VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV VÝKONOVÉ ELEKTROTECHNIKY A ELEKTRONIKY
DEPARTMENT OF POWER ELECTRICAL AND ELECTRONIC ENGINEERING

ŘÍZENÍ BLDC MOTORU
BLDC MOTOR CONTROL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Ondřej Kabelka**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Jan Knobloch, Ph.D.**
SUPERVISOR

BRNO 2019

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Silnoproudá elektrotechnika a výkonová elektronika**
Ústav výkonové elektrotechniky a elektroniky

Student: Bc. Ondřej Kabelka ***ID:*** 146854
Ročník: 2 ***Akademický rok:*** 2018/19

NÁZEV TÉMATU:

Řízení BLDC motoru

POKYNY PRO VYPRACOVÁNÍ:

1. Popište použitý hardware.
2. Naprogramujte a odlaďte řídicí algoritmus motoru.
3. Proveďte měření a diskutujte výsledky.

DOPORUČENÁ LITERATURA:

[1] SKALICKÝ, Jiří. Elektrické servopohony. Vyd. 2. Brno: Vysoké učení technické, 2001. ISBN 80-214-1978-4.

[2] KRISHNAN, R. Permanent magnet synchronous and brushless DC motor drives. Boca Raton, U.S.A.: CRC Press, 2010. ISBN 978-0-8247-5384-9.

[3] XIA, Chang-liang. Permanent Magnet Brushless DC Motor Drives and Controls. China: Tianjin University, 2012. ISBN 978-1-118-18833-0.

Termín zadání: 4.2.2019 ***Termín odevzdání:*** 22.5.2019

Vedoucí práce: Ing. Jan Knobloch, Ph.D.
Konzultant:

doc. Ing. Ondřej Vítek, Ph.D.
předseda oborové rady

Abstrakt

Tato práce se zabývá řízením BLDC motoru. Nejprve je krátce představen princip BLDC motoru, v navazujících kapitolách jsou pak rozebrány jednotlivé metody řízení. Další část se zabývá úvodem do platformy STM32, na které je implementován řídící algoritmus. V rámci práce byl vytvořen program pro senzorové a bezsenzorové řízení BLDC motoru. Na závěr jsou naměřeny kontrolní průběhy napětí a proudu při užití vytvořeného algoritmu.

Klíčová slova

BLDC motor, Hallovy sondy, řízení, bezsenzorové řízení

Abstract

The thesis deals in the control of the BLDC motor. At first, there is a brief introduction of BLDC motor, in the next chapters are explained possible methods of control. The next part of the thesis deals with the platform STM32, on which the control algorithm is implemented. For the thesis a program for sensor and sensorless control of BLDC motor has been created. In the conclusion, the process of voltage and current has been measured, using the created algorithm.

Keywords

BLDC motor, Hall sensors, control, sensorless control

Bibliografická citace:

KABELKA, Ondřej. *Řízení BLDC motoru*. Brno, 2019. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/116844. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav výkonové elektrotechniky a elektroniky. Vedoucí práce Jan Knobloch.

Prohlášení

„Prohlašuji, že svou diplomovou práci na téma Řízení BLDC motoru jsem vypracoval samostatně pod vedením vedoucího 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: **21. května 2019** ……

podpis autora

Poděkování

Děkuji vedoucímu své diplomové práce, panu Ing. Janu Knoblochovi, Ph.D. za jeho obětavý přístup a odbornou pomoc při zpracování mé diplomové práce. Dále děkuji slečně Bc. Martině Sachsové za její psychickou podporu během celého mého studia. V neposlední řadě pak děkuji panu Ing. Jiřímu Čermákovi, za dodání motivace při psaní mé diplomové práce.

V Brně dne: **21. května 2019** ……

podpis autora

Obsah

Seznam obrázků

Seznam tabulek

1 ÚVOD

Bezkartáčové stejnosměrné motory (BLDC) se v poslední době používají ve stále větším spektru systémů. Jejich vysoká spolehlivost a snadné ovládání ovšem byly zastíněny jejich cenou. Nutnost přidávat k motorům Hallovy sondy a řadiče totiž dramaticky navyšuje jejich cenu a omezuje tím možnosti jejich použití.

S rozšířením digitálních mikrokontrolerů již není potřeba nákladné řídící elektroniky a BLDC motory se tak stávají dostupnější pro nejrůznější aplikace, kde je zdrojem napájení stejnosměrné napětí. Používají se například v elektrických vysokozdvižných vozících, v armádních a leteckých pohonných systémech (např. jako čerpadla paliva), v elektrickém ručním nářadí s akumulátorem, v počítačových větrácích nebo v pomocných systémech dopravních prostředků.

Bezsenzorové řízení využívá již zabudovaný mikroprocesor nahrazující řadiče k řízení motoru bez potřeby Hallových sond. Ačkoliv samotný algoritmus řízení je o něco složitější, výhodou tohoto druhu řízení je to, že nepřidává žádný hardware navíc. Naopak se obejde bez dříve nutných senzorů polohy, čímž snižuje celkovou cenu přípravku a činí ho tak dostupnějším.

Tato diplomová práce se zabývá bezsenzorovým řízením BLDC motoru. Nejprve je krátce vysvětlen princip BLDC motoru a jeho řízení, a následně jsou popsány jednotlivé metody bezsenzorového řízení. V další části práce je pak představen hardware a software, který je v práci použit. V poslední kapitole je pak popsána praktická implementace vytvořeného senzorového a bezsenzorového řízení.

2 POPIS BLDC MOTORU

2.1 Konstrukční uspořádání BLDC motoru

Název BLDC motoru je zkratkou pro Brushless DC, tedy bezkartáčový stejnosměrný motor. V klasickém stejnosměrném motoru jsou v jeho rotoru umístěny cívky napájené stejnosměrným proudem přes komutátor. Kontakty těchto cívek jsou řešeny kartáči. Tyto kartáče jsou zdrojem určitých nepříjemností DC motoru, při chodu může vznikat jiskření, přechodový odpor kartáčů kolísá s natočením rotoru a v případě uhlíkových kartáčů dochází k jejich obrušování, což s sebou nese potřebu uhlíky měnit a motor čas od času vyčistit.

Oproti tomu BLDC motor žádné kartáče nemá, jak už vyplývá z jeho názvu. To s sebou nese výhodu tišího chodu a vyšších maximálních otáček. Konstrukční uspořádání je takové, že v rotoru jsou pouze permanentní magnety, zatímco cívky se nacházejí ve statoru. Pro zajištění chodu motoru je třeba určovat natočení rotoru a podle něj pak přepínat proudy jednotlivými cívkami ve statoru, což zajišťuje obvykle integrovaný mikročip. Kvůli tomu se BLDC motory někdy označují jako *Elektronicky komutované motory*.

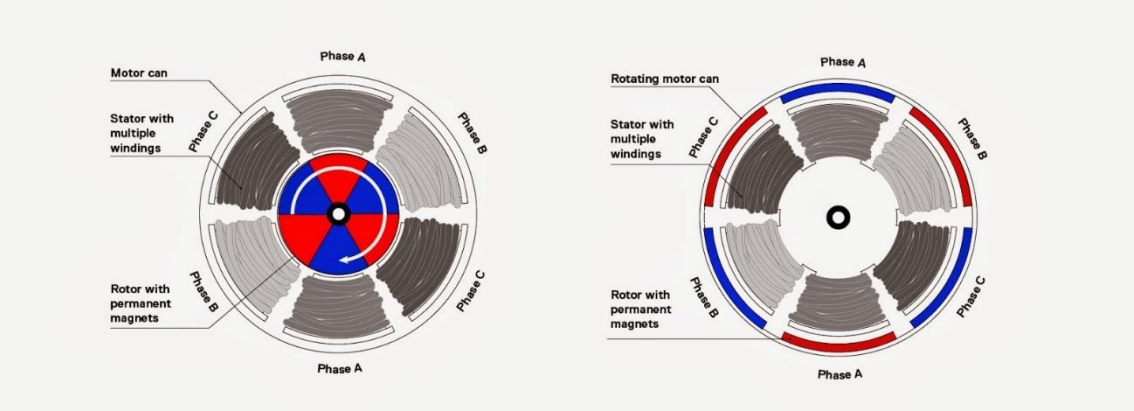
Konstrukce statoru BLDC motoru je podobná jako u asynchronních strojů. Z důvodu omezení ztrát vířivými proudy je stator konstruován z plechů. V drážkách statoru je umístěno vinutí, které obvykle bývá třífázové.

Rotor BLDC motoru je opatřen permanentními magnety. Obvykle se používají neodymové, případně samarium-kobaltové. Tyto magnety limitují výkon motoru, neboť při velkém oteplení může docházet k jejich demagnetizaci.

Základním požadavkem u BLDC motoru je obdélníkový průběh indukovaného napětí. Žádaného tvaru se dosahuje úpravou provedení vinutí, případně tvarem permanentních magnetů v rotoru. Průběhy indukovaného napětí jsou takové, že v každém okamžiku je na jedné fázi proud v kladném směru, na další fázi je stejný proud v záporném směru a poslední fází proud neprochází (komutuje se).

BLDC motory mohou být dvojího typu, s vnitřním rotorem a s vnějším rotorem (outrunner). Motory s vnějším rotorem mají výhodu většího objemu rotoru. Tím pádem lze použít větší permanentní magnety pro dosažení žádané magnetické indukce ve vzduchové mezeře než u klasického řešení. Tyto magnety tak mohou být slabší, a tedy

levnější, než u klasického uspořádání. Nevýhodou tohoto řešení je horší odvod tepla a vyšší setrvačnost rotoru. Tyto motory se používají především v magnetických harddiscích, v optických mechanikách a v počítačových větrácích. [2]



Obr. 2-1: Ilustrační obrázek uspořádání motorů s vnitřním a vnějším rotorem, zdroj [9]

2.2 Hallovy sondy

Hallovy sondy jsou nejběžnějšími senzory polohy používanými v BLDC motorech. Senzory identifikují přítomnost magnetického pole. Fungují na principu Hallova jevu, kdy se na polovodiči, kterým protéká proud I, vloženém do stacionárního magnetického pole vytváří tzv. Hallovo napětí, dané rovnicí

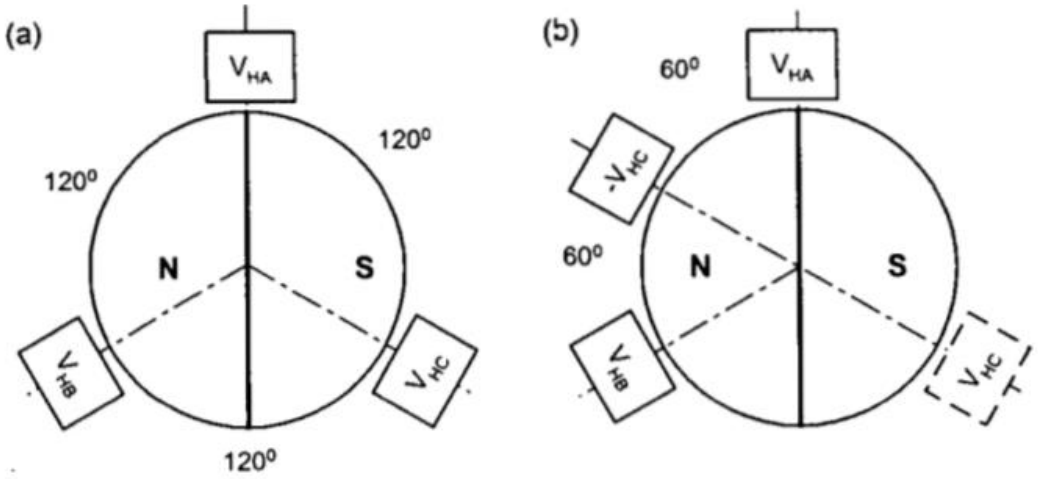
$$U_H = k_H \frac{IB}{d} sin\beta \qquad \textbf{(2-1)}$$

Kde k_H je Hallova konstanta, B je magnetická indukce, d je tloušťka polovodičové destičky a β je úhel, který svírá směr magnetické indukce s plochou polovodiče.
Při měření polohy rotoru Hallovými sondami se používají dvě různá rozložení sond. Tato rozložení jsou dána počtem pólových párů rotoru a fází statoru podle vzorce

$$\alpha = \frac{360}{p.m} \qquad \textbf{(2-2)}$$

Kde α je úhel natočení sond, p počet pólových párů a m je v tomto vzorci počet fází. Při použití třífázového motoru dostáváme 120° natočení, použít se dá i natočení o 60°.

Velkou výhodou Hallových sond je jejich nízká cena a spolehlivost i při frekvencích přesahujících 100kHz. [3]



Obr. 2-2: Rozložení Hallových sond v BLDC motoru, zdroj [3]

3 ŘÍZENÍ BLDC MOTORU

BLDC motor je řízením podobný synchronnímu motoru. Komutátor je zde proveden elektronicky. Napájení motoru je realizováno pomocí třífázového měniče. Používají se dva typy napájení, obdélníkové a sinusové.

3.1 Obdélníkové napájení (six-step)

Spínáním šesti tranzistorů je možné dosáhnout šesti různých stavů. Spínání jednotlivých tranzistorů je vypsáno v komutační tabulce Tabulka 1. Tabulka rozděluje natočení rotoru na šest kroků, z toho plyne název metody Six step. Metoda využívá tří čidel polohy, nejčastěji Hallových sond, kdy jednotlivé sondy detekují, v kterém kroku se rotor aktuálně nachází. V tabulce jsou tranzistory horní a dolní označeny zkráceně jako H. a D.

Tabulka 1: Komutační tabulka BLDC motoru

Natočení rotoru [°el]	Stav Hall. sond			Stavy tranzistorů					
	A	B	C	H. A	D. A	H. B	D. B	H. C	D. C
0° - 60°	1	1	0	Vyp.	Zap.	Vyp.	Vyp.	Zap.	Vyp.
60° - 120°	1	0	0	Vyp.	Zap.	Zap.	Vyp.	Vyp.	Vyp.
120° - 180°	1	0	1	Vyp.	Vyp.	Zap.	Vyp.	Vyp.	Zap.
180° - 240°	0	0	1	Zap.	Vyp.	Vyp.	Vyp.	Vyp.	Zap.
240° - 300°	0	1	1	Zap.	Vyp.	Vyp.	Zap.	Vyp.	Vyp.
300° - 360°	0	1	0	Vyp.	Vyp.	Vyp.	Zap.	Zap.	Vyp.

Při řízení pomocí této metody protéká proud zaráz pouze dvěma fázemi a třetí je vždy odpojena. V každém kroku se tak diskrétně mění magnetické pole statoru, které ovlivňuje polohu rotoru. Z principu metody se periodicky opakuje šest kroků natočení rotoru, což má za následek periodické zvlnění momentu, které je nežádoucí.

V motorech se zpravidla nepoužívá pouze jeden magnet s jednou pólovou dvojicí, jak bylo naznačeno ve vysvětlení. V praxi se používají rotory s více pólovými dvojicemi.

V případě použití více magnetů v rotoru je třeba definovat pojem mechanické a elektrické otáčky. Jedna mechanická otáčka n_{mech} je otočení rotoru o 360°. Elektrická otáčka stroje n_{el} je jedna komutační sekvence. Pro vztah mezi mechanickými a elektrickými otáčkami platí

$$n_{mech} = p.n_{el} \quad \textbf{(3-1)}$$

Kde p je počet magnetů (pólových párů) v rotoru.

3.2 Sinusové napájení

U ideálního BLDC motoru je průběh napájecího napětí lichoběžníkový. V praxi se ovšem často jeho průběh přibližuje spíše sinusovému tvaru. Proto je v některých případech výhodnější použít rovnou sinusové napájení, čímž se BLDC motor stává prakticky PMSM motorem.

V případě použití tohoto napájení je třeba určit polohu hřídele s mnohem jemnějším krokem než u six-stepu, což s sebou nese zvýšené nároky na procesor. V případě řízení se senzory polohy se aktuální stav hřídele interpoluje z údajů ze snímačů.

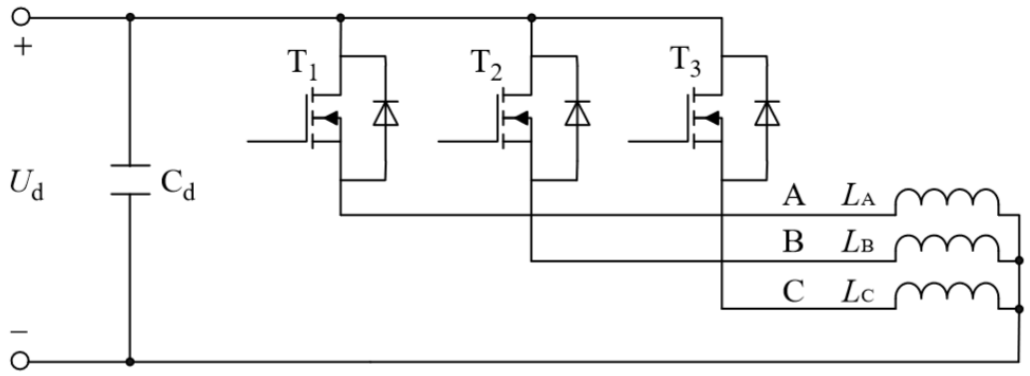
Při použití bezsenzorového řízení je třeba řídit motor pomocí observeru (pozorovatele). Observer je digitální model motoru, pomocí kterého se odhaduje aktuální natočení rotoru. Takto určený úhel natočení se pak používá pro vektorově orientované řízení.

3.3 Řízení BLDC motoru se senzory polohy

Při řízení se BLDC motor chová podobně jako motor stejnosměrný, má lineární závislost otáček na napětí a momentu na proudu. Pro zajištění plynulého chodu je pak třeba zajistit plynulou komutaci. Ta je nejčastěji realizována za pomocí třífázového měniče. Motor je obvykle zapojen třívodičově, tedy nemá vyveden středový bod. Pro komutaci lze použít měniče s polovičním nebo plným můstkem.

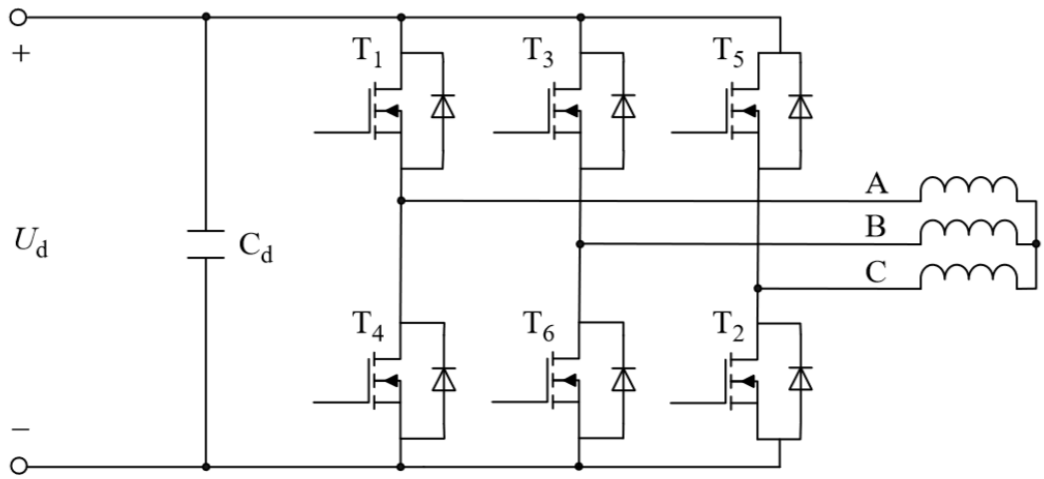
Nejjednodušším způsobem je použití polovičního můstku. Schéma zapojení je na obrázku Obr. 3-1. V tomto zapojení připadá na každou fázi jeden tranzistor, jejichž sepnutím lze na každé vinutí zvlášť přivést napětí dané polarity. I přes výhodu své jednoduchosti se toto zapojení příliš nepoužívá, protože při přepínání fází dochází ke

značným změnám momentu v závislosti na natočení rotoru. Využití nachází u malých pohonů v řádech jednotek Wattů, jako jsou například ventilátory.



Obr. 3-1: Ilustrační zapojení s poloviční můstkem, zdroj [1]

Při použití plného můstku získáme kvalitnější ovládání motoru, proto je také toto ovládání mnohem rozšířenější. Schéma zapojení je na obrázku Obr. 3-2. Toto zapojení a jeho ovládání se označuje jako „Six Step“. U třífázového plného můstku může být u každé fáze sepnut vždy pouze jeden tranzistor, druhý musí být zavřený, aby nedošlo ke zkratování zdroje. Mezi uzavřením jednoho a otevřením druhého tranzistoru je třeba nechat tzv. „deadtime“, což je doba potřebná pro úplné uzavření tranzistoru, aby nedošlo k prohoření větve.



