí, která zpracovává příjem a odesílání dat přes sériovou linku. S GUI vláknem, které obstarává vykreslování okna a obsluhu uživatelských akcí, komunikuje právě pomocí signálů a slotů. Jak již bylo řečeno, Qt zajišťuje synchronizaci propojení mezi dvěma vlákny, díky čemuž odpadá potřeba použití zámků.

3.3 Formát vstupních dat

Jedním z požadavků popsaných v části 1.3.2 je možnost načítat průběhy funkcí z textových souborů. Pro tyto soubory byla zvolena varianta textového formátu CSV[5] [18], který lze editovat většinou tabulkových procesorů nebo v jakémkoliv textovém editoru.

Vstupní data musí splňovat následující předpis:

```
<opakovani>;<nasobeni>
x1;y1
x2;y2
...
xn;yn
```

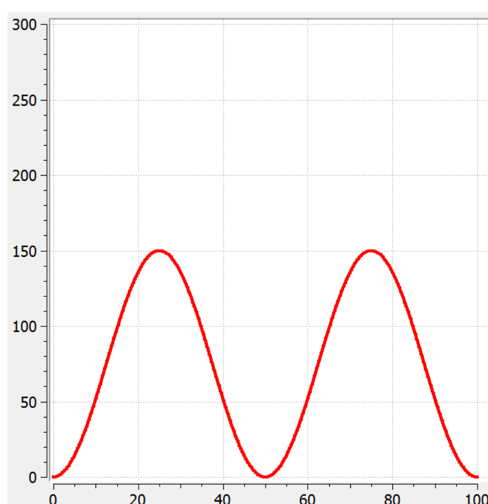
Kde <opakovani> a <nasobeni> jsou celá, kladná čísla. První z nich určuje počet opakování zadaného průběhu, druhým se násobí y hodnota jednotlivých bodů a lze tak měnit amplitudu periodických průběhů.

Poté následují jednotlivé body, které definují samotnou funkci. Na každém řádku je právě jeden bod, přičemž x a y jsou reálná čísla udávající čas v sekundách (x) a polohu motoru (y) jako vzdálenost od počátku v mm. Body zároveň musí tvořit rostoucí posloupnost vzhledem k souřadnici x.

3.4 Generování a editace průběhů

Aplikace obsahuje generátory tří základních průběhů – Sinusoidy, pily a lineární funkce. Každý generátor se zobrazuje ve vlastním dialogovém okně, kde lze nastavit jeho parametry.

[5] Comma separated values. Hodnoty oddělené čárkou.



Obrázek 3.2: Ukázka průběhu sinusoidy

3.4.1 Sinusoida

Tento generátor vytváří průběh podle funkce sinus. Má tři vstupní parametry. Délku l, periodu p, amplitudu Amp , ze kterých generuje $l * f_s$ bodů podle tohoto vzorce [19]:

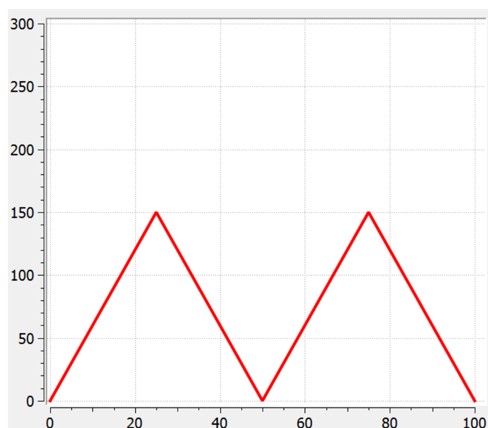
$$x_i = i/f_s$$
$$y_i = \frac{Amp}{2} + \sin((2\pi f x_i) - \frac{\pi}{2}) * \frac{Amp}{2}$$
$$A_i = (x_i, y_i)$$

Proměnná f_s je stanovena napevno na hodnotu 100 a označuje vzorkovací frekvenci, tedy počet bodů, které připadnou na jednu sekundu. Celá křivka je posunuta o $Amp/2$ směrem nahoru, aby se tak vykompenzovaly záporné hodnoty, kterých sinus nabývá na některých intervalech, protože pozice motoru záporná být nemůže. Zároveň dochází k fázovému posunu o $\frac{\pi}{2}$, díky tomu je první vygenerovaný bod (0,0).

3.4.2 Pila

Pila je periodická funkce, která popisuje střídavý pohyb motoru dopředu a následně zpět do výchozí pozice lineární rychlostí.

Její generátor používá, stejně jako generátor sinusoidy, délku l, periodu p a amplitudu Amp jako vstupní parametry. Z těch vygeneruje posloupnost $N+1$



Obrázek 3.3: Ukázka průběhu pily

bodů $A_0, A_1 \dots A_n$, pro které platí:

$$N = 2 * l/p$$
$$A_i = \begin{cases} (i * p/2, 0) \text{ pro } i \text{ sudé} \\ (i * p/2, Amp) \text{ pro } i \text{ liché} \end{cases}$$

Pokud je souřadnice x posledního bodu větší než l, je poslední bod A_n nahrazen bodem B pomocí rovnice lineární funkce:

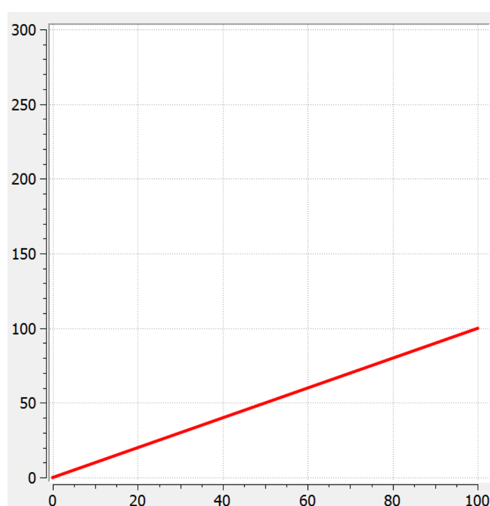
$$x_1 = A_{n-1}.x$$
$$y_1 = A_{n-1}.y$$
$$y = y_1 + (A_n.y - y_1)/(A_n.x - x_1) * (l - x_1)$$
$$B = (l, y)$$

3.4.3 Lineární funkce

Lineární funkce představuje jednoduchý pohyb motoru pouze jedním směrem. Funkce je určena třemi parametry – délkou l, a počáteční pozicí $y1$ a koncovou pozicí a $y2$. Z těchto parametrů je generátorem vytvořen průběh funkce jako dvojice bodů $((0, y1), (l, y2))$.

3.4.4 Ruční zadávání instrukcí

Implementovaný překladač pracuje pouze s funkcí pohybu motoru, kterou překládá na sekvenci instrukcí `MOVE` a `WAIT`. Některé další instrukce jsou použity uvnitř aplikace v reakci na uživatelské akce, např. `RESUME` pro spuštění a `FLUSH` pro zastavení programu. Některé instrukce ale nejsou tímto způsobem



Obrázek 3.4: Ukázka lineárního průběhu

vůbec implementovány. Protože by jejich použití mohlo být někdy užitečné, existuje v aplikaci možnost zadat instrukce přímo. Slouží k tomu vlastní dialogové okno s tabulkou, do které lze instrukce zapisovat.

