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

VYUŽITÍ METODY „MODEL BASED DESIGN“ PRO NÁVRH EMBEDDED APLIKACE

USE OF THE "MODEL BASED DESIGN" METHOD FOR THE DESIGN OF AN EMBEDDED APPLICATION

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE **Bc. Martin Mačišák**

AUTHOR

VEDOUCÍ PRÁCE **doc. Ing. Zdeněk Bradáč, Ph.D.**

SUPERVISOR

BRNO 2021

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Kybernetika, automatizace a měření**

Ústav automatizace a měřicí techniky

Student: Bc. Martin Mačišák ***ID:*** 186134
Ročník: 2 ***Akademický rok:*** 2020/21

NÁZEV TÉMATU:

Využití metody „model based design“ pro návrh embedded aplikace

POKYNY PRO VYPRACOVÁNÍ:

Navrhněte vhodný postup pro použití dostupných nástrojů pro MBD. Ověřte prakticky vybraný nástroj pro návrh řadiče bezkomutátorového elektromotoru (BLDC).

1. Proveďte internetový a literární průzkum.
2. Porovnejte konvenční vývoj SW a MBD přístup. Porovnejte nástroje a metody pro MBD v oblasti automatického generování a verifikace kódu. Definujte využití MBD pro bezpečnostně-kritické aplikace.
3. Vyhodnoťte teoretické výsledky a vyberte vhodné vývojové metody a nástroje pro realizaci řadiče BLDC motoru. Vyberte vhodnou HW platformu.
4. Navrhněte a proveďte modelování algoritmu pro řízení BLDC motoru. Proveďte simulace a ověření navrženého algoritmu. Navrhněte a implementujte rozhraní mezi ručně psaným a generovaným kódem.
5. Implementujte přenos kódu na HW platformu: Vygenerování kódu z modelu, doplnění ručně psaného kódu a propojení. Ověřte funkce řízení reálného BLDC motoru a zhodnoťte dosažené výsledky.

DOPORUČENÁ LITERATURA:

Gabriela Nicolescu & Pieter J. Mosterman : Model-Based Design for Embedded Systems. CRC Press 2010 ISBN 978-1420067859

Dle pokynů vedoucího práce.

Termín zadání: 8.2.2021 ***Termín odevzdání:*** 17.5.2021

Vedoucí práce: doc. Ing. Zdeněk Bradáč, Ph.D.

doc. Ing. Petr Fiedler, Ph.D.
předseda rady studijního programu

UPOZORNĚNÍ:
Autor diplomové práce nesmí při vytváření diplomové 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

Práca vysvetľuje rozdiely medzi prístupmi vo vývoji softwaru. Je vykonaná rešerš v oblasti generovania kódu. Práca opisuje možné nástroje a spôsoby validácie a verifikácii kódu. V ďalšej časti sú podané informácie o používanie model based designu v rámci kriticko-náročných aplikácií. Potom sa práca venuje návrhu platforiem pre riadenie BLDC motoru. Sú vybrané platformy a k ním napísaný príslušný low level software. Je vytvorené užívateľské rozhranie a logika riadenia. Logika riadenia je vytvorená a otestovaná v prostredí Simulink. Následne je spojený vygenerovaný kód s low level vrstvou. Celé riešenie je otestované.

Kľúčové slova

Software, požiadavky, generátor, validácia, verifikácia, kriticko-náročný, VUT, FEKT, BLDC, PWM, task, otáčky, riadenie, Simulink, prerušeniem model, model based design

Abstract

This diploma thesis demonstrates the differences between approaches in software development. A code generation search is performed. The work describes possible tools and methods of code validation and verification. The next part provides information on the use of a design-based model in critical-demanding applications. Further, the work describes the proposal of platforms for controling the BLDC motor. Low leve software is programed for these chosen platforms. The logic of controling and user interface is created. The logic of controling is created and tested in Simulink. Further more whole programmed code is connected with the low level layer. The whole solution of my diploma thesis is tested.

Keywords

Software, requirements, generator, validation, verification, safety critical, BUT, FEKT, BLDC, PWM, task, rotation speed, controling, Simulink, interupt, model, model based design

Bibliografická citace

MAČIŠÁK, Martin. *Využití metody „model based design“ pro návrh embedded aplikace*. Brno, 2021. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/134825. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedoucí práce Zdeněk Bradáč.

Prohlášení autora o původnosti díla

Jméno a příjmení studenta: *Martin Mačišák*

VUT ID studenta: *186134*

Typ práce: *Diplomová práce*

Akademický rok: *2020/21*

Téma závěrečné práce: *Využití metody „model based design“ pro návrh embedded aplikace.*

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně pod vedením vedoucí/ho diplomové 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 diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové 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: 14. května 2021 ---------------------------------------

podpis autora

Poděkování

Děkuji vedoucímu diplomové práce doc. Ing. Zdeňku Bradáčovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé diplomové práce.

Dále bych rád poděkoval Ing. Istvánu Szabó PhD., Ing. Adamu Kolajovi, Ing. Jozefu Blahutovi a Ing. Davidu Svačinovi, za vedení a rady, které mi poskytli v průběhu vypracování mé diplomové práce.

Děkuji celé své rodině za trpělivost a podporu během celého mého studia.

V Brně dne: 14. května 2021 --

podpis autora

Obsah

ZOZNAM OBRÁZKOV 9

ZOZNAM TABULIEK 10

ÚVOD 11

1. PRÍSTUPY VO VÝVOJI SOFTWARU 12

1.1 TRADIČNÝ VÝVOJ SOFTWARE 12
1.1.1 Požiadavky 13
1.1.2 Dizajn softwaru 14
1.1.3 Implementácia 14
1.1.4 Verifikácia 14
1.1.5 Nasadenie 14
1.1.6 Súhrn 14
1.2 VÝVOJ SOFTWARE V RÁMCI MODEL BASED DESIGN 15
1.2.1 Model In The Loop 16
1.2.2 Software In The Loop 17
1.2.3 Processor In The Loop 17
1.2.4 Hardware In The Loop 17
1.3 VÝHODY A NEVÝHODY MODEL BASED DESIGNU 17

2. VALIDÁCIA A VERIFIKÁCIA GENEROVANÉHO KÓDU 19

2.1 PRINCÍP A PRÍSTUPY NÁSTROJOV 19
2.1.1 Numerická metóda 20
2.1.2 Prevencia pred nežiaducou funkcionalitou 21
2.2 MATHWORKS 24
2.2.1 Simulink Coverage 25
2.2.2 Simulink Design Verifier 25
2.2.3 Simulink Fixed Point. 26
2.2.4 Simulation Data Inspector 26
2.2.5 Trasovateľnosť 26
2.2.6 Interface 27
2.3 DSPACE 27
2.3.1 TargetLink 27
2.3.2 Interface 29
2.4 ANSYS SCADE SUITE 29
2.5 OSTATNÉ 30

3. KRITICKO-NÁROČNÉ SYSTÉMY A MODEL BASED DESIGN 31

3.1 NORMY 31
3.1.1 DO-178C 32
3.1.2 DO-331 32
3.2 PRÍKLADY A SÚHRN 33

4. NÁVRH RIEŠENIA 35

4.1 ARCHITEKTÚRA RIEŠENIA 35
4.2 BOX 36

5. UŽÍVATEĽSKÉ ROZHRANIE **37**

5.1 POPIS SOFTWARU 38

5.1.1 Displej task *39*

5.1.2 Zhrnutie *40*

6. RIADENIE A BLDC **41**

6.1 BLDC A JEHO RIADENIE 41

6.2 ZHODNOTENIE BLDC MOTORU 42

6.3 BEZ SENZOROVÉ RIADENIE 42

6.4 PWM 43

6.5 TMS RIADIACI PROGRAM 45

6.5.1 Pripojenie k výkonovému modulu *47*

6.5.2 Štartovanie motoru *48*

6.5.3 Beh motoru - running *48*

6.5.4 Meranie otáčok *48*

6.6 KOMUNIKÁCIA - SCI 49

7. SIMULINK **50**

7.1 MODEL MOTORU 50

7.2 REGULÁTOR 51

7.3 STATE FLOW 53

7.4 VYGENEROVANIE KÓDU 54

7.5 OVERENIE KÓDU A ĎALŠIE APLIKÁCIE 57

7.6 ZHRNUTIE 61

8. SPOJENIE GENEROVANÉHO A RUČNE PÍSANÉHO KÓDU **62**

8.1 EMBEDDED CODER INTERFACE 62

8.2 POUŽITÉ RIEŠENIE PRE VSTUPY A VÝSTUPY 62

9. ZÁVER **64**

LITERATÚRA **66**

ZOZNAM SYMBOLOV A SKRATIEK **69**

ZOZNAM ELEKTRONICKÝCH PRÍLOH **71**

ZOZNAM OBRÁZKOV

Obrázok 1.1 Tradičný kaskádový vývoj SW [27] 12
Obrázok 1.2 WaterFall model používaný v model based design [27] 16
Obrázok 1.3 Možná úprava WaterFall modelu na Y model [27] 18
Obrázok 2.1 Vzťahy testovania medzi modelom a generovaným kódom [10] 20
Obrázok 2.2 Príklad Decision Coverage [12] 22
Obrázok 2.3 Príklad Branch Coverage [27] 23
Obrázok 2.4 Vzťahy medzi požiadavkami modelom a vygenerovaným kódom [27] 23
Obrázok 2.5 Možné použitie nástrojov pri validácii a verifikácii v Simulinku [27] 24
Obrázok 2.6 Typický príklad tvorby a overovania testov [13] 25
Obrázok 2.7 Simulačné módy v TargetLinku [14] 28
Obrázok 4.1Navrhnutá architektúra systému [27] 36
Obrázok 4.2 Spodná časť boxu [27] 36
Obrázok 4.3 Vrchná časť boxu [27] 36
Obrázok 5.1 STM32F746GDISCOVERY kit [20] 37
Obrázok 5.2 Popis SW pre displej interface [27] 38
Obrázok 5.3 Uživateľské rozhranie [27] 40
Obrázok 6.1 TMS320 s dokovacou stanicou [21] 41
Obrázok 6.2 Diagram fáz a prechodov nulami [23] 43
Obrázok 6.3 Možné spínanie MOSFETov pri unipolárnom riadení [25] 44
Obrázok 6.4 Unipolárne komplementárne riadenie MOSFETov [26] 45
Obrázok 6.5 Popis SW riadiacej jednotky [27] 46
Obrázok 6.6 PWM a časovanie [27] 47
Obrázok 6.7 Zapojenie k výkonovému modulu [27] 47
Obrázok 7.1 Model motoru [27] 50
Obrázok 7.2 Prechodová charakteristika regulátoru [27] 52
Obrázok 7.3 Rozhranie Embedded Coder, 1- nastavenie generovania, 2 – interface, 3 – generovanie/build, 4 – nastavenie reportu [27] 54
Obrázok 7.4 Stavový automat, označený blok a jeho kód sa vyznačí vo vygenerovanom kóde [27] 55
Obrázok 7.5 Výsledný stavový automat riadenia [27] 56
Obrázok 7.6 Zobrazenie doplnkov a aplikácii v Simulinku [27] 57
Obrázok 7.7 Requirements manager [27] 57
Obrázok 7.8 Príklad jedného test casu vygenerovaného v Design Verifier [27] 58
Obrázok 7.9 Zapojenie pre testovanie [27] 59
Obrázok 7.10 Sledované vstupy a výstupy pre testovanie [27] 60
Obrázok 7.11 Výsledný produkt [27] 61
Obrázok 8.1 Mapovanie generovaného kódu [27] 62
Obrázok 8.2 Ukážka vo vygenerovanom kóde [27] 63
Obrázok 8.3 Model pre generovanie, na ľavo vstupy, na pravo výstupy [27] 63

ZOZNAM TABULIEK

Tabuľka 2.1 TargetLink Ecosystem 29
Tabuľka 6.1 Rámec posielaný riadiacou jednotkou 49
Tabuľka 6.2 Rámec posielaný užívateľským rozhraním 49

ÚVOD

Mikroprocesory ako také, stretneme v našom živote snáď na každom kroku. Zlepšujú kvalitu nášho života a veľa krát ho zjednodušujú. Vďaka narastajúcej všestrannosti mikroprocesorov, sa naša fantázia a tvorivosť v rôznych dizajnoch posúva na nové hranice. To zapríčinilo dopyt po neustálom zvyšovaní nárokov a zložitosti požadovaných riešení pre všemožné aplikácie. Tieto riešenia následne vyžadujú čoraz viac elektronických súčiastok, či už digitálnych alebo analógových, taktiež spoluprácu viacerých procesorov. Navrhnúť a vytvoriť takéto riešenia je nesmierne náročne či vcelku neefektívne. Preto prichádza na scénu model based design. Objavuje sa ako riešenie týchto náročných aplikácii s vysokou efektivitou, ale poslúži aj na dosiahnutie výpočtových možnosti, ktorých sme zatiaľ nedosiahli. Samotné používanie model based Design pomáha implementovať vysoké nároky v dizajnu, monitorovať cely proces návrhu aplikáciu pre lepšiu optimalizáciu. To nám pomáha ľahšie implementovať a odvodiť riešenie priamo zo špecifikácie. Vizuálne modely a simulácie, následne pomáhajú validovať a overovať software priamo v simulačnom prostredí daného model based design frameworku. V tejto diplomovej práci sa budem venovať vcelku novému pohľadu a spôsobu vývoja softwaru, a to pomocou model based design systémov, hlavne v oblasti následného generovania a implementovania kódu.

Hlavným cieľom diplomovej práce je použite model based design framweroku na vygenerovanie kódu, následne prepojenie takto vygenerovaného kódu s ručne písaným a následná demonštrácia jednoduchého algoritmu na BLDC motoru. Súčasťou je taktiež návrh a riešenie hardwarovej časti ovládania motoru. Pričom sa práca zameriava na generovaný kód, možné nastavanie generátoru prípadné porovnanie rôznych generátorov kódu. Návrh modelu, generovanie kódu, prípadná validácia a verifikácia je vykonaná na platformách od firmy MathWorks. Výstup je následné implementovaný na vývojovej platforme.

V úvode práce si predstavíme prístupy vo vývoji softwaru. Budú porovnané prístupy k riešeniu klasický konvenčný prístup oproti prístupu v rámci model based design. Následné je vykonaná rešerš v oblasti validácie a verifikácie generovaného kódu, v rámci rôznych nástrojov od rôznych firiem. V ďalšej časti je popísaná oblasť v rámci bezpečno-kritických aplikácii. Jednotlivé výsledky a poznatky sú zhodnotené.

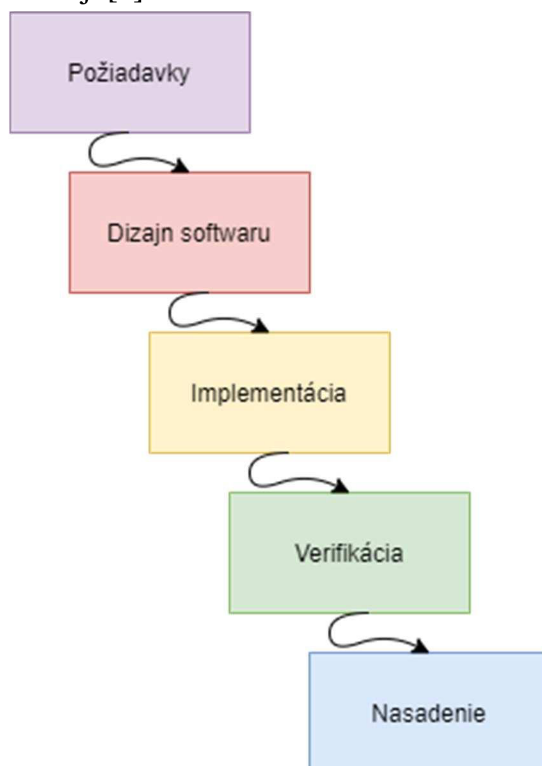
Nasleduje časť, v ktorej sa vyberú platformy k realizácii projektu. Jednotlivé platformy sú zhodnotené, je popísaný BLDC motor a princíp riadenia. Vybrané platformy sú potom naprogramované. K týmto naprogramovaným platformám sa implementuje logika riadenia, ktorá je vytvorená v Simulinku. Pričom logika bude overená a otestovaná. Následne sa vygeneruje kód tejto logiky, ktorý bude vhodným spôsobom prepojený s naprogramovanými platformami. Celé riešenie bude otestované a popísané. Záver práce sa venuje hodnoteniu celkového riešenia tejto diplomovej práce.

1. PRÍSTUPY VO VÝVOJI SOFTWARU

Vývoj softwaru je proces, ktorý zahŕňa časti ako špecifikovanie, návrh dizajnu, dokumentáciu, programovanie, testovanie, opravovanie chýb, ktorých výsledkom je vytvorenie a údržba aplikácii alebo iných softwarových komponentov. U vývoju v rámci embedded systémov, ako riadiace jednotky, je vyžadované aby bol software priamo implementovaný do fyzického produktu a tvoril základ samotného produktu. V dnešnej dobe už existuje mnoho prístupov v projektovom vývoji softwaru, ako životný cyklus vývoja, procesy či modely. V nasledujúcich kapitolách budú predstavené prístupy, a to tradičný prístup, takzvaný model vodopádu a u MBD vývoju takzvaný V model. [1]

1.1 Tradičný vývoj software

Tradičná metodika spočíva v postupnom prechádzaní medzi fázami životného cyklu vývoja softwaru. Preto je celkový proces jednosmerný, každá fáza ma svoje výstupy a dokumentáciu. Jednotlivé fázy prechádzajú dôkladnými kontrolami, od požiadaviek cez návrh, vývoj, testovanie a údržbu. Model sa pomenoval ako vodopádový anglicky WaterFall pretože ako vidíme na obrázku č. 1.1 jednotlivé fázy pretekajú z jednej fázy do druhej. [2]



Obrázok 1.1 Tradičný kaskádový vývoj SW [27]

Prvá takáto reprezentácia modelu vznikla v roku 1956 vytvorená Herbetom D. Bennigtonom. Vymyslel ju pre vývoj software v rámci SAGE, s tým že fázy sa

nedodržiavali striktne smerom z hora nadol ale postupnosť závisela od produktu k produktu. Prvú formálnu zmienku o WaterFall modelu nájdeme v článku od Winstona W. Royce, avšak model nebol predstavený ako vodopádový, ale bol spomenutý ako príklad chybného a nepracujúceho modelu. Snažil sa tým opísať svoj kritický pohlaď na bežne používanie modelu v rámci praxe. Nakoniec roku 1985 bol vydaný dokument DOD-STD-2167A americkým ministerstvom, kde bol implementovaný cyklus ktorý sa skladal z fáz:

1. Softwarové požiadavky
2. Predbežný návrh
3. Podrobný dizajn
4. Unit testy a programovanie
5. Integrácia
6. Testovanie

Postupom času sa model upravil do terajšej podoby. Winston W. Royce následne popísal zásady pre finálny model ako:

1. Vytvoriť dizajn programu pred analýzou a programovaním
2. Dokumentácia musí byt kompletná v rámci každej fázy
3. Ak sa dá, vykonať každú fázu dva-krát
4. Testovanie musí byť plánované, kontrolované a monitorované
5. Čo najviac konzultovať so zákazníkom [2]

1.1.1 Požiadavky

Špecialista, niekedy tím špecialistoch, vytvára požiadavky, ktoré tvoria základ celého softwarového projektu. Okrem toho, že sa zhromaždia komplexné informácie o tom, čo daný projekt vyžaduje, pričom sa vytvárajú požadované ciele a funkcionality projektu, sa musí počas celého projektu dohliadať na proces vývoja. To znamená, že sa musia stanoviť minimálne požiadavky na bezpečnosť či identifikovať vysoko kritické časti projektu. Pri vývoji softwaru musia dohľadať na to, aby pri prirodzených zmenách niektorých funkcionalít bola zachovaná výsledná kvalita a výsledná požadovaná bezpečnosť softwaru.

Požiadavky by mali obsahovať:

- Účel vytváraného softwaru a jeho podrobný popis
- Čo ma software vykonávať a jeho funkcionalitu
- Použitie softwaru v prevádzke
- Nefunkcionálne požiadavky
- Interface
- Konštrukčné obmedzenia, prostredie kde bude software [3]

1.1.2 Dizajn softwaru

Počas tejto fázy sa navrhne systém, pričom nedochádza k programovaniu ale stanovujú sa technické parametre ako hardware, programovací jazyk a užívateľské rozhranie. Výsledkom sú jednotlivé funkcie, ktoré tvoria súbor modulov a submodulov výsledného softwaru. Často sú súčasťou tejto fázy aj UML diagramy, ktorými dokážeme celý návrh vizualizovať a to celé v požadovanom štandarde. V tejto fáze by už mala byť vytváraná aj dokumentácia celého projektu, ktorá poslúži na overenie navrhnutého dizajnu. [4]

1.1.3 Implementácia

Začína sa programovať. Vytváraný kód je typický rozdelený do jednotlivých modulov, ktoré sú následne integrované do jedného modulu na konci tejto fázy. Programátori pracujú so vstupmi vytvorenými v dizajnu softwaru a v požiadavkách. Každá časť, vytváraná programátormi je v tejto fáze testovaná takzvanými Unit Testami. Testy sa zameriavajú na najmenšie funkčné časti, bez ohľadu na funkčnosť v rámci celku. Testy sa vytvárajú ručne alebo sú generované automatizovane.

1.1.4 Verifikácia

Výstup z implementácie je testovaný tak, aby sa zistili či zodpovedá vytvoreným požiadavkám. Testeri vyhľadávajú rôzne bugy a chyby v softwaru. Celý produkt sa testuje, avšak pri vážnych problémoch je pravdepodobné, že sa celý projekt vráti do fázy č. 1 a to definovanie požiadaviek.

Jednými z testov vykonávané testermi sú takzvané integračné testy. Tie slúžia hlavne na tom ak sa do projektu pridávajú nové komponenty a overuje sa funkčnosť či komunikácia medzi ostatnými komponentami. Najpodstatnejšími testami sú systémové testy, kde sa overuje funkcionalita celého systému. Tieto testy sú vykonávané z pohľadu zadávateľa projektu kde si on sám pripravuje testovacie scenáre. Následné sa vykonajú akceptačné testy testermi od zákazníka. Ak testy prejdú, je software akceptovaný a predaný zákazníkovi. Pri aktualizácii softwaru sa vytvárajú takzvane regresné testy, ktoré testujú nové verzie softwaru. [3], [4]

1.1.5 Nasadenie

Produkt je dodaný klientovi. V prostredí vytvoreným klientom sa projekt otestuje, v prípade problémov určený tím pracovníkov vytvára opravy. Podľa zmluvy sa udržuje údržba u klienta.

