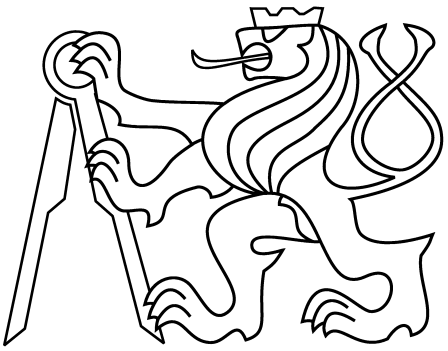
Bakalářská práce

22. května 2020

Mikrořadič jako náhrada obvodu typu Smart Circuit

Jan Bittman



Vedoucí práce: doc. Ing. Jan Fischer, CSc.

České vysoké učení technické v Praze Fakulta elektrotechnická,
Katedra telekomunikační techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Bittman** Jméno: **Jan** Osobní číslo: **466243**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra telekomunikační techniky**

Studijní program: **Elektronika a komunikace**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Mikrořadič jako náhrada obvodu typu Smart Circuit

Název bakalářské práce anglicky:

Microcontroller as Smart Circuit Substitution

Pokyny pro vypracování:

1. Prostudujte literaturu týkající se aplikací obvodu STM32G031.
2. Analyzujte možnosti použití levného mikrořadiče STM32G031 v malém pouzdře jako obvodu typu „Smart Circuit“, kde nahradí funkci specializovaného zákaznického obvodu nebo funkci několika jiných standardních obvodů.
3. Jako příklady zvolte řešení náhrady generátorů impulsů, čítačů, generátorů signálu PWM s modulací střídy, komplexních převodníků ADC, jednoduchých regulátorů, obvodů zpracování signálu enkodérů a obvodů pro převod rozhraní.
4. Navrhněte řešení vybraných příkladů a popište metodiku návrhu tak, aby tato práce mohla sloužit i jako návod při tvorbě aplikací tohoto mikrořadiče v rámci výuky vestavných systémů.
5. Zhodnoťte dosažené výsledky.

Seznam doporučené literatury:

[1] Yiu, J.: The Definitive Guide to ARM Cortex -M0 and Cortex-M0+ processors. Elsevier Science&Technology, 2013. 864 p. ISBN:978-0-12408-082-9.
[2] STMicroelectronics: RM0444 Reference manual, STM32G0x1. [on-line]
[3] STMicroelectronics: DS12992 STM32G031 Data. [on-line]

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Jan Fischer, CSc., katedra měření FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **03.02.2020** Termín odevzdání bakalářské práce: **22.05.2020**

Platnost zadání bakalářské práce: **30.09.2021**

doc. Ing. Jan Fischer, CSc. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. | Podpis studenta

Datum převzetí zadání

Poděkování

Tímto děkuji vedoucímu práce doc. Ing. Janu Fischerovi, CSc. za ochotu a čas vložený do realizace této bakalářské práce, především za vřelé a ochotné rady předané během konzultací.

Prohlášení

Tato práce vznikla v laboratoři videometrie, katedry měření ČVUT - FEL v Praze pod vedením doc. Ing. Jana Fischera, CSc. Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Abstrakt

Cílem této bakalářské práce je navrhnout a realizovat jednoduché obvody typu „Smart circuit“ s obvodem STM32G031J6. Ty mohou nahrazovat funkci již dostupných zákaznických obvodů, např na zpracování dat z enkodéru, čítače impulzů, nebo měřící systémy za využití analogově-digitálního převodníku. Celá práce má zároveň sloužit jako studijní materiál v rámci výuky vestavných systémů, proto byl kladen velký důraz na teoretické vysvětlení funkčnosti jednotlivých částí mikrokontroléru a následně na podrobný popis jednotlivých realizací.

Klíčová slova

Smart circuit, mikrokontrolér, STM32, výuka

Abstract

Objective of this bachelor thesis is to design and implement simple smart circuits using microcontroller STM32G031J6. These can substitute some commonly found integrated circuits, which can be, for example, used as circuit to process data from encoder, to count impulses, or to estabilish measurement systems using analog-digital converter. Whole thesis should as well be used as educational materials for subject „Vestavné systémy“, therefore emphasis was placed on theoretical explanation of whole microcontroler and subsequently to a detailed description of individual implementations.

Key words

Smart circuit, microcontroller, STM32, teaching

Obsah

Seznam obrázků **9**

1 Úvod **11**

2 Rozbor práce **12**
2.1 Co je to mikrokontrolér . . . 13
2.2 Paměť FLASH . . . 13
2.3 Paměť RAM . . . 15
2.4 Frekvence hodinového signálu . . . 15
2.4.1 Základní zdroje hodinového signálu . . . 15
2.4.2 Blok fázového závěsu PLL . . . 16
2.5 Přerušení mikrokontroléru . . . 16
2.6 Sběrnice mikrokontroléru . . . 17

3 Popis jednotlivých částí mikrokontroléru **17**
3.1 Blok RCC pro ovládání hodinové signálu a resetu . . . 18
3.1.1 RESET . . . 18
3.1.2 Nastavení frekvence hodinového signálu . . . 19
3.1.3 Inicializace vstupních/výstupních bran GPIO a periférií . . . 21
3.2 Blok NVIC pro kontrolu přerušení . . . 23
3.2.1 Nastavení a práce s blokem NVIC . . . 23
3.3 Nastavení option bajtů . . . 24
3.4 Blok SYSCFG pro konfiguraci systémových nastavení . . . 25
3.5 Brána universálních vstupů/výstupů - GPIO . . . 25
3.5.1 Mód pinu . . . 26
3.5.2 Způsob výstupu signálu . . . 28
3.5.3 Rychlost výstupu signálu . . . 28
3.5.4 Nastave pull-up/pull-down/floating režimu pinu . . . 28
3.6 ADC - analogově digitální převodník . . . 29
3.6.1 Princip funkce ADC . . . 29
3.6.2 Možnosti využití ADC . . . 30
3.6.3 Nastavení ADC . . . 31
3.7 Blok sériové komunikace USART . . . 33
3.7.1 Asynchronní mód . . . 33
3.7.2 Synchronní mód . . . 34
3.7.3 Základní nastavení USART . . . 34
3.8 Blok pro přímý přístup do paměti - DMA . . . 35
3.8.1 Nastavení DMA v „circular“ režimu pro periférii ADC . . . 36
3.9 Časovače . . . 39
3.9.1 Princip činnosti časovače . . . 40
3.9.2 Možnosti využití časovače . . . 41
3.9.3 Základní nastavení a spuštění časovače . . . 42
3.9.4 Zapnutí čítače v režimu výstupu PWM . . . 43
3.9.5 Nastavení časovače, jako časové reference . . . 44
3.9.6 Nastavení čítače v režimu input-capture . . . 45
3.9.7 Časovač jako zdroj „trigger“ signálu . . . 47

4 Základy práce s mikokontrolérem **48**
4.1 Možnosti programování mikrokontroléru 48
4.2 Možnosti ladění mikrokontrolérů řady STM32G031 48
4.3 Využívání dostupného hlavičkového souboru k programování 49
4.4 Nastavení frekvence hodinového signálu jádra 50
4.5 Praktické využití ADC 51
4.5.1 Postup pro inicializaci ADC se softwarovým spouštěním konverze . . 51
4.5.2 Postup pro inicializaci ADC se softwarovým spouštěním konverze za použití DMA 52
4.5.3 Spouštění ADC pomocí externího spouštěcího signálu 53
4.5.4 Využití mikrokontroléru pro měření odporu 54
4.6 Ovládání RGB diody s digitálním vstupem 57
4.7 Nastavitelný generátor signálu PWM s proměnnou střídou a frekvencí . . . 62
4.8 Aplikace pro měření frekvence a délky impulzů 65
4.9 Aplikace pro řízení DC motoru s enkodérem 68

5 Další možnosti využití procesoru STM32G031J6, jakožto obvod typu „Smart-circuit“ **75**
5.1 Princip funkcionality komunikačního rozhraní 76
5.2 Seznam dostupných funkcionalit po volbě komunikačního protokolu 76
5.2.1 Použití komunikačního rozhraní USART_1 76
5.2.2 Použití komunikačního rozhraní I^2C_1 77
5.2.3 Použití komunikačního rozhraní SPI_1 78
5.3 Další možnosti realizací obvodu typu „Smart-circuit“ 79
5.3.1 Měření proudu vinutím kotvy stejnosměrného motoru 79
5.3.2 Generátor impulzů přesně definované délky 80
5.3.3 Regulace teploty 81
5.3.4 Regulace intenzity osvětlení 82
5.3.5 Měření příkonu 82
5.4 Obrázky reálného zapojení mikrokontroléru na nepájivém poli 84

6 Závěr **85**

Reference **86**

Seznam obrázků

2.1 Základní schéma mikrokontroléru . . . 13
2.2 Adresy paměťového prostoru [1] . . . 14
2.3 Výchozí paměťová adresa ADC[1] . . . 14
2.4 Paměťový offset CR registru[1] . . . 15
2.5 Schéma rozvodu hodinového signálu . . . 15
2.6 Schéma PLL[1] . . . 16
2.7 Schéma sběrnic mikrokontrolérů řady STM32G031 [1] . . . 17
3.1 Schéma PLL [1] . . . 19
3.2 RCC_CR registr[1] . . . 20
3.3 Registr RCC_PLLCFGR [1] . . . 20
3.4 Registr RCC_CFGR[1] . . . 20
3.5 Registr RCC_IOPENR[1] . . . 21
3.6 Registr RCC_IOPRSTR[1] . . . 21
3.7 Registr RCC_AHBENR[1] . . . 22
3.8 Registr RCC_AHBRSTR[1] . . . 22
3.9 Registr RCC_APBENR1[1] . . . 22
3.10 Registr RCC_APBRSTR1[1] . . . 22
3.11 Registr FLASH_OPTR[1] . . . 24
3.12 Registr SYSCFG_CFGR1[1] . . . 25
3.13 Registr GPIO_MODER[1] . . . 26
3.14 Registr GPIOx_IDR[1] . . . 26
3.15 Registr GPIOx_BSRR[1] . . . 27
3.16 Registr GPIOx_AFRL[1] . . . 27
3.17 Registr GPIOx_OTYPER[1] . . . 28
3.18 Registr GPIOx_OSPEEDR[1] . . . 28
3.19 Registr GPIOx_PUPDR[1] . . . 29
3.20 Základní schéma ADC . . . 29
3.21 Princip SAR . . . 30
3.22 Registr ADC_CR[1] . . . 31
3.23 Registr ADC_SMPR[1] . . . 32
3.24 Registr ADC_CHSELR[1] . . . 32
3.25 Registr ADC_CFGR1[1] . . . 32
3.26 USART schéma zapojení - asynchronní mód . . . 33
3.27 USART protokol - asynchronní mód . . . 33
3.28 Registr USART_CR1[1] . . . 35
3.29 Registr ADC_CFGR1[1] . . . 36
3.30 Registr DMAMUX_CxCR[1] . . . 37
3.31 Tabulka obsahující ID jednotlivých periférií[1] . . . 37
3.32 Registr DMA_CPARx[1] . . . 38
3.33 Registr DMA_CMARx[1] . . . 38
3.34 Registr DMA_CNDTRx[1] . . . 38
3.35 Registr DMA_CCRx[1] . . . 39
3.36 Princip činnosti časovače . . . 40
3.37 Schéma čítání nahoru - up . . . 41
3.38 Registr TIM3_PSC[1] . . . 42
3.39 Registr TIM3_ARR[1] . . . 43
3.40 Registr TIM3_CCMR1[1] . . . 44
3.41 Registr TIM3_CCER[1] . . . 44
3.42 Registr TIM3_DIER[1] . . . 45
3.43 Registr TIM2_CCMR1 - režim vstupu[1] . . . 46

3.44 Registr TIM2_CCER - režim vstupu[1] . . . 47
3.45 Registr TIM3_CR2[1] . . . 47
4.1 Ukázka nastavení v programu „STM32 ST-LINK Utility“[4] . . . 48
4.2 Pouzdra LQFP32 a SO8N a jejich rozložení pinů[2] . . . 49
4.3 Registr RCC_APBENR2[1] . . . 50
4.4 Tabulka vstupů spouštěcího signálu[1] . . . 53
4.5 Schéma zapojení napěťového děliče . . . 54
4.6 Celkové schéma zapojení obvodu pro měření odporu . . . 55
4.7 Hodnota odporu na sériovém terminálu . . . 57
4.8 Popis pinů RGB diody[10] . . . 58
4.9 Komunikační protokol RGB diody, definice délky logických stavů[10] . . . 58
4.10 Popis průchodu datových bloků jednotlivými diodami[10] . . . 58
4.11 Schéma kaskádního zapojení RGB LED diod[10] . . . 59
4.12 Princip kaskádního přeposílání dat[10] . . . 59
4.13 Schéma zapojení pro ovládání RGB diody . . . 59
4.14 Schéma časování jednotlivých bitů . . . 60
4.15 Výstup komunikace na sériovém terminálu . . . 62
4.16 Výstup komunikace na sériovém terminálu . . . 65
4.17 Schéma zapojení pro měření frekvence . . . 66
4.18 Měřený signál . . . 67
4.19 Výstup ze sériového terminálu na PC . . . 68
4.20 Rozložení funkcí na jednotlivých pinech . . . 69
4.21 Schéma zapojení obvodu L293D . . . 70
4.22 Schéma zapojení pro USART . . . 70
4.23 Registr TIM2_SMCR[1] . . . 72
4.24 Ukázka výstupu komunikace na sériovém terminálu v PC . . . 75
5.1 Myšlenka obvodu typu Smart-circuit . . . 75
5.2 Schéma zapojení pro USART_1 . . . 77
5.3 Schéma zapojení pro komunikační protokol I^2C_1 . . . 78
5.4 Schéma zapojení pro komunikační protokol SPI_1 . . . 79
5.5 Příklad možného průběhu napětí a proudu u DC motoru . . . 80
5.6 Blokové schéma satelitního generátoru pulzů definované délky . . . 81
5.7 Blokové schéma satelitního regulátoru teploty . . . 81
5.8 Blokové schéma satelitního regulátoru osvětlení . . . 82
5.9 Blokové schéma satelitního měřiče výkonu . . . 83
5.10 Zapojení STM32G031 v pouzdrech SO8N a LQFP32 . . . 84
5.11 Zapojený DC motor . . . 84

1 Úvod

Podnětem ke vzniku této práce byl nápad využít mikrokontrolér jako „chytrý“ obvod, který by zastupoval funkce již dostupných integrovaných obvodů pro zpracovávání jednoduchých dat, jako je zpracování signálu z enkodéru, měření elektrických veličin pomocí analogově-digitálního převodníku, obvod pro generování signálů PWM, či aplikace pro čítání impulzů nebo měření frekvence. Vzhledem ke stávajícímu trendu, kdy se obvody stále rozměrově zmenšují a jsou v nich integrovány spousty periferií, lze vytvořit obvod přímo na míru cílené aplikace. Zároveň s klesajícími rozměry a jejich cenou roste výpočetní výkon těchto součástek, který umožňuje realizace mnohých „chytrých“ obvodů, které by se jinak, za pomoci dostupných zákaznických obvodů, realizovaly velmi těžko.

Pro realizaci využijeme mikrokontrolér od firmy ST, konkrétně se jedná o model STM32G031J6, který je dodáván v pouzdře SO8N, které je velmi kompaktní a nezabere tak v aplikaci, kde by se používalo, zbytečně moc místa. První část mé bakalářské práce se bude věnovat problematice mikrokontroléru STM32G031J6 a studie jeho jednotlivých periférií a bloků. Na základě toho budou analyzovány možnosti jeho využití, jakožto obvodu typu „smart circuit“, kde uvedu dostupné periférie a možnosti jejich kombinací za účelem integrace co nejvíce funkcionalit do jednoho čipu.

Další část se bude věnovat realizaci některých vybraných úloh, jako jsou generátory signálu PWM s modulací střídy a frekvence, převodníky za pomoci využití analogově-číslicových převodníků, obvody pro zpracování dat z enkodéru, nebo jednoduché obvody na měření frekvence signálu a délky pulzu. Tato část bude obsahovat detailní popis jednotlivých kroků při realizaci a reálnou implementaci výsledného kódu v jazyce C. Všechny zhotovené programy budou přiloženy k této bakalářské práci, jako další zdroj informací, které se již do náplně práce nevejdou.

Mikrokontroléry řady STM32G031 jsou, vzhledem k počtu periférií, které v sobě integrují, velmi komplexní. Celá práce má proto zároveň sloužit jakožto výukový materiál pro studenty Fakulty elektrotechnické ČVUT na výuku předmětu Vestavných systémů. Z toho důvodu je v práci kladen velký důraz na teoretickou část popisující jednotlivé periférie. K mikrokontrolérům sice existuje referenční manuál, který je však velmi obsáhlý a pro neznalé velmi těžko srozumitelný. Proto je teoretická část této práce pojata jako shrnutí nejdůležitějších částí referenčního manuálu a jejich jednoduché vysvětlení, tak aby to studenti dobře uchopili.

2 Rozbor práce

Zadání mojí bakalářské práce se týká využití mikrokontroléru STM32G031 a jeho možného použití jako obvodu typu „Smart Circuit“, kde bude nahrazovat jednoduché standardní obvody a zjednoduší tím celkovou složitost zařízení. Jakožto příklady zhotovím jednoduché programy na demonstraci možností použití obvodu, např jako generátor impulsů, generátor PWM s modulovatelnou střídou a frekvencí, jednoduché regulátory nebo ADC převodník. Veškeré potřebné informace k práci s mikokontrolérem budou zpracovány tak, aby je šlo využít jako návod při tvorbě vlastních aplikací v rámci výuky vestavných systémů.

Když si člověk představí mikrokontrolér, většinou se mu vybaví součástka, která slouží k řízení nějakého zařízení a zaujímá funkci jakéhosi „mozku“ celé aplikace. Avšak v současné době, vzhledem ke stále klesajícímu trendu cen mikrokontrolérů a zmenšování jejich velikosti, se dají již tyto součástky využít jakožto náhrada standardních, již sériově vyráběných, obvodů. Dále k většině sériově vyráběných integrovaných obvodů je potřeba mít nějaké externí součástky. Například když si vezmeme známý obvod NE555, používaný jako časovač nebo generátor impulzů, tak při jeho aplikaci je třeba mít připojené k obvodu externí kondenzátory a odpory, aby se nastavil tak, jak uživatel potřebuje. Použitím mikrokontroléru jakožto náhrady se těchto externích součástek zbavíme a celý obvod se stane mnohem přehlednější. Navíc mikrokontrolér je možné využít jako více těchto drobných integrovaných obvodů naráz. Další možností využití tohoto obvodu je, že je schopen komunikovat např pomocí USART s nějakým nadřazeným mikrokontrolérem a posílat mu např. data o tom, zda činnost kterou vykonává probíhá správně, a tudíž se z naší součástky stává samostatně funkční blok, který vykonává svoji činnost a pouze posílá nutné informace nadřazenému mikrokontroléru o své činnosti.

Další výhodou může být využití této součástky např. při měření elektrických i neelektrických veličin. Zde pokud je potřeba signál filtrovat, lze nahradit klasické pasivní RC filtry, filtry softwarovými, čímž lze opět zjednodušit složitost celého obvodu a jeho návrh celkový. V případě že by uživatel chtěl udělat například wattmetr, bez použití mikrokontroléru by bylo třeba použít analogové násobičky, zatímco pomocí mikokontroléru lze změřené hodnoty jednoduše vynásobit a následně zpracovat a převést na správný výsledek bez použití jakýkoliv jiných integrovaných obvodů.

Vzhledem k tomu, že náš mikrokontrolér je vybaven několika časovači, lze jej zároveň využít například jako čítač impulzů s možností výstupu. Tato aplikace lze využít například jako drobné zařízení které měří frekvenci signálu a výsledná data o frekvenci je schopna posílat dalšímu „bloku“ v naší aplikaci.

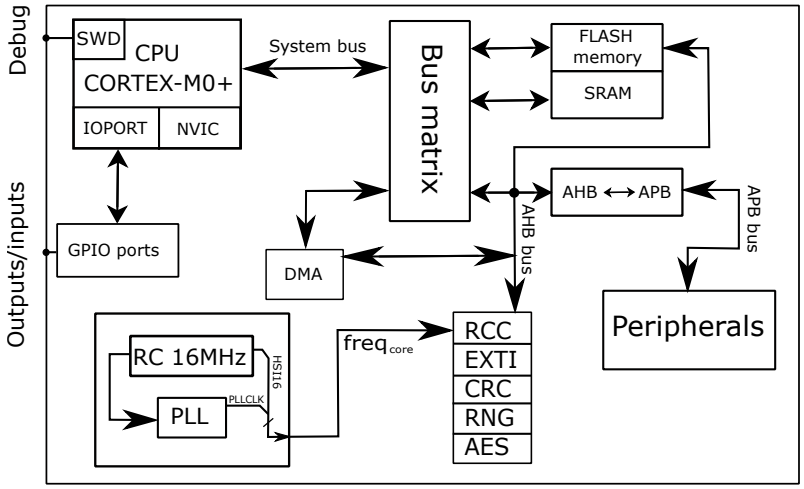
Avšak největší výhodou, při použití součástky tipu „Smart Circuit“ je, že v průběhu celé realizace lze jednotlivé „bloky“ upravovat, bez nutnosti větších zásahů do hardwarové struktury. Stačí pouze upravit software a není třeba vyměňovat mnoho externích součástek, abychom udělali nějakou drobnou změnu do celkového návrhu cílené aplikace. V případě, že uživatel při tomto stylu navrhování výsledného zařízení využije mikrokontrolérů místo klasických integrovaných obvodů, půjde lehce rozšířit jeho oblasti funkčnosti bez větších zásahů do celkového návrhu a tím se celé zařízení stává jakousi modulární součástkou, na které je schopen člověk vrstvit stále další funkční „bloky“.

Samozřejmě celý koncept uvažování o nahrazení standardních obvodů mikrokontrolérem se nehodí na produkty které se vyrábějí ve velkých počtech, neboť cena mikrokontroléru je stále o něco vyšší, než kdybychom tyto součástky používali v již hotových integrovaných obvodech, ale pro zařízení která nejsou vyráběna ve velkém počtu, může být toto zajímavé řešení, které velmi usnadní celkový návrh a zpracování finálního produktu.

2.1 Co je to mikrokontrolér

Jako mikrokontrolér, neboli jednočipový počítač, se označuje součástka, jenž v jednom pouzdře integruje všechny komponenty potřebné k běhu programu. Obsahuje minimálně jedno procesorové jádro, paměť a vstupní/výstupní obvody díky nimž jsme schopni komunikovat s „okolním světem“. Většina mikrokontrolérů má v sobě integrováno spoustu různých periferií, jenž jdou použít pro specifické aplikace. Kupříkladu může obsahovat různé druhy komunikačních periferií, jakožto USART,SPI, či I^2C. Dalším příkladem užitečných periferií mohou být například časovače (timery) nebo ADC (analogově digitální převodník).

V mikrokontrolérech řady STM32G0 je integrováno 32-bitové jádro CORTEX-M0+ o maximální frekvenci hodinového signálu až 64MHz. Samotný mikrokontrolér má přímý výstup na debugovací protokol SWD, má zabudovaný NVIC(nested vector interrupt control), což dává uživateli možnost nastavit si priority jednotlivých přerušených vyvolaných jak jádrem tak periferiemi k němu připojenými.

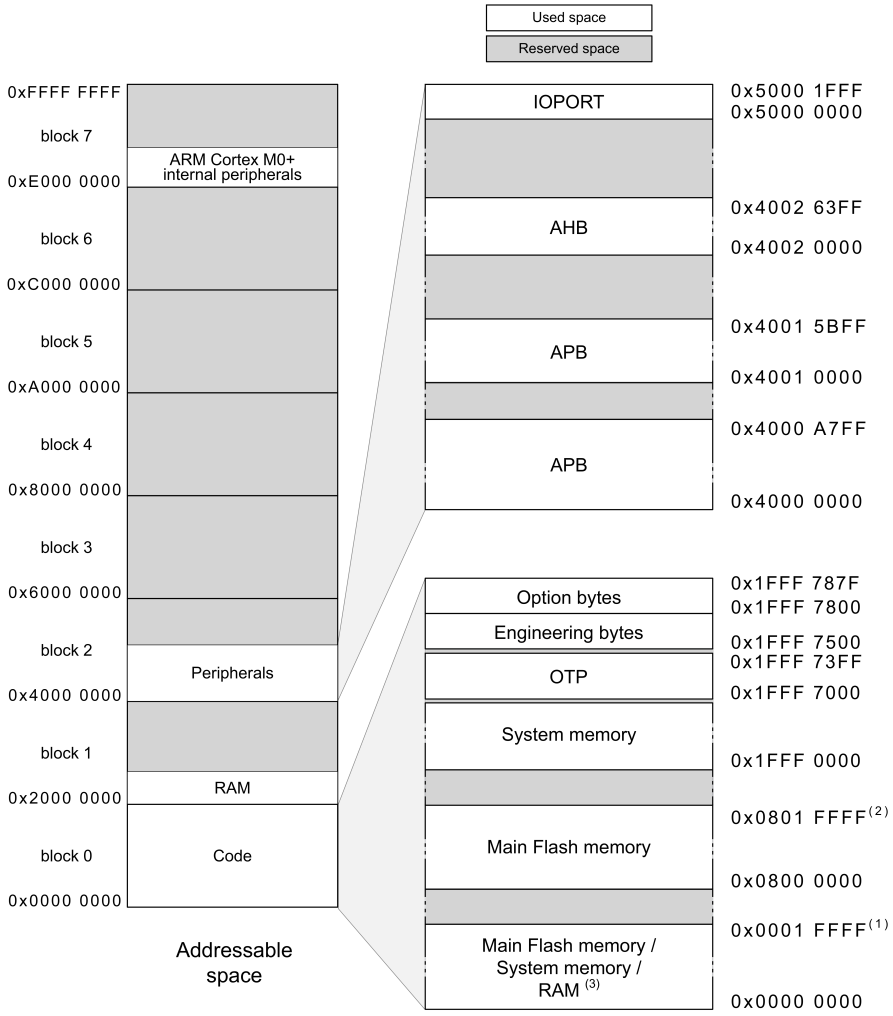


Obr. 2.1: Základní schéma mikrokontroléru

Jádro samotné je pomocí sběrnice System bus (systémová sběrnice) připojeno k bloku Bus matrix (přepínač sběrnic), který zajišťuje komunikaci se všemi ostatními periferiemi a bloky mikrokontroléru. K bloku Bus matrix jsou poté pomocí sběrnice AHB a APB připojeny všechny periferie mikrokontroléru (s vyjímu GPIO portů, ty jsou připojeny napřímo k jádru). Mezi těmito periferiemi lze nalézt například blok RCC pro ovládání resetu a hodinového signálu, blok DMA pro rychlý, na jádru nezávislý, přístup paměti, Analogově digitální převodník, několik časovačů, komunikační periérie jako je USART, SPI, či I^2C, nebo blok EXTI (externa interrupt/event controller), který zajišťuje nastavení externích přerušení mikokontroléru.

2.2 Paměť FLASH

Samotný mikrokontrolér obsahuje interní FLASH paměť, kterou lze využít k ukládání uživatelského programu a konstant, které udávají chod programu. FLASH paměť obsahuje prostor v němž je nahrán firemní program Bootloader, uživatelskou paměť na místo pro jejich instrukce a konstanty a paměť s uloženými konfiguračními bity. Mikrokontroléry řady STM32G031 disponují pouze jedním paměťovým prostorem.



Obr. 2.2: Adresy paměťového prostoru [1]

Celá FLASH paměť je složena ze dvou oblastí. První (hlavní část) obsahuje uživatelský kód, ta druhá (informační blok) je složena ze 3 dílčích částí.

- Nastavovací bity pro uživatelské výchozí nastavení mikrokontroléru
- Systémová paměť obsahující bootloader
- OTP(one-time programmable) oblast, do které je možné zapsat informaci pouze jednou

Dle toho jaká periferie je na jaké sběrnici, je uvedena její výchozí adresa + paměťový offset periferie, ukazující offset jednotlivých bitů periferie vůči výchozí adrese periferie a sběrnice. Viz př. u ADC. Z referenčního manuálu je vidět že začátek adresového prostoru ADC je na adrese (viz obr. 2.3) *0x4001 2400* a offset CR registru je (viz obr. 2.4) *0x08* tudíž, když bychom chtěli zapisovat do CR registru u ADC bude třeba zapisovat na adresu *0x4001 2408* + hexadecimální adresa daného bitu v konkrétním registru.