Instrukce lze také ukládat a načítat z textového souboru v následujícím formátu:

```
<nazev_instrukce1> op1 op2
<nazev_instrukce2> op1 op2
 ...
<nazev_instrukceN> op1 op2
```

Op1 a Op2 jsou nepovinné hodnoty operandů zadané jako celé číslo v desítkové soustavě, <nazev_instrukce1> je textový název instrukce podle tabulky v příloze A.

3.4.5 Validace průběhů

Aby nemohla nastat situace, že motor dostane příkaz k pohybu, který nemůže vykonat, provádí se před každým spuštěním programu validace modelu, která kontroluje, zda průběh splňuje omezení dané vlastnostmi motoru. Pro každý bod průběhu validátor zkontroluje, zda je jeho souřadnice y uvnitř rozsahu intervalu pohybu <0;300> mm. Zároveň pro každé dva sousední body kontroluje požadovanou rychlost, zda nepřekračuje maximální rychlost motoru.

3.4.6 Spojování a opakování průběhů

Jak už bylo řečeno, jednotlivé průběhy lze zadávat a upravovat odděleně. Před spuštěním programu pak kromě vygenerování kódu dojde ke spojení všech průběhů do jednoho, který se zobrazí uživateli. Průběhy jsou spojeny v pořadí, ve kterém jsou uloženy v *ModelHolderu*. To lze měnit v *FileListWidgetu* pomocí operace „táhni a pusť".

Funkce spojení, podobně jako překladač, chápe vzdálenosti ve dvou po sobě následujících modelech průběhu jako relativní. To znamená, že všechny body modelu $N+1$ jsou posunuty o hodnotu posledního bodu průběhu N. Pokud není souřadnice y tohoto bodu rovna 0, aplikace na to uživatele upozorní, ale je jeho starostí, aby rozhodl, zda je to správně a případně data opravil. Zároveň platí, že pokud není první souřadnice y navazujícího úseku průběhu rovna 0, je celý tento průběh posunut směrem dolů do 0. Průběhy s opakováním se tohoto spojení účastní vícekrát. Každý takový průběh je připojen sám za sebe tolikrát, kolikrát se má opakovat. Výsledkem je tedy jeden velký model průběhu bez opakování. Celý postup je shrnut pseudokódem 3.1.

3.5 Překlad průběhů na instrukce

Aby bylo možné zadané průběhy spustit, je nezbytné vygenerovat z nich sekvenci instrukcí `MOVE` a `WAIT`. Jak již bylo řečeno v části 1.4.3, instrukce mají celočíselné operandy, které jsou v tomto případě vázány přímo k vlastnostem motoru, případně řídící jednotky.

Instrukce jsou v aplikaci reprezentovány následující strukturou:

```
struct Instruction
{
        OpCode code;
        int32_t op1;
        uint32_t op2
}
```

OpCode je *enum* typ pro operační znak instrukce.

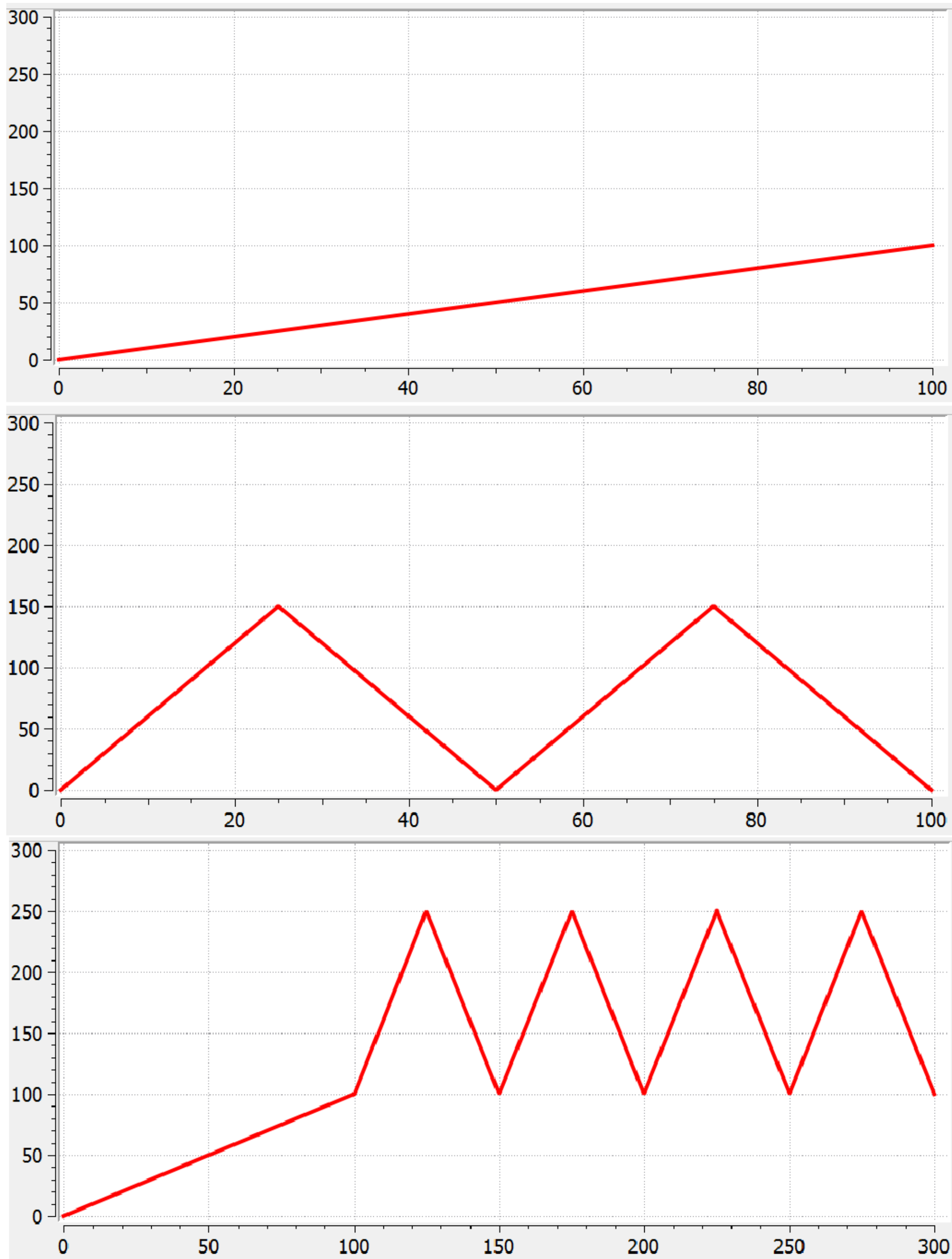
Překladač prochází postupně všechny body a pro každé dva následující spočítá rozdíl jejich souřadnic. Výsledkem je vzdálenost, kterou má motor urazit a doba trvání tohoto pohybu. Mohou nastat tři situace:

- Výsledná vzdálenost je kladná. Motor se bude pohybovat dopředu a vygeneruje se `MOVE`.