Obr. 3-2: Zapojení s plným můstkem, zdroj [1]

3.4 Metody řízení BLDC motoru bez senzorů polohy

Bezsenzorové řízení BLDC motorů spočívá v řízení motoru bez použití senzorů polohy. Tímto přicházíme o znalost polohy rotoru, kterou je pak třeba určovat jinými způsoby. To má své výhody i nevýhody. Mezi výhody patří vysoká spolehlivost a odolnost proti rušení. Také zde nehrozí problémy s momentem vzniklé nepřesným osazením senzorů polohy. V neposlední řadě pak bezsenzorové řízení nezvyšuje velikost motoru a nezvyšuje množství vodičů, které vedou z motoru, čímž zabraňuje nechtěnému rušení. Také spolehlivě funguje i ve ztížených podmínkách, například při vysoké vlhkosti nebo teplotě, kdy mohou senzory polohy vykazovat značnou chybu. Nevýhodou je pak potřeba detekovat nastavení rotoru bez použití senzorů polohy.

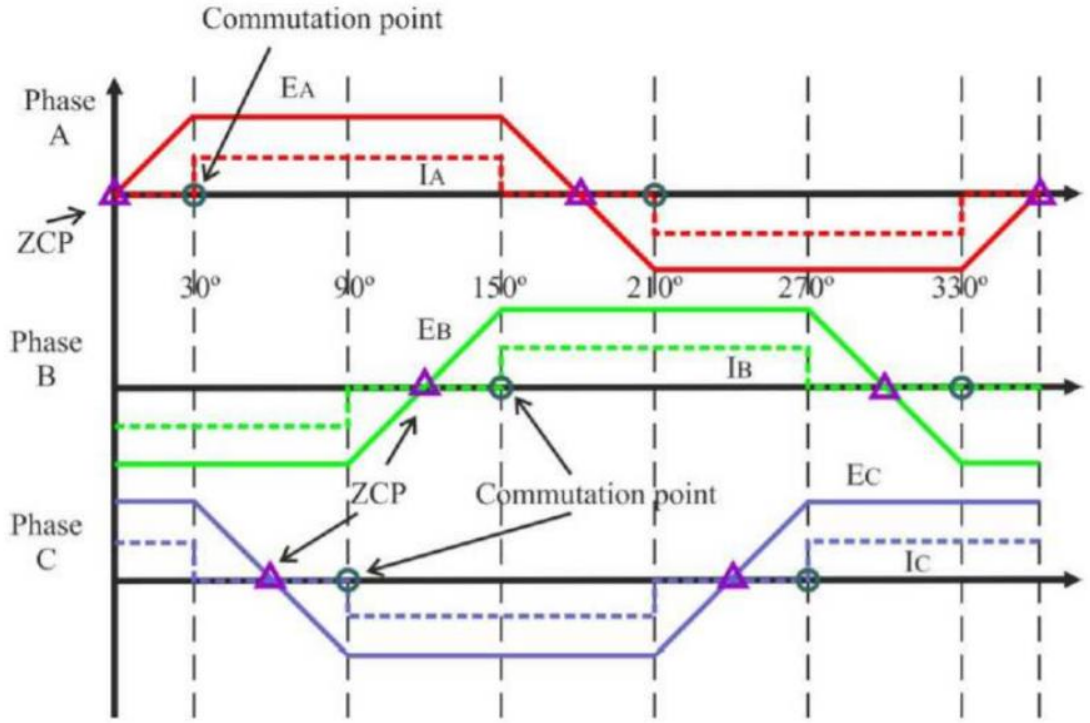
3.5 Přehled metod řízení bez senzorů polohy

Polohu rotoru lze určit několika metodami. Tyto metody jsou čerpány z odborné literatury. Také jsou využity praktické poznatky o jednotlivých metodách, získaných v diplomové práci pana Ing. Jakuba Křižana. [6]

3.5.1 Metoda detekce průchodu indukovaného napětí nulou

Tato metoda je jednou z jednodušších metod řízení BLDC motoru bez použití senzorů. Je založena na vlastnostech napětí a proudů v jednotlivých fázích BLDC motoru. Na obrázku Obr. 3-3 jsou tyto průběhy zobrazené. Průběhy ideálního napětí jsou vyznačeny plnými čarami, průběhy ideálních proudů pak přerušovanými čarami. Z obrázku je patrné, že každá fáze je připojena ke kladné sběrnici i záporné sběrnici stejně dlouho, a to 1/3 celkové doby jedné elektrické otáčky. Zbylou 1/3 doby je fáze odpojena od zdroje napětí. Během této doby dochází k průchodu napětí nulou, z obrázku je patrné, že se to děje vždy v polovině doby, kdy je fáze odpojena, tedy 30° elektrických od odpojení od sběrnice a nebo 30° elektrických před dobou komutace.

Pozn.: „Nulou“ se uvažuje potenciál středu vinutí.



Obr. 3-3: Průběhy napětí a proudů v BLDC motoru, zdroj [11]

Při řízení motoru touto metodou se tedy bude detekovat bod, kdy bude napětí vůči středovému bodu vinutí nulové. Tento okamžik se zaznamená, počká se 30° elektrických a poté se provede komutace.

3.5.2 Metoda spřaženého magnetického toku

Tato metoda zjišťuje polohu rotoru pomocí odhadování spřaženého magnetického toku. Z rovnice napětí

$$\boldsymbol{U} = \boldsymbol{RI} + \frac{d\boldsymbol{\psi}}{dt} \tag{3-2}$$

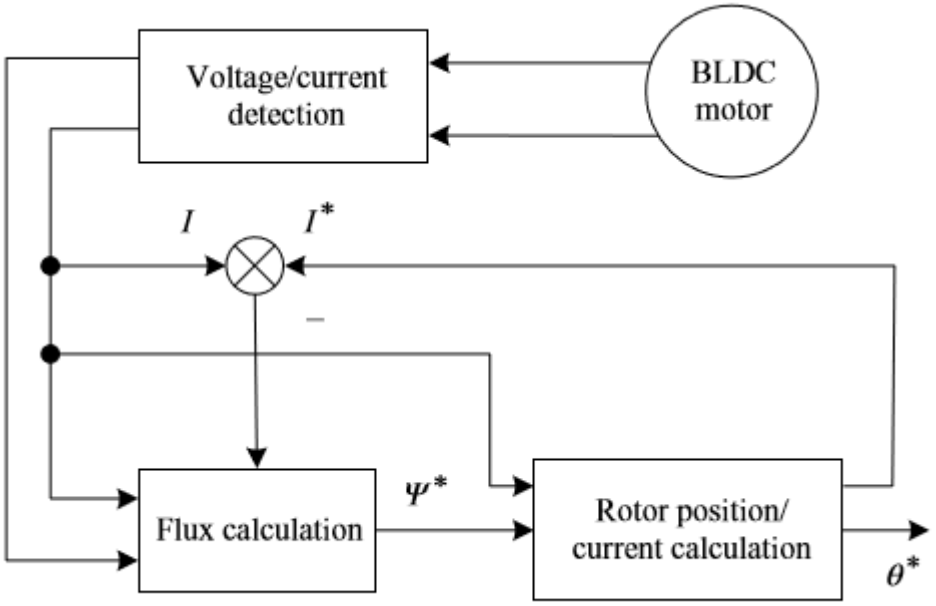
Lze snadno odvodit vztah pro velikost spřaženého magnetického toku. Je to

$$\boldsymbol{\psi} = \int_0^t (\boldsymbol{U} - \boldsymbol{RI})dt \tag{3-3}$$

Kde **U** je matice napětí všech fází, **I** je matice proudů fází a **R** je matice odporů vinutí. **Ψ** pak je matice spřažených toků.

Jestliže je znám vztah mezi natočením motoru a spřaženým magnetickým tokem v každé fázi, lze měřením napětí a proudů na jednotlivých fází dopočítat spřažený magnetický tok a určit tak pozici točícího se rotoru.

Při použití této metody je nutné znát počáteční polohu rotoru, abychom mohli určit integrační konstantu spřaženého magnetického toku. Nevýhodou této metody je, že je silně závislá na parametrech motoru, a navíc není vhodná pro měření nízkých otáček, kde může vznikat kumulativní chyba při integrování. [1]



Obr. 3-4: Princip metody vztaženého magnetického toku, zdroj [1]

3.5.3 Metoda integrace indukovaného napětí

Tato metoda využívá indukované napětí pro určení správného okamžiku komutace. V okamžiku průchodu indukovaného napětí nulou se začne toto napětí integrovat. Po dosažení určené hodnoty v integrátoru je pak provedena komutace a vynulování integrátoru. Využívá se vlastnosti indukovaného napětí, že plocha pod křivkou napětí je při všech rychlostech zhruba stejná (čím rychleji se motor točí, tím je strmější průběh indukovaného napětí, ale zároveň je i o to kratší časový úsek integrace). Není ani nutné

měřit úplně přesně bod průchodu nulou, protože možná odchylka je vyjádřena plochou okolo bodu nula, je tedy velmi malá.

Výhodou této metody je její možnost měřit za nízkých otáček. Naopak nevýhodou je její omezení při vyšších otáčkách.

3.5.4 Metoda integrace třetí harmonické

Metoda integrace třetí harmonické využívá třetí harmonickou indukovaného napětí k určení doby, kdy má dojít ke komutaci. Pomocí Fourierovy řady lze zapsat indukovaná napětí jednotlivých fází následovně:

$$e_A = E_1 \sin(\varphi_{el}) + E_3 \sin(3(\varphi_{el})) + E_5 \sin(5(\varphi_{el})) + \cdots$$
$$e_B = E_1 \sin\left(\varphi_{el} - \frac{2\pi}{3}\right) + E_3 \sin\left(3\left(\varphi_{el} - \frac{2\pi}{3}\right)\right) + E_5 \sin\left(5\left(\varphi_{el} - \frac{2\pi}{3}\right)\right) + \cdots \quad \textbf{(3-4)}$$
$$e_C = E_1 \sin\left(\varphi_{el} - \frac{4\pi}{3}\right) + E_3 \sin\left(3\left(\varphi_{el} - \frac{4\pi}{3}\right)\right) + E_5 \sin\left(5\left(\varphi_{el} - \frac{4\pi}{3}\right)\right) + \cdots$$

Kde:

e_A, e_B, e_C představují indukovaná napětí na jednotlivých fázích

E_1, E_3, E_5,… představují koeficienty členů fourierovy řady

φ_{el} představuje elektrický úhel natočení rotoru

Součtem těchto rovnic se od sebe odečtou všechny harmonické z předchozích rovnic, které nejsou násobky tří:

$$e_A + e_B + e_C = 3E_3 \sin(3(\varphi_{el})) + 3E_9 \sin(9(\varphi_{el})) + 3E_{15} \sin(15(\varphi_{el})) \quad \textbf{(3-5)}$$

kde lze kvůli velikosti zanedbat vyšší harmonické:

$$e_A + e_B + e_C \doteq 3E_3 \sin(3(\varphi_{el})) \quad \textbf{(3-6)}$$

Podle Kirchhoffova zákona platí:

$$i_A + i_B + i_C = 0 \tag{3-7}$$

A velikosti fázových napětí v motoru lze zapsat jako:

$$\begin{aligned} u_A &= Ri_A + (L - M)\frac{di_A}{dt} + e_A \\ u_B &= Ri_B + (L - M)\frac{di_B}{dt} + e_B \\ u_C &= Ri_C + (L - M)\frac{di_C}{dt} + e_C \end{aligned} \tag{3-8}$$

Sečtením těchto rovnic získáme:

$$\begin{aligned} u_{SUM} &= u_A + u_B + u_C \\ u_{SUM} &= \left(R + (L - M)\frac{d}{dt}\right)(i_A + i_B + i_C) + e_A + e_B + e_C \end{aligned} \tag{3-9}$$

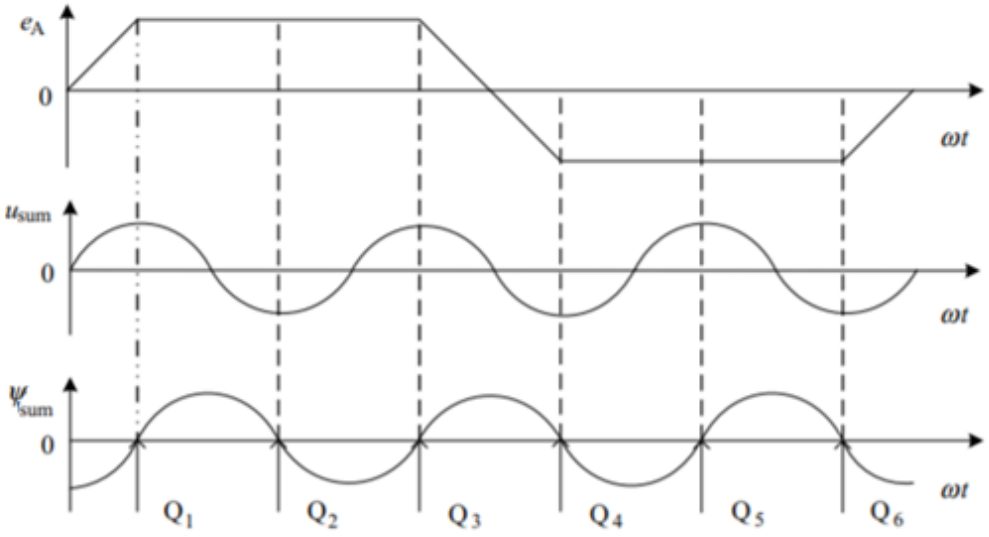
protože součet fázových proudů je nulový, získáváme:

$$u_{SUM} = e_A + e_B + e_C = 3E_3 \sin(3(\varphi_{el})) \tag{3-10}$$

Z této rovnice vyplývá, že suma fázových napětí odpovídá průběhu třetí harmonické. Komutace nastává v maximálních hodnotách funkce sinus, což by se detekovalo jen obtížně. Proto je provedena úprava rovnice. Po integraci získáme:

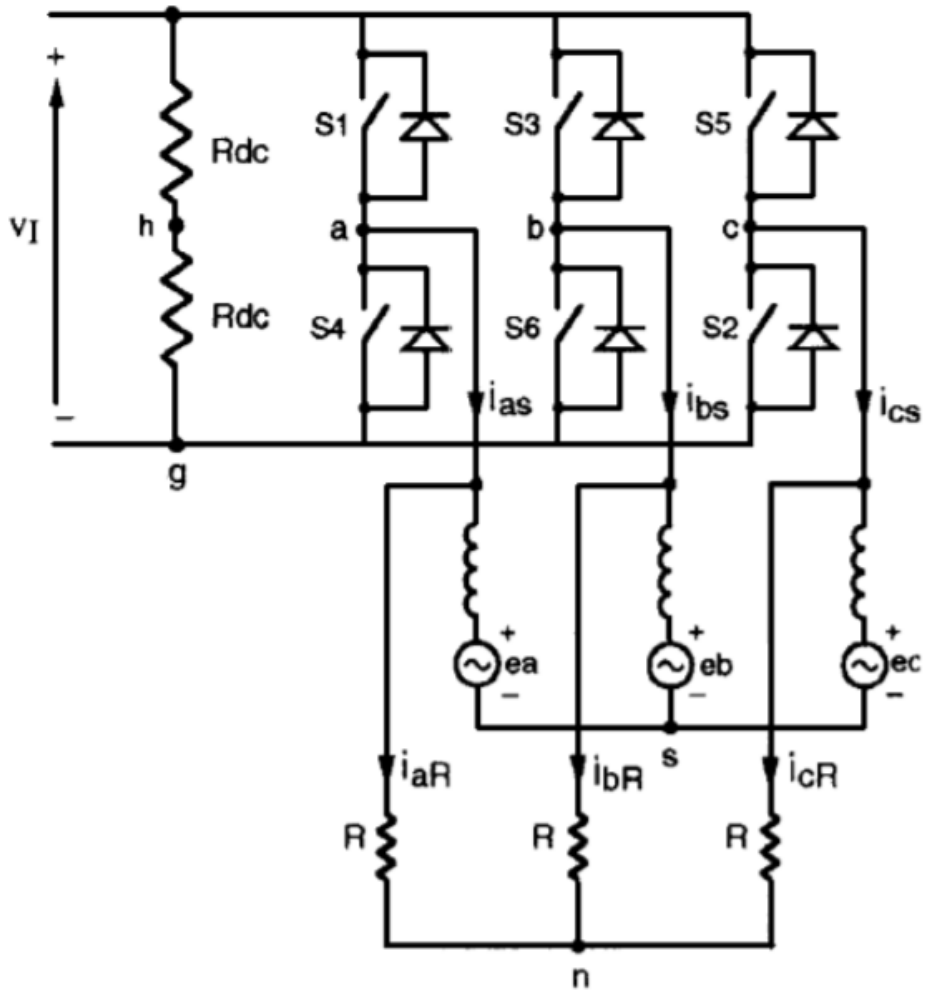
$$\psi_{SUM} = \int u_{SUM}\, dt \tag{3-11}$$

Integrací sumy fázových napětí dostaneme funkci posunutou o 90°, to znamená, že doba komutace nastává v momentech, kdy ψ_3 prochází nulou, což je pro detekci výhodné.



Obr. 3-5: Vysvětlení principu metody integrace třetí harmonické, zdroj [1], upraveno

Důležitou součástí této metody je určení napětí u_{sum}. Protože není k dispozici vyvedený střed vinutí, je třeba napětí měřit za pomocí přídavné rezistorové sítě zapojené do trojúhelníku. Napětí se bude měřit mezi body **n** a **h** na obrázku Obr. 3-6. Dále je třeba správně přiřadit body komutace k jednotlivým fázím – alespoň na jedné fázi je třeba snímat její průchod nulou za použití filtrace.



Obr. 3-6: Zapojení pomocné rezistorové sítě, zdroj [7]

Výhodou této metody je širší rozsah měřených rychlostí (obzvláště v nízkých otáčkách) bez nutnosti zpožďování signálu filtrem.

Naopak nevýhodou je neustálé integrování rušivého šumu při nízkých rychlostech, vedoucí k chybnému výsledku a následně nesprávnému určení doby komutace. Řešením tohoto problému je pravidelné resetování integrátoru, které ale vede k problémům se správným časováním. [1]

3.6 Zhodnocení kapitoly

Z popsaných metod se jako nejvhodnější jeví metoda detekce průchodu indukovaného napětí nulou. V praktickém užití této metody bude zapotřebí měřený signál filtrovat. Budou existovat určité minimální otáčky, pod kterými nebude možné tuto metodu spolehlivě používat. Maximální otáčky budou dány měřicím zařízením. V našem případě je filtrace provedena vhodným časováním vzorku.

4 POUŽITÝ HARDWARE A SOFTWARE

4.1 Volba použitého řídicího hardware

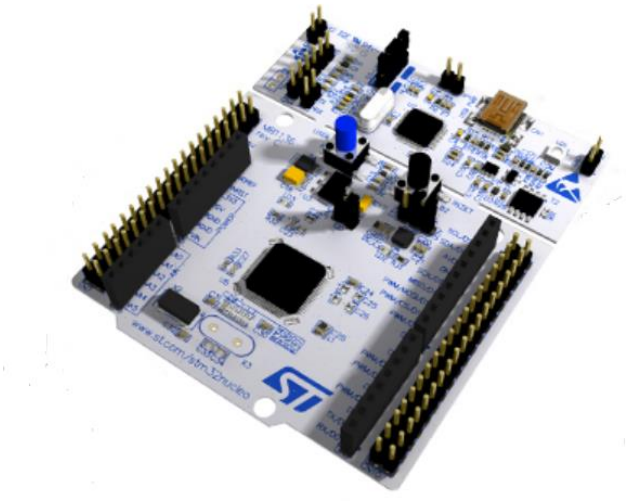
Pro implementaci řídícího algoritmu je použit mikrokontroler STM32 – ARM cortex M4. STM32 jsou 32bitové mikrokontrolery používající ARM jádro. Zkratka ARM znamená Advanced RISC Machine, což znamená jádro používající redukovaný počet instrukcí procesoru (Reduced Instructions Set Computing).

Procesory STM32 využívají Harvardskou architekturu, paměť programu a dat je tedy oddělená.

Důležitým bodem při rozhodování byla i dostupnost přípravku. Platforma je rozšířená a existuje k ní dostatek dokumentace. Další výhodou je cena – procesorovou desku lze koupit od částky několika set korun a vývojové prostředí je zdarma dostupné v plné verzi.

Konkrétní použitou procesorovou deskou je Nucleo F334R8. K ní měla být původně připojena rozšiřující deska se silovou částí pro napájení BLDC motoru IHM07M1. Pro tuto rozšiřující desku existuje od výrobce knihovna funkcí pro řízení BLDC motoru.