0x4001 2400 - 0x4001 27FF	1 KB	ADC

Obr. 2.3: Výchozí paměťová adresa ADC[1]

ADC control register (ADC_CR)

Address offset: 0x08

Reset value: 0x0000 0000

Obr. 2.4: Paměťový offset CR registru[1]

2.3 Paměť RAM

Paměť RAM slouží k běhu programu, ale na rozdíl od FLASH paměti není schopna uchovat svůj poslední stav po vypnutí napájení, tudíž po resetu mikrokontroléru, nebo jeho odpojení od napájení, dojde vždy k jejímu kompletnímu přemazání. Každý mikrokontrolér vyžaduje k funkci paměť RAM. V našem případě u modelu STM32G031 je v mikrokontroléru 8Kb SRAM (statická paměť RAM).

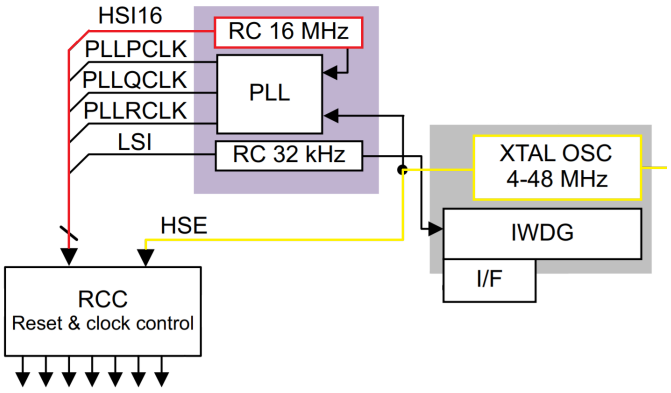
2.4 Frekvence hodinového signálu

Každý mikrokontrolér potřebuje pro svoji činnost zdroj hodinového signálu o určité frekvenci. Od rychlosti tohoto hodinového signálu se odvíjí veškerá činnost, jak jádra, tak periferií mikokontroléru. Proto je pro správný chod třeba mít stabilní zdroj, kterému nebude kolísat frekvence, neboť by mikrokontrolér nemohl správně fungovat. Většina mikrokontrolérů má nějaký zdroj hodinového signálu zabudovaný přímo v pouzdře, tudíž nebývá třeba připojení externího oscilátoru jakožto zdroje hodinového signálu, ale lze využít interního oscilátoru.

2.4.1 Základní zdroje hodinového signálu

Hodinový zdroj signálu pro jádro mikrokontroléru určuje frekvenci, s jakou je jádro schopno vykonávat jednotlivé instrukce. Čím vyšší bude tedy frekvence hodinového signálu přivedená k jádru, tím rychleji se bude vykonávat uživatelský program a také se bude měnit rychlost jednotlivých periferií.

Mikrokontroléry řady STM32G031J6 mají zabudovány interní RC oscilátor o frekvenci 16MHz, jehož výstup je v základní konfiguraci přímo nastaven jako zdroj hodinového signálu pro jádro a všechny periferie(viz obr. 2.5).



Obr. 2.5: Schéma rozvodu hodinového signálu

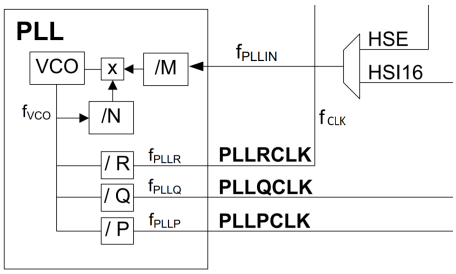
V základní konfiguraci je frekvence jádra nastavena na 16MHz odpovídající internímu RC oscilátoru HSI 16 (viz červená cesta na obr. 2.6). Lze však připojit i externí zdroj

hodinového signálu HSE (viz žlutá cesta na obr. 2.6), tím jsme schopni zajistit přesnější časování aplikací, neboť vnitřní RC oscilátor má vysokou nepřesnost (1%) a navíc se liší kus od kusu již z výroby. Lze jej sice nastavit pomocí HSITRIM registru naladit tak, aby se chyba minimalizovala, ale stále má vysokou teplotní závislost a není to spolehlivý zdroj časové reference. Tomuto jsme schopni pomocí externího krystalového oscilátoru zamezit, neboť externí oscilátory bývají přesnější a méně tepelně závislé.

V případě že chceme docílit jiné frekvence jádra, než je výstupní frekvence oscilátoru, (v případě použití interního RC oscilátoru i externího krystalového), je možné pomocí PLL (phase-locked-loop) zvýšit či snížit frekvenci hodinového signálu.

2.4.2 Blok fázového závěsu PLL

Nastavení správné frekvence hodinového signálu mikrokontroléru je jedním z nejdůležitějších kroků, neboť jinak nebude celý mikrokontrolér fungovat. Blok PLL (phase-locked-loop), neboli fázový závěs, umožňuje uživateli generování hodinového signálu s frekvencí, kterou aplikace vyžaduje a tím lze zajistit správné načasování jednotlivých operací, které chce uživatel vykonávat.



Obr. 2.6: Schéma PLL[1]

Jakožto vstup pro PLL lze nastavit buď zdroj HSI, či HSE. Ten je potřeba nejprve upravit vstupní děličkou minimálně maximální povolenou vstupní frekvenci hodinového signálu pro blok PLL. V našem případě se konkrétně jedná o hodnotu 16MHz. Následně je možné již vydělenou vstupní frekvenci vynásobit číslem a tím ji zvednout, následně je opět možné frekvenci vydělit, aby splňovala maximální možnou frekvenci procesoru. Tímto máme k dispozici jinou frekvenci, než je frekvence interního oscilátoru, a můžeme přepnout zdroj systémového zdroje hodinového signálu na zdroj z výstupu PLL.

2.5 Přerušení mikrokontroléru

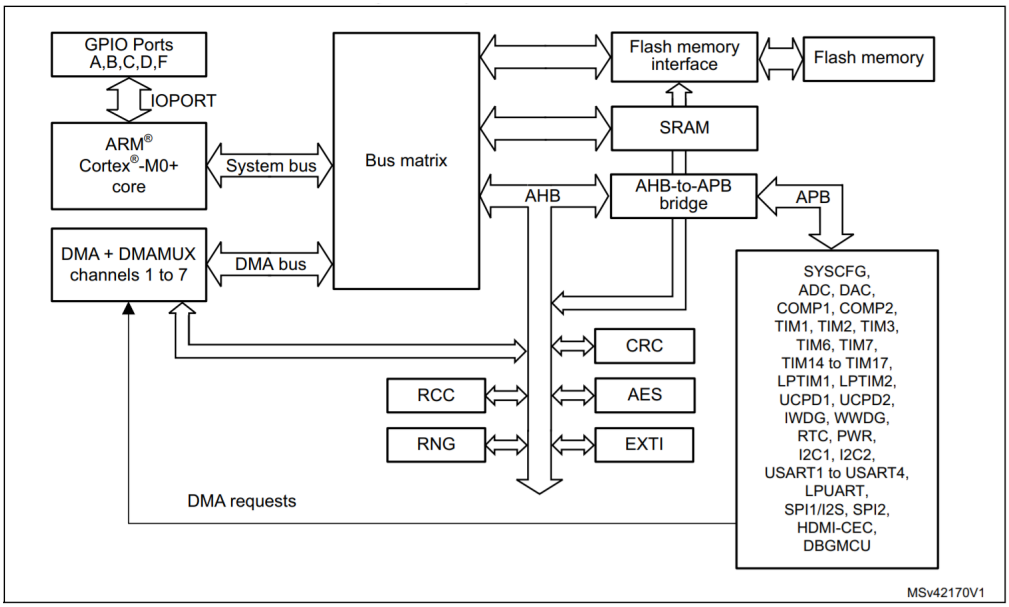
Přerušení u mikrokontroléru značí stav, kdy při určité, uživatelem definované, události přeruší aktuální běh programu začne se vykonávat jiný kus kódu, který slouží k obsluze daného přerušení. Většina periferií mikrokontroléru je schopna generovat přerušení při všech možných stavech. Například periférie ADC umí generovat přerušení po dokončení konverze, časovače pří různých „eventech“ (viz sekce 3.9.1), nebo blok pro komunikaci USART při přijetí nového bloku dat.

Možností jednotlivých přerušení je v celém mikrokontroléru opravdu mnoho. Představují pro uživatele výhodu v tom, že uživatel nemusí při běhu programu stále kontrolovat stavy periférií a jádro tak může vykonávat jiné instrukce, než aby čekalo (například na dokončení konverze ADC - analogově digitální převodník) na danou událost. Pro obsluhu

a nastavování zpracování jednotlivých přerušení slouží blok NVIC, pomocí něhož uživatel nastaví, jaká přerušení chce zpracovávat a přiřadí těmto přerušení dané funkce, které se budou vykonávat.

2.6 Sběrnice mikrokontroléru

Jednotlivé sběrnice v mikrokontroléru zajišťují funkci komunikace mezi periferiemi a jádrem mikrokontroléru. Na jádro jsou přímo připojeny sběrnice System bus(Systémová sběrnice) a IOPORTS (viz obr. 2.7). System bus je přímo připojen na jádro a Bus matrix (přepínač sběrnic), který zajišťuje rozdělení do dalších sběrnic, jako je DMA bus a AHB (advanced high-performance bus).



Obr. 2.7: Schéma sběrnic mikrokontrolérů řady STM32G031 [1]

Na sběrnici AHB jsou připojeny funkce jako, nastavení hodiny(RCC), externí interrupty (EXTI) apod. Následně je na ni připojen blok převádějící signál z APB (advanced peripheral bus), který pak již přivádí signál z periferií jako jsou časovače, ADC, či komunikační periferie jako USART , nebo I^2C.

Jednotlivé sběrnice a na nich připojené registry periferií pak mají každá unikátní adresový prostor s nějakým offsetem vůči začáteční adrese sběrnice (viz kapitola 2.2 FLASH paměť)

3 Popis jednotlivých částí mikrokontroléru

V této části práce se budeme zabývat bližším popisem jednotlivých částí mikokontroléru, jak fungují a k čemu všemu je možné jejich funkce využít. Již vynecháme podrobný popis jak fungují části jako jádro, paměti, nebo sběrnice, ale budeme je brát jako celky které nám zprostředkovávají jisté funkce. Konkrétně se budeme zabývat popisem a základním nastavením periferií mikrokontroléru, jako jsou:

- Blok RCC pro nastavení resetu a hodinového signálu
- Brána vstupů a výstupů GPIO

- Analogově digitální převodník - ADC
- Komunikační rozhraní USART
- Blok pro přímý přístup do paměti - DMA
- Timery

3.1 Blok RCC pro ovládání hodinové signálu a resetu

Tento blok RCC (reset and clock control) je jedním ze základních funkčních bloků v celém mikrokontroléru, protože pomocí něj nastavujeme frekvenci hodinového signálu, zapínáme tím jednotlivé periferie a nastavujeme funkce resetu, kdy a jak k němu má, či nemá dojít. Všechny periferie v našem mikokontroléru jsou totiž vypnuty , kvůli docílení co nejnižší spotřeby. Ve výchozím nastavení je zapnuto pouze jádro, FLASH paměť a paměť SRAM. Uživatel si tudíž zapne jen ty periférie, které bude potřebovat.

Nejprve se podíváme na to, v jakém stavu se procesor nachází po resetu, když jej spustíme bez jakéhokoliv uživatelského nastavení.

- Zdrojem hodinového signálu je interní RC oscilátor o frekvenci 16MHz
- Hodinový signál je přiveden pouze k nezbytným částem mikokontroléru (jádro, paměti, RCC...)
- Všechny piny jsou nastaveny v analogovém režimu
- Všechny periferie jsou vypnuté

3.1.1 RESET

Pod pojmem reset si lze představit požadavek, při kterém v mikrokontroléru dojde k vynulování nastavení všech periferií (až na výjimky - viz RTC reset), vynulování veškerých čítačů, přerušení jakékoliv komunikace a dojde k zastavení programu a jeho vrácení na začátek. Tímto způsobem může uživatel zastavit mikrokontrolér v činnosti kterou vykonávat nemá, způsobenou např špatným uživatelským softwarem, nebo jakýmkoliv jiným problémem, popř. se do režimu resetu dostane mikrokontrolér sám, po určité chybě. Jednotlivé druhy resetu lze kategorizovat. V mikrokontrolérech řady STM32G031 existují celkem 3 různé druhy resetu.

- Systémový reset
- Power reset
- RTC (real time clock) reset

Systémový reset

Systémový reset může být vyvolán, když nastane jedna z následujících událostí :

- Přivedení logické 0 na NRST pin
- Window watchdog event (WWDG reset)
- Independent watchdog event (IWDG reset)
- Sotwarový restart
- Low-power mode security reset
- Power-on reset

- Option byte loader reset

Při tomto resetu jsou nastaveny výchozí hodnoty ve všech registrech kromě RCC_CSR registru a registrů v RTC doméně. V RCC_CSR registru lze následně zjistit, jakým z výše uvedených způsobů byl reset vyvolán.

Power reset

Power reset je vyvolán, pokud dojde k odpojení zdroje napájení (tzv. power-on reset), nebo pokud dojde k poklesu napájecího napětí pod určitou úroveň (tzv. brown-out reset). Při tomto resetu jsou všechny registry nastaveny do výchozího stavu, kromě registrů RTC domény. Dále je vyvolán také při opuštění Standby módu nebo Shutdown módu.

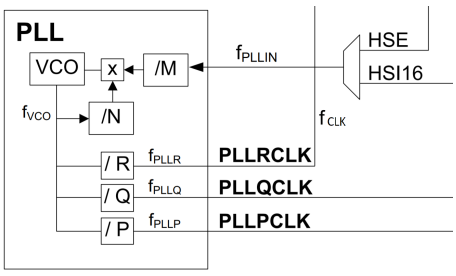
Reset RTC domény

Blok pro práci s reálným časem se nazývá RTC. Tento blok umožňuje uživateli po připojení přesného zdroje hodinového signálu (obvykle 32.768 kHz) práci s reálným časem, popř. datem, přičemž celý blok je možné udržet v chodu i po odpojení napájení a napájet jej separátně, pomocí bateriového zdroje, když mikrokontrolér neběží, nebo je v režimu spánku. Mikrokontrolér STM32G031J6 v pouzdře SO8N však nemá vyvedený pin pro napájení pomocí duplicitního bateriového zdroje, tudíž nelze tento blok udržet v chodu po odpojení hlavního zdroje napájení.

Reset této RTC domény může být vyvolán dvěma způsoby, buďto speciálním způsobem softwarového resetu (vyvolán nastavením RTC_BDCR registru), nebo po přivedení napájení po tom co byl odpojen zdroj napájení V_{DD} i V_{BAT} zároveň.

3.1.2 Nastavení frekvence hodinového signálu

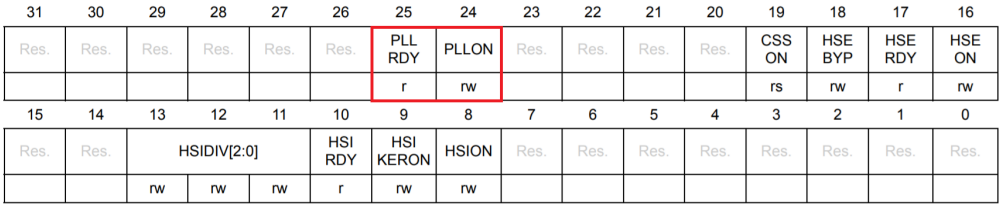
Mikrokontrolér, jakožto celek, je celý řízen hodinovým signálem. Frekvence tohoto hodinového signálu je po resetu nastavena na hodnotu 16MHz (viz 2.4). Když by uživatel chtěl změnit frekvenci hodinového signálu, je buďto třeba připojit externí oscilátor, nebo použít vestavěného bloku PLL a změnit tím frekvenci hodinového signálu f_{clk}. Náš mikrokontrolér (STM32G031J6), vzhledem k omezenému počtu výstupních pinů, nemá možnost připojení externího oscilátoru, tudíž je nutné měnit jeho frekvenci pouze pomocí bloku PLL. Zkratka PLL (phase-locked loop) označuje blok, který je schopen vstupní frekvenci upravit na jinou hodnotu pomocí děliček a násobiček frekvence tak, jak ji uživatel nastaví.



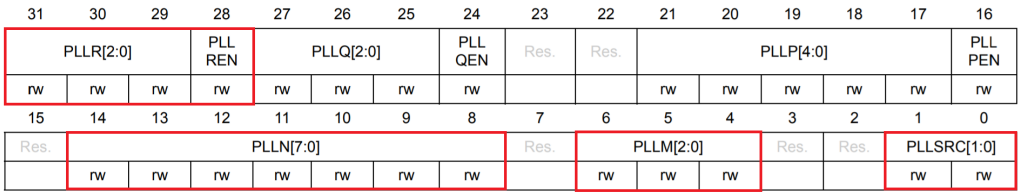
Obr. 3.1: Schéma PLL [1]

Pro správné nastavení PLL je nutné udělat několik korků. Reference ke zmíněným bitům odpovídají schématu PLL viz obr. 3.1. Chceme-li nastavit frekvenci jádra na 64MHz, což je

maximální frekvence, na které je naše jádro schopné pracovat, je třeba nejprve v RCC_CR registru(viz obr. 3.2) vypnout bit PLLON (měl by být ve výchozím nastavení vypnut, ale pokud měníme frekvenci za běhu programu a PLL již byla zapnuta, je třeba zajistit, aby byl nastaven v 0), tím je nám umožněno zapisovat hodnoty do registru RCC_PLLCFGR(viz obr. 3.3). V tomto registru je třeba nastavit bit PLLSRC na hodnotu 2(0b010), což nám nastaví jakožto vstupní hodinový signál bloku PLL interní HSI RC oscilátor (16MHz). Dále nastavíme bit PLLM například na hodnotu 0b0, což nám dá vstupní děličku „/1“, tudíž na vstupu PLL máme 16MHz. Dále nastavíme hodnotu násobičky bitem PLLN na minimální hodnotu 8(0b1000), což nám dá násobení kmitočtu 8x, tudíž máme momentálně frekvenci 128MHz. Pro výstupní děličku R je třeba ještě nastavit bit PLLREN do 1 aby byl povolen výstup z PLL skrze děličku R. Dále nastavíme samotný dělitel R bitem PLLR do hodnoty 1 a tím získáme dělitel /2, což nám dá žádanou výstupní frekvenci 64MHz.

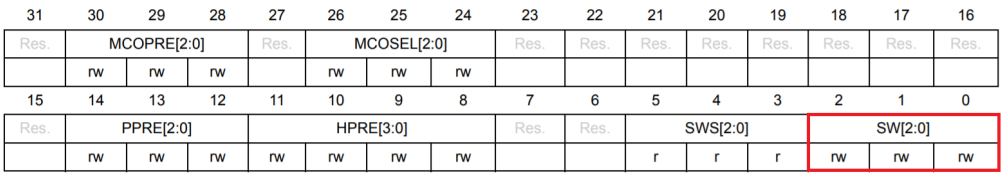


Obr. 3.2: RCC_CR registr[1]



Obr. 3.3: Registr RCC_PLLCFGR [1]

Tímto jsme nastavili samotný blok PLL, dále je třeba jej opět zapnout pomocí bitu PLLON v registru RCC_CR, pak je třeba počkat, než se PLL ustálí, což je signalizováno bitem PLLRDY , který bude systémem nastaven do log.1. Jelikož jsme nastavili frekvenci tak vysoko, že by byl moc rychlý přístup k FLASH paměti, je třeba nastavit ještě odezvu FLASH paměti v registru FLASH_ACR pomocí bitu LATENCY do hodnoty 2(0b01). Specifikace, kolik je třeba nastavit „wait - cycles“ pro FLASH paměť lze najít v referenčním manuálu k STM32G031 na straně 64. Jakožto poslední krok je třeba jen přepnout zdroj vnitřních hodin systému na PLL namísto HSI přímo. To provedeme nastavením bitu SW v registru RCC_CFGR(viz obr. 3.4) do hodnoty 0b010 a tím se zdroj hodinového signálu f_{clk} přepne na 64MHz.



Obr. 3.4: Registr RCC_CFGR[1]

3.1.3 Inicializace vstupních/výstupních bran GPIO a periférií

Pro funkce mikrokontroléru a jeho periferií je potřeba k nim nejprve přivést hodinový signál a tím je uvést do chodu, aby šly správně nastavit. Všechny periferie jsou totiž ve výchozím nastavení, z důvodu snížení spotřeby, vypnuty. To lze udělat pomocí bloku RCC. Je k němu přivedeno veškeré ovládání „toku“ hodinového signálu ke sběrnicím AHB i APB (viz obr. 2.7). Navíc pokud chce uživatel provést reset dané periferie, stačí ji pomocí tohoto bloku resetovat a nastavit ji tak do výchozího nastavení, aniž by proběhl reset celého mikrokontroléru.

Pro zapnutí daného bloku, je nutné přivést k němu hodinový signál. Toho lze docílit pomocí několika různých registrů, přičemž je nutné zvolit ten, ke kterému je naše periférie přiřazena. V základu jsou registry rozděleny na 3 základní sekce.

- RCC_IOPENR registr - zapnutí a vypnutí GPIO portů
- RCC_AHBENR registr - zapnutí a vypnutí periferií na AHB sběrnici
- RCC_APBENRx registry - zapnutí a vypnutí periferií na APB sběrnici

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	GPIOF EN	Res.	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
										rw		rw	rw	rw	rw

Obr. 3.5: Registr RCC_IOPENR[1]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	GPIOF RST	Res.	GPIOD RST	GPIOC RST	GPIOB RST	GPIOA RST
										rw		rw	rw	rw	rw

Obr. 3.6: Registr RCC_IOPRSTR[1]

Díky registru RCC_IOPENR je možné pomocí bitů GPIOxEN zapnout jednotlivé porty GPIO, kde x značí písmeno daného portu (viz obr. 3.5). Vždy je třeba do daného bitu zapsat 1 a pokud tak uživatel účinní, celý port začne fungovat a je již možné jednotlivé GPIO piny v něm nastavit dle toho, jak je bude uživatel využívat. Pokud chce uživatel daný port vypnout, je třeba do stejného registru pouze zapsat na danou pozici portu 0 a tím se port vypne. V případě, že chce uživatel za běhu daný port resetovat do výchozího nastavení, je třeba do registru RCC_IOPRSTR (viz obr. 3.6) na bit GPIOx RST zapsat 1 a následně opět 0. Tím dojde k resetování daného portu, nastavení veškerých registrů do výchozího nastavení a znova spuštění daného portu. Pokud uživatel nenastaví opět hodnotu daného bitu zpět do 0, zůstane celý port v resetu a nebude možné do něj zapisovat.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	RNG EN(1)	Res.	AES EN(1)
													rw		rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	CRC EN	Res.	Res.	Res.	FLASH EN	Res.	Res.	Res.	Res.	Res.	Res.	Res.	DMA EN
			rw				rw								rw

Obr. 3.7: Registr RCC_AHBENR[1]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	RNG RST(1)	Res.	AES RST(1)
													rw		rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	CRC RST	Res.	Res.	Res.	FLASH RST	Res.	Res.	Res.	Res.	Res.	Res.	Res.	DMAR ST
			rw				rw								rw

Obr. 3.8: Registr RCC_AHBRSTR[1]

Obdobně jako u registru RCC_IOPENR, je možné také v registru RCC_AHBENR zapnout či vypnout hodinový signál pro bloky mikrokontroléru připojené na AHB sběrnici. V tomto případě je třeba v registru RCC_AHBENR (viz obr. 3.7) zapsat do příslušného bitu bloku, který chceme spustit, zapsat log.1, a tím jej zapneme a je možné měnit jeho nastavení. V případě vypnutí je potřeba opět přepsat daný bit do 0. Pokud potřebuje uživatel některý blok resetovat za běhu programu, je potřeba, stejně jako u GPIO portů, zapsat do RCC_AHBRSTR (viz obr. 3.8) na bit příslušného bloku log.1 a následně opět log.0. Tím dojde k resetování dané periferie.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LPTIM1 EN	LPTIM2 EN	DAC1 EN	PWR EN	DBG EN	UCPD2 EN	UCPD1 EN	CEC EN	Res.	I2C2 EN	I2C1 EN	LP UART1 EN	USART4 EN	USART3 EN	USART2 EN	Res.
rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	SPI2 EN	Res.	Res.	WWDG EN	RTC APB EN	Res.	Res.	Res.	Res.	TIM7 EN	TIM6 EN	Res.	Res.	TIM3 EN	TIM2 EN
	rw			rw	rw					rw	rw			rw	rw

Obr. 3.9: Registr RCC_APBENR1[1]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
LPTIM1 RST	LPTIM2 RST	DAC1 RST	PWR RST	DBG RST	UCPD2 RST	UCPD1 RST	CEC RST	Res.	I2C2 RST	I2C1 RST	LP UART1 RST	USART4 RST	USART3 RST	USART2 RST	Res.
rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	SPI2 RST	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TIM7 RST	TIM6 RST	Res.	Res.	TIM3 RST	TIM2 RST
	rw									rw	rw			rw	rw

Obr. 3.10: Registr RCC_APBRSTR1[1]

Většina periferí, jako jsou všechny timery, ADC, USART ap., jsou připojeny na sběrnici APB. Z důvodu, že v mikokontroléru je hodně periferií, je registr na pouštění hodinového signálu rozdělen na dva 32-bitové registry RCC_APBENR1 a RCC_APBENR2 a k ním komplementární RCC_APBRSTR1 a RCC_APBRSTR2. V obrázcích 3.9 a 3.10 jsem znázornil pouze první dvojici, neboť ta druhá má naprosto stejnou strukturu, jen obsahuje bity ostatních periferií. Pro zapnutí periferie je stejný postup jako u předešlých. Do bitu

dané periferie v registru RCC_APBENRx zapíšeme 1. Pokud bychom ji chtěli naopak vypnout, zapíšeme tam 0. Pro reset opět zapíšeme do registru RCC_APBRSTRx na bit dané periferie 1 a následně ihned 0. Tím se nám periferie zresetuje a bude se nacházet ve výchozím nastavení.

3.2 Blok NVIC pro kontrolu přerušení

Blok NVIC (Nested vectored interrupt controller) slouží v mikrokontrolérech řady STM32G031 (stejně tomu tak je i u jiných mikrokontrolérů založených na jádru ARM) pro zpracování a nastavování jednotlivých přerušení(viz sekce 2.5), generovanými buďto periferiemi mikrokontroléru, nebo z externích zdrojů. V tomto bloku může uživatel jednotlivým přerušením nastavit jejich prioritu a přiřadit danou funkci (obslužnou rutinu), která se má začít vykonávat, když přerušení nastane.

Priorita přerušení značí „důležitost“ jednotlivých přerušení, tudíž uživatel si může nastavit, která přerušení jsou pro běh mikrokontroléru důležitější, a která méně. Přerušení se následně budou vykonávat v pořadí od nejvyšší priority po nejnižší. Tudíž, když nastanou dvě přerušení naráz, mikrokontrolér začne vykonávat rutinu přiřazenou k přerušení s vyšší prioritou namísto ostatních přerušení. Pokud by při vykonávaní nějakého přerušení s nižší prioritou nastalo přerušení s vyšší prioritou, přeskočí běh kódu na rutinu přiřazenou k přerušení s vyšší prioritou. Po dokončení rutiny pro obsluhu „důležitějšího“ přerušení se kód vrátí k obsluze přerušení s nižší prioritou a běh této obslužné rutiny dokončí. Toto pokračuje do té doby, než se obslouží všechna vnořená přerušení a mikrokontrolér bude vykonávat běh programu mimo jednotlivá přerušení, než opět nastane nějaké nové přerušení.