1.1.6 Súhrn

Metóda WaterFall je najprívetivejšia pri riešeniach, kde sú požiadavky jasne a nebudú sa meniť. Výhodou je ak je projekt predvídateľný a technológiu je vám známa. Vďaka jednotlivým fázam a dokumentáciám dokážeme veľmi ľahko do riešenia zahrnúť nových vývojárov, ktorí sa v projekte zorientujú rýchlo. Jednotlivé postupy fázami poukazujú na

progres v riešení, tým určíme či projekt postupuje vpred. Koniec koncov takto správne vykonaný postup šetri peniaze a čas, nesmierne dôležitá je však detailná dokumentácia a konceptualizácia. [3]

1.2 Vývoj software v rámci model based design

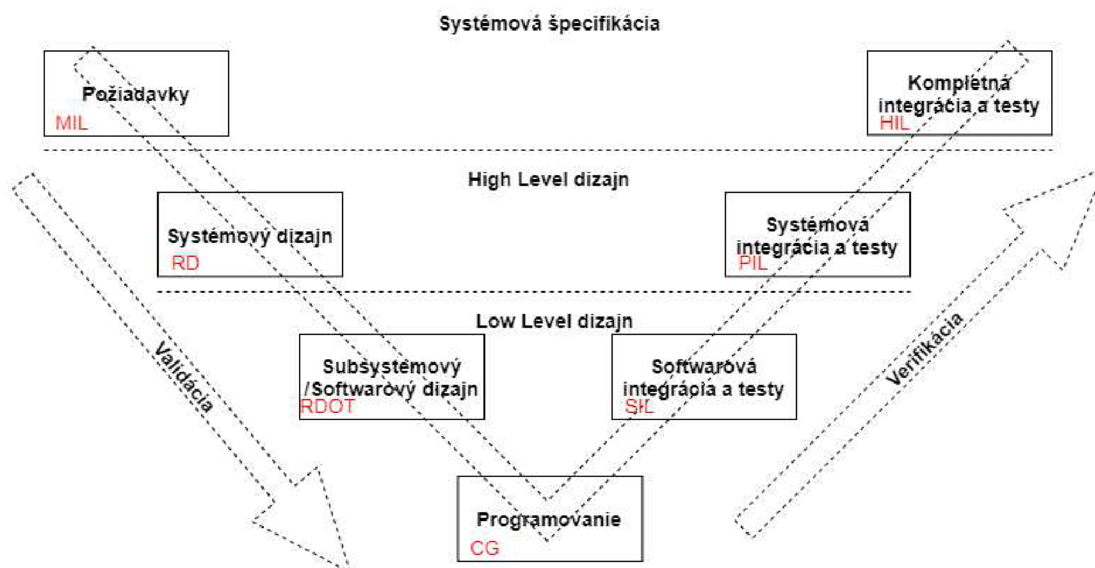
Model based design spája matematické a vizuálne metódy pomocou ktorých dokážeme vyriešiť mnoho problémov spojených s navrhovaním komplexných projektov. Vďaka jeho dizajnu sa vytvára efektívny prístup k vytvoreniu spoločného rámca počas celého procesu návrhu. Tento rámec slúži k spoločnej komunikácii v jednotlivých fázach vývojového cyklu, takzvaného modelu V.

Návrh môžeme všeobecne rozdeliť do týchto štyroch krokoch:

1. Modelovanie
2. Analýza modelov a regulátorov
3. Simulácie
4. Integrácia a nasadenie

Ak sa na problematiku pozeráme z historického hľadiska, kľúčovým bolo vytvoriť prístupy pre model založené na popise, analýze a dizajnu. Systém bol spravidla považovaný ako súbor predmetov s atribútmi a vzťahmi medzi objektmi ale aj jednotlivými atribútmi. Matematický popis systémového modelovania sa neustále hľadal. Nakoniec sa v praxi pristúpilo k menej formálnej a viac intuitívnemu prístupu a to ku softwaru používajúci grafické modelovacie jazyky. Vytvoril sa takzvaný štruktúrovaný dizajn. Jednotlivé systémové objekty sa vložili do komponent, ktoré tvoria súdržný model. B. Yourdon vyriešil problém v rámci štruktúry, páni Wymore a Klir sa podieľali na izomorfnom zachovaní v oblasti funkcii, stavov a udalosti. Matematickú transformáciu medzi funkcionálnymi požiadavkami a parametrami zaobstaral E.N.P. Sih a aj jeho axiomaticky dizajn. [5]

Najviac používaný model vývoja v model based design je takzvaný V model. Niekedy tento model, môžeme nájsť aj pod názvom model validovania a verifikovania. Podobne ako u tradičného Waterfall modelu, každá fáza musí byť dokončená pred začatím tej ďalšej. Pričom testovanie v rámci každej fázy sa pripravuje zároveň s fázou. [5]



Obrázok 1.2 WaterFall model používaný v model based design [27]

Podľa obrázku č. 1.2 je nám hneď jasné, že podobne ako u konvenčného prístupu začíname prvým krokom a to súpisom požiadaviek. Tak isto nasleduje fáza systémového dizajnu v ktorej sa požiadavky analyzujú a navrhuje sa celkový dizajn systému. Do tejto fázy môžeme zahrnúť aj návrh architektúry celého dizajnu.

V ďalšej fáze sa vytvárajú modely a submodely. To rozdelí celý systém do menších modulov a častí. Špecifikuje sa podrobný dizajn pre všetky moduly, vytvoria low-level testy. Je dôležité aby dizajn modelov bol kompatibilný s ostatnými modelmi v architektúre. Vytvorené low-level testy slúžia k eliminácii chýb v čo najrannejšom období vývoja softwaru.

Nasleduje fáza programovania. Programujú sa navrhnuté moduly v predošlej fáze. Kód sa vykonáva pod danými štandardmi a musí prechádzať mnohými kontrolami a optimalizáciami. Vytvorený kód sa posúva do ostatných fáz v rámci verifikácii. No najprv, k pochopeniu celého vývojového cyklu si musíme osvojiť štyri základné pojmy, bez ktorých sa v model based designu nedokážeme orientovať.

- MIL: Model-In-The-Loop
- SIL: Software-In-The-Loop
- PIL: Processor-In-The-Loop
- HIL: Hardware-In-The-Loop [8]

1.2.1 Model In The Loop

Po vytvorení modelov v simulačnom prostredí, je dôležité overiť či nami navrhnutá logika funguje podľa požiadaviek. Na to sa používajú simulácie s navrhnutými testami.

Takýmto spôsobom môžeme otestovať správanie systému ešte pred nasadením na skutočný hardware.

1.2.2 Software In The Loop

Toto testovanie používame vo fáze integrácie a testovania vytvorených subsystémov. Testuje sa vygenerovaný kód, ktorý prakticky nahradí modely. V týchto testoch sa porovnávajú výsledky s výsledkami z testovania modelov.

1.2.3 Processor In The Loop

Ak sme úspešne prešli MIL a SIL, nasleduje fáza integrácii systému. Vygenerovaný kód sa skompiluje a je vykonávaný na hostiteľskom procesore, tj. najčastejšie na vytvorenom vývojárskom moduly alebo kompatibilných kitoch. Porovnaním výsledkov simulácii a testovania na procesoru sa testuje ekvivalencia modelov a vygenerovaného kódu.

1.2.4 Hardware In The Loop

Poslednou fázou je celkový test a integrácia v rámci reálneho hardwaru. Kód je umiestnený do mikrokontroléru, ktorý je priamo spojený so skutočným hardwarom, na ktorom sa celý software vytváral. Sú pripojené jednotlivé vstupy a výstupy ako snímače, napájanie atď. Je to posledná úroveň integrácie, pomocou ktorej je možná oprava jednotlivých funkcii v softwaru.

1.3 Výhody a nevýhody model based designu

Pri vývoji model based design softwaru v rámci V cyklu pri fázach systémového a softwarového dizajnu, MBD predstavuje jednu z výhod ako rapid prototyping. Je to veľmi efektívny a rýchly spôsob ako overiť návrh v ranných štádiách. Jedná z ďalších výhod je generovanie optimalizovaného kódu, ktorý sa vloží priamo do mikrokontroléru alebo ako súčasť nejakej jednotky. V neposlednej rade výhody prináša zjednodušenie verifikácii pomocou „In The Loop“ testovania.

Celkové výhody, ktoré nesúvisia priamo s modelom vývoja sú:

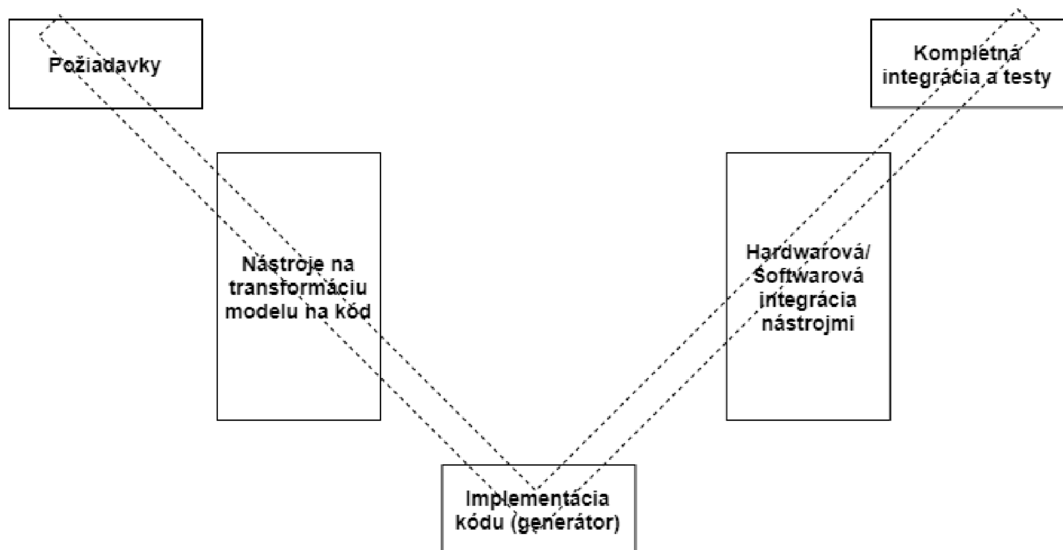
- Konzistentná dokumentácia
- Opätovné použitie kódu vďaka blokom
- Nevyžaduje expertov na programovanie, vďaka generátorom
- Simulácie znižujú potreby debugovania
- Automatické generovanie chýb v modeloch
- Lepšia komunikácia a porozumenie vďaka grafickému rozhraniu

Možné nevýhody:

- Implementácia kódu je veľa krát náročnejšia ako sa na prvý pohľad zdá
- Niektoré špeciálne funkcie sa kvôli limitom blokov v grafickom rozhraní navrhujú zložito [6], [7]

- Implementovať modely vyžaduje zo začiatku mnoho skúsenosti a úsilia

Potenciálnu výhodu nájdeme aj z možným prechodom z klasického V cyklu vývoja na Y cyklus. To by znamenalo ešte výraznejšie zredukovanie vývojového času, nákladov či chýb vytvorenými ľuďmi. Y cyklus prezentuje obrázok č. 1.3. Ako je vidieť, vďaka kvalifikovaným nástrojom dokážeme zautomatizovať celý vývoj až na generovanie kódu. Pri používaní ďalších nástrojov je dokonca možné z automatizovať všetky integračné testy a simulácie. [6], [7]



Obrázok 1.3 Možná úprava WaterFall modelu na Y model [27]

Pri používaní MBD je taktiež kladený väčší dôraz na požiadavky. Modely, ktoré reprezentujú tieto požiadavky sú následne oveľa ľahšie čítanejšie ako pri klasickom softwaru. Jednou z najdôležitejších výhod je neustály nárast počtu nástrojov, ktoré pomáhajú pri validácii a verifikácii.

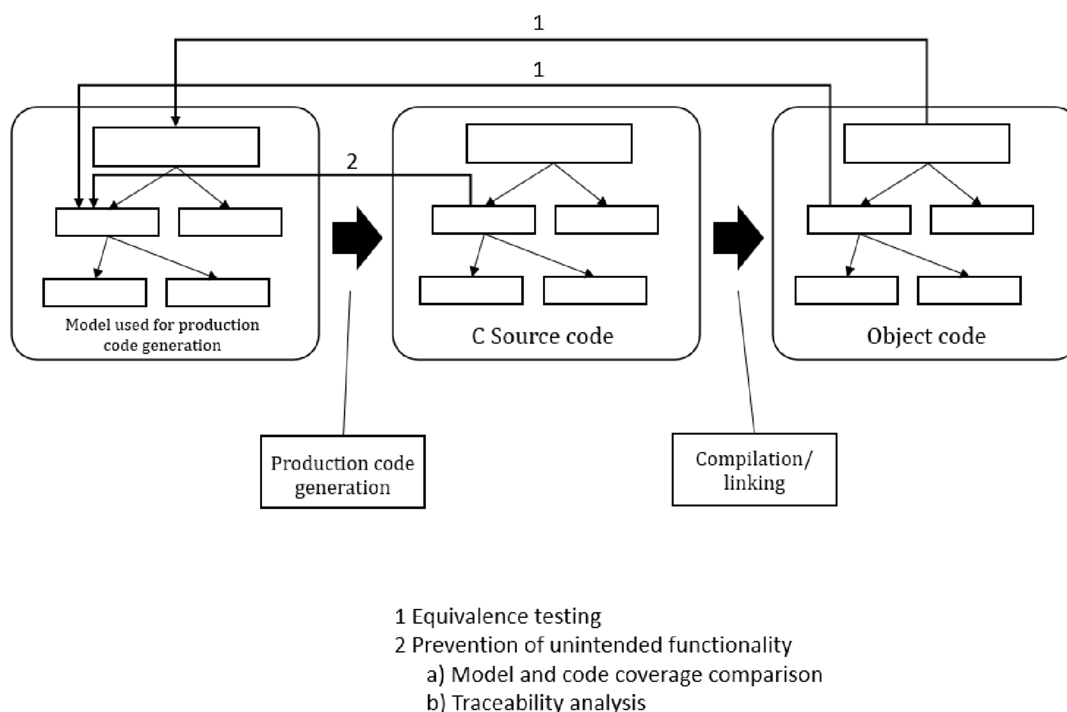
2. VALIDÁCIA A VERIFIKÁCIA GENEROVANÉHO KÓDU

V tejto kapitole budú rozobrané dôležité súčasti validácii a overovania generovaného kódu. Budú zdôraznené jednotlivé prístupy, rôzne nástroje a riešenia v rámci rôznych frameworkov. V oblasti generovania kódu môžeme obecne povedať, že verifikácia definuje otázku na použité produkty, nástroje a komponenty, v ktorých pracujeme. Pýta sa či dané zadanie dokážeme pomocou nich vybudovať, alebo či vytvárame aplikáciu správne. Validácia sa zaoberá druhou stránkou veci, a to či produkt ktorý vytvárame pomocou verifikovaných nástrojov je tým čím ma byť. Definuje otázku na vytváranú aplikáciu: Vytvárame správnu aplikáciu ?

S príchodom používania model based design pre vývoj aplikácii v embedded oblastiach ako letecký alebo automobilový priemysel, kde sú projekty ešte značne náročnejšie a zložitejšie sa museli vytvoriť postupy a nástroje, vďaka ktorým bude riešenie reálne dosiahnuteľne a proces produktívny. Jednotlivé grafické prostredia dosahujú zvýšenie produktivity, zníženie nákladov atď. Kde si veľmi jednoducho vygenerujeme použiteľný kód. Avšak museli sa vytvoriť ďalšie nástroje, ktorými takto vygenerovaný kód vieme overiť. To by nebolo možné dosiahnuť bez jednotlivých nástrojov či postupov. Všeobecné platí, že pre akýkoľvek software sa musia do procesu overovania zahrnúť rôzne kombinácie technik a nástrojov, na hľadanie chýb a overovanie správnej funkčnosti kódu. V podstate takto vygenerovaný kód by sa kľudne mohol overovať štandardným spôsobom ako ručne písaný kód, no to by nebol žiadny prínos v rámci efektívnosti procesov pri generovanom kóde. V skratke, súbor tých správnych nástrojov nám pomáha pri riešení jednotlivých problémov vo vývoji, zjednodušuje prácu, znižuje chybovosť a konečna verifikácia a validácia je omnoho jednoduchšia a dokázateľná. Jednotlivé normy ako IEC 61508 s integritami bezpečnosti SIL1 až SIL 4 či ISO 2626 pre automobilový software s levelmi ASIL A až ASIL D sa nezaoberajú ako správne verifikovať a či validovať automaticky generovaný kód. [9],

2.1 Princíp a prístupy nástrojov

Výsledkom správnej validácie je dosiahnutie toho, že navrhnutý model a jeho správanie sa zachovalo aj počas generovania kódu a jeho kompilácii. Následne pomocou systematických testov, ktorými testujeme rovnocennosť a rovnosť medzi použitými modelmi a vygenerovaným kódom verifikujeme správnu funkčnosť takto vygenerovaného kódu. Týmto dosiahneme vysokej pravdepodobnosti správneho fungovania generátoru kódu, kompilátora ci linkera.



Obrázok 2.1 Vzťahy testovania medzi modelom a generovaným kódom [10]

Ako je znázornene na obr. 2.1, validácia by sa mala skladať z testov, ktoré dokazujú rovnosť medzi model a generovaným kódom. Tieto testy by mali byt aj na úrovni subsystémov, alebo celého systému ako celku. Testy sa netestujú na vygenerovanom C kóde ale priamo už na skompilovanom objekte z generovaného kódu. Následne sa musí porovnať funkčnosť modelu a generovaného kódu. To môžeme rozdeliť na trasovateľnosť a pokrytie.

Preto pristupujeme k validácii a verifikácii buď:

1. Numerickými metódami
2. Prevenciou pred nežiaducou funkcionalitou

2.1.1 Numerická metóda

Táto metóda by mala numericky preukázať rovnosť medzi modelom a vygenerovaným kódom. Preto sa vytvoria testy, vďaka ktorým sa porovná používaný model s jeho následne skompilovaným C kódom. Vytvoria sa testy, ktoré majú identické vstupy ako pre vygenerovaný kód tak aj pre model. Jednotlivé výsledky testov sa následne numericky porovnajú a tým sa preukáže či je logika a funkčnosť vytvoreného modelu zachovaná aj naďalej v skompilovanom objekte. Niekedy sa môže stať, že jednotlivé výstupy pri testoch nie sú známe, tým pádom sa overí funkčnosť aj v prípadoch, ktoré nie sú jasne dané a neboli popísané v zadaní či následných požiadavkách. Správne by sa testy mali dať aplikovať ako na celý design, tak ako na niektoré vybrané komponenty či subsystémy. Dobrý nástroj musí mať možnosť vytvorenia vlastných testov, vďaka ktorým vývojár

dokáže dosiahnuť požadovanú rovnosť medzi modelom a kódom. V prípade, že niektoré časti modelu sa nedajú otestovať, príčina a prípadný dopad sa musí dôkladne zdokumentovať a popísať. Testovanie by malo prebiehať v prostredí, ktoré najviac odpovedá prostrediu kde bude výsledný produkt nasadený. Z toho vyplýva, že v ideálnom prípade by sa mali testy vykonávať aspoň pomocou PIL testov. Ak to daný projekt nedovoľuje, musia sa rozdiely medzi prostrediami testovania a nasadenia brat v úvahu a určiť možný dopad a rozdiel vo výsledkov testov. Po vykonaní dosiahneme určitého výsledku. Často sa však stane, že výsledky jednotlivých testov nebudú úplne rovnaké a to aj v prípade dokonalého generátoru kódu a návrhu. Je to spôsobené rôznymi kompilátormi, rozdielmi v presnostiach, maxím a miním alebo kvantiznačnemu efektu pri používaní fixed-point matematiky, tzv. pohyblivej radovej čiarky. To znamená, že na overenie testov potrebujeme zvoliť vhodný algoritmus, pre náš nastroj, ktorý výsledky testov porovná. Vo vyhodnocovaní testov, by správny nástroj mal obsahovať možnosť určenia prahovej hodnoty, pomocou ktorej dokážeme výsledky testov vyhodnotiť. Preto by nastroja alebo súbor nástrojov na overovanie kódu mal dokázať testy vytvoriť, vykonať a vyhodnotiť. [9]

2.1.2 Prevencia pred nežiaducou funkcionalitou

Druha časť overovania spočíva v dokazovaní toho, že vygenerovaný kód alebo model nevykonáva žiadne nedefinované a nežiadúce aktivity. Väčšinou sa porovnáva štrukturálna ekvivalencia na úrovni modelu a na úrovni kódu. Riešenie by sme mohli rozdeliť na dve oblasti. Jedna oblasť sa bude venovať porovnaniu pokrytia generovaného kódu a modelu. Druhá oblasť sa zaoberá tvarovateľnosťou.

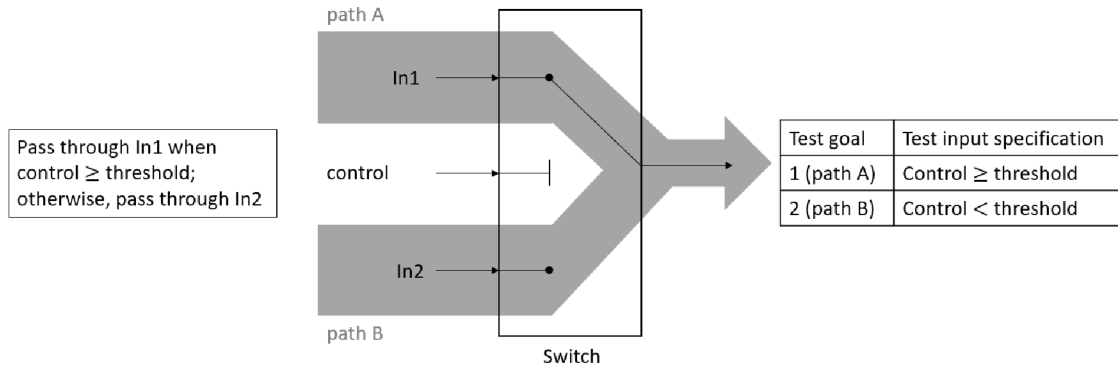
V rámci model based design vývoja sa môžeme pozerať na pokrytie dvomi pohľadmi. Na jednej strane, pokrytie v rámci modelu a na druhej pokrytie vypočítané na základe kódu. V rámci oboch pokrytí uvažujeme o štrukturálnych pokrytiach. To znamená, výsledok takéhoto pokrytia priamo závisí na počtu vykonaní testov pokrytia, pričom novo definované testy sa zameriavajú na elementy, ktoré ešte neboli otestované. Cieľom je dosiahnutie pokrytia 100% .[9]

2.1.2.1 Pokrytie modelu

Neexistujú žiadne obecné kritéria ako by mal nástroj na overovanie pokrytia modelu fungovať. Na trhu sa nám však podarí nájsť nejakú tú spoločnú vlastnosť. Je ešte lepšie ak daný nástroj dokáže merať pokrytie v rámci SIL simulácie. Základná schopnosť nástrojov by mala spočívať v meraní pokrytia modelu aspoň v dvoch základných kritériách.

Decision coverage (DC), ktoré v rámci jedného rozhodovacieho bloku v modelu, urči percento pokrytia rozhodnutia z celkového počtu možných ciest, ktorými simulácia mala prejsť. To znamená, že nástroj by mal byť schopný určiť počet možnosti pre každé

rozhodnutie v rámci rozhodovacieho bloku a porovnať to s počtom vykonaných rozhodnutí pri testoch. [11], [9]



Obrázok 2.2 Príklad Decision Coverage [12]

Condition coverage (CC) analyzuje výstup z modelu na základe kombinácii vstupov. Ak počas simulácii sa každý vstup a výstup logického bloku dostal do úrovni false a true, je pokrytie C1 na úrovní 100%. Kontroluje výstup celej štruktúry ako napríklad switch či if. [11]

Branch coverage (BC) overuje či každý blok v rámci modelu, ako vyber MinMax, switch, logické operátory atď. vykonal všetky operácie aspoň jeden krát. [11]

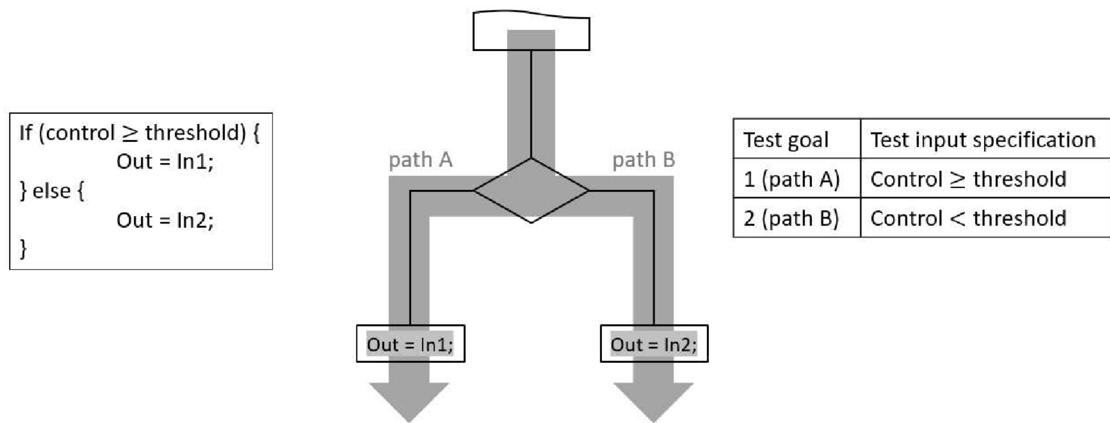
2.1.2.2 Pokrytie kódu

Typická metodika merania pokrytia kódu je zameraná na tok kódu. Následné test procedúry majú za úlohu definovať podmienky používania informácii v toku programu ako sú napríklad premenne, polia atď. Úlohou je zistiť a overiť ako sa priradené dáta menia a interagujú v priebehu programu. [11]

Statement coverage (C0) rozhoduje ako a či bola podmienka v kóde vykonaná aspoň raz. Dá sa implementovať aj ako otázka: Bol každý riadok kódu aspoň raz vykonaný?