V rámci oživování byla dána přednost laboratornímu měniči microStand, který bylo možné ovládat precizněji. Konkrétně bylo možné ovládat všechny tranzistory měniče zvlášť, nezávisle na sobě a také nastavovat deadtime. Tyto možnosti u desky IHM07M1 nejsou dostupné, existující knihovna Middleware přepíná tranzistory podle PWM bez možnosti uživatelské korekce a deadtime se přepočítává a nastavuje automaticky, což nemusí být žádoucí. Další roli hrálo, že knihovna Middleware je určena pro konkrétní typ motoru a bylo by tedy nutné funkce upravovat pro naše použití.



Obr. 4-1: Deska Nucleo F334R8, zdroj [12], upraveno

4.2 Vlastnosti použitého hardware

Deska Nucleo F334R8 v sobě obsahuje 32,768kHz LSE krystalový oscilátor, 1 programovatelné tlačítko, 1 tlačítko reset, 1 LED diodu a FPU – floating point unit. Napájení je možné buď z externího laboratorního zdroje nebo pomocí USB portu. Deska dále obsahuje konektory Arudino™ Uno V3 a ST morpho pro připojení rozšiřujících desek. [10]

4.2.1 Floating point unit

Tento komponent jádra procesoru umožňuje rychlejší zpracování operací s čísly v plovoucí řádové čárce (floating point).

Čísla v tomto formátu se skládají ze znaménka, mantisy a exponentu. Při počítání s těmito čísly je třeba vykonat několika operací. Nejprve je nutné obě čísla zarovnat – upravit jejich exponenty na stejnou hodnotu. Poté lze provést matematickou operaci mezi čísly. Dále je třeba zaokrouhlit výsledek a zapsat jej. Všechny tyto operace jsou v procesoru bez FPU řešeny softwarově pomocí specializovaných funkcí.

V procesoru s FPU jsou tyto operace řešeny hardwarově, většina instrukcí je vykonána v jednom cyklu procesoru. Kompilátor tak pro tyto funkce nepoužívá knihovny, ale rovnou generuje instrukce pro FPU, což značně zvyšuje výpočetní výkon při počítání s čísly v plovoucí řádové čárce.[10]

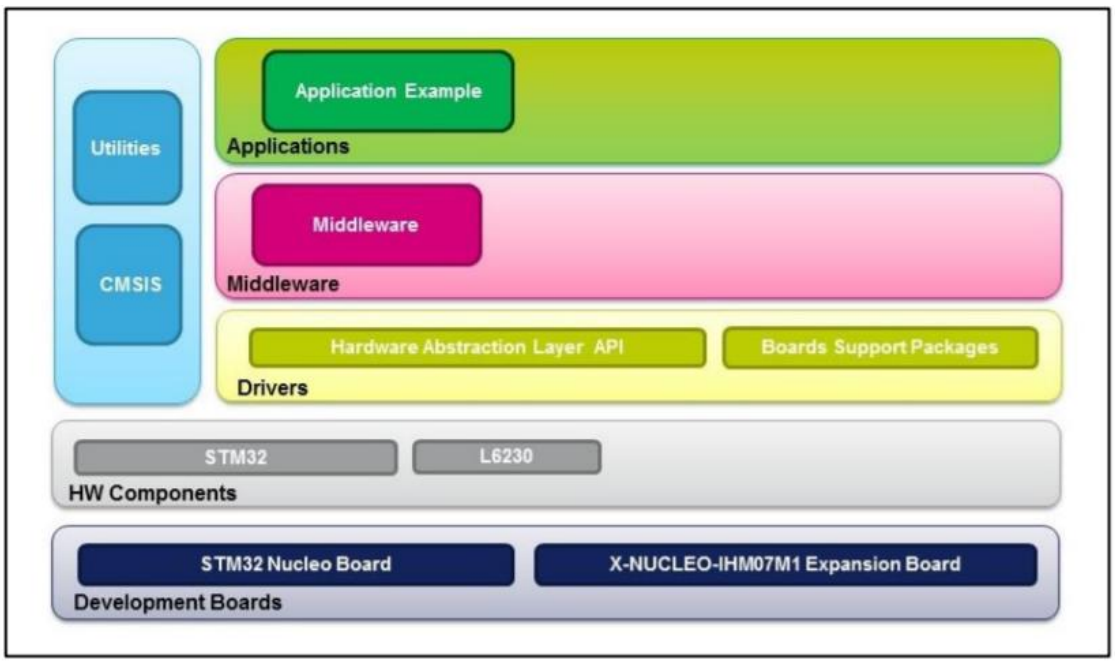
4.3 Použitý software

Pro programování desek byl využit software od výrobce Attolic TrueSTUDIO for STM32. Tento software umožňuje programování v jazycích C a C++.

Dále byl použit program STM32 Cube MX. Tento software je vytvořen speciálně pro programování mikrokontrolerů Nucleo. V programu Cube zadáme požadované nastavení periferií mikrokontroléru a následně je vygenerován kód v jazyce C, který lze použít jako základ programu. Vygenerovaný kód obsahuje příkazy, funkce a makra nastavující periferie do žádaného stavu. V kódu jsou vyhrazena místa pro vytvořené samotného programu uživatelem. Výhodou je, že nastavení lze kdykoliv změnit pomocí programu Cube. Po novém vygenerování kódu dojde pouze ke změně nastavení periferií, program vepsaný uživatelem zůstane zachován.

4.4 Návrh řízení pro platformu STM32

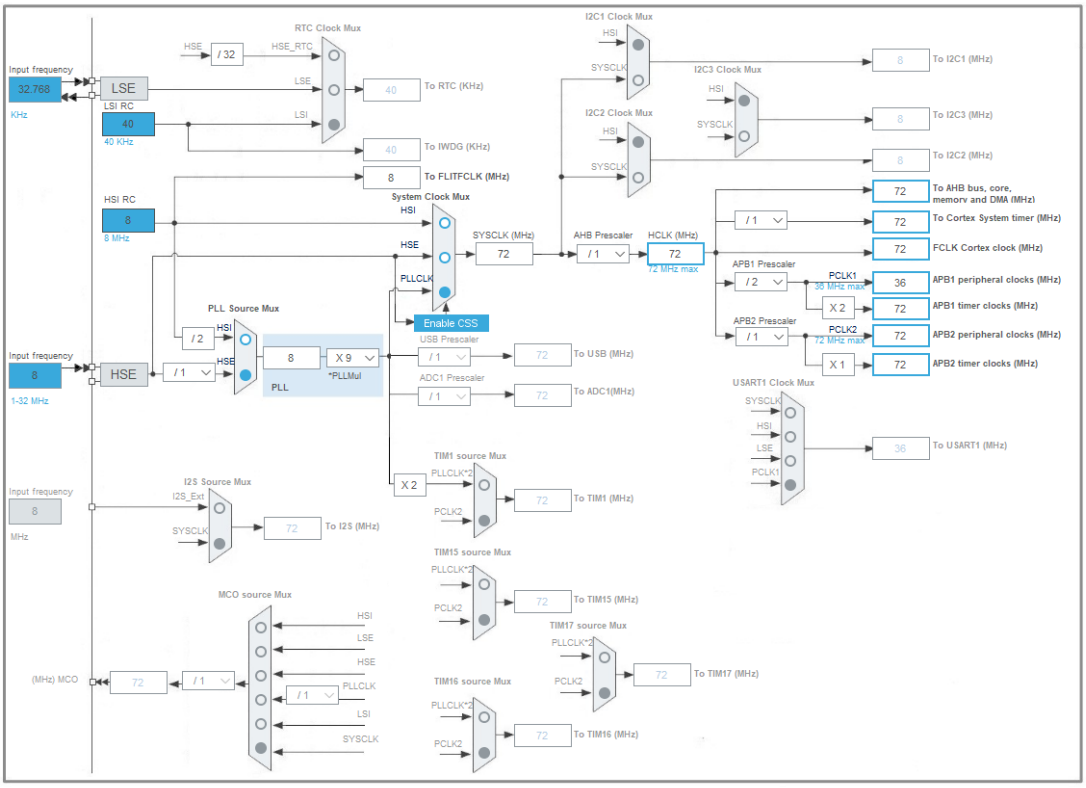
S využitím existujících knihoven bylo nastíněno fungování programu, který měl být v diplomové práci vytvořen. Architektura software od výrobce ST a jeho vrstvy jsou zobrazeny na obrázku Obr. 4-2.



Obr. 4-2: Architektura software X-CUBE-SPN7, zdroj [12]

Nejzákladnější vrstvou v architektuře jsou hardwarové součásti, tedy v tomto případě procesorová deska F334R8. Nad ní stojí vrstva ovladačů. O něco výše je pak vrstva knihoven. Knihovna HAL ovládá periferie, zatímco CMSIS je knihovna pro ARM, která zprostředkovává přístup k registrům za pomocí ukazatelů na ně. Middleware je knihovna, ve které jsou například funkce pro six-step ovládání motoru. Applications pak je přímo uživatelem vytvořený vlastní kód, v tomto případě tedy potenciální programování matematických operací konkrétních zvolených metod.

Z těchto vrstev byly využity v praxi pouze low level ovladače (LL) a knihovna HAL. Knihovna Middleware byla nahrazena uživatelsky vytvořenou knihovnou BLDClib.c, která v sobě obsahuje obecné funkce pro automatické vytvoření komutační tabulky a řízení six step. Tyto funkce jsou psány obecně a lze je použít pro jakýkoliv připojený BLDC motor.

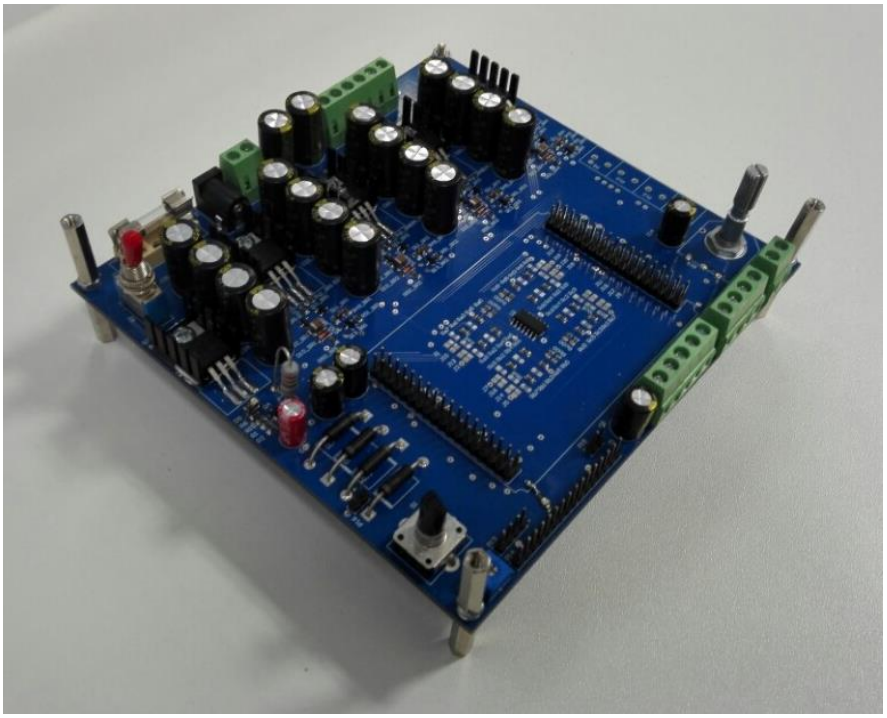


Obr. 4-3: Příklad nastavení hodinového signálu v programu STM32CubeMX

4.5 Měnič microStand

Jako měnič je použit laboratorní přípravek microStand. Je to univerzální měnič řízený pomocí mikrokontroléru. Původně byl vytvořen pro řízení mikrokontrolérem MC56F8257 od firmy NXP Semiconductor. Pro potřeby této práce byl mikrokontrolér nahrazen mikrokontrolérovou deskou Nucleo STM32 – ARM cortex M4. Propojení komponent bylo nejprve řešeno provizorně pomocí kabelů, později byla vytvořena pevná redukce propojující obě desky.

Měnič slouží pro napájení malých motorků s napětím 12 až 24V. Skládá se z výkonové a řídicí části. Výkonová část obsahuje čtyři tranzistorové půlmůstky a je možné na něm měřit napětí a proud. Řídicí část byla nahrazena deskou Nucleo F334R8.



Obr. 4-4: Laboratorní přípravek microStand

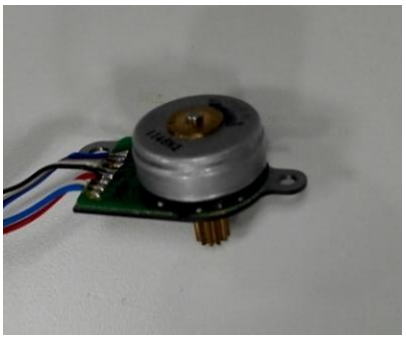
4.6 Použité motory

Pro vytváření řídícího programu byly použity dva motory. Nejprve byl použit starší motor vyrobený v Číně. Tento motor měl více pólů a bylo na něm oživeno řízení s Hallovými senzory. Později se ukázal problém s poměrem mezi spínací frekvencí

tranzistorů a frekvencí napětí ve fázi. Pro měření indukovaného napětí tedy bylo k dispozici málo vzorků během periody.

Čínský motor byl zhruba v polovině práce nahrazen švýcarským motorem Maxon 136222. Motor je dvoupólový, má tedy nízkou elektrickou rychlost a dá se na něm lépe měřit indukované napětí. Jeho rozměry jsou osová délka 70mm a průměr statoru 40mm.

Oba motory jsou osazeny Hallovými senzory, které mezi sebou svírají úhel 120°.



Obr. 4-5: Použitý čínský motor

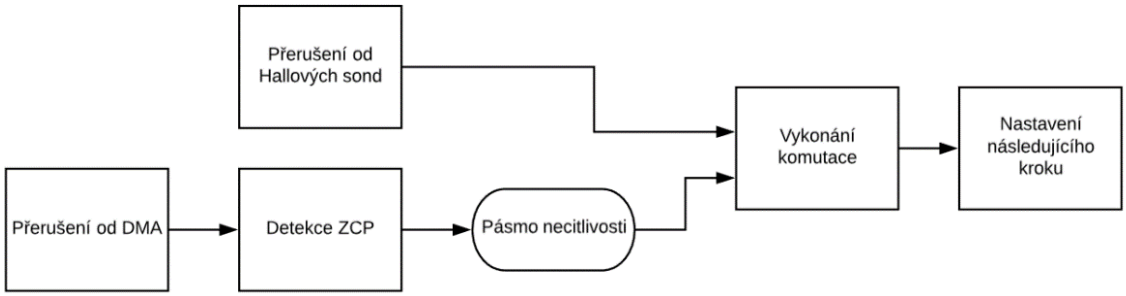
Obr. 4-6: Použitý motor Maxon 136222

5 REALIZACE ŘÍDICÍHO PROGRAMU

Při vytváření programu byl testovaný motor nejprve roztočen pomocí Hallových senzorů. V dalším kroku byl použit bezsenzorový algoritmus využivající detekci průchodu indukovaného napětí nulou. Bezsenzorové řízení využívá vysokofrekvenční přerušení (40kHz) od periferie DMA. Toto přerušení je vyvoláno v okamžiku, kdy jsou připravena data z měření AD převodníkem. V přerušení od DMA se také měří napětí na všech fázích a proud. Z měřených napětí se vypočte fázové indukované napětí, na kterém se provádí detekce průchodu nulou.

Pro potřeby našeho programu byla vytvořena knihovna BLDClib.c, ve které se nacházejí obecné funkce nízké úrovně pro řízení BLDC motoru. Dále byla vytvořena knihovna myapp.c, která obsahuje funkce nadřazené funkcím BLDClib.c. Knihovna myapp.c je specificky určena pro naši konkrétní aplikaci.

Blokové schéma fungování senzorového i bezsenzorového řízení je na obrázku Obr. 5-1.



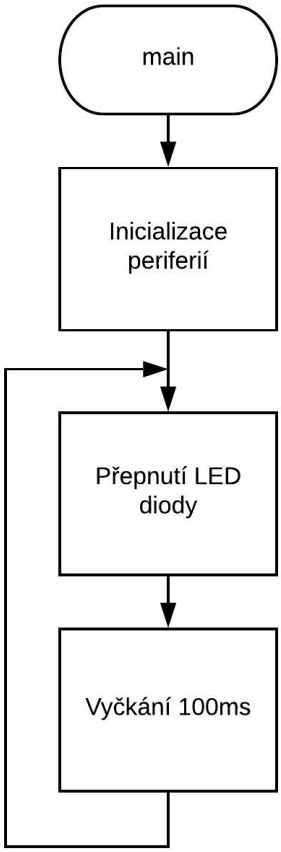
Obr. 5-1: Blokové schéma fungování senzorového a bezsenzorového řízení

V této kapitole je vysvětleno fungování nejdůležitějších funkcí z obou knihoven a funkce main(). Obě knihovny jsou přiloženy na konci dokumentu. Celý program se pak nachází na přiloženém CD.

5.1 Funkce main()

Funkce main zajišťuje pouze počáteční inicializaci některých periferií. Poté se vnoří do nekonečné smyčky, během níž pouze bliká LED dioda na mikrokontroléru. Blikání této diody signalizuje bezproblémový běh programu.

Během nekonečné smyčky dochází k pravidelným přerušením od periferií.



Obr. 5-2: Vývojový diagram funkce main()

5.2 Funkce myHallEdgeISR()

Funkce myHallEdgeISR() obsluhuje přerušení od Hallových sond. Tato funkce zajišťuje senzorové řízení motoru a nachází se v knihovně myapp.c. Protože se funkce

volá v závislosti na stavu Hallových sond, funkce prakticky pouze provede komutaci a následně uloží stav Hallových sond do proměnné hallindex.

Obr. 5-3: Vývojový diagram funkce myHallEdgeISR()

5.2.1 Čtení Hallových sond

Vyčítání stavu Hallových sond je nezbytné nejen pro senzorové řízení, ale využívá se i pro účely testování, při automatickém vytváření komutační tabulky a pro kontrolu funkčnosti bezsenzorového řízení.

Hallovy sondy jsou vyčítány pomocí čítače. Ten reaguje na náběžné i sestupné hrany kteréhokoliv z Hallových senzorů a v módu senzorového řízení při každé změně vyvolá přerušení. Toto přerušení obsluhuje právě funkce myHallEdgeISR().

5.3 Funkce commTabInit()

Vzhledem k tomu, že se program vytvářel a ladil na různých motorech, bylo třeba vytvořit komutační tabulky pro každý motor zvlášť. Proto bylo přistoupeno k vytvoření funkce comTabInit() v knihovně BLDClib.c, která komutační tabulku připojeného motoru vytvoří automaticky.

Komutační tabulka se vytvoří tak, že se připojí jedna fáze ke kladnému pólu zdroje, druhá a třetí fáze k zápornému pólu. Rotor se sám natočí dle polarity a jsou zapsány hodnoty na Hallových sondách. Následně se přepojí druhá fáze ze záporné na kladnou. Rotor se opět pootočí a opět jsou zapsány hodnoty z Hallových sond. Po vystřídání všech šesti kombinací zapojení je hotová tabulka natočení motoru v závislosti na přivedeném napětí. Abychom z této tabulky získali komutační tabulku, je třeba ji upravit, protože vektory v této tabulce jsou posunuty o 90° el vůči vektorům ve skutečné komutační tabulce.

Tabulka 2: Příklad vytváření komutační tabulky

Fáze			**Hallovy sondy**		
A	**B**	**C**	**a**	**b**	**c**
+	-	-	1	0	1
+	+	-	0	0	1
-	+	-	0	1	1
-	+	+	0	1	0
-	-	+	1	1	0
+	-	+	1	0	0

Úprava se provede tak, že se určí, kterou fázi je třeba sepnout pro otočení rotoru. Z tabulky v našem příkladu tedy vznikne komutační tabulka.

Tabulka 3: Příklad komutační tabulky

Hallovy sondy			**Fáze**		
a	**a**	**a**	**A**	**B**	**C**
1	0	1	Nepřipojená	+	-
0	0	1	-	+	Nepřipojená
0	1	1	-	Nepřipojená	+
0	1	0	Nepřipojená	-	+
1	1	0	+	-	Nepřipojená
1	0	0	+	Nepřipojená	-

Ve funkci je nejprve cyklus for pro šest stavů, následně se pak funkce dělí dle switche. V každém cyklu se tak nejprve přepne napětí na fázích a následně se pro toto natočení přečte stav Hallových sond. Z těchto dat se rovnou (bez mezikroku) tvoří výsledná komutační tabulka. Tato tabulka se pak zapisuje do globální proměnné bldcstep.

Nutno poznamenat, že během, této inicializace je třeba na fázi motoru přikládat takové malé napětí, které nezpůsobí průchod většího, než jmenovitého proudu vinutím stojícího motoru, ale dostatečné na to, aby se volný motor mohl protáčet. Pokud se motor nemůže volně protáčet a zastavovat v měřených polohách, komutační tabulka se detekuje chybně.