- Vzdálenost je záporná. Opět se vygeneruje `MOVE` a motor se pohybuje vzad.

```
MoveFunction concat(MoveFunction fn1, MoveFunction fn2)
{

Point offset;
Point negativeOffset;

offset = fn1.last();
int originalSize = fn1.size();
for (int i = 1; i < fn1.repeats(); i++)
{
  for (int j = 0; j < originalSize; j++)
  {
    Point pt = fn1.data[j];
    pt += offset;

    if (fn1.size() == 0 || pt!=_data.last())
          fn1.data.add(pt);
  }
  offset = fn1.data.last();
}
negativeOffset = Point(0, 0) - fn2.data[0];

for (int i = 0; i < fn2.repeats(); i++)
{
  for (int j = 0; j < fn2.size(); j++)
  {
    Point pt = fn2.data[j];
        pt += offset+negativeOffset;

        if (fn1.size() == 0 || pt != fn1.data.last())
              fn1.data.add(pt);
  }
offset = fn1.data.last();
}
return fn1;
}
```

Algoritmus 3.1: Spojení dvou průběhů



Obrázek 3.5: Spojení dvou průběhů s opakováním

- Vzdálenost je nula. Překladač generuje `WAIT`. Pokud má čas čekání přesáhnout 80 s, generuje se více `WAIT` za sebou, aby nedošlo k přetečení operandu.

Následně se vypočítají jednotlivé operandy. Vzdálenost se přepočítá na počet kroků motoru a čas na počet taktů řídící jednotky provedených za jeden krok takto:

$$
\begin{aligned}
STEPSMM &= 100 \\
CLOCK &= 50000000 \\
Steps &= (B.y - A.y) * STEPSMM \\
Time &= (B.x - A.x) * CLOCK
\end{aligned}
$$

Z těchto hodnot je pomocí konstruktoru vytvořena instance *Instruction* jako `Instruction(MOVE,Time/Steps,Steps)` nebo `Instruction(WAIT,Time)`.

```
List<Instruction> translate(MoveFunction fn)
{
List<Instruction> ins;
STEPS_PER_MM = 100;
CLOCK = 50000000;
MAX_DISTANCE = 300;

for(auto i = 0; i < fn.size()-1; i++)
{
        Point a = fn.at(i);
        Point b = fn.at(i+1);

        if (b.y>Engine::MAX_DISTANCE || b.y < 0)
                throw Exception;

        Float time = b.x - a.x;
        Integer ay = (a.y * STEPS_PER_MM);
        Integer by = (b.y * STEPS_PER_MM);

        Integer steps = by - ay;
        if (steps == 0)
        {
                List<Instruction> ret = generateWaits(time);
                ins.concat(ret);
        }
        else
        {
                Integer interval = time * CLOCK / abs(steps);
                ins.add(Instruction(MOVE, interval, steps));
        }
}

return ins;
}
```

Algoritmus 3.2: Překlad na instrukce

3.5.1 Spouštění programů

Spuštění zadaného průběhu je realizováno v několika krocích. Nejprve se spojí jednotlivé průběhy v pořadí, jaké mají v *ModelHolderu*. Výsledný průběh je poté zvalidován a pokud nesplňuje podmínky uvedené v části 3.4.5, spuštění

```
List<Instruction> generateWaits(Float time)
{
List<Instruction> ins;
Integer MAX_SEC = 80;
Integer CLOCK = 50000000;

Integer steps = time / MAX_SEC;

for(int i = 0; i < steps; i++)
        ins.add(Instruction(WAIT, CLOCK*MAX_SEC));

Float resttime = time - steps * MAX_SEC;
ins.add(Instruction(WAIT, CLOCK*resttime));

return ins;
}
```

Algoritmus 3.3: Generování WAIT

se zastaví a uživatel dostane příslušnou chybovou hlášku. V opačném případě komponenta *Program* zavolá na každý model kompilátor, který vygeneruje potřebné instrukce. Toto generování neprobíhá ze spojeného průběhu, protože jsou do něj zahrnuty i modely, které byly vytvořeny přímým zadáním instrukcí a nemusí tedy obsahovat instrukce pro pohyb.

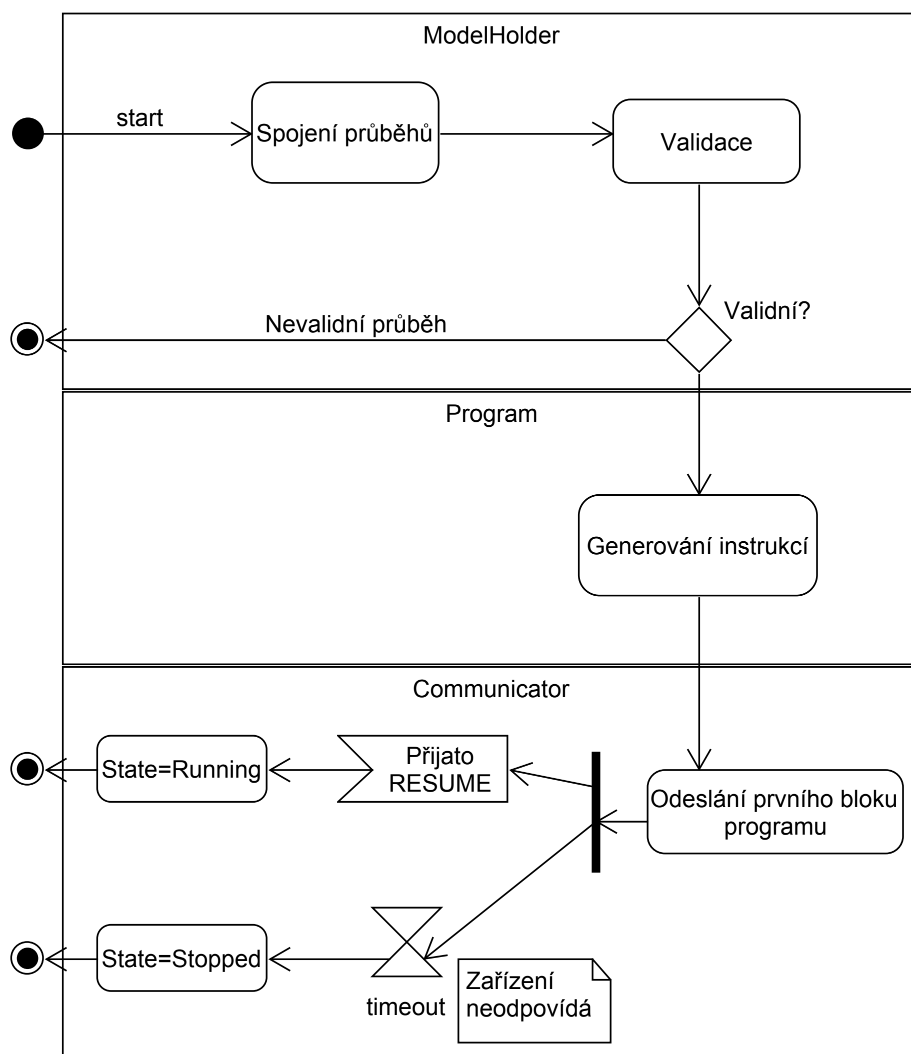
Poté je aktivován *Communicator*, který si vyžádá od *Programu* prvních 384 instrukcí, které odešle řídící jednotce. Následně se odešle ještě `RESUME`[6]. Jakmile je přijata odpověď a potvrzeno spuštění motoru, aktivuje se *Timer*, který s intervalem 500 ms přepočítává odhadovanou polohu a rychlost motoru. Poté už probíhá normální komunikace.

