VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DEPARTMENT OF CONTROL AND INSTRUMENTATION

MODERNÍ METODY NÁVRHU ŘÍDICÍCH SYSTÉMŮ S PODPOROU MATLAB/SIMULINK

NOVEL METHODS OF CONTROL SYSTEMS DESIGN WITH MATLAB/SIMULINK

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE AUTHOR	Vít Válek
VEDOUCÍ PRÁCE SUPERVISOR	doc. Ing. Petr Blaha, Ph.D.

BRNO 2019

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Automatizační a měřicí technika**
Ústav automatizace a měřicí techniky

Student: Vít Válek ***ID:*** 195457
Ročník: 3 ***Akademický rok:*** 2018/19

NÁZEV TÉMATU:

Moderní metody návrhu řídicích systémů s podporou MATLAB/Simulink

POKYNY PRO VYPRACOVÁNÍ:

1. Seznamte se s možnostmi generování zdrojového kódu v jazyce C pro vybrané typy mikrokontrolerů firmy NXP pomocí nástrojů programu MATLAB/Simulink.
2. Prozkoumejte možnosti kombinování algoritmů vytvořených v prostředí MATLAB /Simulink a ručně psaných zdrojových kódů v jazyce C
3. Na jednoduchých příkladech (funkcích) zhodnoťte výhody a nevýhody přístupů z bodů 1 a 2.
4. Otestujte generování zdrojového kódu na řídicí struktuře vektorového řízení pro PMS motor.

DOPORUČENÁ LITERATURA:

[1] NEBORÁK, I. Modelování a simulace elektrických regulovaných pohonů. Učební text Vysoká škola báňská - Technická univerzita Ostrava, 2002, ISBN: 80-248-0083-7

[2] KOZOVSKÝ, M.; BLAHA, P. Simulink Generated Control Algorithm for Nine-phase PMS Motor. In 7th IEEE International Conference on Control System, Computing and Engineering (ICCSCE 2017). Penang: IEEE, 2017. s. 69-74. ISBN: 978-1-5386-3896-5.

Termín zadání: 4.2.2019 ***Termín odevzdání:*** 20.5.2019

Vedoucí práce: doc. Ing. Petr Blaha, Ph.D.
Konzultant:

doc. Ing. Václav Jirsík, CSc.
předseda oborové rady

UPOZORNĚNÍ:
Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

Abstrakt

Obsahem této práce bude představit si nástroje programu MATLAB/Simulink, které umožňují generovat zdrojový kód v jazyce C. Dále bude předvedeno, jak kombinovat zdrojové kódy psané v jazyce C s modely Simulinku a s algoritmy v MATLAB. Pro vybrané funkce bude generován kód a ten pak porovnán s knihovními funkcemi RTCESL. V poslední části bude stručně popsán princip vektorového řízení. Pro zjednodušenou smyčku vektorového řízení bude generován zdrojový kód, který bude následně srovnán s ručně psaným kódem. Pro srovnání je používán mikrokontroler KV46F256 od firmy NXP Semiconductors.

Klíčová slova

Generování kódu, zdrojový kód, celočíselná a plovoucí aritmetika, vektorové řízení, RTCESL

Abstract

The content of this thesis is to introduce the tools of MATLAB/Simulink, which allow to generate the source code in C language. It will be demonstrated how to combine the source code written in C with Simulink model and MATLAB code. The code will be generated for the selected functions and compared with RTCESL library functions. In the last part of this thesis the principle of FOC will be briefly described. For a simplified loop of FOC, the code will be generated and then compared with handwritten code. For comparison, the microcontroller KV46F256 from NXP Semiconductors is used.

Keywords

Code generation, source code, fixed and floating point arithmetic, field-oriented control, RTCESL

Bibliografická citace:

VÁLEK, Vít. *Moderní metody návrhu řídicích systémů s podporou MATLAB/Simulink* [online]. Brno, 2019 [cit. 2019-05-16]. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/119070. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedoucí práce Petr Blaha.

Prohlášení autora o původnosti díla

„Prohlašuji, že svou bakalářskou práci na téma Moderní metody návrhu řídicích systémů s podporou MATLAB/Simulink jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

.

V Brně dne: **16. května 2019** ....

podpis autora

Poděkování

Děkuji vedoucímu bakalářské práce doc. Ing. Petru Blahovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé bakalářské práce.

V Brně dne: **16. května 2019** ....

podpis autora

Obsah

1. Úvod ... 1
2. Generování zdrojového kódu ... 2
2.1 MATLAB a jeho nástroje pro generování kódu ... 2
2.1.1 Příprava MATLAB kódu ... 3
2.1.2 Testování algoritmu ... 4
2.1.3 Generování zdrojového kódu ... 5
2.2 Generování kódu ze Simulinku ... 6
2.3 Nahrazení funkce vlastním algoritmem ... 11
2.4 Algoritmy v pevné a plovoucí řádové čárce ... 12
2.4.1 Fixed-Point Designer ... 13
3. Možnosti kombinování algoritmů ... 14
3.1 Volání C funkce z MATLABu a Simulinku ... 14
3.2 S-Function Builder ... 16
3.2.1 Simulační cyklus modelu ... 16
3.2.2 S-funkce ... 17
3.2.3 Implementace C funkce do Simulinku pomocí S-Function Builder ... 17
3.3 Legacy code tool ... 20
3.4 Ručně psané C MEX S-funkce ... 22
4. Knihovní funkce v porovnání s generovaným kódem ... 24
4.1 Časovač PIT ... 25
4.2 Výpočet odmocniny ... 26
4.2.1 GFLIB_Sqrt ... 26
4.2.2 Použití funkcí z math.h ... 28
4.3 Sinus ... 29
4.3.1 GFLIB_Sin ... 29
4.3.2 Funkce ze standardní knihovny ... 30
4.3.3 Funkce generované ze Simulinku ... 31
4.4 Clarkové transformace ... 33
4.4.1 GMCLIB_Clark ... 34
4.4.2 Algoritmus generovaný Simulinkem ... 35
5. Synchronní motor s permanentními magnety ... 36

5.1 Transformace 36
5.2 Matematický model PMS motoru 38
6. Vektorové řízení 40
6.1 Návrh regulátorů 41
6.2 Vektorové řízení z knihovních funkcí RTCESL 43
6.3 Kód pro vektorové řízení generovaný Simulinkem 45
7. Závěr 48

Seznam symbolů a zkratek

Zkratky:

FPU	…	Matematický koprocesor (Floating-point unit)
PIT	…	Periodic interrupt timer
PMSM	…	Synchronní motor s permanentními magnety (Permanent magnet synchronous motor)
LCT	…	Legacy code tool
TLC	…	Target Language Compiler
FOC	…	Vektorové řízení (Field-Oriented Control)
SVM	…	Space vector modulation
PWM	…	Pulzně šířková modulace (Pulse Width Modulation)

Symboly:

i_d	…	složka proudu v podélné ose dq systému	[A]
i_q	…	složka proudu v příčné ose dq systému	[A]
u_d	…	složka napětí v podélné ose dq systému	[V]
u_q	…	složka napětí v příčné ose dq systému	[V]
R_s	…	odpor statoru	[Ω]
ω_e	…	elektrická úhlová rychlost	[rad/sec]
ω	…	mechanická úhlová rychlost	[rad/sec]
L_d	…	indukčnost v podélné ose dq systému	[H]
L_q	…	indukčnost v příčné ose dq systému	[H]
ψ_{pm}	…	magnetický tok od permanentních magnetů	[Wb]
T_e	…	elektromagnetický moment	[Nm]
T_{load}	…	zatěžovací moment	[Nm]
J	…	moment setrvačnosti rotoru	[$kg \cdot m^2$]
p_p	…	počet pólových dvojic	[-]
p	…	Laplaceův operátor	[-]

Seznam obrázků

Obrázek 2.1 - Proces generování kódu [4] ... 3
Obrázek 2.2 - Generování kódu pomocí MATLAB Coder ... 5
Obrázek 2.3 - Konfigurační okno modelu v Simulinku ... 7
Obrázek 2.4 - S-funkce integrátoru ... 9
Obrázek 2.5 - Nastavení reentrantní funkce ... 9
Obrázek 2.6 - Přístup k signálu portu ... 10
Obrázek 2.7 – Model s goniometrickými funkcemi – ukázka nástroje Code Replacement Tool ... 11
Obrázek 3.1 - Blok MATLAB Function použitý pro volání C funkce ... 15
Obrázek 3.2 - Simulační cyklus modelu [9] ... 16
Obrázek 3.3 - Průběh výstupního signálu integrátoru ... 18
Obrázek 3.4 - Důsledek globálních proměnných S-funkce ... 19
Obrázek 3.5 - Přístup k funkcím pomocí Legacy code tool ... 21
Obrázek 4.1 - Nastavení optimalizací v MCUXpresso ... 24
Obrázek 4.2 - Zapnutí FPU jednotky v MCUXpresso ... 25
Obrázek 4.3 - Model funkce počítající sinus pomocí lookup tabulky ... 33
Obrázek 4.4 - Model Clarkové transformace ... 33
Obrázek 5.1 - Clarkové transformace [10] ... 37
Obrázek 5.2 - Parkova transformace [10] ... 37
Obrázek 5.3 - Simulace otáček PMS motoru ... 39
Obrázek 6.1 - Transformace signálů ve vektorovém řízení [12] ... 40
Obrázek 6.2 – Řídicí struktura vektorového řízení PMS motoru [11] ... 41
Obrázek 6.3 - Struktura navržených PI regulátorů ... 42
Obrázek 6.4 - Model řídicí struktury vektorového řízení ... 43
Obrázek 6.5 - Nastavení regulátoru přes dialogové okno ... 46

Seznam tabulek

Tabulka 4.1 - Délka výpočtu funkce GFLIB_Sqrt_F16() 27
Tabulka 4.2 - Délka výpočtu funkce GFLIB_Sqrt_FLT() 27
Tabulka 4.3 – Délka výpočtu funkce sqrtf() 28
Tabulka 4.4 – Délka výpočtu funkce sqrt() 28
Tabulka 4.5 – Délka výpočtu funkce GFLIB_Sin_F16 29
Tabulka 4.6 – Délka výpočtu funkce GFLIB_Sin_FLT 30
Tabulka 4.7 - Délka výpočtu funkce sinf() 31
Tabulka 4.8 - Délka výpočtu algoritmu pro blok Sine, Internal rule priority for lookup table: Speed 32
Tabulka 4.9 - Délka výpočtu algoritmu pro blok Sine, Internal rule priority for lookup table: Precision 32
Tabulka 4.10 - Délka výpočtu funkce GMCLIB_Clark_FLT 34
Tabulka 4.11 - Délka výpočtu funkce generované Simulinkem pro Clarkové transformaci 35
Tabulka 6.1 - Parametry navržených PI regulátorů 43
Tabulka 6.2 - Vybrané funkce RTCESL pro bloky z modelu na obrázku 6.4 44

1. ÚVOD

MATLAB a jeho nadstavba Simulink vyvíjené společností MathWorks jsou v dnešní době velice populární a mocné programy. Své uplatnění najdou v různých oborech spadajících pod ekonomiku, průmysl, zdravotnictví a další velké sféry. Díky aplikačním knihovnám, tzv. toolboxy, o které je možné MATLAB a Simulink obohatit, je možné zaměřit se na konkrétní problematiku. Jednotlivé toolboxy spadající jak pod MATLAB, tak pod Simulink spolu můžou spolupracovat za předpokladu správného propojení dílčích částí. Již se základními znalostmi tak můžeme vyvíjet vlastní algoritmy, vytvářet, testovat a simulovat modely nebo zpracovávat získané informace z různých zdrojů.

Programování je v současnosti často diskutované téma. Ačkoliv máme k dispozici několik programovacích jazyků, ne všechny můžou být uplatněny ve všech aplikacích. Ruční psaní zdrojových kódů může být náchylné na chyby. Proto je snahou vyvíjet programy a možnosti, díky kterým bude programování pro uživatele přívětivější, časově úspornější a celkově bude efektivnější ve všech směrech. Jednou z možností jsou programy, které umožňují generovat požadovaný zdrojový kód v konkrétním programovacím jazyce. Pro uživatele je k dispozici prostředí, které je více intuitivní. Zde sestaví algoritmus například z hotových funkcí nebo grafických bloků. Z tohoto algoritmu je následně vygenerován kód v daném programovacím jazyce.

V ideálním případě je nejvhodnější algoritmus nejdříve otestovat na modelu reálného zařízení a poté z něj generovat kód. Předejme případnému zničení reálného zařízení, které by mělo dopad na finanční stránku věci. MATLAB/Simulink nabízí relativně dobré spojení těchto dvou požadavků – simulace a generování kódu.

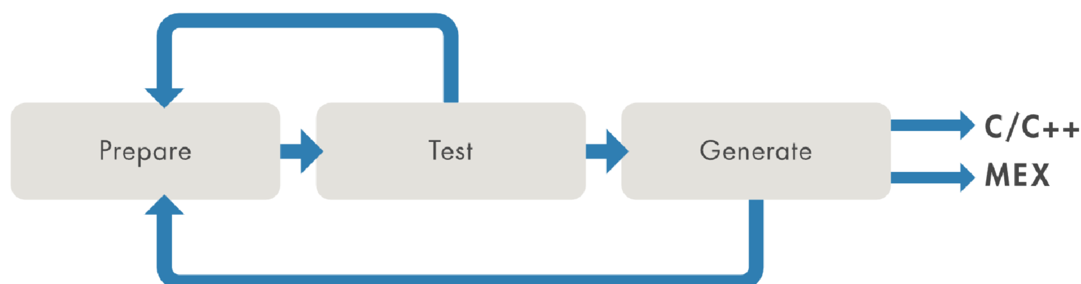
2. GENEROVÁNÍ ZDROJOVÉHO KÓDU

MATLAB a Simulink jsou díky svým možnostem vhodné při vyvíjení algoritmů pro různé aplikace. Následující manuální implementace těchto algoritmů do jazyka C, případně automatické generování zdrojových kódů však s sebou přináší značné komplikace. MATLAB je oproti jazyku C interpretovaný jazyk, který explicitně nevyžaduje specifikaci datových typů. Jako výchozí datový typ používá double, který je náročný jak na výpočetní výkon, tak na paměť. Při použití PC nás tento fakt většinou neomezuje. V případě mikrokontrolerů bez matematického koprocesoru se při práci s čísly s pohyblivou řádovou čárkou můžeme potýkat až s několikanásobným zvýšením výpočetního času. Následky pak můžou být různé.

Před samotným vývojem algoritmu bychom měli zvážit, jaký typ mikrokontroleru budeme později používat. Budeme mít na výběr mezi jednotkou bez matematického koprocesoru nebo s matematickým koprocesorem. Dále již budeme používat pro matematický koprocesor zkratku FPU. Mikrokontrolery s FPU můžeme dále rozlišovat podle toho, s jakým formátem čísla s pohyblivou řádovou čárkou pracují. Obvykle se setkáme s formáty single precision a double precison neboli s jednoduchou přesností a dvojitou přesností. Tyto dva formáty jsou definovány standardem IEEE754 [2]. Mikrokontrolery řady KV4x od firmy NXP s jádrem ARM Cortex M4 disponují single precision FPU [3]. Jsou určeny pro řízení motorů, kde je práce s pohyblivou řádovou čárkou často potřebná. Pro příklad si můžeme uvést počítání goniometrických funkcí. V dalších kapitolách budeme testovat funkce právě na mikrokontroleru z řady KV4x, jehož typ a parametry bude uvedeny později. Nyní se zaměříme na přípravu kódu v MATLABu, a poté konfiguraci modelu v Simulinku. To je jeden z předpokladů pro dosažení dobrých výsledků při generování zdrojových kódů.

2.1 MATLAB a jeho nástroje pro generování kódu

Pro generování zdrojového kódu v jazyce C z MATLABu budeme používat toolbox MATLAB Coder. Ten mimo jiné umožňuje generovat statické a dynamické knihovny, spustitelné soubory a MEX funkce. Celý proces generování kódu můžeme rozdělit na několik částí, jak je vidět na obrázku 2.1.



Obrázek 2.1 - Proces generování kódu [4]

V první části je třeba se zaměřit na samotný algoritmus v jazyce MATLAB. Aby jej bylo možné dále zpracovat, musíme dodržet určitou syntaxi kódu a používat funkce, které jsou toolboxem MATLAB Coder podporovány [5].

Následuje testování a ladění vytvořeného algoritmu. Pomocí MEX funkce ověřujeme, zdali vytvořený algoritmus prochází unit testy. V případě nekorektního chování opakujeme proces od začátku.

V poslední fázi volíme výstupní soubory a specifikujeme některé jejich vlastnosti. V našem případě se budeme převážně zabývat zdrojovými soubory v jazyce C. Vytváření spustitelných souborů, statických a dynamických knihoven probíhá také v této poslední části.

2.1.1 Příprava MATLAB kódu

Jazyky MATLAB a C se liší v mnoha směrech. MATLAB pracuje implicitně s datovým typem double. Umožňuje libovolně mezi sebou přiřazovat proměnné různých datových typů, aniž bychom museli řešit typovou konverzi. Dále je možné dynamicky vytvářet a měnit proměnné. Jazyk C se chová v podstatě opačně. Vždy musíme definovat datový typ, velikost a další vlastnosti proměnné. Nelze libovolně přiřazovat proměnné jiných datových typů. Dynamická alokace je možná pouze tehdy, bude-li tato skutečnost známa před samotným překladem zdrojového kódu.

Již při psaní algoritmu v MATLABu můžeme definovat datové typy a vlastnosti vstupních signálů a proměnných. Níže následuje jednoduchý příklad, na kterém si vysvětlíme, jak pracovat s datovými typy. Předpokládejme funkci *example_fce*, u které požadujeme vstupy *x1*, *x2* datového typu double a *x3* typu int32. Datové typy na vstupech ošetříme pomocí funkce *isa()*. Ve výsledném kódu budeme používat statickou a globální proměnnou. Statická je vložena pomocí klíčového slova *persistent*. Při generování kódu je hodnota této proměnné implicitně inicializována jako prázdná matice. Pokud nedojde k inicializaci proměnné, nebude možné dokončit generování zdrojových souborů. Definice a inicializace proměnné je v podmínce testování prázdné

matice. Obdobně definujeme globální proměnnou pomocí *global*. U té však nemusíme řešit implicitně přiřazenou nulovou matici. Definice a deklarace globálních proměnných budou v generovaném kódu v samostatném zdrojovém a hlavičkovém souboru. Pro úspěšné vygenerování kódu s globálními proměnnými musíme nejdříve spustit MATLAB funkci, např. příkazem *example_fce(1, 3, int32(5))*. Pokud bychom v této chvíli chtěli vytvořit MEX funkci pro následné používání, zavoláme příkaz *codegen example_fce*. Další informace o syntaxi, kterou je třeba dodržovat, můžeme nalézt v dokumentaci [5].