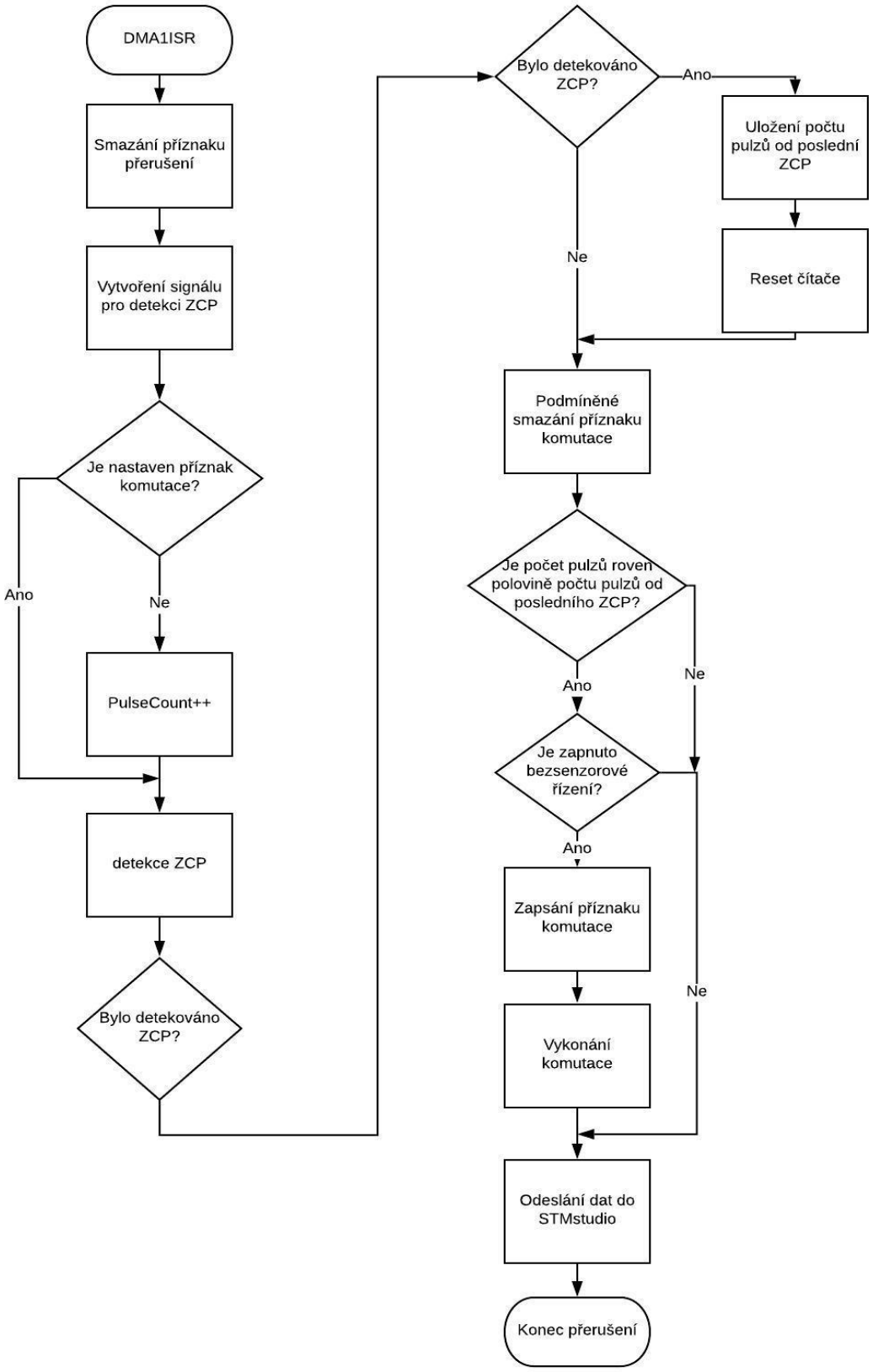
Vytváření komutační tabulky pomocí této funkce trvá několik sekund. Aby bylo urychleno spouštění programu při ladění, byly známé komutační tabulky pro použité motory uloženy do komentářů v programu a volání funkce při inicializaci bylo také zakomentováno. Při připojení nového motoru je tedy třeba odkomentovat volání této funkce ve funkci myInit().

```
void commTabInit(void)
{

	/*
	 * Commutation table initialization
	 */

	for (int i=1; i < 7; i++)
	{
		/*
		 * Turning the motor by switching the phases on and off
		 */

		switch(i)
		{
		case 1:

			LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1, LL_TIM_OCMODE_PWM1);
			LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH1 | LL_TIM_CHANNEL_CH1N);


			LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2, LL_TIM_OCMODE_FORCED_INACTIVE);
			LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH2 | LL_TIM_CHANNEL_CH2N);

			LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3, LL_TIM_OCMODE_FORCED_INACTIVE);
			LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH3 | LL_TIM_CHANNEL_CH3N);
			break;

                                        ***
		}
		LL_TIM_GenerateEvent_COM(TIM1);

		HAL_Delay(COMMTABDELAY);


		/*
		 * Filling the bldcstep field with the read values of Hall sensors transformed into the
commutation table
		 */

		//hallcommutation[i] = hallRead();
		//hallcommutation[hallRead()] = i ;
		bldcstep[hallRead()] = i;
	}
```

Výpis 5.1: Ukázka funkce comTabInit()

5.4 Funkce myDMA1ISR()

Funkce myDMA1ISR() je funkce knihovny myapp.c a zajišťuje obsluhu přerušení od DMA. V této funkci je prováděno měření napětí a nachází se zde hlavní část řešení bezsenzorového řízení. Na obrázku Obr. 5-4 je znázorněn značně zjednodušený vývojový diagram funkce. Kvůli své velikosti je diagram zhruba v polovině rozdělen a pokračuje opět odshora dolů.



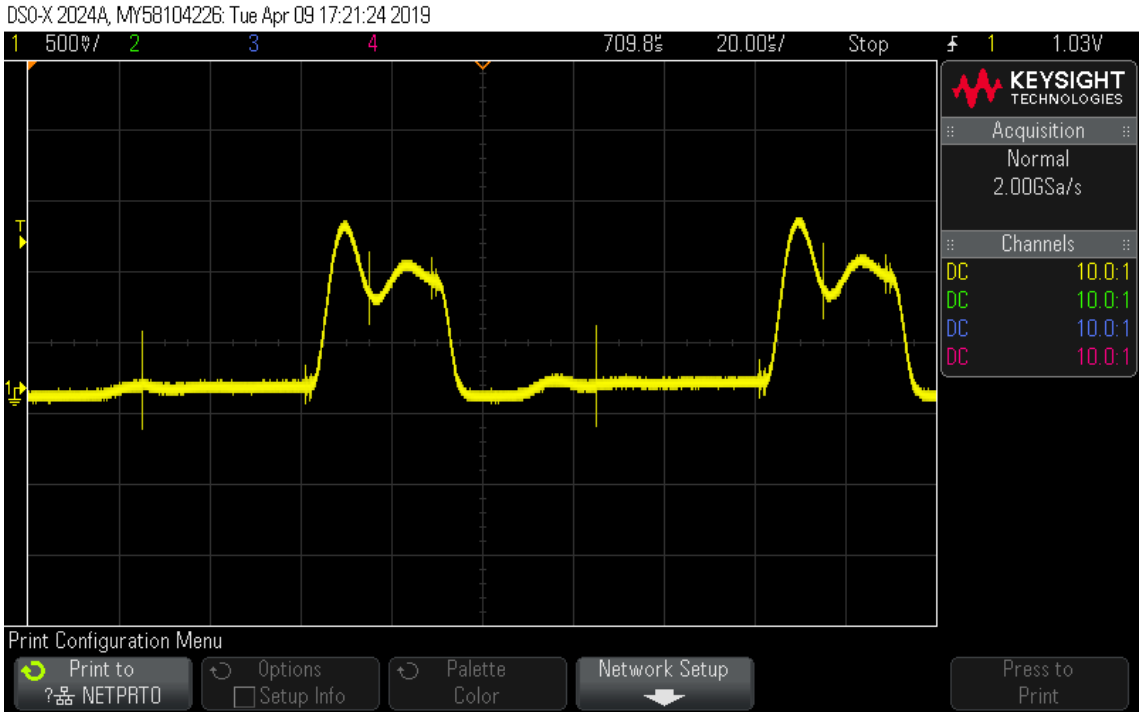
Obr. 5-4: Zjednodušený vývojový diagram funkce myDMA1ISR()

5.4.1 Měření napětí pomocí AD převodníku

Měření indukovaného napětí na jednotlivých fázích probíhá pomocí analogově digitálního převodníku (dále jen ADC), konkrétně ADC1. ADC je kalibrován pomocí vnitřní reference o hodnotě asi 1,2V. Tato hodnota není nicméně zcela přesná. Vnitřní reference se pohybuje v malém rozmezí a pro každý konkrétní čip je při výrobě přesně změřena a vepsána do paměti pouze pro čtení. ADC tedy může při měření zároveň měřit i hodnotu reference a podle ní kalibrovat ostatní naměřené hodnoty. Tento postup je výhodný v případech, kdy není zcela konstantní napájecí napětí, které se může s časem mírně měnit, například vlivem zahřívání desky. V naší aplikaci ovšem postačí spoléhat na přesnost zdroje 3,3V pro napájení ADC.

Vlastností ADC je, že při měření se nabíjení vzorkovacího kondenzátoru projevuje na měřeném signálu jehlovitým průběhem. Toho lze s výhodou využít při sledování signálu osciloskopem, kdy je možné přesně určit okamžik, kdy byl na signálu naměřen vzorek. Nicméně je třeba vhodným způsobem nastavit dostatečně dobu vzorkování, tedy připojení vnitřního vzorkovacího kondenzátoru k signálu. Tato doba by měla být větší než trojnásobek časové konstanty měřicího RC obvodu. V tomto obvodu je C kapacita vnitřního vzorkovacího kondenzátoru ADC a R je výstupní odpor děliče měřícího indukované napětí.

Na obrázku Obr. 5-5 je průběh indukovaného napětí na jedné fázi. Jsou zde jasně patrné čtyři velké jehlovité průběhy, což jsou právě okamžiky vzorkování. Drobné jehly na konci zvlnění jsou způsobeny parazitními jevy a rušením, které je způsobeno přepnutím jiné fáze. Zvlnění vzorkovaného signálu je způsobeno přechodovými jevy při změně napětí.



Obr. 5-5: Ukázka měření pomocí ADC

5.4.2 Rozdělení měřeného napětí na signály H a L

Ze čtyř viditelných vzorků na obrázku Obr. 5-5 byly vytvořeny dva signály. Signál H (High) a signál L (Low). Pro potřeby zpracování signálu je totiž zapotřebí, aby signál kopíroval průběh celkového napětí, ale PWM způsobuje napěťové skoky mezi každým vzorkem. V tomto případě se měří dva vzorky na jednu periodu PWM a následně jsou vždy oba vzorky ukládány odděleně, což vytváří dva různé signály. Pro malou velikost střídy je vhodnější používat signál L, protože u signálu H hrozí, že bude naměřen během přechodových dějů. Při velké střídě je vhodnější používat signál H, protože tento bude naměřen ustálený, zatímco signál L může být zarušen přechodovými ději. Před samotným použitím je třeba signál upravit. Signál L lze použít pro detekci bez úprav, zatímco od signálu H je třeba odečíst polovinu napětí meziobvodu, pak bude signál H procházet nulou přesně v polovině času mezi komutacemi.

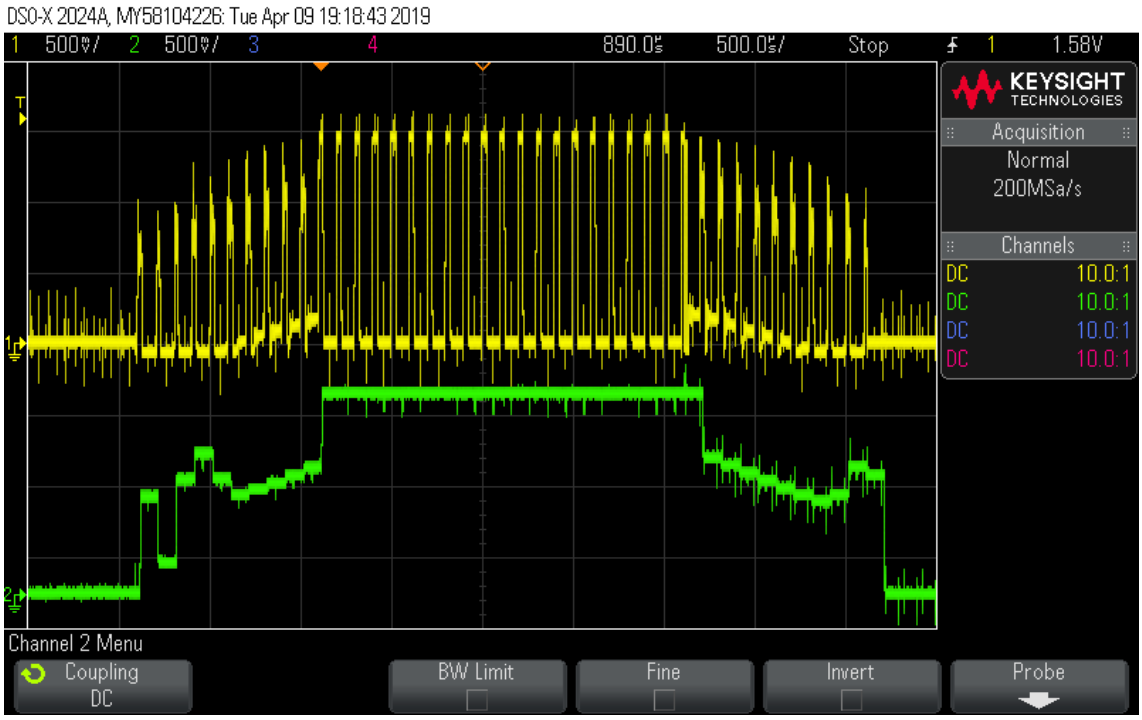
Rozdělení na signály probíhá pomocí funkce myDMA1ISR(), jejíž část je vidět ve výpisu Výpis 5.2. Tato funkce je vlastně obsluhou přerušení od DMA1.

```
void myDMA1ISR(void)
{

	if(LL_DMA_IsActiveFlag_TC1(DMA1))
		LL_DMA_ClearFlag_TC1(DMA1);
	//LL_TIM_COUNTERDIRECTION_DOWN;
	if(LL_TIM_GetDirection(TIM1) == LL_TIM_COUNTERDIRECTION_UP)
	{
		LL_DMA_SetMemoryAddress(DMA1, LL_DMA_CHANNEL_1,
(uint32_t)&ADC_Result[0]);
	}

	if(LL_TIM_GetDirection(TIM1) == LL_TIM_COUNTERDIRECTION_DOWN)
	{
		LL_DMA_SetMemoryAddress(DMA1, LL_DMA_CHANNEL_1,
(uint32_t)&ADC_Result[1]);
	}

			w		***
}
```

Výpis 5.2: Ukázka funkce myDMA1ISR()

5.4.3 Vizuální kontrola naměřeného napětí

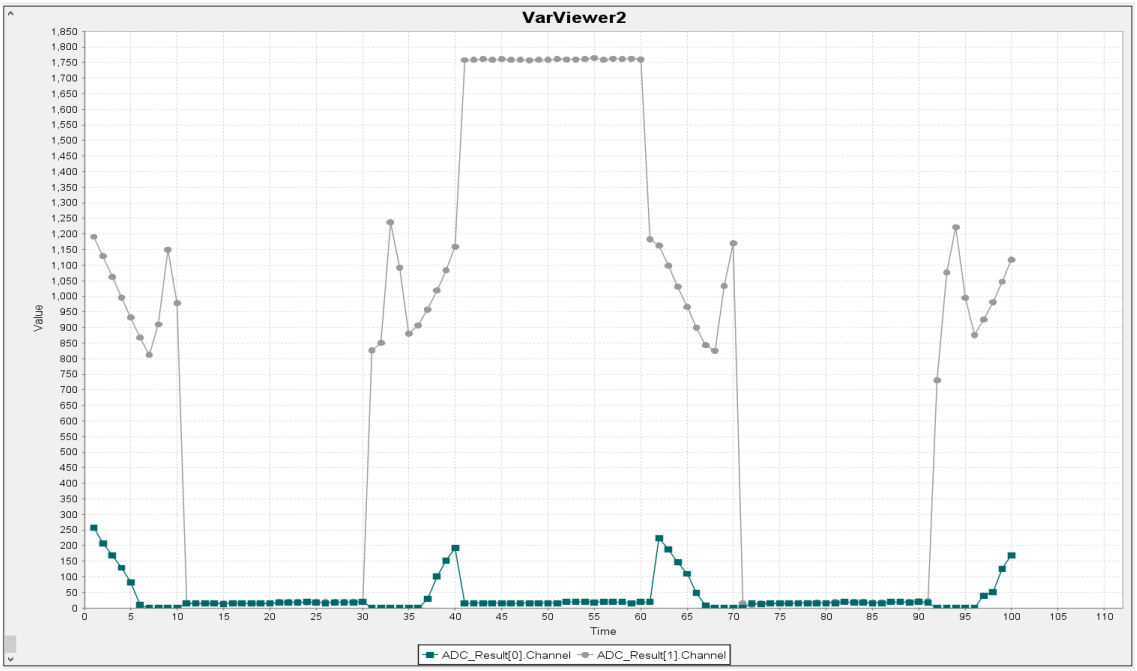
Pro zkontrolování správného změření napětí v okamžicích vzorkování bylo třeba zobrazovat navzorkovaný signál. To bylo možné dvěma způsoby. Prvním způsobem je měření osciloskopem. Aby se na osciloskopu zobrazovalo opravdu naměřené napětí, je třeba připojit sondu osciloskopu na výstup digitálně-analogového převodníku (dále jen DAC) desky. Na DAC jsou přivedena data přímo z ADC, bez jakékoliv úpravy. Na obrázku Obr. 5-6 je zachyceno toto řešení. Původní analogový signál je napětí fáze s patrnou PWM (žlutě). Tento signál byl navzorkován a následně předán na výstup DA převodníku (zeleně). Je patrné, že až na demagnetizaci na začátku a konci daného signálu je navzorkovaný průběh správný. Původ magnetizace/demagnetizace je ve vypínání nebo zapínání PWM jiné fáze.



Obr. 5-6: Ověření správnosti naměřeného napětí fáze

Druhou možností je využití program STMStudio, kterým lze zobrazovat (i graficky) vnitřní proměnné programu. Pro tyto účely byly do programu přidány předpřipravené knihovny v adresáři softTrace. Výsledný průběh je pak vidět na obrázku

Obr. 5-7. Je zde opět patrné, že kromě počátečního a koncového zvlnění je signál navzorkován velmi dobře. Je tedy možné měřit věechny fáze a následně začít zpracovávat signál.