3.6 Komunikace s řídící jednotkou

3.6.1 Knihovna QSerialPort

Pro implementaci komunikace byla použita knihovna QSerialPort, která je součástí Qt. Její stejnojmenná hlavní třída umožňuje čtení a zápis na vybraném portu v blokujícím a neblokujícím režimu [20]. Při blokujícím režimu je volající vlákno při čtení zablokováno dokud, se na portu neobjeví data, nebo dokud nevyprší zadaný čas. Tento režim tedy nelze použít v hlavním vlákně aplikace, protože by to znamenalo zablokování uživatelských akcí a GUI. Na-

[6]Resume se odesílá vždy jako poslední aby nebyla prázdná fronta při spuštění motoru.



Obrázek 3.6: Diagram procesu spuštění programu

opak při neblokujícím režimu se využívá smyčky událostí Qt a signálů a k nim připojených slotů, které se volají pouze, pokud je co číst.

3.6.2 Communicator

Komponenta Communicator se stará o komunikaci mezi řídící jednotkou a aplikací. Tu realizuje pomocí objektu třídy *QSerialPort* s neblokujícím voláním. Aby nebyla komunikace případně zdržována obsluhou GUI akcí, má Communicator vlastní vlákno se smyčkou událostí. Ty jsou obsluhovány objektem třídy *CommWorker*. Obsluha těchto událostí se liší v závislosti na stavech, které jsou popsány dále.

3.6.3 Komunikační stavy

Komunikace mezi aplikací a řídící jednotkou je řízena stavovým automatem, který je znázorněn diagramem na obrázku 3.7. Jednotlivé stavy definují chování aplikace vzhledem k událostem, které generuje řídící jednotka a nebo vzhledem k uživatelským vstupům. Zároveň poskytují zpětnou vazbu skrz uživatelské rozhraní.

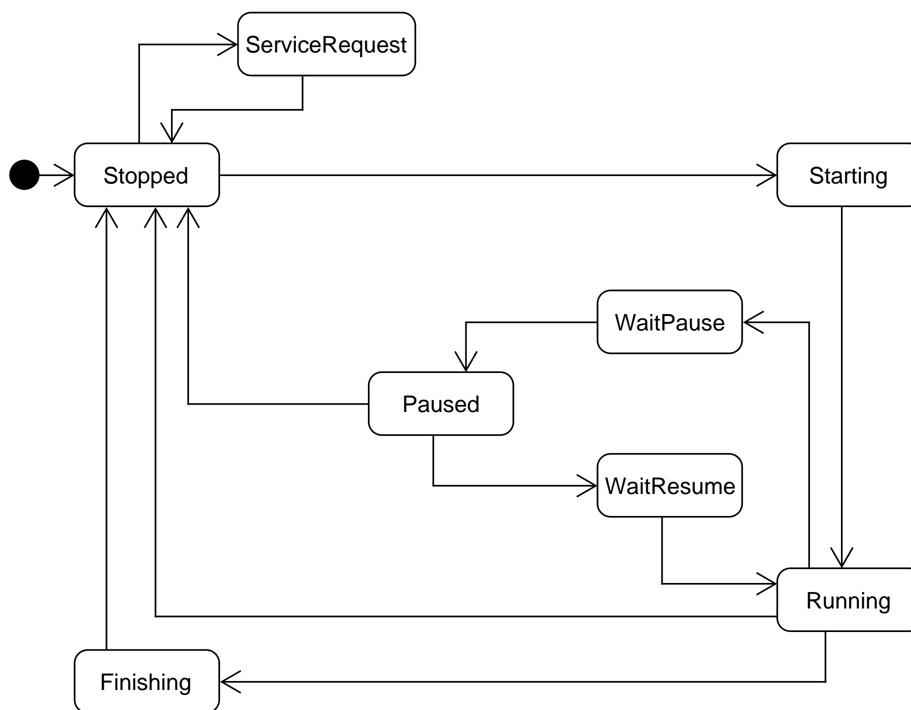
3.6.4 Hlavní stavy

Stavy `Stopped`, `Paused`, `Running` a `ServiceRequest` reprezentují stav motoru a řídící jednotky.

Stopped představuje stav, při kterém neprobíhá žádná komunikace s řídící jednotkou, její instrukční fronta je prázdná a tím pádem i samotný motor stojí. V tomto stavu se aplikace nachazí po svém spuštění nebo poté, kdy uživatel zmáčkne tlačítko stop.

Running V tomto stavu probíhá komunikace s řídící jednotkou, která vykonává zadaný program a motor běží. Aplikace reaguje na události generované řídící jednotkou a postuppně odesílá jednotlivé bloky instrukcí. Po odeslání všech instrukcí přejde aplikace do stavu `Finishing`. Stisknutí tlačítka Pauza způsobí přechod do stavu `WaitPause` a stisknutím tlačítka Stop přejde aplikace do stavu `Stopped`.

Paused Během tohoto stavu motor stojí, ale instrukční fronta není prázdná. Poté, kdy uživatel stiskne tlačítko Pokračovat, vrátí se aplikace odesláním instrukce `RESUME` přes stav `WaitResume` do stavu `Running`, motor se uvede do chodu a řídící jednotka pokračuje ve vykonávání programu od místa, ve kterém byla zastavena.



Obrázek 3.7: Diagram stavů komunikace

ServiceRequest Tento stav představuje zkratku cyklu `Stopped` - `Starting` - `Running` - `Finishing` - `Stopped` s tím rozdílem, že řídící jednotka nevykonává žádné instrukce pro pohyb motoru. Místo toho je jednotce odeslána následující posloupnost instrukcí:

```
<Instrukce>
STATUS_REQUEST
RESUME
```

Kde `<Instrukce>` představuje jednu z instrukcí, které nastavují parametry řídící jednotky a motoru. Po přijetí odpovědi na `STATUS_REQUEST` se odešle ještě `FLUSH` a aplikace přejde zpět do stavu `Stopped`.