Condition coverage (C2) týka sa výrazov, ktoré obsahujú relačne operátory a logickú negáciu ale neobsahujú logické operátory ako && a ||. Vyhodnotí či výsledok bol aspoň raz vyhodnotení ako pravda a aspoň raz ako nepravda.

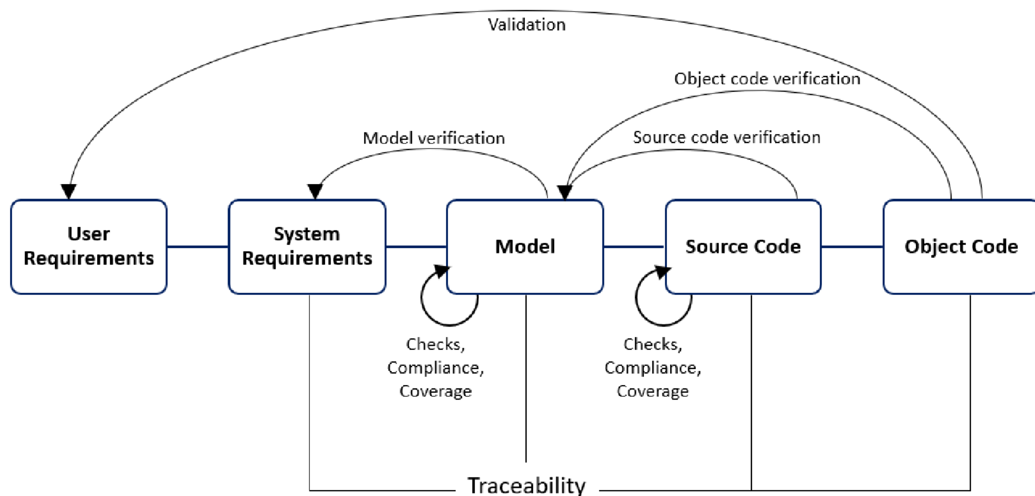
Branch coverage (C1) cieľ je prejsť každou vetvou programu až k jeho ukončeniu, či riadiacemu cyklu. Zase všetky možné výstupy každej štruktúry musia byť aspoň raz vyhodnotené.



Obrázok 2.3 Príklad Branch Coverage [27]

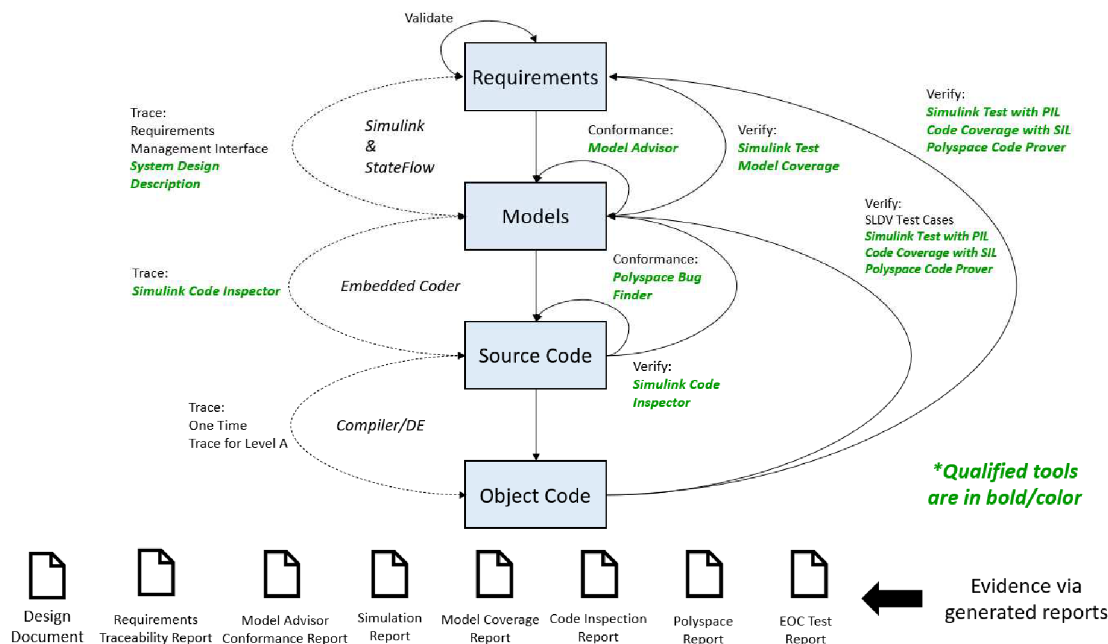
2.1.2.3 Trasovateľnosť

Najjednoduchšie povedané, trasovateľnosť je nevyhnutná na to, aby sme dokázali sledovať to čo vytvárame. Vďaka matici trasovateľnosti by malo byť možné dohľadať splnenie každej požiadavky. Ak nedokážeme vytrasovať časť kódu na základe splnenia nejakej požiadavky, nemal by tam byť. Trasovateľnosť je vice-versa a to musí byť preukázateľné medzi systémovými požiadavkami, modelom, zdrojovým kódom a objektom.



Obrázok 2.4 Vzťahy medzi požiadavkami modelom a vygenerovaným kódom [27]

V nasledujúcich kapitolách budú popísané jednotlivé riešenia firiem na trhu, čo sa v rámci generovania, validácii a verifikácii kódu týka. Ako nástroje, ktoré ponúkajú a taktiež napríklad rozhrania medzi generovaným a ručne písaným kódom.



Obrázok 2.5 Možné použitie nástrojov pri validácii a verifikácii v Simulinku [27]

2.2 MathWorks

MathWorks je riešenie v oblasti model based design, ktoré pozná každý. Kompletný framework vznikne iba spojením viacerých nástrojov. Takéto mocné a zároveň najpoužívanejšie nástroje ponúka práve riešenie od firmy MathWorks. Nástroje, ktoré vytvárajú na modelovanie, simulácie a generovanie kódu sú jednými z najkvalitnejších na trhu. Patria medzi nich Matlab, Simulink, Matlab Coder, Simulink Coder, StateFlow a StateFlow Coder. Pre generovanie C kódu z modelu sa používa Embedded Coder.

Embedded coder rozširuje Matlab coder a Simulink coder o pokročilé generovanie C kódu. Generuje čitateľný, kompaktný a stručný kód s pokročilou optimalizáciu a presnou kontrolou nad generovanými funkciami, súbormi a dátami. Podporuje generovanie pre AUTOSAR a môže generovať kód v rámci MISRA C. Samotný embedded coder pri vytváraní projektu jednotlivé časti kódu dokumentuje a trasuje. V neposlednom rade je kvalifikovateľný na verifikáciu software pre DO178, IEC 61508 či ISO 26262 normy. Vygenerovaný kód je univerzálny, preto sa dá použiť na každom procesore. No postupom času MathWorks pridáva do svojich knižníc procesory s vlastnými ovládačmi, na ktorých je kód vysoko optimalizovaný.

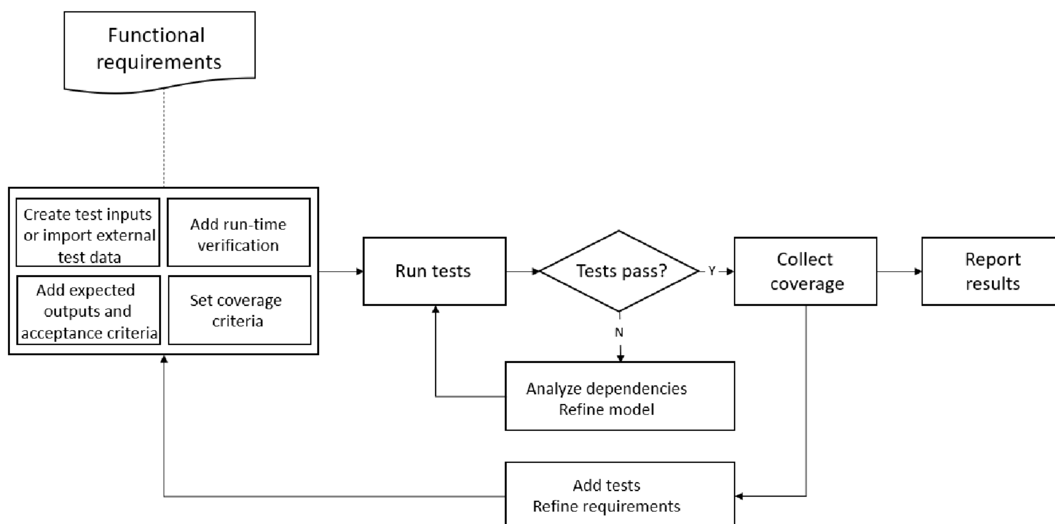
Simulink je grafické prostredie pre modelovanie a analýzu dynamických systémov pomocou grafických nástrojov ako blokové diagramy spolu so sadmi knižníc blokov. Spolupracuje a je integrovaný s prostredím MATLAB pričom mu môže byť nadriadený alebo byť riadený. Simulink sa využíva na automatické riadenie či napríklad digitálne spracovanie obrazu.

2.2.1 Simulink Coverage

Nástroj vykonáva analýzu pokrytia modelu aj kódu. Jednotlivé testy vychádzajú z daných požiadaviek. Nástroj produkuje interaktívny report, ktorý zaznamenáva, ako často a koľko krát boli vytvorené funkcie v Matlabe, S-funkcie alebo generovaný kód použité. Tento nástroj je možné použiť aj na numerické porovnanie ekvivalencie medzi modelom a kódom. Pokrytie sa môže vyhodnocovať v SIL alebo PIL testoch a okrem základných metrík ako Decision Coverage (DC), Condiiton Coverage (CC), Branch coverage (BC) sa dajú využiť metriky, ako Signal Range Coverage (RC) vďaka, ktorej Simulink zaznamená minimálne a maximálne hodnoty signálov v každom bloku v modelu. Modified Condition/Decision Coverage (MCDC) je funkcia, ktorá testuje nezávislosť blokov alebo prechod v StateFlow. Test pre blok dosiahne úplne pokrytie ak pri zmene jediného vstupu (akéhokoľvek) sa výstup nezávisle zmení. Pre StateFlow je pokrytie dosiahnuté ak existuje aspoň jedna zmena ktorá vykoná prechod do ďalšieho stavu a to pre každú podmienku. Nástroj obsahuje ďalšie z mnoho metrík ako saturácie, veľkosti atď.

2.2.2 Simulink Design Verifier

Jeho primárnou úlohou je identifikovanie skrytých chýb v dizajnu modelov ako sú pretečenia, nesprávna logika, delenie nulou, zacyklenie atď. Systematicky overuje model pomocou požiadaviek, verifikuje a overuje či sú vlastnosti modelu držané pod maximálnymi limitmi, a taktiež optimalizuje a analyzuje veľké komplexne modely. Pre generovaný kód je najpodstatnejšie to, že rozširuje informácie o pokrytí pomocou testových požiadaviek, pričom jednotlivé testy môžeme upravovať a hlavne vytvárať ďalšie nove testy so vstupmi, ktoré neobsahovali požadované testy. Použitím týchto novo vytvorených testov sa nájdu chýbajúce časti v zadaní a požiadavkách.



Obrázok 2.6 Typický príklad tvorby a overovania testov [13]

Na obrázku č. 2.6 je zobrazená typická štruktúra vytváranie a overovanie testov. Je jasné ,že testy môžeme rozširovať o ručne napísané testy aby sme zvýšili pokrytie modelu. Taktiež je tu možnosť vytvárať test cases zo systémových požiadavkou a v neposlednej rade vytvárať tests cases pre vygenerovaný C kód a volaný externý C kód v Simulinku. [13]

Všetky navrhnuté testy sa následne vykonávajú priamo vďaka Embedded coderu v SIL alebo PIL prostredí. Následne po vykonaní testov sa s použitím ďalších nástrojov výsledky testov porovnajú. Máme možnosť navrhnúť algoritmy pre porovnanie výsledkov testov priamo v jazyku MATLAB. Najviac používaným algoritmom na porovnanie výsledkov testov je maticový rozdielový výpočet.

2.2.3 Simulink Fixed Point.

Čísla v digitálneho hardware sú reprezentované s pevnou desatinnou čiarkou alebo s plávajúcou desatinnou čiarkou. Pre obe typy reprezentácii sú veľkosti zafixované ako počet bitov. Fixed-point reprezentácia lepšie aproximuje čísla, zjednodušuje implementáciu systému, znižuje spotrebu procesoru a má mnoho ďalších výhod. Aby sa vykonávali skutočne bitové simulácie modelov a dokázali sa vyfiltrovať nepresnosti maximálnych hodnôt a používaním fixed point reprezentácii sa vytvoril nástroj Simulink Fixed Point. Pri použití s Embedded kóderom umožňuje tento nástroj generovanie pravého C kódu. Následne porovnávanie fixed point modelu s fixed point kódom uľahčuje vyhodnocovanie signálov, ktoré je potrebné vyhodnotiť pri testoch. Takto vzniká dostatočná podobnosť kódu s modelom, čím sa dokazuje že model a kód je vygenerovaný správne a sú totožné. [11]

2.2.4 Simulation Data Inspector

Ak na vyhodnocovanie testov nepoužijeme vlastne algoritmy tak jedná z možnosti je vyhodnocovanie testov pomocou tohto nástroja.

2.2.5 Trasovateľnosť

Na generovanie matice trasovateľnosti a sledovacích informácii až do vygenerovaného kódu je možne využiť automaticky generované komentáre blokov Simulink. Oddiel v Embedded Coder – Traceability Report pomáha používateľom trasovať požiadavky medzi modelom a kódom. Používaním možnosti „Code to lock highlighting" generujeme hyperlinky, ktoré poukazujú na modely z ktorých bola časť kódu vygenerovaná. Možnosť "Block to code highlighting" funguje naopak, to znamená že pre každý blok vidíme vygenerovaný kód. Vďaka možnosti v rámci IEC Certification Kit maticu tvarovateľnosti exportujeme priamo do Excelu. Existuje ešte možnosť, že do generovaného kódu bude zahrnutý zdrojový kód ako komentár. [11]

2.2.5.1 3rd Party tools

Dnes už existuje nespočetné množstvo nástrojov tretích strán podporovane Embedded Coderom a Simulinkom. Pre výpočty pokrytia modelu alebo kódu je najznámejší nástroj BullseyeCoverage či LDRA TestBed. Sú to nástroje vytvorené firmami, ktoré sa zameriavajú iba na testovanie softwaru. Preto sú tieto nástroje veľmi intuitívne, profesionálne a výrazne zjednodušujú a zrýchľujú porovnanie pokrytia modelu a kódu.

2.2.6 Interface

Embedded Coder nedokáže ponúknuť používateľovi úplnú kontrolu nad rozhraním generovaného kódu. Prakticky sa dajú využiť dve možnosti generovania. To funkcií opakovateľné a neopakovateľné. Ak sú funkcie opakovateľné tak vstupy a výstupy sú nastavené ako argumenty funkcii. Embedded Coder nepodporuje návrat funkcií. V Simulinku je však dovolené konfigurovať blok ako funkciu, ktorú následne môžeme volať v ručne písanom kóde. V úvahu pripadajú aj štruktúry, ktoré taktiež vznikajú úpravou blokov alebo vytváranie globálnych premenných. Ručne písaný kód či externe vygenerovaný kód voláme pomocou takzvaných C a S funkcii v Simulinku. S príchodom Embedded Coder Interface sa situácia výrazne zmenila a používateľ dostáva oveľa viacej priestoru nad kontrolu rozhraním medzi vygenerovaným a ručne písaným kódom. [11]

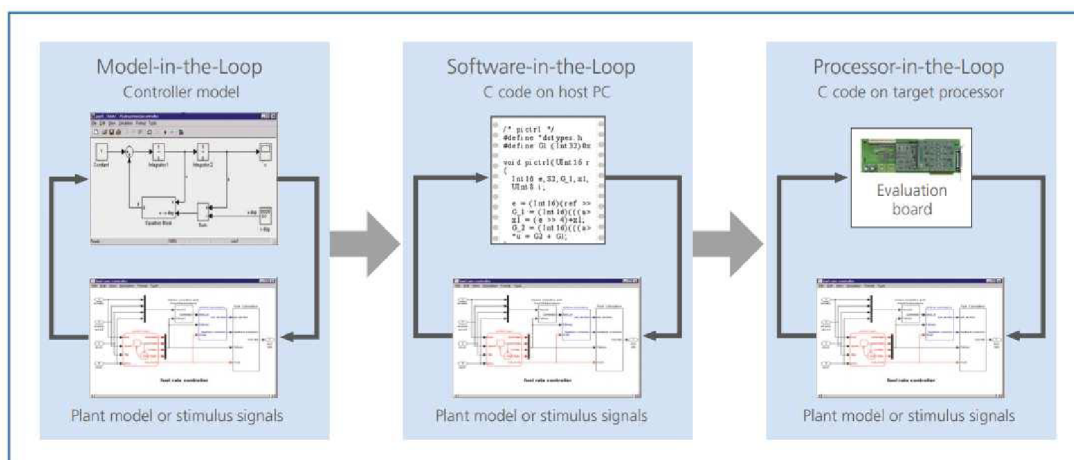
2.3 dSpace

Spojenie model based design s možnosťou generovania C kódu a prepojenia priamo na embedded systémy je jedným z riešení od spoločnosti dSpace. Generátor kódu TargetLink dokáže generovať vysoko efektívny C kód priamo z prostredia Simulink a StateFlow. DSpace vyvíja a distribuuje software a elektroniku pre riadiace jednotky, hlavne v automobilovom priemysle, robotike ale aj aerospace.

2.3.1 TargetLink

Ponúka vygenerovanie kódu s plávajúcou desatinnou čiarkou priamo z Simulinku/Stateflow modelov. Generovaný kód je zväčša použitý pre AUTOSAR-(Automotive Open System Architecture). Využíva automatického škálovania, s kompiláciu v štandarde MISRA. Ak neimportujeme modely, môžeme ich navrhnúť pomocou vlastných TargetLink knižníc, ktoré sú prakticky rovnaké ako tie v Simulinku. Preto pochopiteľne TargetLink, slúži hlavne na konvertovanie grafických modelov zo Simulinku alebo StateFlowu, ktorý TargetLink konvertuje na vysoko kvalitný produkčný kód. Preto DSpace ako platforma s ich generátorom sa používa v hojnom množstve práve na validáciu a verifikáciu modelu a kódu. [14]

Ak sa budeme zaoberať testovaním numerickými metódami TargetLink podporuje tri rôzne simulačné módy na testovanie. Jedna sa o testy v rámci MIL, SIL a PIL.



Obrázok 2.7 Simulačné módy v TargetLinku [14]

Pri testoch v rámci PIL podporuje obrovské množstvo procesorov a dSpace ako taký ponúka riešenia na svojich HW vývojových platformách. Zmena medzi MIL/SIL/PIL je veľmi jednoduchá a praktický k tomu stačí iba jeden klik. Testované dáta TargetLink automaticky loguje a vykresľuje podľa nastavenia užívateľom. Výsledky medzi testami v MIL/SIL/PIL sa taktiež automatický porovnávajú. To je pre užívateľa veľmi prívetivé a veľmi jednoducho, bez zložitého nastavovania a upravovania, či písania skriptov dokáže porovnať výsledky testov a namerané dáta. PIL testy automatický merajú využitie procesoru, aktuálnu zabranú veľkosť pamäti RAM alebo ROM pre každú funkciu. [14]

Sám TargetLink ma implementovaný nástroj ktorý meria prekrytie kódu a modelu. Vykonáva analýzu C0 alebo C1. Vygeneruje následný report a užívateľ si pomocou neho dokáže navrhnúť ďalšie nové testy. Pre jednotlivé parametre a signály môžeme nastavovať minimálne a maximálne rozsahy, tie sa automaticky šíria do ďalších subblokov či funkcií. Vďaka automatickému škálovaniu vieme nastaviť worst-case scenár, kde sa nastavia hraničné hodnoty jednotlivých parametrov. Výsledný kód dokážeme generovať ako pre celý systém tak aj pre jednotlivé subsystémy. To všetko sprevádza dokumentácia, ktorá sa automaticky aktualizuje a generuje s postupným vytváraním a upravovaním kódu. V neposlednom rade TargetLink ponúka automatickú model-code trasovateľnosť, ktorá môže byť generovaná v rôznych formátoch a následne analyzovaná.

2.3.1.1 TargetLink EcoSystem

Ak nám vlastnosti TargetLinku nepostačujú, využijeme akékoľvek nástroje v rámci ekosystému, ktorý TargetLink ponúka. [14]

Tabuľka 2.1 TargetLink Ecosystem

Požiadavky	Model Coverage	Tvorba testov a model-based testovanie
TargetLink	Simulink Coverage	Reactis
Simulink Requirements	BTC EmbeddedTester	BTC EmbeddedTEster
IBM Ration DOORS		MES Test Manager
PTC Integrity		
BTC EmbeddedSpecifier		
Code Coverage	**Softwarove testovanie**	**Architektura a iné**
Testwell CTC++	dSPACE VEOS	TPT
BTC EmbeddedTester	dSPACE Real-Time HW	SystemDesk, DaVinci Developer
		Polyspace Code Prover, Astrée
		StackAnalyzer, TimingProfiler
		QA-C, PC-lint

2.3.2 Interface

TargetLink ponúka jednoduché grafické rozhranie kde si môžeme slobodne ovplyvňovať rozhranie vygenerovaného kódu. Prostredie umožňuje užívateľovi zvoliť výstupne hodnoty pre substystémy ako návratové hodnoty, premenne, parametre atď. Pre každý signál sa dá vytvoriť návratová funkcia a ta je volaná napríklad v ručne písanom kóde. Taktiež ako v Simulinku sa tu nachádza podpora blokov, ktoré volajú externe ručne písaný kód.

2.4 Ansys SCADE Suite

Ansys SCADE Suite je moderne vývojové prostredie v model based design. Je jedna z mála firiem, ktorým sa podarilo definovať vlastný jazyk – Scade, ktorý umožňuje modelovať, vyvíjať, simulovať, overovať a generovať plne kvalifikovateľný a verifikovateľný kód spolu s ďalšími nástrojmi. To znamená oblasti záujmu v rámci Suite sú k návrhu softwaru v areospace, kde sa riadia motory, automatický piloti, palivové čerpadla či v rámci riadenia vlakov, elektrárni alebo automobilov. Riešenia sľúbia pomerne vysoké zníženie nákladov na certifikáciu a vytvorený generátor kódu generuje kód, ktorý splňuje normy DO-178B/A alebo DO-330 či DO178C. [15]

Vytváranie testov a upravovanie je implementované priamo v prostredí SCADE Suite. Testy môžeme spúšťať a generovať ich reporty taktiež priamo v SCADE Suite, ktoré sa automatický nahrávajú a ukladajú. Šablóny testov môžu byť kľudne napísané v Microsoft Exceli a následne importované. Ak si užívateľ chce na testy pridať rôzne grafy a tlačidla, použije nástroj Test Rapid Prototyper. Automatické škálovanie je samozrejmosťou, presnosti v rámci dát sa dajú ručne či automatický nastaviť.