3.2.1 Nastavení a práce s blokem NVIC

Pro jednodušší práci s blokem NVIC nalezneme v .h souboru jménem „core_cm0plus.h“ sadu funkcí pro obsluhu tohoto bloku. Nejdůležitějším z nich jsou funkce:

- *NVIC_EnableIRQ(IRQn_Type IRQn)*
- *NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)*

První z dvojice funkcí nám povolí zpracovávat přerušení do dané periférie. Vstupem funkce je „zdroj“ přerušení, tedy vektor daného přerušení specifický pro každou periférii. Seznam „odkazů“, na jednotlivé periférie, je zadefinovaný v souboru „startup_stm32g031xx.s“. U všech „odkazů“ se jedná vždy nejprve o název celé periférie následovaný doplňkem „IRQn“. Tedy například kdybychom chtěli povolit přerušení od periférie USART_1, funkce bude mít následující tvar:

```
NVIC_EnableIRQ(USART1_IRQn);
```

Tímto jsme úspěšně povolili v bloku NVIC přerušení od dané periférie. Dále je potřeba nastavit danému přerušení jeho prioritu. Pro to využijeme druhou z funkcí:

- *NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)*

Tato funkce požaduje na vstupu opět zdroj přerušení, tedy stejně jako v minulém případě, název celé periférie následovaný doplňkem „IRQn“. Jakožto druhý parametr funkce je potřeba nastavit samotnou prioritu k přerušení přiřazenou. U mikrokontrolérů řady STM32G031 lze jednotlivým přerušením nastavit prioritu 0-4, přičemž nižší hodnota značí vyšší prioritu přerušení, tedy přerušení s prioritou 0 bude „nejdůležitější“ a naopak přerušení s prioritou 4 nejméně „důležité“. Pokud bychom chtěli nastavit například prioritu přerušení od periferie USART_1, funkce bude vypadat takto:

```
NVIC_SetPriority(USART1_IRQn,0);
```

Pokud máme povolené přerušení, nastavenou jeho prioritu a zapnuté samotné generování přerušení v konfiguračních registrech jednotlivých periférií (viz nastavení jednotlivých periférií mikrokontroléru), můžeme přejít na jeho obsluhu pomocí uživatelské rutiny (funkce). Tu vytvoříme na základě vektoru přiřazenému k danému přerušení. Seznam vektorů k jednotlivým přerušením a jejich definice lze nalézt v souboru „startup_stm32g031xx.s“. Zde najdeme přerušení, které chceme obsluhovat a vytvoříme pro něj funkci typu ***void*** která bude mít název vektoru, který vybereme. Tudíž kdybychom chtěli obsluhovat přerušení od periférie USART_1, bude funkce vypadat takto:

```
void USART1_IRQHandler(void)
    {
    //user code
    }
```

Pozor, při obsluze přerušení nesmí uživatel zapomenou na nulování registrů u jednotlivých periferií, kde se zapisuje, zda došlo k přerušení. Pokud by tak neučinil, vyvolá se toto přerušení po jeho ukončení ihned znovu.

3.3 Nastavení option bajtů

Option bajty jsou část paměti FLASH, kde jsou uloženy výchozí nastavení určitých funkcí mikrokontroléru, od nichž se odvíjí chování mikrokontroléru při spuštění. Jedná se o nastavení resetu mikokontroléru, nastavení režimů bootování, nebo chování jednotlivých pinů, jako je např reset pin (NRST). Tyto bajty je třeba nastavit dle uživatelské aplikace podle toho, jak se bude mikrokontrolér využíván.

Pokud chce uživatel změnit nastavení chování některých výchozích nastavení resetu, nebo např. režimu bootování firmwaru, je třeba změnit ve FLASH paměti konfiguraci option bajtů v FLASH_OPTR registru (viz obr. 3.11).Tento registr je načten při spuštění mikrokontroléru a dle něj je určeno z jaké paměti bude bootovat, jakým způsobe bude bootovat, nastavení jak se má chovat NRST pin mikrokontroléru, či nastavení mezního napětí pro brown-out reset.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	IRHEN	NRST_MODE [1:0]		n BOOT0	n BOOT1	nBOOT _SEL	Res.	RAM_PARITY _CHECK	Res.	Res.	WWDG _SW	IWGD _STDBY	IWDG _STOP	IWDG _SW
		r	r	r	r	r	r		r			r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
nRST_ SHDW	nRST_ STDBY	nRST_ STOP	BORR_LEV[1:0]		BORF_LEV[1:0]		BOR_ EN	RDP[7:0]							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Obr. 3.11: Registr FLASH_OPTR[1]

Nastavení NRST pinu

V našem případě, kdy využíváme velmi malé pouzdro mikrokontroléru, je třeba zanedbat co nejvíce pinů, které obsluhují funkce mikrokontroléru, jako je například reset pin, a zanechat je pro využití programem pro naši aplikaci. Pokud bychom tedy chtěli používat NRST pin jakožto výstupní GPIO pin je třeba vypnout jeho funkci reset pinu v tomto registru. Toho docílíme nastavením NRST_MODE bitů do hodnoty 10(viz obr. 3.11), což nám zcela odstaví funkci resetu pomocí tohoto pin a tudíž ho lze standardně využívat jakožto vstup, či výstup.

Nastavení vstupu do boot režimu

V případě, že chce uživatel využívat možnost bootování pomocí rozhraní USART skrze interní Bootloader, je třeba neprve přenastavit option bajty mikrokontroléru. Pokud bychom chtěli změnit režim jakým mikrokontrolér vstoupí do bootovací sekvence, z režimu, kdy je ovládán hodnotou nBOOT0 bitu (výchozí hodnota), na režim, kdy vstoupí do bootovací sekvence když, je na BOOT0 pinu přivedena logická 1, je třeba nastavit bit nBOOT_SEL do hodnoty 0 (viz obr. 3.11).

3.4 Blok SYSCFG pro konfiguraci systémových nastavení

Blok SYSCF (System configuration controller) slouží pro ovládání systémových nastavení některých periférií, jako je I^2C, nebo modul pro generování IR (infračerveného) signálu, nebo také pro přemapování některých pinů namísto jiných. Funkci pro přemapování pinů budeme často využívat, neboť na našem mikrokontroléru (STM32G031J6) máme k dispozici pouze 6 pinů pro využití v aplikacích (2 piny jsou pro napájení mikrokontroléru).

Tento blok se nachází na sběrnici APB, tudíž pro jeho funkci je potřeba jej nejprve připojit ke zdroji hodinového signálu (viz sekce 3.1.3). Následně můžeme začít měnit jednotlivá nastavení. V této sekci si naznačíme pouze návod na nastavení přemapování jednotlivých pinů, neboť nastavení pro periférii I^2C, IR modulátor, ani jiné, nebudeme v této práci využívat. Konfigurační bity pro přemapování pinů se nacházejí v SYSCFG_CFGR1 registru (viz obr. 3.12). Zde máme možnost přemapování dvou pinů. Můžeme přemapovat buďto pin PA9 namísto pinu PA11, nebo pin PA10 namísto pinu PA12. Přemapování pinu PA11 na pin PA9 povolíme zapsáním log.1 na pozici bitu PA11_RMP a přemapování pinu PA12 na pin PA10 zapsáním log.1 na pozici bitu PA12_RMP. Toto nastavení budeme využívat primárně pro provoz periférie USART, neboť jinak nelze na tomto pouzdře využívat SWD pro programování, a USART zároveň, neboť spolu sdílí fyzické piny na pouzdře mikrokontroléru.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	I2C_PA10_FMP	I2C_PA9_FMP	I2C2_FMP	I2C1_FMP	I2C_PB9_FMP	I2C_PB8_FMP	I2C_PB7_FMP	I2C_PB6_FMP
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	UCPD2_STROBE	UCPD1_STROBE	BOOSTEN	IR_MOD [1:0]		IR_POL	PA12_RMP	PA11_RMP	Res.	MEM_MODE [1:0]	
					w	w	rw	rw		rw	rw	rw		rw	rw

Obr. 3.12: Registr SYSCFG_CFGR1[1]

3.5 Brána universálních vstupů/výstupů - GPIO

Jedná se o bránu zprostředkovávající fyzický výstup logických signálů z mikrokontroléru, popř jejich vstup do mikrokontroléru. Zkratka GPIO z dokumentace (general pourpose input/output) značí, že lze tuto bránu využít k veškerým periferiím, které vyžadují fyzický vstup/výstup. Lze pomocí ní vypínat/zapínat logické úrovně na pinu, komunikovat pomocí USART, SPI ap., či např. číst analogové napětí pomocí ADC. Celý GPIO blok je rozdělen do sekcí, podle písmen jednotlivých portů. Každý z portů lze nakonfigurovat různě, podle potřeb uživatele.

Mikrokontroléry řady STM32G031 disponují skupinou 5 různých GPIO portů (konkrétně A, B, C, D a F, kde každý může mít vyvedeno až 16 pinů (v závislosti na velikosti

pouzdra jsou vyveden různé piny na fyzické kontakty na pouzdře). Než začne uživatel pin využívat k dané činnosti je třeba jej nakonfigurovat do správného režimu.

U jednotlivých pinů lze nastavit několik věcí :

- Mód
- Způsob výstupu signálu
- Rychlost výstupu signálu
- Pull-up/pull-down/floating režim pinu

3.5.1 Mód pinu

Každému pinu v daném portu může uživatel přiřadit některou z vybraných funkcí. Pro nastavení správného módu pinu je potřeba zapsat k příslušnému pinu správnou hodnotu do GPIOx_MODER registru(viz obr. 3.13), kde x je písmeno portu, např. pro pin PA3 to bude GPIOA.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Obr. 3.13: Registr GPIO_MODER[1]

Jedná se vždy o dvojici bitů, kde můžeme vybrat ze 4 možných módu pro každý pin.

- Mód vstupního digitáního pinu - 00
- Mód digitálního výstupu - 01
- Mód alternativní funkce - 10
- Analogový mód(výchozí nastavení) - 11

Mód digitálního vstupu

V tomto režimu pin funguje tak, že do svého registru zapisuje log. hodnotu, která je na pin připojena. Pro nastavení pinu do režimu digitálního vstupu je potřeba na pozici daného pinu v GPIOx_MODER registru zapsat hodnotu 0b01. Následně můžeme z registru GPIOx_IDR (viz obr. 3.14) vyčíst na pozici daného bitu, zda je na něj přivedena log.1 nebo log.0. Aby nezůstal pin v tzv „floating“ režimu, kdy je pin odpojený (např. při používání tlačítka), je potřeba k němu přiřadit buďto pull-up nebo pull-down resistor, aby byla přesně specifikován logický stav pinu ve všech případech. Pull-up či pull-down resisor můžeme připojit buďto externě (separátní součástkou), nebo interně pomocí GPIOx_PUPDR registru (viz sekce 3.5.4).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID15	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Obr. 3.14: Registr GPIOx_IDR[1]

Mód digitálního výstupu

V módu digitálního výstupu lze na pin zapsat buď logická 1 nebo 0 v nastavení registru GPIOx_BSRR(viz obr. 3.15), kde x je referencí na daný port.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Obr. 3.15: Registr GPIOx_BSRR[1]

Když chce uživatel zapsat logickou 1 na výstup daného pinu je třeba zapsat do BSx bitu (x je reference čísla daného pinu) hodnotu 0b1. Tím se nám na fyzickém výstupním pinu objeví logická 1. Když chceme změnit stav pinu opět na log.0, je potřeba zapsat na pozici příslušného BRx bitu hodnotu 0b1, tím se opět resetuje logická hodnota na výstupu a na pinu se objeví logická 0.

Mód alternativní funkce

V případě, kdy chce uživatel k GPIO pinu vnitřně přivést signál z nějaké periferie, je třeba pin nastavit do alternativního můdu (např. jako výstup PWM z timeru, nebo komunikační piny Tx a Rx pro USART atd.). Když je pin správně nastaven v alternativním režimu, je třeba ještě dodatečně nastavit správnou alternativní funkci pinu. Každý z pinů může mít až 8 různých alternativních funkci (viz datasheet k mikrokontrolérům řady STM32G031), ale ne všechny bývají obsazeny.

Vzhledem k tomu, že registr může mít maximální délku 32 bitů a pro každý z 16ti pinů je třeba mít 4 bity (8 unikátních alternativních funkcí), tak je registr rozdělen na dva „subregistry“ GPIOx_AFRL (alternate function register low) a GPIOx_AFRH (alternate function register high). Registr GPIOx_AFRL (viz obr. 3.16) se stará o obsluhu prvních 8 pinů (0-7), zatímco GPIOx_AFRH o druhou polovinu(8-15).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFSEL7[3:0]				AFSEL6[3:0]				AFSEL5[3:0]				AFSEL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFSEL3[3:0]				AFSEL2[3:0]				AFSEL1[3:0]				AFSEL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Obr. 3.16: Registr GPIOx_AFRL[1]

Analogový mód

V tomto režimu se se pin chová jako analogový vstup, či výstu, tudíž jsou hardwarově odstaveny interní pull-up a pull-down rezistory, je vypnut výstupní buffer a Schmitt trigger (komparátor s hysterezí) je odstavený od funkce.

3.5.2 Způsob výstupu signálu

Způsob výstupu signálu je třeba nastavit v registru GPIOx_OTYPER (viz obr. 3.17) buďto 1 nebo 0, dle toho jestli chceme mít výstup ve formě open-drain, nebo push-pull (výchozí hodnota)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Obr. 3.17: Registr GPIOx_OTYPER[1]

3.5.3 Rychlost výstupu signálu

Rychlost výstupu signálu u jednotlivých pinů značí rychlost změny log.1 na log.0 a obráceně. Tato hodnota nemá nic společného s rychlostí hodinové signálu, ale je to pouze otázka výstupního CMOS budiče. Rychlost výstupu signálu lze nastavit na čtyři různé hodnoty.

- Velmi nízká rychlost - hodnota 0b00
- Nízká rychlost - hodnota 0b01
- Vysoká rychlost - hodnota 0b10
- Velmi vysoká rychlost - hodnota 0b11

Pro nastavení těchto hodnot je třeba zapsat danou dvojici bitů do GPIOx_OSPEEDR registru na pozice příslušící pinu (OSPEEDx, kde x značí námi zvolený pin), pro který hodnotu nastavujeme (viz obr. 3.18). Pokud bychom tedy nastavovali např. pin Px3, bude třeba zapsat na pozice 6 a 7 hodnoty dle rychlosti, kterou chceme využívat.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEED15 [1:0]		OSPEED14 [1:0]		OSPEED13 [1:0]		OSPEED12 [1:0]		OSPEED11 [1:0]		OSPEED10 [1:0]		OSPEED9 [1:0]		OSPEED8 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEED7 [1:0]		OSPEED6 [1:0]		OSPEED5 [1:0]		OSPEED4 [1:0]		OSPEED3 [1:0]		OSPEED2 [1:0]		OSPEED1 [1:0]		OSPEED0 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Obr. 3.18: Registr GPIOx_OSPEEDR[1]

3.5.4 Nastave pull-up/pull-down/floating režimu pinu

Každému pinu lze interně přiřadit buďto pull-up nebo pull-down rezistor, nebo nechat pin v režimu floating, kdy není žádný rezistor připojený. Hodnota těchto pull-up či pull-down rezistorů je velmi nepřesná a jejich hodnoty se pohybují, viz dokumentace, mezi 25kΩ-55kΩ (typická hodnota je uvedena 40kΩ). Jednotlivým volbám pull-up/pull-down/floating odpovídají následující bitové hodnoty:

- Floating - 0b00
- Pull-up - 0b01
- Pull-down - 0b10

Dle výběru je potřeba dvojici bitů zapsat na pozici příslušného pinu, pro který hodnotu nastavujeme (viz obr. 3.19). Pokud bychom tedy nastavovali např. pin Px3, bude třeba zapsat na pozice 6 a 7 hodnoty dle konfigurace připojení odporu, kterou chceme využívat.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPD15[1:0]		PUPD14[1:0]		PUPD13[1:0]		PUPD12[1:0]		PUPD11[1:0]		PUPD10[1:0]		PUPD9[1:0]		PUPD8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPD7[1:0]		PUPD6[1:0]		PUPD5[1:0]		PUPD4[1:0]		PUPD3[1:0]		PUPD2[1:0]		PUPD1[1:0]		PUPD0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

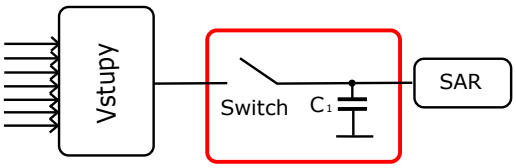
Obr. 3.19: Registr GPIOx_PUPDR[1]

3.6 ADC - analogově digitální převodník

Analogově digitální převodník (dále jen ADC) nám umožňuje převést analogové napětí do digitální podoby. V mikrokontrolérech řady STM32G031 se nachází konkrétně jeden 12-bitový (umožňuje převést rozsah vstupního napětí na 4096 hodnot) ADC s postupnou aproximací. Má možnost multiplexovat na jeho vstup až 19 separátních kanálů, z čehož 16 je vyvedeno na fyzické piny mikrokontroléru a 3 jsou interní. Jedním z interních kanálů je kanál pro měření teploty na vestavěném teplotním senzoru, další na měření referenční napětí kvůli kalibraci ADC a poslední na měření připojeného bateriového zdroje pro funkci RTC (real-time clock). Doba převodu jednoho kanálu odpovídá při 12-bitovém rozlišení 0.4μs (nezáleží při tom na nastavení frekvence), ale je možno ji snížit snížením rozlišení ADC na nižší počet bitů.

3.6.1 Princip funkce ADC

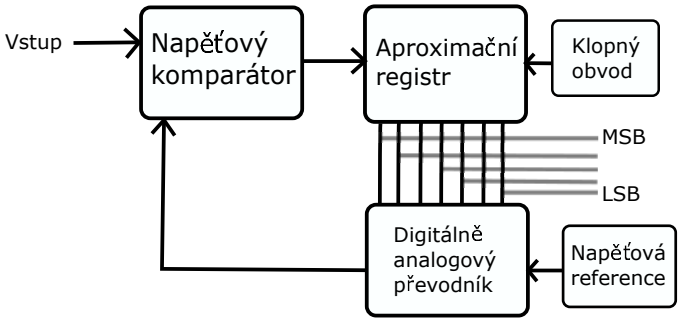
Analogově digitální převodník s postupnou aproximací funguje v N cyklech, podle toho kolika bitový je. Nejprve je nutno si však signál navzorkovat. To je docíleno pomocí Sample & Hold obvodu (viz obr. 3.20 - červená sekce). Tento obvod vždy sepne spínač, a pomocí toho je nabit kondenzátor C1. Dobu sepnutí spínače lze nastavit, a čím delší je, tím přesnější je napětí na kondenzátoru vůči vstupnímu napětí, ale snížíme tím rychlost odebírání vzorků. Pokud bychom zvolili sampling time moc krátký, můžeme způsobit velkou chybu měření, neboť se kondenzátor nestihne nabít na hodnotu napětí stejnou jako má signál, který měříme. Další možnost vzniku chyby u Sample & Hold obvodu může vzniknout, pokud bychom dali moc velký vstupní odpor před vstupem ADC, neboť ovlivňujeme proud tekoucí do kondenzátoru a tím prodlužujeme dobu, za kterou se kondenzátor nabije.



Obr. 3.20: Základní schéma ADC

Následně je již ovzorkovaný signál zpracován SAR (Succesive approximation register - viz obr. 3.21). Tento proces spočívá v použití DA převodníku (digitálně analogový převodník), který postupně generuje napětí od MSB (nejvýznamnější bit) po LSB (nejméně významný bit). Celý proces konverze začne od toho, že je nastaveno pomocí DA (digitálně-analogového) převodníku napětí odpovídající polovině maximálního vstupního napětí (1000 0000 0000), to je v napěťovém komparátoru porovnáno se vstupním napětím, a podle toho zda je větší nebo menší, je do aproximačního registru zapsána log 0, nebo 1.

Další takt je stejný jen s nižším bitem. Jakožto MSB bit DA převodníku je vzat bit z aproximačního registru, a bit o jedna nižší v DA převodníku je nastaven na 1, tudíž výsledné číslo je (0100 0000 0000 - v případě, že vstupní napětí bylo menší než 1/2), toto napětí z DA převodníku je opět porovnáno komparátorem se vstupem, a výsledek opět zapsán do aproximačního registru. Takto se cyklus bude opakovat, dokud není zkontrolováno všech 12 bitů.(znázornění cyklu viz obr. 3.21)



Obr. 3.21: Princip SAR

Dále je již digitální podoba systémů zapsána do data registru ADC a je možno k ní jakožto uživatel přistupovat a zpracovávat ji, nebo nastavit pomocí DMA, aby se zapisovala do nějaké proměnné v uživatelském programu.

3.6.2 Možnosti využití ADC

ADC lze využít v několika režimech.

- Softwarové spouštění konverze
- Spouštění konverze externím signálem
- Spouštění konverze pomocí časovače
- Analog-watchdog mód

Programově řízené spouštění konverze

Funguje v nejvíce základním režimu, kdy si uživatel vždy spustí ADC když chce změřit hodnotu napětí. Značíme jej zkratkou SW (softwarové) spouštění. Je vhodné pro měření napětí které se nemění rychle, nebo když nepotřebujeme hodnotu napětí v konkrétním bodě.

Spouštění konverze externím signálem

Místo toho, aby uživatel spouště ADC zápisem do registru v programu, ADC se spustí vždy s příchodem náběžné/sestupné (lze nastavit) hrany vnějšího připojeného signálu.

Spouštění konverze pomocí časovače

Funguje obdobně jako spouštění konverze pomocí externího signálu, ale namísto něj je využit generovaný impulz z časovače. Časovač může být nastaven aby generoval trigger signál při různých "eventech"(viz více sekce 3.9.7). Využívá se často při měření proudu, který je generován PWM (pulzně šířková modulace) výstupem časovače našeho mikrokontroléru, neboť jsme pak schopni změřit proud vždy ve stejném bodě.

Analog-watchdog mód

Tento mód funguje jako sledovač napětí. Klasické odebírání vzorků, řízené buď uživatelem, nebo nějakým externím signálem, stále funguje, ale pro ADC nastaven "treshold"(mezní hodnoty), při kterém má vyvolat přerušení a dát tak uživateli vědět že napětí překročilo určitou mez. Vzorky se tedy stále odebírají normálním způsobem, ale není třeba je programově zpracovávat a vyčítat. Tím odlehčíme práci jádru s výsledným zpracováním všech vzorků a zpracováváme jen ty, která nás zajímají (ty, které překročily uživatelem nastavenou mez). Lze to využít například pokud chceme vytvořit zařízení jako je osciloskop (kvůli spuštění vzorkování) nebo pokud chce uživatel nezávisle na běhu programu kontrolovat, zda napětí nepřesáhlo nějakou mez a nezatěžovat tím výkon jádra.

3.6.3 Nastavení ADC

ADC jakožto periferie mikrokontroléru má ve výchozím nastavení vypnutý přístup hodinového signálu, tudíž ji nelze aktivovat do doby, než zapneme v RCC_APBENR2 bit ADCEN (viz 3.1.3), který nám přivede hodinový signál k periferii a zapne ji ve výchozím nastavení.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ADCAL	Res.	Res.	ADVREGEN	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
rs			rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	ADSTP	Res.	ADSTART	ADDIS	ADEN
											rs		rs	rs	rs

Obr. 3.22: Registr ADC_CR[1]

Periférie ADC může mít chybu měření, proto je nutné před začátkem měření provést její kalibraci. To lze provést pouze pokud není samotná periférie zapnuta (viz další krok, bit ADEN v ADC_CR registru). Pro kalibraci slouží interní referenční zdroj napětí, který má přesnou hodnotu 1.212V, dle které se provede kalibrace ADC. Pro spuštění samotné kalibrace je nejprve nutné zapnout referenční zdroj napětí pomocí bitu ADVREGEN v ADC_CR registru (viz obr. 3.22). Nyní je potřeba počkat, než se výstupní napětí referenčního zdroje ustálí a je potřeba vložit pauzu několika cyklů jádra. Následně již může uživatel zapnout samotnou kalibraci ADC. To lze provést pomocí zapsání log.1 na pozici bitu ADCAL v registru ADC_CR. Nyní je spuštěná kalibrace ADC a je potřeba počkat, než bude dokončena. Po jejím skončení bude bit ADCAL opět automaticky nastaven do 0. Dle něj uživatel pozná, že kalibrace proběhla. V ADC_DR registru lze vyčíst hodnotu „offsetu“ naměřeného při kalibraci. Avšak není třeba si jej ukládat (pokud nechce uživatel přímo tuto hodnotu zjistit), protože od této doby bude výsledná hodnota měření, zapsaná do ADC_DR registru, automaticky zkorigována touto hodnotou. Výjimku tvoří situace, kdy by uživatel při běhu programu resetoval periferii ADC, neboť tím se tato hodnota vymaže a je třeba provést kalibraci znovu.

Nyní, když byla provedena úspěšná kalibrace ADC, je nutné zapnout samotnu periferii v ADC_CR registru pomocí bitu ADEN(viz obr. 3.22). Následně je třeba nastavit dobu odběru vzorku (viz obr. 3.23). Ve výchozím nastavení je nastaveno 1,5 cyklu ADC. Tato hodnota lze nastavit od 1,5 cyklu až po 160,5 cyklu hodinového signálu ADC. Hodnoty lze nastavit 2 různé (bity SMP1 a SMP2) a každému z 17 kanálů přiřadit (viz bity SMPSELx podle jednotlivých kanálů ADC) jednu z těchto dvou hodnot. Dobu odběru vzorku musíme

volit s ohledem na vnitřní odpor zdroje signálu a rychlost s jakou potřebujeme napětí měřit (viz sekce 3.6.1).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	SMPSEL18	SMPSEL17	SMPSEL16	SMPSEL15	SMPSEL14	SMPSEL13	SMPSEL12	SMPSEL11	SMPSEL10	SMPSEL9	SMPSEL8
					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMPSEL7	SMPSEL6	SMPSEL5	SMPSEL4	SMPSEL3	SMPSEL2	SMPSEL1	SMPSEL0	Res.	SMP2[2:0]			Res.	SMP1[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw		rw	rw	rw

Obr. 3.23: Registr ADC_SMPR[1]

Ještě je nutné nastavit kanály, které chceme na ADC používat. Ty lze vybrat v ADC_CHSELR registru (viz obr. 3.24). Námi vybrané kanály budou v sekvenci postupně přepínány a jeden po druhém budou převedeny pomocí ADC. Pořadí kanálů v jakém pořadí konverze proběhne, lze nastavit uživatelem specifická sekvence kanálů, nebo ve výchozím nastavení probíhá konverze vždy od kanálu s nejnižším číslem po nejvyšší.

Nyní je vše potřebné pro režim, kdy spouštíme ADC, softwarově připraveno a je třeba pouze zapsat do ADC_CR registru bit ADSTART(viz obr. 3.22). Následně proběhne konverze, a po dokončení konverze lze data vyčíst z ADC_DR registru hodnotu napětí(v rozmezí 0-4095 - 12bitů).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	CHSEL18	CHSEL17	CHSEL16
													rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CHSEL15	CHSEL14	CHSEL13	CHSEL12	CHSEL11	CHSEL10	CHSEL9	CHSEL8	CHSEL7	CHSEL6	CHSEL5	CHSEL4	CHSEL3	CHSEL2	CHSEL1	CHSEL0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Obr. 3.24: Registr ADC_CHSELR[1]

V případě že, bychom chtěli využívat ADC v režimu, kdy je konverze spouštěna externím signálem, nikoliv uživatelem v softwaru, je nejprve třeba povolit externí zdroj spouštění konverze v ADC_CFGR1 registru (viz obr. 3.25) dvojici bitů EXTEN[1:0], kde lze zvolit zda chce uživatel spouštět konverzi sám (softwarově) nebo pří náběžné, sestupné, či náběžné i sestupné hraně spouštěcího signálu. Následně je třeba ještě nastavit bity EXTSEL[2:0] v ADC_CFGR1 registru, jimiž lze nastavit zdroj externího spouštěcího signálu (buď výstup časovače, nebo externí vstup signálu z oblasti mimo mikrokontrolér).

Tímto jsme vysvětlili základní nastavení ADC, při němž je však nucen uživatel vždy sám vyčítat jeho hodnoty po konverzi z datového registru ADC. To lze nahradit využitím DMA, ale jeho nastavení probereme v samostatné kapitole (viz 3.8.1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	AWD1CH[4:0]					Res.	Res.	AWD1EN	AWD1SGL	CHSELRMOD	Res.	Res.	Res.	Res.	DISCEN
	rw	rw	rw	rw	rw			rw	rw	rw					rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AUTOFF	WAIT	CONT	OVRMOD	EXTEN[1:0]		Res.	EXTSEL[2:0]			ALIGN	RES[1:0]		SCANDIR	DMACFG	DMAEN
rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw	rw	rw

Obr. 3.25: Registr ADC_CFGR1[1]

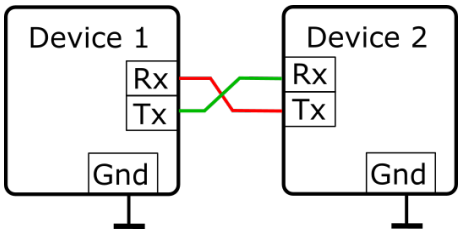
3.7 Blok sériové komunikace USART

Zkratka USART (universal synchronus/asynchronus recivertr-transmitter) značí zařízení, které je dedikované pro vzájemnou sériovou komunikaci dvou zařízení po 2 (popř. 3) vodičích. Tento blok lze najít téměř v každém mikrokontroléru. Není tomu jinak ani u mikrokontrolérů řady STM32G031. Pokud chce uživatel mít možnost k jednoduchému způsobu komunikace s jiným zařízením, je toto nejjednodušší způsob.