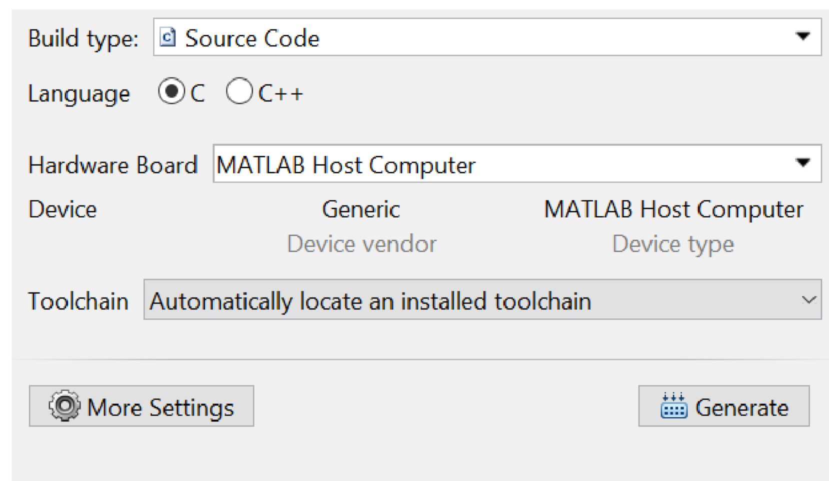
2.1.2 Testování algoritmu

Testování probíhá již v aplikaci, kterou spustíme přes APPS – CODE GENERATION – MATLAB Coder. Po jejím spuštění nejprve vybereme funkci, pro kterou budeme generovat kód. Dále zvolíme datové typy vstupních portů, a to buď ručně nebo automaticky. Ve druhém případě musíme vložit skript, který naši požadovanou funkci volá. Na základně vstupních hodnot MATLAB přiřadí datové typy vstupních portů. V části *Check for Run-Time Issues* vybereme skript nebo funkci, která bude testovat MEX funkci. V případě rozdílných výsledků MATLAB funkce a z ní vygenerované MEX funkce budeme na tuto skutečnost upozorněni.

```
function [y1,y2] = example_fce( x1,x2,x3 ) %#codegen
% kontrola datovych typu vstupu
assert(isa(x1, 'double'));
assert(isa(x2, 'double'));
assert(isa(x3, 'int32'));

persistent temp1;        % static
if isempty(temp1)
   temp1 = int32(4);   % pretypovani, inicializace
end

global temp2;       % global
temp2 = int16(3);    % datovy typ promenne - int16, inicializace

temp1 = temp1 + 1;
y1 = x1+ x2 + (double(temp1));
y2 = x2 + double(x3);
end
```

2.1.3 Generování zdrojového kódu

Poslední a pro nás nejzajímavější částí je generování kódu. Jelikož má nastavení generování několik možností, shrneme si pouze ve zkratce, co je nám k dispozici. Opět budeme používat aplikaci MATLAB Coder a v ní záložku *Generate Code*.



Obrázek 2.2 - Generování kódu pomocí MATLAB Coder

Na obrázku 2.2 vidíme, že požadované generované soubory budou zdrojové kódy v jazyce C. Kompilátor bude vybrán automaticky. Vzhledem k omezení používané licence nemůžeme zvolit v položce *Hardware Board* jinou cílovou platformu než MATLAB. Otevřením *More Settings* se dostaneme k dalšímu nastavení.

Speed – Povolení/zakázání přetékání, používání pouze celočíselných datových typů. Pokud bude zakázáno přetékání proměnných, budou v generovaném kódu algoritmy kontrolující minimální a maximální hodnoty proměnných.

Memory – Zde například nastavíme, zda bude vytvořená funkce reentrantní.

Code Appereance – Nastavení generování souborů pro funkce *initialize* a *terminate*. Můžou být zvlášť nebo dohromady s generovanou funkcí. Dále můžeme zakázat vložení komentářů, povolit bitový posuv pro násobení/dělení dvěma a jiné.

Debugging – Vložení funkcí do kódu, které upozorní na chybu při běhu programu. Vytvoření zprávy (report), přes kterou jde nahlédnout do generovaných souborů.

Custom Code – Vložení vlastních hlavičkových souborů. Volání dalších funkcí z inicializační a ukončovací funkce (*initialize*, *terminate*).

All Settings – Jsou zde zahrnuty všechny položky, které se dají nastavit. Také je zde navíc nastavení *Advanced*. Za povšimnutí stojí *Function Inlining*. Vhodným nastavením můžeme zajistit tzv. inline funkce. Můžeme tak buď zkrátit čas vykonání algoritmu nebo ušetřit paměť, kterou bude algoritmus zabírat.

2.2 Generování kódu ze Simulinku

Pro generování zdrojového kódu ze Simulinku máme k dispozici toolboxy Simulink Coder a Embedded Coder. První zmíněný je zaměřen spíše na desktopové aplikace, zatímco druhý generuje kód přívětivější pro embedded zařízení. Jelikož je práce směřována k mikrokontrolerům, budeme tedy používat Embedded Coder.

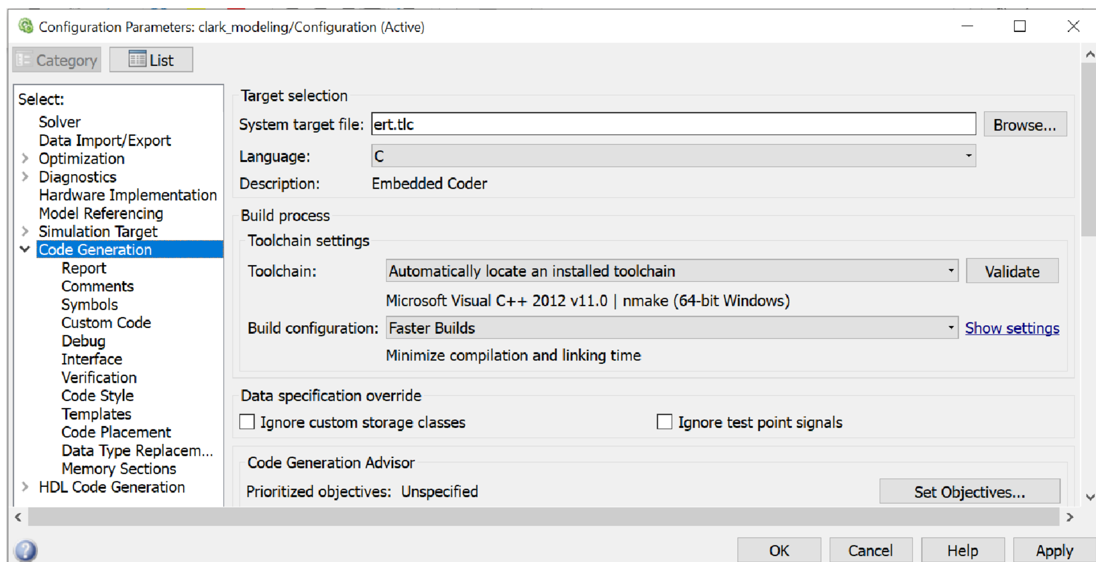
Abychom mohli generovat kód, musíme mít v parametrech modelu nastaven pevný krok simulace: *Configuration Parameters – Solver – Type: Fixed-step*. Je vhodné v modelu používat pouze bloky, které pracují diskrétně. Lze použít i prvky spojité, avšak v důsledku může být generovaný kód zbytečně složitý a nepřehledný kvůli výpočtu dalšího kroku simulace a jiných záležitostí. Povolit generování kódu z bloků pracujících spojitě můžeme v *Configuration Paramaters – Code Generation – Interface*. Zde je třeba aktivovat položky *absolute time* a *continuous* time. Použití spojitých prvků je ve výsledku zbytečné, jelikož výpočetní jednotka, na které bude kód běžet, pracuje diskrétně.

V dalším kroku budeme volit typ cílového zařízení. V parametrech modelu vybereme položku *Code Generation*, ve které otevřeme nabídku pro *System Target File*. Výchozí nastavení *grt.tlc* je určeno pro aplikace běžící v reálném čase. Generovaný kód není příliš čitelný a pro embedded zařízení je nevhodný. Budeme proto používat *ert.tlc* (Embedded Coder). Programovací jazyk zvolíme C. V části *Code Generation Advisor* můžeme nastavit kontrolu modelu před generováním kódu. Po otevření *Set Objectives* vybereme, jaké máme požadavky na kód, tedy jestli požadujeme kód úsporný na paměť nebo naopak kód, který se bude rychle vykonávat a spotřeba paměti nás neomezuje. Tlačítkem *Check Model* spustíme kontrolu. Poté nám vyskočí okno, kde je doporučené nastavení některých parametrů. Ty můžeme buď změnit nebo nabídku ignorovat. Toto doporučené nastavení záleží na požadavcích, které byly zvoleny v *Set Objectives*. Pro komprimování vygenerovaných souborů do souboru ZIP můžeme zvolit nastavení *Package code and artifacts*.

V záložce *Code Generation – Report* volíme obsah zprávy o generování kódu. Ta se při aktivování *Open report automatically* otevře v případě úspěšného generování. Obsahuje náhled na vygenerované zdrojové soubory, informace o vstupních a výstupních portech, o parametrech, a také je zde uvedeno, v jakém souboru a na jakém řádku jsou tyto argumenty použity. Volbou *Model-to-code* můžeme sledovat, ke kterému bloku v modelu se vztahuje určitá část vygenerovaného kódu. Pokud model obsahuje soubor, který umožňuje nahradit funkce v generovaném kódu za námi definované funkce, můžeme si seznam těchto funkcí nechat zobrazit ve zprávě o generování kódu. Aktivujeme *Summarize which blocks triggered code replacements*. Jak toto nahrazení funkcí (Code replacement library) funguje bude popsáno na konci kapitoly.

Nastavení pro vkládání komentářů je v záložce *Coder Generation – Comments*. Pokud chceme komentovat jednotlivé bloky, klikneme pravým tlačítkem na blok a

vybereme *Properties – General*. Do okna *Description* vložíme text, který bude sloužit jako komentář ve vygenerovaném kódu.



Obrázek 2.3 - Konfigurační okno modelu v Simulinku

V *Code Generation – Symbols* můžeme změnit formát definování globálních typů a proměnných, názvy funkcí volající subsystémy a jiné. Jako příklad si ukážeme změnu názvu globálních typů. Předpokládejme, že máme v modelu externí vstupy (Simulinkové bloky *In1*) a v *Code Generation – Symbols* toto nastavení: *Global types: NM*, *System-generated idetifiers: Shortened*. Název struktury, která v sobě nese signály vstupních portů, se bude jmenovat ExtU. Změnou *Global types* na *NM_T* bude název struktury ExtU_T. Nastavení v *Symbols* má tedy vliv na formální úpravu generovaného kódu.

Budeme-li chtít do generovaného kódu vložit další hlavičkové soubory, případně deklarovat nebo volat C funkce, můžeme tak učinit v *Code Generation – Custom Code*. Toto vkládání provádíme v části *Insert custom C code in generated*, změny pak můžeme pozorovat v souborech *nazevModelu.h* a *nazevModelu.c*. Deklarace funkcí píšeme do *Source file*. Přidání knihoven do zdrojového kódu je možné v *Header file*. Volat další funkce můžeme buď v inicializační nebo v ukončovací funkci. Vybíráme tedy z *Initialize function* a *Terminate function*. Pokud budeme chtít ke generovaným souborům fyzicky přidat hlavičkový soubor, specifikujeme jej v *Include list of additional* v nabídce *Souce files*.

V záložce *Code Generation – Interface* můžeme zvolit podporu například bloků se spojitým časem, čísel s pohyblivou řádovou čárkou, komplexních čísel, S-funkcí, které nejsou inline. Dále zde můžeme specifikovat hlavičky funkcí, které Simulink generuje pro jednotlivé části simulace. Konkrétně jde o tyto funkce: inicializace, počítání výstupů (hlavní funkce *step()*), aktualizace diskrétních stavů a ukončovací funkce *terminate*. Při

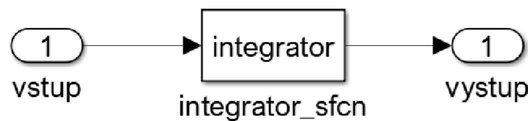
nastavení *Code interface – Code interface packaging: Nonreusable function* nebudou mít funkce argumenty. Budou pracovat přímo s globálními proměnnými. Pokud bychom požadovali funkce s parametry, nastavíme *Code interface packaging* na *Reusable function*. Tímto nastavením můžeme zajistit reentrantní funkce. Pokud model obsahuje diskrétní stavy, můžeme je aktualizovat buď ve funkci počítající výstupy nebo v samostatné funkci, která bude mít na starost pouze aktualizaci těchto stavů. Toto nastavení volíme v *Code interface – Single output/update function*. Pro vlastní název a hlavičku inicializační a funkce počítající výstupy (resp. funkce, která se volá v každém kroku simulace) otevřeme tlačítkem *Configure Model Functions* konfigurační okno. V rozbalovací nabídce *Function specification* vybereme *Model specific C prototypes* a potvrdíme tlačítkem *Get Default Configuration*. Následně můžeme změnit názvy funkcí, argumentů a možnost jejich předávání (hodnota, ukazatel).

Další nastavení generovaného kódu je možné specifikovat *Code Generation – Code Style*. Můžeme zde nastavit optimalizaci rozhodovací podmínky *if-else* nebo vytvoření přepínače *switch* pro více větví *if-else*. Vyskytují-li se v modelu signály a proměnné násobené a dělené mocninou o základu 2, můžeme tyto operace nahradit bitovým posuvem.

Některé optimalizace generovaného kódu je možné nastavit v konfiguračních parametrech modelu v záložce *Optimization*. Můžeme zde potlačit generování bloků, které nejsou potřebné pro výsledný kód, nebo třeba inicializaci některých datových struktur a proměnných. Za povšimnutí stojí v *Signals and Parameters* parametr *Default parameter behavior*. Nastavení na *Tunable* zajistí, že všechny parametry bloků nebudou v kódu vloženy jako konstanty, ale jako proměnné, které jsou uloženy ve struktuře parametrů.

V záložce *Optimization* můžeme mimo jiné zvolit výchozí datový typ, který Simulink používá pro nedefinované datové typy konstant, signálů a celkově všeho, co nějakým způsobem souvisí s datovým typem. Výchozím datovým typem je double. Ten můžeme změnit v *Default for underspecified data type* na *single*. Toto nastavení nám může ulehčit práci při vývoji algoritmu na mikrokontroler se single precision FPU. Poté si jen musíme dát pozor, abychom v některém z bloků nezměnili datový typ na double. Ve výsledném kódu by se tak neměl objevit datový typ double, pokud jsme jej sami nikde nenastavili.

Simulink ve výchozím nastavení generuje struktury pro vstupní a výstupní porty, parametry bloků a pracovní vektory. Předpokládejme S-funkci realizující integrátor vytvořenou pomocí nástroje LCT, který je popsán v další kapitole práce. Vstup i výstup jsou celočíselného datového typu o velikosti 16 bitů.

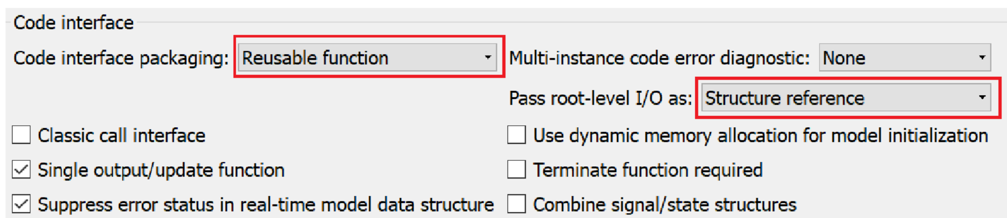


Obrázek 2.4 - S-funkce integrátoru

V konfiguračních parametrech modelu nastavíme pevný simulační krok a *System target file* na *ert.tlc (Embedded Coder)*. Ostatní parametry ponecháme ve výchozím nastavení. Ve vygenerovaném souboru *nazevModelu.h* bude vstupní a výstupní port definován ve struktuře:

```
typedef struct {
  int16_T vstup;                /* '<Root>/vstup' */
} ExtU_integrator_mdl_T;

typedef struct {
  int16_T vystup;               /* '<Root>/vystup' */
} ExtY_integrator_mdl_T;
```

V jednotlivých funkcích, které jsou vygenerovány, se k těmto strukturám přistupuje jako ke globálním proměnným. Takové funkce nejsou reentrantní a jejich používání nás může při pozdější implementaci omezovat. Reentrantnost můžeme zajistit v *Configuration Parameters – Code Generation – Interface* nastavením podle obrázku 2.5.



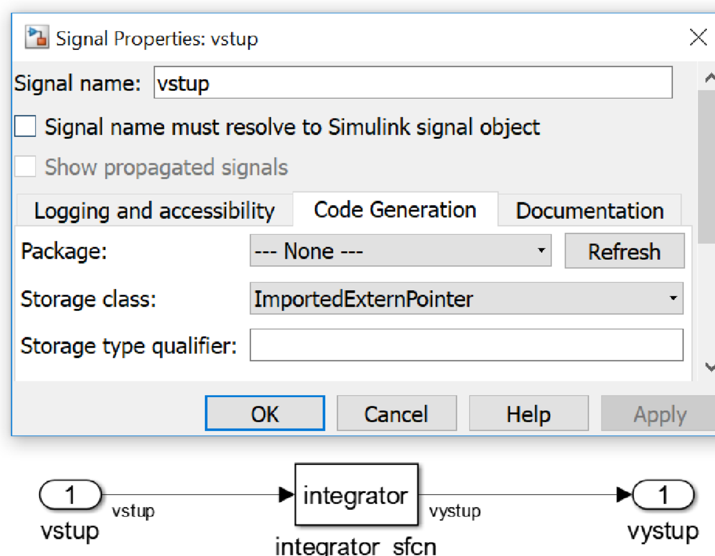
Obrázek 2.5 - Nastavení reentrantní funkce

Jak můžeme níže vidět, nově vygenerovaná funkce přebírá ukazatele na struktury. V těchto strukturách jsou uloženy pracovní vektory, parametry bloků, signály vstupních a výstupních portů.

```
extern void integrator_mdl_step(RT_MODEL_integrator_mdl_T *const
integrator_mdl_M, ExtU_integrator_mdl_T *integrator_mdl_U, ExtY_integrator_mdl_T
*integrator_mdl_Y);
```

Přistupovat k signálům portů v generovaném kódu lze i jinak. Opět předpokládejme nastavení pevného kroku simulace a *System target file: ert.tlc*. Po kliknutí pravým tlačítkem na signál spojující požadovaný port a S-funkci se rozbalí nabídka, ve které vybereme *Properties*. Zadáme název signálu a zvolíme možnost, jak bude signál definován viz obrázek 2.6. V našem případě zvolíme *ImportedExternPointer* pro vstupní i výstupní signál. Půjde tedy o ukazatele. Nyní ve vygenerovaném zdrojovém kódu nejsou porty ve struktuře, nýbrž jsou definovány jako ukazatele. Ve funkcích se k nim přistupuje jako ke globálním proměnným.

```
extern int16_T *vystup;        /*'<Root>/integrator_sfcn' */
extern int16_T *vstup;        /* '<Root>/vstup' */
```



Obrázek 2.6 - Přístup k signálu portu

Obsahuje-li model jeden nebo více subsystémů, máme opět více možností, jak bude kód popisující daný subsystém vygenerován. Pokud vytvoříme subsystém a nijak neměníme jeho parametry, tak se jeho popis pomocí jazyka C přímo vloží do funkce *step()*. Stejným způsobem se kód vygeneruje, jestliže provedeme následující nastavení. Pravým tlačítkem na subsystém rozbalíme nabídku, ve které vybereme *Block Parameters (Subsystem)*. Aktivujeme položku *Treat as atomic unit*. V záložce *Code Generation* nastavíme *Function packaging: Inline*. Dále budeme vycházet z toho, že máme povolenou volbu *Treat as atomic unit*.