Vykonávané testy sú kvalifikované v rámci DO-178C/DO330 a garantujú správnu funkčnosť modelu na riadiacej jednotke.

SCADE Test Model Coverage ako doplnok SCADE Suite analyzuje pokrytie modelu a kódu. Tento nástroj môže byť aplikovaný high-level požiadavkami. Aj pri použití SCADE Test Target Execution, sme v rámci PIL schopný merať jednotlivé pokrytia. Sú testované nasledovné metriky: Branch coverage (BC), Decision coverage (DC), a Modified condition/decision coverage (MC/DC). [15]

SCADE Suite udržuje automatický trasovateľnosť až ku generovanému kódu. Čo sa týka použití 3rd party nástrojov, je to zase nespočetné množstvo ako u TargetLinku od dSpace.

Je potrebné dodať, že bez zakúpenia produktu od SCADE je veľmi obťažné získať nejaké informácie v rámci možných postupov alebo nástrojov. No výsledky vidíme na mnohých projektov a ich riešenia patria medzi tie najlepšie na svete.

2.5 Ostatné

Spoločnosť ALTAIR vytvorila framework Altair Embed s jednoduchým grafickým prostredím pre model based design, kde sľubujú kompaktný kód so šetrením pamäte a času. Model môžeme priamo vyskúšať v prostredí pre podporované procesory, pričom nepoužijeme ani vetu ručne písaného kódu. Avšak čo sa týka nástroju je pomerne mladý, jediná vec zdokumentovaná je v rámci jednotlivých testov v SIL, PIL a HIL. Nástroje na trasovateľnosť, pokrytie modelu a kódu, vytváranie testov zatiaľ nie sú dotiahnuté do konca.

V oblasti nájdeme aj riešenia zdarma. No tie nie sú väčšinou kompletné, chýbajú nástroje, generátor nie je zdokumentovaný alebo nepodporujú vstupy z model based design programov.

3. KRITICKO-NÁROČNÉ SYSTÉMY A MODEL BASED DESIGN

Za jednu z mnohých definícii bezpečnosti môžeme považovať stav bezpečia, ktorý chráni pred nežiadúcimi vplyvmi ci ujmou. Preto je to stav, ktorý nás dokáže ochrániť pred situáciami ako smrť, zranenie, či poškodenie alebo straty komponentu, majetku alebo aj pred poškodením životného prostredia.

Pojem kriticko-náročný v rámci softwaru je často veľmi subjektívny pojem. Kriticko-náročný software definujeme ako taký software, ktorý je používaný v aplikáciách a systémoch v ktorých môže dôjsť k neakceptovateľným stavom. To znamená, že pri systémoch, ktorého činnosť alebo chyba môže viesť k nežiadúcemu (hazardnému) stavu. Software musí byť schopný zotaviť systém z tohto hazardného stavu a zmierniť jeho následky.

Definícia podľa NASA hovorí že software je bezpečne kriticko-náročný ak splňuje aspoň jedno z týchto kritérií:

1. Je súčasťou bezpečno-kritického systému a splňuje aspoň jedno z nasledujúcich:
   - Procesor je spoločný aj pre systém
   - Zmierňuje riziká vytvorené hazardnými stavmi
   - Kontroluje, ovláda, spracováva bezpečno-kritické funkcie či príkazy
   - Ak sa systém dostane do hazardného stavu, detekuje, reportuje alebo vykonáva akcie
   - Spôsobuje alebo prispieva k nebezpečným stavom
2. Spracováva dáta, ktoré môžu priamo viesť k bezpečnostným rozhodnutiam
3. Jeho úlohou je validácia alebo verifikácia bezpečno-kritického systému a to buď hardwaru, softwaru alebo aj aj. [6]

3.1 Normy

Pri vývoji takéhoto softwaru sa spravidla musia dodržať niekoľko dôležitých faktorov:

1. Požiadavky a systémová architektúra musí byť kvalitne zdokumentované.
2. Využívanie bezpečných praktík na každej úrovni vývoju. Počas celého vývoja je dôležite identifikovať a ukladať všetky potencionálne riziká.
3. Správna implementácia. Všetky požiadavky musia byť implementované bez nežiaducich aktivít.
4. Kvalifikovaný personál. Pri vývoji takéhoto softwaru by vybraný ľudia, hlavne vedúce posty mali zastávať perfekcionisti.
5. Testovanie. Testovať software na každom stupni vývoja.

V rámci rôznych odvetví je táto kapitola zameraná na normy, ktoré sú požívané v leteckých aplikácii. Je jasné, že práve toto odvetvie je jedno z najviac náročných na vývoj aplikácií. Preto aj keď nasledujúce normy budú určené pre letecký priemysel, ich dodržiavanie a používanie môžeme vykonávať aj pri iných oblastiach priemyslu.

3.1.1 DO-178C

Dokumenty noriem DO-178C a DO-278A sú základné dokumenty, ktoré poskytujú informácie a koncepty na ktorých sú ďalšie dokumenty či normy postavené. Obidva dokumenty sú praktický rovnaké s tým rozdielom, že DO-178C je zameraná na software v lietadle a DO-278A je zameraná na software používaný na zemi. Pretože sú skoro rovnaké, celkovo je viac obľúbená norma DO-178C. Norma je rozdelená postupne do 12-tich sekcií. Od prvej sekcii kde je základné zoznámenie s dokumentom. Následne sa zaoberá framework, ktorý korešponduje so smernicou ARP4754A. Ďalšie sekcie sa zaoberajú životným cyklom procesu, vysvetlením plánovania procesu či aktivít v rámci vývojového procesu. Následne sa vysvetľuje verifikačný proces, manažment konfigurácie v rámci objektív a procesom zaistenia kvality. Tak je zhrnutý proces certifikácie, identifikovaný životný cyklus dát pri plánovaní, vývoji a verifikovaní softwaru. Na konci dokumentu sa nachádzajú apendixy, či pokyny k niektorým ďalším úvahám v rámci predošlých sekcií. [16]

3.1.2 DO-331

Táto norma je založená na norme DO-178C. Norma taktiež modifikuje a pridáva postupy riadenia pre MBD systémy. MBD vývoj a verifikácia MBD je obsiahnutá v tejto norme. Poskytuje pokyny ako pre simulácie modelov, pokrytie modelov a dizajn modelov. Najväčšie rozdiely medzi DO-331 a DO178C sú v sekciách 5 a 6 tj. vývojový proces a proces verifikácii. Norma definuje model ako abstraktné časti softwaru a systému, vďaka ktorým podporujeme vývoj softwaru alebo verifikáciu softwaru. Model musí byť úplne popísaný a explicitne identifikovaný, obsahovať softwarové požiadavky alebo definíciu softwarovej architektúry a taktiež jeho forma a typ sa používa pre priamu analýzu alebo hodnotenie chovania. Pri splnení týchto vlastností, norma hovorí o modelu. Modely môžu vstupovať do hierarchie požiadavkami ako systémové či softwarové požiadavky alebo ako softwarový dizajn. Aj keď inžinieri už roky používajú modely ku grafickému znázorneniu ovládacích prvkov, použite kvalifikovaných modelovacích nástrojov, ktoré automaticky generujú kód a testovacie vektory je relatívne nedávny posun. Teraz budú vypichnuté najdôležitejšie rozdiely, aspekty v rámci normy DO-331. [6]

Fáza plánovania. Počas fázy plánovania norma požaduje, aby plány vysvetlili použitie modelovania a jeho začlenenie do životného cyklu softwaru. Plány musia taktiež určiť a vysvetliť aké dáta jednotlivé modely predstavujú v rámci cyklu, aké štandardy budú použité a aký je prístup k overeniu modelu. Pri vývoji kriticko-náročného softwaru,

norma požaduje aby sa v tejto fáze definovalo prostredie modelovania a používané metódy či nástroje. Následne podľa toho, ako sú simulácie používané sa musí simulátor kvalifikovať. Kvalifikáciu takýchto nástrojov popisuje v norme kapitola 13.

Pri používaní modelov musia byt používané štandardy, ktoré vysvetľujú princípy a techniky modelovania. Každý z štandardov by mal obsahovať zdôvodnenie a vysvetlenie jednotlivých nástrojov a metód, identifikáciu používaného jazyka a zdôraznenie jeho funkcii a obmedzení. Pokyny a komplexne pravidla ktoré umožňujú, aby bolo modelovanie jednoznačne a deterministické v rámci DO-331 cieľov. Standard musí obsahovať aj informácie a návody ktoré zaisťujú správne používanie knihovní pri modelovaní. V neposlednom rade štandard by mal obsahovať metódy k identifikácii požiadaviek a to aj tých odvodených. Štandard musí identifikovať hocijaké ďalšie elementy, ktoré nie sú softwarové požiadavky alebo dizajn.

Knihovne modelových prvkov musia mať zaistenú príslušnú úroveň softwaru podľa DO-178C. V podstate všetky knihovni musia prejsť rovnakým procesom ako software vyvíjaný pre kriticko-náročne aplikácie. Knižnica, ktorá neprešla takýmto vývojom a nezodpovedá príslušnej úrovni nesmie byť použitá.

Norma definuje pokrytie modelu ako analýzu ktorá určuje, ktoré požiadavky vyjadrene dizajnom modelu neboli overené vo verifikačnom procese. Tieto požiadavky sú založené na požiadavkách, z ktorých bol vytváraný dizajn modelu. Účel tejto analýzy je podpora detekcie nezamýšľaných funkcii v dizajnu modelu.

Poslednou z najviac odlišných časti normy oproti DO-178C je okruh modelových simulácii. Norma poskytuje konkrétne pokyny k simuláciám modelov. Ak sa kvôli splneniu niektorých požiadaviek používajú modelové simulácie je nutne jednotlivé simulačne prípady preskúmať a výsledky simulácie vysvetliť a preskúmať. Je potrebné overiť, že sú simulačne prípady, simulačne procedúry a výsledky správne. [6]

3.2 Príklady a súhrn

Kriticko-náročné systémy z hľadiska bezpečnosti musia splňovať štandardy, ktoré často potrebujú nesmierne množstvo času a nákladne procesy. Preto sa model based design stáva čím ďalej viac používaným softwarovým inžinierstvom. Perfektnou ukážkou je Boeing a ich jednotka Air Data Reference Function pre model 777X. Táto nesmierne dôležitá jednotka, ktorá spracováva signály z niekoľko sond a snímačov, ktorá následne počíta parametri ako je rýchlosť letu alebo nadmorská výška bola celá vyvinutá pomocou model based designu. Vďaka model based designu a jeho včasného integrovania testovacieho modelu v simulátoru pomohlo overiť a opraviť rozhranie a chovanie systému dlho predtým ako by to bolo konvenčným spôsobom vývoja. Postupom času a rýchlym vývojom model based design sa vytvárajú centralizované informácie o modeloch, ktoré sa nazývajú „jediný zdroj pravdy“. Takýto model podporuje celý životný cyklus. Vývoj jednotiek ako tejto pre Boeing posúva model based design

nesmierne dopredu, a model based design sa stáva čím ďalej tým väčšou samozrejmosťou pre kriticko-náročne aplikácie. [17]

V Nemecku bolo vďaka model based designu v prostredí Simulink a Matlab spoločnosťou Bombardier Transportation vytvorená riadiaca jednotka pre pohonný systém vo vlaku. V odvetvi ako železničná doprava sa nedajú často vykonávať systémové testy pokiaľ nie je kompletný celý vlak a nebude na koľajniciach. Ak sa napokon zistia nedostatky ci nedorozumenia medzi zadávateľom a dodávateľom je to mimoriadne nákladne. Inžinieri však dokázali skrátiť dodacie časy a znížiť náklady na vývoj práve používaním model based designu. Boli vytvorené modely elektrického pohonu a riadiacej jednotky práve v prostredí Simulink, v ktorom sa následne otestovali všetky požiadavky. Potom sa vygeneroval C kód pomocou Embedded Coderu. Ten sa otestoval na HIL testoch. Výsledky takéhoto prístupu boli ohromujúce. Náklady boli znížene o 45% a časová náročnosť o 35%. Implementácia kódu bola zrýchlená a taktiež prijatie takéhoto pracovného postupu bolo v spolupráci s inžiniermi z MathWorks veľmi efektívne. [18]

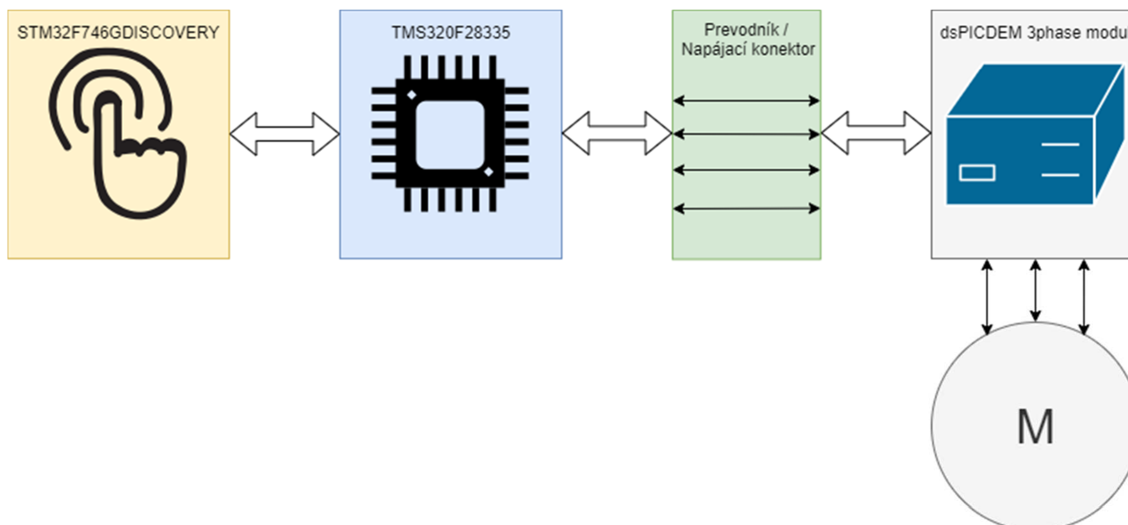
Práve spoločnosť MathWorks je pri vývoji kriticko-náročného softwaru veľmi zhovievavá a ponúka nespočetné množstvo školení a podpory.

4. NÁVRH RIEŠENIA

Pri návrhu koncepcii výsledného systému rozhodovalo niekoľko faktorov. Jeden z najdôležitejšieho faktora bola dostupnosť a univerzálnosť. V rámci výkonovej časti som hľadal vhodné robustné riešenie od firmy s bohatými skúsenosťami. Preto pri návrhu výkonovej časti, riadenia BLDC motoru som sa rozhodol pre riešenie od firmy Microchip, ktorá ponuka robustný modul pre 3 fázové motory. Ďalší z potrebných hardwarov bol vhodný mikrokontrolér pre ovládanie tohto modulu a tým aj motoru. V tomto prípade vyhral 32-bitovy mikrokontrolér od firmy TMS. Rozhodol som sa pre 32bitovy TMS320f28335 s architektúrou delfino. Mikrokontrolér je optimalizovaný pre snímanie rôznych procesov a hlavne je odporúčaný na aplikácie ako motorové drivery či do elektrických aut. Poslednou otázkou celého návrhu zostávalo užívateľské rozhranie. Rozhodoval som sa medzi klasickými vstupmi ako tlačidlá a potenciometre, ktoré by boli priamo napojené na riadiaci mikrokontrolér alebo medzi nejakým užívateľský prívetivejším a možno modernejším riešením. Nakoniec som sa rozhodol, že pre užívateľské rozhranie použijem dotykový displej. Zvíťazil vývojársky modul od firmy STM. Modul STM32F746GDISCOVERY obsahuje displej a 32bitový STM32f746 procesor, vďaka ktorému mohlo vzniknúť toto rozhranie.[19]

4.1 Architektúra riešenia

Na obrázku 4.1 je zobrazený výsledný návrh architektúry konceptu. Užívateľ si na dotykovom displeji bude jednoducho ovládať jednotlivé riadiace prvky. Taktiež na displeji budú informácie o aktuálnom stave riadiacej jednotky. Pomocou sériovej asynchrónnej komunikácie bude užívateľské rozhranie komunikovať s riadiacim mikrokontrolérom. Komunikácia bude obojsmerná. Následne podľa posielaných dát bude tento miokrokontrolér riadiť výkonovú časť. Už zostáva iba na výkonovej časti aby podľa vstupov od riadiaceho mikrokontroléru spínala tranzistory a tým riadila otáčky motoru. Výkonový modul však funguje na logike 5V. Kvôli tomu bol vytvorený jednoduchý prevodník konektoru, ktorý zároveň napája časti výkonovej logiky. Tento prevodník okrem napájania samozrejme prevádza 5V logiku na 3V a naopak.

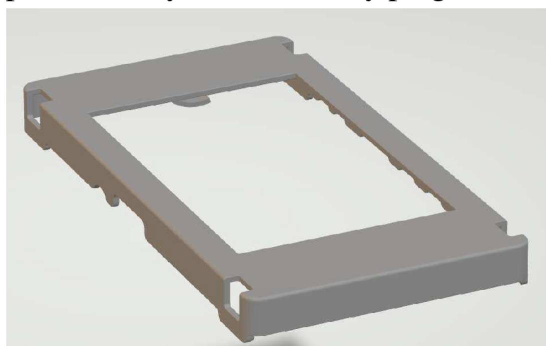


Obrázok 4.1Navrhnutá architektúra systému [27]

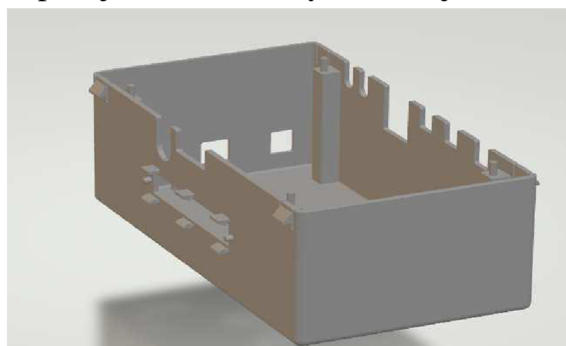
DPS prevodníka bol navrhnutý v programe Eagle. Ku jeho napájaniu je potrebných aspoň 8V. Jeho schéma a doska plošných spojov sa nachádza v prílohe. Jedná sa o jednoduché zapojenia, deliče napätí a odporúčané zapojenia jednotlivých regulátorov napätia. Troj fázový modul je možno napájať širokým rozsahom do 48V. Vývojové kity, či STM alebo TMS sú napájanie 5V cez USB.

4.2 Box

Pri celkovom riešení som sa rozhodol vytvoriť krabičku do ktorej by sa dali vhodne vložiť obe vývojove dosky. Krabička bola nakreslená v softwaru NX11 od Siemens. Je to profesionálny konštruktérsky program, ktorý disponuje veľmi kvalitnými nástrojmi.



Obrázok 4.3 Vrchná časť boxu [27]



Obrázok 4.2 Spodná časť boxu [27]

Navrhnutý box bol následne vytlačený 3D tlačiarňou Průša. Vo vrchnej časti boxu je napasovaný displej ktorý je prepojený s riadiacim mikrokontrolérom.

5. UŽÍVATEĽSKÉ ROZHRANIE

Zvolená vývojová doska STM32F746GDISCOVERY disponuje veľkým množstvom vstupno/výstupných periférii a podporou rôznych senzorov. Je to vývojárska doska, na ktorej je možné otestovať takmer čokoľvek. Zvolil som ho, pre jeho dostatočne veľký dotykový displej, vysoký výkon a aj kvôli tomu že sa momentálne nachádzal v skladoch firmy UNIS, ktorý mi ho ponúkla. Na doske je osadený STM32F746NGH6 ktorého základ je 32bitovy ARM procesor.[20]

Mikrokontrolér disponuje pamäťami:

- 1Mbyte flash pamäťou
- 340Kyte RAM
- Externá 128Mbitova SDRAM

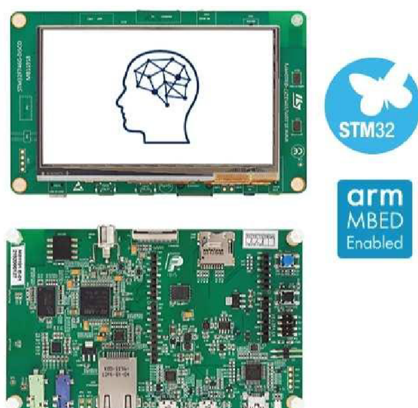
Podporuje rôzne interface ako:

- UART/USART
- I2C
- SPI
- CAN

Pre tento projekt najdôležitejšie:

- 4.3 palcový RGB dotykový displej s rozlíšením 480x272
- Debugovacím módom

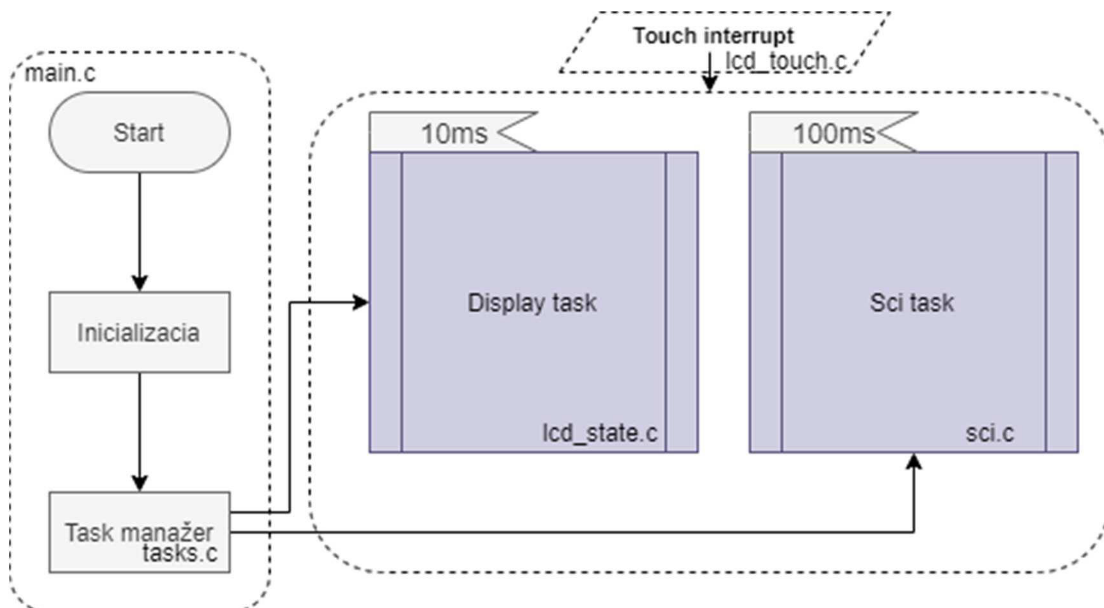
Ako je vidieť mikrokontrolér bude úplne stačiť požiadavkám na užívateľské rozhranie. Vďaka veľkým pamätiam a podporám rôznych komunikácii je tento mikrokontrolér s displejom vhodný. Pri programovaní a následnom debugovaní je využívaný debugger ST-LINK/V2-1, na ktorý sa jednoducho dá pripojiť pomocou USB mini konektoru.[20]



Obrázok 5.1 STM32F746GDISCOVERY kit [20]

Ak by som vybral ako riadiacu jednotku z platforiem ARDUINO tak tento vývojársky kit disponuje priamo konektorom, pomocou ktorého je možné sa priamo na túto platformu pripojiť. Pri programovaní bolo použité vývojárske prostredie, ktoré je priamo určené na programovanie procesorov STM a to Atollic true studio. Všetky tieto vývojárske nástroje sú v súčasnej dobe veľmi podobné. Jednoducho sa dajú nastaviť projekty, debugovať, nastavovať breakpointy či merať rôzne časy. Existuje viacero prístupov k naprogramovaniu tohto procesoru. Jedným z riešení je použiť nástroj STM32CubeIDE. Pomocou tohto grafického nástroju si užívateľ dokáže nastaviť všetky periférie, časovače či komunikačné protokoly. Následne sa vygeneruje kód. Takto vygenerovaný kód používa HAL drivery a však je podstatné dodať, že tento vygenerovaný kód nie je veľmi priehľadný. Hlavná výhoda spočíva vo veľkom množstve návodov a jednoduchosti. Druhý zo spôsobov by bol v návrhu vlastných driverov, nalinkovania registrov a následným programovaním. Takýto proces pri ktorom by sa dokázali vytvoriť overené a funkčné drivery môže trvať roky. Preto som sa rozhodol pri tejto práci pre tretiu možnosť a to použitím originál driverov od STM – HAL driverov, pričom ale všetky periférie, inicializácie a celý program bude napísaný ručne. STM ponúka aj aplikáciu na displej. Aplikácia TouchGFX však nebola použitá, ale boli použité iba jednotlivé drivery na displej.