Nyní něco málo k principům funkčnosti. Jak již z názvu vyplývá, lze jej použít buď v synchronním, či asynchronním režimu. Zatímco asynchronní režim nevyužívá pro komunikaci hodinový signál, ale pouze Tx a Rx piny, synchronní režim má jeden vodič navíc využit k vedení hodinového signálu, a další dva pro Tx a Rx piny. V tomto režimu je pak třeba nastavit kdo bude „master“ a kdo „slave“.

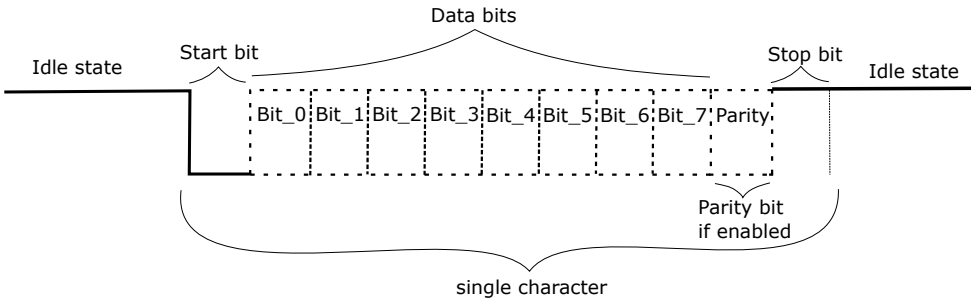
3.7.1 Asynchronní mód

V asynchronním módu je jeden (Tx - výstup) určen pro vysílání a druhý (Rx - vstup) pro přijímání. Tudíž Tx výstup USARTu jednoho zařízení musí být připojen na Rx vstup druhého zařízení a Tx výstup druhého na Rx vstup prvního (viz obr. 3.26).



Obr. 3.26: USART schéma zapojení - asynchronní mód

Jelikož v tomto režimu nemáme vedený hodinový signál spolu s datovým, je třeba nastavit tzv. „Baud rate“. Baud rate je označení pro rychlost komunikace a je udávaný v bitech/s. Tímto lze v asynchronním režimu docílit správné synchronizace bitů na straně vysílače a přijímače. Celý komunikační protokol má strukturu viz obr. 3.27.



Obr. 3.27: USART protokol - asynchronní mód

Komunikační protokol je složen z několik částí:

- Start bit
- Datové bity v pořadí od LSB (Bit_0) po MSB (Bit_7)
- Parity bit, pokud se používá

- Stop bit

Startbit je první bit , kterým je zahájena komunikace, z důvodu aby přijímač věděl, že vysílač začal přenášet data. Následně dojde k přenesení 8 datových bitů, přičemž bity jsou odesílány postupně od bitu „Bit_0“ (LSB) po bit „Bit_7“ (MSB). Po odeslání všech bitů je celý přenos ukončen stop bitem (lze nastavit i více než jeden stopbit v případě že uživatel používá rychlejší komunikaci - baud a tím docílí delší mezery mezi jednotlivými datovými segmenty), ale je třeba, aby o tomto nastavení věděla jak vysílací, tak přijímací strana, jinak by docházelo k chybám s načasováním.

Kvůli kontrole správnosti přenosu dat, lze navíc odeslat tzv. parity bit. Ten je nastaven buď do 0 (sudý parity bity), nebo do 1 (lichý parity bit). Funkce parity bitu spočívá v tom, že pokud je nastaven do 0 (sudý), tak když je posláno 9 bitové číslo (8-bitů dat + parity bit), tak ať je prvních 8bitů (data) sudé, či liché číslo je na poslední místo vždy umístěna 0, tudíž celých 9 bitů na vysílací straně je vždy sudé číslo. To je následně odesláno k přijímači a ten, když číslo přijme, tak zkontroluje počet 1 v celém čísle a pokud je jejich počet sudý, stejně jako na vysílací straně (žádná informace nebyla cestou poničena), tak přijímač ví, že bity dorazily v pořádku a můžeme je dále zpracovávat. Pokud však napočítá lichý počet 1, je jasné, že došlo k nějaké ztrátě v průběhu přenosu a číslo je zahozeno, neboť není správně přeneseno. To samé platí pokud nastavíme parity bit do 1, ale s lichými čísly, neboť když na poslední místo vložíme 1, ať je vstupní číslo jakékoliv, vždy bude liché, a tudíž bude mít lichý počet 1 v celkových 9 bitech. To je stejným způsobem spočítáno na straně přijímače, a pokud počet 1 není lichý, tak je číslo zahozeno, neboť došlo při přenosu k chybě.

Tímto způsobem kontroly budeme mít kontrolu nad přenosem a můžeme špatně přenesená data zahodit a poslat je např. znovu. O tomto nastavení musí opět vědět jak vysílač, tak přijímač, a být shodně nastaveny jinak by docházelo k chybám.

3.7.2 Synchronní mód

Tento mód je velmi podobný asynchronnímu, jen je třeba navíc vést jeden vodič s hodinovým signálem, který určuje kdy se má číst stav přenášených dat. Zařízení které generuje hodinový signál nazýváme „master“, a zařízení které se jím řídí nazýváme „slave“ Tento vodič „supluje“ nastavený baud rate na obou zařízeních a určuje jim kdy mohou číst jednotlivé bity komunikace. Tudíž nejsme již vázáni tím, že obě zařízení (vysílač i přijímač) musí mít shodně nastavený baud rate, ale řídí se dle hodinového signálu vedeného od mastera. Zároveň tímto se zbavíme nutnosti posílání start a stop bitů, neboť s přijímač začne číst s načtením prvního pulzu hodinového signálu.

Tento mód přenosu může být vhodný například pokud bychom při běhu programu chtěli měnit přenosovou rychlost, neboť nebude třeba posílat příkaz do druhého zařízení aby si změnilo přenosovou rychlost na nějakou jinou hodnotu, po které budeme momentálně vysílat.

3.7.3 Základní nastavení USART

Jakožto periferie mikrokontroléru připojená na APB sběrnici, je třeba nejprve zapnout hodinový signál pro periferii. To docílíme zapnutím USART1EN bitu v RCC_APBENR2 registru (viz 3.1.3). Následně je třeba zapnout hodinový signál GPIO portu na které je USART_1 periferie vyvedena. V našem konkretním případě u mikrokontroléru STM32G031J6 se jedná o piny PB6 a PB7, tudíž je třeba zapnout celý GPIO port B (viz sekce 3.1.3). Následně je třeba oba piny nastavit na alternativní funkci (viz 3.5.1. V tomto

konkrétním případě je alternativní funkce pinu v AFR registru namapována na hodnotě 0 ,tudíž ten již není třeba upravovat.

Tímto máme nastavené piny a hodinový signál přivedený jednak k periferii samotné, tak k výstupním pinům periferie. Nejprve si ukážeme jak nastavit USART v asynchronním režimu.

Asynchronní mód - nastavení

V tomto případě je nutné nastavit baud rate přenosu. Abychom jej nastavili správně je třeba nejprve spočítat hodnotu USART_BRR registru, jejíž hodnota nastavuje rychlost přenosu. Pro výpočet je nutné počítat s frekvencí, která je přivedena k periferii USART1 a s rychlostí, kterou chceme data přenášet (baudem). Ve výchozím nastavení je interní frekvence jádra i USART1 periferie (po zapnutí hodinové signálu) nastavena na 16MHz. Dejme tomu, že chceme komunikovat na baudu 9600 bit/s. Výslednou hodnotu dostaneme dosazením do rovnice 3.1

$$USART\_BRR = \frac{frequency}{baud} \tag{3.1}$$

Výslednou hodnotu po výpočtu je třeba zapsat do USART_BRR registru a tím nastavíme požadovanou hodnotu pro rychlost komunikace. V druhém zařízení je třeba nastavit stejnou hodnotu, aby komunikace fungovala správně.

Následně je třeba v USART_CR1 registru(viz obr. 3.28) povolit funkci vysílače a přijímače pomocí bitů RE (recive enable) a TE (transmitt enable) které nastavíme do 1. Následně již jen do stejného registru zapíšeme hodnotu 1 na pozici bitu UE (usart enable) a periferie USART v asynchronním režimu je zapnuta v základním nastavení kdy jsme schopni posílat data typu $uint8\_t$(8-bitů) jeden po druhém, zatím bez použití fronty. Pokud chce uživatel odeslat data, je třeba zapsat je do USART_TDR registru. Po zapsání jsou data automaticky pomocí periferie odeslána bit po bitu (viz obr. 3.27). Pokud chce naopak přečíst data, která přišla, je možné je vyčíst z USART_RDR registru. Tímto jsme si vysvětlili nejzákladnější nastavení USART periferie v asynchronním režimu.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RXF FIE	TXFEIE	FIFO EN	M1	EOBIE	RTOIE	DEAT[4:0]					DEDT[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	CMIE	MME	M0	WAKE	PCE	PS	PEIE	TXFNFIE	TCIE	RXFNEIE	IDLEIE	TE	RE	UESM	UE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Obr. 3.28: Registr USART_CR1[1]

3.8 Blok pro přímý přístup do paměti - DMA

Blok mikokontroléru DMA (direct memmory acces) zprostředkovává přenos dat buďto mezi registrem periferie a zvolenou oblastí paměti, přímo z nějaké oblastí paměti do jiné, nebo z periférie do periférie. Díky použití tohoto bloku, po správné konfiguraci, se již nemusí uživatel starat o vyčítání hodnot z registrů periferií (např. u ADC, či USART) a tím nebude tolik zahlcováno jádro operacemi, neboť to DMA udělá za nás. V mikrokontrolérech řady STM30G031 se nachází konkrétně DMA s 5 separátními kanály pro využití přenosu v následujících režimech:

- Z periférie do paměti
- Z paměti do periférie
- Z paměti do paměti
- Z periférie do periférie

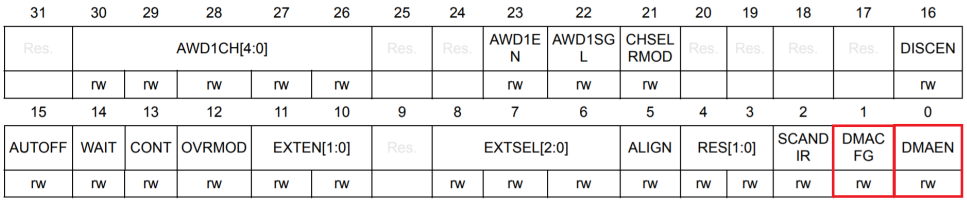
Blok DMA lze obsluhovat buďto jednotlivými požadavky na přesení dat, nebo v tzv „circular“ režimu. V režimu s jednotlivými požadavky musí uživatel vždy zahájit nový přenos určitého bloku dat softwarově, zatímco v „circular“ režimu je nový přenos vždy spuštěn znovu, a probíhá neustále, bez nutnosti spouštění pomocí softwarového příkazu. Zmíněný „circular“ režim lze však použít pouze v režimu memory-to-peripheral/peripheral-to-memory (z paměti do periférie/z periférie do paměti), nelze jej využít v případech kdy uživatel používá přenos memmor-to-memmory/peripheral-to-peripheral (z paměti do paměti/z periférie do periférie). Tento režim se hodí např. v případech, že vyčítáme naměřené hodnoty z ADC (režim z periférie do paměti), nebo naopak v případě kdy přenášíme hodnoty do periférie, např při použití komunikačního rozhraní USART (z paměti do periférie).

V této sekci se budeme věnovat základnímu nastavení DMA v jednotlivých režimech, tak aby za nás vykonávalo přenos dat z periférie do paměti a naopak. Zmíníme co vše je třeba nastavit v konfiguračních registrech DMA a zároveň jak na stavit konfigurační registry u periférií tak, aby byly schopny spolupracovat s DMA.

3.8.1 Nastavení DMA v „circular“ režimu pro periférii ADC

V této sekci se budeme zabývat nastavením bloku DMA tak, aby za uživatele přenášelo data mezi ADC_DR registrem a zvolenou oblastí paměti bez nutnosti vyčítání dat softwarově. Budeme vycházet z nastavení ADC popsané v sekci 3.6.3, pouze doplníme několik změn, které spustí přenos pomocí DMA. Zda uživatel využívá ADC v režimu spouštění pomocí softwaru, či pomocí externího spouštění, nezáleží na funkčnosti DMA, to bude v tomto případě stále přenášet nově naměřená data z datového registru ADC do zvolené oblasti v paměti.

Za předpokladu, že je ADC správně nastaveno, je třeba ještě přímo v registru ADC_CFGR1 registru zapnout funkce DMA pro periférii ADC. To provedeme pomocí dvojice bitů DMAEN a DMACFG (viz obr. 3.29). Zapsáním 1 do DMAEN bitu povolíme funkce DMA pro periférii ADC. Zapsáním 1 do DMACFG bitu nastavíme druh DMA přístupu na „circular“ mód, kdy budou nové hodnoty vyčteny vždy když ADC změří nový vzorek. Kdybychom tento bit nechali nastavený v 0, zůstal by nastavený mód na „single shot“, kdy by bylo třeba spouštět vždy nový převod dat do paměti softwarově.



Obr. 3.29: Registr ADC_CFGR1[1]

Mikrokontroléry řady STM32G031 na rozdíl od většiny ostatních (s výjimkou řad STM32H7, STM32L4+ a STM32WB) disponují blokem DMAMUX (DMA request multiplexer). Ten zajišťuje nastavení přenosu požadavků pro spouštění DMA od zvolené periférie (v našem případě ADC). Je třeba tento blok správně nastavit tak, aby DMA dostalo

požadavek po každé konverzi z ADC. Jinak by nám DMA nefungovalo, neboť by nevědělo, z jaké periferie má požadavky očekávat. Proto je třeba podívat se na registr DMAMUX_CxCR (viz obr. 3.30), kde lze toto nastavit. Písmeno x odpovídá námi zvolenému kanálu DMA, který budeme využívat. V tomto registru je třeba do skupiny bitů DMAREQ_ID zapsat ID námi využívané periférie. Tabulka ID jednotlivych periferií, viz obr. 3.31. Nyní jen zapíšeme správné ID (v našem případě 5, tudíž v binární podobě 0b101) na zmíněnou pozici v registru a celý blok DMAMUX je tímto správně nastavený.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	SYNC_ID[4:0]					NBREQ[4:0]					SPOL[1:0]		SE
			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	EGE	SOIE	Res.	Res.	DMAREQ_ID[5:0]					
						rw	rw			rw	rw	rw	rw	rw	rw

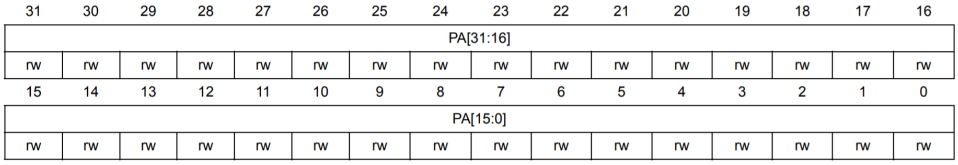
Obr. 3.30: Registr DMAMUX_CxCR[1]

DMA request MUX input	Resource	DMA request MUX input	Resource	DMA request MUX input	Resource
1	dmamux_req_gen0	22	TIM1_CH3	43	TIM15_UP
2	dmamux_req_gen1	23	TIM1_CH4	44	TIM16_CH1
3	dmamux_req_gen2	24	TIM1_TRIG_COM	45	TIM16_TRIG_COM
4	dmamux_req_gen3	25	TIM1_UP	46	TIM16_UP
5	ADC	26	TIM2_CH1	47	TIM17_CH1
6	AES_IN	27	TIM2_CH2	48	TIM17_TRIG_COM
7	AES_OUT	28	TIM2_CH3	49	TIM17_UP
8	DAC_Channel1	29	TIM2_CH4	50	USART1_RX
9	DAC_Channel2	30	TIM2_TRIG	51	USART1_TX
10	I2C1_RX	31	TIM2_UP	52	USART2_RX
11	I2C1_TX	32	TIM3_CH1	53	USART2_TX
12	I2C2_RX	33	TIM3_CH2	54	USART3_RX
13	I2C2_TX	34	TIM3_CH3	55	USART3_TX
14	LPUART_RX	35	TIM3_CH4	56	USART4_RX
15	LPUART_TX	36	TIM3_TRIG	57	USART4_TX
16	SPI1_RX	37	TIM3_UP	58	UCPD1_RX
17	SPI1_TX	38	TIM6_UP	59	UCPD1_TX
18	SPI2_RX	39	TIM7_UP	60	UCPD2_RX
19	SPI2_TX	40	TIM15_CH1	61	UCPD2_TX
20	TIM1_CH1	41	TIM15_CH2	62	Reserved
21	TIM1_CH2	42	TIM15_TRIG_COM	63	Reserved

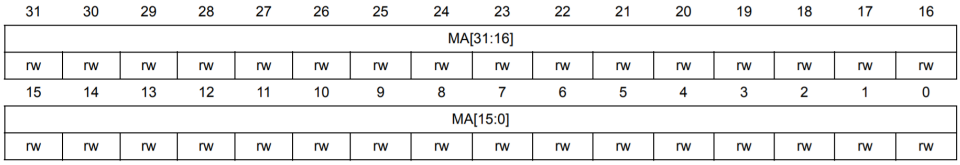
Obr. 3.31: Tabulka obsahující ID jednotlivých periférií[1]

Po tomto nastavení na straně ADC a DMAMUX je třeba správně nakonfigurovat samotný blok DMA. Nejprve je potřeba zapnout hodinový signál k celému DMA bloku. To provedeme pomocí RCC_AHBENR registru (viz 3.1.3), kde nastavíme bit DMA1EN do 1. Dále je třeba nastavit periferii DMA jako takovou. Jakožto první je třeba zadat paměťové adresy registru periferie, odkud chceme hodnoty vyčítat a adresu proměnné, kam je chceme ukládat. Paměťovou adresu periférie lze nastavit v DMA_CPARx registru (viz obr. 3.32), kde x značí číslo kanálu DMA, který budeme chtít používat. Pro zapsání adresy vložíme, v

našem případě, do registru adresu ADC_DR registru. Stejným postupem je třeba nastavit adresu proměnné/pole proměnných (v případě, že máme u ADC nastavenou sekvenci konverzí několika kanálů). Tu zapíšeme do registru DMA_CMARx (viz obr. 3.33), kde x opět značí kanál DMA, který uživatel používá. Do tohoto registru vložíme odkaz na blok paměti, do kterého bude uživatel data ukládat.

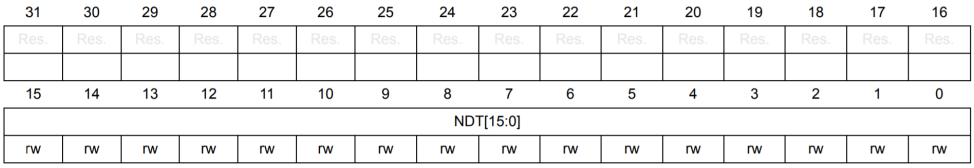


Obr. 3.32: Registr DMA_CPARx[1]



Obr. 3.33: Registr DMA_CMARx[1]

Pokud jsou obě adresy správně nastaveny, je třeba dále nastavit počet hodnot, které má celkově DMA přenést (v podstatě délku pole, kam budeme naše data ukládat, neboť pokud by byl v tomto registru špatně nastavený počet, tak by se mohlo stát, že se nám přepíšou předešlé hodnoty novými). To lze nastavit zapsáním příslušné délky do DMA_CNDTRx registru (viz obr. 3.34), kde x opět značí používaný kanál. Například pokud bychom měřili 3 kanály ADC, je tato hodnota třeba nastavit na hodnotu 3 a mít v DMA_CMARx registru vloženou adresu pole, které má 3 prvky. Tento registr je vždy na začátku nastaven na maximální hodnotu (v našem případě 3) a při každé převedené proměnné dekrementován o 1 a vždy je posunut ukazatel v poli na další segment, tím je docíleno že každý kanál ADC bude v separátní proměnné. Když tento registr dosáhne 0, je převod dalších proměnných ukončen, pokud není DMA nastaveno v „circular“ režimu. V tom případě se vždy nastaví zpět do maximální hodnoty a aktuální segment pozice v poli proměnných, kam zapisujeme hodnoty, je nastaven na pozici 0 a celý proces se opakuje.



Obr. 3.34: Registr DMA_CNDTRx[1]

Nyní je třeba nastavit správně konfigurační bity DMA v DMA_CCRx registru (viz obr. 3.35). Nejprve je nutné zde nastavit hodnotu bitu MINC do 1. Ten nám zajistí, že se bude inkrementovat pozice kam DMA zapisuje v případě, že zapisujeme více hodnot do pole proměnných. Dále je třeba nastavit bit CIRC do 1, který nám zapne funkci „circular“ režimu, kdy se DMA po skončení všech definovaných převodů (viz hodnota DMA_CNDTRx registru) spustí znovu. Ještě je zde třeba nastavit hodnoty dvojic bitů MSIZE a PSIZE. Dvojice bitů MSIZE nám určuje velikost proměnné v paměti, kam data přenášíme. Lze ji nastavit na následující hodnoty:

- 8 bitů - hodnota 0b00

- 16 bitů - hodnota 0b01
- 32 bitů - hodnota 0b10

Vzhledem k tomu, že naše ADC je 12 bitové a hodnoty budeme ukládat do 16 bitové proměnné, je třeba ji nastavit do hodnoty 0b01. Stejné je třeba nastavit i pro velikost proměnné na straně periférie, odkud budeme data brát, pouze je namísto do dvojice bitů MSIZE zapíšeme do dvojice bitů PSIZE. Zapsané velikosti odpovídají velikostem zmíněným k MSIZE. V našem případě, kdy ADC_DR registr má velikost 16 bitů je potřeba opět zapsat hodnotu 0b01. Pozor, toto číslo slouží pouze k určení velikosti jedné proměnné v celém poli, kam data ukládáme, k délce tohoto pole se vztahuje číslo zapsané v DMA_CNDTRx registru.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	MEM2 MEM	PL[1:0]		MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR	TEIE	HTIE	TCIE	EN
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Obr. 3.35: Registr DMA_CCRx[1]

Nyní už zbývá pouze zapnout samotný blok DMA pomocí EN bitu nastavením jej do 1. Nyní máme kompletně nastavený blok DMA.

3.9 Časovače

Časovač (timer) je jedna ze základních periférií, kterou lze najít v skoro každém mikrokontroléru. Jedná se o periférii, kterou lze použít v mnoha režimech. Základní funkcí časovače je inkrementovat po každých x cyklech řídícího hodinového signálu svůj registr a tím měřit uběhlý čas, vzdálenost, frekvenci, popř jinou veličinu převedenou na počet impulzů. Periférie jako taková může fungovat ve vstupním i výstupním režimu (více viz 3.9.6 a 3.9.4). Dále je možné ji využívat bez jakéhokoliv vstupu, či výstupu, a využívat ji pouze jako časovou referenci k nějakému přerušení, které periférie obsluhuje (viz 3.9.5).

V mikrokontrolérech řady STM32G031 se nachází celkem 8 samostatných binárních časovačů, viz tabulka č.1. Některé jsou svojí výbavou identické a některé odlišné, proto pro ně neexistuje jednotný návod na inicializaci a jejich používání. Výhodou je že časovače jsou všechny očíslovány (např. TIM1, TIM3, TIM14 atd.) a ve většině mikrokontrolérů od firmy ST se nacházejí vždy stejné (pozor, ne však všechny mikrokontroléry obsahují všechny čítače, pouze když se ve dvou mikrokontrolérech nachází čítač s názvem TIM2, je například jasné, že se bude jednat o časovač, který má rozlišení 32bitů i na jiném mikrokontroléru s časovačem TIM2). Tudíž když se člověk naučí pracovat s časovačem TIM2 na jednom mikrokontroléru, lze to samé uplatnit na jiném mikrokontroléru jiné řady, který také obsahuje také TIM2, a jeho nastavení a ovládání bude stejné.

Každý časovač má určitá specifika, která jej charakterizují. Jedním z hlavních parametrů časovače je počet bitů. Počet bitů u časovače určuje, jaké má rozlišení načtených impulzů. Jelikož se jedná o binární čítač, maximální hodnota čítání je vždy 2^N, kde N je rozlišení čítače. Dalším specifikem může být maximální frekvence, na které je schopen časovač pracovat, nebo také směr, kterým je schopen čítat. Tím je myšleno, jakým směrem je inkrementována proměnná při příchodu nového impulzu řídícího hodinového signálu. Čítat lze buďto směrem nahoru, dolů, nebo obousměrně (více vysvětleno v sekci 3.9.1).

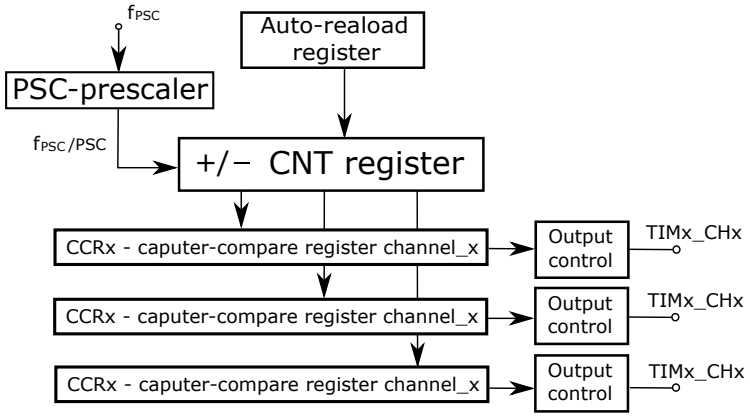
Název	Rozlišení	Maximální provozní frekvence	Směr inkrementování	Možnost generování DMA požadavku
TIM1	16 bitů	128 MHz	Nahoru, dolů, nahoru/dolů	Ano
TIM2	32 bitů	64 MHz	Nahoru, dolů, nahoru/dolů	Ano
TIM3	16 bitů	64 MHz	Nahoru, dolů nahoru/dolů	Ano
TIM14	16 bitů	64 MHz	Nahoru	Ne
TIM16	16 bitů	64 MHz	Nahoru	Ano
TIM17	16 bitů	64 MHz	Nahoru	Ano
LPTIM1	16 bitů	64 MHz	Nahoru	Ne
LPTIM2	16 bitů	64 MHz	Nahoru	Ne

Tabulka 1: Seznam časovačů

Časovače v mikrokontrolérech řady STM32G0 jsou jednou z nejkomplexnějších periférií v celém mikokontroléru. Jejich možnosti nastavení a způsobů, jak inicializovat časovače pro různé případy, je velmi mnoho, proto v této kapitole nebudeme moci pokrýt celou problematiku časovačů, ale jen ty nejdůležitější způsoby, jak je lze inicializovat a využít.

3.9.1 Princip činnosti časovače

Obecná funkce časovače spočívá v inkrementování určitého registru při příchodu náběžné nebo sestupné hrany řídícího hodinového signálu. V principu je řídící hodinový signál f_{PSC} (viz obr. 3.36) přiveden k bloku PSC (prescaler). Ten slouží jakožto dělička vstupního signálu f_{PSC} pro časovač. Z bloku PSC vystupuje hodinový signál f_{PSC} vydělený hodnotou nastavenou v PSC bloku, tedy $\frac{f_{PSC}}{PSC}$. Ten je následně přiveden do čítače samotného (CNT). Zde je s každým příchodem nového impulzu inkrementována/dekrementována hodnota CNT registru. V tomto registru je přímo uložena hodnota, kolik cyklů hodinového signálu bylo zaznamenáno.