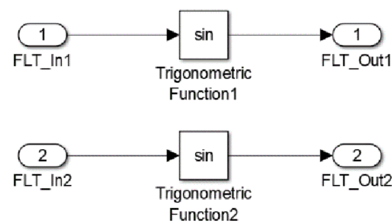
Další varianta je vytvořit pro subsystém unikátní funkci. V záložce *Code Generation* vybereme *Function packaging: Nonreusable function*. Nastavením zbývajících parametrů v *Code Generation* můžeme zvolit název funkce v generovaném kódu nebo můžeme zajistit, že se pro funkci vygeneruje vlastní zdrojový a hlavičkový soubor. Při výchozím nastavení se kód subsystému generuje do souboru *nazevModelu.c*.

Budeme-li chtít vygenerovat pro subsystém reentrantní funkci, nastavíme *Function packaging: Reusable function*. Můžeme opět volit vlastní název hlavičky funkce nebo generování kódu do samostatných souborů. Výhoda při nastavení *Reusable function* spočívá v tom, že se vygeneruje pouze jedna funkce, která je volána s různými parametry. Oproti inline funkcím nebo nereentrantním funkcím tak většinou šetříme paměť procesoru.

Záleží na okolnostech, jak nastavíme subsystém. Měli bychom zvážit, jak náročný je algoritmus pro subsystém. Musíme brát v potaz, že režie při volání podprogramů trvá nějakou dobu. Je tedy vhodné zvážit mezi inline a reentrantními funkcemi. Další kritérium může být kolikrát subsystém voláme, případně jak často.

2.3 Nahrazení funkce vlastním algoritmem

Někdy se může stát, že v simulaci používáme funkci, ale ve vygenerovaném kódu ji chceme nahradit za vlastní funkci. Nahradit ji můžeme ručně. To je pochopitelně neefektivní, pokud bychom měli těchto funkcí více. Mohlo by se pak stát, že některou funkci vynecháme nebo nahradíme nesprávnou funkcí. MATLAB i Simulink disponují nástrojem, který dokáže nahradit určité funkce automaticky. Jde o tzv. Code Replacement Tool. My se budeme zabývat pouze variantou nahrazování funkcí v Simulinku. Pro MATLAB jsou postupy podobné. Jako příklad si ukážeme nahrazení goniometrické funkce sinus z modelu na obrázku 2.7.



Obrázek 2.7 – Model s goniometrickými funkcemi – ukázka nástroje Code Replacement Tool

Vstupní port *FLT_In1* je datového typu single, druhý vstupní port je double. Matematická knihovna je vybrána C99 (ISO). To zajistí vkládání single precision funkcí v generovaném kódu. Je však nutné, aby signál vstupující do bloku byl datového typu single. Nastavit matematickou knihovnu můžeme v parametrech modelu v záložce *Code Generation – Interface – Standard math library*. Pokud nyní necháme vygenerovat kód, bude funkce *step()* obsahovat následující kód:

```
void CRT_example_step(void)
{
CRT_example_Y.FLT_Out1 = sinf(CRT_example_U.FLT_In1);
CRT_example_Y.FLT_Out2 = sin(CRT_example_U.FLT_In2);
}
```

Do příkazového řádku v MATLAB zadáme příkaz *crtool*. Základem je tabulka, kterou vytvoříme ve *File – New table*. V pravé části ji pojmenujeme a potvrdíme *Apply*. Nyní přidáme vstup tabulky, kterým definujeme funkci, která se má nahradit: *File – New entry – Function*. Přejdeme do záložky *Mapping Information*. Ve *Function* vybereme funkci, kterou budeme chtít nahradit, v našem případě to je *sin*.

Conceptual arguments – zde se nastavuje, pro jaké datové typy vstupů a výstupů funkce se bude funkce nahrazovat. Pro *u1* a *y1* zvolíme *single*.

Replacement function – do políčka *Name* vkládáme funkci, která bude použita jako náhrada funkce *sin* nebo kterékoliv jiné zvolené. Opět volíme datové typy vstupů a výstupů funkce. Tak jako v *Conceptual arguments*, tak i zde nastavíme pro *u1* a *y1* *single*. Tlačítkem *Validate entry* zkontrolujeme korektnost vstupu tabulky. V záložce *Build Information* můžeme přidat cesty ke zdrojovým a hlavičkovým souborům, jenž obsahují nahrazenou funkci. Vytvořenou tabulku uložíme.

Ve zbývajících dvou krocích zaznamenáme naši novou tabulku do MATLAB/Simulinku. Nejdříve vytvoříme registrační soubor: *File – Generate registration file*. Zvolíme libovolný název, do *Table list* napíšeme název, pod kterým jsme tabulku uložili a potvrdíme. Do příkazového řádku v MATLAB zadáme *RTW.TargetRegistry.getInstance('reset')*.

Opět přejdeme do konfigurace parametrů modelu pro generování kódu (*Code Generation – Interface*). Po otevření nabídky *Code replacement library* zde máme na výběr i tabulku, kterou jsme si sami vytvořili. Při použití vlastní tabulky pak může vygenerovaný kód pro model na obrázku 2.7 vypadat následovně:

```
void CRT_example_step(void)
{
CRT_example_Y.FLT_Out1 = GFLIB_Sin_FLT(CRT_example_U.FLT_In1);
CRT_example_Y.FLT_Out2 = sin(CRT_example_U.FLT_In2);
}
```

2.4 Algoritmy v pevné a plovoucí řádové čárce

Jak již bylo řečeno, MATLAB i Simulink používají jako výchozí datový typ double. Pro simulaci to nepředstavuje žádné překážky, při implementaci generovaného kódu

v plovoucí aritmetice na mikrokontrolery pracující v celočíselné aritmetice však mnohdy narazíme na nepřijatelné výsledky. Tento problém můžeme řešit dvojím způsobem. První a asi i jednodušší je vhodně zvolit mikrokontroler. Budeme vybírat takový, který obsahuje i FPU. Jestli FPU s jednoduchou nebo dvojitou přesností bude záležet na algoritmech, které budou prováděny. Toto řešení může být znatelně finančně náročné, pokud by se taková změna pohybovala v řádech tisíců kusů mikrokontrolerů. Druhá varianta je psát kód v celočíselné aritmetice. To přináší komplikace v podobě náročnějšího vývoje algoritmu. MATLAB však nabízí aplikační knihovnu *Fixed-Point Designer*, pomocí které můžeme převádět algoritmy z plovoucí aritmetiky do celočíselné aritmetiky.

2.4.1 Fixed-Point Designer

Tento nástroj je poměrně složitý, a proto si shrneme jen základní možnosti, které nabízí. Jak již bylo řečeno, slouží pro převod z plovoucí do celočíselné aritmetiky. Ještě zmíníme, že umožňuje převod čísel s dvojitou přesností na čísla s jednoduchou přesností.

Princip převodu je následující. Fixed-Point Designer na základě simulace zjistí, jakých minimálních a maximálních hodnot dosahují signály a proměnné v algoritmu (model nebo MATLAB kód). Druhá varianta je, že mu minima a maxima zadáme ručně. Na základě těchto hodnot vytvoří frakční datové typy, které mohou být následně používány.

Abychom měli přehled nad výsledky, máme možnost srovnávat výsledky, kterých bylo dosaženo při použití plovoucí aritmetiky a celočíselné aritmetiky. Tím pádem jsme schopni vyhodnotit, jaký vliv měl převod do celočíselné aritmetiky na výsledné chování systému.

Nyní si ve stručnosti popíšeme, jak by mohl vypadat převod z plovoucí do celočíselné aritmetiky. Předpokládejme čísla v pohyblivé řádové čárce ležící v rozsahu <-1; 1). Tyto čísla budeme chtít vyjádřit pomocí 16bitového celého čísla. Nejvyšší bit použijeme na vyjádření znaménka. Zbývajících 15 bitů nám zbývá pro vyjádření frakční části čísla. Zároveň nám udává rozlišení, s jakým mohou být čísla vyjádřena. V tomto případě jde o rozlišení 2^{-15}. Pro číslo x = 0.5 bude převod následující:

$$x * 2^{15} = 16384_{10} = 0100\ 0000\ 0000\ 0000_2 \tag{2.1}$$

Tato frakční aritmetika je mimo jiné používána v knihovně RTCESL. Existují další formáty pro vyjádření plovoucích čísel v pevné řádové čárce. Více se o této problematice můžeme dočíst v [8].

3. MOŽNOSTI KOMBINOVÁNÍ ALGORITMŮ

MATLAB a Simulink nabízí možnost kombinovat zdrojový kód v jazyce C s funkcemi napsanými v MATLABu, s modely nebo Stateflow diagramy vytvořenými v Simulinku. Tyto algoritmy je možné simulovat a následně z nich generovat zdrojový kód.

Začlenit funkce psané v jazyce C je možné několika způsoby. Od verze MATLAB R2018b je v Simulinku dostupný blok *C Caller*, ze kterého můžeme volat algoritmy napsané v C jazyce. My však zmíníme pouze možnosti, které jsou k dispozici ve verzi R2015b.

3.1 Volání C funkce z MATLABu a Simulinku

Přímé volání funkcí psaných v jazyce C není možné. Můžeme však vytvořit MEX funkci, kterou lze následně volat z funkcí a skriptů MATLAB. Pokud budeme chtít generovat zdrojový kód s vlastním algoritmem v C, není nutné MEX funkce vytvářet. Níže je uveden příklad, na kterém bude vysvětleno začlenění jednoduché funkce *my_power* napsané v jazyce C. Předpokládejme, že máme k dispozici soubory *my_power.h* s deklarací a *my_power.c* obsahující tělo funkce počítající mocninu čísla.

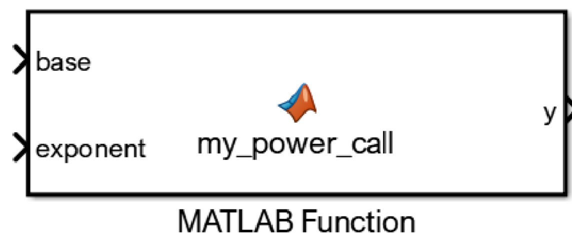
Budeme vytvářet klasickou funkci v MATLABu. Na začátku funkce nejdříve ošetříme datové typy vstupních signálů pomocí příkazu *isa()*. Kontroluje, zdali při volání funkce posíláme na konkrétní vstup signál s žádaným datovým typem. Pokud toto ošetření provedeme, nemusíme při generování souborů pomocí příkazu *codegen* specifikovat datové typy vstupních parametrů funkce. V případě, že budeme pro generování používat aplikaci ze záložky APPS, je třeba v části *Define Input Types* nastavit stejné datové typy jako v příkaze *isa()*. Pokud se budou typy lišit, nebude vygenerován kód, který pro svou činnost používá vstupní parametry.

Pokud budeme naši funkci volat příkazem *my_power_call*, bude splněna první podmínka. V případě generování MEX funkce nebo zdrojového kódu se vykonají příkazy za *else*. Právě tato druhá větev podmínky je pro nás zajímavá. Pomocí příkazu *coder.updateBuildInfo()* přidáváme zdrojový soubor, ve kterém je tělo volané funkce. Pro vložení knihovny slouží *coder.cinclude()*. Volání funkce je pak: *y=coder.ceval('my_power',base, exponent)*. V této chvíli je náš algoritmus nachystán pro vygenerování MEX funkce nebo zdrojového kódu. MEX funkci vygenerujeme příkazem *codegen my_power_call*. Je možné konfigurovat nastavení ovlivňující generovanou MEX funkci. Pro ověření funkčnosti postačí výchozí nastavení. Jak již bylo zmíněno, takto vytvořenou funkci lze volat ze skriptů a z dalších funkcí. Pokud budeme chtít generovat zdrojové soubory z algoritmů, které již MEX funkci obsahují, nebude nám to umožněno.

```
function y = my_power_call( base, exponent ) %#codegen
% my_power_call
% MATLAB funkce - vypocet mocniny cisla
% MEX funkce, generovany kod - vypocet mocniny pouze pro cele kladne exponenty
assert(isa(base, 'double'));
assert(isa(exponent, 'double'));
y = 0.0;                    % predinicializace

if coder.target('MATLAB')
  % pri spousteni MATLAB funkce
  result = power(base, exponent);
  y = result;
else
  % generovani kodu, MEX funkce
 coder.updateBuildInfo('addSourceFiles','my_power.c');
 coder.cinclude('my_power.h');
 y = coder.ceval('my_power', base, exponent);
end
end
```

Pomocí příkazu *coder.ceval()* je možné volat z jedné MATLAB funkce více funkcí psaných v C. Vhodným nastavením můžeme v argumentu funkce předávat pointery a struktury.

Také v Simulinku můžeme volat C funkce. Slouží k tomu blok *MATLAB Function* z knihovny *Simulink/User-Defined Functions* viz obrázek 3.1. Do něj můžeme vložit stejný kód jako je uveden výše. Pak je potřeba nastavit v konfiguraci parametrů modelu zdrojový soubor v položce *Simulation Target – Custom Target – Include list of additional – Source files*. V našem případě do okna zapíšeme *my_power.c*. Nyní můžeme model simulovat, případně z něj vygenerovat zdrojový kód.



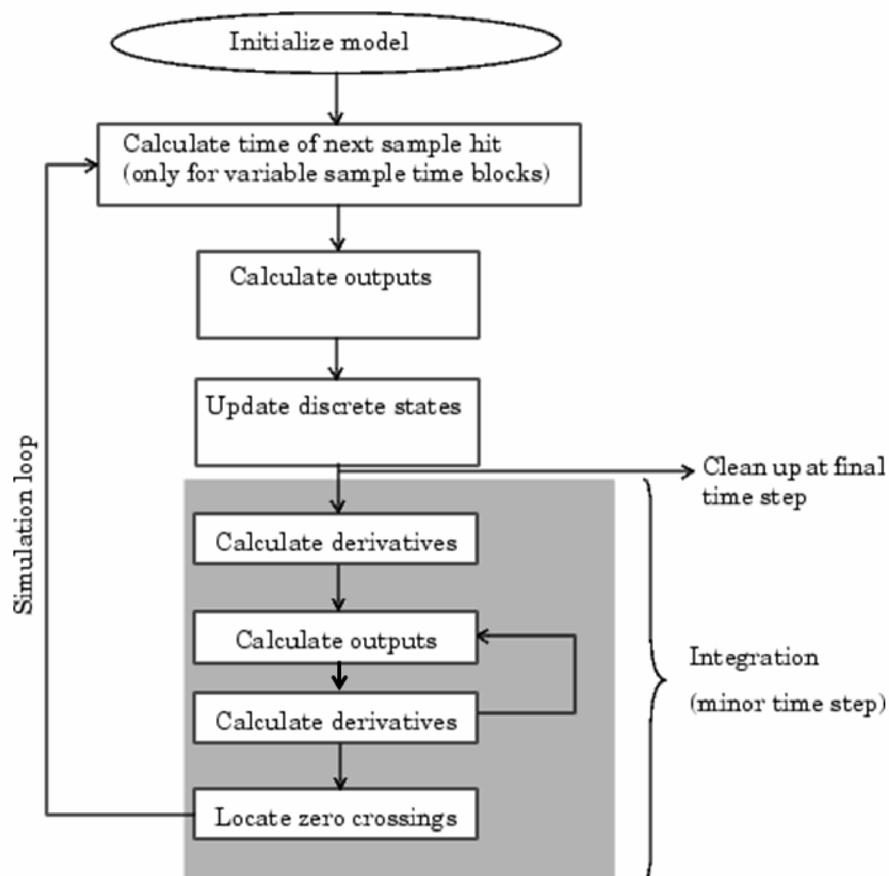
Obrázek 3.1 - Blok MATLAB Function použitý pro volání C funkce

3.2 S-Function Builder

Další možnost implementace C kódu do Simulinku je pomocí bloku *S-Function Builder* z knihovny *Simulink/User-Defined Functions*. Princip spočívá ve vygenerování tzv. S-funkce, kterou následně můžeme použít v modelu. Pro lepší pochopení jak s blokem *S-Function Builder* pracovat si nejdříve vysvětlíme simulační cyklus modelu v Simulinku a princip S-funkce.

3.2.1 Simulační cyklus modelu

Proces simulace sestává z několika dílčích kroků. Prvním z nich je inicializace, ve které je alokována potřebná paměť pro simulaci, dále jsou nastaveny periody vzorkování bloků, datové typy a rozměr vstupních a výstupních portů. Po inicializaci se periodicky vykonává simulační smyčka. Pro bloky s proměnnou periodou vzorkování je nejdříve vypočítán následující časový krok. Následuje výpočet výstupů a poté aktualizace diskrétních stavů. V poslední části *Integration* jsou např. počítány spojité stavy, pokud takové model obsahuje.



Obrázek 3.2 - Simulační cyklus modelu [9]

3.2.2 S-funkce

Popisuje chování bloku v Simulinku. Může být napsána v jazycích MATLAB, C/C++ nebo Fortran. Princip S-funkce vychází ze simulačního cyklu modelu. Nejdříve jsou definovány datové typy a rozměr portů, pak jsou nastaveny vzorkovací periody. Poté se v cyklu počítají výstupy, aktualizují se diskrétní stavy, popřípadě spojité stavy. Podrobnější popis chování S-funkce je uveden v dokumentaci [9].

3.2.3 Implementace C funkce do Simulinku pomocí S-Function Builder

V následujícím příkladě si ukážeme možnou implementaci C funkce realizující integrátor do Simulinku. Jsou použity zdrojové kódy z knihovny RTCESL (REV 4.4). Budeme používat knihovny pro mikrokontrolery ARM Cortex M0+, a to konkrétně *GFLIB_Integrator_A32.h, MLIB_Add_F32.h, gflib.h, gflib_types.h, mlib.h* a *mlib_types.h*. Nyní přejděme k nastavení bloku *S-Function Builder* v Simulinku.

Dvojklikem na blok otevřeme konfigurační okno. Do kolonky *S-function name* zapíšeme jméno výsledné generované S-funkce. V záložce *Initialization* volíme, kolik bude mít funkce diskrétních a spojitých stavů, a také jejich inicializační hodnoty. My zvolíme 1 diskrétní stav, ostatní hodnoty jsou nulové. Mód vzorkování *Inherited*.

Datové typy a rozměr parametrů, vstupních a výstupních portů volíme v záložce *Data Properties*. Pro naši funkci využijeme jeden vstupní a jeden výstupní port. Oba budou skaláry s datovým typem int16 (*Dimensions: 1-D, Rows: 1, Complexity: real, Bus: off, Data type: int16*).