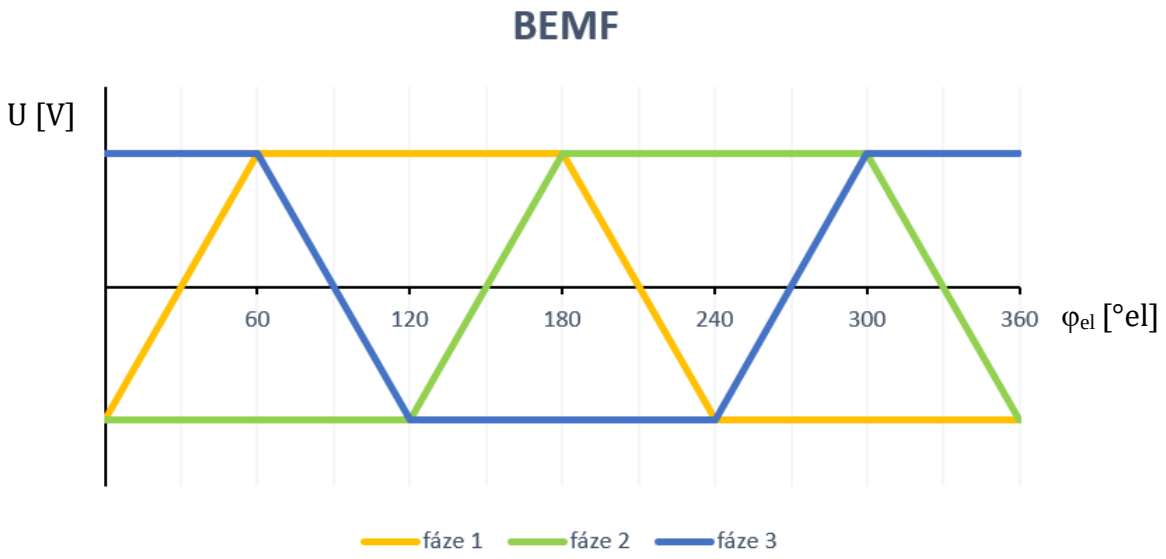


Obr. 5-7: Průběhy napětí na fázi připravené pro zpracování, šedý průběh je signál H, zelený průběh signál L

5.4.4 Detekování průchodu nulou

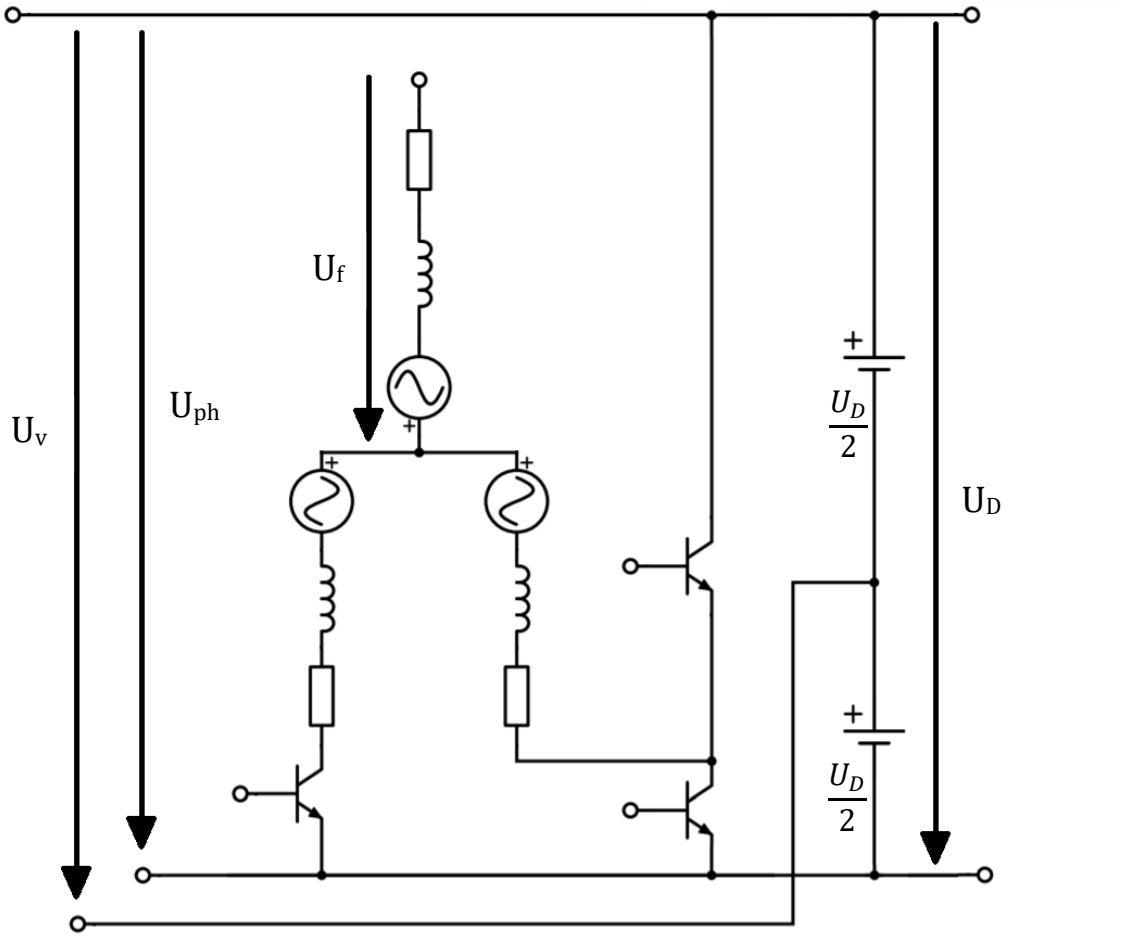
Aby bylo možné detekovat průchod nulou, stačí mít k dispozici jen malou část celkového signálu indukovaného napětí. Kritické části jsou dvě 60° široká pásma, kdy signál mění svoji hodnotu. Jak lze vidět na následujícím obrázku, v každém ze šesti pásem mění právě jeden signál svoji hodnotu. Toho lze výhodně využít k měření napětí pomocí pouze jednoho kanálu a po dosáhnutí konstantního signálu přepnout na následující fázi. Tento přístup zajistí, že všechny vzorky budou změřeny přesně v pravidelných intervalech, což by při použití více kanálů nebylo možné.

V algoritmu tedy bylo realizováno přenastavování měřícího kanálu za chodu programu, kdy v momentě komutace je kanál přepnut na následující fázi. Výsledný tvar naměřeného signálu je pila, u níž přesně víme, která její část náleží které fázi motoru, jak je vidět na obrázku Obr. 5-8.



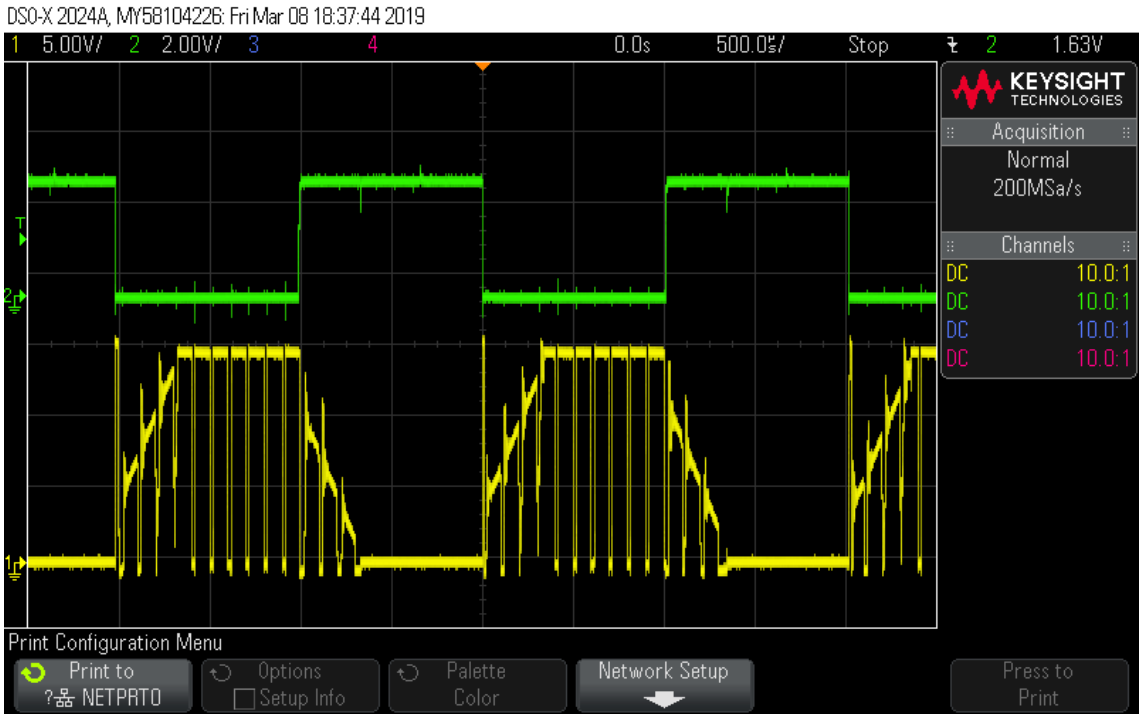
Obr. 5-8: Průběh indukovaného napětí na fázích motoru

Pro samotné zpracování signálu je třeba si uvědomit, co jsou naměřené hodnoty. Na obrázku Obr. 5-9 je zobrazeno náhradní schéma motoru.



Obr. 5-9: Náhradní schéma motoru, vysvětlení veličin

Napětí U_f je napětí fáze proti středu, který ale z motoru není vyveden. Reálně měřené napětí je na obrázku označeno jako U_{ph}. Toto je ale třeba přepočítat na hodnotu U_V, což je napětí, které lze v metodě použít. U_V lze vypočítat z U_{ph} přičtením $\frac{U_D}{2}$, kde U_D je napájecí napětí. Toto platí pro měřený signál L, měřený signál H odpovídá U_V. To je dáno spínáním tranzistorů.



Obr. 5-10: Ilustrační graf průběhu stavu jedné Hallovy sondy (zeleně) a napětí jedné fáze (žlutě) na čínském motoru

Reálný průběh naměřených hodnot je na obrázku Obr. 5-10. ZCP nastává v momentě, kdy se signál L zvýší na nenulovou hodnotu. Napětí nemůže nabývat záporné hodnoty kvůli antiparalelní diodě, místo toho je tedy nulové. Další možností je odečíst polovinu napájecího napětí od signálu H, a určit průchod nulou tohoto signálu.

Odečtení přesně poloviny napětí meziobvodu by bylo řešením v ideálním případě. Bohužel v reálné aplikaci toto pravidlo neplatí a bylo třeba přistoupit ke korekci tohoto napětí, aby odpovídalo signálu L. Eventuelně bylo přistoupeno na empirické zjištění přepočtového koeficientu. Opakovanou korekcí signálu a jeho zkoušením na celém rozsahu byla nakonec určena vhodná konstanta, kterou se odečítané napětí násobilo. Hodnota odečítaného napětí se pohybovala mezi 45 až 60% napětí meziobvodu a byla závislá na teplotě motoru.

Určování průchodu nulou bylo nejprve realizováno pomocí analogového watchdogu, který u každého naměřeného vzorku hlídal, zda nepřekročil určitou hranici napětí. V případě, že ano, vyvolal watchdog přerušení, ve kterém se spustil časovač. Tento časovač měl být nastaven pomocí střídy. V momentě, kdy odpočítávání časovače doběhne do konce, měla být vyvolána komutace. První problém se ukázal být ten, že

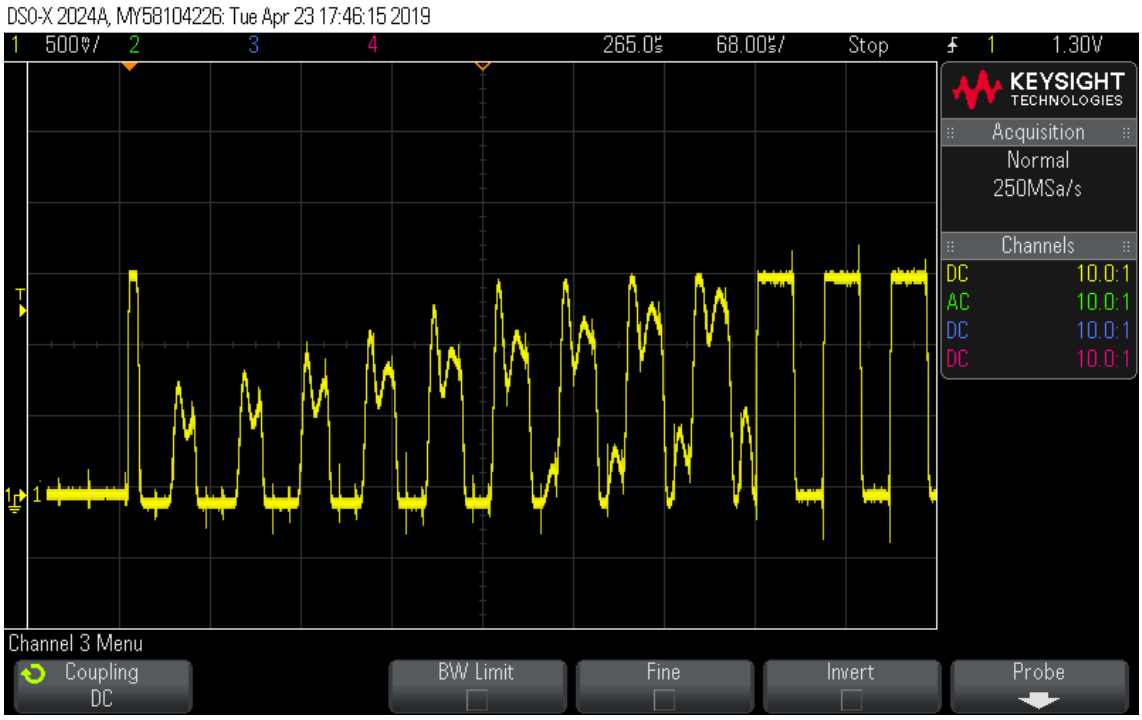
analogový watchdog nelze lehce přenastavovat pro naši potřebu za chodu programu. Tato obtíž nebyla nicméně nepřekonatelná.

Podstatný problém tohoto postupu spočíval v tom, že metoda je závislá na velikosti střídy. Čím vyšší střída, tím jsou změny signálu strmější a tím méně vzorků je naměřeno. Od určité velikosti střídy pak metoda již nelze použít, protože jednoduše není dost vzorků pro určení průchodu nulou.

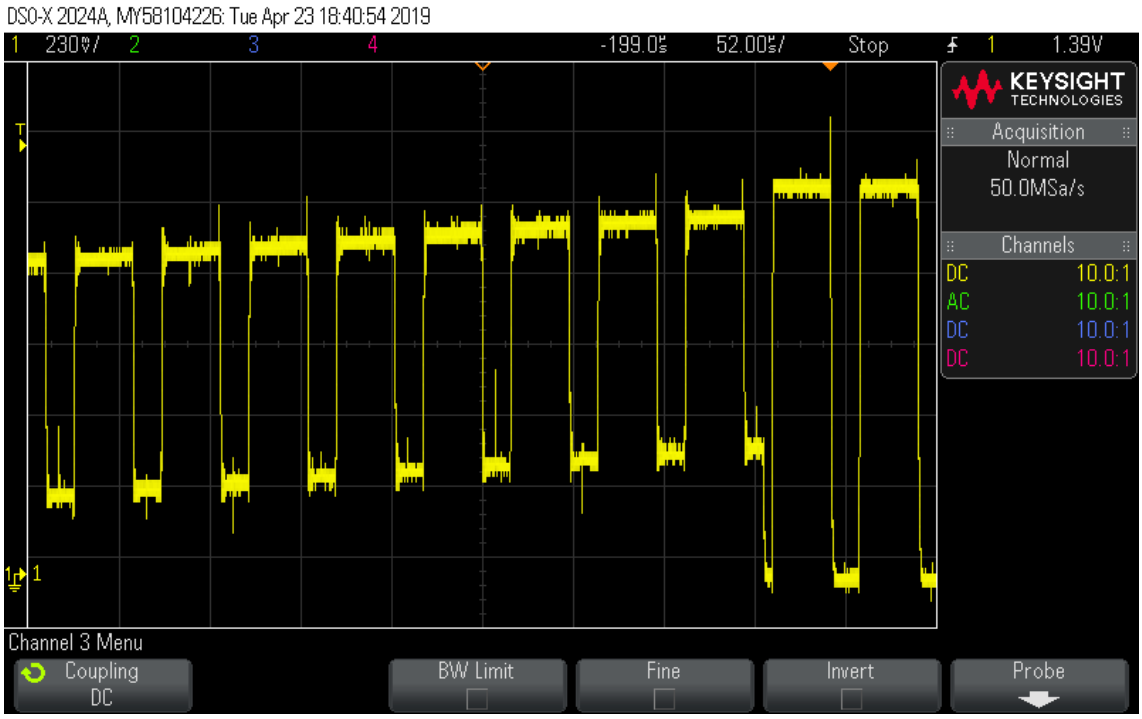
Řešením tohoto problému je zvýšit frekvenci PWM (pro tuto aplikaci byla původně stanovena na 10kHz). Při zdvojnásobení frekvence se zároveň zdvojnásobí počet vzorků, ale bohužel se tím sníží čas pro odeznění přechodových dějů v motoru. Bohužel, ADC nebyl schopen na této frekvenci spolehlivě měřit správné hodnoty vlivem neustálených přechodových dějů a celý program tak nefungoval. Proto bylo rozhodnuto, že tato metoda není pro naše použití vhodná.

5.4.5 Použití algoritmu na motoru Maxon

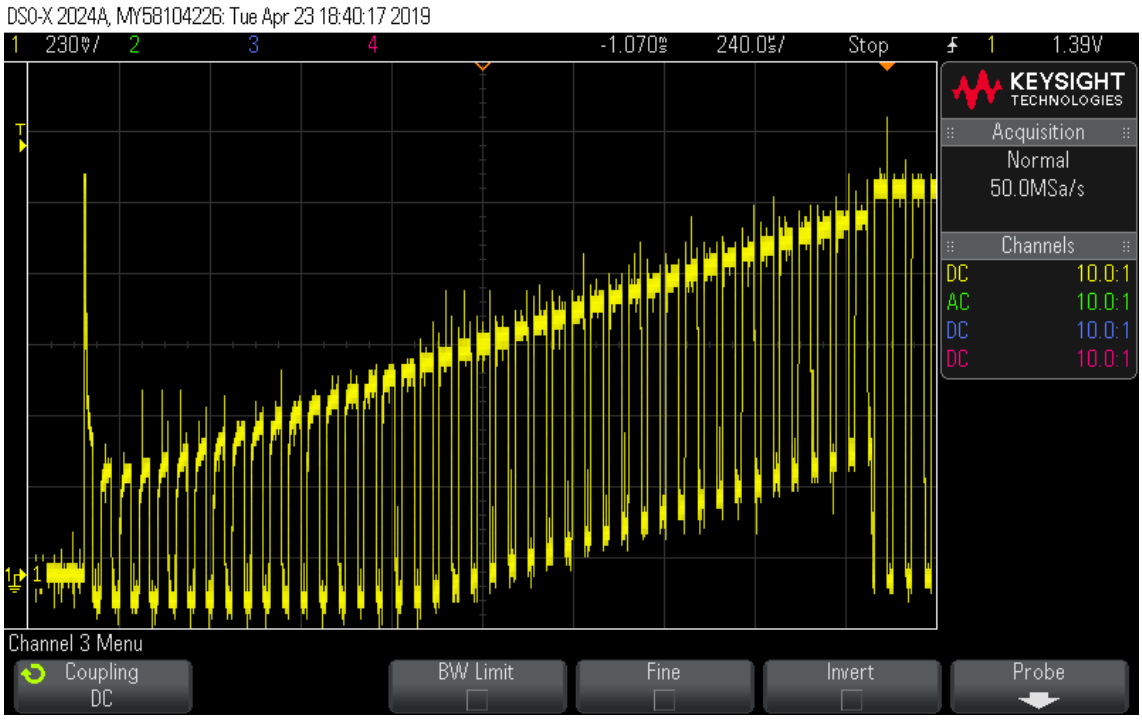
Průběhy napětí čínského motoru s výraznými přechodnými ději se ukázaly jako velmi obtížné na analýzu a zpracování. Protože byl program teprve ve fázi zprovozňování, byl vyměněn malý motor, na kterém se testovalo oživení, za motor Maxon 136222. Tento motor má menší počet pólů, a tím i nižší elektrickou rychlost při stejných mechanických otáčkách. To vede k vyššímu počtu vzorků a zároveň lepší detekci a celkové funkci programu i u vyšších stříd. Na následujících obrázcích je jasně vidět tento rozdíl.



Obr. 5-11: Průběh měřeného napětí u původního čínského motoru



Obr. 5-12: Detail měřeného napětí u motoru Maxon z Obr. 5-13



Obr. 5-13: Celkový průběh měřeného napětí u motoru Maxon

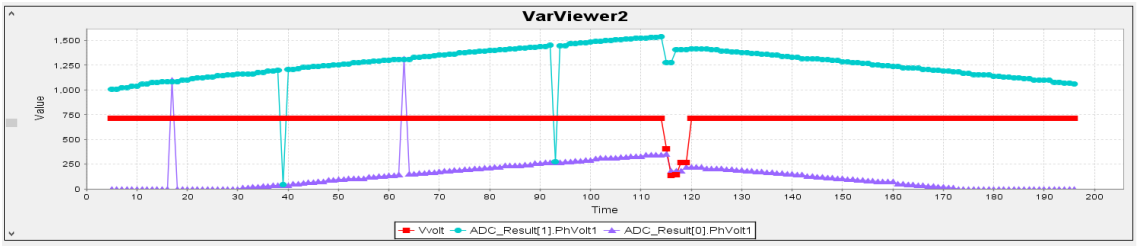
U motoru Maxon je jasně patrné, že přechodné jevy jsou téměř neznatelné.

5.4.6 Ošetření falešně pozitivních detekcí průchodu nulou

Při detekci průchodu napětí nulou se objevilo mnoho bodů, které algoritmus vyhodnotil jako ZCP, ale ve skutečnosti byly způsobené šumem, případně i nesprávným zápisem dat z ADC.

5.4.7 Špatně zapsaná data

Špatný zápis dat byl způsoben dlouhou dobou přenosu dat do programu STMstudio. Měření bylo realizováno v přerušení, avšak při zobrazování příliš mnoha proměnných v režimu snapshot byla doba čtení dat delší, než doba do nového zavolání tohoto přerušení. Z tohoto důvodu občas (náhodně) nedošlo k přenastavení ADC a tím byly zapsány vzorky ze signálu H do signálu L nebo naopak. Tento výsledek je vidět na obrázku Obr. 5-14. Protože k tomuto fenoménu docházelo zcela nepředvídatelně a náhodně, nebylo snadné zachytit tento jev na monitoru. Z tohoto důvodu je na obrázku i dočasně neužitečný signál (červeně).