5.1 Popis softwaru



Obrázok 5.2 Popis SW pre displej interface [27]

Celý algoritmus pracuje nasledovne. Pri resete procesora či štarte procesora sa v maine volajú všetky inicializačné funkcie, ktoré potrebujeme pre správne pracovanie mikrokontroléru.

Inicializácia:

- Displeju
- HAL driverov
- Systémové a periférne clocky – procesor beží na 210MHz
- Taskov (úloh), AD prevodníka a sériovej komunikácie

Následne sa v nekonečnej slučke volajú úlohy (ďalej ako task). Celkovo sa v programe nachádzajú dva tasky a to displej a sci task. Displej task sa stará o celkový interface. Task sci periodicky zasiela potrebné informácie zadané a požadované užívateľom, ktoré nastavil počas displej tasku. Tieto informácie sú zasielané do riadiaceho mikrokontroléru po sériovej linke.

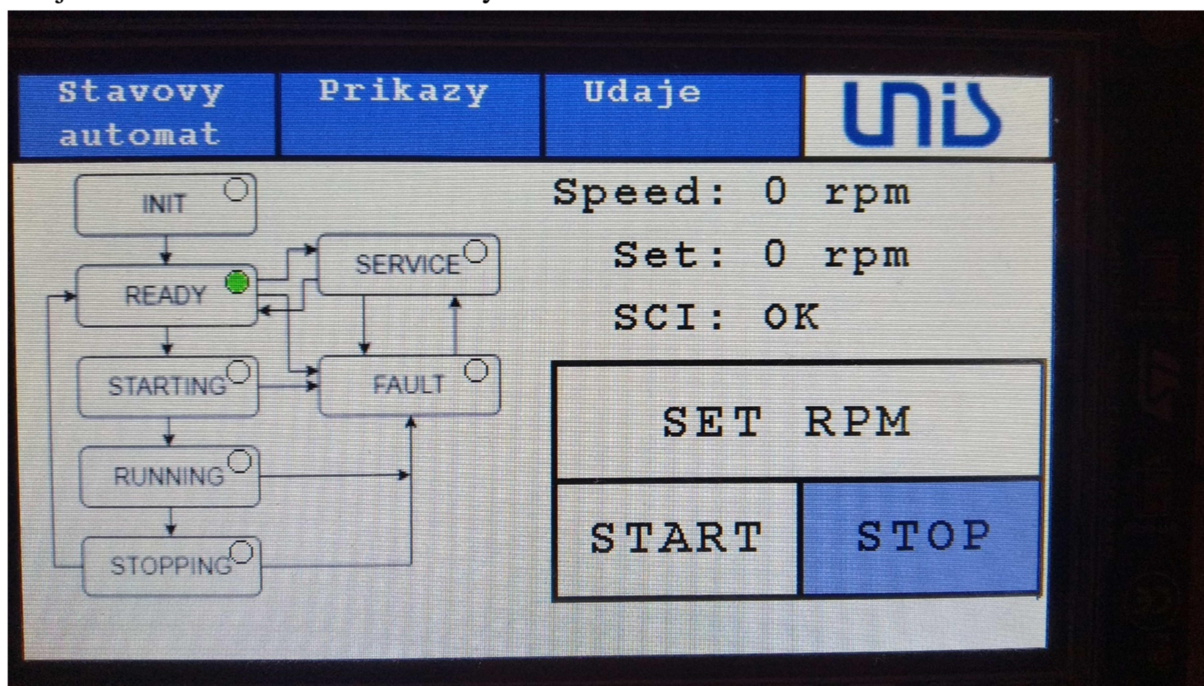
5.1.1 Displej task

Displej task pracuje s najhlavnejšou informáciou a to súradnicami X a Y, ktoré odpovedajú poslednému dotyku užívateľa. Pri každom dotyku sa aktivuje externé prerušenie, ktorého handler sa nachádza v zdrojovom súboru lcd_touch.c. Následne v samotnom displej tasku sa podľa týchto súradníc rozhoduje v switchy, čo sa bude diať ďalej. Tento switch naviguje užívateľa v menu. Na displeji sa nachádza v hornej lište jednoduché menu. Dotyk na toto menu sa kontroluje vždy aby mohlo dôjsť k prechodu do iného podmenu. Hlavné menu obsahuje podmenu ako: Automat, Príkazy, Údaje. Hlavná časť displeju, ktorá zobrazuje jednotlivé možnosti jednotlivých podmenu je vždy rozdelená na polovicu. V ľavej polovici sa nachádza pravé vybrane menu a jeho možnosti. Na pravej polovici sú vždy vypísane aktuálne otáčky, nastavené otáčky, chyby a ich bity. V pravej časti je taktiež možnosť START a STOP. Pri stlačení jedného z nich sa nastaví jeden z údajov, ktorý je následne posielaný riadiacemu procesoru. V pravej časti sa taktiež nachádza tlačidlo pre nastavovanie otáčok. Pri stlačení tohto tlačidla sa dostaneme do stavu, ktorý zaberá cely displej. Tam sa nachádza jednoduché šipkové ovládanie pomocou, ktorého je možno nastaviť otáčky. Po stlačení OK sa otáčky nastavia.

Prvé podmenu skrýva grafické zobrazenie stavového automatu, ktorý bol navrhnutý v rámci riadenia v Simulinku. Informáciu o aktuálnom stavu posiela riadiaci procesor, a podľa aktuálneho stavu sa rozsvieti zelený kruh v tom stave kde sa práve nachádza. Takýmto spôsobom si jednoducho obsluha dokáže zistiť čo sa stane, deje alebo stalo. To umožňuje ľahšiu orientáciu a pripadne hľadanie jednotlivých chýb v programu pri testovaní.

Druhé menu obsahuje príkazy. Príkazy môžu byt rôzne, závisí od daných požiadaviek. Ja som sa rozhodol pre príkaz pre prechod do servisného módu, zopnutie jednotlivých tranzistorov a zadanie smeru rotácie motoru. Aby nedošlo k rýchlim zmenám pri dotyku, sú jednotlivé dotyky týchto príkazov obmedzene na časový interval a to momentálne 200ms na reakciu. To znamená že nedochádza k rýchlej aktivácii, následnej deaktivácii a zase aktivácii pri dlhšom podržaní užívateľa.

Posledné podmenu sú údaje. Údaje obsahujú napätie a prúd posielane od riadiaceho procesoru, teplotu procesoru displeja. Ak by to bolo potrebné a pri obsiahlejších projektoch by sa do tohto menu dali pridať rôzne vstupy a výstupy či hodnoty od snímačov, čím by užívateľ jednoducho videl a mohol overiť stavy a veličiny. Na nasledujúcom obrázku 5.3 sa nachádza výsledné užívateľské rozhranie.



Obrázok 5.3 Uživateľské rozhranie [27]

5.1.2 Zhrnutie

V tomto prípade mikrokontrolér bohato vystačil na svoju úlohu. Ak by celá vývojová platforma nebola osadená neskutočným množstvom periférii a mala vyvedených viacero výstupných pinov dalo by sa to využiť aj na riadenie BLDC motora.

6. RIADENIE A BLDC

Zvolená riadiaca jednotka je použitý mikrokontrolér od firmy TMS a to TMS320f28335 s architektúrou delfino. Tento vývojársky modul bol použitý spolu s dokovacou stanicou. Mikrokontrolér na doske nemá pripojené žiadne periférie, preto je tu možnosť použiť takmer všetky jeho piny. Pre jeho úlohu v tomto systéme mikrokontrolér disponuje :

- 6 PWM kanálmi
- UART, I2C, SPI, USART, CAN
- 12bitovým AD prevodníkom

Prevodník má široké spektrum možného nastavenia to iste platí aj pre PWM generovanie. V tomto prípade sa jedná o špičku na trhu. Mikrokontrolér komunikuje po sériovej linke s užívateľským rozhraním. Na druhej strane je pripojený k výkonovej časti k modulu od Microchipu. [21]

Obrázok 6.1 TMS320 s dokovacou stanicou [21]

6.1 BLDC a jeho riadenie

Motor je veľmi podobný normálnemu trojfázovému asynchrónnemu statoru alebo synchrónnemu stroju. Hlavný rozdiel medzi BLDC a klasickým motorom je to, že jeho vinutia sa nachádzajú na statore. Rotor je teda potom tvorený z permanentných magnetov, ktoré sú vyrobene zo vzácnych kovov. Tieto motory sú typické svojou dlhou životnosťou a spoľahlivosťou, pričom môžu dosahovať vysoké otáčky aj pri malom momente. Je to spôsobene konštrukciou motora pričom sa podarilo odstrániť uhlíkové kefky. [22]

Princíp správnej činnosti spočíva v napájaní jednotlivých fázových vinutí motora. V každom okamžiku môžu byt spínane naraz iba dve fázy statorového vinutia. Takto

spínané fázy tým pádom tvoria požadovaný vektor statorového magnetického toku. Pri komutácii by mala byť splnená podmienka kolmosti vektora magnetického poľa rotora a statora. Jedna fáza je budená kladným napätím a druha fáza záporným napätím. Pre správny beh je taktiež dôležité aby sa magnetické pole vygenerovane vinutím posúvalo a postupne doháňalo magnetické pole zo statoru. Pre správne fungovanie a udržanie otáčok a prepínanie jednotlivých budičov je nutne vedieť aktuálnu polohu motoru v týchto momentoch. Preto existujú dve základne metódy riadenia motoru a to je bez senzorové riadenie a senzorové riadenie. Senzorové riadenie ma základne sektory, ktoré sú definovane pomocou zvyčajne troch halových senzorov. Výstupy snímačov sú vyvedené do riadiacej jednotky a pomocou rôznych kombinácii sú následne jednotlivé tranzistory spínane. Celkovo vďaka snímačom vznikajú 6 kombinácie a vždy je aktivovaná iba jedna. Toto riadenie je jednoduchšie a všeobecne platí, že stačí vedieť čítať výstupy jednotlivých sond a podľa kombinácie prepínať polohy v diagrame. V tejto práci sa však zameriavam na bez senzorové riadenie, ktoré je vysvetlene v ďalších kapitolách.[22]

6.2 Zhodnotenie BLDC motoru

Výhody sú :

- riadenie otáčok vstupným napätím
- veľké aplikačne možnosti
- da sa použiť aj ako krokový motor
- vysoká účinnosť a rýchlosť
- malé rozmery a nízka hlučnosť ci hmotnosť a vysoká životnosť
- užitočný najmä v priemyslových aplikáciách

Nevýhody

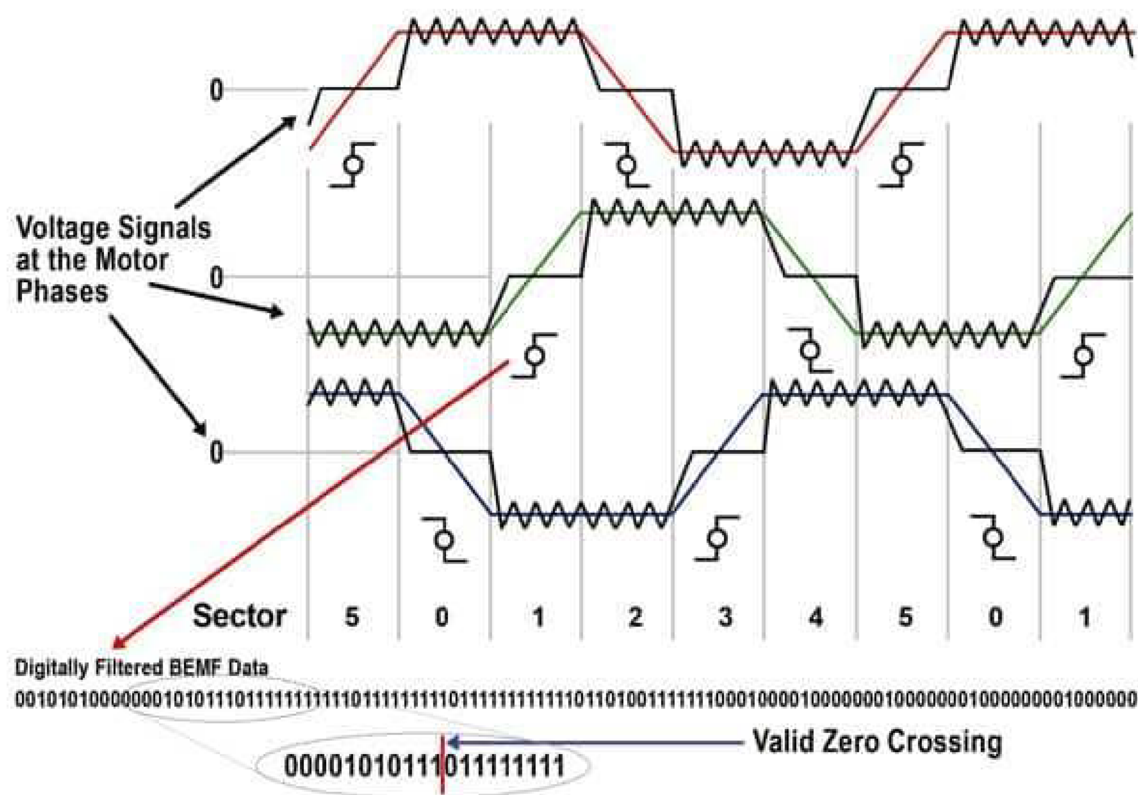
- riadiaca logika
- náchylný na teploty
- cena

BLDC motor sa stále viac a viac stáva populárnejším a preto predpokladám rozšírenie týchto motorov aj do zariadení v domácnostiach.

6.3 Bez senzorové riadenie

Pri bez senzorovom riadení motora nemáme jednoduchú možnosť riadenia, ktoré poskytujú spätné signály z halových sond. Preto pri riadení používame takzvanú spätnú elektromotorickú silu - BEMF. V známejšom slove takzvaný prechod nulou sa všeobecne používa na synchronizáciu komutácie jednotlivých fáz, ktoré sú od seba posunuté o 120 stupňov. Pričom v každej fáze je tento prechod uskutočnený dvakrát. Detekovanie tohto signálu prebieha na voľnej fáze, pretože počas kroku sú elektricky napájane dve fázy vinutia. Jednotlivé prechody nulou sú zobrazene na obrázku 6.2. Riešenie v rámci modulu

od Microchipu dovoľuje prechod nulou detekovať viacerými spôsobmi. Jeden zo spôsobov je realizovaný pomocou niekoľko operačných zosilňovačov, ktoré sú zapojene ako komparátory. Pri napätí na fáze, ktorá je výšia ako polovica napájacieho napätia, sa komparátor aktivuje a tým sa detekuje prechod nulou. A však v tejto práci bola využitá druhá voľba, a to spätnou väzbou napätí jednotlivých fáz. Toto napätie je vyfiltrované a potom privedené cez prevodník, kde sú zase vyfiltrované na prevodník mikrokontroléru. Pri tejto detekcii vzniká jeden podstatný problém. Problém nastáva v tom, že aby bolo možné detekovať prechody nulami, je potrebné dosiahnuť minimálnu rýchlosť motoru, aby bolo BEMF napätie dostatočne veľké.[24]



Obrázok 6.2 Diagram fáz a prechodov nulami [23]

Na obrázku 6.2 vidíme fázy U,V,W a to v poradí ako červená, zelená a modrá. Celkový počet krokov je šesť a tie sa periodický opakujú. Z obrázku je jasné, že prechody nulou vznikajú vždy na voľnej fáze, pričom jedna fáza je napájaná kladným napätím a druhá záporným.

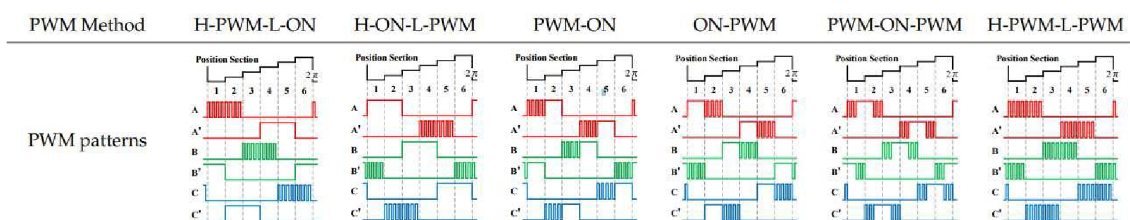
6.4 PWM

Najčastejšou konfiguráciu pre trojfázové motory v oblasti výkonovej časti je zapojený mostík so šiestimi tranzistormi. Kolektory horných tranzistorov sú spojené na pozitívny potenciál, na napájacie napätia. Emitory dolných tranzistorov sú spojené do kopy a sú pripojené k zemi. Potom sú na motor pripojené tri výstupy, ktoré vznikli medzi hornými

a dolnými tranzistormi. Pri použití veľkých, vysoko napäťových motorov sa používajú IGBT tranzistory. Však oveľa častejšie pri motoroch do 100V sú efektívnejšie a používanejšie MOSFET tranzistory.

Nastavovanie konečných otáčok motora je realizovane pomocou pulznej šírkovej modulácie PWM, ktorej strieda je regulovaná. Pomocou generovaných PWM signálov sa spínajú jednotlivé tranzistory mostíka zapojeného vo výkonovej časti modulu. Spínanie tranzistorov môže byt rôzne. Dá sa povedať, že rozlišujeme 3 hlavné druhy: unipolárne, bipolárne a komplementárne spínanie.

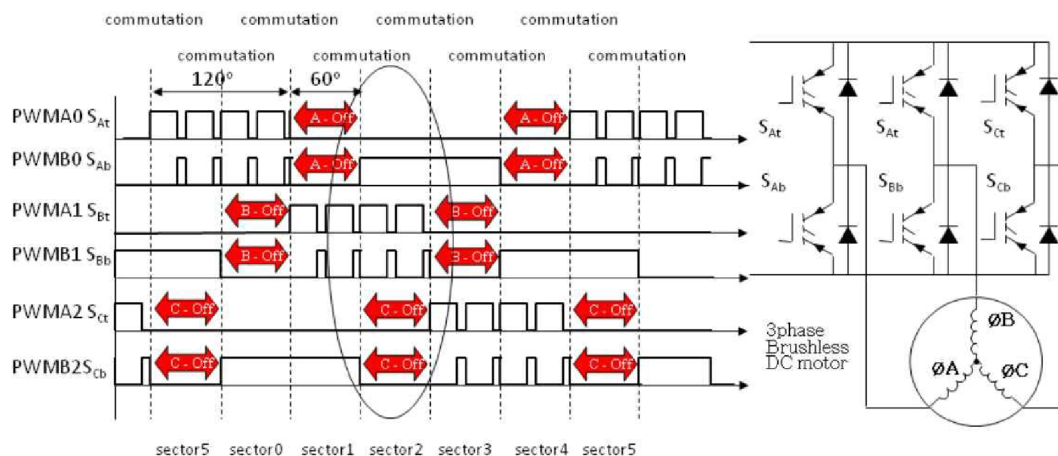
Unipolárne spínanie spočíva v spínaní jednotlivých vinutí iba jednou polaritou napätia. Tranzistory sa spínajú postupne, pričom na každej fáze je vždy spínaný iba jeden. Na prvý horný tranzistor sa privedie napätie čím sa zopne a taktiež sa privedie napätie na dolný tranzistor druhej fázy. Ďalším krokom sa privedie napätie na dolný tranzistor tretej fázy, potom sa zopne horný tranzistor druhej fázy a tento vzor pokračuje a opakuje sa. Výhodou u unipolárného prepínania PWM je lepšia elektromagnetická kompatibilita. Existujú všemožné spôsoby spínania tranzistorov ako je vidieť na nasledujúcom obrázku.[22]



Obrázok 6.3 Možné spínanie MOSFETov pri unipolárnom riadení [25]

Bipolárne spínanie je kríženie tranzistorov. Najprv sa privádza napätia na horný tranzistor jednej fázy a na dolný druhej fázy. V ďalšom kroku sú tranzistory vymenené. To znamená, že je zopnutý tranzistor fáze, ktorej bol naposledy zopnutý horný tranzistor a to platí opačne pre druhú fázu. Toto riešenie nie je veľmi populárne, no však je lepšie pre snímanie polohy rotora bez snímača. [22]

Komplementárne spínanie je spínanie, pri ktorom je použitá komplementárna PWM. To znamená, že pri každom jednom stave je pre práve ovládanú fázu spínaný horný tranzistor a dolný taktiež ale opačnou PWM. Tento spôsob môže byť aj bipolárny alebo unipolárny. V tejto práci je použité unipolárne komplementárne spínanie.