V *Libraries* můžeme vložit knihovny a deklarovat funkce, které bude S-funkce používat. Níže je uveden kód, který napíšeme do *Includes*. Funkci je třeba inicializovat makrem *GFLIB_IntegratorInit_F16()*. Jelikož není možné umístit vlastní kód pro inicializaci, pomůžeme si inicializací při deklaraci proměnných. V dokumentaci RTCESL se více dočteme o frakčních datových typech a strukturách, které jsou v těchto knihovnách používány.

```
#include "gflib.h"
static frac16_t f16Result = FRAC16(0.0);
// sParam - gain, initVal, previousVal
static GFLIB_INTEGRATOR_T_A32 sParam={ACC32(0.1), FRAC32(0),0 };
```

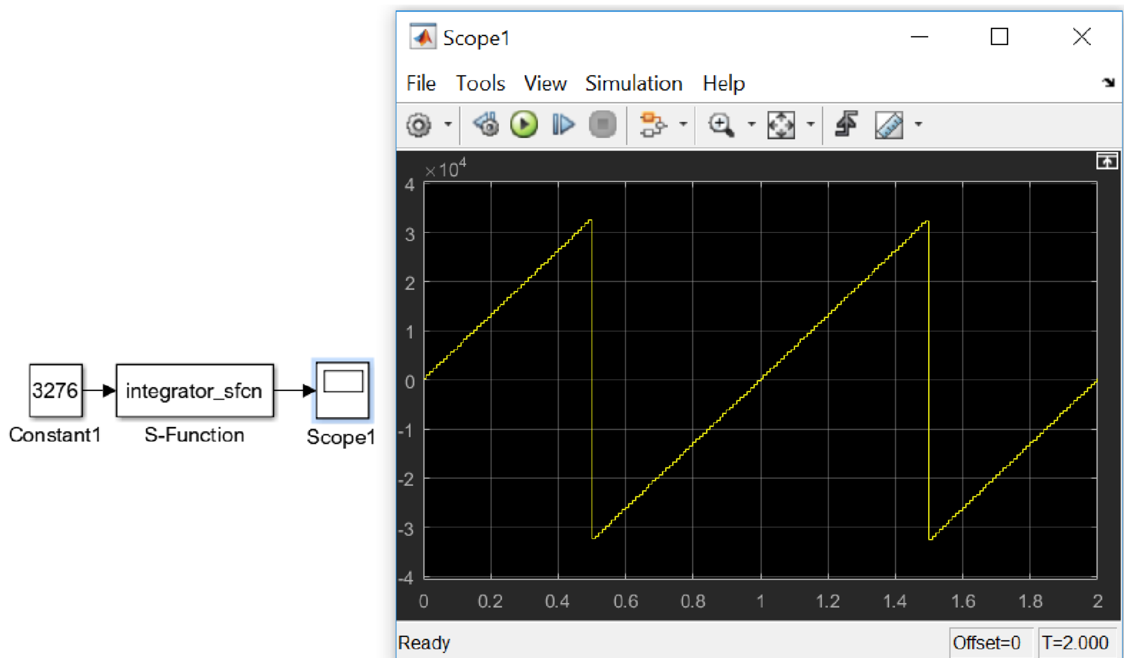
Krok, ve kterém se aktualizují diskrétní stavy, využijeme pro volání makra, které realizuje funkci integrátoru. Do *Discrete Update* zapíšeme:

```
f16Result = GFLIB_Integrator_F16(u0[0], &sParam);
```

Nyní již stačí posílat vypočítanou hodnotu na výstup S-funkce v položce *Outputs*:

```
y0[0] = f16Result;
```

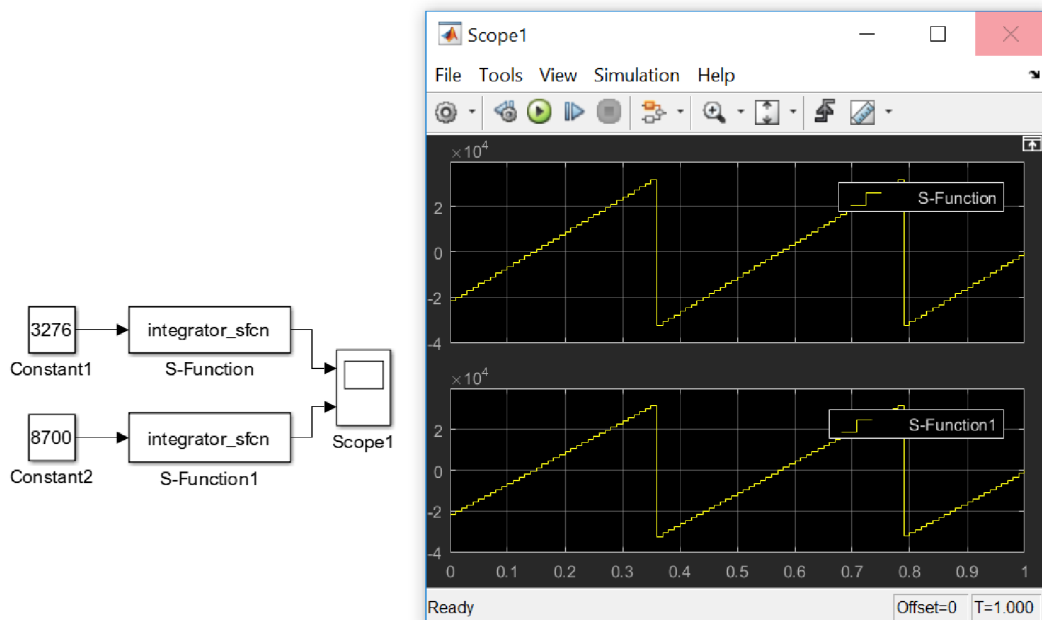
V tomto kroku máme blok nachystaný pro vytvoření S-funkce. Pomocí tlačítka *Build* generujeme žádané soubory, které volíme v *Build Info*. Pro nás je momentálně zajímavý soubor s příponou *mexw64* (případně *mex32*). Tento soubor v sobě obsahuje algoritmus realizující v našem případě integrátor. Jeho výhodou je přenositelnost mezi modely. Do modelu přidáme blok *S-Function*, ve kterém vyplníme parametr *S-functin name*. Po připojení vstupního signálu můžeme po simulaci sledovat průběh na výstupu, jak je zobrazeno na obrázku 3.3.



Obrázek 3.3 - Průběh výstupního signálu integrátoru

Použití *S-Function Builderu* má také své nevýhody, které si vysvětlíme na výše uvedeném příkladu. Jeden z parametrů, který makro přebírá, je ukazatel na strukturu *GFLIB_INTEGRATOR_T_A32*. Tato struktura v sobě nese informaci o zesílení a akumulovaných hodnotách. Pokud bychom chtěli změnit zesílení nebo offset integrátoru, museli bychom tak učinit již v části *Libraries – Includes*. Aby se změny projevily, musíme generovat novou S-funkci. Další z možností je měnit parametry struktury při simulaci. V krocích *Outputs* nebo *Discrete update* bychom je měnili pomocí dalšího vstupního signálu nebo parametru předávaného do S-funkce. I toto

řešení není efektivní, jelikož by se v každém kroku simulace zapisovalo do struktury. Při vhodném použití podmínek by se dalo eliminovat, jak často budeme parametry struktury měnit. Zde však budou algoritmus zpožďovat podmínky. Ačkoliv je vygenerovaná S-funkce přenositelná mezi modely, je prakticky nepoužitelná v případě, že ji ve stejném modelu použijeme vícekrát než jednou. To z toho důvodu, že v S-funkci pracujeme s globálními proměnnými. Simulink nevytvoří pro každý nový blok S-funkce vlastní globální proměnnou. Při simulaci tak bude tato globální proměnná přepisována. Toto chování se projeví již při druhé simulaci. V důsledku se bude posouvat inicializační hodnota (offset) integrátoru. V případě použití dvou a více těchto S-funkcí s rozdílnou hodnotou na vstupu bude výstupní signál pro oba bloky stejný, což je nežádoucí. Tato situace je na obrázku 3.4.



Obrázek 3.4 - Důsledek globálních proměnných S-funkce

Je tedy zřejmé, že tato metodika implementace C kódu do Simulinku má značné omezení. Na druhou stranu může být nápomocná při ručním psaní C MEX S-funkcí, a to díky souborům, které umí generovat. Simulink kromě S-funkce generuje TLC soubor, C MEX S-funkci a soubor napsaný v jazyce C, který zapouzdřuje námi vložený kód do *S-function Builderu* (tzv. *C-wrapper*). TLC soubor specifikuje, jak jsou jednotlivé bloky Simulinku „přeloženy“ do jazyka C. Jelikož je psaní TLC souborů značně náročné, nebudeme se touto problematikou zabývat. C MEX S-funkce obsahuje popis S-funkce v jazyce C. Najdeme zde nastavení portů, parametrů a funkcí, které popisují dílčí kroky S-funkce při simulaci.

3.3 Legacy code tool

Pomocí této metody můžeme vzít jednu nebo více funkcí napsaných v jazyce C, vhodně nastavit datovou strukturu a vygenerovat S-funkci, kterou lze následně volat ze Simulinku. LCT také může vytvářet TLC soubory a C MEX S-funkce. Použití LCT spočívá v nadefinování několika položek předpřipravené struktury, ze které jsou následně generovány požadované výstupní soubory. Níže si opět ukážeme možné začlenění knihovní funkce realizující integrátor, která byla použita v kapitole 3.2 S-Function Builder.

Funkce, resp. makra, které následně volají knihovní funkce integrátoru, přebírají v argumentu ukazatel na strukturu. Tuto strukturu si nejdříve musíme v MATLAB nadefinovat jako sběrnici a přiřadit jí knihovnu, ve které je deklarována:

```
GFLIB_INTEGRATOR_T_A32 = Simulink.Bus;
GFLIB_INTEGRATOR_T_A32.HeaderFile = 'GFLIB_Integrator_A32.h';
```

Poté přidáme jednotlivé prvky struktury s odpovídajícím názvem, rozměrem, aritmetikou a datovým typem. Pro ukázku je níže uvedena definice pouze jednoho parametru ze tří. Všechny prvky struktury nakonec přiřadíme do sběrnice, která bude představovat strukturu.

```
param(1) = Simulink.BusElement;
param(1).Name = 'a32Gain';
param(1).Dimensions = 1;
param(1).DimensionsMode = 'Fixed';
param(1).DataType = 'int32';
...                % pridani zbyvajicich parametru struktury
GFLIB_INTEGRATOR_T_A32.Elements = param;
```

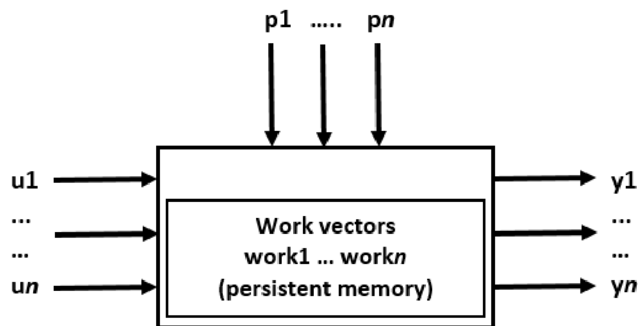
Takto vytvořenou strukturu budeme moci použít jako datový typ argumentů volaných C funkcí. Jelikož knihovna nenabízí funkci, pomocí které by se dal nastavit parametr zesílení ve struktuře *GFLIB_INTEGRATOR_T_A32*, vytvoříme si vlastní inicializační funkci viz příloha č. 2 – *initGain.h*. V této chvíli můžeme přistoupit k nastavení datové struktury LCT.

V prvním kroku vytvoříme prázdnou datovou strukturu, zadáme názvy hlavičkových a zdrojových souborů, a zvolíme název S-funkce. Soubory, se kterými bude MATLAB pracovat, musí být v aktuálně otevřeném adresáři (*Current Folder*), případně musí být v datové struktuře nadefinovaná cesta k těmto souborům.

```
def = legacy_code('initialize');    % vytvoreni datove struktury
def.SourceFiles = {'initGain.c'};
def.HeaderFiles = {'gflib.h','initGain.h'};
```

```
def.SFunctionName = 'integrator';   % nazev S-funkce
```

Abychom dokázali správně implementovat algoritmus, musíme si uvědomit, jak lze pomocí LCT přistupovat k funkcím. Na obrázku níže vidíme, že S-funkce může mít celkem čtyři typy argumentů. Vstupní porty jsou značené u_1 až u_n. Ty jsou „fyzicky“ připojené k bloku tak jako výstupní označované y_1 až y_n. Skrz dialogové okno můžeme do S-funkce posílat parametry p_1 až p_n. Jako vnitřní paměť slouží tzv. pracovní vektory, které se zkráceně značí $work_1$ až $work_n$. V případě použití tohoto posledního typu argumentů závisí funkčnost S-funkce na jeho začlenění do volaných funkcí. Je nutné dávat si pozor, s jakým argumentem volané C funkce tento pracovní vektor „spojíme“. Jen nepatrnou změnou si můžeme nechtěně přepisovat paměť v S-funkci, což může narušit celý algoritmus.



Obrázek 3.5 - Přístup k funkcím pomocí Legacy code tool

LCT umožňuje zadat dvě funkce, které se v průběhu simulace vykonají pouze jednou. Nevýhodou je, že nemůžou pracovat se vstupními nebo výstupními porty. V našem případě přiřadíme parametru *InitializeConditionsFcnSpec* makro z knihovny, které volá inicializační funkci integrátoru. Počáteční podmínku, která bude nastavena, předáváme v modelu přes parametr p_1 v dialogovém okně S-funkce. Strukturu, kterou bude S-funkce používat pro uložení akumulovaných hodnot, budeme ukládat do pracovního vektoru $work_1$. Hranaté závorky značí, že se jedná o ukazatel. Parametr *StartFcnSpec* bude obsahovat funkci, která se zavolá před spouštěním S-funkce. V našem případě zde budeme volat funkci, která byla pro tento účel dodatečně vytvořena. Nastaví ve struktuře zesílení integrátoru na hodnotu, která je předána přes parametr p_2. Funkce je uvedena v příloze č.2 – *initGain.c*.

```
def.InitializeConditionsFcnSpec =
'void GFLIB_IntegratorInit_F16(int16 p1, GFLIB_INTEGRATOR_T_A32 work1[1])';
def.StartFcnSpec =
'void integrator_InitGain(int32 p2, GFLIB_INTEGRATOR_T_A32 work1[1])';
```

Dále definujeme periodicky vykonávanou funkci, která zajišťuje výpočet akumulované hodnoty. Přebírá hodnotu vstupního signálu a ukazatel na strukturu. Vrací akumulovanou hodnotu, která je posílána na výstup y_1.

```
def.OutputFcnSpec =
'int16 y1 = GFLIB_Integrator_F16(int16 u1, GFLIB_INTEGRATOR_T_A32 work1[1])';
```

V posledním kroku navolíme, jaké soubory chceme generovat. Tak jako u S-Function Builderu můžeme nechat vygenerovat C MEX S-funkci. V případě potřeby ji můžeme modifikovat podle sebe, a pak z ní příkazem *mex* generovat spustitelnou S-funkci. Spustitelná S-funkce je opět přenositelná mezi modely. V případě potřeby generování kódu ze Simulinku si necháme vygenerovat TLC soubor. Pokud necháme generovat blok v Simulinku, otevře se nám nový model, ve kterém bude předpřipravený blok S-funkce. Je také možné definovat generování C MEX S-funkce, spustitelné S-funkce a TLC souboru jedním parametrem, a to *generate_for_sim*.

```
legacy_code('sfcn_cmex_generate',def);  % C MEX S-funkce
legacy_code('compile',def);       % spustitelna S-funkce
legacy_code('sfcn_tlc_generate',def);  % TLC soubor
legacy_code('slblock_generate', def);  % blok v Simulinku
legacy_code('generate_for_sim', def);
```

V této chvíli máme nachystané potřebné soubory pro simulaci, případně pro generování kódu. V Simulinku vložíme blok *S-Function*. V dialogovém okně je potřeba zadat název S-funkce a zároveň její parametry. V opačném případě bude program hlásit chybu, která nám neumožní pokračovat dále.

Výhodou LCT oproti S-Function Builderu je možnost inicializačních funkcí. Nevýhodou této metody je, že nemůžeme volat více funkcí napsaných v C v jedné S-funkci. Řešením by bylo napsat jednu C funkci, která bude volat další požadované funkce. To by však mohlo dělat problém, pokud bychom při simulaci nedostávali předpokládané výsledky a museli bychom hledat zdroj chyby. Jednodušší a přehlednější je tedy mít pro každou funkci samostatnou S-funkci. LCT nepodporuje diskrétní stavy. Je však otázkou, jak moc nás toto omezuje, jelikož diskrétní stavy můžou být pouze datového typu double.

3.4 Ručně psané C MEX S-funkce

Nejsložitější metodou začlenění C funkcí je skrz ručně psané C MEX S-funkce. Nabízí však nejvíce možností oproti metodám, které již byli zmíněny výše. Nevýhodu je

nutnost psaní TLC souborů pro tyto funkce, pokud budeme chtít dosáhnout rozumných výsledků při generování kódu.

C MEX S-funkce má v sobě několik dalších funkcí, které jsou odvozeny z jednotlivých kroků, kterými model, resp. S-funkce prochází při simulaci. V těchto dílčích funkcích je definováno chování celé C MEX S-funkce. Volíme zde např. rozměr, datové typy vstupních nebo výstupních portů. V konkrétních funkcích pak můžeme volat C funkce, které chceme začlenit. Jelikož je psaní C MEX S-funkcí náročné, vysvětlíme si princip pouze na jednoduchém příkladě, který je dostupný po instalaci MATLABu a Simulinku. Bude se jednat o funkci *timestwo.c* v adresáři *matlabroot\toolbox\simulink\simdemos\simfeatures\src*. Zdrojový kód je také možno najít v příloze č.2 – timestwo.c.

V první části definujeme název S-funkce, její úroveň a vkládáme knihovnu *simstruc.h*, se kterou pracujeme. V této knihovně je definována datová struktura *SimStruct*, kterou Simulink používá pro ukládání dat S-funkce. Dále jsou v této knihovně definována makra, pomocí kterých pracujeme se signály, které se k S-funkci vztahují.

Následující čtyři funkce jsou nezbytné pro překlad C MEX S-funkce. Nemusí však obsahovat kód. Kompilaci spustíme příkazem *mex nazev_s_funkce.c*, v našem případě tedy *mex timestwo.c*.

V první funkci *mdlInitializeSizes* se nastavují rozměry, datové typy a další vlastnosti vstupních a výstupních portů, parametrů předávaných přes dialogové okno S-funkce a pracovních vektorů. V našem případě bude mít S-funkce jeden vstupní a jeden výstupní port. Rozměr obou bude dynamicky nastavitelný. Signál posílaný na vstup může být nanejvýš vektor.

Funkce *mdlInitializeSampleTimes* se vztahuje ke vzorkování. Naše funkce bude mít stejnou periodu vzorkování jako celý model.

Výpočet výstupních hodnot probíhá ve funkci *mdlOutputs*, která se periodicky opakuje. Nejdříve jsou definovány všechny proměnné a porty. Poté ve *for* cyklu probíhá násobení vstupních signálů a posílání hodnot na výstup, jehož rozměr záleží na rozměru vstupního portu.

Poslední funkce *mdlTerminate* může obsahovat algoritmus, který se vykoná na konci simulace.