Obr. 5-14: Ukázka propisování napětí ze signálu H (tyrkysový) do signálu L (fialový) a naopak

Řešení těchto potíží spočívá v omezení počtu proměnných v programu STMstudio a častějšího používání osciloskopu pro ladění programu.

5.4.8 Odstraňování rušení signálu

Pro potřeby algoritmu bylo třeba zbavit indukované napětí rušení a nesprávných hodnot (filtrovat). Byl vytvořen softwarový filtr signálu, který ovšem zkresloval užitečný signál. Dále byly vyzkoušeny metody podmíněného ignorování nevhodných vzorků, ovšem všechny tyto postupy se ukázaly jako příliš složité, nespolehlivé a pro aplikaci nevhodné.

Nakonec byl tento problém vyřešen pomocí dlouhého pásma necitlivosti v kombinaci s rampou, která omezuje změnu počtu pulzů mezi dvěma průchody nulou. Řešení v jazyce C je naznačeno ve výpisu Výpis 5.3.

Proměnná ZCPst se nastaví v momentě, kdy dojde k detekci průchodu nulou, proměnná CommSt pak při komutaci. Dohromady tedy zajistí to, že od první detekce průchodu nulou až po komutaci nemůže být detekován žádný jiný průchod nulou (falešně pozitivní). Proměnná CommSt pak tuto dobu necitlivosti prodlužuje i chvíli po komutaci, aby bylo zajištěno, že se na detekci neprojeví demagnetizace.

Rampa pro proměnnou PulseNum zajišťuje, že časový rozdíl mezi dvěma sousedními kroky Six-step řízení bude maximálně 5 pulzů PWM. To zajistí, že i při falešně pozitivní detekci motor nebude komutovat příliš brzo. Zároveň toto řešení ošetřuje skokovou změnu střídy na plynulou. Motor na novou střídu reaguje postupnou korekcí počtu pulzů mezi komutacemi. V programu je i rampa omezující změnu střídy pro odstranění prudkého škubání motoru při skokové změně rychlosti. Ze střídy 0 na střídu 1 motor najede za 1 sekundu.

```
void myDMA1ISR(void)
{
                      * * *
       if(ZCPst == 0)
       {
              if(Last <= 0)
              {
                     if(Vvolt > 0)
                     {
                            PulseNumLast = PulseNum;
                            PulseNum = PulseCount/2;
                            ZCPst = 1;
                            PulseCount = 0;
                     }
              }
                            * * *
       }

       if(PulseNum > PulseNumLast+5)
       {
              PulseNum = PulseNumLast+5;
       }
       if(PulseNum < PulseNumLast-5)
       {
              PulseNum = PulseNumLast-5;
       }


       if(CommSt > 0)
       {
              CommSt++;
       }
       if(CommSt > 20)
       {
              CommSt = 0;
              ZCPst = 0;
       }

       // commutation
       if(PulseNum == PulseCount)
       {
              CommSt = 1;
              makeSensorlessStep();
              if(Sensorless == 1)
              {
                     CommutationExecution();
              }
       }
                     * * *
}
```

Výpis 5.3: Ukázka funkce myDMA1ISR() – obsluha přerušení od DMA

5.5 Funkce BLDCsetStep2()

Tato funkce knihovny BLDClib.c zajišťuje generování PWM a ovládání tranzistorů měniče.

PWM je v programu vytvářena pomocí časovače (TIM1). Střída PWM se ukládá pomocí proměnné pulse. Střídě 1 odpovídá hodnota proměnné 3600. Pro uživatelskou přehlednost je zavedena proměnná *duty*, do které se zadává požadovaná střída v intervalu 0 až 1. Frekvence PWM v programu je 20kHz.

Funkce BLDCsetStep2() zároveň nastavuje žádané natočení rotoru. Na začátku funkce je dělení šesti se zbytkem (modulo 6) a funkce je pak pomocí zbytku po dělení rozdělena na šest případů pomocí switche. Šest stavů je zde kvůli six-step řízení. Výhodou je, že nezáleží na počtu pólů připojeného motoru, funkce pracuje s elektrickými otáčkami a díky dělení se zbytkem následuje po šestém případu opět ten první, proměnná tak nepřetéká a motor se plynule točí. Samotná komutace je volána pomocí funkce CommutationExecution().

Drobná matematická korekce modulo 6 je ve funkci proto, aby výsledek byl v intervalu 1 až 6, protože při použití tohoto dělení obdržíme výsledek mezi 0 až 5.

```
int BLDCsetStep2(int s)
{
	s = ((s - 1) % 6) + 1;
	switch(s)
	{
	case 1:
		//Disable both transistors
		LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1,
LL_TIM_OCMODE_PWM1);
		LL_TIM_CC_DisableChannel(TIM1, LL_TIM_CHANNEL_CH1N |
LL_TIM_CHANNEL_CH1);

		//Generate pwm signal and its negation
		LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2,
LL_TIM_OCMODE_PWM1);
		LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH2 |
LL_TIM_CHANNEL_CH2N);

		//Turn on bottom transistor
		LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3,
LL_TIM_OCMODE_FORCED_INACTIVE);
		LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH3 |
LL_TIM_CHANNEL_CH3N);
		break;

				***

	default:
		// shutdown PWM

		break;

	}

	return s;
}
```

Výpis 5.4: Ukázka funkce BLDCsetStep2()

5.6 Funkce initSensorlessStep()

Tato funkce knihovny BLDClib.c umožňuje beznárazové přepínání na bezsenzorové řízení za chodu motoru a bezsenzorový rozběh motoru.

5.6.1 Beznárazové přepínání na bezsenzorové řízení za chodu motoru

Laděný program byl vytvořen pro řízení s pomocí senzorů polohy. Proto byla pro bezsenzorové řízení vytvořena proměnná SensorlessStep, která simuluje použití senzorů polohy. Tato proměnná se inkrementuje s každou detekcí průchodu nulou a v případě, že je zapnuto bezsenzorové řízení, přebírá funkci proměnné hallindex, která uchovává informaci o stavu Hallových sond.

Pro beznárazové přepínání je třeba proměnnou inicializovat. Jestliže je inicializace provedena během nastavení senzorového řízení, pak je do proměnné vepsán aktuální stav Hallových senzorů a od tohoto bodu dále je udržována informace o poloze rotoru virtuálně algoritmem. V případě zastavení motoru algoritmus náhodně detekuje šum na vypnutých fázích a virtuální informace o natočení rotoru přestane být aktuální. Funkce také resetuje pomocné proměnné pro bezsenzorový chod.

5.6.2 Bezsenzorový rozběh motoru

V programu je řešeno i bezsenzorové roztáčení motoru. Při rozběhu se výhodně využívá právě rušení, které nebylo z programu odstraněno. Při vypnutém motoru na fázích není dokonale nulové napětí, šum způsobuje neustálé náhodné zvyšování a snižování hodnoty signálu v malém rozmezí okolo nuly. Tyto změny program vyhodnocuje jako průchody nulou (falešně pozitivní). V případě, že je nastavena nulová střída program na tyto falešně pozitivní výsledky nijak nereaguje. V případě, že je hodnota střídy nenulová, program provede po určité době komutaci. Tato doba je dána proměnnou PulseNum, a je určena z reálného zpomalování motoru. Díky rampě, která omezuje změnu proměnné PulseNum na +/- 5 pulzů za jednu komutaci je následující komutace provedena za téměř stejnou dobu od průchodu nulou jako ta předchozí. Tyto počáteční komutace vzniklé detekováním šumu tak začnou pravidelně roztáčet motor a již po několika (řádově jednotky) komutacích se motor začne točit dostatečně rychle pro detekci reálných průchodů nulou. Poté začne algoritmus korektně pracovat a najede na rychlost odpovídající požadované střídě.

Pro funkčnost tohoto startu je zapotřebí, aby všechny proměnné ovládající bezsenzorové řízení motoru byly inicializovány. Toho je dosaženo tak, že po resetu

programu je volána funkce initSensorlessStep(). Jestliže je nastaveno bezsenzorové řízení (nastavuje se po resetu automaticky), provede se několik komutací, které algoritmu umožní se správně nastavit. Tyto komutace nejsou volány podle stavu Hallových senzorů ani podle bezsenzorového řízení, ale v konstantních časových intervalech. Pro zabránění zpětného cuknutí rotoru je na začátku funkce přečten stav Hallových sond, což ale pro funkčnost inicializace není nezbytné.

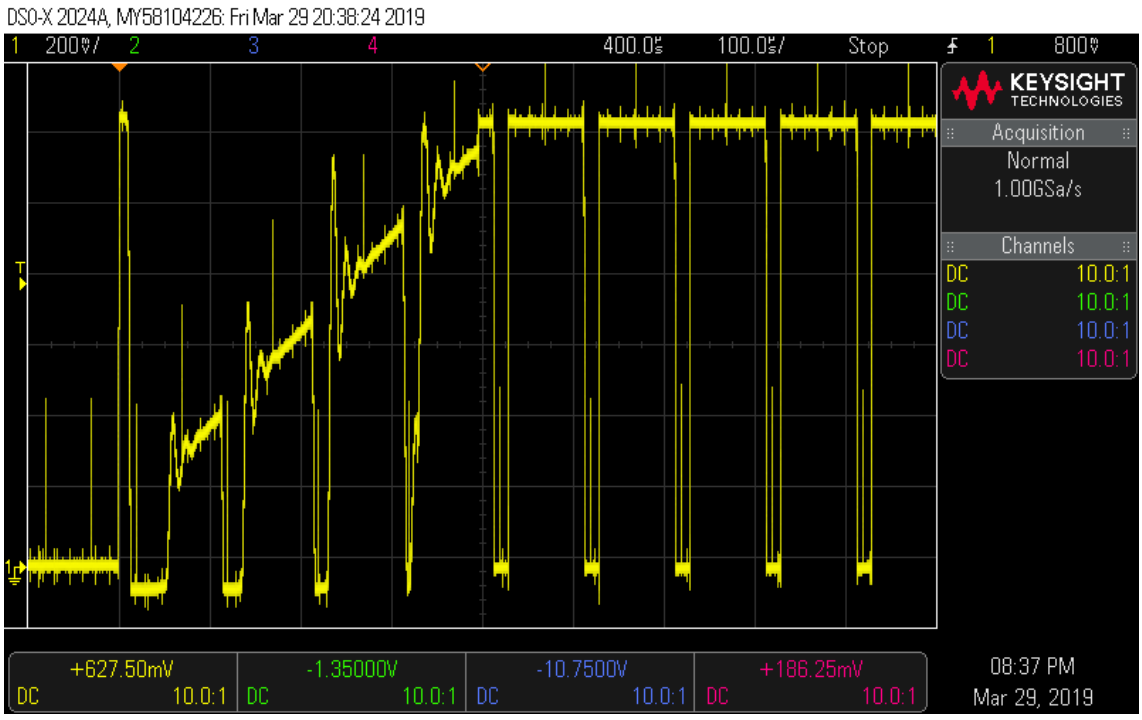
6 PRAKTICKÉ PROBLÉMY PŘI REALIZACI

6.1 Časování PWM

Během programování došlo k problému, kdy při spuštění motoru byl na osciloskopu pozorován propad těsně před ustálením napětí. Tento propad byl způsoben nevhodným časováním jednotlivých funkcí a nevhodným přepínáním stavů kanálu. Propad je jasně patrný na obrázku Obr. 6-1.

Problém spočíval v tom, že po dosažení maximální hodnoty napětí byl vyslán signál pro maskování PWM. Tímto se kanál přepnul z neaktivního režimu do režimu PWM, ale protože toto přepnutí nebylo synchronizováno, byla střída automaticky nastavena na hodnotu 0. Správná hodnota střídy byla nastavena až při dalším cyklu PWM, což mělo za následek krátkou dobu, kdy byla fáze odpojena od napětí v momentě, kdy měla být připojena. Tento krátký výpadek se negativně projevoval na plynulosti chodu motoru.

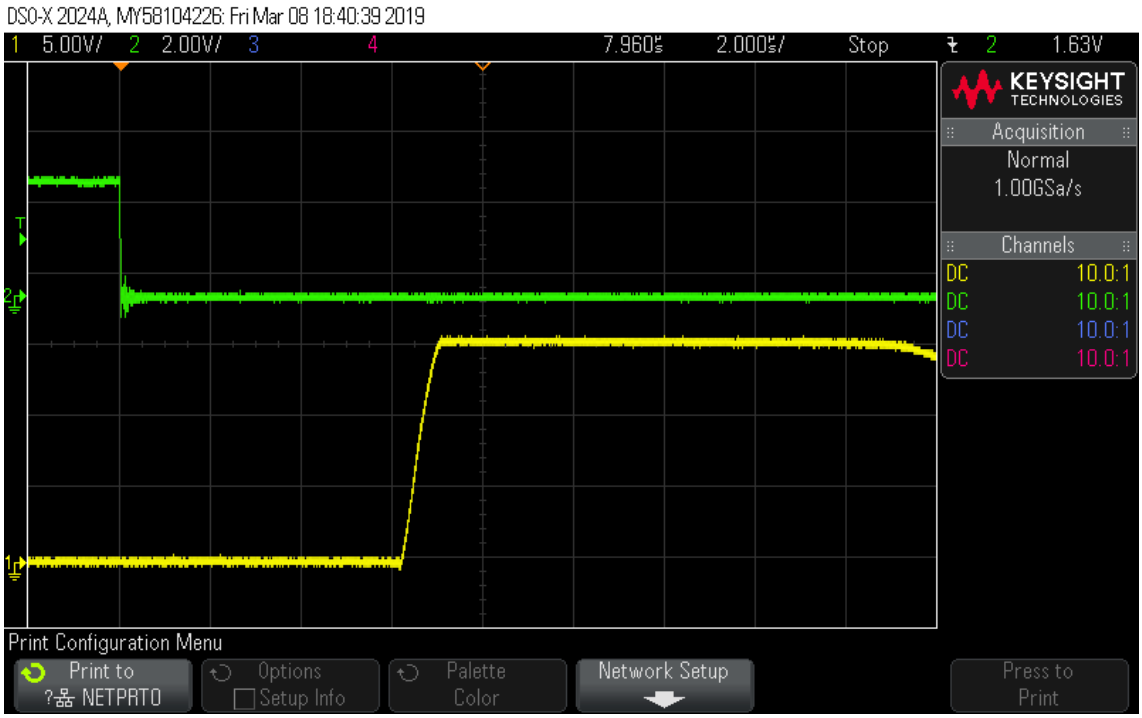
Problém byl nakonec vyřešen tak, že na kanálu je natrvalo nastaven PWM režim a výstup kanálu se dle potřeby pouze připojuje nebo odpojuje, odpadá tak nutná inicializace při přepnutí režimu.



Obr. 6-1: Ukázka problému s časováním PWM

6.2 Zpoždění komutace oproti hraně signálu z Hallových sond

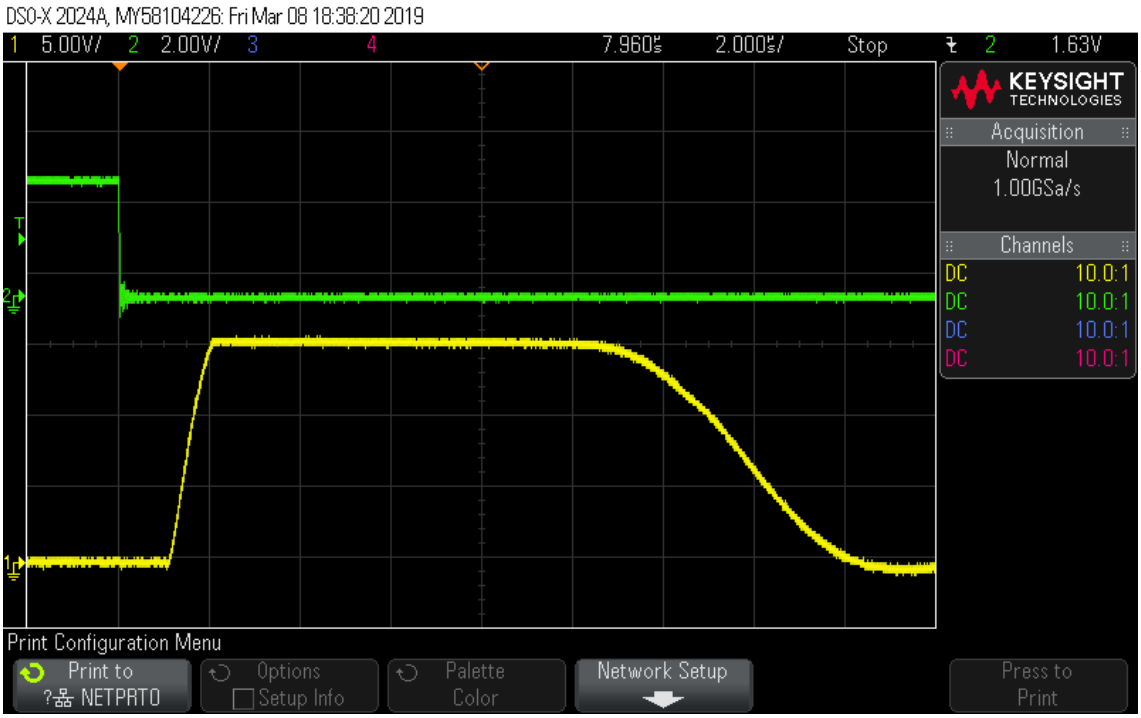
Další drobným nedostatkem byla nepřesná komutace v reakci na změnu výstupu Hallovy sondy. Z obrázku je patrné, že komutace nastává až asi 5µs po změně stavu senzoru.



Obr. 6-2: Velké zpoždění komutace oproti změně z Hallovy sondy

Aby bylo možné nastavit komutaci přesně na hranu z Hallovy sondy, bylo třeba optimalizovat program. Původní problém byl v času, který program potřeboval pro výpočet následujícího kroku. Procesor zahájil přepočet přesně v okamžiku, kdy mělo dojít ke komutaci, což způsobovalo zpoždění. Z tohoto důvod byl program změněn, byla přidána predikce příštího stavu (komutační tabulka se načítá pro následující stav, aby se při příští komutaci rovnou mohla použít a nebylo potřeba čekat na výpočet). Těmito úpravami bylo dosaženo zpoždění pouze 1µs.

Ideální řešení problému by bylo spouštět událost komutace hardwarově a nečekat na softwarové přerušení. Protože ale cílem práce bylo vytvořit bezsenzorové řízení, které hardwarově spouštěnou událost nemůže používat, byla použita softwarová optimalizace i pro řízení s Hallovými sondami.



Obr. 6-3: Optimalizované zpoždění komutace.

6.3 Měření napětí

Další problém, který bylo třeba řešit se vyskytl při měření napětí na všech třech fázích. Původní rozvržení bylo takové, že ADC měl pro každou fázi zvláštní kanál (rank) a měřil tak všechny tři fáze při každém průchodu. To ovšem vedlo k tomu, že fáze nebyly měřeny ve stejném bodě periody. Řešení tohoto problému je takové, že ADC používá pouze jeden kanál, který vhodně přepíná pro měření všech tří fází. Informaci o tom, kterou fázi je třeba aktuálně měřit pak program získává z údajů o komutaci.

6.4 Demagnetizace

Další potíž při zpracování signálu se ukázala být demagnetizace. Na obrázku Obr. 5-11 je zřetelně patrné, že na začátku náběžné hrany indukovaného napětí je napětí na okamžik rovno maximálnímu. U sestupné hrany je toto napětí naopak na okamžik nulové. Tyto skoky a propady jsou způsobeny demagnetizací. Jakékoliv naměřené vzorky v tomto pásmu jsou nesprávné. Je proto nutné nastavit, aby po každé komutaci bylo několik vzorků zahozeno, než je jistota, že demagnetizace již neovlivňuje měření. Při vyšším proudu (vyšší střída nebo při zatížení motoru momentem) se demagnetizace prodlužuje.

7 SROVNÁNÍ SENZOROVÉHO A BEZSENZOROVÉHO ŘÍZENÍ

Motor byl spuštěn pomocí senzorů polohy, a také bezsenzorovu metodou pomocí detekce průchodu nulou. Byly změřeny průběhy indukovaných napětí na fázích a průběh proudu. Velikost střídy při měření byla 50 %. Na všech obrázcích je proud (žlutě) měřen na první fázi, jejíž indukované napětí je na obrázcích zeleně. Všechna napětí jsou měřena na vstupu AD převodníku (tj. za napěťovým děličem) proti zemi řízení.