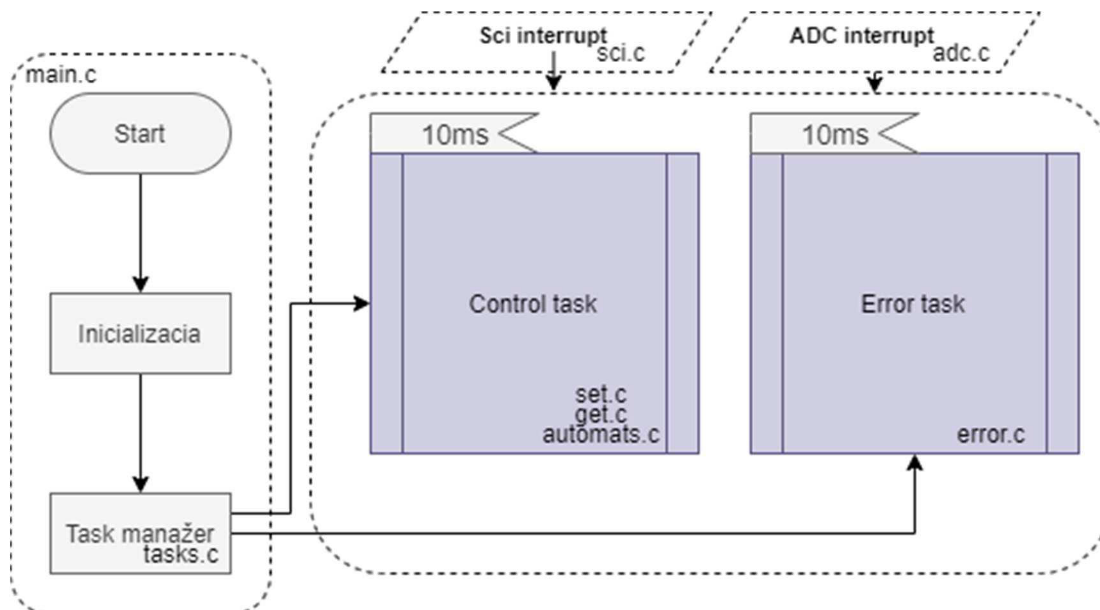


Obrázok 6.4 Unipolárne komplementárne riadenie MOSFETov [26]

Vďaka takémuto riadeniu sa motor dokáže dostať do štvrtého kvadrantu, kde brzdí smerom dopredu. Taktiež sa spínané MOSFETy až tak teplotne nezaťažujú. Celkovo je tento spôsob náročnejší. Mohla by nastať situácia, pri ktorej sú zopnuté obidva tranzistory v rámci jednej fázy. Je to spôsobene dobami spínania a rozopnutia tranzistora a inými prechodovými javmi. Aby sme sa takejto situácii vyvarovali zavádzame určitú pauzu po ktorej je zopnutý druhy tranzistor. To znamená, že hrana, ktorá spína tranzistor je oneskorená. Tento čas sa nazýva dead time. [26]

6.5 TMS riadiaci program

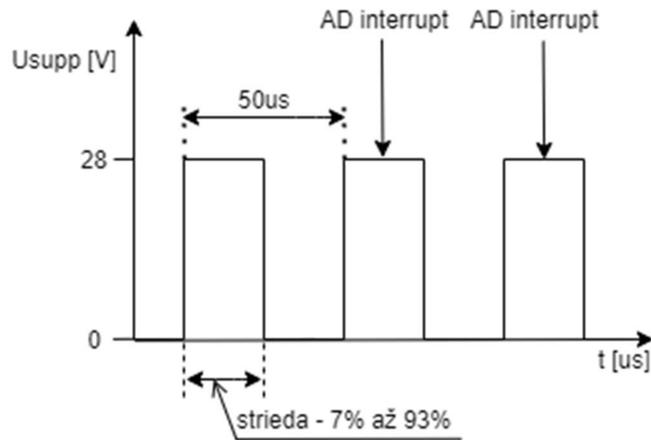
Štruktúra programu je zobrazená na obrázku 6.5. Podobne ako u užívateľského rozhrania sa v kóde nachádza task manager, ktorý v určitých intervaloch vola jednotlivé tasky. Tasky vytvorené pre tuto aplikáciu boli error task a task vygenerovaného kódu riadenia v Simulinku. Oba tasky bežia na 10ms. Po inicializácii periférii a hodín sa v nekonečnej slučke volajú tieto tasky. Pričom v tejto slučke sa taktiež kontroluje flag, ktorý sa aktivuje pri príjme dát po sériovej linke. Ak je tento flag aktivovaný, mikrokontrolér odpovedá po sériovej linke naspäť. Detailnejší popis je v ďalšej kapitole.



Obrázok 6.5 Popis SW riadiacej jednotky [27]

V error tasku sa nastavujú bity chýb. Prebieha kontrola maximálneho prúdu, ktorý je odoberaný zo zdroja. Taktiež je možne nastavovanie errorov jednotlivých napätí fáz. Jedna z ďalších chýb je chyba napájacieho napätia, ktoré by nemalo byť nad alebo pod určený limit. Chyba sériovej komunikácie je ak nesedí CRC. Posledné dve chyby sa nastavujú pri požiadavkách na štart pri behu motoru. Prvá štart chyba sa nastavuje v štartovacej sekvencii popísanej nižšie. Druhá run chyba je nastavená ak pri spustenom motoru sa jednotlivé prechody nulou prestanú detekovať.

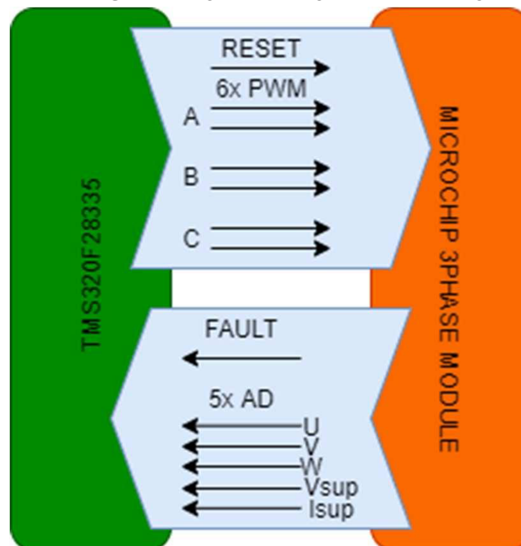
Procesor beží na 150MHz, pri riadení je používaný 12bitový AD prevodník a jeho PWM periférie. PWM beží na 20kHz. Vzorkovanie jednotlivých AD vstupov je spojené s generovaným PWM signálom a preto beží na 20kHz pričom sa vždy v strede jednotlivých pulzoch merajú vstupy. Cele časovanie je zobrazene na nasledujúcom obrázku 6.6. Maximálna strieda môže byť nastavená na hodnotu 3500 čo odpovedá 93% striede. Dead time je nastavený na približne 3us. V každom kanály sú použité komplementárne PWM signály. Pri nastavovaní hodnôt na logické úrovne, bez modulácie je použitý trip zone modul.



Obrázok 6.6 PWM a časovanie [27]

6.5.1 Pripojenie k výkonovému modulu

Mikrokontrolér je k modulu pripojený pomocou prevodníka, ale pre zjednodušenie sa na obrázku nenachádza. Ako je vidieť k riadeniu stačí posielať signály na spínanie tranzistorov a ešte jeden signál na zresetovanie modulu. Pri každom štarte jednotky by sa mal predtým vyslať reset pomocou ktorého sa modul inicializuje taktiež. Modul obsahuje veľké množstvo vnútorných ochrán, a pri nad prúdu, rozpojení či prekročení teploty sám odpojí fázy od napájania. Ak nejaká z týchto chýb nastane vyšle signál fault.



Obrázok 6.7 Zapojenie k výkonovému modulu [27]

Na riadiacu jednotku sú taktiež pripojené výstupné napätia, cez odporové deliče. Jedná sa o napätia na fázach, celkové napätie a napätie, ktoré odpovedá odoberanému prúdu.

6.5.2 Štartovanie motoru

Štartovanie BLDC motoru je zväčša riešené takzvanou štartovacou rampou, ktorá je následne upravená podľa správania BLDC motoru pri prvých pokusov o štart. Avšak existujú aj iné spôsoby štartovania. V práci som sa pokúsil o univerzálnejší spôsob, ktorý nie je až tak odlišný od takejto štartovacej rampy. Avšak tento jeden spôsob štartu funguje v rámci širokého spektra BLDC motorov, pričom parametrov, ktoré sa musia meniť je minimálne množstvo.

Pri štartovacej sekvencii sa začne volať funkcia, ktorá je volaná v každej AD interruptovej rutine. Táto funkcia potom zopína jednotlivé tranzistory podľa daného vzoru. Pri každom stavu sa postupne zväčšuje strieda PWM signálu a čaká sa na minimálnu zmenu napätia, ktorá je v tomto prípade braná ako nárast prúdu. Ak dôjde k minimálnej zmene tak sa inkriminuje krok. K inkrementácii kroku dôjde aj po prekročení určitého časového úseku. Avšak ak ku prechode dochádza iba po tomto časovom úseku znamená to, že nárast prúdu nenastáva. Preto sa spínanie tranzistorov vypne a nastaví sa start chyba. Tento stav sa berie ako stav, pri ktorom sa motor nedokázal naštartovať. Taktiež po zavolaní tejto funkcie sa v interruptovej rutine kontroluje prechod nulou na jednej fáze. Po detekovaná určitého množstvu prechodov sa motor prepne do ďalšej fázy a to running.

6.5.3 Beh motoru - running

Ak sa podarilo naštartovať motor tak stav je prepnutý do módu running. Funkcia riadenia je taktiež volaná v AD prevodníkovej rutine. Tato funkcia inkrementuje stavy tranzistorov už priamo pomocou detekcie prechodov nulou. Napätia na všetkých troch fázach sú merané. Potom sa vyhodnocuje to, že ak napätie na voľnej fáze je vyššie ako priemer dvoch napätí na druhých dvoch fázach tak dochádza k prechode nulou. V algoritme získavame čas od posledného prechodu nulou. Aby bola následná komutácia v správnom čase je potrebné tento čas medzi prechodmi nulami vydeliť dvomi. Tým sa ďalšia komutácia oneskorí o 30 stupňov a takýto čas sa uloží do periódy časovača 2. Tento časovač po čase vyvoláva prerušenie, v ktorom sa komutuje ďalšia fáza. Takýmto spôsobom je už ďalej periodicky vykonávane riadene BLDC motoru. Striedu nastavuje regulátor vytvorený v Simulinku, ktorý sa volá v control tasku. Ak ku prechodom nedôjde v určitom časovom úseku, je motor zastavený a je vyhlásená run chyba.

6.5.4 Meranie otáčok

Otáčky sa kvôli nepresnostiam merajú iba pri každej celej mechanickej otáčke motoru a nie pri každej komutácii fázy. Ak došlo k celej otáčke je perióda časovača, ktorá odpovedá času od minulej otáčky vydelená a vynásobená konštantami, ktoré závisia na nastavení časovača a počte pólov motoru. Hodnota je filtrovaná jednoduchým filtrom. Jedná sa o funkciu control_speed_calc.

6.6 Komunikácia - SCI

Mikrokontroléry TMS a STM spolu komunikujú po sériovej linke.
Nastavenie je nasledujúce :

- Baud: 115200
- Stop bit: 1
- Start bit: 1
- Parity: Bez parity

Rámec, ktorý posiela TMS je nasledovný :

Tabuľka 6.1 Rámec posielaný riadiacou jednotkou

RPM horný byte	RPM dolný byte	Napájacie napätie horný byte	Napájacie napätie dolný byte	Prúd horný byte	Prúd dolný byte	Error bitfield horný byte	Error bitfield dolný byte	Stav automatu	CRC

Rámec, ktorý posiela a prijíma STM je nasledovný:

Tabuľka 6.2 Rámec posielaný užívateľským rozhraním

Pož. RPM horný byte	Pož. RPM dolný byte	Príkazy horný byte	Príkazy dolný byte	Prúd horný byte	CRC

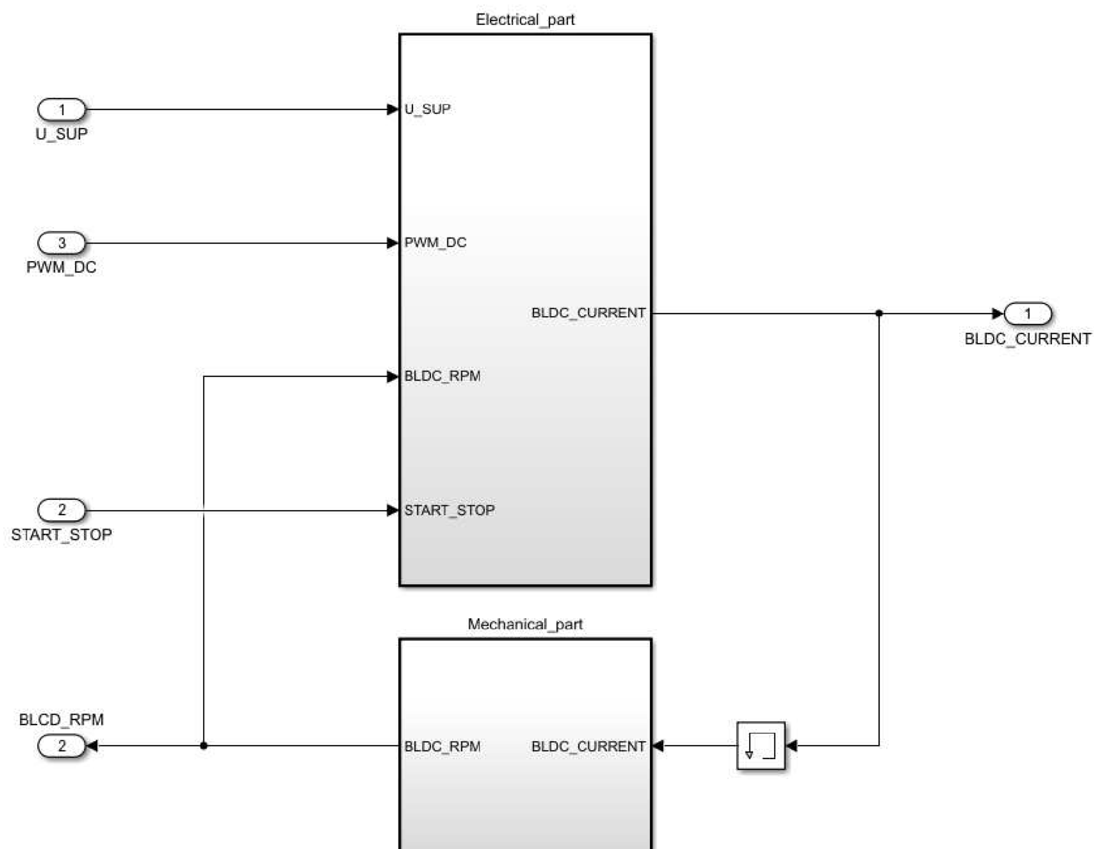
STM procesor vždy posiela dáta ako prvý a TMS čaká na ich prijatie. Po prijatí všetkých bytov, ktoré sú nastavene pomocou SCI registrov sa vyvolá prerušenie a dáta sú spracované. Následne TMS odošle odpoveď procesoru STM. STM túto odpoveď spracuje v prerušení. Prijatie a odosielanie sa kontrolujú pomocou driverov.

7. SIMULINK

Logika, algoritmus riadenia bol vytváraný v známom prostredí Matlab/Simulink. V tomto prostredí je navrhnutý stavový automat a to v rozšírení StateFlow. Pričom v Simulinku je taktiež navrhnutý jednoduchý regulátor otáčok. Toto vývojove prostredie bolo vybrané hlavne z prostého dôvodu, a to že momentálne sa na trhu nenachádza nič lepšie a dá sa povedať, že konkurencia sa ešte nedostala do takej všestrannosti pričom sa to tak skoro nestane. Pri návrhu regulátora sa pracovalo iba s zjednodušeným modelom motoru.

7.1 Model motoru

Model motoru je zjednodušený a preto stačí na ovládanie jednoduché napätie, ktoré simuluje našu PWM striedu. Na výstupe modelu dostávame výstupnú rýchlosť a prúd.



Obrázok 7.1 Model motoru [27]

Model obsahuje elektrickú a mechanickú časť. V elektrickej časti sa pomocou parametrov ako napájacie napätie, vstupy - jednosmerne napätie a aktuálne otáčky

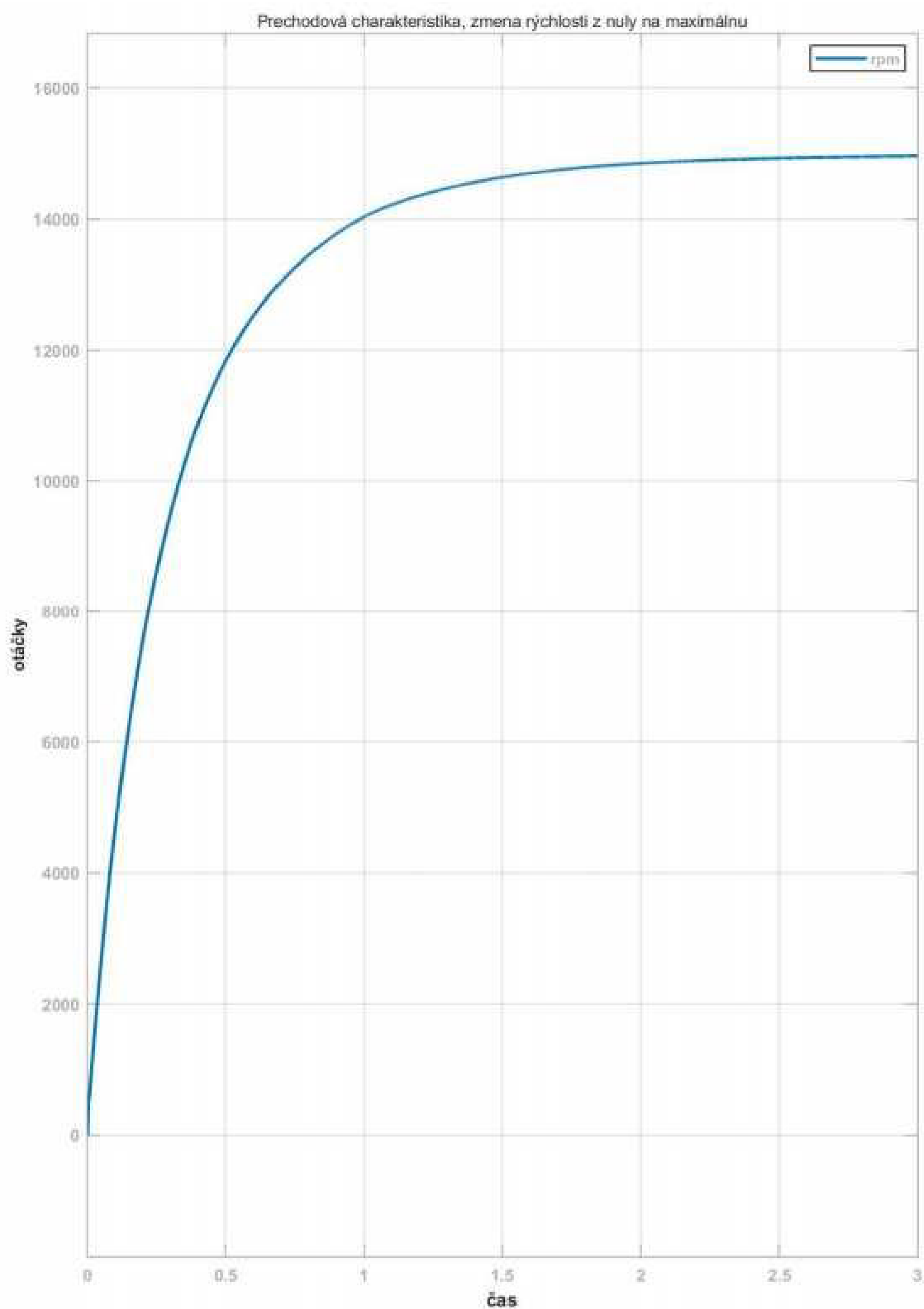
z mechanickej časti a pomocou nameraných reálnych konštánt R, L a Ke vypočíta aktuálny prúd. Tento prúd následne vstupuje do mechanického bloku, v ktorom sa prúd prepočítava na otáčky. Zdôrazním, že v prvých fázach pokusov došlo aj k simulovaniu prepracovanejšieho reálneho modelu BLDC motoru kde sa spínali jednotlivé tranzistory priamo v Simulinku. Avšak tieto simulácie trvali aj niekoľko hodín, pričom výsledok bol par sekundový výstup. Preto došlo k zvoleniu tohto zjednodušeného modelu.

Parametre motoru:

- L = 13e-6 [H]
- R = 26e-3 [Ohm]
- Ke = 1/630 [V/rpm]
- Km = 0.015 [Nm/A]
- U = 28 [V]
- Max. otáčky = 15000 [ot/s]

7.2 Regulátor

Navrhnutý PI regulátor v Simulinku, sa nachádza v priloženej prílohe. Do regulátoru vstupujú aktuálne otáčky motoru a požadované otáčky. Ďalej k riadeniu som pridal požiadavku na zapnutie regulátoru a vypnutie zo stavového automatu. Jedna sa o jednoduchý PI regulátor, ktorého výstup je obmedzený obmedzovacom o maximálnu zmenu 3% striedy. Regulátor bol ladený pomocou simulácii. Charakteristiku pri štarte až po maximálne otáčky môžeme vidieť na obrázku 7.2. Výsledná proporcionálna zložka je 0.0012 a integračná zložka je 0.00045. Na maximálne otáčky sa dostane približne za 2,5 sekundy. Avšak regulátor bude fungovať až keď je motor rozbehnutý, preto pri normálnych zmenách by obmedzovač nemal zasahovať.



Obrázok 7.2 Prechodová charakteristika regulátoru [27]

7.3 State flow

Najvyššia logika riadenia bola implementovaná v stavovom automate vytvoreným v StateFlow. Stateflow od Mathworks je rozšírenie, ktoré poskytuje vlastný graficky programovací jazyk. Jeho hlavným blokom sú stavy a prechody medzi nimi. Môžu byt vytvárane tabuľky prechodov alebo pravdivostné tabuľky. Takýmto spôsobom sa dajú vytvoriť rôzne manažéry úloh, užívateľské rozhrania, riadiace jednotky. Tvorbou takéhoto automatu vznikne cely blok, ktorý je následne možno debugovať a simulovať priamo v Simulinku. Tento nastroj pre modelovanie sekvenčných automatov pomocou stavových diagramov sa da využiť v prostredí Matlab/Simulink od verzii R2019a. Stateflow je plne integrovaný do Simulinku, pričom vytvorený diagram sa ukladá ako jeden model. Tento model taktiež podporuje generovanie kódu v jazyku C. Ma vlastné grafické prvky pre vytváranie diagramov a zavádza pojmy ako stav a prechod.

V Simulinku sa pre vytvorenie automatu použije z knihovne blok chart. V tomto bloku už môžeme následne zakladať nove stavy a prechody medzi nimi. Prechod medzi stavmi vytvárame jednoducho pretiahnutím myškou z hrany jedného stavu na druhý. Každému stavu pri jeho zakladaní zvolíme názov, pričom každý stav musí byť jedinečný. Pomocou Model Exploreru dokážeme cez „add -> inputs a outputs“ pridávať vstupy a výstupy do nášho automatu. Jednotlivé nadefinované dáta definujeme ako vstup, výstup, lokálna premenná či parameter alebo konštanta. Najpodstatnejšie sú pre nás eventy a dáta. Event je vstup, ktorý vyvolá udalosť a diagram sa zavolá. Eventov môže byť viacero a však vstup bude stále iba jeden, pričom sa automaticky vytvorí vektor. Eventy vo vektore sú následne jedinečne pomenované a môžu byť súčasťou podmienok prechodov medzi stavmi. Pri každom zmene eventu sa vyvolá funkcia automatu.

Data, sú premenné pomocou ktorých pracujeme v stavovom automate. Tieto premenné sa dajú klasicky porovnávať, nastavovať a dá sa rozhodovať pomocou nich. Sú to klasické vstupy do vytvoreného stavového automatu. Informácie o premenných, udalostiach, stavov, prechodov nájdeme v „view -> Property Inspector“.

Prechody medzi stavmi definujeme buď eventami alebo omnoho častejšie sú používane pravé premenné a práca s nimi. Ak chceme upraviť podmienky prechodu medzi stavom stačí jednoducho kliknúť na danú šípku prechodu a podmienku dopísať. Podmienka prechodu sa v StateFlow píše do hranatých zátvoriek. Ak sú splnene naraz viaceré prechody, tak splnený prechod sa urči podľa priority prechodu. Pre zmenu jednotlivých priorít stačí použiť kontextové menu prechodu a tam zmeniť „Execution Order“. U prechodov okrem prechodovej podmienky môžeme definovať takzvané podmienkové akcie a prechodové akcie. Podmienková akcia sa vykoná pri priechode cez prechod aj keď sa nakoniec cely prechod nevykoná. Prechodová akcia sa vykoná až po prejdení prechodom. Obecný zápis je teda nasledovný :

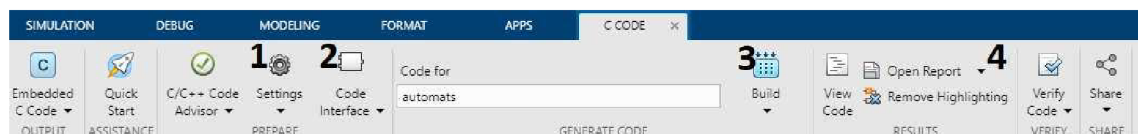
E [C] {CA}/{TA}

Pričom CA – Condition Aciton, TA – Transition Action.