3.6.5 Přechodné stavy

Přechodné stavy představují situace, které nastanou většinou po nějaké uživatelské akci. Typicky například spuštění nebo pauza programu. Aplikace odešle příslušné instrukce řídící jednotce a následně čeká v přechodném stavu na

odpověď. Její přijetí znamená potvrzení, že jednotka vykonala danou instrukci a následuje přepnutí do jednoho z hlavních stavů.

Starting Po spojení s řídící jednotkou je odeslán první blok instrukcí a následně instrukce `RESUME`. Jakmile aplikace přijme odpověď, přejde odsud do stavu `Running`.

Finishing Pokud aplikace obdrží od řídící jednotky událost `NEXT_256` a nemá už dostatek instrukcí k odeslání, znamená to, že program bude dokončen. Odešle proto ještě instrukci `ECHO` a přepne se do stavu `Finishing`. Během něj čeká na odpověď této instrukce, která znamená dokončení programu. Následně odesílá `FLUSH` a přechází do stavu `Stopped`.

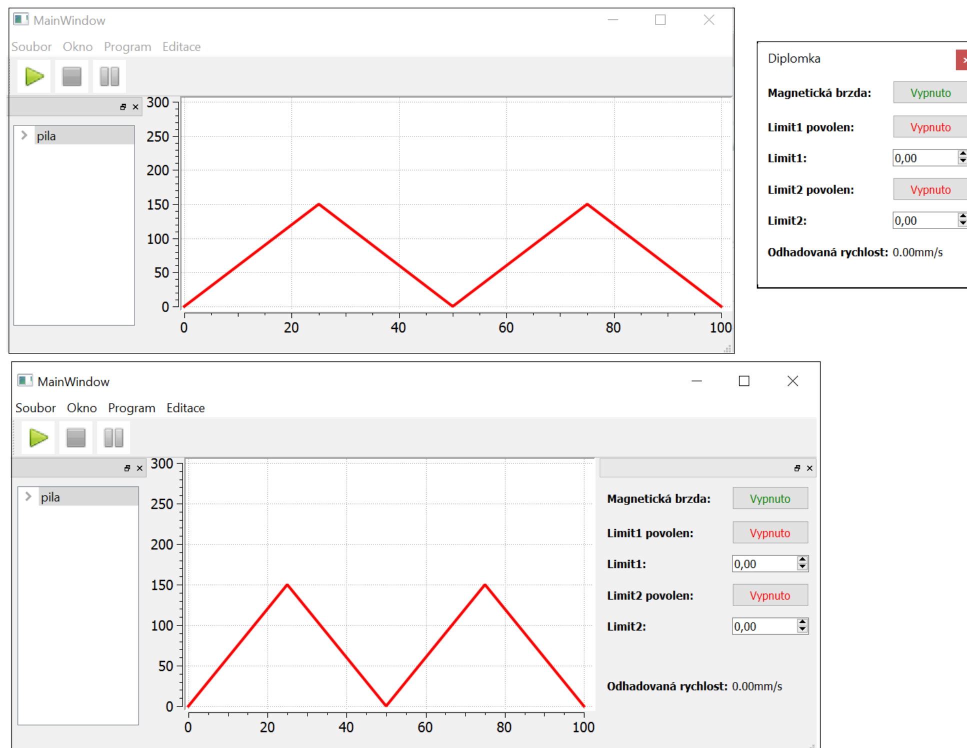
WaitPause Tento stav tvoří přechod mezi stavem `Running` a `Pause`. Aplikace odešle řídící jednotce instrukci `HALT` a čeká na potvrzovací odpověď. Poté se přepne do stavu `Paused`.

WaitResume je opakem stavu `WaitPause` a tvoří tedy přechod mezi `Pause` a `Running`. Aplikace zde čeká na odpověď instrukce `RESUME`.

3.7 View

3.7.1 Hlavní okno

Rozložení hlavního okna se prakticky neliší od návrhu na obrázku 2.5. Jedniným rozdílem je využití Qt kontejneru jménem *QDockWidget* pro umístění jednotlivých widgetů. Jeho použití umožňuje variabilitu uživatelského rozhraní podle potřeby uživatele. S *QDockWidgetem* lze volně pohybovat pomocí myši, ukotvit ho k jednomu z okrajů hlavního okna, nebo ho nechat zobrazit jako samostatné okno [21].



Obrázek 3.8: Použití QDockWidgetu

3.7.2 PlotWidget

Graf průběhu funkce zobrazený v centrální části hlavního okna. Je vytvořen s použitím knihovny Qwt. Jde o externí knihovnu, která není součástí hlavní distribuce Qt a je mimo jiné určena pro tvorbu 2D grafů [22].

3.7.3 Informace o běžícícm průběhu

Pokud je řídící jednotka aktivní, zobrazuje se v okně aplikace kromě grafu průběhu i odhadovaná pozice a rychlost motoru. Obě tyto hodnoty jsou počítány lokálně vzhledem k nemožnosti získat přesné údaje z jednotky. Aktualizace těchto hodnot se provádí vždy jednou za sekundu na základě události vygenerované objektem třídy *QTimer*. Aktuální poloha je potom zobrazena jako svislá čára v grafu průběhu. Odhadovaná rychlost se zobrazuje v panelu *EngineStatusWidget*

```
Double speedAt(MoveFunction fn, Double time)
{
if (fn.size()==0 || time<fn.data[0].x
        || time >= fn.data.last().x)
        return 0;

Integer mid = fn.lastSpeedIndex;
Integer max = fn.size();
Integer min = 0;

if(fn.data[mid].x >= time && time < fn.data[mid+1].x)
  speedBetween(fn.data[mid],fn.data[mid+1])

while (min <= max)
{
  if (fn.data[i].x < time)
    min = mid + 1;
  else
        max = mid - 1;
  i = (min + max) / 2;
}
if (max < 0)
  max = min;

fn.lastSpeedIndex = min;

return speedBetween(fn.data[min],fn.data[max]);
}
```

Algoritmus 3.4: Algoritmus výpočtu rychlosti

3.7.4 Odhad rychlosti

Odhad rychlosti se provádí metodou *speetAt(time)* třídy *MoveFunction*, jejíž instance tvoří model průběhu. Metoda vyhledá dva body, jejichž souřadnice x se nejvíce blíží hodnotě parametru *time* a následně spočítá rychlost mezi nimi. Protože se předpokládá, že metoda bude volána opakovaně s pravidelným intervalem, pamatuje si *MoveFunction* poslední vrácený index. Je totiž pravděpodobné, že při dalším volání se budou odpovídající body nacházet na následujícím indexu. Pokud tomu tak není, je použit upravený algoritmus binárního vyhledávání [23], který vrací dva nejbližší body.

Kapitola **4**

Testování

Pro ověření fungování aplikace byla navržena série testů s různou úrovní závislosti na spolupráci s řídící jednotkou a motorem. Nejprve se začalo s jednoduchými jednotkovými testy překladače a editoru průběhů, po kterých následovalo testování správné implementace komunikace po sériovém portu. Jako poslední přišel test s řídící jednotkou.