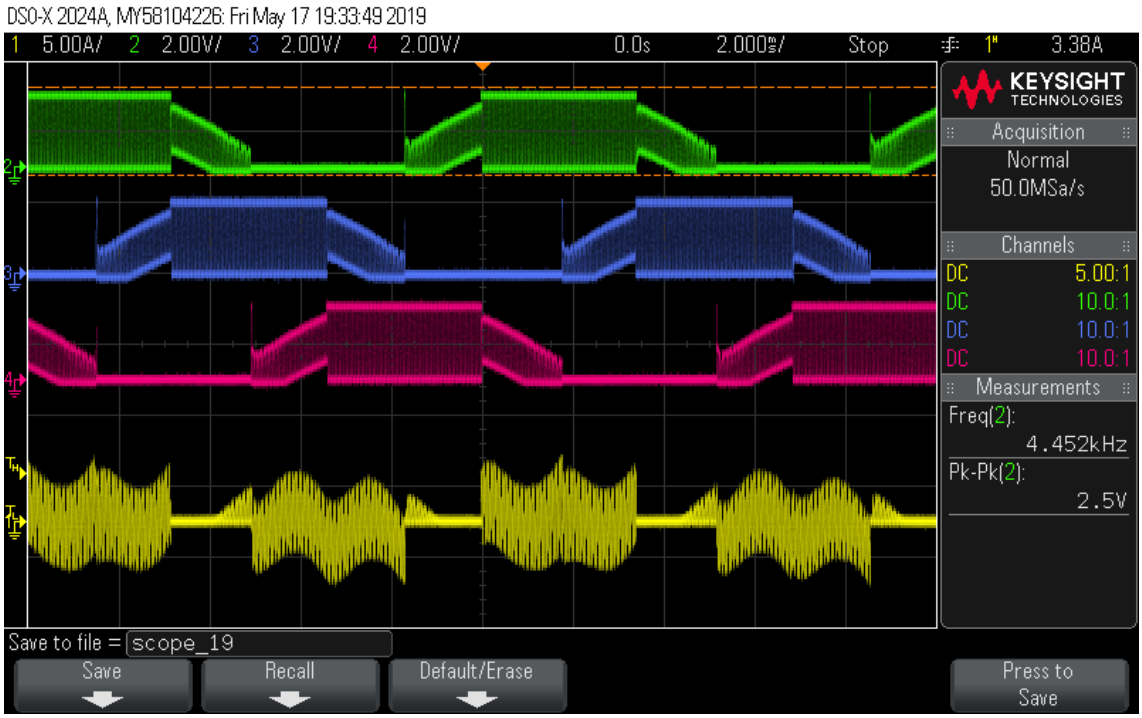
7.1 Porovnání běhu naprázdno

Bezsenzorové řízení naprázdno funguje s uspokojivou přesností, proud je pouze mírně zkreslený. Algoritmus tedy pracuje dle očekávání.

Z obrázků Obr. 7-1 a Obr. 7-2 můžeme porovnat chod naprázdno s a bez použití Hallových senzorů polohy. Při obou případech byla nastavena střída PWM napevno na 50 %, a protože motor má jeden pólový pár, otáčky jsou cca 6000 min^{-1}. Ze srovnání obou průběhů je patrné, že bezsenzorový algoritmus může fungovat téměř se stejným výsledkem, jako algoritmus senzorový.

Z průběhu proudu vidíme, že zvlnění proudu je ΔI = 2,5A, frekvence PWM byla 20kHz, napájecí napětí U_D bylo 20V. Použitím vzorce pro zvlnění proudu a jeho úpravou na rovnici (7-1) zjistíme, že indukčnost vinutí je cca 50 µH. Je také vidět, že proud protéká nejen v době, kdy je daná fáze aktivní, ale také když je zrovna vypnutá a má se na ni měřit indukované napětí. To ale pouze v případě, kdy by napětí mělo být záporné, tehdy se totiž otevře nulová dioda.

$$L = \frac{U_D}{2f\Delta I}(1-s)s \qquad \textbf{(7-1)}$$



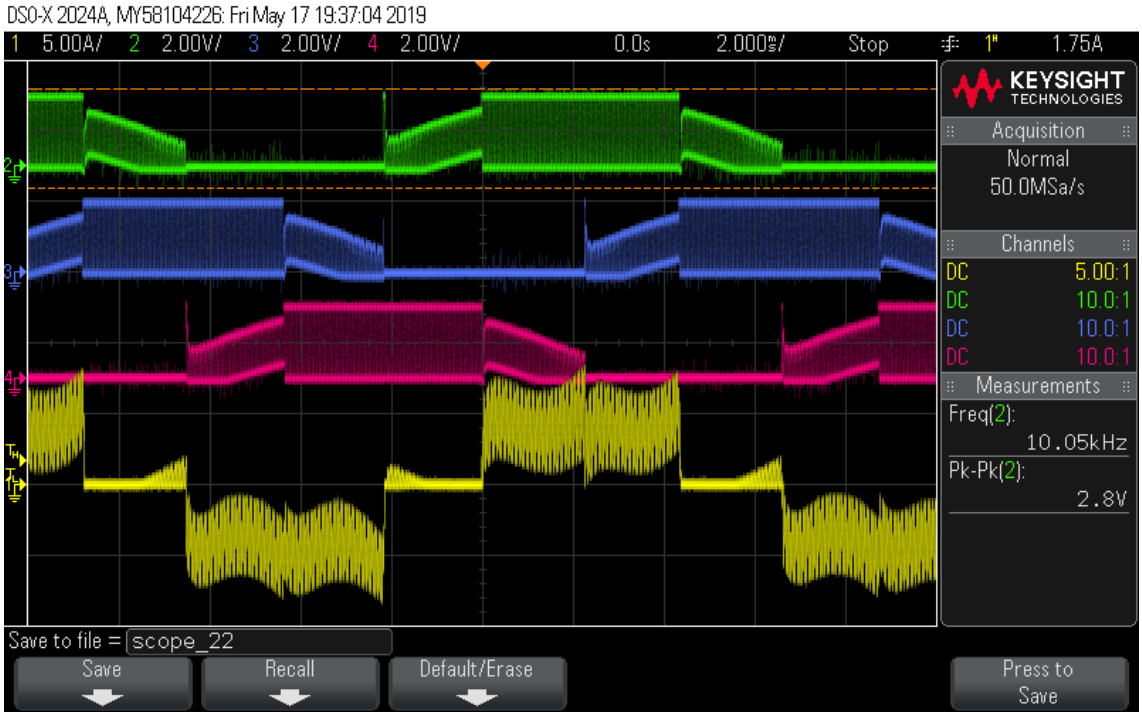
Obr. 7-1: Průběhy napětí a proudu s užitím Hallových senzorů, bez zátěže

Obr. 7-2: Průběhy napětí a proudu s užitím bezsenzorové metody, bez zátěže

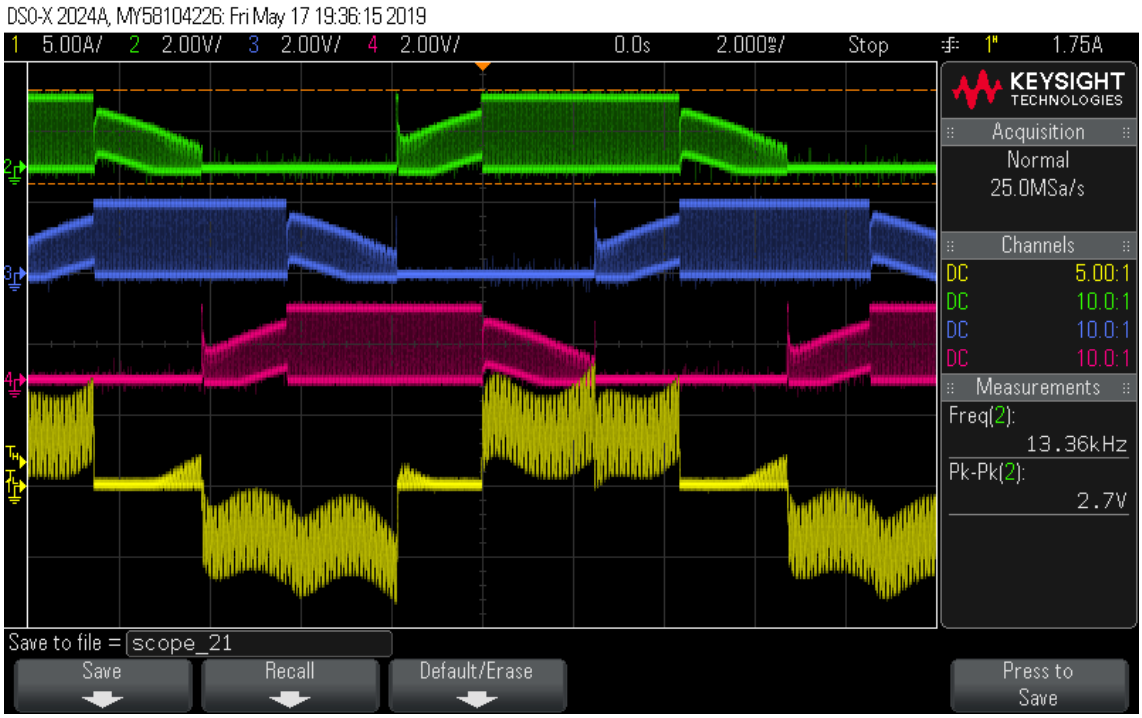
7.2 Porovnání běhu v zatížení

Při zatížení motoru v bezsenzorovém řízení vytvořený algoritmus nefunguje ideálně a je patrné, že se střídá delší a kratší doba mezi komutacemi. Tento problém mohl být způsobený nepřesně nastaveným offsetem u změřeného indukovaného napětí. I přes to ale algoritmus udržel motor v relativně plynulém chodu.

Zatížení motoru bylo provedeno brzděním hřídele. Z oscilogramu je patrné, že proud vzrostl na 3 až 4 A. Bohužel není k dispozici technická dokumentace motoru a nelze tedy říci, jak je tento pracovní bod vzdálený od jmenovitého.



Obr. 7-3: Průběhy napětí a proudu s užitím Hallových senzorů, zatíženo



Obr. 7-4: Průběhy napětí a proudu s užitím bezsenzorové metody, zatíženo

8 ZÁVĚR

V diplomové práci byla řešena problematika bezsenzorového ovládání BLDC motoru. Nejprve je krátce představen BLDC motor, jeho princip, konstrukce a způsoby jeho napájení.

První část práce tvoří teoretický popis několika způsobů bezsenzorového řízení BLDC motoru. Jsou zde rozepsány metody detekce průchodu indukovaného napětí nulou, metoda spřaženého magnetického toku, metoda integrace indukovaného napětí a metoda integrace třetí harmonické indukovaného napětí. Z těchto metod byla pro realizaci vybrána metoda průchodu indukovaného napětí nulou.

Další část tvoří představení hardwaru, který je využit pro ovládání motoru. Jedná se o vývojovou sadu Nucleo F334R8 a laboratorní měnič microStand. Mikrokontorlér byl vybrán pro svoje vhodné vlastnosti a příznivou cenu.

Následující kapitola se zabývá vytvořeným programem. Nejprve je vysvětleno řazení knihoven a ovladačů, a poté je vysvětlena konkrétní implementace senzorového i bezsenzorového řízení. V kapitole se nachází i podrobný rozbor vybraných funkcí programu.

Poslední kapitola obsahuje porovnání senzorového a bezsenzorového řízení při provozu motoru naprázdno a se zatížením. Z naměřených dat je patrné, že se motor podařilo úspěšně řídit s použitím Hallových senzorů i bezsenzorově. Zatížení motoru bylo provedeno pouze orientačně, připojování na dynamometr nebylo prozatím zamýšleno.

Rozšířením práce by mohlo být využití více periferií mikrokontroléru. Například použitím analogového watchdogu, případně integrovaných komparátorů, by bylo možné algoritmus zjednodušit a ušetřit tím procesorový čas potřebný pro výpočty. Další možnost vylepšení algoritmu by mohla spočívat v úpravách měřeného indukovaného napětí, které se následně používá v bezsenzorových algoritmech.

Literatura

[1] *CHANG-LIANG, Xia. Permanent magnet brushless dc motor drives and controls.* Singapore: *John Wiley & Sons Singapore Pte., 2012. ISBN 978-1-118-18833-0.*

[2] *2-2-2 Structure And Application Of Brushless DC Motors | Nidec Corporation. Nidec Corporation* [online]. 2014 [cit. 2018-15-10]. Dostupné z: http://www.nidec.com/en-NA/technology/motor/basic/00019/

[3] GIERAS, Jacek F a Mitchell WING. *Permanent magnet motor technology: design and applications.* 2nd ed., rev. and expanded. New York: Marcel Dekker, c2002. ISBN 0824707397.

[4] KRISHNAN, Ramu. *Permanent Magnet Synchronous and Brushless DC Motor Drives*. United States of America: Taylor & Francis Group, 2010. ISBN 978-0-8247-5384-9.

[5] Moreira, J.C., *Indirect Sensing for Rotor Flux Position of Permanent Magnet AC Motors Operating Over a Wide Speed Rang*e, [online], 1996, [cit. 30.12.2018] Dostupné z: https://ieeexplore.ieee.org/document/345473?tp=&arnumber=345473&url=htt

[6] KRIŽAN, J. *Bezsenzorové řízení BLDC motoru.* Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2012. 70 s. Vedoucí diplomové práce doc. Ing. Robert Grepl, Ph.D.

[7] Moreira, J.C., *Indirect Sensing for Rotor Flux Position of Permanent Magnet AC Motors Operating Over a Wide Speed Range*, [online], 1996, [cit. 5.1.2019] Dostupné z: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=345473&url=htt p%3A%2F%2Fieeexplore.ieee.org%2Fstamp%2Fstamp.jsp%3Ftp%3D%26arnu mber %3D345473

[8] VOREL, Pavel. *Synchronní stroje s permanentními magnety*. Brno: Akademické nakladatelství CERM, 2005. ISBN 80-720-4417-6.

[9] Rclab.info: *The basics of electric power: Brushless motors.* [online]. [cit. 2018-17-12]. Dostupné z: http://www.rclab.info/2014/01/thebasics-of-electric-power-brushless.html

[10] NUCLEO-F334R8: STM32 Nucleo-64 development board with STM32F334R8 MCU, supports Arduino and ST morpho connectivity. STMicroelectronics [online]. 2019 [cit. 2019-01-07]. Dostupné z: https://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-mpu-eval-tools/stm32-mcu-mpu-eval-tools/stm32-nucleo-boards/nucleo-f334r8.html

[11] Gamazo, J.C., Vázquez, E., Gómez, J., *Position and Speed Control of Brushless DC Motors Using Sensorless Techniques and Applicatin Trends*, University of Valladolid,Valladolid, Spain, [online], 2010, [cit. 20.12.2018]

Dostupné z: <http://www.mdpi.com/1424-8220/10/7/6901/pdf>

[12] UM2124 User manual: *Getting started with the six-step firmware library for STM32 Nucleo boards based on STM32F microcontrollers. STMicroelectronics*, [online], 2017, [cit. 3.1.2019] Dostupné také z: https://www.st.com/content/ccc/resource/technical/document/user_manual/group0/90/b6/9f/0c/50/c1/47/1d/DM00334922/files/DM00334922.pdf/jcr:content/translations/en.DM00334922.pdf

[13] PILCH, Tomáš Univerzální měnič na malé napětí: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav výkonové elektrotechniky a elektroniky, 2017. 74 s. Vedoucí práce byl Ing. Jan Knobloch

Seznam symbolů, veličin a zkratek

BLDC	Brushless DC, bezkartáčový motor
STM32	řada 32bitových mikrokontrolérů
ARM M4	jádro procesoru
Jazyk C	programovací jazyk
kW	kilowatt
DC	stejnosměrný proud/napětí
I	proud
U_{H}	Hallovo napětí
k_{H}	Hallova konstanta
B	magnetická indukce
d	tloušťka polovodičové destičky
β	úhel magnetické indukce a plochy polovodiče
α	úhel natočení sond
p	počet pólových dvojic
m	počet fází
H. A, B, C	horní tranzistor větve A, B nebo C
D. A, B, C	dolní tranzistor větve A, B nebo C
n_{mech}	mechanické otáčky
n_{el}	elektrické otáčky
PSMS	synchronní motor s permanentními magnety (permanent magnet synchronous motor)
T	tranzistor
U_{d}	napětí zdroje
C_{d}	vyhlazovací kapacita
A, B, C	větve zátěže
L_{A}, L_{B}, L_{C}	indukčnosti větví zátěže
ZCP	bod průchodu nulou (zero crossing point)
$\boldsymbol{U}$	matice napětí všech fází
$\boldsymbol{R}$	matice odporů
$\boldsymbol{I}$	matice proudů
$\boldsymbol{\psi}$	matice spřažených toků
t	čas
e_A, e_B, e_C	indukovaná napětí na jednotlivých fázích
E_1, E_3, E_5	koeficienty členů fourierovy řady
φ_{el}	elektrický úhel natočení rotoru
i_{A}, i_{B}, i_{C}	proudy jednotlivými fázemi
u_{A}, u_{B}, u_{C}	fázová napětí
R	odpor

L	indukčnost
M	vzájemná indukčnost
u_{SUM}	součet fázových napětí
ψ_{SUM}	spřažený magnetický tok všech fázových napětí
ω	úhlová frekvence
n, h	body pomocné rezistorové sítě
FPU	floating point unit
LSE	low speed external (oscillator) – nízkofrekvenční externí oscilátor
ARM	Advanced RISC Machine
RISC	Reduced Instructions Set Machine – počítač s omezenou sadou instrukcí
Nucleo F334R8	procesorová deska
IHM07M1	rozšiřující deska s měničem
PWM	pulzní šířková modulace
ST	Semiconductor Technologies, výrobce desek Nucleo
CMSIS	knihovna zprostředkování přístupu k registrům
HAL	Hardware Abstraction Layer
LL	Low Level ovladače
V	Volt
mm	milimetr
DMA	Direct Memory Access, kanál přímého přístupu do paměti
kHz	kilohertz
ADC	analogově-digitální převodník
C	kapacita
H, L	zpracované signály
DAC	digitálně-analogový přvodník
BEMF	indukované napětí
U_{f}	napětí fáze
$\underline{U}_{\mathrm{ph}}$	měřené napětí
U_{D}	napětí zdroje
$\mathrm{U_V}$	indukované napětí
µs	mikrosekunda
s	střída
f	frekvence
ΔI	zvlnění proudu
A	Ampér