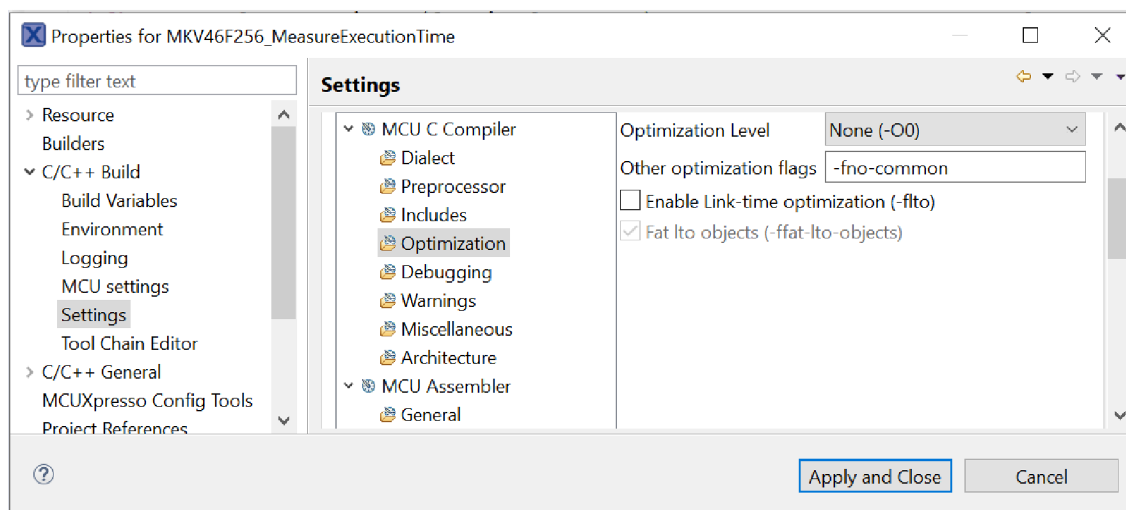
4. KNIHOVNÍ FUNKCE V POROVNÁNÍ S GENEROVANÝM KÓDEM

Následující kapitola se bude zabývat porovnáním optimalizovaných knihovních funkcí a generovaného zdrojového kódu v jazyce C ze Simulinku. Pro tyto účely budeme používat funkce z knihovny RTCESL 4.5 od firmy NXP. Tato knihovna je volně dostupná na stránkách NXP [6]. Než se pustíme do samotného srovnávání, představíme si mikrokontroler, na kterém budou algoritmy spouštěny, a použité vývojové prostředí.

Jak již bylo v úvodu zmíněno, pro otestování algoritmů byl zvolen mikrokontroler z řady KV4x. Konkrétně jde o typ MKV46F256VLL16 s jádrem ARM Cortex M4, který je obohacen o single precision FPU. Pracuje na frekvenci 168MHz. FLASH paměť má 256KB, zatímco SRAM 32KB. Podrobnější informace jsou dohledatelné v referenčním manuálu [3].

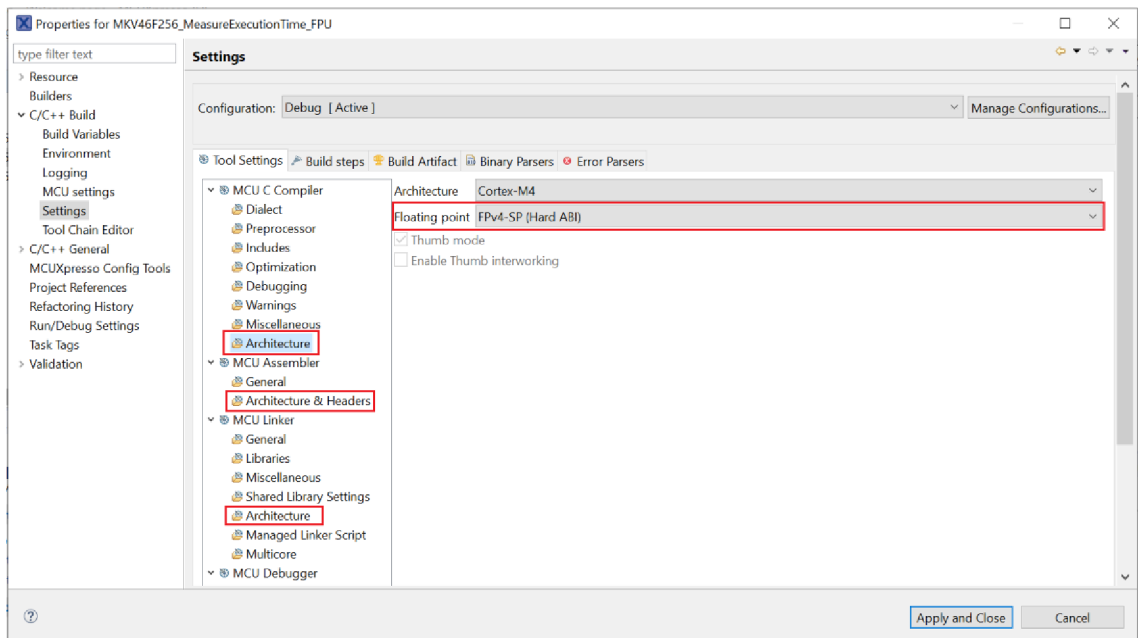
Jako vývojové prostředí bylo použito MCUXpresso IDE verze 10.3.1, do kterého byl importován SDK balíček MKV46F256xxx16. Jak vývojové prostřední, tak i SDK balíček je po registraci volně stažitelný na [7].

Časová náročnost algoritmů byla měřena pro různé vstupní hodnoty funkcí. Zároveň byly využity i optimalizace, které je možné zapnout při překladu viz obrázek 4.1. Pravým tlačítkem na projekt vybereme *Properties*. V záložce *C/C++ Build* rozbalíme *Settings*. Optimalizace se dají zvolit v nastavení kompilátoru v záložce *Optimization* v rozbalovacím políčku *Optimization Level*. Změny je nutné potvrdit tlačítkem *Apply and Close*.



Obrázek 4.1 - Nastavení optimalizací v MCUXpresso

Výpočetní čas je také ovlivněn použitím FPU jednotky. Tu můžeme zapnout v nastavení projektu. Pravým tlačítkem klikneme na projekt a vybereme *Properties*. V záložce *C/C++ Build* rozklikneme *Settings*, kde se nachází další záložky. Ve *Floating point* vybereme *FPv4-SP (Hard ABI)*, a to ve všech záložkách, které jsou červeně vyznačeny na obrázku 4.2. Pokud bychom chtěli FPU jednotku vypnout, vybereme u všech *Floating point* možnost *None*.



Obrázek 4.2 - Zapnutí FPU jednotky v MCUXpresso

4.1 Časovač PIT

Pro měření času, který je potřebný pro výpočet algoritmu, je použit časovač PIT. Jeho inicializace je prostá:

```
/* Enable clock to PIT */
SIM->SCGC6 = SIM_SCGC6_PIT_MASK;

/* Configure PIT page 1035*/
PIT->MCR = PIT_MCR_MDIS(0) | PIT_MCR_FRZ(1) ;
PIT->CHANNEL[0].LDVAL = 0xffffffff;
```

Nejdříve je povolen hodinový signál pro PIT modul. Poté je nastavena frekvence časovače PIT, v našem případě půjde o 21 MHz. Pro korektní odměření času je PIT v módu debugování zakázán pomocí makra *PIT_MCR_FRZ(1)*. Jako poslední zbývá nastavit registr *PIT_LDVAL0*, který udává periodu časovače počítajícího směrem dolů.

Pokud bychom ještě povolili přerušení od PIT, generovalo by se tak každých cca 205 sekund. Pro povolení časovače, vyčtení aktuální hodnoty registru a zakázání časovače slouží opět registry. Je nutné, aby vyčtení z registru *PIT_CVAL* proběhlo dříve, než zastavení časovače zápisem do registru *PIT_TCTRL*. V opačném případě nemáme zaručeno, že přečteme správnou hodnotu.

```
PIT->CHANNEL[0].TCTRL |= PIT_TCTRL_TEN_MASK;         // zapnout PIT0
// mereny algoritmus
StopTime = PIT->CHANNEL[0].CVAL;
PIT->CHANNEL[0].TCTRL &= ~PIT_TCTRL_TEN_MASK;        // vypnout PIT0
```

4.2 Výpočet odmocniny

Pro výpočet odmocniny byly použiti dvě funkce z RTCESL a dvě ze standardní matematické knihovny *math.h*. Není-li jinak nastaveno, Simulink v generovaném kódu nahrazuje výpočet odmocniny právě funkcemi ze standardní matematické knihovny pro jazyk C.

4.2.1 GFLIB_Sqrt

Knihovní funkce z RTCESL je dostupná ve třech variantách. První dvě počítají s čísly v pevné řádové čárce. Slouží bohužel jen pro výpočet odmocniny z čísel v rozsahu <0;1). Pro vyjádření desetinného čísla slouží frakční aritmetika. Deklarace měřené funkce je následující:

```
frac16_t GFLIB_Sqrt_F16(frac16_t f16Val)
```

Jak bylo zmíněno výše, existují dvě varianty počítající s pevnou řádovou čárkou. Ty se liší pouze rozlišením vstupního parametru. *GFLIB_Sqrt_F16* má jako vstupní argument 16bitové frakční číslo, druhá varianta pak číslo 32bitové. Pro přepočet nabízí knihovna makra *FRAC16* a *FRAC32*:

`frac16_t f16Val1 = FRAC16(In1)` **(4.1)**

`frac32_t f16Val2 = FRAC32(In2)` **(4.2)**

kde In1, In2 jsou čísla v rozsahu <0;1), f16Val1 je číslo v rozsahu <-32768;32767>, f16Val2 pak v rozsahu <-2147483648;214783647>.

Třetí varianta již počítá odmocninu s použitím 32bitových plovoucích čísel, tedy čísel s jednoduchou přesností. Deklarace funkce:

```
float_t GFLIB_Sqrt_FLT(float_t fltVal)
```

Následující tabulky zobrazují, jak dlouho trval výpočet algoritmu pro různé vstupní hodnoty a optimalizace.

Tabulka 4.1 - Délka výpočtu funkce GFLIB_Sqrt_F16()

Optimalizace	Vstup	Čítač	Čas [µs]	Vstup	Čítač	Čas [µs]
O0	0	6	0.29	0.5	25	1.19
	0	6	0.29	0.5	25	1.19
	0	7	0.33	0.5	25	1.19
	0	8	0.38	0.5	25	1.19
	0	7	0.33	0.5	26	1.24
O1	0	5	0.24	0.5	20	0.95
	0	6	0.29	0.5	20	0.95
	0	6	0.29	0.5	22	1.05
	0	5	0.24	0.5	20	0.95
	0	5	0.24	0.5	21	1.00
O3	0	5	0.24	0.5	23	1.10
	0	6	0.29	0.5	24	1.14
	0	5	0.24	0.5	22	1.05
	0	4	0.19	0.5	22	1.05
	0	4	0.19	0.5	22	1.05

Tabulka 4.2 - Délka výpočtu funkce GFLIB_Sqrt_FLT()

Optimalizace	Vstup	Čítač	Čas [µs]	Vstup	Čítač	Čas [µs]
O0	0	12	0.57	8192	11	0.52
	0	14	0.67	8192	11	0.52
	0	13	0.62	8192	13	0.62
	0	11	0.52	8192	14	0.67
	0	14	0.67	8192	14	0.67
O1	0	10	0.48	8192	7	0.33
	0	7	0.33	8192	9	0.43
	0	9	0.43	8192	6	0.29
	0	9	0.43	8192	8	0.38
	0	8	0.38	8192	6	0.29
O3	0	5	0.24	8192	6	0.29
	0	7	0.33	8192	7	0.33
	0	6	0.29	8192	7	0.33
	0	7	0.33	8192	7	0.33
	0	7	0.33	8192	6	0.29

Volání funkcí: *GFLIB_Sqrt_F16(FRAC16(Vstup))*; *GFLIB_Sqrt_FLT(Vstup)*. Sloupec *Čítač* udává rozdíl 0xffffffff-PIT_CVAL. Hodnoty v tabulce 4.2 byly měřené při zapnuté FPU jednotce.

4.2.2 Použití funkcí z math.h

Opět byly měřeny dvě funkce z této knihovny – *sqrtf()* a *sqrt()*. Druhá zmíněná pro demonstraci vysokého nárůstu výpočetního času při absenci FPU s dvojitou přesností. Při měření je zapnuta FPU jednotka.

Tabulka 4.3 – Délka výpočtu funkce sqrtf()

Optimalizace	Vstup	Čítač	Čas [µs]	Vstup	Čítač	Čas [µs]
O0	0	15	0.71	8192	14	0.67
	0	15	0.71	8192	16	0.76
	0	16	0.76	8192	17	0.81
	0	16	0.76	8192	16	0.76
	0	15	0.71	8192	17	0.81
O1	0	6	0.29	8192	5	0.24
	0	4	0.19	8192	4	0.19
	0	4	0.19	8192	5	0.24
	0	5	0.24	8192	5	0.24
	0	6	0.29	8192	4	0.19
O3	0	3	0.14	8192	4	0.19
	0	4	0.19	8192	6	0.29
	0	6	0.29	8192	4	0.19
	0	5	0.24	8192	5	0.24
	0	6	0.29	8192	4	0.19

Tabulka 4.4 – Délka výpočtu funkce sqrt()

Optimalizace	Vstup	Čítač	Čas [µs]	Vstup	Čítač	Čas [µs]
O0	0	27	1.29	8192	495	23.57
	0	30	1.43	8192	493	23.48
	0	29	1.38	8192	495	23.57
	0	28	1.33	8192	495	23.57
	0	29	1.38	8192	493	23.48
O1	0	23	1.10	8192	494	23.52
	0	24	1.14	8192	494	23.52
	0	24	1.14	8192	494	23.52
	0	24	1.14	8192	495	23.57
	0	23	1.10	8192	495	23.57

O3	0	23	1.10	8192	497	23.67
	0	24	1.14	8192	497	23.67
	0	24	1.14	8192	497	23.67
	0	25	1.19	8192	496	23.62
	0	23	1.10	8192	496	23.62

Jak můžeme z tabulek vidět, nejrychleji je vypočtena odmocnina pomocí funkce *sqrtf()* při zapnutých optimalizacích O3. Podobných výsledků při stejné optimalizaci lze také dosáhnout s funkcí *GFLIB_Sqrt_FLT()*, kdy se výpočet pohybuje v rozmezí 5 – 7 taktů, což odpovídá asi 0.3 µs. Taktéž se potvrdil předpokládaný nárůst času pro funkci *sqrt()*, která používá čísla s dvojitou přesností. Je tedy zřejmé, že použití této funkce např. při vektorovém řízení motorů není přijatelné v případě použití mikrokontrolerů bez double precision FPU jednotky.

4.3 **Sinus**

Goniometrické funkce jsou často používány při výpočtech souvisejících s vektorovým řízením motorů. Proto i malé časové rozdíly výpočtu dvou funkcí se mohou ve výsledku odrazit na funkčnosti celého algoritmu vektorového řízení.

4.3.1 GFLIB_Sin

Knihovna RTCESL opět nabízí více variant pro výpočet funkce sinus. Všechny pro výpočet používají Taylorův polynom. Funkce pro výpočet v pevné řádové čárce počítá v rozsahu <-π; π), což odpovídá argumentu funkce s rozsahem <-1;1). Vstupem funkce je tedy opět frakční číslo, které je možné zadat pomocí makra. Deklarace funkce:

```
frac16_t GFLIB_Sin_F16(frac16_t f16Angle)
```

Varianta pro pohyblivou řádovou čárku s jednoduchou přesností počítá v rozsahu <-π; π), vstupním argumentem funkce jsou čísla v téže rozsahu. Deklarace funkce:

```
float_t GFLIB_Sin_FLT(float_t fltAngle)
```

Tabulka 4.5 – Délka výpočtu funkce GFLIB_Sin_F16

Optimalizace	Vstup	Čítač	Čas [µs]	Vstup	Čítač	Čas [µs]	Vstup	Čítač	Čas [µs]
O0	0	16	0.76	0.25	18	0.86	0.5	16	0.76
	0	16	0.76	0.25	18	0.86	0.5	17	0.81
	0	17	0.81	0.25	18	0.86	0.5	18	0.86
	0	17	0.81	0.25	17	0.81	0.5	16	0.76
	0	17	0.81	0.25	17	0.81	0.5	16	0.76

O1	0	15	0.71	0.25	13	0.62	0.5	13	0.62
	0	14	0.67	0.25	12	0.57	0.5	11	0.52
	0	14	0.67	0.25	13	0.62	0.5	14	0.67
	0	15	0.71	0.25	13	0.62	0.5	11	0.52
	0	14	0.67	0.25	12	0.57	0.5	13	0.62
O3	0	13	0.62	0.25	13	0.62	0.5	13	0.62
	0	14	0.67	0.25	13	0.62	0.5	13	0.62
	0	14	0.67	0.25	12	0.57	0.5	13	0.62
	0	12	0.57	0.25	14	0.67	0.5	12	0.57
	0	13	0.62	0.25	12	0.57	0.5	13	0.62

Tabulka 4.6 – Délka výpočtu funkce GFLIB_Sin_FLT

Optimalizace	Vstup	Čítač	Čas [µs]	Vstup	Čítač	Čas [µs]	Vstup	Čítač	Čas [µs]
O0	0	21	1.00	0.25π	22	1.05	0.5π	25	1.19
	0	22	1.05	0.25π	23	1.10	0.5π	26	1.24
	0	22	1.05	0.25π	23	1.10	0.5π	23	1.10
	0	24	1.14	0.25π	24	1.14	0.5π	22	1.05
	0	22	1.05	0.25π	22	1.05	0.5π	25	1.19
O1	0	17	0.81	0.25π	18	0.86	0.5π	17	0.81
	0	18	0.86	0.25π	18	0.86	0.5π	16	0.76
	0	17	0.81	0.25π	16	0.76	0.5π	18	0.86
	0	18	0.86	0.25π	17	0.81	0.5π	17	0.81
	0	16	0.76	0.25π	19	0.90	0.5π	18	0.86
O3	0	17	0.81	0.25π	16	0.76	0.5π	17	0.81
	0	16	0.76	0.25π	17	0.81	0.5π	17	0.81
	0	18	0.86	0.25π	16	0.76	0.5π	17	0.81
	0	19	0.90	0.25π	17	0.81	0.5π	18	0.86
	0	17	0.81	0.25π	18	0.86	0.5π	16	0.76

4.3.2 Funkce ze standardní knihovny

Opět byla vybrána funkce z knihovny *math.h*, a to konkrétně *sinf()*. Do funkce je posílán argument *Vstup×(π/180)*. Varianta pro čísla s dvojitou přesností zde není uvedena. Výpočet trvá déle než 30 µs a její použití stojí za zvážení pro mikrokontrolery bez FPU jednotky s dvojitou přesností.

Tabulka 4.7 - Délka výpočtu funkce sinf()

Optimalizace	Vstup [°]	Čítač	Čas [µs]	Vstup [°]	Čítač	Čas [µs]	Vstup [°]	Čítač	Čas [µs]
O0	0	88	4.19	45	100	4.76	90	112	5.33
	0	89	4.24	45	99	4.71	90	114	5.43
	0	89	4.24	45	99	4.71	90	113	5.38
	0	90	4.29	45	100	4.76	90	113	5.38
	0	89	4.24	45	99	4.71	90	113	5.38
O1	0	74	3.52	45	85	4.05	90	99	4.71
	0	75	3.57	45	87	4.14	90	99	4.71
	0	75	3.57	45	87	4.14	90	101	4.81
	0	74	3.52	45	86	4.10	90	101	4.81
	0	73	3.48	45	85	4.05	90	101	4.81
O3	0	73	3.48	45	84	4.00	90	101	4.81
	0	76	3.62	45	85	4.05	90	101	4.81
	0	75	3.57	45	86	4.10	90	101	4.81
	0	74	3.52	45	85	4.05	90	100	4.76
	0	75	3.57	45	85	4.05	90	100	4.76

4.3.3 Funkce generované ze Simulinku

Simulink nabízí několik možností pro výpočet goniometrických funkcí. Při použití bloku *Trigonometric Function* nahrazuje Simulink při generování kódu tento blok funkcemi ze standardní matematické knihovny. Toto platí pro nastavení parametru bloku *Approximation method* na *None*. Pro výchozí nastavení parametrů modelu jsou vkládány funkce počítající s čísly s dvojitou přesností. Vkládání funkcí s jednoduchou přesností pohyblivé řádové čárky vybereme v nastavení parametrů modelu: *Model Configuration Parameters – Code Generation – Interface.* Zde v rozbalovacím okně *Standard math library* zvolíme *C99(ISO).* Důležité také je, aby do bloku vstupoval signál s datovým typem *single*. Pro takto nastavené parametry je doba výpočtu funkce změřena viz kapitola 4.3.2.