Tento rozdiel medzi týmito dvomi akciami ma však zmysel iba pri použití križovatiek. Ako z názvu vyplýva, križovatky slúžia jednoducho na kríženie jednotlivých prechodov kde sa rozvetvujú alebo spájajú. Pri definovaní stavov rozšijeme tri základne akcie. A to entry, during a exit. Akcia v entry sa vykoná iba raz a to pri vstupe do stavu. During sa vykonáva pri zotrvaní v stavu a opakuje sa. Exit akcia sa vykoná pri odchádzaní zo stavu. Za každý príkaz sa píše klasicky bodkočiarka. Ak je nejaký stav zložitejší môžeme do neho bez problémov vkladať ďalšie podstavy. A však pri každom takomto vkladaní je dosť často vhodne použiť funkciu history, vďaka ktorej si stav bude pamätať v akom podstave sa nachádzal. Pri celom automate a týchto podstavoch je doležíte definovať „Default Transition", vďaka ktorej StateFlow vie v ktorom stave ma začínať. V StateFlow máme možnosť volať externe funkcie napísane v C, Simulink funkcie alebo Matlab funkcie.

7.4 Vygenerovanie kódu

Výsledný dizajn pozostáva teda zo stavového automatu a základného riadenia. Pri väčších projektov sa v riadení nachádzajú zákony pomocou, ktorých sa riadi napríklad riadiaca jednotka pre lietadlový motor, automobil, lode a iné systémy. Stavový automat dostáva potrebne vstupy od riadiaceho procesoru TMS, ktoré vyhodnocuje. Následne je jeho výstup posunutý riadeniu, kde sa v našom prípade nachádza regulátor otáčok. Celé toto riešenie môže byť otočené naopak, to znamená že by sa najprv vyhodnocovali podľa vstupných premenných jednotlivé úkony, dáta zo snímačov a potom by sa zavolal stavový automat, ktorý by pracoval až s týmito vyhodnotenými údajmi. Ak už je vytvorená takáto štruktúra programu nezostáva nič iné ako kód vygenerovať. Kód v Simulinku generujeme pomocou aplikácii Embedded Coder.

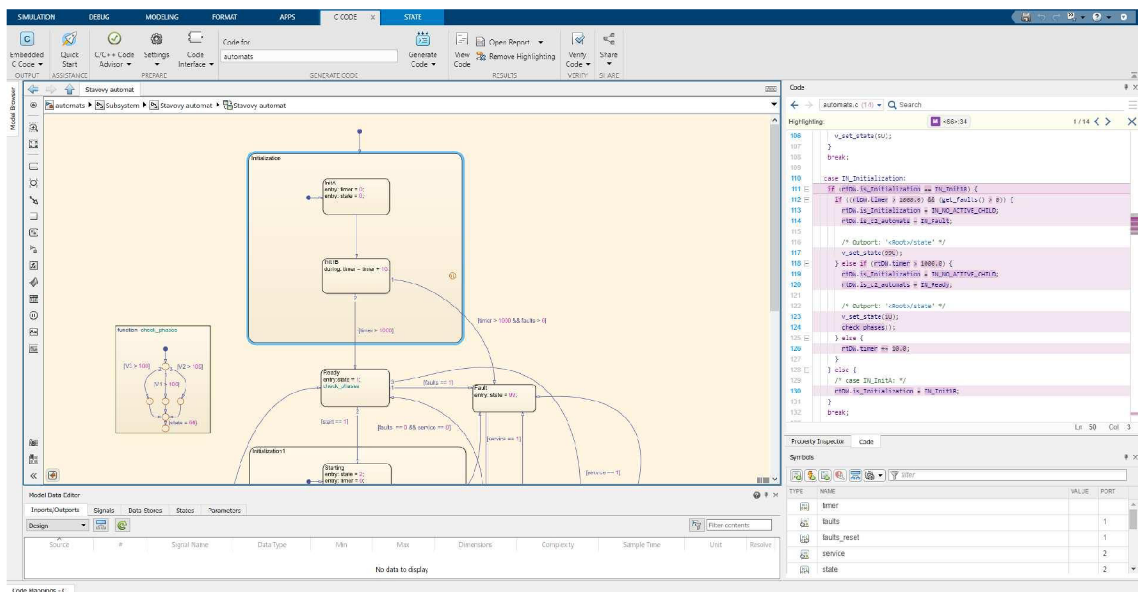


Obrázok 7.3 Rozhranie Embedded Coder, 1- nastavenie generovania, 2 – interface, 3 – generovanie/build, 4 – nastavenie reportu [27]

Ako výstup môžeme použiť C kód, C++ alebo inak nadefinovaný výstup pomocou externej knižnice. V následnom nastavení si môžeme nastaviť compiler, linker, a celkovú konfiguráciu ako pri kompilovaní kódu v iných vývojových prostrediach. Pri generovaní konečného kódu doporučujem zaškrtnúť možnosť „Package code and artifacts" a pridať do „zip file name" hodnotu „packNGo". Týmto generátor vygeneruje všetko do jedného zazipovaného súboru aj so všetkými potrebnými knižnicami. Ak to vygenerujeme bez toho je potrebné si knižnice, ktoré Simulink/Matlab používa dohľadať a následne includovať už do výsledného programu kde používame aj ručne písaný kód. Generátor disponuje s nespočetným množstvom optimalizácii, ktoré si môžeme nastaviť. Taktiež

nastavujeme veľkosti jednotlivých typov, názvy globálnych premenných, štruktúr, metód, lokálnych premenných. Ak používame a voláme v programe externe písaný kód je doležíte nalinkovať ich zdrojové súbory v kolónke „Custom Code“. Veľmi dôležitý je pre nás generátor reportu. Report môže obsahovať dostatočne obstojný počet informácii. Vygenerovaný report sa nachádza v elektronickej prílohe.

Celkovo po všetkom nastavovaní mame dve možnosti. A to kód skompilovať a vygenerovať alebo iba vygenerovať. Ako už z toho vyplýva pri vygenerovaní kódu sa kód vygeneruje no neoveruje sa. To si už musí vývojár overiť sám. Pri kompilácií kódu je na druhú stranu potrebné už mať správne nalinkované externe súbory a pripravené v adresáru generovania. Pri generovaní kódu sa automaticky spusti „Diagnostics Viewer“, v ktorom vidíme prehľad práve vykonávajúcich dejov. Výsledkom by malá byť úspešná generácia kódu a ak chceme tak výsledný report. Kód sa nám zobrazí v „Property Inspector“ a pri každom kliknutí na blok v Simulinku, premennú, parameter alebo prechod, sa zobrazí časť vygenerovaného kódu, ktorá tomu odpovedá. Všetko je spätne trasovateľné a jednoducho vyhľadateľné.



Obrázok 7.4 Stavový automat, označený blok a jeho kód sa vyznačí vo vygenerovanom kóde [27]

Navrhnuté riadenie je na nasledujúcom obrázku. Pričom:

1 – Stav Inicializácia

2 – Stav Ready

3 – Stav Starting

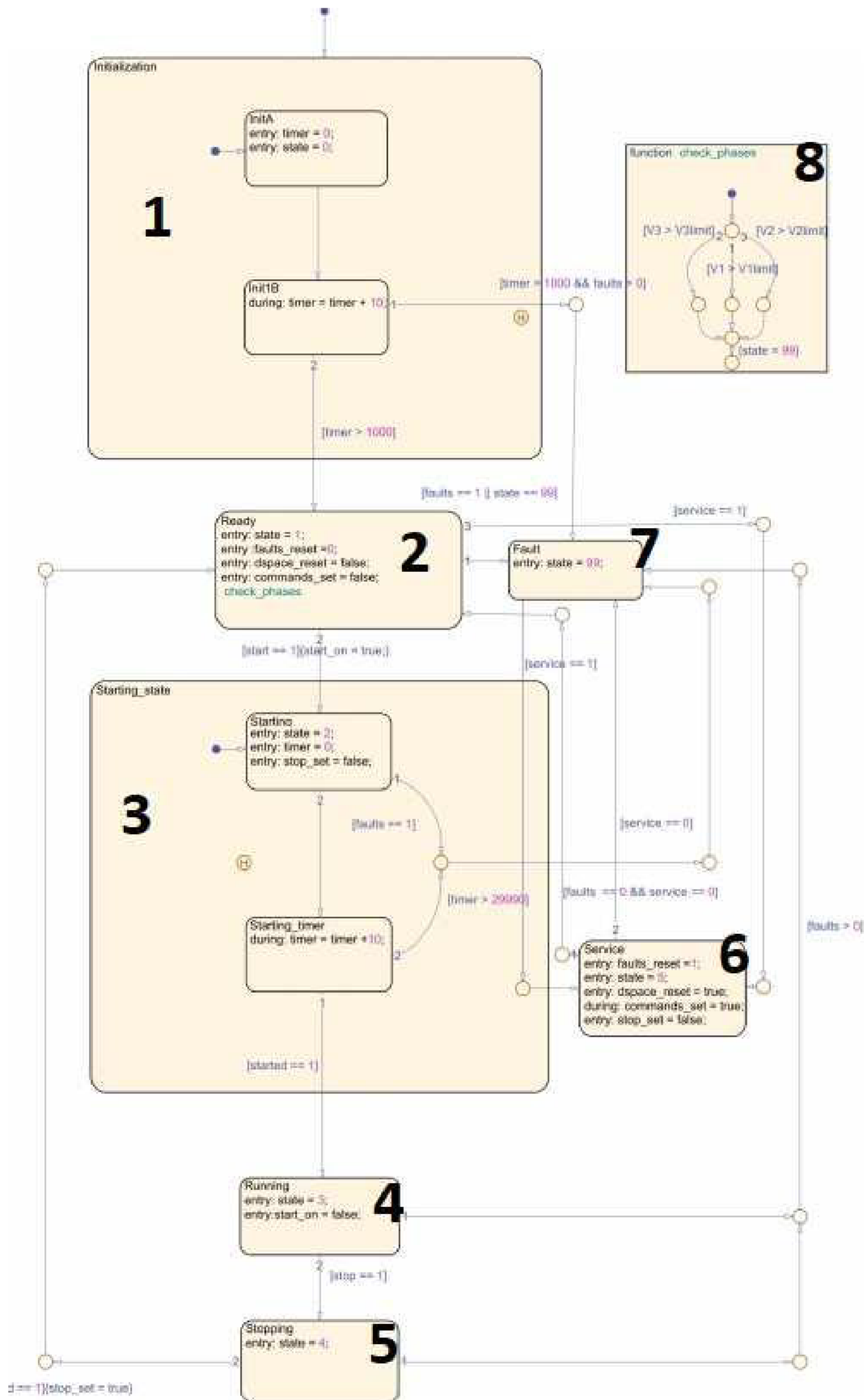
4 – Stav Running

5 – Stav Stopping

6 – Stav Service

7 – Stav Fault

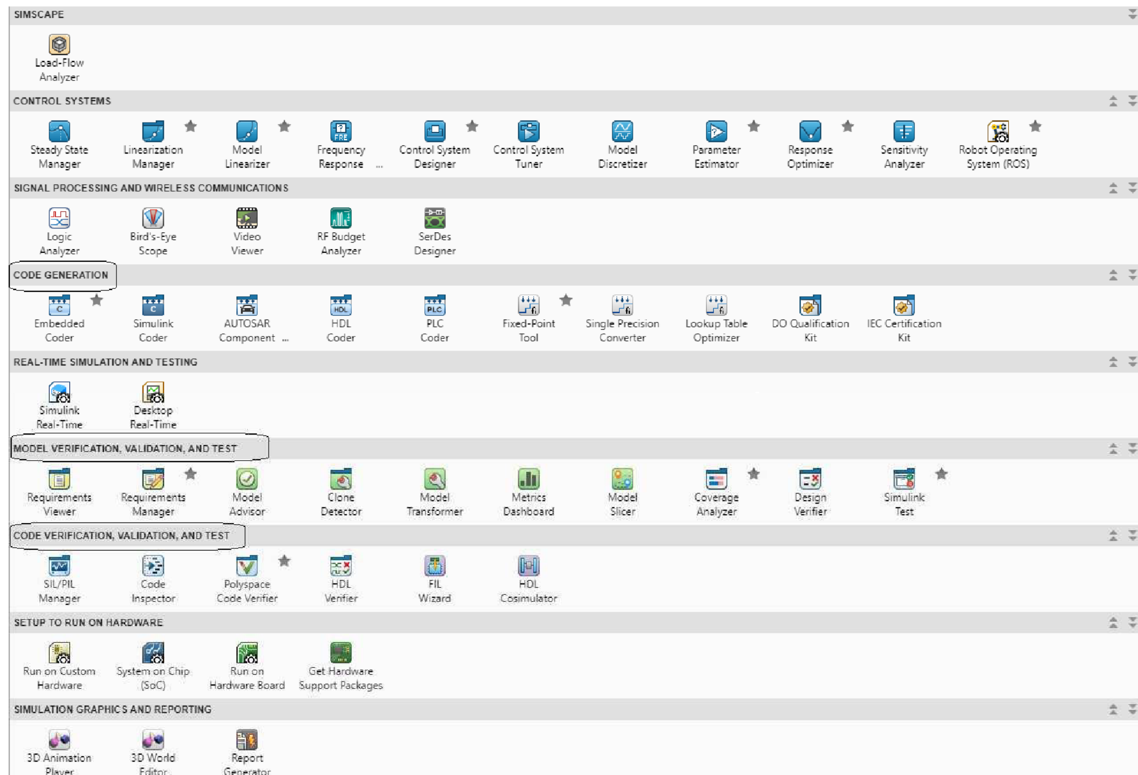
8 – Funkcia kontroluje, či sa motor už netočí



Obrázok 7.5 Výsledný stavový automat riadenia [27]

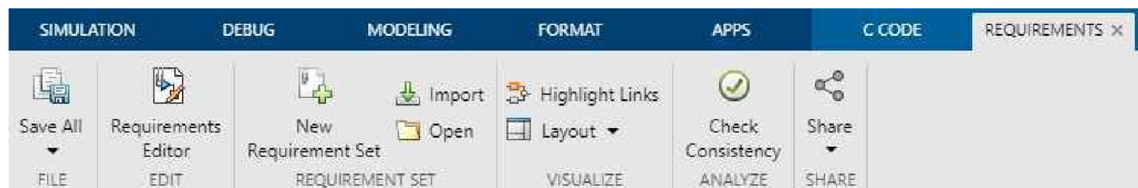
7.5 Overenie kódu a ďalšie aplikácie

Najnovšia verzia Matlab/Simulink disponuje obrovským množstvom aplikácii, ktoré sa môžu použiť v rámci vývoja a overenia kódu. Na obrázku 7.5 sú zobrazene možné aplikácie a doplnky, ktoré je možno používať. Ak sa zameriame na realizáciu a test požiadaviek, verifikáciu a testovanie vidíme že vyznačených blokov nie je malo.



Obrázok 7.6 Zobrazenie doplnkov a aplikácii v Simulinku [27]

Ak sa zamerane na požiadavky, budeme používať „Requirements Manager“. V tomto manažéri sa zobrazí nasledujúca lišta.

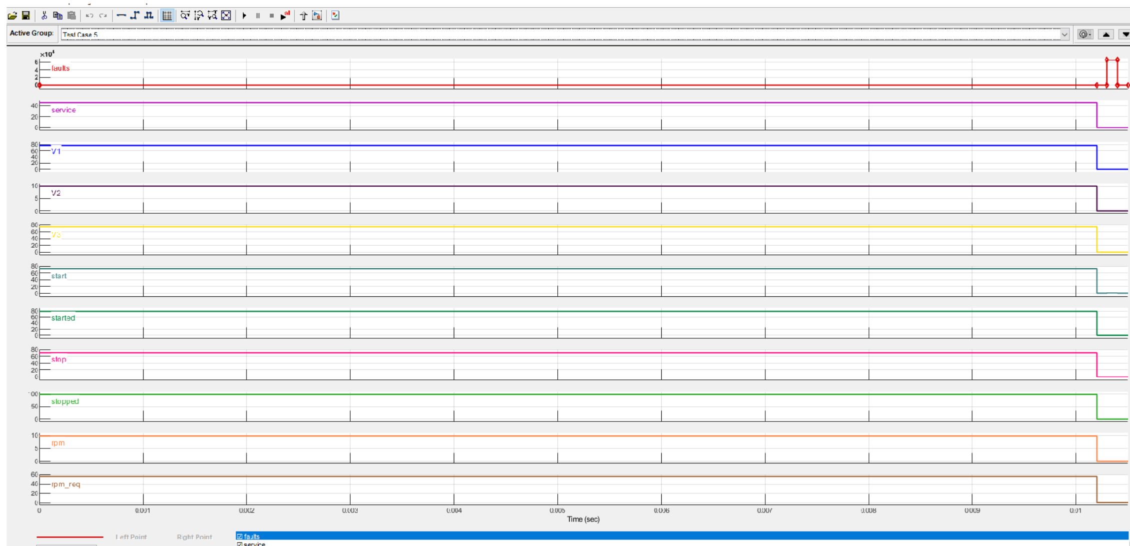


Obrázok 7.7 Requirements manager [27]

Pre nás je dôležíte zvýraznenie požiadaviek a to pomocou kolónky „Highlight Links“. Taktiež pod možnosťou „Share“ môžeme vygenerovať report trasovateľnosti požiadaviek. Jedna z možnosti napísania požiadaviek je písanie priamo v prostredí Simulink a to pomocou takzvaných „Requirement set“. Užívateľ si vytvorí takýto set, na počte nezáleží a v nich definuje jednotlivé požiadavky. Je to užívateľsky prívetivé, do popisov sa dajú vkladať kľúčové slova, typ požiadavky a iné. Po vytvorení takýchto

požiadaviek sa jednoducho klikne pravým tlačidlom myši na časť kódu či bloku a pomocou kontextového menu vyberie „Requirements“. Pri výbere v tomto menu „Requirement browser“ dokážeme požiadavku nalinkovať. V praxi sú však požiadavky písane ešte pred programom. To nie je problém, pretože pomocou tohto kontextového menu dokážeme nalinkovať taktiež na externý Excel či Word. Tato aplikácia ma ešte veľmi podstatnú funkciu a to analýza – „Check Consistency“. Vďaka tejto funkcii mame možnosť vyberania z veľkého množstva parametrov, kde môžeme skontrolovať správnosť požiadaviek, ale taktiež tento modul skontroluje štandard pre DO-254, DO-178C, či ISO 26262. Na ukážku bola vytvorená jednoduchá požiadavka v Excelu a iná v „Requirements Modulu“. Tie sú nasledovne nalinkovane v modelu.

Ďalšou aplikáciou je „Design Verifier“. Tato aplikácia sama vytvorí testy, pričom sa snaží nájsť skryté chyby v modeloch, pretečenia atď. Aplikácia vytvorí všetky možné kombinácie jednotlivých vstupov a snaží sa otestovať všetky stavy. Užívateľ si testy a zapojenie dokáže sám upraviť. Na obrázku 7.7 je jeden vygenerovaný test. Tieto testy si jednoducho užívateľ môže upravovať alebo vytvárať celé nové testy. Okrem týchto testov, sa skúšajú vykonávať všetky podmienky a overuje sa správanie, pričom sa podrobné informácie nachádzajú v reporte. Report sa nachádza v elektronickej prílohe. Týmto sa výsledne pokrytie modelu a jeho report dá jednoducho skontrolovať a vygenerovať.

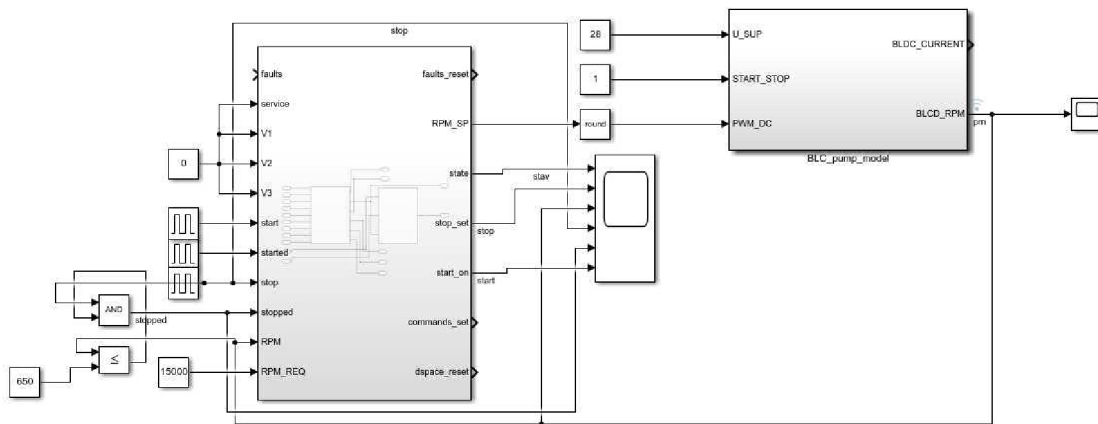


Obrázok 7.8 Príklad jedného test casu vygenerovaného v Design Verifier [27]

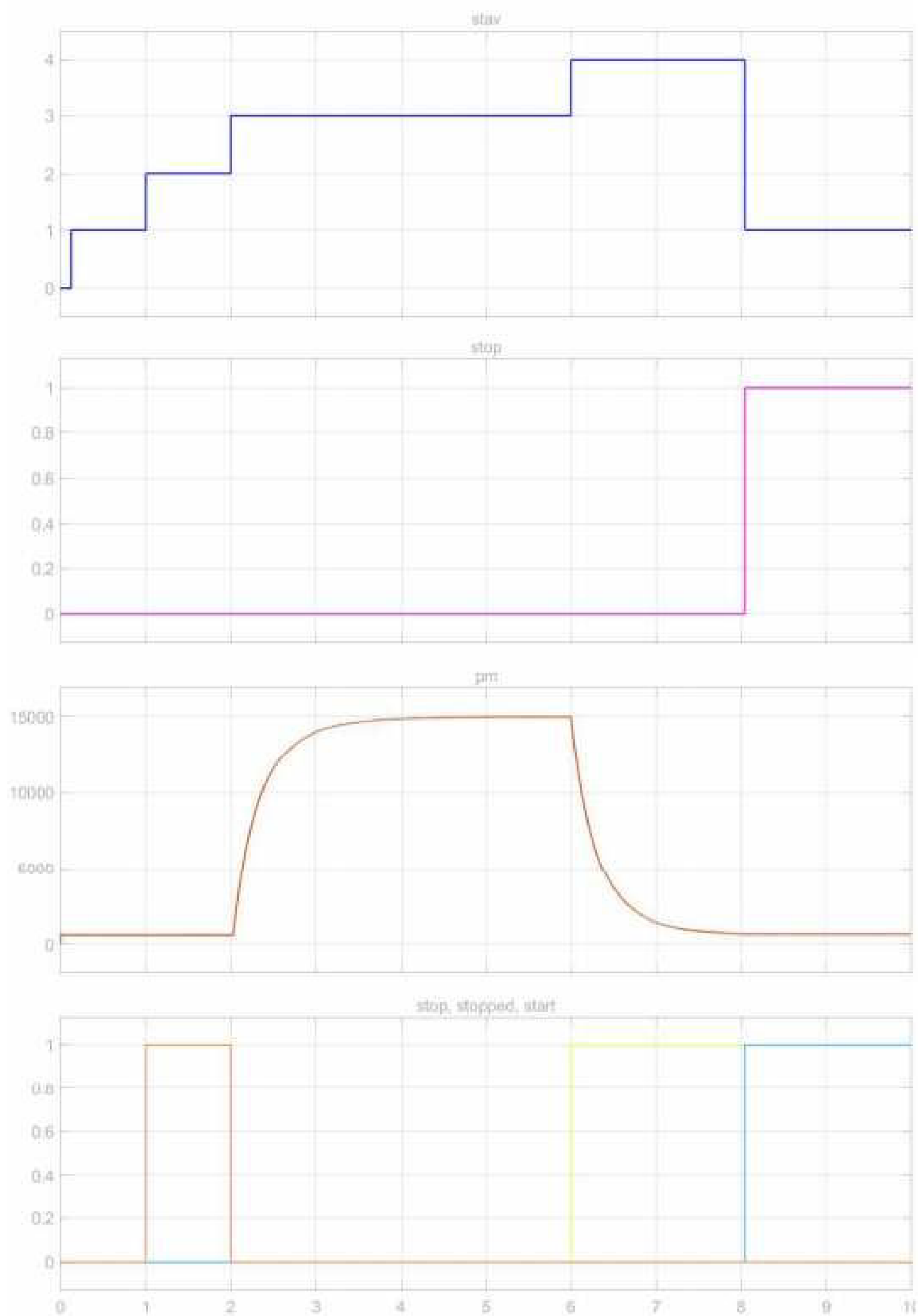
Pri aplikácii „Model Coverage“ Simulink debuguje program a užívateľ si musí sám namodelovať testované prípady. Všetko vidíme v reálnom čase a keď sú požiadavky a testy správne vytvorené mali by sa nájsť všetky nedostatky. Takto zjednodušene povedane verifikovaný software môžeme následne verifikovať pomocou HIL/PIL verifikačného modelu. Vďaka tomuto modelu sa dajú jednoducho logovať signály, ktoré

chceme. Každopádne pre PIL simuláciu musí byť model správne nakonfigurovaný a procesor, ktorý používame musí byt podporovaný. V rámci tohto projektu simulácie PIL a SIL neboli vykonané.