4.1 Test překladače a editoru

K prvotnímu otestování posloužilo několik testovacích scénářů, které lze provést bez nutnosti komunikace se zařízením.

- Test načítání a ukládání – testuje implementaci funkcí pro čtení a zápis průběhů do souboru. Zdrojový a výsledný soubor se musí pro jeden průběh shodovat.
- Test generování instrukcí – instrukce vygenerované ze zadaného souboru musí odpovídat zadanému vzoru.
- Test instrukce `WAIT` – testuje zda čekání delší než 80 s vygeneruje více `WAIT` instrukcí.
- Test spojení průběhů – testuje spojení dvou průběhů oproti předem zadanému výsledku.

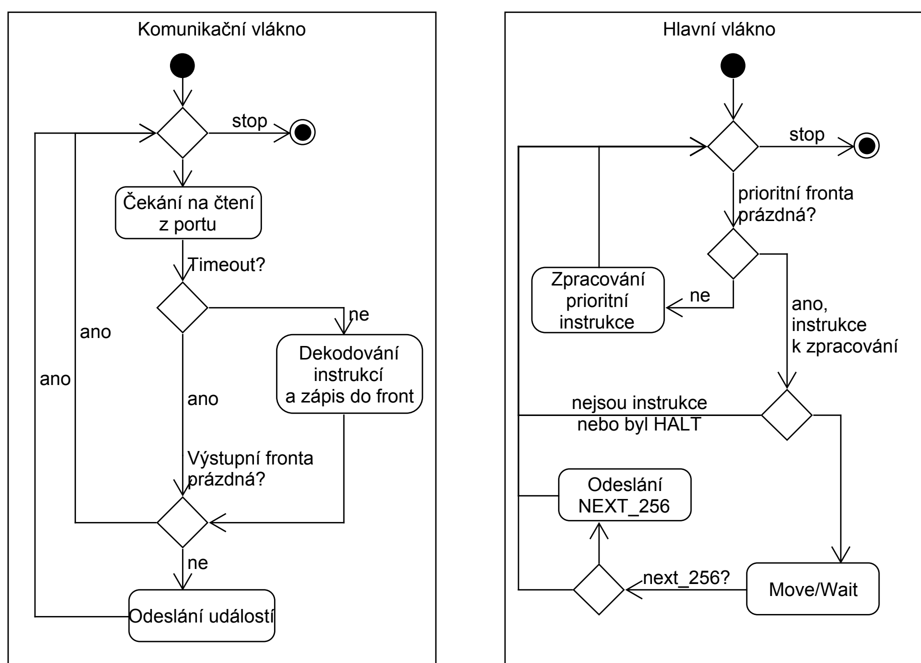
4.2 Softwarová simulace

Jako podpora testování byl vytvořen softwarový simulátor řídící jednotky a motoru. Ten byl použit k otestování správné implementace komunikace pomocí sériového portu a vyladění aplikace před testováním s reálným hardwarem, nebo k testům ve chvílích, kdy nebyl hardware k dispozici.

4.2.1 Popis simulátoru

Simulátor je v Qt napsaná konzolová aplikace a funguje odděleně od hlavního programu. Při spuštění očekává jako parametr název sériového portu. Svou činnost zaznamenává na standardní výstup. K jeho použití je nutné mít dva vzájemně propojené UART převodníky zapojené do USB portů počítače.

Simulátor funguje na principu producent-konzument [24]. Tento princip spočívá ve spolupráci dvou vláken, kdy jedno, "producent", zapisuje data do zásobníku/fronty a druhé, "konzument", je čte. V tomto případě hlavní vlákno programu jakožto konzument simuluje instrukční cyklus řídící jednotky. Druhé, komunikační vlákno poslouchá na sériovém portu, čte a dekóduje z něj data. Vlákna spolu komunikují prostřednictvím čtení a zápisu do tří front. Dvě z nich slouží jako instrukční fronty rozdělené pro prioritní a neprioritní instrukce. Třetí je výstupní fronta, do které se zapisují události a odezvy instrukcí, které poté komunikační vlákno odesílá sériovým portem. Přístup k frontám je chráněn mutexy.



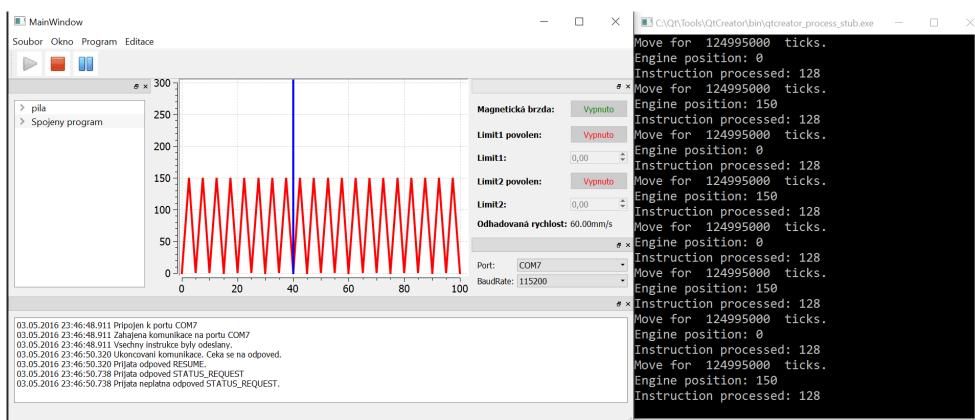
Obrázek 4.1: Schéma činnosti simulátoru

Protože hlavní vlákno funguje jako simulátor řídící jednotky, nedochází k jeho uspání pokud je fronta prázdná. Namísto toho pouze projde `IDLE` sta-

vem instrukčního cyklu. Tím pádem nemusí řešit problém uváznutí producenta a konzumenta [24].

V simulátoru jsou naimplementovány pouze základní instrukce `MOVE`, `WAIT`, `HALT`, `RESUME`, `FLUSH`, `BREAK_EN`, `BREAK_DIS`, jejich příslušné odpovědi a události `NEXT_256` a `QUEUE_OVERLOADED`.

Doba vykonávání instrukcí v simulátoru neodpovídá přesně specifikaci řídící jednotky. K simulaci `MOVE`, `WAIT` je použito volání funkce *sleep_for* [25] s argumentem doby uspání v nanosekundách, nicméně skutečná délka uspání vlákna je závislá na rozlišení časových intervalů plánovače operačního systému. U běžných desktopových OS a obzvláště Windows je tento interval řádově větší [26]. Vykonání většiny instrukcí simulátorem tedy trvá déle než ve skutečnosti. Na druhou stranu se jeho výstup dá lépe sledovat za běhu.



Obrázek 4.2: Testování aplikace se simulátorem

4.3 Test s hardwarem

Protože se s Fakultou strojní ČVUT nepodařil domluvit přístup do laboratoří, nebylo možné provést test reálného pohybu motoru. Tento test tedy proběhl pouze s řídící jednotkou implementovanou na FPGA čipu Spartan 3.