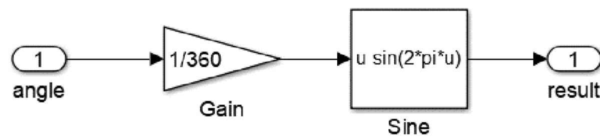
Další změřené funkce počítají pomocí lookup tabulky. Pro tyto varianty slouží blok *Sine*. Parametrem *Internal rule priority for lookup table* je možné nastavit, jestli bude při výpočtu kladen důraz na rychlost nebo na přesnost výsledku. Počet koeficientů pro lookup tabulku volíme pomocí *Number of data points for lookup table*. Jakého datového typu bude výstup funkce ovlivňuje parametr *Output word length*. Ten značí počet bitů, který je potřebný pro uložení výstupní hodnoty. Pro čísla 0 až 7 bude výstup datového typu *int8*, pro čísla 8 až 15 pak *int16* apod. Model, ze kterého byl generován kód pro měření, je na obrázku 4.3. Vstupem je úhel v rozsahu 0° až 360°.

Tabulka 4.8 - Délka výpočtu algoritmu pro blok Sine, Internal rule priority for lookup table: Speed

Optimalizace	Vstup [°]	Čítač	Čas [μs]	Vstup [°]	Čítač	Čas [μs]	Vstup [°]	Čítač	Čas [μs]
O0	0	96	4.57	45	116	5.52	90	115	5.48
	0	95	4.52	45	117	5.57	90	115	5.48
	0	96	4.57	45	118	5.62	90	116	5.52
	0	95	4.52	45	117	5.57	90	115	5.48
	0	96	4.57	45	118	5.62	90	115	5.48
O1	0	63	3.00	45	79	3.76	90	77	3.67
	0	62	2.95	45	78	3.71	90	76	3.62
	0	62	2.95	45	78	3.71	90	76	3.62
	0	62	2.95	45	79	3.76	90	76	3.62
	0	64	3.05	45	78	3.71	90	77	3.67
O3	0	63	3.00	45	83	3.95	90	82	3.90
	0	62	2.95	45	83	3.95	90	81	3.86
	0	63	3.00	45	84	4.00	90	83	3.95
	0	63	3.00	45	84	4.00	90	81	3.86
	0	63	3.00	45	83	3.95	90	83	3.95

Tabulka 4.9 - Délka výpočtu algoritmu pro blok Sine, Internal rule priority for lookup table: Precision

Optimalizace	Vstup [°]	Čítač	Čas [μs]	Vstup [°]	Čítač	Čas [μs]	Vstup [°]	Čítač	Čas [μs]
O0	0	96	4.57	45	117	5.57	90	115	5.48
	0	97	4.62	45	117	5.57	90	116	5.52
	0	96	4.57	45	117	5.57	90	116	5.52
	0	97	4.62	45	118	5.62	90	116	5.52
	0	96	4.57	45	117	5.57	90	115	5.48
O1	0	64	3.05	45	78	3.71	90	77	3.67
	0	63	3.00	45	78	3.71	90	77	3.67
	0	62	2.95	45	79	3.76	90	76	3.62
	0	63	3.00	45	77	3.67	90	76	3.62
	0	62	2.95	45	78	3.71	90	77	3.67
O3	0	38	1.81	45	63	3.00	90	63	3.00
	0	37	1.76	45	63	3.00	90	63	3.00
	0	38	1.81	45	65	3.10	90	63	3.00
	0	37	1.76	45	64	3.05	90	62	2.95
	0	37	1.76	45	65	3.10	90	64	3.05



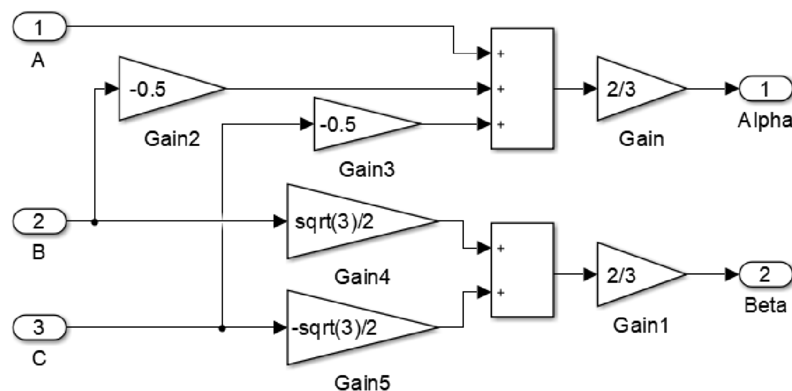
Obrázek 4.3 - Model funkce počítající sinus pomocí lookup tabulky

Pro tabulky 4.8 a 4.9 byl model z obrázku 4.3 zabalen do subsystému a nastaven jako *nonreusable*. Počet koeficientů pro lookup tabulky byl 33 a parametr *Output word length*: 17, tedy výstup s datovým typem int32. Přestože výstupní signál je celočíselného datového typu, tak funkce obsahuje výpočty v pohyblivé řádové čárce. Nevýhodou je nečitelnost generovaného kódu. Dále se projevilo odlišné chování, než jaké bylo předpokládáno. Při nastavení parametru *Internal rule priority for lookup table*: *Speed* trvá výpočet déle než pro volbu *Precision*, což je možné vidět v tabulkách 4.8 a 4.9. Proč k tomuto došlo nebylo zjištěno. Vygenerovaný kód pro tyto funkce je v příloze 1.

Pokud bychom měli zvolit, kterou z výše uvedených variant pro goniometrické funkce vybereme, nejspíše bychom volili některou z knihovny RTCESL. Časově jsou o poznání rychlejší. Další výhodou je optimalizovaná funkce pro mikrokontrolery bez FPU jednotky, jejíž výpočet trvá přibližně 0.6 µs. Pro variantu s pohyblivou řádovou čárkou je výpočet o 0.2 µs delší. Těchto časů je možné dosáhnout při nastavení optimalizací O1 nebo O3.

4.4 Clarkové transformace

Slouží pro převod soustavy z trojfázového systému *abc* na dvojfázový systém *αβ*. Transformace bude popsána níže v kapitole 5.1. Měřeny byly dvě varianty funkce. První je funkce z knihovny RTCESL, která je pro čísla s pohyblivou řádovou čárkou. Druhá představuje algoritmus, který byl vygenerován z modelu, který je na obrázku 4.4.



Obrázek 4.4 - Model Clarkové transformace

Pro vstupní a výstupní porty je nastaven datový typ *single*, stejně tak zesílení a suma mají výstupní signál datového typu *single*.

4.4.1 GMCLIB_Clark

Jak již bylo řečeno, byla měřena funkce pro čísla s pohyblivou řádovou čárkou, a to s jednoduchou přesností. Knihovna nabízí i variantu pro čísla s pevnou řádovou čárkou. Deklarace měřené funkce:

```
void GMCLIB_Clark_FLT(const GMCLIB_3COOR_T_FLT *psIn, GMCLIB_2COOR_ALBE_T_FLT *psOut)
```

První argument funkce je ukazatel na strukturu reprezentující trojfázovou soustavu *abc*. Druhý argument je ukazatel na strukturu, do níž se po výpočtu uloží hodnoty odpovídající dvojfázové soustavě $\alpha\beta$.

Tabulka 4.10 - Délka výpočtu funkce GMCLIB_Clark_FLT

Optimalizace	Vstup			Čítač	Čas [us]	Vstup			Čítač	Čas [us]	Vstup			Čítač	Čas [us]
	A	B	C			A	B	C			A	B	C		
O0	0.5	0.5	-1	24	1.14	0.5	-1	0.5	22	1.05	-1	0.5	0.5	23	1.10
	0.5	0.5	-1	23	1.10	0.5	-1	0.5	23	1.10	-1	0.5	0.5	22	1.05
	0.5	0.5	-1	24	1.14	0.5	-1	0.5	24	1.14	-1	0.5	0.5	24	1.14
	0.5	0.5	-1	22	1.05	0.5	-1	0.5	22	1.05	-1	0.5	0.5	23	1.10
	0.5	0.5	-1	24	1.14	0.5	-1	0.5	24	1.14	-1	0.5	0.5	23	1.10
O1	0.5	0.5	-1	5	0.24	0.5	-1	0.5	6	0.29	-1	0.5	0.5	4	0.19
	0.5	0.5	-1	4	0.19	0.5	-1	0.5	7	0.33	-1	0.5	0.5	5	0.24
	0.5	0.5	-1	5	0.24	0.5	-1	0.5	2	0.10	-1	0.5	0.5	4	0.19
	0.5	0.5	-1	6	0.29	0.5	-1	0.5	5	0.24	-1	0.5	0.5	4	0.19
	0.5	0.5	-1	6	0.29	0.5	-1	0.5	6	0.29	-1	0.5	0.5	5	0.24
O3	0.5	0.5	-1	2	0.10	0.5	-1	0.5	2	0.10	-1	0.5	0.5	2	0.10
	0.5	0.5	-1	3	0.14	0.5	-1	0.5	3	0.14	-1	0.5	0.5	2	0.10
	0.5	0.5	-1	2	0.10	0.5	-1	0.5	1	0.05	-1	0.5	0.5	2	0.10
	0.5	0.5	-1	2	0.10	0.5	-1	0.5	1	0.05	-1	0.5	0.5	2	0.10
	0.5	0.5	-1	2	0.10	0.5	-1	0.5	2	0.10	-1	0.5	0.5	2	0.10

4.4.2 Algoritmus generovaný Simulinkem

Algoritmus vygenerovaný z modelu na obrázku 4.4 spočívá pouze v násobení a sčítání. Simulink nahradil výraz pro druhou odmocninu v zesílení *Gain4* a *Gain5* číslem. Vygenerovaný kód je následující:

```
mdl_Y.Alpha = ((-0.5F * mdl_U.B + mdl_U.A) + -0.5F * mdl_U.C) * 0.666666687F;
mdl_Y.Beta = (0.866025388F * mdl_U.B + -0.866025388F * mdl_U.C) * 0.666666687F;
```

Jak můžeme vidět, Simulink při generování používá struktury, ve kterých má uložené vstupní a výstupní signály (proměnné).

Tabulka 4.11 - Délka výpočtu funkce generované Simulinkem pro Clarkové transformaci

Optimalizace	Vstup			Čítač	Čas [us]	Vstup			Čítač	Čas [us]	Vstup			Čítač	Čas [us]
	A	B	C			A	B	C			A	B	C		
O0	0.5	0.5	-1	23	1.10	0.5	-1	0.5	23	1.10	-1	0.5	0.5	23	1.10
	0.5	0.5	-1	23	1.10	0.5	-1	0.5	23	1.10	-1	0.5	0.5	23	1.10
	0.5	0.5	-1	23	1.10	0.5	-1	0.5	23	1.10	-1	0.5	0.5	23	1.10
	0.5	0.5	-1	22	1.05	0.5	-1	0.5	23	1.10	-1	0.5	0.5	22	1.05
	0.5	0.5	-1	23	1.10	0.5	-1	0.5	23	1.10	-1	0.5	0.5	22	1.05
O1	0.5	0.5	-1	13	0.62	0.5	-1	0.5	13	0.62	-1	0.5	0.5	13	0.62
	0.5	0.5	-1	13	0.62	0.5	-1	0.5	13	0.62	-1	0.5	0.5	13	0.62
	0.5	0.5	-1	13	0.62	0.5	-1	0.5	13	0.62	-1	0.5	0.5	12	0.57
	0.5	0.5	-1	12	0.57	0.5	-1	0.5	12	0.57	-1	0.5	0.5	13	0.62
	0.5	0.5	-1	13	0.62	0.5	-1	0.5	13	0.62	-1	0.5	0.5	13	0.62
O3	0.5	0.5	-1	11	0.52	0.5	-1	0.5	13	0.62	-1	0.5	0.5	11	0.52
	0.5	0.5	-1	12	0.57	0.5	-1	0.5	12	0.57	-1	0.5	0.5	11	0.52
	0.5	0.5	-1	11	0.52	0.5	-1	0.5	12	0.57	-1	0.5	0.5	11	0.52
	0.5	0.5	-1	12	0.57	0.5	-1	0.5	13	0.62	-1	0.5	0.5	11	0.52
	0.5	0.5	-1	12	0.57	0.5	-1	0.5	13	0.62	-1	0.5	0.5	11	0.52

Ač jde o pouhé násobení a sčítání, tak se opět ukázala jako rychlejší funkce z RTCESL. Při optimalizacích O3 je *GMCLIB_Clark_FLT* až pětkrát rychlejší než algoritmus generovaný Simulinkem, což při větším počtu těchto transformací může ušetřit dostatek času.

5. SYNCHRONNÍ MOTOR S PERMANENTNÍMI MAGNETY

Synchronní motory s permanentními magnety jsou hojně využívány díky své vysoké účinnosti a dobrému poměru výkon k rozměru Kompaktní rozměry jsou zajištěny díky dobrému poměru momentu k průměru rotoru. Tyto motory naleznou své uplatnění v oblasti servomechanizmů, čerpadel, ventilátorů nebo také v domácích spotřebičích.

Stator je tvořen třífázovým vinutím, přičemž jednotlivé fáze jsou posunuty o 120°. Rotor obsahuje permanentní magnety, které přispívají k vytvoření magnetického pole ve vzduchové mezeře. Po připojení harmonického napětí na statorové vinutí vzniká díky průchodu elektrického proudu točivé magnetické pole, které roztáčí rotor s permanentními magnety. Rychlost rotoru je stejná jako rychlost točivého magnetického pole statorového vinutí. Otáčky rotoru a magnetického pole vinutí jsou tak synchronní.

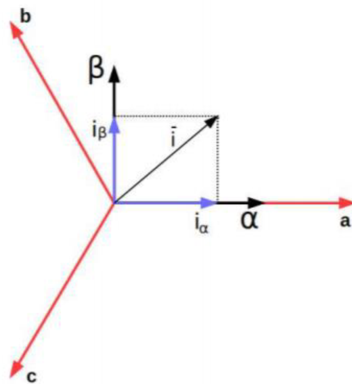
5.1 Transformace

Modelovat a popisovat PMS motory můžeme v několika souřadnicových systémech. Za základní můžeme považovat tříosý souřadnicový systém *abc*, jelikož osy *abc* odpovídají statorovým vinutím motoru. Zavedením Clarkové transformace pomocí rovnice (5.1) získáme model ve dvouosém souřadnicovém systému *αβ*.

$$\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} = \frac{2}{3} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} \tag{5.1}$$

Zpětná Clarkové transformace (5.2) slouží pro přepočet systému v *αβ* souřadnicích do třísého systému *abc*.

$$\begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} \tag{5.2}$$



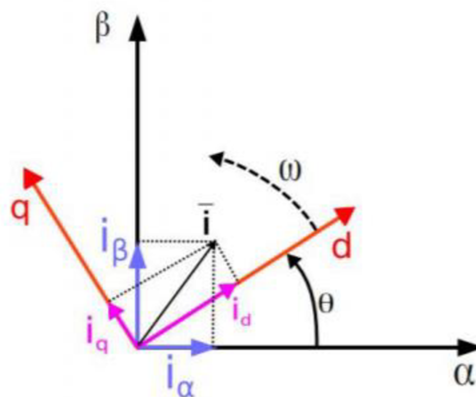
Obrázek 5.1 - Clarkové transformace [10]

Další souřadnicový systém je dq. Ten je spojený s rotorem motoru otáčejícího se úhlovou rychlostí ω. Je popsán pomocí dvou os d, q. Systém v dq souřadnicích je oproti systému v $\alpha\beta$ souřadnicích posunut o úhel ϑ. Pro převod mezi souřadnicovými systémy $\alpha\beta$ a dq se používá Parkovy transformace (5.3).

$$\begin{bmatrix} i_d \\ i_q \end{bmatrix} = \begin{bmatrix} \cos\vartheta & \sin\vartheta \\ -\sin\vartheta & \cos\vartheta \end{bmatrix} \begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} \quad \textbf{(5.3)}$$

Pro přenesení modelu ze souřadnicového systému dq zpět do systému $\alpha\beta$ slouží zpětná Parkova transformace (5.4).

$$\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} = \begin{bmatrix} \cos\vartheta & -\sin\vartheta \\ \sin\vartheta & \cos\vartheta \end{bmatrix} \begin{bmatrix} i_d \\ i_q \end{bmatrix} \quad \textbf{(5.4)}$$



Obrázek 5.2 - Parkova transformace [10]

5.2 Matematický model PMS motoru

Pro popis matematického modelu synchronního motoru s permanentními magnety budeme uvažovat následující zjednodušující předpoklady (Doporučená literatura [1]):

- Průběh magnetické indukce ve vzduchové mezeře je sinusový
- Parametry (R, L) jsou konstantní a stejné ve všech třech fázích
- Ztráty v železe jsou zanedbány
- Nulový vodič není připojen

Elektromagnetické a elektromechanické děje jsou v souřadnicovém systému *dq* popsány těmito rovnicemi:

$$u_d = L_d \frac{di_d}{dt} + R_s i_d - \omega_e L_q i_q \quad \textbf{(5.5)}$$

$$u_q = L_q \frac{di_q}{dt} + R_s i_q + \omega_e L_d i_d + \omega_e \psi_{pm} \quad \textbf{(5.6)}$$

kde u_d, u_q představuje napětí v *dq* souřadnicích

i_d,i_q jsou složky statorového proudu

R_s je statorový odpor

L_d, L_q jsou složky indukčností v osách *d* a *q* souřadného systému *dq*

ω_e je elektrická úhlová rychlost

ψ_{pm} je magnetický tok od permanentních magnetů

Generovaný elektromagnetický moment udává rovnice (5.7):

$$T_e = \frac{3}{2} p_p \left(\psi_{pm} i_q + \left(L_d - L_q \right) i_d i_q \right) \quad \textbf{(5.7)}$$

kde p_p udává počet pólových dvojic.