Přílohy

Knihovna myapp.c

```
/*
 *           myapp.c
 *
 *      Author: Ondrej Kabelka
 */

#include "main.h"
#include "myapp.h"
#include "BLDClib.h"

#include "dataAcq.h"

int hallindex;
//extern int bldcstep[]; // = {0, 0, 0, 0, 0, 0, 0, 0 };
volatile struct{
      volatile uint16_t PhVolt1;
} ADC_Result[2];

enum {
      PhLo = 0,
      PhHi
};

volatile uint16_t DCLinkVolt;
volatile uint16_t DCLinkVolt2;

float Duty, DutyRamp, DutySlope = 0.5e-3;

uint32_t Sensorless = 0;
int PulseCount = 0, PulseNum = 150, PulseNumLast = 150;
int Cvolt, Vvolt, Last = 0, SecondLast, ZCP = 1000, Current, diff,
VDDmultiply = 48, Integration = 0;
uint32_t CommSt = 0, ZCPst = 0, sourceV, CommStF, SLIntMethod = 0;
int SampleDiff = 200;

int HallStep, SLStep;
int SensorlessInit;
volatile int StepDiff, StepDiffArray[4], StepDiffIndex;


uint32_t Vrefp;

uint32_t pulse = 0, step =1;

int hallcommutation[8];

//extern DAC_HandleTypeDef hdac1;

/*
```

```
 * myinit: My initializing function
 */
void myinit(void)
{

	LL_TIM_CC_EnableChannel(TIM1,
				LL_TIM_CHANNEL_CH1 | LL_TIM_CHANNEL_CH1N |
				LL_TIM_CHANNEL_CH2 | LL_TIM_CHANNEL_CH2N |
				LL_TIM_CHANNEL_CH3 | LL_TIM_CHANNEL_CH3N);
	LL_TIM_EnableAllOutputs(TIM1);
	LL_TIM_EnableCounter(TIM1);
	LL_TIM_OC_SetCompareCH1(TIM1, pulse );
	LL_TIM_OC_SetCompareCH2(TIM1, pulse );
	LL_TIM_OC_SetCompareCH3(TIM1, pulse );

	LL_TIM_CC_EnablePreload(TIM1);


	LL_TIM_HALLSENSOR_InitTypeDef tim3hallInit;
	LL_TIM_HALLSENSOR_StructInit(&tim3hallInit);
	LL_TIM_HALLSENSOR_Init(TIM3, &tim3hallInit);

	// enable interrupt on Hall sensor edge
	HAL_NVIC_EnableIRQ(TIM3_IRQn);
	LL_TIM_ClearFlag_CC1(TIM3);
	LL_TIM_EnableIT_CC1(TIM3);

	/* Analog converter init */

	/* DMA setup */
	LL_DMA_SetPeriphAddress(DMA1, LL_DMA_CHANNEL_1,
LL_ADC_DMA_GetRegAddr(ADC1, LL_ADC_DMA_REG_REGULAR_DATA));
	LL_DMA_SetMemoryAddress(DMA1, LL_DMA_CHANNEL_1,
(uint32_t)&ADC_Result[0]);
	LL_DMA_SetDataLength(DMA1, LL_DMA_CHANNEL_1, sizeof(ADC_Result[0]) /
sizeof(uint16_t) );

	LL_DMA_SetPeriphAddress(DMA1, LL_DMA_CHANNEL_2,
LL_ADC_DMA_GetRegAddr(ADC2, LL_ADC_DMA_REG_REGULAR_DATA));
	LL_DMA_SetMemoryAddress(DMA1, LL_DMA_CHANNEL_2,
(uint32_t)&DCLinkVolt);
	LL_DMA_SetDataLength(DMA1, LL_DMA_CHANNEL_2, 1 );

	// Enable
	LL_DMA_EnableChannel(DMA1, LL_DMA_CHANNEL_1);
	LL_DMA_EnableChannel(DMA1, LL_DMA_CHANNEL_2);
	LL_DMA_EnableIT_TC(DMA1, LL_DMA_CHANNEL_1);
	//LL_DMA_EnableIT_TC(DMA1, LL_DMA_CHANNEL_2);



	  /* ADC Setup */
	  // Turn on internal Vref regulator
	  LL_ADC_EnableInternalRegulator(ADC1);
	  //LL_ADC_EnableInternalRegulator(ADC2);
	  // while(LL_ADC_IsInternalRegulatorEnabled(ADC1) == 0);
	  // wait at least 10us to stabilize regulator:
```

```
        HAL_Delay(10);

        // Connect Vref and TempSensor to analog inputs (because of an error
in CubeMX)
        //LL_ADC_SetCommonPathInternalCh(__LL_ADC_COMMON_INSTANCE(ADC1),
        //LL_ADC_PATH_INTERNAL_VREFINT | LL_ADC_PATH_INTERNAL_TEMPSENSOR);

        // Start calibration
        LL_ADC_StartCalibration(ADC1, LL_ADC_SINGLE_ENDED);
        LL_ADC_StartCalibration(ADC2, LL_ADC_SINGLE_ENDED);
        HAL_Delay(10);

        // Enable ADC:
        LL_ADC_Enable(ADC2);
        LL_ADC_Enable(ADC1);
        // Wait until the ADC is ready:
        while(LL_ADC_IsEnabled(ADC2) == 0)
        while(LL_ADC_IsActiveFlag_ADRDY(ADC2) == 0);
        while(LL_ADC_IsActiveFlag_ADRDY(ADC1) == 0);

        LL_ADC_REG_StartConversion(ADC1);
        LL_ADC_REG_StartConversion(ADC2);

        /* Initialize the softTrace for STMStudio */
        ClearBuffer();

      // open DC-LINK-ON transistor
      DCLINKON_GPIO_Port->ODR |= DCLINKON_Pin;


      // Commutation table initialization:
      /*
      Duty = DutyRamp = 0.1;
      HAL_Delay(10);
      commTabInit();
       */

      // Perform first step after reset
      BLDCsetStep2(BLDCgetActualStep()) ;
      LL_TIM_GenerateEvent_COM(TIM1);


      //Perform Sensorless initialization    after reset
      Sensorless = 1;
      initSensorlessStep();
}

/*
 * myDMA1ISR: Interrupt service routine for DMA1 (half PWM period)
 */
void myDMA1ISR(void)
{

      GPIOC->ODR |= GPIO_PIN_9;

      if(LL_DMA_IsActiveFlag_TC1(DMA1))
            LL_DMA_ClearFlag_TC1(DMA1);
```

```
	if(LL_TIM_GetDirection(TIM1) == LL_TIM_COUNTERDIRECTION_UP)
	{
		LL_DMA_SetMemoryAddress(DMA1, LL_DMA_CHANNEL_1,
(uint32_t)&ADC_Result[0]);
		//HAL_DAC_SetValue( &hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R,
(uint32_t)ADC_Result[1].PhVolt1 );
	}
	else
	{
		LL_DMA_SetMemoryAddress(DMA1, LL_DMA_CHANNEL_1,
(uint32_t)&ADC_Result[1]);
		//HAL_DAC_SetValue( &hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R,
(uint32_t)ADC_Result[0].PhVolt1 );
	}

	DCLinkVolt2 = LL_ADC_REG_ReadConversionData12(ADC2);

	//counter of pulses between ZCP
	Vvolt = (int)(ADC_Result[PhHi].PhVolt1) - 0.01*VDDmultiply *
DCLinkVolt2;
	PulseCount++;

	if (ZCPst == 0) {
		if (Last <= 0) {
			if (Vvolt > 0) {
				PulseNumLast = PulseNum;
				PulseNum = (PulseCount-1) / 2;
				ZCPst = 1;
				PulseCount = 0;

			}
		}

		if (Last > 0) {
			if (Vvolt <= 0) {
				PulseNumLast = PulseNum;
				PulseNum = (PulseCount) / 2;

				ZCPst = 1;
				PulseCount = 0;
			}
		}

		SampleDiff = Vvolt - SecondLast;
		SecondLast = Last;
		Last = Vvolt;
	}


	if (PulseNum > PulseNumLast + 5) {
		PulseNum = PulseNumLast + 5;
	}
	if (PulseNum < PulseNumLast - 5) {
		PulseNum = PulseNumLast - 5;
	}
```

```
	if (CommSt > 0) {
		CommSt++;

	}
	if (CommSt > 20) {
		CommSt = 0;
		ZCPst = 0;
	}

	// commutation
	if (PulseNum == PulseCount) {
		makeSensorlessStep();
		if (Sensorless == 1) {
			if(SLIntMethod == 0){
				CommSt = 1;
				CommutationExecution();
			}
		}
	}


	GPIOC->ODR &= ~GPIO_PIN_6;

	HallStep = BLDCgetActualStep();
	SLStep = getSensorlessStep();

	/*
	 * Manual initialization of sensorless
	 */
	if(SensorlessInit)
	{
		initSensorlessStep();
	}


	/* SoftTrace for STMStudio */
	DumpTrace();

	GPIOC->ODR &= ~GPIO_PIN_9;

}

/*
 * myTIM2ISR: Interrupt service routine for TIM2 (every Hall signal edge)
 */
void myHallEdgeISR(void)
{
	/*
	 * Clear interrupt flag
	 */
	LL_TIM_ClearFlag_CC1(TIM3);
	/*
	 * Execute the commutation
	 */
	if(Sensorless == 0)
		CommutationExecution();
```

```
	/*
	 * Read Hall sensors state
	 */
	hallindex = hallRead();
}

void CommutationExecution(void) {
	/*
	 * Generate commutation event
	 */
	LL_TIM_GenerateEvent_COM(TIM1);
	/*
	 * Set the commutation state for next step
	 */

	if (Sensorless) {
		SetADCPhaseToread(getSensorlessStep());
		BLDCsetStep2(getSensorlessStep() + 1);
	} else {
		SetADCPhaseToread(BLDCgetActualStep());
		BLDCsetStep2(BLDCgetActualStep() + 1);
	}

}


/*
 * myloop: infinity loop function
 */
void myloop(void)
{
	  LD2_GPIO_Port->ODR ^= LD2_Pin;
	  HAL_Delay(100);

	// Calculation of internal temperature
	// TempC = __LL_ADC_CALC_TEMPERATURE(TEMPSENSOR_CAL_VREFANALOG,
ADC_Result.Temp , LL_ADC_RESOLUTION_12B);
	// Calculation of AVDD - analog supply voltage
	// Vrefp = __LL_ADC_CALC_VREFANALOG_VOLTAGE(ADC_Result[0].VrefInt ,
LL_ADC_RESOLUTION_12B);

}

/*
 * myTick1ms: 1ms period tick
 */
void myTick1ms(void)
{

	/*
	 * Ramp generation
	 */
	//Duty, DutyRamp, DutySlope;
	float DutyDiv;
	DutyDiv = Duty - DutyRamp;
	if(DutyDiv > DutySlope) // up
```

```
            DutyRamp += DutySlope;
        else if(DutyDiv < DutySlope) // down
            DutyRamp -= DutySlope;
        else DutyRamp = Duty;

        /*
         * Calculation compare vaklue from duty cycle
         */
        pulse = (uint32_t)(DutyRamp * (float)TIM1->ARR);
        /*
         * Set compare values
         */
        TIM1->CCR1 = TIM1->CCR2 = TIM1->CCR3 = pulse;

}
```

Knihovna BLDClib.c

```
#include "main.h"
#include "myapp.h"
#include "BLDClib.h"

#define COMMTABDELAY 500

//commutation table for testing motor. If using different motor, uncomment
commTabInit in myinit in myapp.c
//int bldcstep[] = {0, 2, 6, 1, 4, 3, 5, 0}; //small motor
int bldcstep[] = {0, 6, 2, 1, 4, 5, 3, 0}; //maxon motor
//int bldcstep[] = {0, 6, 2, 1, 4, 5, 3, 0}; // chinese motor
//int bldcstep[] = {0, 0, 0, 0, 0, 0, 0, 0};
//int bldcstep[] = {0, 5, 1, 6, 3, 4, 2, 0}; // motor 2kW

extern uint32_t Sensorless, CommSt, ZCPst, PulseNum, PulseNumLast, Vvolt,
PulseCount, Duty;
extern int Last, SensorlessInit;
int SensorlessStep = 1;
uint32_t SLcounter = 0;
/*
 * Get current step state
 */
int getSensorlessStep(void)
{
        return SensorlessStep;
}

/*
 * make a step and get current step state
 */
int makeSensorlessStep(void)
{
        SensorlessStep++;
        if(SensorlessStep > 6) SensorlessStep = 1;
        return SensorlessStep;
}
```

```
/*
 * Init step state according to Hall state
 */
void initSensorlessStep(void) {
	SLcounter++;
	if(Sensorless == 0)
	{
		SensorlessStep = BLDCgetActualStep();
		PulseNum = 60;
		PulseNumLast = 60;
		PulseCount = 0;
		SensorlessInit = 0;
	}
	else
	{

		if((SLcounter % 200) == 0)
		{
			LL_TIM_GenerateEvent_COM(TIM1);
			SetADCPhaseToread(getSensorlessStep());
			BLDCsetStep2(getSensorlessStep() + 1);
		}
		if(SLcounter > 2000)
		{
			SensorlessInit = 0;
			SLcounter = 0;
		}
	}
}


void commTabInit(void)
{

	/*
	 * Commutation table initialization
	 */

	for (int i=1; i < 7; i++)
	{
		/*
		 * Turning the motor by switching the phases on and off as we
want and reading the values of the Hall sensors
		 */

		switch(i)
		{
		case 1:

			LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1,
LL_TIM_OCMODE_PWM1);
			LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH1 |
LL_TIM_CHANNEL_CH1N);
```

```
                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2,
LL_TIM_OCMODE_FORCED_INACTIVE);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH2 |
LL_TIM_CHANNEL_CH2N);

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3,
LL_TIM_OCMODE_FORCED_INACTIVE);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH3 |
LL_TIM_CHANNEL_CH3N);
                    break;

            case 2:

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1,
LL_TIM_OCMODE_PWM1);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH1 |
LL_TIM_CHANNEL_CH1N);


                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2,
LL_TIM_OCMODE_PWM1);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH2 |
LL_TIM_CHANNEL_CH2N);

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3,
LL_TIM_OCMODE_FORCED_INACTIVE);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH3 |
LL_TIM_CHANNEL_CH3N);
                    break;

            case 3:

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1,
LL_TIM_OCMODE_FORCED_INACTIVE);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH1 |
LL_TIM_CHANNEL_CH1N);

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2,
LL_TIM_OCMODE_PWM1);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH2 |
LL_TIM_CHANNEL_CH2N);

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3,
LL_TIM_OCMODE_FORCED_INACTIVE);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH3 |
LL_TIM_CHANNEL_CH3N);
                    break;

            case 4:

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1,
LL_TIM_OCMODE_FORCED_INACTIVE);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH1 |
LL_TIM_CHANNEL_CH1N);

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2,
LL_TIM_OCMODE_PWM1);
```

```
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH2 |
LL_TIM_CHANNEL_CH2N);

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3,
LL_TIM_OCMODE_PWM1);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH3 |
LL_TIM_CHANNEL_CH3N);
                    break;

            case 5:

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1,
LL_TIM_OCMODE_FORCED_INACTIVE);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH1 |
LL_TIM_CHANNEL_CH1N);

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2,
LL_TIM_OCMODE_FORCED_INACTIVE);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH2 |
LL_TIM_CHANNEL_CH2N);

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3,
LL_TIM_OCMODE_PWM1);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH3 |
LL_TIM_CHANNEL_CH3N);
                    break;

            case 6:

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1,
LL_TIM_OCMODE_PWM1);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH1 |
LL_TIM_CHANNEL_CH1N);

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2,
LL_TIM_OCMODE_FORCED_INACTIVE);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH2 |
LL_TIM_CHANNEL_CH2N);

                    LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3,
LL_TIM_OCMODE_PWM1);
                    LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH3 |
LL_TIM_CHANNEL_CH3N);
                    break;

            }


            LL_TIM_GenerateEvent_COM(TIM1);

            HAL_Delay(COMMTABDELAY);

            /*
             * Filling the bldcstep field with the read values of Hall
sensors transformed into the commutation table
             */
```

```
			bldcstep[hallRead()] = i;

		}
}

int BLDCgetActualStep(void)
{
		return bldcstep[hallRead()];
}


int BLDCsetStep2(int s)
{
		s = ((s - 1) % 6) + 1;
		switch(s)
		{
		case 1:
			//Disable both transistors
			LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1, LL_TIM_OCMODE_PWM1);
			LL_TIM_CC_DisableChannel(TIM1, LL_TIM_CHANNEL_CH1N |
LL_TIM_CHANNEL_CH1);

			//Generate pwm signal and its negation
			LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2, LL_TIM_OCMODE_PWM1);
			LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH2 |
LL_TIM_CHANNEL_CH2N);

			//Turn on bottom transistor
			LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3,
LL_TIM_OCMODE_FORCED_INACTIVE);
			LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH3 |
LL_TIM_CHANNEL_CH3N);
			break;

		case 2:
			// LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1,
LL_ADC_CHANNEL_6);

			LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1,
LL_TIM_OCMODE_FORCED_INACTIVE);
			LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH1 |
LL_TIM_CHANNEL_CH1N);

			LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2, LL_TIM_OCMODE_PWM1);
			LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH2 |
LL_TIM_CHANNEL_CH2N);

			LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3, LL_TIM_OCMODE_PWM1);
			LL_TIM_CC_DisableChannel(TIM1, LL_TIM_CHANNEL_CH3N |
LL_TIM_CHANNEL_CH3);
			break;

		case 3:
			//LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1,
LL_ADC_CHANNEL_7);

			LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3, LL_TIM_OCMODE_PWM1);
```

```
            LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH3 |
LL_TIM_CHANNEL_CH3N);

            LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2, LL_TIM_OCMODE_PWM1);
            LL_TIM_CC_DisableChannel(TIM1, LL_TIM_CHANNEL_CH2N |
LL_TIM_CHANNEL_CH2);

            LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1,
LL_TIM_OCMODE_FORCED_INACTIVE);
            LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH1 |
LL_TIM_CHANNEL_CH1N);

            break;

      case 4:
            //LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1,
LL_ADC_CHANNEL_6);

            LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1, LL_TIM_OCMODE_PWM1);
            LL_TIM_CC_DisableChannel(TIM1, LL_TIM_CHANNEL_CH1N |
LL_TIM_CHANNEL_CH1);

            LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2,
LL_TIM_OCMODE_FORCED_INACTIVE);
            LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH2 |
LL_TIM_CHANNEL_CH2N);

            LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3, LL_TIM_OCMODE_PWM1);
            LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH3 |
LL_TIM_CHANNEL_CH3N);
            break;

      case 5:
            //LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1,
LL_ADC_CHANNEL_8);

            LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1, LL_TIM_OCMODE_PWM1);
            LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH1 |
LL_TIM_CHANNEL_CH1N);

            LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2,
LL_TIM_OCMODE_FORCED_INACTIVE);
            LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH2 |
LL_TIM_CHANNEL_CH2N);

            LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3, LL_TIM_OCMODE_PWM1);
            LL_TIM_CC_DisableChannel(TIM1, LL_TIM_CHANNEL_CH3N |
LL_TIM_CHANNEL_CH3);
            break;

      case 6:


            LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH1, LL_TIM_OCMODE_PWM1);
            LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH1 |
LL_TIM_CHANNEL_CH1N);
```

```
            LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH2, LL_TIM_OCMODE_PWM1);
            LL_TIM_CC_DisableChannel(TIM1, LL_TIM_CHANNEL_CH2N |
LL_TIM_CHANNEL_CH2);

            LL_TIM_OC_SetMode(TIM1,LL_TIM_CHANNEL_CH3,
LL_TIM_OCMODE_FORCED_INACTIVE);
            LL_TIM_CC_EnableChannel(TIM1, LL_TIM_CHANNEL_CH3 |
LL_TIM_CHANNEL_CH3N);
            break;

      default:
            // shutdown PWM

            break;

      }

      return s;
}

void SetADCPhaseToread(int step)
{
      switch(step)
      {
      case 1:
            LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1,
LL_ADC_CHANNEL_6); // Phase1
            break;
      case 2:
            LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1,
LL_ADC_CHANNEL_8); // Phase3
            break;
      case 3:
            LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1,
LL_ADC_CHANNEL_7); // Phase2
            break;
      case 4:
            LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1,
LL_ADC_CHANNEL_6); // Phase1
            break;
      case 5:
            LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1,
LL_ADC_CHANNEL_8); // Phase3
            break;
      case 6:
            LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1,
LL_ADC_CHANNEL_7); // Phase2
            break;
      }

}


//* hallRead: Function to collect the three states of Hall sensors

int hallRead(void)
{
```

```
	int hi;

	hi = (H1_GPIO_Port->IDR & (H1_Pin|H2_Pin|H3_Pin)) / H1_Pin;
	return hi;
}
```

Soubor main.c

```
/* USER CODE BEGIN Header */
/**

******************************************************************************
*
  * @file           : main.c
  * @brief          : Main program body

******************************************************************************
*
  ** This notice applies to any and all portions of this file
  * that are not between comment pairs USER CODE BEGIN and
  * USER CODE END. Other portions of this file, whether
  * inserted by the user or by software development tools
  * are owned by their respective copyright owners.
  *
  * COPYRIGHT(c) 2019 STMicroelectronics
  *
  * Redistribution and use in source and binary forms, with or without
modification,
  * are permitted provided that the following conditions are met:
  *   1. Redistributions of source code must retain the above copyright
notice,
  *      this list of conditions and the following disclaimer.
  *   2. Redistributions in binary form must reproduce the above copyright
notice,
  *      this list of conditions and the following disclaimer in the
documentation
  *      and/or other materials provided with the distribution.
  *   3. Neither the name of STMicroelectronics nor the names of its
contributors
  *      may be used to endorse or promote products derived from this
software
  *      without specific prior written permission.
  *
  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS"
  * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE
  * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
LIABLE
  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL
  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR
```

```
  * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER
  * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY,
  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
USE
  * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
  *

**********************************************************************
*
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------
-*/
#include "main.h"
#include "adc.h"
#include "dma.h"
#include "tim.h"
#include "usart.h"
#include "gpio.h"

/* Private includes ----------------------------------------------------
-*/
/* USER CODE BEGIN Includes */
#include "myapp.h"

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------
-*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------
-*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------
-*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------
-*/

/* USER CODE BEGIN PV */
volatile uint32_t ADCtest;
volatile uint32_t ADCtest2;

/* USER CODE END PV */
```

```c
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_USART2_UART_Init();
  MX_TIM1_Init();
  MX_ADC2_Init();
  MX_ADC1_Init();
  MX_TIM3_Init();
  /* USER CODE BEGIN 2 */
  myinit();

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
```

```
  {
        myloop();
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
  RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

  /**Initializes the CPU, AHB and APB busses clocks
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_BYPASS;
  RCC_OscInitStruct.HSEPredivValue = RCC_HSE_PREDIV_DIV1;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL9;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /**Initializes the CPU, AHB and APB busses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
  {
    Error_Handler();
  }
  PeriphClkInit.PeriphClockSelection =
RCC_PERIPHCLK_TIM1|RCC_PERIPHCLK_ADC12;
  PeriphClkInit.Adc12ClockSelection = RCC_ADC12PLLCLK_DIV1;
  PeriphClkInit.Tim1ClockSelection = RCC_TIM1CLK_HCLK;
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
  {
    Error_Handler();
  }
}

/* USER CODE BEGIN 4 */
```

```
/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state
*/

  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(char *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line
number,
     tex: printf("Wrong parameters value: file %s on line %d\r\n", file,
line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

/************************ (C) COPYRIGHT STMicroelectronics *****END OF
FILE****/
```