S ohledem na absenci motoru bylo nutné pozměnit test instrukcí `MOVE` a `WAIT` a zaměřit se hlavně na testování ovládání řídící jednotky, tedy na práci s instrukcemi jako `STATUS_REQUEST`, `SET_LIMIT_1`, `EN_LIMIT_1` a dalšími, které nejsou implementovány v simulátoru.

4.4 Výsledky testů

4.4.1 Zjištěné problémy

Překrývání operačních kódů Během testů se simulátorem a řídící jednotkou se zjistilo, že návratové hodnoty některých instrukcí v tabulce 1.3 se překrývají s kódy událostí 1.5. To způsobovalo problém v komunikačním cyklu, kdy namísto na očekávané odpovědi instrukcí, reagovala aplikace na událost se stejným operačním kódem.

Zamrznutí Communicatoru Při testu ovládání magnetické brzdy a nastavení limit docházelo v některých případech k zamrznutí Communicatoru ve stavu `ServiceRequest`. Příčinou byl rozdíl v rychlosti vykonávání instrukcí, komunikace po sériové lince a zpracování přijatých dat. Výsledkem bylo přijetí několika odpovědí naráz, s čímž implementace jejich zpracování *Communicatorem* nepočítala.

4.4.2 Změny implementace na základě testů

Oprava operačních kódů Po konzultaci s vedoucím práce se ukázalo, že překrývání operačních kódů v tabulkách 1.3 a 1.5 je pouze chyba v textu původní práce a v řídící jednotce jsou tyto kódy naimplementovány správně. Následně se podařilo získat jejich správnou definici a provést příslušné opravy v aplikaci i simulátoru. Správné kódy všech odpovědí jsou v tabulkách přílohy B.

Úprava Communicatoru Zpracování přijatých událostí bylo upraveno tak, aby *Communicator* kontroloval a zpracovával všechny přijaté bajty.

Závěr

Tato práce měla za cíl zanalyzovat chování řídící jednotky lineárního motoru a na základě této analýzy navrhnout a implmentovat desktopovou obslužnou grafickou aplikaci, která by pro jednotku umožnila zadávat a spouštět uživatelské programy bez potřeby hlubší znalosti instrukční sady řídící jednotky.

Výsledkem je aplikace napsaná pomocí C++ frameworku Qt. Je navržena na základě Qt modifikace návrhového vzoru MVC zvané Model-View, která odděluje data a logiku od zobrazení. Aplikace umožňuje zadávat a spouštět pohyby motoru pomocí textových souborů s definicí funkce pohybu v podobě jednotlivých bodů. Zároveň umožňuje generovat jednoduché průběhy přímo, zadáním několika základních parametrů. Zadané průběhy lze modifikovat a spojovat do větších celků. Pomocí knihovny Qwt je pak realizováno zobrazení jednotlivých pohybových funkcí v podobě planárního grafu závislosti polohy a času. Aplikace také zobrazuje některé parametry řídícího motoru a jednotky a umožňuje jejich nastavení. K tomu využívá vnitřní překladač, který generuje požadovné bloky instrukcí.

S řídící jednotkou komunikuje aplikace pomocí sériového portu. Obsluha této komunikace je realizována na základě stavového automatu v odděleném aplikačním vláknu, takže se neovlivňuje s událostmi uživatelského rozhraní.

Výslednou aplikaci se podařilo otestovat na základě několika jednotkových testů a také oproti simulátoru řídící jednotky, který vznikl jako doplňková konzolová aplikace a implementuje pouze základní instrukce. Dále proběhlo testování spolupráce přímo s řídící jednotkou na FPGA čipu Spartan 3. Bohužel se nepodařilo zajistit spolupráci s Fakultou strojní a otestovat fungování aplikace přímo s motorem v reálném provozu.

Toto testování je tedy dalším logickým krokem pro rozvoj aplikace. Pokud se ho podaří realizovat, lze uvažovat o jejím dalším rozšíření. Nabízí se například

možnost vytváření složitějších modelů přímo, vylepšení paměťové náročnosti pomocí vnitřní komprese, nebo zobrazení doplňkových dat z uživatelských vstupů a dalších připojených čidel.

Literatura

[1] Bartík, M.: *Návrh a vývoj pokročilé řídící jednotky lineárního motoru pro přesná laboratorní měření v biomechanice*. Diplomová práce, ČVUT FIT, 2014.

[2] Motor, O.: Motorized Cylinders EZ Limo EZCII Series. online.

[3] Axelson, J.: *Serial Port Complete: COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems*. Lakeview Research, druhé vydání, 2007.

[4] TIOBE Index. duben 2016. Dostupné z: `http://www.tiobe.com/tiobe_index`

[5] Java Communications API. 2016. Dostupné z: `http://www.oracle.com/technetwork/java/index-jsp-141752.html`

[6] Serial Port - class. duben 2016. Dostupné z: `https://msdn.microsoft.com/cs-cz/library/system.io.ports.serialport(v=vs.110).aspx`

[7] Boost C++ Libraries. 2016. Dostupné z: `http://www.boost.org/`

[8] Boost Softwre License. 2016. Dostupné z: `http://www.boost.org/users/license.html`

[9] Overview - wxWidgets. 2016. Dostupné z: `http://wxwidgets.org/about/`

[10] About Qt. duben 2016. Dostupné z: `https://wiki.qt.io/About_Qt`

[11] Using the Meta-Object Compiler (moc). 2016. Dostupné z: `http://doc.qt.io/qt-5/moc.html`

[12] Erich Gamma, R. J. J. V., Richard Helm: *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[13] Model/View Programming. duben 2016. Dostupné z: http://doc.qt.io/qt-5/model-view-programming.html

[14] Signals & Slots. duben 2016. Dostupné z: http://doc.qt.io/qt-5.6/signalsandslots.html

[15] Object Trees & Ownership. 2016. Dostupné z: http://doc.qt.io/qt-5/objecttrees.html

[16] Threads Events QObjects. duben 2016. Dostupné z: https://wiki.qt.io/Threads_Events_QObjects

[17] Qt Namespace. duben 2016. Dostupné z: http://doc.qt.io/qt-5.6/qt.html

[18] Shafranovich, Y.: RFC: Common Format and MIME Type for Comma-Separated Values (CSV) Files. říjen 2005. Dostupné z: https://tools.ietf.org/html/rfc4180

[19] Mathworks: Sine Wave. Dostupné z: http://www.mathworks.com/help/simulink/slref/sinewave.html

[20] QSerialPort Class. 2016. Dostupné z: http://doc.qt.io/qt-5/qserialport.html

[21] QDockWidget Class. duben 2016. Dostupné z: http://doc.qt.io/qt-5/qdockwidget.html

[22] Qwt - Qt Widgets for Technical Applications. duben 2016. Dostupné z: http://qwt.sourceforge.net/index.html

[23] Lehmer, D.: Teaching combinatorial tricks to a computer. *Proceedings of Symposia in Applied Mathematics*, ročník 10, 1960.