Navrhnuté riadenie som okrem simulácii samotných modelov otestoval testami. Na model som priviedol nasimulované vstupy pričom som kontroloval požadované výstupné signály. Na motor sa priviedol signál start. Po čase mu bol privedený nasimulovaný signál started. Následne pár sekund bežal a potom bol privedený príkaz stop. Na obrázku 7.10 sa nachádzajú kontrolované vstupy a výstupy. Automat sa z initu postupne dostal cez ready, starting až k stavu running. Po príkazu stop, začal stav stopping. Ak otáčky padli na požadovanú rýchlosť, pri ktorej je možné motor zastaviť tak sa motor zastavil a automat prešiel naspäť do ready. Testovanie ostatných vlastností prebiehalo podobne. Všetky tieto vlastnosti sa otestovali aj už pomocou reálneho hardwaru.



Obrázok 7.9 Zapojenie pre testovanie [27]



Obrázok 7.10 Sledované vstupy a výstupy pre testovanie [27]

Ak by tento projekt bol kriticko bezpečnostne náročný určite by nám pomohol DO Qualficiatoion kit, ktorý dôkaze kvalifikovať Simulink nástroje v normách DO-178C a DO-278A. Tento kit poskytuje širokú dokumentáciu, vygenerovanie matice

trasovateľnosti a návody, ktoré dôkazu kvalifikovať jednotlivé použite nástroje. V rámci takto kritického kódu je doležíte spomenúť ďalší nástroj a to Polyspace. Certifikovaný software, vďaka ktorému môžeme analyzovať zdrojový kód v rámci normy DO-178 a štandardu MISRA. Tento software je taktiež výborný pomocník pre verifikovanie ručne písaného kódu. Samozrejme testovanie sa môže uskutočniť aj bez týchto pomocníkov.

7.6 Zhrnutie

Matlab r2020b a jeho doplnky dôkazu byt veľkým pomocníkom a cestou vývoja nového softwaru. Vďaka množstvu aplikácii dokážeme verifikovať software na požadovanej úrovni, vytvárať test procesy a prehľadný software. To všetko aj bez dovedností programovania v C. Moje riešenie zahŕňa použitie Stateflow, ktorý uľahčuje naprogramovanie celého riadenia a regulátoru. Podarilo sa mi kód overiť. Myslím si že každá z aplikácii je užívateľsky prívetivá a ľahko porozumiteľná.

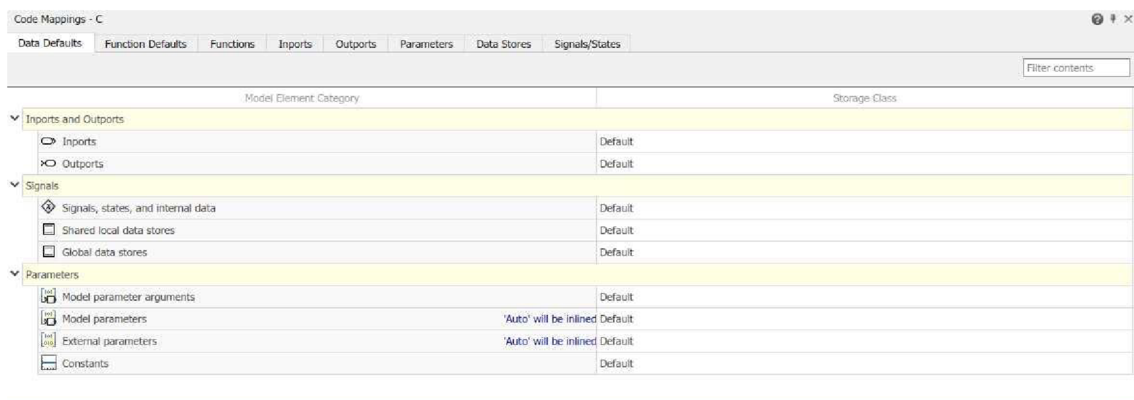
Obrázok 7.11 Výsledný produkt [27]

8. SPOJENIE GENEROVANÉHO A RUČNE PÍSANÉHO KÓDU

Po vytvorení modelu, jeho overení, naprogramovania low-level vrstvy riadiaceho procesoru a rozhrania pre užívateľa zostávala posledná časť a to je ako vyriešiť spojenie takto vygenerovaného kódu s ručne písaným kódom.

8.1 Embedded Coder interface

Pri použití aplikácie Embedded Coder pod záložkou „Code Interface“ si zobrazíme namapovanie všetkých vstupov, výstupov, funkcii atď.



Obrázok 8.1 Mapovanie generovaného kódu [27]

Ako vstupy a výstupy používame bloky z knižnice a to „inport“ a „outport“. Pod záložkou „Functions“ sú zobrazené funkcie, ktoré sa vygenerujú a tie je možné samostatne volať už v ručne písanom kóde. Funkcie môžeme viazať na každý model a pridávať ich v „Embedded Code Dictionary“. V tomto prípade boli použité funkcie dve a to inicializácia a automats_step(), čo je celý vygenerovaný stavový automat s regulátorom.

8.2 Použité riešenie pre vstupy a výstupy

Ak chceme vidieť všetky možné prístupy k predávanie vstupov a výstupov tak využijeme „Embedded Coder Dictionary“. Obsahuje široké spektrum možností, ako vstupy a výstupy definovať a to napríklad:
ExportedGlobal – globálna premenná

- ImportedExtern/Pointer – import vstupu z externého zdrojového kódu
- BitField – premenná sa rozpíše do štruktúry definovanými bitmi
- Const, define, volatile, struct – známe kľúčové slová v C jazyku, ktoré poznáme a môžeme s nimi definovať nové premenné

- GetSet – volanie cez funkcie, ktoré sú definované v externom zdrojovom kóde

V tejto práci som sa rozhodol o predávanie pomocou funkcií a to je voľba „GetSet“. Celý princíp spočíva vo vytvorení zdrojového c súboru, ktorý obsahuje funkcie get a set. Každý vstup a výstup má zadefinované svoje meno funkcie, ktoré sa nastavuje pomocou „Property Inspector“. Pri kompilácií kódu musíme zadefinovať tieto zdrojové súbory do externých súborov v nastavení. Pri generovaniu to nie je nutné. Po vygenerovaní sa vstupy a výstupy volajú a nastavujú pomocou týchto funkcií.

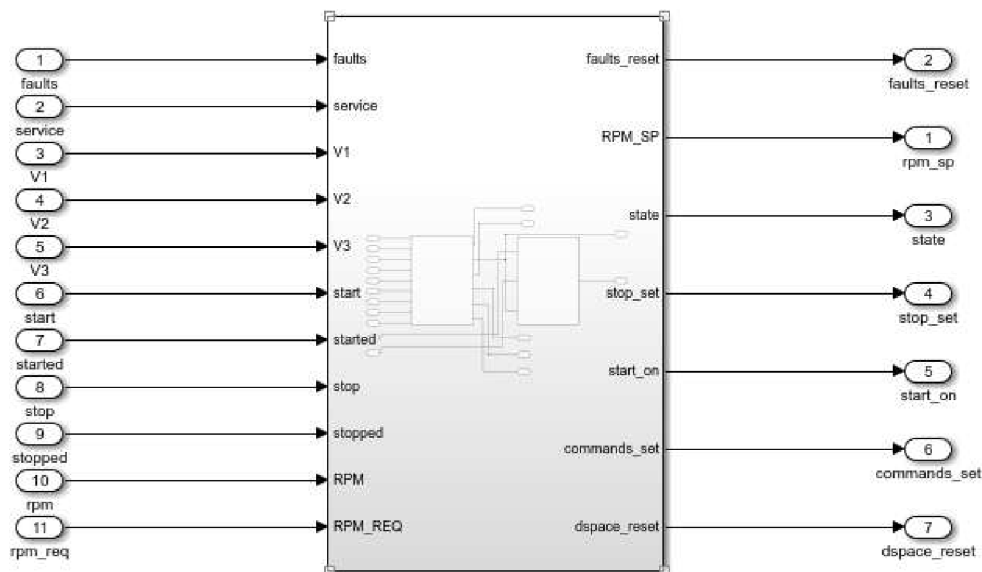
```
switch (rtDW.is_c2_automats) {
 case IN_Fault:
  if (get_service() == 1) {
   rtDW.is_c2_automats = IN_Service;

   /* Outport: '<Root>/faults_reset' */
   v_set_faults_reset(1U);

   /* Outport: '<Root>/state' */
   v_set_state(5U);
  }
```

Obrázok 8.2 Ukážka vo vygenerovanom kóde [27]

Myslím si, že toto je jedna z najlepších možností ako túto problematiku vyriešiť. Ak to nestačí, užívateľ si dokáže nadefinovať sám spôsob predávania. Celé vygenerovanie a všetky nastavenia sa dajú taktiež ovládať pomocou jazyku Matlab. Preto si myslím, že by vôbec nebolo zlé napísať rovno nejakú aplikáciu, ktorá by si dokázala všetky vstupy a výstupy vytiahnuť napríklad z Excelu a vytvoriť a nastaviť celý interface bez zbytočne zdĺhavého nastavovania vstupov a výstupov po jednom.



Obrázok 8.3 Model pre generovanie, na ľavo vstupy, na pravo výstupy [27]

9. ZÁVER

Na začiatku diplomovej práce bola vykonaná rešerš a porovnané prístupy vo vývoji SW a to pre konvenčný a MBD prístup. Pre konvenčný prístup bol podrobne opísaný vývoj v rámci vodopádového modelu a pre MBD takzvaný V model. V rámci V modelu sú stručne vysvetlené možne simulácie a testy pre MIL, SIL, PIL a HIL a je predstavená ich úloha. Jednotlivé výhody a nevýhody boli nasledovne zhrnuté.

V ďalšej časti diplomovej práce je vykonaný prieskum trhu v oblasti nástrojov na generovanie, validáciu a verifikáciu generovaného kódu. Boli predstavené základné prístupy, akými môžeme verifikáciu a validáciu vykonávať. Vysvetlené sú taktiež všetky základne pojmy ako požiadavky, trasovateľnosť a ich prínosy vo validácii a verifikácii.

Podrobne boli popísané nástroje od firmy MathWorks. Je to zapríčinené hlavne tým, že tieto nástroje sú momentálne najpoužívanejšie a rozhodol som sa ich použiť pri riešení mojej diplomovej práce. Okrem toho boli podrobnejšie popísané riešenia od firmy dSpace a z časti poukázané ostatné riešenia.

Nasleduje časť popisujúca vývoj MBD v oblasti kriticko-náročných aplikácii. V diplomovej práci sú popísane normy ako DO-178C a norma DO-331 pre MBD. Sú zdôraznené jednotlivé fázy pri vývoji takéhoto softwaru a možné benefity v rámci MBD. Taktiež sú spomenuté reálne príklady použitia MBD pre takéto aplikácie. Tieto teoretické znalosti a poznatky z oboru tykajúceho sa model-based-design a automatického generovania kódu som využil v ďalšej časti

V práci som sa taktiež venoval výberu vhodných hardwarových a softwarových platforiem. K vybraným platformám som vybral vývojové prostredia, navrhol ich vzájomnú komunikáciu a zapojenia. Pre vybrané platformy som navrhol jednoduchú krabičku, do ktorých boli vložené.

V nasledujúcej kapitole som navrhol low-level vrstvu pre oba mikrokontroléry. Boli nastavené všetky periférie a vytvorené užívateľské rozhranie. Navrhol a vytvoril som algoritmy riadenia BLDC motoru. Všetky takto vytvorené programy sú napísané v jazyku C. Boli popísane jednotlivé PWM metódy a ich výhody a nevýhody. Taktiež som popísal BLDC motory ich princípy a druhy ovládania.

Po vytvorení low-level vrstvy, som pokračoval v naprogramovaní logiky v prostredí Simulink. V tomto prostredí som vytvoril riadenie, ktoré obsahuje stavový automat a regulátor otáčok. Následne som logiku riadenia v Simulinku a jej vytváranie v StateFlow popísal a takto vytvoril jednoduchý návod Vytvorenú logiku riadenia som overil a otestoval pomocou dostupných aplikácii a vygeneroval reporty, ktoré sú súčasťou príloh. Pre celý model som navrhol spôsob komunikácie vygenerovaného kódu s low level vrstvami.

Pre vytvorené modely som vygeneroval kód. Vytvoril som rozhranie pre spojenie generovaného a ručne písaného kódu. Tento celý postup generovania kódu, postup nastavenia rozhrania som popísal.

Vygenerovaný kód som spojil s napísaným kódom pre mikrokontrolér. Takto spojený kód som otestoval spolu s užívateľským rozhraním implementovaným na druhom mikrokontroléru.

Výsledný program funguje, pričom som overil správnosť navrhnutej low level vrstvy a navrhnutého modelu.

LITERATÚRA

[1] MAVURU, Indusree. Traditional vs. Agile Software Development Methodologies. KPI Partners blog [online]. New Jersey: KPI Partners, 2018 [cit. 2021-01-03]. Dostupné z: https://www.kpipartners.com/blog/traditional-vs-agile-software-development-methodologies

[2] Waterfall model. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2021 [cit. 2021-01-03]. Dostupné z: https://en.wikipedia.org/wiki/Waterfall_model

[3] MŮČKA, Ján. Vývoj software: jaké jsou základní bezpečnostní principy? MasterDC [online]. Brno: MasterDC, 2020 [cit. 2021-01-03]. Dostupné z: https://www.master.cz/blog/vyvoj-software-zakladni-bezpecnostni-principy/

[4] System Design Of The Waterfall Implementation Model Information Technology Essay. UKessays [online]. Nottingham: UKEssays, 2021 [cit. 2021-01-03]. Dostupné z: https://www.ukessays.com/essays/information-technology/system-design-of-the-waterfall-implementation-model-information-technology-essay.php

[5] NICOLESCU, Gabriela a Pieter J. Monsterman. Model-Based Deisgn for Embedded Systems. 4th edition. Boca Raton: CRC Press, 2010. ISBN 9781138114722.

[6] RIERSON, Leanna. Developing safety-critical software. New York: CRC Press, 2013. ISBN 978-1-4398-1368-3.

[7] BERGMANN, Arno. Benefits and Drawbacks of Model-based Design. ResearchGate [online]. London: ResearchGate, 2014 [cit. 2021-01-03]. Dostupné z: https://www.researchgate.net/publication/270494068_Benefits_and_Drawbacks_of_Model-basedd_Design

[8] SDLC- V-Model. Tutorialspoint [online]. tutorialspoint, 2020 [cit. 2021-01-03]. Dostupné z: https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm

[9] SCHIEFERDECKER, Ina a Pieter MOSTERMAN, ZANDER, Justina, ed. Model-Based Testing for Embedded Systems. Boca Raton: CRC Press, 2011. ISBN 9781439818459.

[10] CONRAD, Martin. Fig. 4 Translation validation process for generated code. In: Semanticscholar [online]. 2009 [cit. 2021-01-03]. Dostupné z: https://www.semanticscholar.org/paper/Testing-based-translation-validation-of-generated-Conrad/cd0cfe5665e7907d1e006df05c1096372ca7307d

[11] Mathworks [online]. USA: Mathworks, 2021 [cit. 2021-01-03]. Dostupné z: https://www.mathworks.com/products/matlab.html

[12] BARESEL, André. Decision coverage on model level -pathways through a Switch block and test goals. In: ResearchGate [online]. London: ResearchGate, 2003 [cit. 2021-01-03]. Dostupné z: https://www.researchgate.net/publication/247918185_The_Interplay_between_Model_Coverage_and_Code_Coverage

[13] Perform Functional Testing and Analyze Test Coverage. Mathworks [online]. Massachusetts: mathworks, 2020 [cit. 2021-01-03]. Dostupné z: https://www.mathworks.com/help/sltest/ug/functional-testing-and-coverage-analysis.html

[14] TargetLink. Dspace [online]. Paderborn: dspace, 2020 [cit. 2020-12-10]. Dostupné z: https://www.dspace.com/en/pub/home/products/sw/pcgs/targetlink.cfm

[15] SCADE [online]. SCADE, 2020 [cit. 2020-01-09]. Dostupné z: https://www.ansys.com/products/embedded-software/ansys-scade-suite

[16] HILDERMAN, Vance a Tony BAGHAI. Avionics Certification. 2nd. Lessburg: Avionics Communications, 2013. ISBN 978-1-885544-33-9.

[17] Developing Airplane Systems Faster and with Higher Quality through Model-Based Engineering. Boeing [online]. USA [cit. 2021-01-03]. Dostupné z: https://www.boeing.com/features/innovation-quarterly/may2017/feature-technical-model-based-engineering.page

[18] FREEMAN, Colin. Bombardier - Train Zero Model Based Design Facility. NI - Engineer Ambitiously [online]. UK: Ni [cit. 2021-01-03]. Dostupné z: https://www.ni.com/en-us/innovations/case-studies/19/bombardier-train-zero-model-based-design-facility.html

[19] TYLER, Eric, Farhad SHAHNIA a S. RAJAKARUNA. Application notes and recommendations on using TMS320F28335. ResearchGate [online]. Australia: ResearchGate, 2014 [cit. 2021-3-20]. Dostupné z: https://www.researchgate.net/publication/286651203_Application_notes_and_recommendations_on_using_TMS320F28335_digital_Signal_Processor_to_control_voltage_source_converters

[20] Discovery kit with STM32F746NG MCU. STM [online]. Ženeva: STMicroelectronics, 2021 [cit. 2021-4-12]. Dostupné z: https://www.st.com/en/evaluation-tools/32f746gdiscovery.html

[21] TMDSDOCK28335: TMS320F28335 Experimenter Kit. Texas Instruments [online]. Dallas: Texas Instruments, 2021 [cit. 2021-4-18]. Dostupné z: https://www.ti.com/tool/TMDSDOCK28335

[22] KOZÁČEK, BC., Peter. RIADENIE BLDC MOTORA V OBLASTI NÍZKYCH OTÁČIEK. Brno, 2015. Diplomová práca. VUT. Vedoucí práce Ing. Libor Veselý, Ph.D.

[23] Sensorless BLDC Motor Control with Back-EMF Filtering Using a Majority Function. DigiKey [online]. Minnesota: DigiKey, 2011 [cit. 2021-5-14]. Dostupné z: https://www.digikey.cz/en/articles/sensorless-BLDC-motor-control-with-back-emf-filtering-using-a-majority-function

[24] AN2009 APPLICATION NOTE. STM [online]. Ženeva: STMicroelectronics, 2011 [cit. 2021-5-14]. Dostupné z: https://www.st.com/resource/en/application_note/cd00041736-pwm-management-for-3phase-bldc-motor-drives-using-the-st7mc-stmicroelectronics.pdf

[25] HUH, Nam, Hyung-Seok PARK, Man HYUNG LEE a Jang-Mok KIM. Hybrid PWM Control for Regulating the High-Speed Operation of BLDC Motors and Expanding the Current Sensing Range of DC-link Single-Shunt. Energies. 2019, 2019(1), 13. Dostupné z: https://www.mdpi.com/1996-1073/13/19/5212/pdf

[26] PROKOP, Libor. Using Motor Control eFlexPWM (mcPWM) for BLDC Motors. NXP Semiconductors [online]. Arizona, 2011 [cit. 2021-4-28]. Dostupné z: https://www.nxp.com/docs/en/application-note/AN4429.pdf

[27] Archív autora

ZOZNAM SYMBOLOV A SKRATIEK

Skratky:

FEKT	Fakulta elektrotechniky a komunikačních technologií
VUT	Vysoké učení technické v Brně
MBD	Model-Based-Design
SW	Software
HW	Hardware
SIL	Software In the Loop
PIL	Processor In the Loop
HIL	Harware In the Loop
MIL	Model In the Loop
PWM	Pulse Width Modulation
AD	Analog to Digital
BLDC	Brushless DC Electric motor
DC	Direct Current
UML	Unified Modeling Language
CC	Condition Coverage
BC	Branch Coverage
RC	Range Coverage
MCDC	Modified Condition Decision Coverage
LCD	Liquid Crystal Display
SCI	Serial Communication Interface
RGB	Red Green Blue
LDRA	Liverpool Data Research Associates
AUTOSAR	AUTomotive Open System ARchitecture
MISRA	Motor Industry Software Reliability Association
RAM	Random Access Memory
SDRAM	Synchronous Dynamic Random Access Memory
UART	Universal asynchronous receiver-transmitter
USART	Universal Synchronous / Asynchronous Receiver and Transmitter
I2C	Two Wire Interface
NASA	National Aeronautics and Space Administration
DPS	Doska Plošných Spojov
USB	Universal Serial Bus
CAN	Controller Area Network
SPI	Serial Peripheral Interface
IGBT	Insulated Gate Bipolar Transistor
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
BEMF	Back Electromotive Force

atď	a tak ďalej
CRC	Cyclic Redundancy Check

Symboly:

V	volt
ms	mili sekundy
µs	mikro sekundy
kHz	kilo hertz
MHz	mega hert

ZOZNAM ELEKTRONICKÝCH PRÍLOH

Všetky elektronické prílohy sú súčasťou priloženého CD a sú taktiež elektronický spoločne s textom diplomovej práce.

I. Projekt pre riadiaci procesor TMS
II. Projekt pre užívateľské rozhranie STM
III. Model regulátoru s modelom motoru
IV. Model stavového automatu zapojený s regulátorom
V. Polyspace projekt a report
VI. Deisgn verifier a Coverage report
VII. Test case pre model stavového automatu s regulátorom
VIII. Vygenerovaný kód a report
IX. Požiadavky
X. Schéma a DPS prevodníka konektora
XI. Návrh krabičky