Mechanická rychlost je dána výsledným momentem a setrvačností rotoru:

$$\frac{d\omega}{dt} = \frac{1}{J} \left(T_e - T_{load} \right) \quad \textbf{(5.8)}$$

ω je mechanická úhlová rychlost

T_e je elektromagnetický moment

T_{load} je zatěžovací moment

J je moment setrvačnosti rotoru

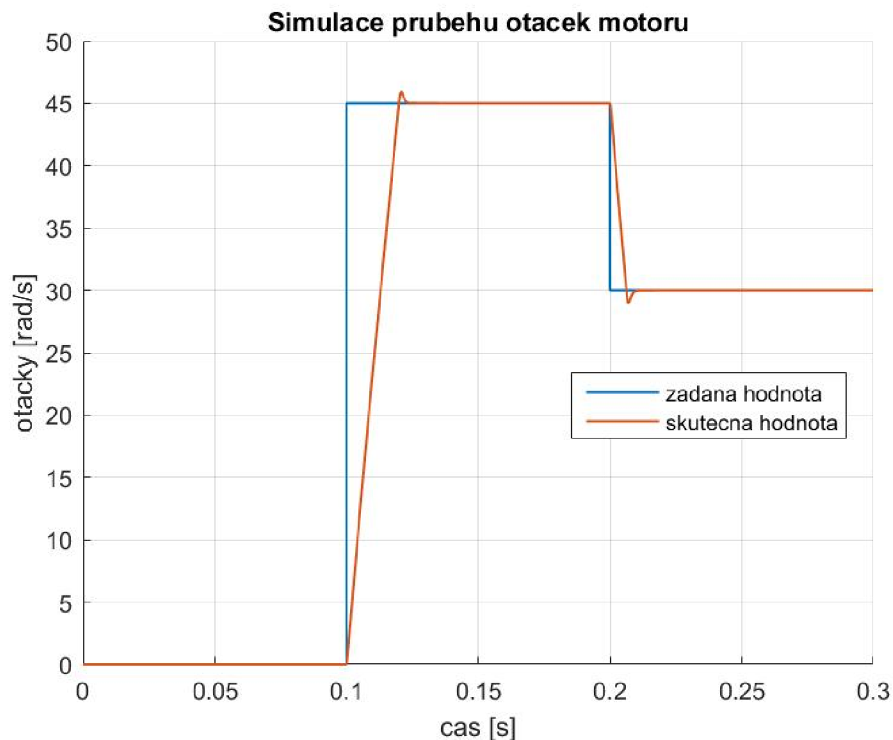
Vztah mezi mechanickou a elektrickou rychlostí udává rovnice (5.9):

$$\omega_e = p_p \omega \quad \textbf{(5.9)}$$

Pro simulaci v této práci jsou použity parametry PMS motoru převzaté z doporučené literatury [1]:

- ψ_{pm} = 0.378 Wb
- J = 0.0104 kg·m^2
- R_s = 0.14 Ω
- L_d = L_q = 1.29 mH
- p_p = 3

Na obrázku 5.3 je simulace průběhu otáček motoru při zatěžovacím momentu T_{load} = 0.001 Nm. Motor je modelován v souřadnicovém systému *dq* podle rovnic (5.5) až (5.8). Při simulaci byly použity regulátory navržené v kapitole 6.1. Výpočetní metoda simulace byla *ode23 (Bogacki-Shampine)*.



Obrázek 5.3 - Simulace otáček PMS motoru

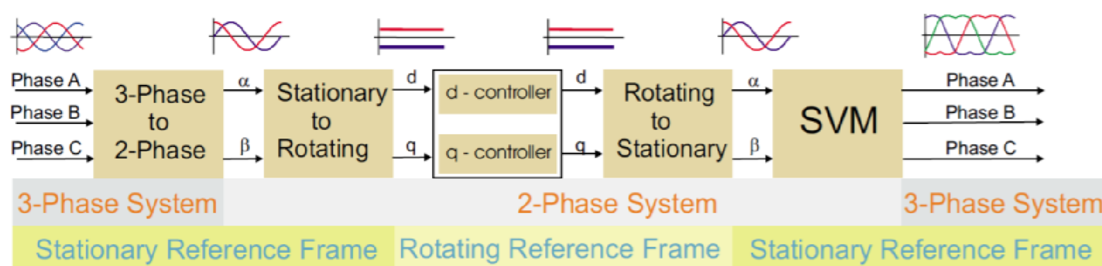
6. VEKTOROVÉ ŘÍZENÍ

Vektorové řízení je jedna z metod pro řízení synchronního motoru s permanentními magnety. V anglické literatuře se setkáme s pojmem Field-Oriented Control (FOC). Při vektorovém řízení můžeme vycházet ze struktury, která je na obrázku 6.2. Bývá rozdělena na rychlejší a pomalejší smyčku, přičemž rychlejší je proudová smyčka a v pomalejší se počítá akční zásah regulátoru otáček.

V prvním kroku jsou změřeny velikosti proudů jednotlivých fází *a, b, c*. Tyto hodnoty jsou pomocí Clarkové a Parkovy transformace převedeny do složek *d, q*. Poté následuje regulační proces složek *d, q*. Velikost akčního zásahu je přepočítána do souřadnic α, β. Blok *SVM PWM* nakonec zajišťuje generování stříd pulsní šířkové modulace. Pro simulaci nám však stačí místo bloku *SVM PWM* použít transformaci Clarkové, kde pak fáze *a, b, c* připojíme přímo na model motoru, pokud jej máme namodelován v trojfázovém systému *abc*.

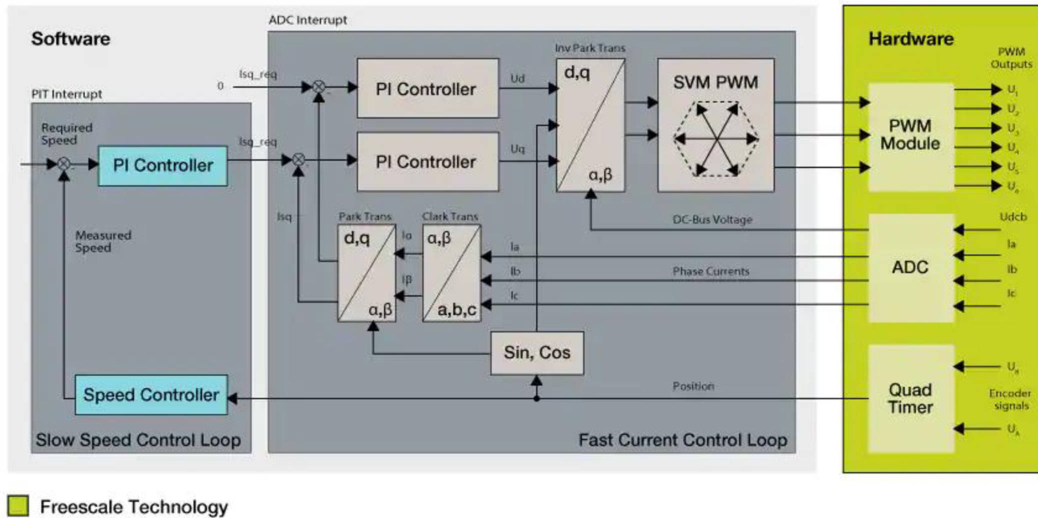
Jak můžeme na obrázku 6.2 vidět, algoritmus vektorového řízení potřebuje znát polohu rotoru. Polohu můžeme získat buď měřením nebo estimací na základě stavu motoru.

Na obrázku 6.1 jsou znázorněny průběhy signálů, které prochází řídicí strukturou vektorového řízení. Složku *d* někdy nazýváme „tokotvornou“, zatímco složku *q* „momentotvornou“.



Obrázek 6.1 - Transformace signálů ve vektorovém řízení [12]

Pokud se indukčnost L_d rovná indukčnosti L_q, bude moment motoru dán pouze součinem složky i_q, počtem pólových dvojic a magnetickým tokem od permanentních magnetů (viz rovnice (5.7)). Žádanou hodnotu složky i_d pak můžeme regulovat na nulovou hodnotu, jak je zobrazeno na obrázku 6.2.



Obrázek 6.2 – Řídicí struktura vektorového řízení PMS motoru [11]

6.1 Návrh regulátorů

V této kapitole bude nastíněno nastavení regulátorů proudů a regulátoru otáček. Budeme vycházet z rovnic (5.5) až (5.8), ze kterých můžeme vyjádřit přibližný přenos dílčích soustav a na ty pak navrhovat regulátory.

Přenos soustavy, kterou budeme uvažovat pro proudovou smyčku i_d, odvodíme z rovnice (5.5):

$$F_{SI}(p) = \frac{i_d(p)}{u_d(p)} = \frac{1}{R_S + pL_d} = \frac{\frac{1}{R_S}}{1 + p\frac{L_d}{R_S}} = \frac{K_{SI}}{1 + pT_{SI}} \qquad \textbf{(6.1)}$$

kde p je Laplaceův operátor. Zesílení soustavy je $K_{SI} = 1/R_s$, časová konstanta soustavy $T_{SI} = L_d/R_s$. Zvolíme PI regulátor, který vykompenzuje pól soustavy F_{SI}. Výsledný přenos řízení pak bude mít vlastnosti setrvačného článku. Přenos PI regulátoru bude:

$$F_{RI}(p) = \frac{K_{RI}(1 + pT_{RI})}{p} \qquad \textbf{(6.2)}$$

Pokud $T_{RI} = T_{SI}$, bude přenos otevřené smyčky:

$$F_{0I}(p) = F_{SI}(p) \cdot F_{RI}(p) = \frac{K_{SI}K_{RI}}{p} = \frac{K_0}{p} \qquad \textbf{(6.3)}$$

kde $K_0 = K_{SI} \cdot K_{RI}$ je zesílení otevřené smyčky. Přenos řízení proudové smyčky i_d:

$$F_{WI}(p) = \frac{F_{0I}(p)}{1+F_{0I}(p)} = \frac{1}{p\frac{1}{K_0}+1} \qquad \textbf{(6.4)}$$

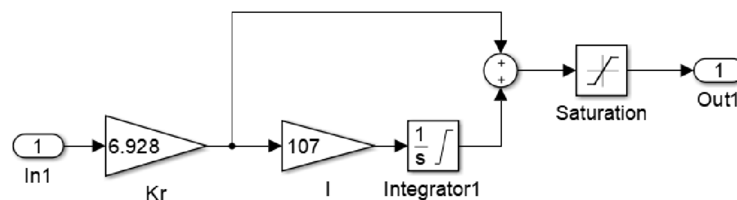
Regulátor proudové smyčky i_q bude odvozen stejně jako regulátor proudové smyčky i_d. Díky rovnosti indukčnosti v podélné a příčné složce ($L_d = L_q$) budou přenosy těchto dvou regulátorů totožné. Taktéž i přenosy řízení obou smyček jsou stejné.

Soustava, na kterou budeme navrhovat regulátor rychlosti, sestává z přenosu řízení proudové smyčky i_q a soustavy, kterou lze vyjádřit z rovnic (5.7) a (5.8). Předpokládejme nulový zatěžovací moment ($T_{load} = 0$). Výsledný přenos je pak:

$$F_{sw}(p) = F_{WI}(p) \cdot \frac{3p_p\psi_{pm}}{2J} \cdot \frac{1}{p} = \frac{1}{\frac{1}{K_0}p+1} \cdot \frac{3p_p\psi_{pm}}{2Jp} \qquad \textbf{(6.5)}$$

Regulátor otáček zvolíme PI s přenosem jako v (6.2). Přenos otevřené smyčky bude mít dva póly v počátku a jeho frekvenční charakteristika tak začíná se sklonem -40 dB na dekádu. Nulu regulátoru se pokusíme umístit tak, abychom zajistili sklon -20 dB na jednu dekádu frekvenční charakteristiky otevřeného obvodu. Zbývá doladit proporcionální zesílení regulátoru. Frekvenční charakteristiku otevřeného obvodu posuneme tak, aby osu 0 dB protínala zhruba v polovině části se sklonem -20 dB na dekádu.

Pro parametry PMS motoru z kapitoly 5.2 byly pomocí programu MATLAB a jeho nástroje Sisotool navrhnuty PI regulátory proudů a PI regulátor otáček. Všechny regulátory mají stejnou strukturu, která je na obrázku 6.3. Přenos regulátoru popisuje rovnice (6.6). Navržené parametry můžeme vidět v tabulce 6.1.



Obrázek 6.3 - Struktura navržených PI regulátorů

Parametry z tabulky 6.1 platí pro přenos s tvarem:

$$F_R(p) = K_r(1 + I\frac{1}{p}) \qquad \textbf{(6.6)}$$

Tabulka 6.1 - Parametry navržených PI regulátorů

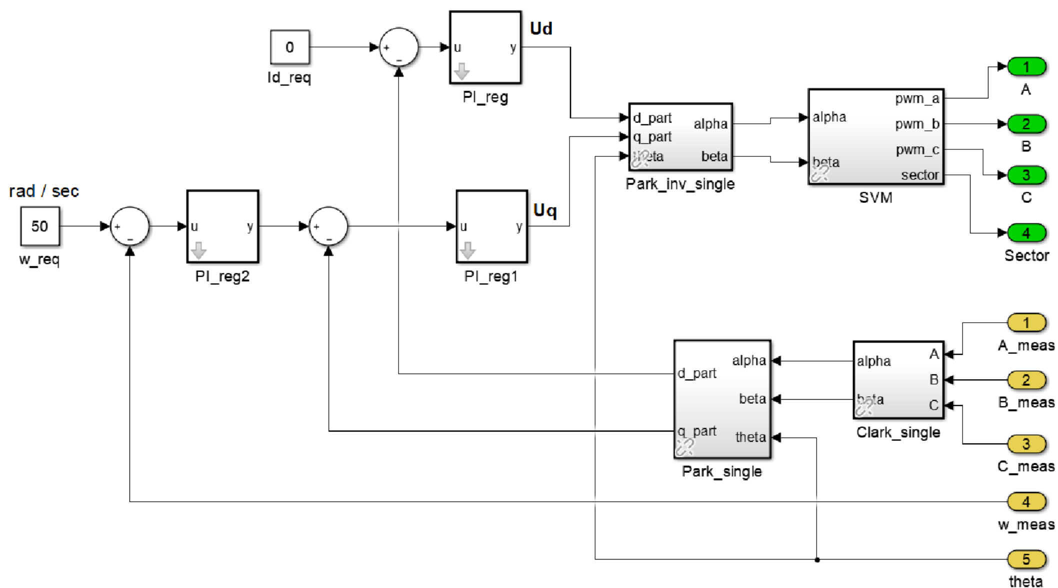
Regulovaná soustava	Kr	I	Saturace
Proudová smyčka id	6.928	107	±100
Proudová smyčka iq	6.928	107	±100
Otáčky	11.9	714.3	±16

Pro regulátory bylo nastaveno i omezení, které se týká jak bloku integrátoru, tak bloku saturace.

6.2 Vektorové řízení z knihovních funkcí RTCESL

Jak již bylo v úvodu této práce nastíněno, knihovna RTCESL nabízí algoritmy pro vektorové řízení. Z vybraných funkcí této knihovny sestavíme řídicí strukturu FOC, změříme čas potřebný pro výpočet vektorového řízení, a nakonec uvedeme výhody a nevýhody používání algoritmů z RTCESL.

Struktura kódu a uspořádání funkcí bude vycházet z modelu na obrázku 6.4. Jak můžeme vidět, model obsahuje pouze základní prvky pro FOC. Chybí zde části, bez kterých by v reálné situaci řízení motoru postrádalo smysl. Jedná se např. o měření proudů vinutím nebo měření, respektive výpočet polohy rotoru. My budeme předpokládat situaci, kdy již tyto hodnoty známe. Tyto hodnoty však nebudou odpovídat reálné situaci. Dále poznamenejme, že budeme používat funkce pro celočíselnou aritmetiku. V tabulce 6.2 jsou uvedeny, které knihovní funkce jsou vybrány pro konkrétní bloky z modelu na obrázku 6.4.



Obrázek 6.4 - Model řídicí struktury vektorového řízení

Tabulka 6.2 - Vybrané funkce RTCESL pro bloky z modelu na obrázku 6.4

Model	Funkce RTCESL
Clark_single	GMCLIB_Clark_F16
Park_single	GMCLIB_Park_F16
Park_inv_single	GMCLIB_ParkInv_F16
SVM	GMCLIB_SvmStd_F16
PI_reg, PI_reg1, PI_reg2	PCLIB_CtrlPI_F16
Suma	MLIB_Sub_F16

Výsledný algoritmus poskládaný z knihovních funkcí je následující:

```
sAbcClark.f16A = A_meas;
sAbcClark.f16B = B_meas;
sAbcClark.f16B = C_meas;
sAnglePark.f16Sin = GFLIB_Sin_F16(theta);
sAnglePark.f16Cos = GFLIB_Cos_F16(theta);

// transformace
GMCLIB_Clark_F16(&sAbcClark, &sAlphaBetaClark);
GMCLIB_Park_F16(&sAlphaBetaClark, &sAnglePark, &sDQPark);

// akcni zasah otacky
f16InErrPIw = MLIB_Sub_F16(FRAC16(0.4),w_meas);            // suma
f16ResultPIw = PCLIB_CtrlPI_F16(f16InErrPIw, &sParamPIw);   // regulator otacek

// akcni zasah slozka Iq
f16InErrPIq = MLIB_Sub_F16(f16ResultPIw,sDQPark.f16Q);     // suma
f16ResultPIq = PCLIB_CtrlPI_F16(f16InErrPIq, &sParamPIq);   // regulator Iq

// akcni zasah slozka Id
f16InErrPId = MLIB_Sub_F16(FRAC16(0),sDQPark.f16D);        // suma
f16ResultPId = PCLIB_CtrlPI_F16(f16InErrPId, &sParamPId);   // regulator Iq

// Park inverse
sDQParkInv.f16D = f16ResultPId;
sDQParkInv.f16Q = f16ResultPIq;
GMCLIB_ParkInv_F16(&sDQParkInv, &sAnglePark, &sAlphaBetaParkInv);

// SVM
u16SectorSVM = GMCLIB_SvmStd_F16(&sAlphaBetaParkInv, &sAbcSVM);
Sector = u16SectorSVM;
A = sAbcSVM.f16A;
B = sAbcSVM.f16B;
C = sAbcSVM.f16C;
```

V příloze 4 je celý zdrojový kód vektorového řízení, který obsahuje i deklaraci a inicializaci proměnných a struktur. Ještě jednou zmíníme, že hodnoty vstupující do algoritmů nejsou korektní. Slouží pouze pro algoritmus, aby nepočítal s nulovými hodnotami. To by mohlo ovlivnit čas, který je potřebný pro výpočet algoritmu. Tento důsledek můžeme vidět např. v tabulkách 4.1 nebo 4.7, kdy je výpočetní čas rozdílný pro nulové a nenulové vstupní hodnoty.

Čas potřebný pro výpočet vektorového řízení byl měřen pro kód uvedený výše. Opět byly použity některé optimalizace, které je možné v MCUXpressu nastavit. Bez optimalizací byl průměrný čas výpočtu 5.07 µs. Při optimalizacích O1 a O3 byl průměrný čas přibližně 4.1 µs. Do měření je zahrnut i vypočet akčního zásahu regulátoru otáček. Ten bývá obvykle začleněn do pomalejší smyčky. Funkce *PCLIB_CtrlPI_F16()*, která realizuje PI regulátor, trvá přibližně 0.6 µs při vypnutých optimalizacích. Je tedy jasné, že separováním regulátoru otáček bychom dospěli k rychlejšímu výpočtu proudové smyčky. Pokud bychom zvolili vzorkovací frekvenci proudové smyčky na 16 kHz, máme dostatečnou velkou časovou rezervu pro další algoritmy v této rutině.

Jelikož jsou knihovní funkce odladěné a optimalizované, bude jejich vykonání trvat poměrně krátkou dobu. Výsledný kód s optimalizacemi nám pak může ušetřit spoustu času. Další výhodou je zpracování funkcí z pohledu uživatele. Funkce, které spolu souvisí, na sebe navazují a není tak nutná další manipulace nebo přepočet výsledků. To souvisí i s tím, že funkce vesměs přijímají a vracejí stejný datový typ, pokud se nejedná o strukturu. Díky tomu můžeme psát přehledný strukturovaný kód.

Jako nevýhodu bychom mohli z počátku vidět frakční čísla a makra pro převod. Při zapisování do struktur si musíme dávat pozor, jakého datového typu je konkrétní parametr. Při použití nesprávného makra bychom mohli dostávat jiné výsledky, než očekáváme. Otázkou však je, jestli jsou frakční čísla používané v RTCESL nevýhodou, když nám v podstatě zajišťují desetinná čísla reprezentované v celočíselné aritmetice.