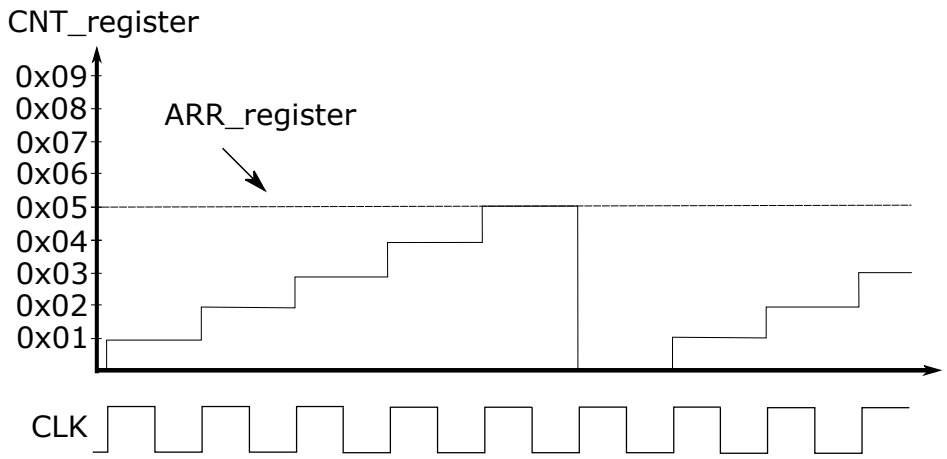


Obr. 3.36: Princip činnosti časovače

Na CNT registr je napojen Auto-reload registr (ARR). Ten umožňuje uživateli nastavit hodnotu, kdy se má CNT registr opět vynulovat a začít načítat hodnoty znovu, v případě čítání nahoru nebo dolů (viz obr. 3.37 - čítání nahoru), popř začít čítat zpět do nuly, viz čítání nahoru/dolů. Nastavením tohoto registru tudíž určuje kapacitu, kolik impulzů chce

uživatel maximálně načíst před vynulováním celého čítače. Směr čítání CNT registru lze nastavit na hodnoty nahoru, dolů, nahoru/dolů (pokud to časovač umožňuje - viz tab. 1). Tím změníme, zda se CNT registr bude inkrementovat od 0 do hodnoty ARR registru, nebo naopak dekrementovat od hodnoty ARR registru až k nule. Poslední možností je čítání nahoru/dolů. To nastaví časovač do režimu, kdy nejprve čítá impulzy do hodnoty ARR registru a následně je dekrementuje k nule.

Výstup ze samotného bloku čítače CNT může být poté napojen na bloky capture-compare. Ty mohou pracovat buďto v režimu vstupu, či výstupu. V režimu vstupu se do jejich registru zapíše hodnota CNT registru vždy při příchodu náběžné/sestupné hrany vnějšího signálu připojeném k mikrokontroléru, tím lze realizovat například měřičky frekvence, či měřičky délky pulzu. V režimu výstupu spočívá jejich funkce v tom, že při překročení určité hodnoty, čítače nastaví na své výstupu danou logickou hodnotu, nebo stávající hodnotu invertují. Ten může být následně převeden na fyzický výstup mikrokontroléru (způsob pulsně šířkové modulace - PWM), nebo při použití One-pulse módu na výstup jednoho pulzu přesně definované délky.



Obr. 3.37: Schéma čítání nahoru - up

3.9.2 Možnosti využití časovače

Ne všechny časovače lze využívat ve všech režimech. Jak bylo již zmíněno, je třeba si dát pozor při volbě časovače, zda je správně vybaven, aby splňoval požadavky výbavy, která je pro aplikaci potřeba. Například pokud chce uživatel využívat časovač pro výstup signálu PWM je potřeba, aby časovač disponoval CCR (capture-compare) blokem. Zde je několik příkladů, pro jaké různé aplikace lze časovač využít.

- Využití pro časovou referenci
- Výstup jednoho pulzu definované délky
- Výstup PWM signálu
- Vstup enkodéru
- Vstup pro čítání impulzů z vnějšího zdroje

Využití časovačů přináší uživateli několik možností. Jednak je lze využít jakožto časovou referenci, kdy při přivedení hodinového signálu při každém přetečení ARR registru se vyvolá přerušení (update-event) a tím lze nějaká rutina mikokontroléru provádět periodicky po stejných intervalech, např. 1ms, zatímco v čase mezi může jádro vykonávat jiné úlohy. Dále je lze využít pro generování PWM signálu pro řízení otáček motoru, svit LED diody, nebo jakéhokoliv jiného zařízení, které je ovládáno velikostí napětí. Dále lze pomocí čítačů zpracovávat signál z enkodéru, nebo realizovat čítačku impulzů pro měření frekvence. Navíc samotnou funkcí čítače není nijak zatěžován výkon jádra a tím zbude více výpočetního výkonu na vykonávání uživatelského program, který se nemusí starat o funkce, které časovač zastupuje.

3.9.3 Základní nastavení a spuštění časovače

Jako každý blok mikrokontroléru, tak i časovač potřebuje pro svoji funkci nejprve povolit přístup hodinového signálu k periférii. To lze provést pomocí RCC_APBENRx registru (viz sekce 3.1.3). Zde je třeba na pozici bitu TIMxEN (x značí uživatelem zvolený časovač) zapsat log.1, a tím bude k časovači přiveden hodinový signál a zapne se (spouštěcí signál časovače lze nastavit separátně, ale ve výchozím nastavení časovače je tento signál k němu přivedený brán zároveň jako spouštěcí).

Nyní jsme zapnuli samotnou periférii daného časovače, nyní je třeba časovač nastavit a spustit jeho činnost. Každý časovač má toto nastavení trochu odlišné, ale u všech je postup velmi podobný, jen jsou jednotlivé časovače různě limitovány. Pro příklad si toto základní nastavení a spuštění periférie předvedeme na časovači TIM3 (specifikace časovače viz tab.1).

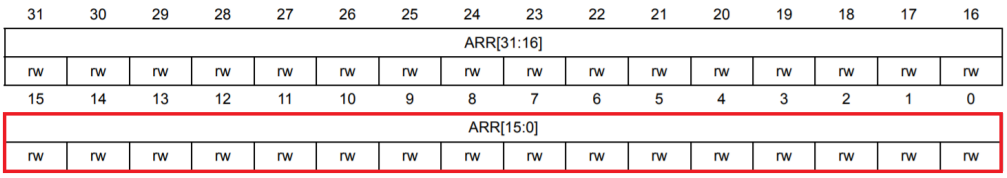
Nejprve je nutné nastavit vstupní před-děličku (pre-scaler) signálu v TIM3_PSC registru (viz obr. 3.38). Zapsáním daného 16-bitového čísla nastavíme frekvenci vstupního hodinového f_{trig} signálu dle rovnice č. 3.2. Tudíž pokud bychom měli nastavenou frekvenci hodinového signálu na APB sběrnici na hodnotu 64MHz, tak frekvence signálu f_{trig} bude podělena hodnotou $TIM3\_PSC + 1$.

$$f_{trig} = \frac{f_{apb}}{TIM3\_PSC + 1} \tag{3.2}$$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Obr. 3.38: Registr TIM3_PSC[1]

Dále je potřeba nastavit TIM3_ARR registr (viz obr. 3.39). Ten nám určuje, do jaké hodnoty bude čítač počítat (ve výchozím nastavení čítače fungují v režimu čítání „up“, tudíž od nuly, k hodnotě ARR registru). Pokud bychom používali čítač v režimu, kdy je buzen periodickým signálem, například hodinovým signálem jádra, tak tento registr, spolu s hodnotou předděličky (registr TIMx_PSC) nám vpodstatě určuje „frekvenci“, s jakou mikrokontrolér dosáhne své maximální hodnoty čítání. Vzhledem k tomu, že časovač TIM3 je pouze 16-bitový, tak horní polovina registru není dostupná. Nejvyšší možné číslo, do kolika umí časovač počítat je $2^{16} - 1$.



Obr. 3.39: Registr TIM3_ARR[1]

Nyní máme časovač správně nastavený v režimu, kdy se s určitou frekvencí naplňuje CNT registr až do hodnoty ARR. Pak dojde k vytvoření „Update-eventu“, kdy časovač „přeteče“ a čítání začne znovu od 0.

3.9.4 Zapnutí čítače v režimu výstupu PWM

Jednou ze základních aplikací pro využití čítače je generování PWM signálu s určitou střídou a frekvencí. Ten pak lze použít pro buzení různých zařízení. Lze pomocí něj například měnit intenzita svitu LED diody, měnit otáčky DC motoru, generovat zvuk pomocí piezo bzučáku a mnoho dalšího.

Střída hodinového signálu nám určuje, po jakou část jedné periody se výstup nachází v log.1 (high) a jakou část v log.0 (low). Toto pomezí, kdy dojde k překlopení signálu ze stavu „high“ do stavu „low“, nám určuje hodnota CCRx registru, kde x značí číslo kanálu, který využíváme. Frekvenci PWM signálu nám určuje naopak kombinace registrů TIMx_PSC a TIMx_ARR. Například pokud bychom chtěli vytvořit signál o frekvenci 1kHz, bude potřeba zajistit, aby se ARR registr naplnil 1000x za 1s. To docílíme (za předpokladu, že frekvence hodinového signálu přivedeného k časovači je 64MHz) nastavením registru TIMx_PSC na hodnot 63 a následným nastavením TIMx_ARR registru na hodnotu 999. Výsledná frekvence PWM signálu (f_{update}) lze vyjádřit dle rovnice 3.3, kde proměnná f_{apb} označuje frekvenci hodinového signálu přivedenou k časovači skrze APB sběrnici, ve výchozím nastavení se jedná o frekvenci hodinového signálu pro jádro (f_{clk}).

$$f_{update} = \frac{f_{apb}}{(TIMx\_PSC + 1) \times (TIMx\_ARR + 1)} \tag{3.3}$$

Nyní, po správném nastavení časovače na žádanou frekvenci, je potřeba zapnout výstup a nastavit blok „Capture-compare“ tak, aby byl výstup promítnut na výstupní pin mikrokontroléru. Nejprve musíme zapnout daný GPIO port v RCC_IOPENR registru. Následně bude potřeba nakonfigurovat vstupní/výstupní bránu GPIO do režimu alternativní funkce na pinu, kde budeme chtít mít k dispozici PWM signál. To provedeme (viz sekce 3.5) nastavením pinu do alternativní funkce pomocí registru GPIOx_MODER a následné vybráním správné alternativní funkce v GPIOx_AFR registru. V našem konkrétním případě je kanál 2 časovače TIM3 přiveden na pin PA7, tudíž se jedná o registr GPIOA_MODER (viz 3.13), kde zapíšeme na pozici pinu PA7 hodnotu 0b10. Dále je potřeba vybrat správnou alternativní funkci. Jelikož se jedná o pin PA7, tak budeme nastavovat hodnotu do GPIOA_AFRL registru (registr pro prvních 8 pinů, pro dalších 8 pinu je registr GPIOA_AFRL). Na pozici pinu PA7 zapíšeme tedy hodnotu 0b001, neboť alternativní funkce TIM3_CH2 se nachází pod touto kombinací.

Nyní máme nastavenu vstupní/výstupní bránu a je potřeba správně nakonfigurovat blok „Capture-compare“. Nejprve je potřeba se podívat na registr TIM3_CCMR1 registr (viz obr. 3.40). Do něj je potřeba zapsat hodnotu 0b0110 na pozici bitů OC2M. To nám

zapne „Caputure-compare“ jednotu v režimu výstup PWM signálu, kdy výstup je v log.1 do té doby, než čítač dosáhne hladiny CCRx registru a následně bude výstup „převrácen“ do log.0 (od hodnoty CCRx registru až do konce čítání - hodnota ARR registru). Následně začne čítač znovu čítat od 0 se zapnutým signálem (log.1) atd..

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	OC2M [3]	Res.	Res.	Res.	Res.	Res.	Res.	Res.	OC1M [3]
							rw								rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2CE	OC2M[2:0]			OC2PE	OC2FE	CC2S[1:0]		OC1CE	OC1M[2:0]			OC1PE	OC1FE	CC1S[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Obr. 3.40: Registr TIM3_CCMR1[1]

Nyní máme správně nakonfigurovanou jednotu „Capture-compare“ a musíme povolit její funkci. K tomu slouží registr TIM3_CCER (viz obr. 3.41), kde můžeme jednotlivé kanály povolit. Pro správnou funkci povolíme funkci capture-compare zapsáním log.1 na pozici bitu CC2E. Nyní už může uzivatel zapsat libovolnou hodnotu (v rozmezí 0-TIM3_ARR) do registru TIM3_CCR2 a tato hodnota nám určí, kdy se nám „překlopí“ signál z log.1 do log.0, jinak řečeno, tímto registrem nastavujeme střídu PWM signálu. Kdybychom tedy do tohoto registru zapsali například hodnotu 499 (registr ARR máme nastavený na hodnotu 999), budeme mít na výstupu signál PWM s 50% střídou.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Res.	CC4P	CC4E	CC3NP	Res.	CC3P	CC3E	CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E
rw		rw	rw	rw		rw	rw	rw		rw	rw	rw		rw	rw

Obr. 3.41: Registr TIM3_CCER[1]

Pozor, u časovačů, které mají zabudovaný tzv. „Brek-input“ (jedná se o blok, který po přivedení log.1 na jeho vstup vypne kompletně výstup časovače), například časovač TIM1 v našem mikrokontroléru, je potřeba do registru TIMx_BDTR na pozici bitu MOE zapsat log.1, jinak by výstup signálu nefungoval. Tento „Break-input“ pin se využívá například pro detekci příliš vysokého proudu u řízení elektromotorů, kdy je před tímto pinem zařazen komparátor a analogově se tak hlídá hodnota proudu (převedeného na napětí) s nějakou referenční hodnotou, a když tato hodnota překročí referenční limit, je právě tento bit (MOE) vypnut a tím se vypne celý výstup signálu z časovače a lze tak zamezit poškození motoru jeho spálením.

3.9.5 Nastavení časovače, jako časové reference

Představme si, že máme nějakou funkci (rutinu), kterou chceme provádět pravidelně v nějakém přesném časovém intervalu. V tu chvíli je pro přesnou časovou referenci potřeba spustit časovač v režimu, kdy nám bude generovat přerušení s určitou časovou periodou. K tomuto přerušení pak přiřadíme naši rutinu, která se bude periodicky vykonávat se stejnými časovými odstupy. Pro tento příklad budeme opět využívat časovač TIM3, kde k časové referenci budeme využívat „přetečení“ ARR registru - tzv. „Update-event“.

Postup pro základní nastavení čítače použijeme ze sekce 3.9.3 a hodnoty nastavíme stejné jako v příkladu nastavení PWM (viz 3.9.4). Tudíž hodnota TIM3_PSC registru bude nastavena na 63 a hodnota TIM3_ARR registru na hodnotu 999, což nám dá výslednou frekvenci „Update-eventu“ 1kHz (viz 3.3).

Pokud jsme takto časovač správně nastavili a spustili, je potřeba zapnout generovaní přerušení při „Update-eventu“. To nastavíme pomocí bitu UIE v registru TIM3_DIER (viz obr. 3.42). Nyní máme správně nastavené generování přerušení od periférie a je třeba jej zpracovat pomocí bloku NVIC. Tento blok má již předhotovené funkce pro nastavení a zpracování přerušení od periférií (viz sekce 3.2).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	Res.	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res.	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

Obr. 3.42: Registr TIM3_DIER[1]

Pokud budeme chtít využít a zpracovávat kus kódu při přerušení, je potřeba si jej v bloku NVIC povolit. To provedeme pomocí dvojice funkcí NVIC_EnableIRQ a NVIC_SetPriority. Funkce NVIC_EnableIRQ má jako parametr zdroj přerušeni, které chceme nastavit, v našem případě se bude jednat o TIM3_IRQn a funkce NVIC_SetPriority má parametr opět zdroj přerušení, a druhým parametrem je priorita přerušení, tu nastavíme na 0 (nejvyšší priorita). Následně už stačí pouze zadefinovat funkci TIM3_IRQHandler(), ve které se začne vykonávat uživatelský kus kódu při vyvolaní přerušení od časovače. Podrobnější popis přerušení viz 2.5 a 3.2.

```
void TIM3_IRQHandler(void);
NVIC_EnableIRQ(TIM3_IRQn);
NVIC_SetPriority(TIM3_IRQn,1);
void TIM3_IRQHandler(void)
{
TIM3->SR = 0x0;
//user_code
}
```

Pozor, při obsluze přerušení nesmí uživatel zapomenou vynulovat „interrupt-flag“ (viz řádek TIM3->SR = 0x0) od přerušení, který značí, že bylo přerušení vytvořené, jinak by se přerušení po ukončení rutiny vyvolalo znovu. Pokud by chtěl uživatel zapnout jiný zdroj přerušení, který se bude generovat z časovače TIM3, stačí pouze zapsat na pozice příslušných přerušení v registru TIM3_DIER log.1 a přerušení se začne spouštět, přičemž se začne vykonávat vždy kód ve funkci TIM3_IRQHandler().

3.9.6 Nastavení čítače v režimu input-capture

Čítač v režimu input-capture funguje tak, že pokud je na kanál čítače přiveden nějaký signál, tak časovač při detekci náběžné, sestupné, nebo náběžné i sestupné hrany si zapíše „time-stamp“ (údaj o tom , kdy byl signál zaznamenán) do TIMx_CCRy registru a následně může, či nemusí, vyvolat přerušení. Tímto způsobem lze tedy realizovat různá měřící zařízení pro měření buďto délky jednotlivých pulzů signálu, nebo frekvenci nějakého periodického signálu.

V tomto režimu obvykle není nutné nastavovat před-děličku (PSC), ani ARR registr u časovač, neboť budeme pouze zpětně ze změřených „time-stampů“ dopočítávat rozdíly náběžných/sestupných hran, abychom získali výslednou hodnotu frekvence. Nejprve je však nutné nastavit vstupní pin GPIO do režimu alternativní funkce a vybrat správnou alternativní funkci z nabídky. To provedeme stejně jako v minulém případě (3.9.4), nebo viz sekce 3.5 s podrobným popisem.

Nyní je potřeba správně nastavit časovač. Pro tento příklad si ukážeme nastavení konkrétně na časovači TIM2, kanálu č. 2. Budeme předpokládat, že máme správně nasta-

vený příslušný GPIO pin (který odpovídá danému kanálu našeho časovače) do alternativní funkce a vybranou správnou alternativní funkci. Nyní je potřeba přivést hodinový signál k časovači pomocí RCC_APBENR1 registru (viz 3.1.3). Když máme toto hotové, je potřeba nakonfigurovat správně čítač.

Čítač do režimu „Input-capture“ nastavíme pomocí registru TIM2_CCMR1, ale tentokrát jej budeme využívat pro „Input-capture“ mód, nikoliv „Output-compare“, jako tomu bylo u generování PWM signálu. Tento registr má totiž dva různé režimy. Jednak pro výstup signálu, a druhak pro vstup signálu. To, zda se dané bity registru chovají jakožto registry pro nastavovaní vstupu, nebo výstupu, změníme pomocí bitů CCxS (x značí číslo kanálu, pro který to nastavujeme) ve stejném registru TIM2_CCMR1. Když na jejich pozici zapíšeme hodnotu 0b01, tak se pomocí ostatních bitů (vztažených ke kanálu x, nikoliv v druhému kanálu, v našem případě kanál č.2) budou nastavovat funkce pro vstup. Jinými slovy, pomocích těchto bitů nastavujeme, zda se kanál bude chovat v režimu vstupu, či výstupu. Po zapsání hodnoty na pozici zmíněných bitů se změní struktura registru (pouze část s konfiguračními bity pro kanál č.2, bity pro kanál č.1 zůstanou stejné, viz obr. 3.40) na strukturu viz obr. 3.43.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IC2F[3:0]				IC2PSC[1:0]		CC2S[1:0]		IC1F[3:0]				IC1PSC[1:0]		CC1S[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Obr. 3.43: Registr TIM2_CCMR1 - režim vstupu[1]

Následně je možné na pozice bitů IC2PSC zapsat hodnotu „děličky“, která určuje , po kolikáté náběžné/sestupné hraně se má generovat „Capture-event“. Tímto lze efektivně načítat každou x-tou náběžnou, či sestupnou hranu nějakého signálu. Následně pomocí bitů IC2F lze nakonfigurovat vstupní filtr. Ten funguje tak, že po jeho nastavení je potřeba N stejných vzorků (N odpovídá nastavení cyklu), které budou v pořadí za sebou, aby bylo jisté, že nešlo o zákmit, ale opravdu o námi žádanou náběžnou/sestupnou hranu. Zároveň jinou kombinací bitů se mění i vzorkovací frekvence (více viz referenční manuál k k procesorů STM32G031). V našem případě nastavíme tento filtr na hodnotu 0b0011, která nám nastaví vzorkovací frekvenci stejnou, jaká je přivedena k časovači a počet po sobě jdoucích stejných vzorků (tedy číslo N) musí být 8.

Následně musíme zapnout kanál časovače, který používáme. ten zapneme pomocí bitu CC2E v registru TIM2_CCER (viz obr. 3.44). Následně pomocí dvojice bitů CC2P a CC1NP nastavíme zda má ke „Caputere-eventu“ dojít při náběžné, sestupné, či náběžné i sestupné hraně signálu přivedeného ke kanálu časovače. Pozor, tyto dva bity se chovají odlišně v režimu výstupu, kdy se pomocí nich nastavuje polarita výstupního signálu z časovače. Kombinace jsou následující, kde x značí kanál, který uživatel využívá:

- Náběžná hrana - CCxNP=0, CCxP=0
- Sestupná hrana (invertovaný výstup) - CCxNP=0, CCxP=1
- Náběžná i sestupná hrana - CCxNP=1, CCxP=1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Res.	CC4P	CC4E	CC3NP	Res.	CC3P	CC3E	CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E
rw		rw	rw	rw		rw	rw	rw		rw	rw	rw		rw	rw

Obr. 3.44: Registr TIM2_CCER - režim vstupu[1]

Nyní už zbývá pouze zapnout časovač pomocí bitu CEN v registru TIM2_CR1. Pokud jsme vše správně nastavili, měl by se nám při každém příchodu náběžné/sestupné hrany signálu (dle volby CCxP a CCxNP bitů) zapsat „time-stamp“ do CCRx registru (v našem případě CCR2) v časovači. Nyní je potřeba jej zpracovat, nejlépe hned po „caputer-eventu“. Proto si zapneme generovaní přerušení od „capture-eventu“ v TIM2_DIER registru pomocí bitu CC2IE a nastavíme odpovídající přerušení do NVIC bloku (viz 3.9.4, pouze namísto „update-eventu“ zvolíme CCR2 „event“ a namísto TIM3_IRQn nastavíme blok NVIC pro přerušení od TIM2_IRQn a to samé uděláme i s funkcí v našem kódu, která se bude volat, tudíž TIM2_IRQHandler).

Tímto máme plně připravený mikrokontrolér na aplikace pro měření délky impulzu, či čítání frekvence. Více ohledně reálné aplikace pro měření frekvence a délky pulzu signálu viz sekce 4.8.

3.9.7 Časovač jako zdroj „trigger“ signálu

Časovače lze, kromě jiného, využívat i jako zdroj „trigger“ (spouštěcího) signálu k jiným periferiím. Generovaní tohoto spouštěcího signálu lze nastavit na různé „eventy“ které v časovači mohou nastat (například „Update event“, „Capture-compare“ event, nebo při spuštění časovače). Při správné konfiguraci je při některém z „eventů“ vyslán signál na TRGO (trigger-output) výstup časovače, odkud je možné jej pomocí ostatních periférií využívat ke spouštění různých funkcí.

Základní postup na zapnutí čítače je stejný jako v sekci 3.9.3. Pro zapnutí „trigger“ signálu musíme pouze správně nakonfigurovat „master“ funkce časovače. Tato funkce určuje, jakým způsobem se má časovač chovat jako „master“, čili jako blok, který je nadřazený jinému bloku a řídí jej. Příklad nastavení si ukážeme opět na časovači TIM3 a budeme předpokládat, že je v základním nastavení již spuštěn. Nyní je potřeba pouze zapsat správnou hodnotu do registru TIM3_CR2 na pozici bitů MMS (viz obr. 4.1).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TI1S	MMS[2:0]			CCDS	Res.	Res.	Res.
								rw	rw	rw	rw	rw			

Obr. 3.45: Registr TIM3_CR2[1]

V této sekci jsme si popsali základní nastavení a použití jednotlivých časovačů. Možností využití lze nalézt mnohem víc, než jsem zmínil, ale vzhledem k tomu, že časovače celkově jsou velmi obsáhlé a multifunkční periférie, tak je nelze v krátkosti celé popsat, a tím zmínit veškerá využití, na které je lze aplikovat. Některé příklady reálných aplikací s časovači lze nalézt v sekci 4.

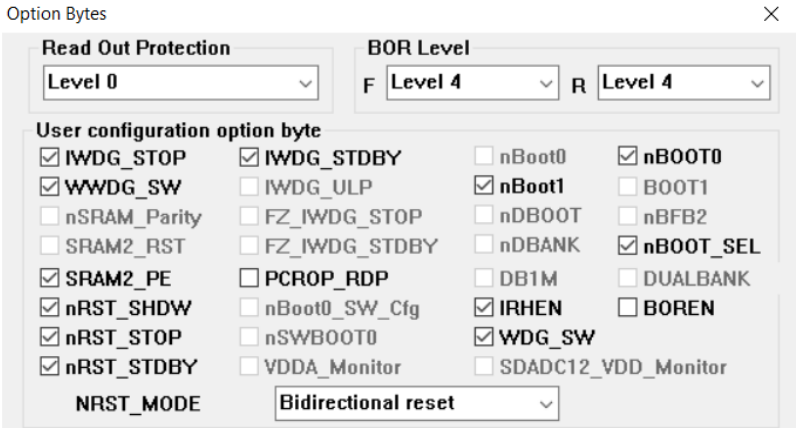
4 Základy práce s mikokontrolérem

V této sekci se budeme zabývat základní prací s mikokontrolérem, vysvětlíme jak začít programovat na souboru drobných příkladů k jednotlivým periferiím, a jak přistoupit k realizaci složitějších aplikačních úloh. Zmíníme jak efektivně využívat dostupný hlavičkový soubor s definicemi adres jednotlivých bloků, periférií a bitů.

4.1 Možnosti programování mikrokontroléru

Procesory řady STM32G031, stejně jako všechny ostatní procesory řady STM32 je možné naprogramovat více různými způsoby. Buď přes SWD, nebo s využitím bootloaderu. Programování využitím SWD sebou nese jisté výhody, jako je možnost debuggování, ale je třeba programovat procesor pomocí ST-Linku. Jedná se o modul sloužící k debuggování, a tudíž je to již součástka navíc, v provozu nepotřebná. Naopak díky internímu bootloaderu není třeba k naprogramování žádné externí součástky. Procesor lze naprogramovat jak pomocí USART (RS232-pozor, je třeba upravit napěťové hladiny na 3V3), tak díky I^2C nebo SPI.

Celkový postup pro nastavení option bitů pro nastavení mikrokontroléru, aby jej šlo programovat pomocí rozhraní USART a ovládání vstupu do boot režimu pomoci BOOT0 pinu je jednoduchý. Z výroby však není tento režim bootování nastavený. Proto musí uživatel nejprve mikrokontrolér připojit k PC pomocí SWD, například pomocí Nucleo kitu, a nastavit dané bity (viz kapitola 3.3) aby bylo možné bootovat pomocí USART s využitím BOOT0 pinu. Popř. může uživatel ještě vypnout funkce reset tlačítka a tím získat další pin navíc. Po připojení pomocí ST-Link (SWD) následně uživatel může zapnout aplikaci, např „STM32 ST-LINK Utility“ nebo „STM32CubeProgrammer“, kde se k mikrokontroléru připojí a vybere nastavení option bitů, kde vše nastaví jak potřebuje a uloží do mikrokontroléru pro budoucí použití.