[24] Dijkstra, E. W.: Information streams sharing a finite buffer. *Information Processing Letters*, ročník 1, 1972: s. 170–180.

[25] sleep_for - C++ Reference. duben 2016. Dostupné z: http://www.cplusplus.com/reference/thread/this_thread/sleep_for/

[26] Lentfer, A.: Microsecond Resolution Time Services for Windows. duben 2016. Dostupné z: http://www.windowstimestamp.com/description

Příloha A

Instrukční sada řídící jednotky

Instrukční soubor hardwarové části řídícího systému					
Instrukce	Slovní popis	Binární reprezentace	Operand č.1	Operand č.2	Prioritní
NOP	Prázdná instrukce	0000 0000	Není	Není	Ne
MOVE_WAIT	Čekání mezi instrukcemi	1000 0001	Délka čekání v hodinových taktech	Není	Ne
MOVE	Pohyb motoru	1000 0000	Interval mezi 2 kroky motoru	Poček kroků motoru	Ne
HALT	Zastavení činnosti	0000 0001	Není	Není	Ano
FLUSH	Zastavení činnost a vymázání prováděného programu	0000 0011	Není	Není	Ano
RESUME	Zahájení/obnovení činnosti	0000 0010	Není	Není	Ano
RESETCMD	Restart celé řídící jednotky	0000 0100	Není	Není	Ano
SET_POS	Nastavení výchozí pozice	0100 0000	Výchozí pozice	Není	Ne
READ_POS	Přečtení pozice	0100 0001	Není	Není	Ne
READ_R_POS	Přečtení pozice získané z kvadraturního enkodéru	0100 0010	Není	Není	Ne
SET_LIMIT_1	Ohraničení dovoleného prostoru	0001 0000	Hranice prostoru	Není	Ne
DISEN_LIMIT_1	Vypnutí první hranice	0001 0001	Není	Není	Ne
EN_LIMIT_1	Zapnutí první hranice	0001 0010	Není	Není	Ne
SET_LIMIT_2	Ohraničení dovoleného prostoru	0001 0011	Hranice prostoru	Není	Ne
DISEN_LIMIT_2	Vypnutí první hranice	0001 0100	Není	Není	Ne
EN_LIMIT_2	Zapnutí první hranice	0001 0101	Není	Není	Ne
BRAKE_EN	Aktivace magnetické brzdy	0001 0110	Není	Není	Ne
BRAKE_DIS	Deaktivace magnetické brzdy	0001 0111	Není	Není	Ne
READ_TIME	Přečtení času jednotky	0010 0001	Není	Není	Ne

Instrukce	Slovní popis	Binární reprezentace	Operand č.1	Operand č.2	Prioritní
SET_TIME	Nastavení času jednotky	0010 0000	Výchozí čas	Není	Ne
STATUS_REQ	Žádost o stavové informace	0001 1000	Není	Není	Ne
ECHO	Odpověď zařízení	1100 0011	Není	Není	Ne
RELAY1_DOWN	Rozepnutí relé 1	1110 0000	Není	Není	Ne
RELAY1_UP	Sepnutí relé 1	1110 0001	Není	Není	Ne
RELAY2_DOWN	Rozepnutí relé 2	1110 0010	Není	Není	Ne
RELAY2_UP	Sepnutí relé 2	1110 0011	Není	Není	Ne
ALARM_CLEAR	Vymazání alarmu na řídící jednotce ESMC–C2	1010 0101	Není	Není	Ne

Tabulka A.1: Instrukční sada řídící jednotky.

Příloha B

Odpovědi vysílané řídící jednotkou

Odezvy instrukcí generované řídící jednotkou			
Instrukce generující odezvu	Binární reprezentace odezvy – hlavička	Délka odezvy	Význam předávané hodnoty
HALT	1101 0001	1 B	Není
FLUSH	1101 0011	1 B	Není
RESUME	1101 0010	1 B	Není
READ_POS	0010 0101	5 B	Pozice motoru
READ_R_POS	0001 0101	5 B	Pozice získaná z kvadraturního enkodéru
READ_TIME	0100 0101	5 B	Času jednotky
BRAKE_EN	1111 0110	1 B	Není
BRAKE_DIS	1111 0111	1 B	Není
STATUS_REQ	1010 0100	4 B	Stavové informace, struktura odpovědi – tabulka 1.4
ECHO	1100 0011	1 B	Není
RELAY1_DOWN	1110 0000	1 B	Není
RELAY1_UP	1110 0001	1 B	Není
RELAY2_DOWN	1110 0010	1 B	Není
RELAY2_UP	1110 0011	1 B	Není
ALARM_CLEAR	0101 1010	1 B	Není

Tabulka B.1: Odezvy instrukcí generované řídící jednotkou

Odezvy generované událostí řídící jednotky		
Událost generující odezvu	Binární reprezentace odezvy – hlavička	Délka odezvy
WATCHDOG	1010 1010	1 B
QUEVE_OVERLOADED	0011 1100	1 B
NEXT_256	1111 1001	1 B
REACHED_L1	1001 0001	1 B
REACHED_L2	1011 0001	1 B
BRAKE_MANUAL	0010 0101	1 B
ALARM	1010 0101	1 B

Tabulka B.2: Odezvy událostí generované řídící jednotkou

Příloha C

Seznam použitých zkratek

CSV Comma Separated Values

FPGA Field Programmable Gate Array

GUI Graphical User Interface

MOC Meta Object Compiler – nástroj Qt pro generování kódu

MVC Model-View-Controller – návrhový vzor

UART Universal Asynchronous Reciever/Transmitter

USB Universal Serial Bus

Příloha D

Instalační příručka

Potřebné softwarové knihovny

- Qt 5.5
- Qwt 6.1.2

Spuštění Ve složce exe se nachází zkompilované binární soubory aplikace i simulátoru pro Windows x64. Pro spuštění stačí nakopírovat všechny soubory do cílového adresáře a spustit příslušný exe soubor. Simulátor lze spustit z příkazové řádky s parametrem nazvu portu. Obě aplikace včetně přiložených knihoven byly zkompilovány pomocí kompilátoru MSVC 2013 a otestovány na Windows 10.

Kompilace Na CD se nacházejí i zdrojové kódy frameworku Qt 5.5 a knihovny Qwt 6.1.2. Součástí zdrojových kódů jsou i projektové soubory pro Visual Studio 2013 respektive QtCreator.

Příloha E

Obsah přiloženého CD

```
readme.txt ........................... stručný popis obsahu CD
exe ................................... adresář se spustitelnou formou implementace
  app .................................. binární soubory aplikace
  simulator ............................ binární soubory simulátoru
src
  impl ................................. zdrojové kódy implementace
  app .................................. zdrojové kódy aplikace
  simulator ............................ zdrojové kódy simulátoru
  lib .................................. zdrojové kódy použitých knihoven
  thesis ............................... zdrojová forma práce ve formátu LaTeX
text .................................. text práce
  thesis.pdf ........................... text práce ve formátu PDF
```