6.3 Kód pro vektorové řízení generovaný Simulinkem

Při generování zdrojového kódu budeme opět vycházet z modelu na obrázku 6.4. Model připravený pro generování kódu je v příloze 3. Bloky realizující transformace jsou modelovány podle rovnic (5.1), (5.3) a (5.4). Algoritmus bloku *SVM* byl vytvořen na základě dokumentace k funkci GMCLIB_SvmStd z knihovny RTCESL. PI regulátory jsou realizovány podle modelu na obrázku 6.3. Malou změnou je výměna spojitého integrátoru za diskrétní.

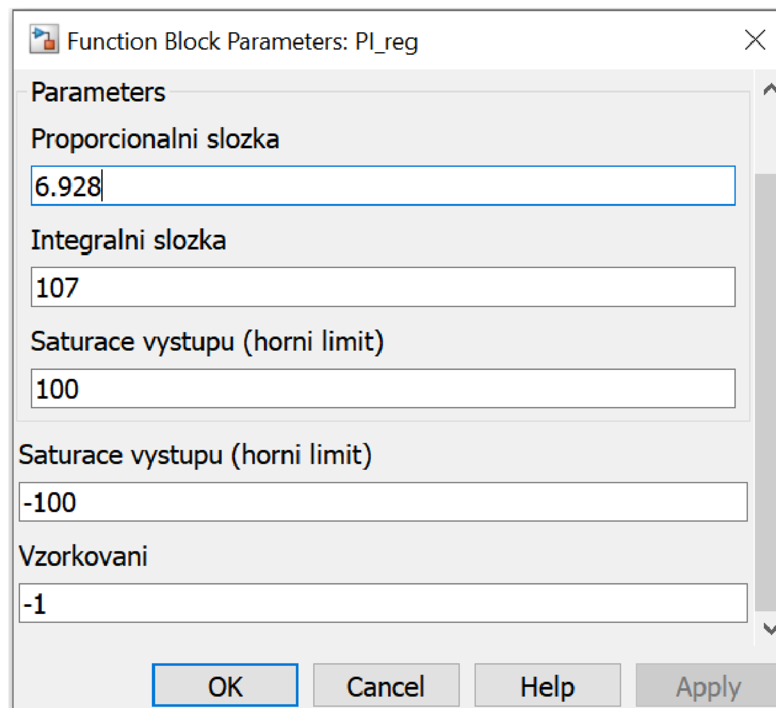
Generovaný kód bude v plovoucí aritmetice. Vzhledem k tomu, že máme k dispozici mikrokontroler se single precision FPU, budeme používat nanejvýš čísla s pohyblivou řádovou tečkou s jednoduchou přesností. V konfiguraci parametrů modelu otevřeme *Optimization*. Parametr *Default for underspecified data type* nastavíme na *single*. Datový typ výstupního signálu u bloků sumace bude *signle*. Vstupní a výstupní porty

transformací a SVM musí být taktéž datového typu *single* Výjimkou je port *sector* u SVM, který je typu *uint16_t*.

Parkovy transformace obsahují goniometrické funkce *sin* a *cos*. Aby pro ně byla v generovaném kódu vložena odpovídající funkce pro čísla s jednoduchou přesností, nastavíme matematickou knihovnu. V parametrech modelu v záložce *Code Generation – Interface* nastavíme *Standard math library* a *C99 (ISO)*.

Všechny PI regulátory mají stejnou strukturu a budou volány vícekrát, jen s rozdílnými parametry. Proto je nastavíme jako *Reusable function* v parametrech bloku. Dále je pro blok regulátoru vytvořena tzv. maska. To zajistí nastavení parametrů skrz dialogové okno. Při práci v Simulinku pak nemusíme vstupovat do subsystému a měnit parametry uvnitř subsystému.

Tak jako regulátory, tak i další subsystémy v modelu jsou nastaveny na *Reusable function*.



Obrázek 6.5 - Nastavení regulátoru přes dialogové okno

Zbývá nám nastavit cílové zařízení, pro které budeme generovat kód. Opět vybereme *Embedded Coder – ert.tlc* v záložce *Code Generation*. Pak už můžeme nechat vygenerovat kód. Soubory s generovaným kódem najdeme v příloze 3. Nyní si popíšeme, co se v těchto souborech nachází.

FOC_Simulink.h – Nejdříve jsou vloženy knihovny, které bude algoritmus používat. Poté je deklarováno několik struktur. Tyto struktury se vztahují ke konkrétním blokům a signálům modelu. U některých struktur není na první pohled jasné, k čem slouží. Nás

však hlavně zajímají ty struktury, pomocí kterých budeme přistupovat ke vstupním a výstupním portům (hodnotám). Na obrázku 6.4 se jedná o žluté a zelené porty. V generovaném kódu jsou struktury pro tyto účely hezky nachystány. Tento hlavičkový soubor zakončuje deklarace inicializační funkce a funkce, která představuje jednu iteraci kódu.

FOC_Simulink_private.h – Obsahuje deklarace funkcí, které reprezentují subsystémy v modelu.

rtwtypes.h – Obsahuje vytvořené uživatelské datové typy a mezní hodnoty pro 8, 16 a 32bitová celá čísla.

FOC_Simulink.c – Najdeme zde zdrojové kódy všech funkcí, které jsou deklarovány v *FOC_Simulink.h* a *FOC_Simulink_private.h*. Nyní se zaměříme na funkci *FOC_Simulink_step()*. Jak již bylo zmíněno, představuje jednu iteraci modelu. Vygenerovaný kód je strukturovaný a poměrně čitelný. Bylo by také možno vhodně okomentovat jednotlivé funkce v kódu. Vygenerované hlavičky funkcí pro subsystémy dělají kód lehce nečitelný. Například pro výpočet regulátoru by bylo vhodnější předávat do funkce ukazatel na strukturu obsahující parametry regulátoru místo jednotlivých parametrů regulátoru. Také u transformací by bylo vhodnější předávat data pomocí struktur. Pokud bychom chtěli toto předávání do funkcí změnit, museli bychom upravit subsystém tak, aby jeho vstupní (výstupní) port neměl rozměr skaláru, ale vektoru. V subsystému bychom si tento signál rozdělili pomocí bloku *Bus Selector* a dílčí signály rozvedly podle potřeby. Do subsystému by opět musel vstupovat vícerozměrný signál, který bychom zajistili spojením jednotlivých signálů pomocí *Bus Creator*.

Doba trvání výpočtu byla měřena pro funkci *FOC_Simulink_step()* při zapnuté FPU jednotce. Bez optimalizací byl průměrný čas výpočtu 26 µs. Při optimalizacích O1 21.8 µs. Při nejvyšším stupni optimalizací O3 trval výpočet průměrně 19.9 µs. Měřená smyčka opět obsahuje výpočet regulátoru rychlosti, jehož doba výpočtu je přibližně 1.38 µs bez optimalizací a s optimalizací O3 je čas zhruba poloviční. Podíváme-li se na generované funkce Parkovy transformace, vidíme v každé funkci výpočet *sinf* a *cosf*. Oba subsystémy jsou tak namodelovány. Při měření v kapitole 4 jsme si ověřili, že goniometrické funkce jsou časově poměrně náročné. Pro zrychlení celého algoritmu bychom na začátku iterace vypočítali hodnoty *sin(ϑ)*, *cos(ϑ)* a ty pak posílali do subsystémů pro Parkovu transformaci jako tomu je u knihovních funkcí RTCESL.

7. ZÁVĚR

Cílem práce bylo seznámit se s možnostmi, které MATLAB/Simulink nabízí pro generování kódu a výsledky generovaného kódu srovnat s ručně psaným kódem v jazyce C. Větší pozornost byla věnována Simulinku a generování kódu z něj. Důvod je ten, že Simulink je stavěn na modelování a simulaci dynamických systémů, do kterých spadá i oblast řízení motorů. Abychom mohli v Simulinku testovat algoritmy napsané v jazyce C, představili jsme si možnosti, jak takové algoritmy začlenit do Simulinku. Nástroj Legacy code tool byl autorem této práce shledán jako nejpřívětivější.

Velká část této práce byla věnována porovnání kódu generovaného Simulinkem a ručně psaného kódu. Pokud je to možné, Simulink nahrazuje bloky v generovaném kódu standardními funkcemi pro jazyk C. Generované názvy proměnných jsou občas matoucí. Otázkou je, jak by si Simulink při generování kódu poradil s komplexnějšími modely. Jedna věc je více než jistá. Vytvoření modelu a vygenerování kódu bude pravděpodobně rychlejší než ruční psaní. Troufám si tvrdit, že výsledný kód bude však méně efektivní než ručně psaný. Koneckonců, o tom jsme se přesvědčili v této práci při srovnávání algoritmů ručně psaných a generovaných Simulinkem. Ručně psané a optimalizované algoritmy byly v našem případě vesměs rychlejší než kód, který vygeneroval Simulink.

V práci byl kladen důraz na používání vhodných datových typů. To je jeden z předpokladů správného fungování algoritmu. Obzvláště při vytváření modelu v Simulinku za účelem generování kódu musíme pečlivě zvážit, jaké datové typy budeme používat.

Obvykle je pro již známou a prozkoumanout problematiku k dispozici optimalizované řešení, a tak by bylo zbytečné používat generovaný kód, který by byl méně efektivní. Generování kódu ze Simulinku bych viděl přínosné v případě, že chceme vytvořit nový algoritmus. Simulink by sloužil jako nástroj, ve kterém částečně otestujeme funkčnost. Generovaným kódem bychom se mohli inspirovat a posléze ručně napsat efektivnější kód.

Literatura

[1] GMCLIB User's Guide [online]. NXP Semiconductors, 2016 [cit. 2019-04-27]. Dostupné z: https://www.nxp.com/docs/en/user-guide/CM4GMCLIBUG.pdf

[2] 754-2008 - IEEE Standard for Floating-Point Arithmetic [online]. IEEE Xplore Digital Library, 2008 [cit. 2019-04-29]. Dostupné z: https://ieeexplore.ieee.org/document/4610935

[3] KV4x: Documentation [online]. NXP Semiconductors, 2016 [cit. 2019-04-29]. Dostupné z: https://www.nxp.com/products/processors-and-microcontrollers/arm-based-processors-and-mcus/kinetis-cortex-m-mcus/v-seriesreal-time-ctlm0-plus-m4-m7/kinetis-kv4x-168-mhz-high-performance-motor-power-conversion-mcus-based-on-arm-cortex-m4:KV4x?tab=Documentation_Tab

[4] The Joy of Generating C Code from MATLAB [online]. 2016 [cit. 2018-12-15]. Dostupné z: https://www.mathworks.com/company/newsletters/articles/the-joy-of-generating-c-code-from-matlab.html

[5] MATLAB Programming for Code Generation [online]. The MathWokrs, 2015 [cit. 2018-12-29]. Dostupné z: https://www.mathworks.com/help/releases/R2015b/coder/matlab-algorithm-design.html

[6] RTCESL [online]. NXP Semiconductors, 2016 [cit. 2018-12-18]. Dostupné z: https://www.nxp.com/products/processors-and-microcontrollers/additional-processors-and-mcus/digital-signal-controllers/real-time-control-embedded-software-motor-control-and-power-conversion-libraries:RTCESL

[7] MCUXpresso Software and Tools [online]. 2019 [cit. 2019-04-29]. Dostupné z: https://www.nxp.com/support/developer-resources/software-development-tools/mcuxpresso-software-and-tools:MCUXPRESSO

[8] Fixed point arithmetic [online]. Pavel Tišnovský, 2006 [cit. 2019-05-10]. Dostupné z: https://www.root.cz/clanky/fixed-point-arithmetic/?ic=serial-box&icc=text-title

[9] What is an S-Function [online]. The MathWorks, 2018 [cit. 2018-12-18]. Dostupné z: https://www.mathworks.com/help/simulink/sfg/what-is-an-s-function.html

[10] KOZUMPLÍK, K. Modelování elektrických pohonů na platformě CompactRIO s využitím FPGA. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2016. 92s. Vedoucí diplomové práce byl Ing. Libor Veselý, Ph.D.

[11] RD56F84789: PMSM FOC of Industrial Drives Reference Design [online]. NXP Semiconductors, 2019 [cit. 2019-05-14]. Dostupné z: https://www.nxp.com/products/processors-and-microcontrollers/additional-processors-and-mcus/digital-signal-controllers/pmsm-foc-of-industrial-drives-reference-design:RD56F84789

[12] 3-phase Sensorless Single-Shunt CurrentSensing PMSM Motor Control Kit with MagniV MC9S12ZVM [online]. NXP Semiconductors, 2016 [cit. 2019-05-14]. Dostupné z: https://www.nxp.com/docs/en/application-note/AN5327.pdf

Seznam příloh

Příloha 1 - Generovaný kód z bloku Sine 52
Příloha 2 - Zdrojový a hlavičkový soubor pro LCT, C MEX S-funkce 52
Příloha 3 - Kód FOC generovaný Simulinkem 52
Příloha 4 - Kód pro vektorové řízení (RTCESL) 53

Příloha 1 - Generovaný kód z bloku Sine

Soubory se nachází ve složce ***Priloha_1*** – pro tištěnou verzi na přiloženém CD, pro elektronickou verzi v přiložením komprimovaném souboru. Zdrojový kód funkcí je v souboru ***my_mdl.c***. Nastavení parametru *Internal rule priority for lookup table*: *Speed* odpovídá funkce *extern void my_mdl_sin_1()*. Nastavení parametru *Internal rule priority for lookup table*: *Precision* odpovídá funkce *void my_mdl_sin_2()*.

Příloha 2 - Zdrojový a hlavičkový soubor pro LCT, C MEX S-funkce

Soubory se nachází ve složce ***Priloha_2*** – pro tištěnou verzi na přiloženém CD, pro elektronickou verzi v přiloženém komprimovaném souboru.

Příloha 3 - Kód FOC generovaný Simulinkem

Soubory se nachází ve složce ***Priloha_3*** – pro tištěnou verzi na přiloženém CD, pro elektronickou verzi v přiloženém komprimovaném souboru.

Příloha 4 - Kód pro vektorové řízení (RTCESL)

```
#include <stdio.h>
#include "mlib.h"
#include "gflib.h"
#include "gmclib.h"
#include "pclib.h"

// Pomocne I/O
static frac16_t A,B,C, A_meas, B_meas, C_meas, w_meas, theta;
static uint16_t Sector;

// Clark
static GMCLIB_2COOR_ALBE_T_F16 sAlphaBetaClark;
static GMCLIB_3COOR_T_F16 sAbcClark;

// Park
static GMCLIB_2COOR_DQ_T_F16 sDQPark;
static GMCLIB_2COOR_SINCOS_T_F16 sAnglePark;

// Park inverse
static GMCLIB_2COOR_ALBE_T_F16 sAlphaBetaParkInv;
static GMCLIB_2COOR_DQ_T_F16 sDQParkInv;
static GMCLIB_2COOR_SINCOS_T_F16 sAngleParkInv;

// PI regulatory
static frac16_t f16ResultPId, f16InErrPId, f16ResultPIq, f16InErrPIq, f16ResultPIw, f16InErrPIw;
static PCLIB_CTRL_PI_T_F16 sParamPId, sParamPIq, sParamPIw;

// SVM
static uint16_t u16SectorSVM;
static GMCLIB_3COOR_T_F16 sAbcSVM;

int main(void)
{
	// Clark
	sAbcClark.f16A = FRAC16(0.0);
	sAbcClark.f16B = FRAC16(0.6);
	sAbcClark.f16C = FRAC16(-0.3);

	// Park
	sAnglePark.f16Sin = FRAC16(0.0);
	sAnglePark.f16Cos = FRAC16(1.0);

	// Park inverse
	sDQParkInv.f16D = FRAC16(0.0);
	sDQParkInv.f16Q = FRAC16(0.0);
	sAngleParkInv.f16Sin = FRAC16(0.0);
	sAngleParkInv.f16Cos = FRAC16(1.0);
```

```
	// PI-Id
	f16InErrPId = FRAC16(-0.4);
	sParamPId.f16Kp = FRAC16(0.1);
	sParamPId.f16Ki = FRAC16(0.2);
	sParamPId.f16IntegralUpperLimit = FRAC16(0.9);
	sParamPId.f16IntegralLowerLimit = FRAC16(-0.9);
	sParamPId.f16UpperLimit = FRAC16(0.9);
	sParamPId.f16LowerLimit = FRAC16(-0.9);
	PCLIB_CtrlPIInit_F16(&sParamPId);

	// PI-Iq
	f16InErrPIq = FRAC16(-0.4);
	sParamPIq.f16Kp = FRAC16(0.1);
	sParamPIq.f16Ki = FRAC16(0.2);
	sParamPIq.f16IntegralUpperLimit = FRAC16(0.9);
	sParamPIq.f16IntegralLowerLimit = FRAC16(-0.9);
	sParamPIq.f16UpperLimit = FRAC16(0.9);
	sParamPIq.f16LowerLimit = FRAC16(-0.9);
	PCLIB_CtrlPIInit_F16(&sParamPIq);

	// PI-w
	f16InErrPIw = FRAC16(-0.4);
	sParamPIw.f16Kp = FRAC16(0.1);
	sParamPIw.f16Ki = FRAC16(0.2);
	sParamPIw.f16IntegralUpperLimit = FRAC16(0.9);
	sParamPIw.f16IntegralLowerLimit = FRAC16(-0.9);
	sParamPIw.f16UpperLimit = FRAC16(0.9);
	sParamPIw.f16LowerLimit = FRAC16(-0.9);
	PCLIB_CtrlPIInit_F16(&sParamPIw);

	A_meas = FRAC16(0.0);
	B_meas = FRAC16(0.6);
	C_meas = FRAC16(-0.3);
	theta = FRAC16(0.0);

while(1)
{
	sAbcClark.f16A = A_meas;
	sAbcClark.f16B = B_meas;
	sAbcClark.f16B = C_meas;

	sAnglePark.f16Sin = GFLIB_Sin_F16(theta);
	sAnglePark.f16Cos = GFLIB_Cos_F16(theta);

	// transformace
	GMCLIB_Clark_F16(&sAbcClark, &sAlphaBetaClark);
	GMCLIB_Park_F16(&sAlphaBetaClark, &sAnglePark, &sDQPark);
```

```
        // akcni zasah otacky
        f16InErrPIw = MLIB_Sub_F16(FRAC16(0.4),w_meas);              // suma
        f16ResultPIw = PCLIB_CtrlPI_F16(f16InErrPIw, &sParamPIw);    // regulator otacek

        // akcni zasah slozka Iq
        f16InErrPIq = MLIB_Sub_F16(f16ResultPIw,sDQPark.f16Q);             // suma
        f16ResultPIq = PCLIB_CtrlPI_F16(f16InErrPIq, &sParamPIq);          // regulator Iq

        // akcni zasah slozka Id
        f16InErrPId = MLIB_Sub_F16(FRAC16(0),sDQPark.f16D);          // suma
        f16ResultPId = PCLIB_CtrlPI_F16(f16InErrPId, &sParamPId);    // regulator Iq

        // Park inverse
        sDQParkInv.f16D = f16ResultPId;
        sDQParkInv.f16Q = f16ResultPIq;
        GMCLIB_ParkInv_F16(&sDQParkInv, &sAnglePark, &sAlphaBetaParkInv);

        // SVM
        u16SectorSVM = GMCLIB_SvmStd_F16(&sAlphaBetaParkInv, &sAbcSVM);

        Sector = u16SectorSVM;
        A = sAbcSVM.f16A;
        B = sAbcSVM.f16B;
        C = sAbcSVM.f16C;
    }
    return 0 ;
}
```