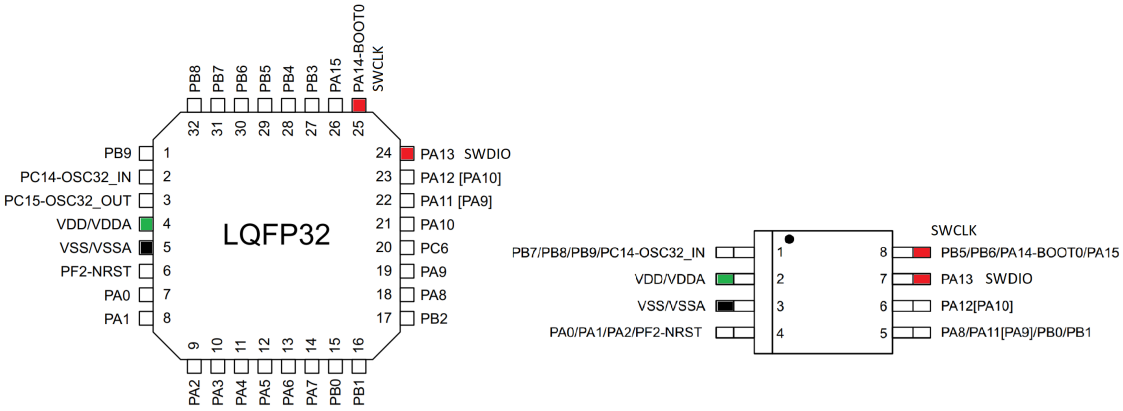
Obr. 4.1: Ukázka nastavení v programu „STM32 ST-LINK Utility“[4]

4.2 Možnosti ladění mikrokontrolérů řady STM32G031

Ladění uživatelského kódu (debuggování) lze provádět skrze ST-Link pomocí SWD protokolu pro komunikaci s čipem. Pro ladění za běhu programu je potřeba ponechat piny, na kterých je vyvedeno SWD, pro komunikaci s mikrokontrolérem. Tyto funkce jsou lokalizovány v procesorech řady STM32G031 na pinech PA13 a PA14. Po jejich připojení (pin PA13 - SWDIO a pin PA14 - SWCLK) je možné pomocí debuggeru na PC ladit kód,

vyčítat hodnoty z paměti, nebo do paměti přímo zapisovat. Většina dostupných IDE pro programování mikrokontroléru založených na jádře ARM podporuje ladění skrze ST-Link.

Vzhledem k tomu, že náš mikrokontrolér (STM32G031J6) má pouze 8 fyzických pinů, tak je vhodné pro vyvíjení nového softwaru pro tento čip použít jeho verzi s 32 fyzickými piny (STM32G031K8). Tyto dva mikrokontroléry mají totožné periférie a jejich rozložení na jednotlivých GPIO portech. Liší se pouze v počtu vyvedených fyzických pinů a velikosti paměti FLASH (32 pinové pouzdro má 64kB paměti FLASH, na rozdíl od 32kB). Výhodou je, že můžeme na větším pouzdře implementovat funkce mikrokontroléru na piny, které má vyvedené pouzdro s 8 piny a zároveň s tím využívat funkce ladění skrze SWD. Následné pouze zdrojový kód, který je již odladěny a stabilní, nahrajeme do menšího pouzdra, pomocí USART bootloaderu, kde již pro samotnou funkčnost není potřeba SWD využívat.



Obr. 4.2: Pouzdra LQFP32 a SO8N a jejich rozložení pinů[2]

4.3 Využívání dostupného hlavičkového souboru k programování

Ke každému mikokontroléru z platformy STM32 existuje hlavičkový soubor se zadefinovanými adresami jednotlivých registrů periférií a bloků včetně pozic jednotlivých bitů. V našem konkrétním případě se jedná o soubor „stm32g0xx.h“ který je potřeba přidat do našeho projektu (ať už v prostředí Keil, IAR, či jakémkoliv jiném). Pokud jej pak pomocí klíčového slova „#include“ zmíníme na začátku našeho programu budeme mít všechny definice v něm k dispozici. Celý tento proces velmi usnadní programovaní mikrokontroléru a hlavně se stane kód přehlednějším, než kdybychom vše zapisovali přímo na adresu paměti.

Celý .h soubor je v podstatě soubor struktur jednotlivých periférií. Jednotlivé periférie a bloky mají každá svoji unikátní strukturu a v ní jsou obsaženy všechny její registry jakožto 32bitové proměnné. Dále obsahuje definice jednotlivých bitů v registru, které mají pro každý bit registru vždy 1 v bitové podobě na správné pozici oproti danému registru. Ty nejsou již ve strukturách, ale každá má svůj unikátní název. Navíc každá bitová definice má v názvu nejprve název celého bloku, následně název registru a následně název bitu, na jehož pozici je v definici 1. Například pokud bychom chtěli zapisovat do registrů bloku RCC, tak k jednotlivým registrům přistoupíme následovně.

```
RCC->APBENR2 |= RCC_APBENR2_TIM1EN;
```

Tímto řádkem jsme například nastavili bit, který povolí hodinový signál pro periférii TIM1. Díky bitovému operátoru „|=“ (logický operátor OR) jsme ovlivnili pouze bity, které

jsou v definici „RCC_APBENR2_TIM1EN“ jako 1. Když se totiž podíváme na definici „RCC_APBENR2_TIM1EN“, zjistíme, že se jedná o číslo 0x00000800, což je v binární soustavě 0b1000 0000 0000. Má 1 pouze na 12. pozici(viz obr. 4.3), což odpovídá pozici pro zapnutí periférie TIM1. Tudíž v celém registru jsme přepsali pouze 12. bit a všechny ostatní nechali v původní podobě a nijak je neovlivnili.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	ADC EN	Res.	TIM17 EN	TIM16 EN	TIM15 EN
											rw		rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TIM14 EN	USART1 EN	Res.	SPI1 EN	TIM1 EN	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	SYS CFG EN
rw	rw		rw	rw											rw

Obr. 4.3: Registr RCC_APBENR2[1]

Pokud bychom chtěli daný bit nastavit do nuly a periférii tím vypnout, přičemž bychom nechtěli narušit obsah zbylých bitů v registru, je třeba požít bitový operátor „&=“ (logický operátor AND). Vzhledem k tomu, že definice bitu má na dané pozici bitu 1 a všude jinde 0 a mi potřebujeme přesný opak, je třeba celou definici znegovat. To lze udělat operátorem „~“. Nyní si ukážeme přímo řádek kódu, který nastaví daný bit do 0 a všechno ostatní nechá nezměněné.

```
RCC->APBENR2 &= ~(RCC_APBENR2_TIM1EN);
```

Naprosto stejný postup lze aplikovat na všechny bloky a periférie mikokontroléru. Celkové psaní programu pomocí těchto definici ho činí čitelnějším a mnohem lépe si člověk uvědomí, co který řádek dělá. Ano, člověk si může napsat i vlastní hlavičkový soubor s definicemi adres jednotlivých periférií, ale je to zbytečně složité a nepraktické, když už řešení existuje a je všem k dispozici.

4.4 Nastavení frekvence hodinového signálu jádra

Pro různé aplikace je potřeba různý výpočetní výkon, proto je k dispozici možnost jak změnit frekvenci hodinového signálu bez využití externího oscilátoru. V našem mikrokontroléru je interní HSI oscilátor o frekvenci 16MHz. Chceme nastavit frekvenci jádra 64MHz, což je maximální frekvence, na které je naše jádro schopné pracovat.

Tudíž nejprve vypneme v RCC_CR registru bit PLLON, tím je nám umožněno zapisovat hodnoty do registru RCC_PLLSYSCFGR. V tomto registru je třeba nastavit bit PLLSRC na hodnotu 10, což nám nastaví jako vstupní hodinový signál bloku PLL interní HSI RC oscilátor. Dále nastavíme bit PLLM například na hodnotu 0, což nám dá vstupní děličku „/1“, tudíž na vstupu PLL máme 16MHz. Dále nastavíme hodnotu násobičky bitem PLLN na minimální hodnotu 8(0b1000), což nám dá násobení kmitočtu 8x, tudíž máme momentálně frekvenci 128MHz. Pro výstupní děličku R je třeba ještě nastavit bit PLLREN do 1 aby byl povolen výstup z PLL skrze děličku R. Dále nastavíme samotný dělitel R bitem PLLR do hodnoty 1 a tím získáme dělitel /2, což nám dá žádanou výstupní frekvenci 64MHz.

Tímto jsme nastavili samotný blok PLL, dále je třeba ho opět zapnout pomocí bitu PLLON v registru RCC_CR, pak je třeba počkat, než systém v tom samém registru nastaví hodnotu bitu PLLRDY do 1, což nám značí, že inicializace PLL proběhla úspěšně.

Jelikož jsme nastavili frekvenci tak vysoko, že by byl moc rychlý přístup k FLASH paměti, tudíž je třeba nastavit ještě odezvu FLASH paměti v registru FLASH_ACR pomocí bitu LATENCY do hodnoty 2(0b01). Jakožto poslední krok je třeba již jen přepnout zdroj vnitřních hodin systém na PLL namísto HSI přímo. To provedeme nastavením bitu SW v registru RCC_CFGR do hodnoty 0b010, a tím by se měly hodiny přepnout na 64MHz.

```
FLASH->ACR |= FLASH_ACR_LATENCY_0;
RCC->PLLCFGR |= RCC_PLLCFGR_PLLSRC_1;
RCC->PLLCFGR &= ~(RCC_PLLCFGR_PLLN_4);
RCC->PLLCFGR |= RCC_PLLCFGR_PLLN_3 ;
RCC->PLLCFGR |= RCC_PLLCFGR_PLLR_0;
RCC->PLLCFGR |= RCC_PLLCFGR_PLLREN;
RCC->CR |= RCC_CR_PLLON;
RCC->CFGR |= RCC_CFGR_SW_1;
```

4.5 Praktické využití ADC

V této sekci probereme postupy, jak zinicializovat a používat ADC v nejběžnějších režimech. Ukážeme si, jak používat ADC, které je spouštěno jednak softwarově, nebo pomocí externího spouštění a jak z ADC vyčítat data, buďto přímou cestou, softwarově, nebo za pomocí DMA, kdy se o přenos dat nemusí uživatel starat. Podrobný popis nastavení celé periférie lze nalézt v sekci 3.6.3.

4.5.1 Postup pro inicializaci ADC se softwarovým spouštěním konverze

- Zapnutí hodinového signálu k periferii

```
RCC->APBENR2 |= RCC_APBENR2_ADCEN;
```

- Provedení kalibrace bloku ADC

```
ADC1->CR |= ADC_CR_ADVREGEN;
delay(10);
ADC1->CR |= ADC_CR_ADCAL;
while(ADC1->CR & ADC_CR_ADCAL)
{
}
```

- Zapnutí příslušného kanálu ADC pomocí ADC_CHSEL registru (kanál č. 1)

```
ADC1->CHSELR |= ADC_CHSELR_CHSEL1;
```

- Nyní nastavíme dobu odběru vzorku nastavením bitů v ADC_SMPR registru (viz 3.6.3) na nejvyšší možnou hodnotu, 160,5 cyklu

```
ADC1->SMPR = ADC_SMPR_SMP1_2 | ADC_SMPR_SMP1_1 | ADC_SMPR_SMP1_0;
```

- Tímto je celá periférie nastavena, jak a má a je potřeba ji pouze spustit. To provedeme pomocí bitu ADEN v ADC_CR registru. Po jeho zapnutí je potřeba počkat, než se ADC správně stpustí, což signalizuje bit ADRDY v ADC_ISR registru.

```
ADC1->CR |= ADC_CR_ADEN;
while(ADC1->ISR & ADC_ISR_ADRDY)
{
}
```

- Nyní již jen pomocí bitu ADSTART v ADC_CR registru zapneme konverzi námi zvolených kanálu

```
ADC1->CR |= ADC_CR_ADSTART;
```

- Jakmile skončí konverze ADC, je nastaven bit EOC v ISR registru do 1 a můžeme z DR registru vyčíst hodnotu posledního převedeného kanálu

```
myAdcData = ADC1->DR;
```

Nyní je třeba takto změřená data nějak upravit na reálné hodnoty napětí, tudíž je třeba vzít v úvahu velikost napájecího napětí mikrokontroleru (v našem případě 3,3V) a rozlišení ADC (12bitů). Jestliže jsme si data z ADC uložili do proměnné „myAdcData“, tak výsledné měřené napětí Um vypočteme dle vzorce:

$$U_m = \frac{myAdcData}{4096} \times 3.3 \tag{4.1}$$

Nyní jsme zprovoznili ADC v režimu kdy, každou novou konverzi spouštíme pomocí softwaru. V následujících ukázkách nastavíme ADC tak, aby se nám spouštělo pomocí časovače a data se nám pomocí DMA automaticky přesouvala do námi zvolené paměti a nemuseli jsme tím zatěžovat výkon samotného jádra procesoru.

4.5.2 Postup pro inicializaci ADC se softwarovým spouštěním konverze za použití DMA

V tomto příkladu budeme vycházet z nastavení ADC jako v minulém příkladu. Pouze zapneme navíc blok DMA a spustíme ADC v režimu kompatibilním s DMA. Vše lze provést v několika krocích. To provedeme nastavením dvojice bitů DMAEN a DMACFG v ADC_CFGR1 registru (viz řádky č. 1-2). Následně je třeba zapnout hodinový signál pro periférii DMA viz řádek č.3. Nyní bude následovat konfigurace samotného bloku DMA (pozor, jednotlivé kanály v .h souboru jsou pojmenovány vždy DMA1_Channelx, kde x značí číslo kanálu a každý z nich je unikátní struktura, tudíž každý má své unikátní registry, stejně jako například u GPIO portů). Nejprve jsou třeba nastavit adresy periférie a paměti mezi kterými se budou data přenášet, to provedeme nastavením registru CPAR a CMAR pro paměť, respektive periférii (viz řádky 4-5). Dále je třeba nastavit CNDTR registr, v našem případě do hodnoty 1, viz řádek č. 6, neboť měříme jen jeden kanál ADC. Dále je třeba nastavit čtveřici bitů, a to MINC, MSIZE, PSIZE, CIRC. Bitem MINC (viz řádek č. 7) nastavíme automatické inkrementování paměti po úspěšném převodu. Velikosti proměnné MSIZE i PSIZE (viz řádky č. 8 a 9) nastavíme na 16 bitů. A následně bitem CIRC (viz řádek č. 10) spustíme DMA v „circular“ režimu, což nám bude spouště nový převod automaticky, bez nutnosti softwarového spouštění. Nyní máme celý blok DMA správně nakonfigurován a můžeme jej zapnout. Ještě nám však chybí nakonfigurovat blok DMAMUX. V tom je třeba nastavit ADC, jakožto vstup pro spouštění nového přenosu. Tudíž do DMAREQ_ID zapíšeme hodnotu 5 (0b101), viz řádky č. 12 a 13. Řádek 12 nám zapíše 1 na první pozici a řádek 13 na 3. pozici, čímž získáme výsledné číslo 5 (0b101).

```
1  ADC1->CFGR1 |= ADC_CFGR1_DMAEN;
2  ADC1->CFGR1 |= ADC_CFGR1_DMACFG;
3  RCC->AHBENR |= RCC_AHBENR_DMA1EN;
4  DMA1_Channel1->CPAR = (uint32_t)(&ADC1->DR);
5  DMA1_Channel1->CMAR = (uint32_t)(&myAdcData);
6  DMA1_Channel1->CNDTR = 1;
7  DMA1_Channel1->CCR |= DMA_CCR_MINC;
8  DMA1_Channel1->CCR |= DMA_CCR_MSIZE_0;
9  DMA1_Channel1->CCR |= DMA_CCR_PSIZE_0;
10 DMA1_Channel1->CCR |= DMA_CCR_CIRC;
11 DMA1_Channel1->CCR |= DMA_CCR_EN;
12 DMAMUX1_Channel0->CCR |= DMAMUX_CxCR_DMAREQ_ID_0;
13 DMAMUX1_Channel0->CCR |= DMAMUX_CxCR_DMAREQ_ID_2;
```

Pokud jsme vše správně nastavili, už stačí pouze zapnout novou konverzi ADC a jakmile bude dokončena, hodnota ADC_DR registru se nám automaticky uloží do naší proměnné (v našem případě „myAdcData“). V následujícím příkladu si ukážeme, jak nastavit externí spouštění ADC pomocí časovače.

4.5.3 Spouštění ADC pomocí externího spouštěcího signálu

Pro určitá měření, kdy uživatel potřebuje spustit odebírání vzorku pro ADC v konkrétní okamžik, se může hodit režim externího spouštění konverze ADC. Tento režim se může hodit například při tvorbě osciloskopu, k měření proudu v jeho maximu, pro měření s přesnými časovými rozestupy. V tomto příkladu si ukážeme, jak nastavit spouštění konverze s přesnými časovými rozestupy, pomocí časovače TIM3, ale postup je obdobný, pouze s jinými hodnotami, i u jiných druhů externího spouštění.

Pro správné nastavení budeme postupovat jako v ukázce nastavení ADC pro softwarové spouštění, ať už s využitím DMA, nebo bez něj, postup bude stejný (viz 4.5.1 nebo 3.8.1). Budeme tedy uvažovat, že máme zapnuté veškeré potřebné hodinové signály, k periférii i GPIO portům, správně nastavený kanál/skupinu kanálů pro konverzi, dobu odběru vzorku a zapnutou samotnou periférii. Pozor, nesmíme mít aktuálně bežící konverzi ADC, tudíž bit ADSTART v ADC_CR registru musí být nastaven v 0. Dále je potřeba v registru ADC_CFGR1 (viz 3.6.3) nastavit na pozici bitů EXTEN hodnotu 0b01. Tímto uživatel spustí možnost externího spouštění ADC při náběžné hraně spouštěcího signálu. Možností 0b10 by umožnil spouštění konverze na sestupné hraně a možností 0b11 na náběžné i sestupné hraně tohoto signálu. Dále je třeba nastavit vstup spouštěcího signálu. Ten nastavíme ve stejném registru pomocí trojice bitů EXTSEL. Zde zapíšeme bity odpovídající námi zvolenému vstupu spouštěcího signálu, viz 4.4 (v našem případě se jedná o hodnotu 0b010 -TIM2_TRGO).

```
1  ADC1->CFGR1 |= ADC_CFGR1_EXTEN_0;
2  ADC1->CFGR1 |= ADC_CFGR1_EXTSEL_1;
```

Name	Source	EXTSEL[2:0]
TRG0	TIM1_TRGO2	000
TRG1	TIM1_CC4	001
TRG2	TIM2_TRGO	010
TRG3	TIM3_TRGO	011
TRG4	TIM15_TRGO	100
TRG5	TIM6_TRGO	101
TRG6	Reserved	110
TRG7	EXTI line 11	111

Obr. 4.4: Tabulka vstupů spouštěcího signálu[1]

Nyní je potřeba nastavit správně časovač. Opět je potřeba zapnout hodinový signál přivedený k periférii (viz 3.1.3). Nyní je potřeba nastavit, aby se generoval spouštěcí signál pro ADC při každé „Update“ události (eventu). To lze nastavit pomocí trojice bitů MMS v TIM3_CR2 registru. V našem případě nás zajímá hodnota 0b010, která nám zvolí žádaný „Update event“. Dále musí uživatel nastavit vstupní předděličku v registru TIM3_PSC a hodnotu Auto-reload registru (registr TIM3_ARR) na takovou hodnotu, aby se ARR registr naplnil za dobu, kdy chce uživatel spouště znovu konverzi ADC (viz 3.9.5). Pokud bychom chtěli odebírat vzorky s frekvenci např 100Hz, lze například registr (za předpokladu frekvence hodinového signálu 64MHz) PSC nastavit na hodnotu 0xF9FF (63999) a hodnotu ARR registru na 0xA (10). Tímto zajistíme generování „Update eventu“ (přetečení ARR

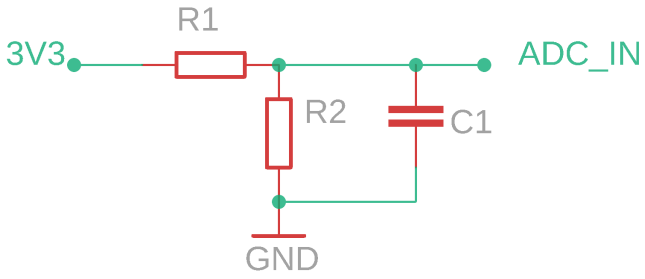
registru) každých 0,01s a tím pádem frekvenci odběru vzorků 100Hz. Nyní už zbývá pouze zapnout samotný časovač pomocí bitu CEN v TIM3_CR1 registru.

```
RCC->APBENR1 |= RCC_APBENR1_TIM3EN;
TIM3->CR1 |= TIM_CR2_MMS_1;
TIM3->PSC = 0xF9FF;
TIM3->ARR = 0xA;
TIM3->CR1 |= TIM_CR1_CEN;
```

V tuto chvíli už pouze stačí zapnout v periférii ADC v registru ADC_CR bit ADSTART a již se nám začne spouštět nová konverze ADC při každé náběžné hraně spouštěcího signálu od „Update eventu“ časovače TIM3.

4.5.4 Využití mikrokontroléru pro měření odporu

V tomto příkladu si ukážeme, jak lze využít periférie ADC k měření odporů a následně budeme změřenou hodnotu odporu odesílat pomocí bloku USART do PC, kde bude zobrazena. Nejprve je nutné stanovit metodu měření odporu. Pro měření odporu budeme využívat principu děliče napětí (viz obr. 4.5), kde odpor R1 bude mít uživatelem známou hodnotu a odpor R2 bude mít hodnotu neznámu. Pomocí ADC pak budeme měřit napětí na odporu R2, dle kterého bude dopočtena velikost tohoto odporu.



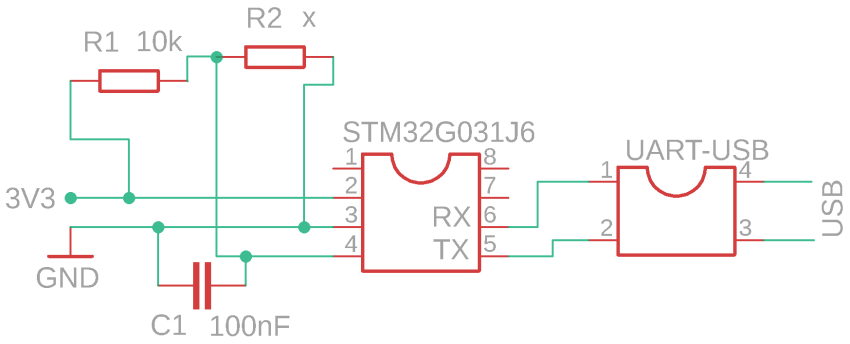
Obr. 4.5: Schéma zapojení napěťového děliče

Jako referenční odpor R1 budeme využívat odpor 10kΩ. Kvůli velkému vnitřnímu odporu zdroje napětí, který měříme, byl do obvodu doplněn blokovací kondenzátor C1, který zajistí, že vzorkovač stihne za dobu svého sepnutí zcela nabít a nebude tím vznikat nežádoucí chyba měření způsobená velkým vnitřním odporem zdroje měřeného napětí.

Hledanou hodnotu odporu R2 získáme upravením známého vzorce pro dělič (viz rovnice 4.2). V této rovnici máme jedinou neznámou, a to námi měřený odpor, R2. Rovnici tedy upravíme na tvar viz 4.3 a tím získáme žádanou hodnotu odporu R2 v závislosti na velikosti odporu R1, vstupního napětí a úbytku napětí naměřeném na odporu R2.

$$U_{R2} = U_{IN} \times \frac{R2}{R1 + R2} \tag{4.2}$$

$$R2 = R1 \times \frac{1}{\frac{U_{IN}}{U_{R2}} - 1} \tag{4.3}$$



Obr. 4.6: Celkové schéma zapojení obvodu pro měření odporu

Nyní se podíváme na softwarovou část problému. Pro celou aplikaci budeme využívat 3 periférie. Periférii ADC, pro měření napětí, časovač TIM3 pro časovou referenci a periférii USART pro komunikaci. Budeme předpokládat, že hodinový signál je nastaven na hodnotu 64MHz (viz 4.4).

Nejprve nastavíme periférii USART_1 pro komunikaci. Vzhledem k tomu, že na procesoru STM32G031J6 nejsou ve výchozím nastavení vyvedeny piny PA9 a PA10, které lze využít pro funkce USART_1_Tx a USART_1_Rx piny, je potřeba je nejprve přemapovat pomocí bloku SYSCFG (viz 3.4). Zapneme tudíž hodinový signál pro blok SYSCFG a následně v SYSCFG_CFGR1 registru nastavíme bit PA11_RMP a PA12_RMP na log.1 (viz řádky č. 1, 2 a 3). Sice by bylo možné využít pin PB6, který na pouzdře vyveden je, ale pokud bychom chtěli aplikaci rozšiřovat např. o zpracování dat z enkodéru, tak by nešel využít, neboť by se překrýval s piny, které bychom potřebovali pro zpracování dat z enkodéru. Dále je potřeba zapnout hodinový signál k GPIO portu A a samotné periférii USART_1 (viz řádky č. 4 a 5) a následně nastavit režim alternativních funkcí v registru GPIOx_MODER a v GPIOx_AFR registru nastavit správnou volbu alternativní funkce (viz řádky č. 6-9) pro jednotlivé piny.

Nyní zbývá pouze nastavit samotný blok USART_1. Nastavím správnou rychlost přenosu (Baud-rate) v USART_BRR registru (řádek č. 10) na hodnotu 9600 b/s a povolíme funkce Tx, Rx a v poslední řadě i funkce periférie samotné v USART_CR1 registru pomocí bitů RE, TE a UE (viz řádky č. 11, 12, 13).

```
1  RCC->APBENR2 |= RCC_APBENR2_SYSCFGEN;
2  SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA12_RMP;
3  SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA11_RMP;
4  RCC ->IOPENR |= RCC_IOPENR_GPIOAEN;
5  RCC->APBENR2 |= RCC_APBENR2_USART1EN;
6  GPIOA->MODER &= ~(GPIO_MODER_MODE9_0);
7  GPIOA->AFR[1] |= GPIO_AFRH_AFSEL9_0;
8  GPIOA->MODER &= ~(GPIO_MODER_MODE10_0);
9  GPIOA->AFR[1] |= GPIO_AFRH_AFSEL10_0;
10 USART1->BRR = 0x1A0C;
11 USART1->CR1 |= USART_CR1_RE;
12 USART1->CR1 |= USART_CR1_TE;
13 USART1->CR1 |= USART_CR1_UE;
```

Tímto jsme správně nastavili periférii USART_1, nyní je potřeba nastavit periférii ADC. Využijeme nastavení, kdy jsme řídili spouštění ADC pomocí externího spouštěcího signálu z časovače TIM3 viz 4.5.3). Pouze zapneme navíc bit EOCIE v registru ADC_IER, který nám povolí přerušení od ADC při dokončení konverze, tudíž budeme po každé dokončené konverzi schopni zpracovávat data a odesílat je do PC pomocí rozhraní USART.

Dále je potřeba povolit v bloku NVIC přerušení od ADC a nastavit jeho prioritu. Pro to máme již v souboru „core_cm0plus.h“ předpřipravenou dvojici funkcí (viz řádky č. 2 a 3). Podrobněji popsáno v sekci 3.2).

```
1 ADC1->IER |= ADC_IER_EOCIE;
2 NVIC_EnableIRQ(ADC1_IRQn);
3 NVIC_SetPriority(ADC1_IRQn,0);
```

Nyní už jen zbývá zadefinovat funkci, která se bude vykonávat, když nastane přerušení o dokončení konverze periférie ADC a doplnit kód pro přenášení výsledků do terminálu na PC. Podrobné vysvětlení přerušení a práce s nimi viz 2.5 a 3.2. V této funkci nejprve načteme naměřenou hodnotu napětí z ADC_DR registru do naší proměnné (myData). Následně ji přepočteme na hodnotu odporu (viz rovnice 4.3) a uložíme do nové proměnné (myResistorValue). Následně provedeme filtraci hodnot, neboť velikost změřeného napětí může trochu kolísat. Využijeme pro to filtr klouzavých průměrů a výslednou hodnotu přičteme do nové proměnné (myresistorValue_filtered). Obě proměnné musí být globálního charakteru, neboť při filtraci je potřeba si vždy minulou změřenou hodnotu, aby šla filtrace provést, pamatovat.

Nyní máme výsledek (vyfiltrovanou hodnotu a hodnotu aktuální). Je třeba ji tedy odeslat pomocí bloku USART_1 do PC. Hodnotu budeme odesílat společně s doplňujícím textem, aby výsledek byl čitelný. Vytvoříme si lokální proměnnou „myTxData“, do níž vložíme text, který budeme odesílat. Naše pole bajtů naplníme pomocí funkce „sprintf“ obsaženou v knihovně „stdio.h“. Tímto jsme zároveň převedli proměnnou typu „uint32_t“ na skupinu bajtů a je možné již celé pole, bajt po bajtu, odeslat. Funkce „sprintf“ nám zároveň vrací počet bajtů, které do pole zapsala, tudíž si je uložíme do proměnné (size) a následně zpracujeme celé pole od 0, až po proměnnou size a odešleme jednotlivé bajty zapsáním jejich hodnoty do USART_TDR registru. Po každém odeslaném bajtu je potřeba vyčkat, než se data kompletně odešlou (signalizováno bitem TC v USART_ISR registru).

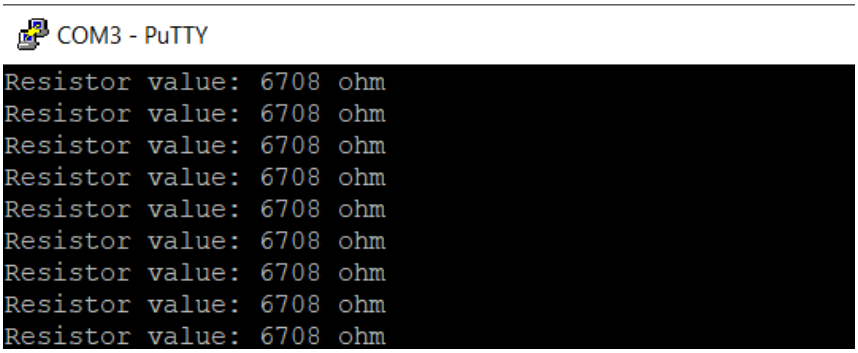
```
uint32_t myResistorValue = 0;
uint32_t myResistorValue_filtered = 0;
uint16_t capacity = 200;

void ADC1_IRQHandler(void)
{
 uint16_t myData = ADC1->DR;
 myResistorValue = (float)R_DEFAULT*(float)(1.0/((4096.0/(float)myData)-1));

 int32_t tmp_0, tmp_1;
 tmp_0 = ((int32_t)myResistorValue - (int32_t)myResistorValue_filtered);
 tmp_0 = tmp_0 << 5;
 tmp_1 = tmp_0 / (int32_t)capacity;
 tmp_1 = tmp_1 >> 5;
 myResistorValue_filtered += tmp_1;

 char myTxData[30];
 uint8_t size = sprintf(myTxData, "Resistor value: %d ohm\r\n",
     ↪ myResistorValue_filtered);
 for(uint8_t i = 0;i<size;i++)
 {
  USART1->TDR = myTxData[i];
  while ((USART1->ISR & USART_ISR_TC) != USART_ISR_TC)
  {
  }
 }
}
```

Pokud jsme vše nastavili správně, měli bychom po spuštění terminálu na PC vidět výslednou hodnotu připojeného odporu R2 (viz obr. 4.7).



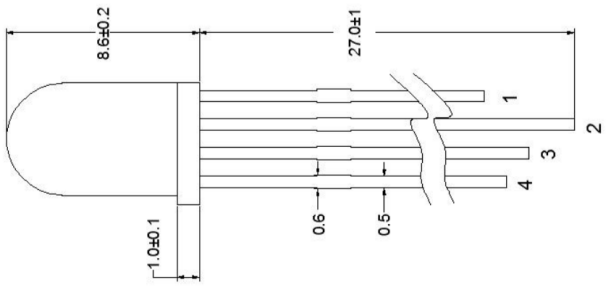
Obr. 4.7: Hodnota odporu na sériovém terminálu

4.6 Ovládání RGB diody s digitálním vstupem

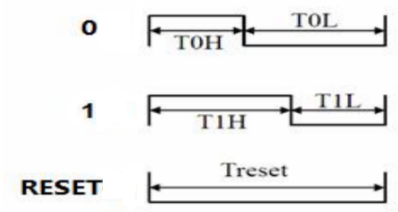
V této sekci si popíšeme návrh komunikačního protokolu na ovládání RGB diody s digitálním vstupem. Konkrétně se jedná o RGB diodu WS2812D-F5 od firmy Worldsemi. Tato RGB LED dioda se neovládá, jako klasická RGB dioda, PWM signálem přivedeným ke každé barevné složce světla zvlášť, ale pouze jedním digitálním výstupem z mikrokontroléru. Dioda ve svém pouzdře integruje celý řídící obvod. Ten obsahuje tři generátory PWM signálu, budiče LED a komunikační řadič. Tento řídící obvod se následně stará, na základě vstupu, o buzení jednotlivých LED diod v pouzdře.

Celá dioda je napájena zdroje 5V, má jeden vstupní (pin DIN) a jeden výstupní pin (pin DOUT) (viz obr. 4.8) pro datovou komunikaci. Na vstupní pin datové komunikace musí být přivedena sekvence časově přesně definovaných pulzů, kde každý z nich má specifikovanou délku (viz obrázky 4.9 a 4.10) a definuje, zda byla na vstup odeslána log.1 nebo log.0. Každou ze tří barevných složek lze definovat pomocí 8 odeslaných bitů které definují intenzitu jednotlivých barevných složek diody. Tudíž pro nastavení diody je potřeba odeslat celkem 24 bitů ve kterých je zakódovaná intenzita jednotlivých složek.

Číslo pinu	Funkce
1	DOUT
2	VCC
3	GND
4	DIN



Obr. 4.8: Popis pinů RGB diody[10]



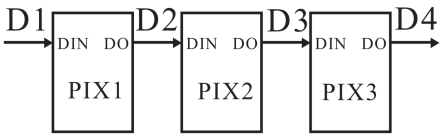
Obr. 4.9: Komunikační protokol RGB diody, definice délky logických stavů[10]

T0H	0 code, high voltage time	220ns~380ns
T1H	1 code, high voltage time	580ns~1µs
T0L	0 code, low voltage time	580ns~1µs
T1L	1 code, low voltage time	580ns~1µs
RES	Frame unit, low voltage time	>280µs

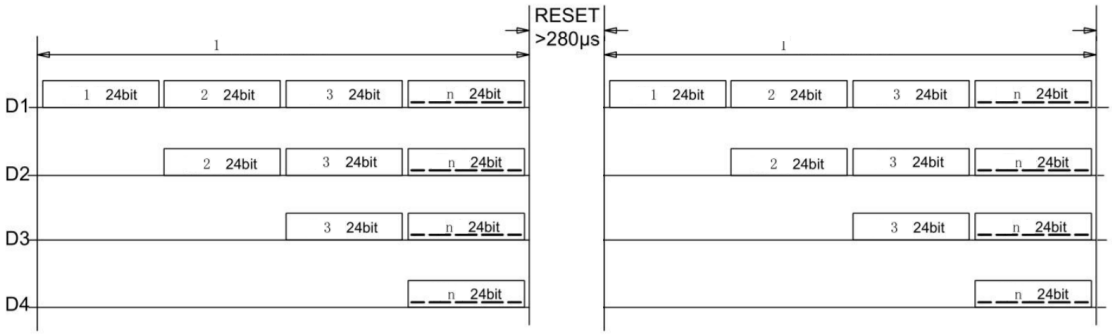
Obr. 4.10: Popis průchodu datových bloků jednotlivými diodami[10]

Řadič v každém pouzdře zároveň umožňuje kaskádní řazení RGB LED diod za sebou, kde vždy dioda přijme prvních 24 bitů a všechny další přijaté bity přepošle beze změny na svůj výstupní pin, který může být připojen k další diodě (viz obr. 4.11). Takto jsou data konstantě přeposílána skrze všechny diody, dokud nenastane příznak přerušení komunikace („RESET“). Ten je definován jako prodleva mezi jednotlivými sekvencemi bitů delší než 280μs, tedy stav 0V na vstupu diody po dobu více než 280μs. Pokud dojde k „RESETU“, tak následující sekvence 24 bitů opět bude brána jako nastavení dané diody a až následující bity budou odesílány dál.

Obrázek 4.11 znázorňuje jednotlivé diody a signály D1-D4 do nich vstupující, přičemž signál D1 je sekvence bitů vyslaná mikrokontrolérem. Následující signály D2, D3 a D4 jsou již data přeposlána do další diody. Ta jsou vždy „ochuzena“ o část, kterou do sebe uložila dioda předchozí. Princip toku dat lze vidět na obrázku 4.12, kde vidíme odeslání čtveřice 24bitových bloků, které jsou postupně přeposlány mezi 4 diody a následně, po odmlce 280μs je odeslána další čtveřice dat



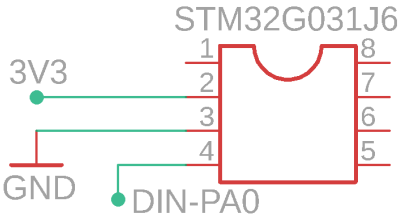
Obr. 4.11: Schéma kaskádního zapojení RGB LED diod[10]



Obr. 4.12: Princip kaskádního přeposílání dat[10]

Nejprve je potřeba odeslat 8 bitů pro intenzitu červené, následně 8 bitů pro zelenou a jako poslední 8 bitů pro modrou. Po správném odeslání teto sekvence bitů se načtené hodnoty v diodě převedou na intenzity jednotlivých barev a výsledná složená hodnota jednotlivých intenzit bude vidět v podobě světelného výstupu diody. Pin DOUT pro výstup dat nebudeme využívat, slouží jako „můstek“ pro přenos dat do více diod zapojených v sérii (sériové zapojení myšleno z hlediska datové komunikace).

Nyní tedy zapojíme diodu. Pin VCC připojím ke zdroji 5V a pin GND k zemnícímu pinu. Následně pin DIN zapojíme přímo k našemu mikrokontroléru k některému z GPIO pinů (já jsem zvolil pin PA0, viz schéma zapojení 4.13). Pro řízení nebudeme využívat časovač, ale použijeme programově řízený GPIO pin a trvání logických stavů si nadefinujeme pomocí „delay“ funkce, která vždy nějakou dobu po sepnutí pinu do log.1 nějakou dobu počká a opět ho vypne. Ano, tento problém by šel velmi elegantně vyřešit pomocí časovačů, ale chtěl jsem tímto příkladem ukázat, že to lze realizovat i za pomoci GPIO pinů a dost vysoké frekvence jádra. Navíc celková úloha se tak stane z hlediska periférií velmi jednoduchou, neboť budeme pouze rychle zapínat a vypínat výstupní pin.

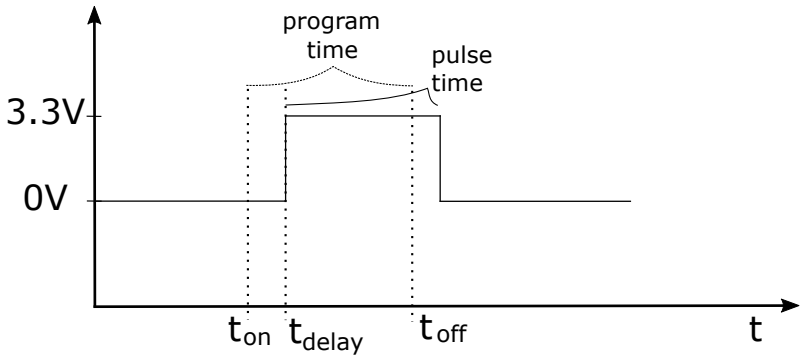


Obr. 4.13: Schéma zapojení pro ovládání RGB diody

Nyní se podíváme na implementaci softwaru. Celkový program je velmi jednoduchý, nejprve nastavíme nejvyšší možnou frekvenci jádra, tudíž 64MHz. Postup na nastavení PLL bloku lze nalézt v sekci 4.4. Tímto máme nastavenou frekvenci hodinového signálu na 64MHz a je potřeba zapnout periférii USART. Pro komunikaci pomocí bloku USART_1 budeme používat piny PA9 a PA10. Ty však nejsou v základním rozvržení na fyzických pinech dostupné, proto je potřeba je přemapovat (řádky č. 1 - 3). Následně zapneme port

GPIOA a periférii USART (řádky č. 4 - 5). Dále nastavíme pin PA0 do režimu digitálního výstupu (řádky č. 7 - 11) a piny PA9 a PA10 nastavíme do režimu alternativní funkce (řádky č. 14 - 17) pro periférii USART. Jako poslední nastavíme samotnou periférii a povolíme, aby generovala přerušení, včetně konfigurace bloku NVIC (řádky č.19 - 26).

```
1  RCC->APBENR2 |= RCC_APBENR2_SYSCFGEN;
2  SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA11_RMP;
3  SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA12_RMP;
4  RCC ->IOPENR |= RCC_IOPENR_GPIOAEN;//enable gpioa port
5  RCC->APBENR2 |= RCC_APBENR2_USART1EN; //enable usart1
6
7  GPIOA->MODER |= GPIO_MODER_MODE0_0;
8  GPIOA->MODER &= ~(GPIO_MODER_MODE0_1);
9  GPIOA->OTYPER &= ~(GPIO_OTYPER_OT0);
10 GPIOA->OSPEEDR |= GPIO_OSPEEDR_OSPEED0;
11 GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPD0);
12
13 //USART1 setup
14 GPIOA->MODER &= ~(GPIO_MODER_MODE9_0); // PA9,PA10
15 GPIOA->AFR[1] |= GPIO_AFRH_AFSEL9_0;
16 GPIOA->MODER &= ~(GPIO_MODER_MODE10_0);
17 GPIOA->AFR[1] |= GPIO_AFRH_AFSEL10_0;
18
19 USART1->BRR = 0x1A0C;
20 USART1->CR1 |= USART_CR1_RE;
21 USART1->CR1 |= USART_CR1_TE;
22 USART1->CR1 |= USART_CR1_UE;
23 USART1->CR1 |= USART_CR1_RXNEIE_RXFNEIE;
24
25 NVIC_EnableIRQ(USART1_IRQn);
26 NVIC_SetPriority(USART1_IRQn,1);
```



Obr. 4.14: Schéma časování jednotlivých bitů

Nyní, když máme veškeré funkcionality spuštěné, napíšeme jednoduchý algoritmus odeslání celkových 24 bitů dat pro intenzity svícení jednotlivých barevných složek diody. Funkce „delay“ nám zajišťuje formu vyčkání určitého časového úseku, abychom mohli „namodelovat“ žádanou délku pulzu. Funkce „sendLedHigh“ nám odešle bit s log.1 a funkce „sendLedLow“ nám odešle bit s log.0. Hodnoty funce delay byly experimentálně změřeny pomocí osciloskopu tak, aby výsledný pulz odpovídal specifikacím diody (viz obr. 4.10). Zároveň dobu celkového času sepnutí ovlivní doba zápisu do GPIOx_BRR registru, kdy vypínáme výstup, neboť několik cyklů jádra trvá změna údajů v registru a to se nám promítne na celkovém čase délky sepnutí. Na obrázku 4.14 můžeme vidět jednoduché schéma, kde jsou znázorněny jednotlivé časy, kdy program začal vykonávat danou instrukci a oproti tomu vidíme, kdy byla instrukce dokončena změnou signálu.

- t_{on} - Začátek zápisu do GPIOx_BSSR registru pro změnu signálu z „low“ na „high“, tedy zapnutí
- t_{delay} - začátek čítání „delay“ funkce pro implementaci zpoždění
- t_{off} - začátek zápisu do GPIOx_BRR registru pro změnu signálu z „high“ na „low, tedy vypnutí“

Z tohoto grafu můžeme vidět, že i bez „delay“ funkce bychom nebyly schopni vygenerovat velmi krátký pulz, neboť nějakou dobu trvá samotný zápis do registru výstupní brány GPIO.

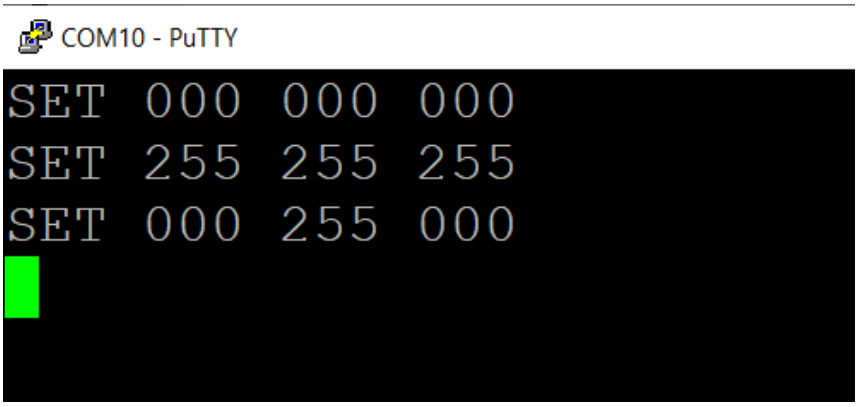
```
void delay(int32_t time)
{
        volatile int j = 0;
        for(int32_t i = 0; i<= time;i++)
        {
                j++;
        }
}
void sendLedHigh()
{
                GPIOA->BSRR |= GPIO_BSRR_BS0;
                delay(2);
                GPIOA->BRR |= GPIO_BRR_BR0;
}
void sendLedLow()
{
                GPIOA->BSRR |= GPIO_BSRR_BS0;
                GPIOA->BRR |= GPIO_BRR_BR0;
                delay(1);
}
```

Následně pouze sestavíme funkci, která dostane na vstupu 3 složky intenzity jednotlivých barev (8-bitové číslo, hodnota 0-255) a podle jejich hodnot odešle daných 8 bitů do RGB diody, která dle vstupu nastaví výslednou barvu.

```
void setColor(uint8_t redValue, uint8_t greenValue, uint8_t blueValue)
{
        for (uint8_t i = 0; i < 8; i++)
        {
                if(redValue & 1<<i)
                {
                sendLedHigh();
                }
                else
                {
                sendLedLow();
                }
        }//Red
        for (uint8_t i = 0; i < 8; i++)
        {
                if(greenValue & 1<<i)
                {
                sendLedHigh();
                }
                else
                {
                sendLedLow();
                }
        }//Green
        for (uint8_t i = 0; i < 8; i++)
        {
```

```
27                      if(blueValue & 1<<i)
28                      {
29                      sendLedHigh();
30                      }
31                      else
32                      {
33                      sendLedLow();
34                      }
35              }//blue
36  }
```

Nyní ještě využijeme periférii USART, abychom byli schopni dle poslaného příkazu, například z PC, nastavovat intenzity jednotlivých barev. Vytvoříme si funkci, která bude zpracovávat přijatá data z periférie USART a pokud dostane na vstupu string ve tvaru "SET xxx yyy zzz", kde x, y a z značí čísla v rozmezí 0-255 určující intenzitu barevných složek (viz obr. 4.15), tak zavolá funkci „setCollor“ a tím nastaví danou barvu pro RGB diodu. Princip implementace USART protokolu lze nalézt v přiloženém souboru, nebo je popsán v příkladu 4.7.



Obr. 4.15: Výstup komunikace na sériovém terminálu

Celá tato úloha může sloužit jakožto návod pro využití obvodu jako řadič pro ovládání skupiny RGB LED diod. Ty by šlo využít například společně s příkladem 4.5.4, kdy by výsledná barva diody mohla odpovídat velikosti změřeného odporu. Dále pak při zapojení několika diod v kaskádě za sebou by bylo možné zhotovit zajímavý ukazatel baterie, kde by mohly jednotlivé diody v řadě za sebou podle kapacity baterie měnit intenzitu, či barvu svitu. V neposlední řadě by šlo zhotovit zapojení diod do matice, kde bychom procesoru posílali např. ASCII znaky a ten by je mohl vykreslovat pomocí skupiny LED diod, jako na displayi.

4.7 Nastavitelný generátor signálu PWM s proměnnou střídou a frekvencí

V tomto příkladu si ukážeme využití mikrokontroléru STM32G031J6 v režimu, kdy bude ovládán jakožto „Smart-circuit“ pomocí jiného, nadřazeného mikrokontroléru, kde bude zastupovat funkci obvodu, kde pomocí USART rozhraní budeme nastavovat danou frekvenci a třídu signálu PWM.

Nejprve nastavíme periférii USART_1. Použijeme nastavení popsané v sekci 3.7.3 až na dvě změny, kdy budeme přijatá data zpracovávat v přerušení a použijeme přemapování pinů pro datovou komunikaci. Přemapování pinů provedeme pomocí řádku č. 1 - 3, kde nejprve zapneme blok SYSCFG a následně přemapujeme pin PA11 na pin PA9 a pin PA12

na pin PA10 (viz 3.4). Následně pomocí řádku č. 5 - 9 nastavíme dané Rx a Tx piny (PA9-Rx a PA10-Tx) pro komunikaci do alternativní funkce USART. Nyní na řádcích č.11-16 nastavíme samotnou periférii USART_1, přičemž na řádku č.16 povolíme přerušení od periférii, pokud byla přijata nějaká nová data. Následně pomocí řádků č. 18 - 19 nastavíme blok NVIC pro zpracování přerušení, abychom mohli obsluhovat funkce USART_1 vždy když byla přijata nová data.

```
1  RCC->APBENR2 |= RCC_APBENR2_SYSCFGEN;
2  SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA11_RMP;
3  SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA12_RMP;
4
5  RCC ->IOPENR |= RCC_IOPENR_GPIOAEN;
6  GPIOA->MODER &= ~(GPIO_MODER_MODE9_0);
7  GPIOA->AFR[1] |= GPIO_AFRH_AFSEL9_0;
8  GPIOA->MODER &= ~(GPIO_MODER_MODE10_0);
9  GPIOA->AFR[1] |= GPIO_AFRH_AFSEL10_0;
10
11 RCC->APBENR2 |= RCC_APBENR2_USART1EN;
12 USART1->BRR = 0x1A0C;
13 USART1->CR1 |= USART_CR1_RE;
14 USART1->CR1 |= USART_CR1_TE;
15 USART1->CR1 |= USART_CR1_UE;
16 USART1->CR1 |= USART_CR1_RXNEIE_RXFNEIE;
17
18 NVIC_EnableIRQ(USART1_IRQn);
19 NVIC_SetPriority(USART1_IRQn,1);
```

Nyní provedeme nastavení časovače. Pro výstup PWM využijeme časovač TIM3 (obdobné nastavení a jeho popis naleznete v sekci 3.9.4), přičemž budeme využívat kanál č.2. Nejprve nastavíme výstupní bránu GPIO (viz řádky č.1-4) do alternativní funkce a nastavíme rychlost výstupu brány. Následně zapneme hodinový signál k periférii TIM3 (viz řádek č. 6) a následně nakonfigurujeme základní nastavení časovače na výchozí frekvenci $f_{update} = 1kHz$ (později ji budeme měnit pomocí USART příkazů) a zapneme výstup časovače (viz řádky č. 8 - 13). Samotný časovač pomocí bitu CEN však zapínat nebudeme, aby byl ve výchozím nastavení bez zadání příkazu časovač vypnutý. Tímto bychom měli časovač nastavený a připravený ke spuštění tak,s aby generoval PWM signál po zapsání do CCR2 registru a spuštění samotného časovače bitem CEN.

```
1  GPIOA->MODER &= ~(GPIO_MODER_MODE7_0);
2  GPIOA->OTYPER &= ~(GPIO_OTYPER_OT7);
3  GPIOA->OSPEEDR |= GPIO_OSPEEDR_OSPEED7;
4  GPIOA->AFR[0] |= GPIO_AFRL_AFSEL7_0;
5
6  RCC->APBENR1 |= RCC_APBENR1_TIM3EN;
7
8  TIM3->PSC = 639;
9  TIM3->ARR = 99;
10
11
12 TIM3->CCMR1 |= TIM_CCMR1_OC2M_2 | TIM_CCMR1_OC2M_1;
13 TIM3->CCER |= TIM_CCER_CC2E;
```

Když jsme všechny periférie správně nastavili, je potřeba napsat software, který bude ovládat pomocí USART příkazů výstup a nastavení časovače TIM3. Pro celkové ovládání si vytvoříme „protokol“, pomocí nějž budeme schopni výstup pwm:

- Zapnout - příkaz „START“
- Vypnout - příkaz „STOP“

- Nastavit střídu - příkaz „DUTY xxxxx“, kde x značí zadanou střídu v %
- Nastavit frekvenci - příkaz „FREQ xxxxx“, kde x značí zadanou frekvenci v Hz

Pro oddělovací znak mezi jednotlivými příkazy zvolíme znak „
n“, který značí tzv-„line-feed“ a provede odřádkování na nový řádek.

Nyní se podíváme na samotnou softwarovou implementaci protokolu. Pro celé ovládání si vytvoříme řídící strukturu, ve které se budou ukládat jednak přijatá data a zároveň nastavení pro časovač se zadanou frekvencí a střídou. Zde v hlavní struktuře typu „pwmGenerator_HandleTypedef“ budeme ukládat střídu do proměnné „pwmDutyCycle“ a frekvenci do proměnné „pwmFrequency“. V proměnné myUart (struktura) se bude nacházet pole bajtů, kam se bude ukládat přijatý „string“ dat pomocí USART a zároveň proměnná Rx_indx, kterou budeme potřebovat pro zjišťování, kolik bajtů dat bylo celkem načteno. Následně si zadefinujeme ještě funkci, která nám budem zpracovávat přijatá data z USART a podle nich nastavovat časovač na žádanou frekvenci a střídu (Rx_handler). Ta bude mít jakožto vstup ukazatel na naši strukturu typu „pwmGenerator_HandleTypedef“. Tuto funkci pak vložíme do funkce void USART1_IRQHandler(void), která se bude volat po každém novém přijatém bajtu dat (podrobné vysvětlení nastavení přerušení od bloku NVIC viz 3.2).

```
#define RX_BUFFER_LENGTH 20
typedef struct
{
  char Rx_indx;
  char Rx_Buffer[RX_BUFFER_LENGTH];
}uartControl_HandleTypeDef;

typedef struct
{
        uint32_t pwmFrequency;
        uint32_t pwmDutyCycle;
        uartControl_HandleTypeDef myUart;
}pwmGenerator_HandleTypedef;

void Rx_handler(pwmGenerator_HandleTypedef *pwm);
```

Samotná implementace funkce Rx_handler lze nalézt v přiložených souborech (projekt STM32G031J6_pwm_generator) a celá spočívá v načítnání nových bajtů do pole Rx_Buffer a při načtení znaku „\n“ (v ASCII kódování hodnota 10) je načítání přerušeno a celé pole bajtů (charů) je analyzováno a na základě vstupu je čítač buďto vypnu, zapnut nebo změněna frekvence, či střída. Zapnutí, vypnutí a nastavování frekvence a třídy PWM signálu je prováděno pomocí následujících částí kódu:

- Zapnutí:

```
TIM3->CR1 |= TIM_CR1_CEN;
TIM3->EGR |= TIM_EGR_UG;
TIM3->BDTR |= TIM_CCER_CC2E;
```

- Vypnutí:

```
TIM3->CR1 &= ~(TIM_CR1_CEN);
TIM3->BDTR &= ~(TIM_CCER_CC2E);
TIM3->EGR |= TIM_EGR_UG;
```

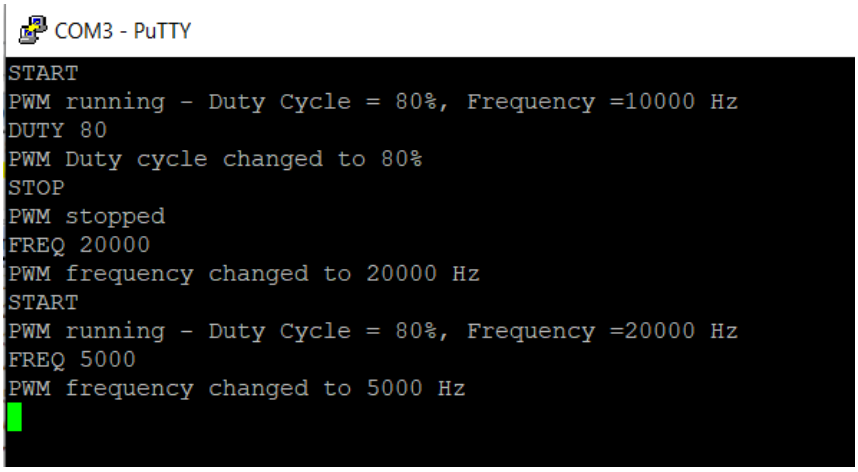
- Nastavení střídy

```
TIM3->CCR2 = getNumberFromString(Rx_Buffer,5);
```

- Nastavení frekvence

```
uint16_t desiredFrequency = getNumberFromString(Rx_Buffer,4);
uint16_t localFreq = (uint16_t)(CORE_FREQUENCY/desiredFrequency);
uint16_t myPsc = 0;
uint16_t myArr = 100;
myPsc = (uint16_t)(((float)localFreq)/((float)myArr));
TIM3->PSC = myPsc-1;
TIM3->ARR = myArr-1;
```

Zapnutí a vypnutí výstupu PWM signálu je provedeno pomocí trojice bitů. Bitu CEN v registru TIM3_CR1, který zapne/vypne čítání čítače. Bitu CC2E v registru TIM3_CCER, který zapne „Capture-compare“ blok a bitu UG v registru TIM3_EGR, který vygeneruje „update-event“, který vynuluje čítač a nastaví registr TIM3_CNT do hodnoty 0. Nastavení střídy funguje pouhým vyčtením čísel z přijatých dat pomocí funkce getNumberFromString() a její výsledek je zapsán přímo do CCR registru, neboť ARR registr je trvale nastaven na hodnotu 99 (implementace funkce getNumberFromString() lze nalézt podrobně v přiloženém projektu). Nastavení frekvence funguje podobně, jako nastavení střídy, tedy vyčtením požadované frekvence z přijatého stringu, ale následně je třeba dle rovnice 3.3 vypočíst hodnotu předdělicky TIM3_PSC na takovou hodnotu, abychom ve výsledku získali požadovanou frekvenci signálu (registr TIM3_ARR si zafixujeme na danou hodnotu, aby se výpočet ulehčil).



Obr. 4.16: Výstup komunikace na sériovém terminálu

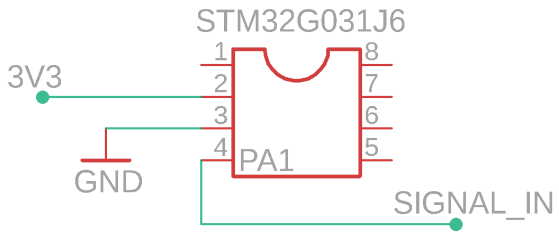
Tímto máme zhotovený „Smart-circuit“, který je schopen pomocí jiného mikrokontroléru prostřednictvím USART příkazů spouštět, vypínat a nastavovat parametry generovaného PWM signálu.

4.8 Aplikace pro měření frekvence a délky impulzů

V této ukázce si ukážeme, jak zpracovat jednoduchý obvod pro měření frekvence a délky pulzu signálu. Pro měření obou parametrů využijeme časovač TIM2 v režimu „Input-capture“, kdy budeme zaznamenávat „time-stampy“ náběžných a sestupných hran signálu a z nich dopočítávat frekvenci signálu a dobu, po jakou je signál v log.1, tedy ve stavu „high“.

Obvod pro měření frekvence a délky impulzů bude zapojen viz schéma 4.17. Pro samotné měření využijeme kanál č .2 časovače TIM2, který se nachází na pinu PA1, cože je

pin č .4 na našem mikrokontroléru. Ve výchozím stavu tento pin sdílí pozici s NRST pinem, tudíž je potřeba funkci resetu vypnout, nebo využít jiný kanál časovače. Na tento pin přivedeme zdroj obdélníkového periodického signálu, který budeme měřit. V případě, že takový signál nemáme pro testovací účely k dispozici, můžeme využít zdroj signálu PWM generovaný naším mikrokontrolérem. Návod, jak nastavit výstup signálu PWM viz sekce 3.9.4, nebo 4.7.



Obr. 4.17: Schéma zapojení pro měření frekvence

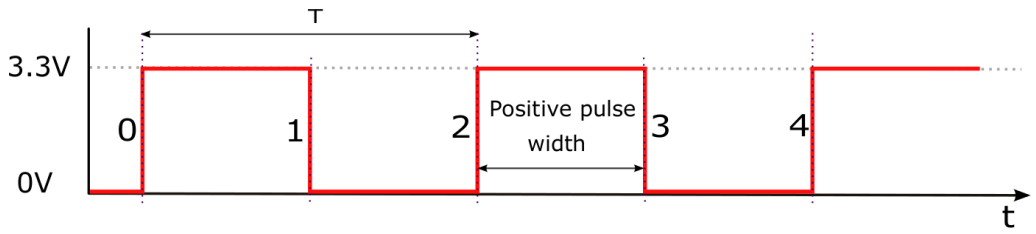
Jelikož budeme časovač využívat v režimu vstupu, je potřeba správně nastavit GPIOA port (bližší popis viz 3.5). Nejprve povolíme zdroj hodinového signálu pro port GPIOA (řádek č .1) a nakonfigurovat vstupní pin PA1 do alternativní funkce (řádky č. 2 a 3). Dále povolíme přerušení pro časovač TIM2 v bloku NVIC (viz řádky č. 5 a 6). Podrobnější vysvětlení bloku NVIC viz 3.2. Když máme správně nastavený blok NVIC, musíme nakonfigurovat samotný časovač (řádky č. 9 - 14) do režimu „Input-capture“ a zapnout přerušení při „Capure-eventu“, kdy budeme změřené „time-stampy“ zpracovávat. Režim „Input-capture“ nakonfigurujeme tak, aby se „time-stamp“ zapisoval do CCR2 registru při příchodu náběžné i sestupné hrany. Podrobné vysvětlení nastavení „Input-capture“ režimu časovače lze nalézt v sekci 3.9.6.

```
1  RCC->IOPENR |= RCC_IOPENR_GPIOAEN;
2  GPIOA->MODER &= ~(GPIO_MODER_MODE1_0);
3  GPIOA->AFR[0] |=  GPIO_AFRL_AFSEL1_1;
4
5  NVIC_EnableIRQ(TIM2_IRQn);
6  NVIC_SetPriority(TIM2_IRQn,0);
7
8  RCC->APBENR1 |= RCC_APBENR1_TIM2EN;
9  TIM2->CCMR1 |= TIM_CCMR1_CC2S_0 | TIM_CCMR1_IC2F_0 | TIM_CCMR1_IC2F_1;
10 TIM2->CCER |= TIM_CCER_CC2E;
11 TIM2->DIER |= TIM_DIER_CC2IE;
12 TIM2->CCER |= TIM_CCER_CC2P;
13 TIM2->CCER |= TIM_CCER_CC2NP;
14 TIM2->CR1 |= TIM_CR1_CEN;
```

Nyní, když máme vše správně nakonfigurované, je potřeba data zpracovat. Zpracovávat je budeme vždy při novém příchodu náběžné/sestupné hrany měřeného signálu, tudíž při vyvolaném přerušení od časovače. Pro celou funkci měření si obdobně, jako v příkladu 4.7 pro řízení, vytvoříme strukturu typu „frequencyMeasure_HandleTypedef“ pro lepší orientaci v proměnných. Proměnné „timeStamp“ a „timeStampLast“ slouží k ukládání posledních dvou hodnot náběžných hran sloužících pro výpočet frekvence. Dále zadefinujeme proměnnou „freq“, do které budeme ukládat výslednou hodnotu změřené frekvence. Následně pro výpočet délky pulzu máme dvě proměnné„timeStampPulseWidth“ a „timeStampPulseWidthLast“. Do proměnné „pulsewidth“ budeme ukládat výsledný počet pulzů načítaný za dobu, kdy je hodinový signál v log.1. Proměnná edgeIndex slouží k identifikaci, kolikátou náběžnou hranu jsme v danou chvíli zaznamenali.

```
typedef struct
{
uint32_t timeStamp;
uint32_t timeStampLast;
uint32_t freq;
uint8_t edgeIndex;
uint32_t timeStampPulseWidthLast;
uint32_t timeStampPulseWidth;
uint32_t pulsewidth;
}frequencyMeasure_HandleTypedef;
```

Vzhledem k tomu, že pro měření využíváme pouze jeden kanál časovače a kvůli zaznamenávaní délky pulzu potřebujeme, aby se nám zapisovaly „time-stampy“ jak náběžných, tak sestupných hran, musíme si vést údaj o tom, kolikátá náběžná hrana byla načtena. Pokud bychom to nedělali, nebyli bychom schopni zjišťovat zda se jedná o náběžnou, či sestupnou hranu signálu a frekvenci bychom nemohli vypočíst. Pokud tak učiníme, budeme schopni zjistit, zda se jedná o náběžnou, či sestupnou hranu signálu a z toho vypočítat žádané veličiny. Z obrázku 4.18 je patrné, že pro měření frekvence budeme využívat vždy buďto pouze náběžné, nebo pouze sestupné hrany.



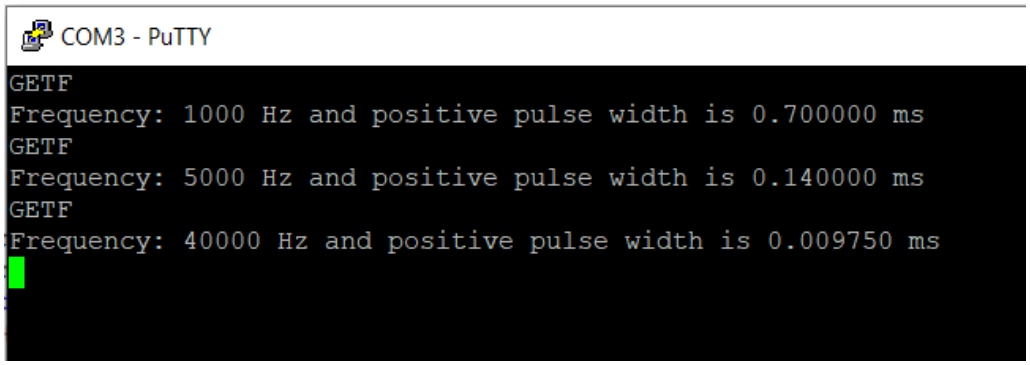
Obr. 4.18: Měřený signál

Proto si do proměnných pro zpracování frekvence budeme ukládat hodnotu CCR2 registru (řádky č. 5 a 6) pouze ve chvíli, kdy proměnná „edgeIndex“ bude např. lichá (sestupné hrany). Naopak do proměnných pro měření frekvence potřebujeme hodnotu zapisovat vždy tak, že při náběžné hraně se zapíše hodnota do proměnné „timeStampPulseWidth“, tedy při sudých násobcích „edgeIndex“ (řádek č. 21) a do proměnné „timeStampPulseWidthLast“ při lichých násobcích (řádek č. 17). Řádky č. 7 - 15 a řádky č. 22 - 32 slouží pro ošetření hodnot, když časovač TIM2 „přeteče“, aby došlo ke korekci chyby.

```
void TIM2_IRQHandler(void)
{
    if(myFreq.edgeIndex == 1)
    {
        myFreq.timeStampLast = myFreq.timeStamp;
        myFreq.timeStamp = TIM2->CCR2;
        if(myFreq.timeStamp<myFreq.timeStampLast)
        {
            myFreq.timeStampLast = UINT32_MAX-myFreq.timeStampLast;
            myFreq.freq = 64000000/(myFreq.timeStamp+myFreq.timeStampLast);
        }
        else
        {
            myFreq.freq = 64000000/(myFreq.timeStamp-myFreq.timeStampLast);
        }
        myFreq.edgeIndex = 0;
        myFreq.timeStampPulseWidthLast = TIM2->CCR2;
    }
    else
    {
        myFreq.timeStampPulseWidth = TIM2->CCR2;
        if(myFreq.timeStampPulseWidth < myFreq.timeStampPulseWidthLast)
```

```
23          {
24              myFreq.timeStampPulseWidthLast = UINT32_MAX-myFreq.
                    ↪ timeStampPulseWidthLast;
25              myFreq.pulsewidth = myFreq.timeStampPulseWidth-myFreq.
                    ↪ timeStampPulseWidthLast;
26              myFreq.edgeIndex++;
27          }
28          else
29          {
30           myFreq.pulsewidth = myFreq.timeStampPulseWidth-myFreq.
                 ↪ timeStampPulseWidthLast;
31           myFreq.edgeIndex++;
32          }
33      }
34  }
```

Nyní, když máme vše nastavené a výsledky se nám ukládají do proměnných, můžeme implementovat interaface pro ovládání. Ten bude mít stejný postup pro implementaci, jako v příkladu 4.7. Po příchodu nějakého příkazu skrze USART periférii odešleme nazpět informaci o změřené frekvenci a načtené dálce pulzu. Zobrazené výsledky mohou mít například tvar viz obr. 4.19. Zde jsme nastavili příkaz na tvar „GETF“. Po jeho odeslání z terminálu nám mikrokontrolér vrátí zmíněné údaje.



Obr. 4.19: Výstup ze sériového terminálu na PC

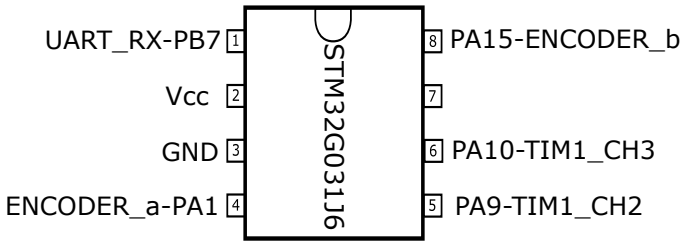
4.9 Aplikace pro řízení DC motoru s enkodérem

Mikrokontrolér STM32G031J6 lze použít jakožto „Smart-circuit“ pro jednoduché řízení DC motorku s enkodérem, kde budeme schopni z našeho mikrokontroléru vyčítat data, jako je rychlost, či poloha a motor ovládat. Nejprve si řekneme, co vše bychom od takového zařízení chtěli. Jako první bude potřeba zjišťovat rychlost motoru. Proto budeme potřebovat 2 vstupy mikrokontroléru jakožto vstup pro enkodér. Dále budeme chtít ponechat komunikaci s ostatními zařízeními pomocí periférie USART, aby se tím dal motor ovládat. V poslední řadě budeme potřebovat výstup PWM signálu pro buzení měniče k elektromotoru.

Vzhledem k omezenému počtu pinů je potřeba uvolnit všech 6 zbylých pinů na mikrokontroléru, tudíž je potřeba odebrat funkci resetu z pinu PF2 (viz 3.3). Dále je třeba nastavit ovládání vstoupení do boot režimu pomocí logické hodnoty na B00T0 pinu mikrokontroléru (viz 3.3). Tímto zajistíme možnost programování mikrokontroléru pomocí USARTu namísto SWD a BOOT0 pin půjde po naprogramování normálně využívat jakožto klasický vstup/výstup. Tímto jsme sice přišli o možnost debugování a resetu mikrokontroléru pomocí tlačítka, ale získali jsme tak k dispozici všech 6 využitelných pinů k běhu naší aplikace.

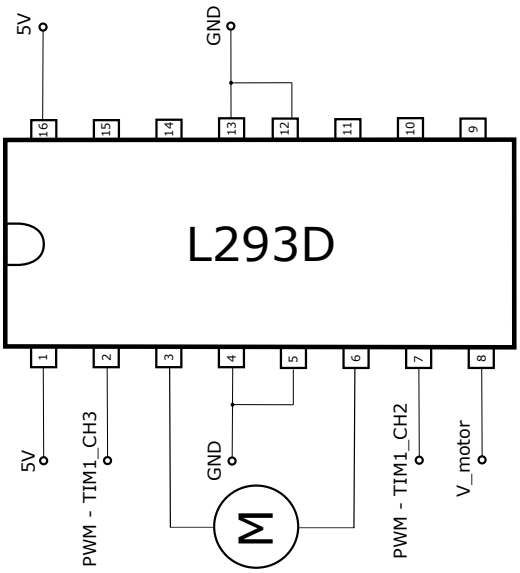
Dále je třeba rozvážit rozmístění jednotlivých funkcí na dostupné piny (viz obr. 4.20).

Po bližším zkoumání možných funkcionalit na jednotlivých pinech zjistíme, že pokud chce mít uživatel k dispozici dva výstupy PWM a zároveň dva piny pro vstup enkodéru, nebude zbývat místo pro oba piny rozhraní USART. Na výstup signálu PWM využijeme časovač TIM1, neboť se nám jeho výstupní piny, kanál č. 2 a kanálu č. 3, hodí do celkového rozložení na našem pouzdře (jedná se o piny PA10-TIM1_CH2 a PA9-TIM1_CH3). Bude však třeba zapnout přemapování pinů PA9 a PA10 namísto pinů PA11 a PA12. Pro vstup enkodéru (dva piny schopné pracovat v režimu vstupu enkodéru) využijeme časovač TIM2, který disponuje 32bitovým CNT registrem, tudíž má velký rozsah čítání a je tímto pro vstup enkodéru vhodný. Konkrétně se jedná o piny PA15 a PA1. Těmito funkcionalitami jsme si však obsadili všechna dostupná místa pro periferii USART, aby mohla pracovat v asynchronním režimu po obou vodičích (Rx/Tx). Zbývá nám však pin PB7, kde lze zapnout funkci přijímače (Rx). Budeme tedy nuceni využívat USART_1 v tzv „Half-duplex“ režimu, kdy bude komunikace probíhat v obou směrech po jednom vodiči, avšak nikdy naráz. Tímto máme hotové rozvržení jednotlivých funkcionalit na pinech pouzdra. Jakožto časovou referenci pro běh programu budeme využívat časovač TIM3, ale v režimu bez výstupu , pouze s ním budeme kontrolovat a upravovat rychlost motoru.



Obr. 4.20: Rozložení funkcí na jednotlivých pinech

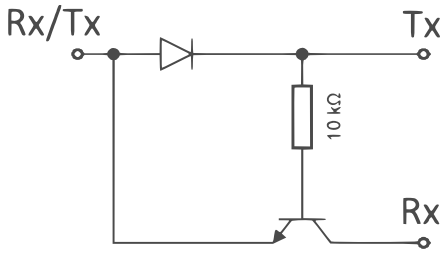
Pro spínání motoru budeme využívat integrovaný obvod L293D od firmy ST. Ten nám umožňuje řídit směr otáčení motoru pomocí dvou nezávislých signálů PWM (viz schéma zapojení 4.21). Tento obvod v sobě zároveň integruje diody zapojené v závěrném směru, aby při spínání motoru nevznikaly na tranzistorech napěťové špičky. Obvod jako takový má na pinu č.16 přivedené napájení pro vnitřní logiku (5V), pin 1 musí být přiveden také ke zdroji 5V (povoluje spínání motoru). Pin č. 8 slouží k přivedení napájecího napětí pro motor a pomocí dvojice pinů č. 3 a č. 6 budeme motor budit pomocí PWM signálu z mikrokontroléru. Piny č. 4, 5, 12 a 13 slouží jakožto zemnící piny.



Obr. 4.21: Schéma zapojení obvodu L293D

Nyní je potřeba připojit k mikrokontroléru enkodér. Výstupní piny enkodéru můžeme připojit napřímo k mikrokontroléru, za předpokladu, že je enkodér napájen pomocí 3,3V, jinak je třeba ještě zapojit obvod pro úpravu napěťové hladiny, abychom nezničili vstup mikrokontroléru.

Nyní už zbývá pouze zapojení pro USART. Jelikož budeme komunikovat s mikrokontrolérem pomocí PC a převodníku USART-USB (např. skrze Nucleo kit), bude třeba připravit jednoduchý obvod pro komunikaci přes jeden vodič, neboť z výstupu PC budeme signál vysílat po obou vodičích (Rx i Tx) a pouze na straně mikrokontroléru budeme nastavovat, kdy má poslouchat a kdy vysílat. Tento obvod (viz obr. 4.22) nám zajistí, že komunikace mezi PC a mikrokontrolérem bude probíhat v pořádku. Nyní pouze připojíme Rx/Tx k pinu PB7 na našem mikrokontroléru a piny Tx a Rx k našemu převodníku USART-USB.



Obr. 4.22: Schéma zapojení pro USART

Tímto bychom měli již kompletně sestavený hardware pro naši aplikaci. Nyní je třeba začít se softwarovou částí. Nejprve nastavíme hodinový signál pro mikrokontrolér (viz 4.4). Tímto máme frekvenci hodinového signálu nastavenou na hodnotu 64MHz. Následně začneme nastavovat jednotlivé časovače, jak pro výstup PWM pomocí TIM1, zpracování dat z enkodéru pomocí TIM2, tak pro časovač TIM3 na zprostředkovaní časové reference. Nejprve je třeba přemapovat výstupní piny PA9 a PA10 namísto pinů PA11 a PA12. To

provedeme pomocí SYSCFG_CFGR1 registru (viz 3.4) zapsáním log. 1 na pozice bitů PA11_RMP a PA12_RMP.

```
RCC->APBENR2 |= RCC_APBENR2_SYSCFGEN;
SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA11_RMP;
SYSCFG->CFGR1 |= SYSCFG_CFGR1_PA12_RMP;
```

Dále je třeba zapnout hodinový signál pro všechny tři časovače, GPIO porty a periférii USART pro komunikaci. Skrze dvojici registrů RCC_APBENR1 a RCC_APBENR2 zapíšeme na pozice bitů TIM1EN, TIM2EN, TIM3EN a USART1EN log. 1. Pro zapnutí GPIO portů zapíšeme na pozice bitů GPIOAEN a GPIOBEN v RCC_IOPENR registru log.1. Tímto jsme zapnuli jednotlivé porty a periférie.

```
RCC ->IOPENR |= RCC_IOPENR_GPIOAEN;
RCC->IOPENR |= RCC_IOPENR_GPIOBEN;
RCC->APBENR2 |= RCC_APBENR2_USART1EN;
RCC->APBENR2 |= RCC_APBENR2_TIM1EN;
RCC->APBENR1 |= RCC_APBENR1_TIM2EN;
RCC->APBENR1 |= RCC_APBENR1_TIM3EN;
```

Nyní je potřeba nastavit samotné časovače. Začneme například časovačem TIM1 pro výstup PWM signálu. Nastavení provedeme stejným způsobem, jako v sekci 3.9.4, až na rozdíl, že budeme používat časovač TIM1 namísto časovače TIM3 a zapneme dvě jednotky „Capture-compare“ v režimu výstupu signálu PWM. Konkrétně se jedná o kanály CH_2 a CH_3. Avšak zatím nebudeme zapisovat žádnou hodnotu do CCR registrů, to až budeme chtít motor spustit. Zároveň ještě nakonfigurujeme výstupní GPIO piny do režimu alternativní funkce.

```
TIM1->CR1 |= TIM_CR1_CEN;
TIM1->BDTR |= TIM_BDTR_MOE;
TIM1->CCMR2 |= TIM_CCMR2_OC3M_2 | TIM_CCMR2_OC3M_1;
TIM1->CCER |= TIM_CCER_CC3E;
TIM1->CCMR1 |= TIM_CCMR1_OC2M_2 | TIM_CCMR1_OC2M_1;
TIM1->CCER |= TIM_CCER_CC2E;
TIM1->PSC = 63;
TIM1->ARR = 100;

GPIOA->MODER &= ~(GPIO_MODER_MODE10_0);
GPIOA->AFR[1] |= GPIO_AFRH_AFSEL10_1;
GPIOA->MODER &= ~(GPIO_MODER_MODE9_0);
GPIOA->AFR[1] |= GPIO_AFRH_AFSEL9_1;
```

Dále nastavíme časovač v režimu pro zpracování dat z inkrementálního enkodéru. Ten bude fungovat tak, že po otáčení motoru doleva, či doprava se nám bude inkrementovat, či dekrementovat CNT registr našeho časovače. Budeme využívat časovač TIM2, který je pro tuto aplikaci vhodný, neboť má čítací rozlišení 32-bitů. Časovač TIM2 lze nakonfigurovat přímo na zpracování dat z enkodéru (upravená funkce režimu Input-capture) zvolením správní funkce v TIM2_SMCR registru pomocí konfigurace SMS (Slave-mode selection) bitů (viz obr. 4.23). Ty nastavíme do hodnoty 0b011 (viz řádek č. 1), čímž zvolíme režim enkodéru na kanálech č. 1 a 2. V tomto režimu se nám budou pulzy z enkodéru načítat/odečítat na náběžné hraně obou vstupů (CCxP a CCxNP bity v TIM2_CCER registru, ale ve výchozím nastavení jsou již takto nastaveny, viz sekce 3.9.6). Následně pouze samotný časovač zapneme a po správném připojení enkodéru k mikrokontroléru bychom měli v TIM2_CNT registru vidět načítanou hodnotu pulzů, která se budem jedním směrem inkrementovat a druhým dekrementovat.

```
TIM2->SMCR |= TIM_SMCR_SMS_0|TIM_SMCR_SMS_1;
TIM2->CR1 |= TIM_CR1_CEN;
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TS[4:3]		Res.	Res.	Res.	SMS[3]
										rw	rw				rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ETP	ECE	ETPS[1:0]		ETF[3:0]				MSM	TS[2:0]			OCCS	SMS[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Obr. 4.23: Registr TIM2_SMCR[1]

Nyní je potřeba nakonfigurovat časovač TIM3, jakožto časovou referenci, pro pravidelné vykonávání nějaké obslužné rutiny. Nastavení provedeme stejně jako v sekci 3.9.5 (viz řádky č. 1 - 5). Zároveň s tím povolíme generování přerušení od periférie TIM3 při „Update-eventu“(řádky č. 7 - 8). V tomto „Update-eventu“ budeme následně vyčítat rychlost enkodéru, a naimplementujeme jednoduchý PI regulátor pro regulování otáček motoru. Tento časovač nastavíme tak, aby k přetečení ARR registru (Update-event) došlo například každou 1ms (hodinový signál má frekvenci 64MHz). Z této časové reference a rozdílu posledních dvou hodnot CNT registru jsme schopni zjistit rychlost otáčení motoru.

```
1  TIM3->PSC = 63;
2  TIM3->ARR = 1000;
3  TIM3->DIER |= TIM_DIER_UIE;
4  TIM3->CR1 |= TIM_CR1_CEN;
5  TIM3->CR1 |= TIM_CR2_MMS_1;
6
7  NVIC_EnableIRQ(TIM3_IRQn);
8  NVIC_SetPriority(TIM3_IRQn,0);
```

Jako poslední nám chybí nakonfigurovat periférii USART_1. Zde nakonfigurujeme rychlost přenosu (Baud-rate) na 9600bit/s , povolíme vysílání i přijímání dat (bity TE a RE v USART1_CR1 registru). Povolíme generování přerušení od periférie při novém přijatém znaku (bit RXNEIE_RXFNEIE v USART1_CR1 registru) a povolíme funkce periférie samotné (bit UE v USART1_CR1 registru). Podrobné vysvětlení naleznete v sekci 3.7.3. Následně zapneme funkci alternativní funkce na pinu PB7, jakožto funkci USART_Rx. Komunikaci po jednom vodiči budeme řešit pomocí bitu SWAP v USART1_CR2 registru, který nám unitář periférie prohodí piny Tx a Rx a budeme schopni pomocí tohoto pinu následně data odesílat, nikoliv přijímat. Algoritmus pro čtení a odesílání dat bude popsán níže.

```
1  USART1->BRR = 0x1A0C;
2  USART1->CR1 |= USART_CR1_RE;
3  USART1->CR1 |= USART_CR1_TE;
4  USART1->CR1 |= USART_CR1_RXNEIE_RXFNEIE;
5  USART1->CR1 |= USART_CR1_UE;
6
7  GPIOB->MODER &= ~(GPIO_MODER_MODE7_0);
```

Nyní máme veškeré periférie zapnuté a je potřeba implementovat software pro jejich ovládání pomocí USART příkazů. Nejprve naimplementujeme software pro čtení rychlosti z enkodéru a jednoduchý PI regulátor. Celá tato část kódu se bude vykonávat při přerušení od časovače TIM3, tedy každou 1ms. Rychlost zde budeme počítat podle aktuální změřené polohy (stavu TIM2_CNT registru), od které odečteme poslední načtenou hodnotu. V případě, že by se motor otáčel obráceně, odečteme hodnoty opačně (viz řádky kódu pod). Směr otáčení zde máme uložený ve struktuře pod proměnnou „direction“, kde 1 značí CW (clockwise - směr hodinových ručiček) a 0 značí CCW (counter-clockwise - opačný směr otáčení hodinových ručiček). Do proměnné RPM je následně uložena výsledná hodnota změřené rychlosti přepočtená na jednotku RPM (revolutions per minute - otáčky za minutu). V této rovnici výpočtu rychlosti nám SINGLE_ROTATE značí počet impulzů

enkodéru na jednu otáčku motoru a SPEED_UPDATE_TIME nám značí čas frekvenci, s jakou tento výpočet opakujeme (frekvence jak časo je zpracovávno přerušení časovače TIM3). Tato rychlost je následně ještě filtrována jinou funkcí. Implementace této funkce lze nalézt v přiloženém projektu v souboru „dc_library.c“ spolu s ostatními věcmi, jako je PI regulátor, nebo kód pro výpočet náběžné rampy rychlosti (implementace rychlosti akcelerace) motoru.

```
1  if(pMc->speedControl.direction)
2    {
3      if(pMc->encoder.prevPosition < pMc->encoder.position)
4        {
5          pMc->speedControl.RPM = (((float)(pMc->encoder.position-pMc->encoder
               ↪ .prevPosition)*60000000.0)/((float)SINGLE_ROTATE*(float)
               ↪ SPEED_UPDATE_TIME));
6        }
7        if(pMc->encoder.prevPosition == pMc->encoder.position)
8        {
9          pMc->speedControl.RPM = 0;
10       }
11   }
12   else
13   {
14       if(pMc->encoder.prevPosition > pMc->encoder.position)
15       {
16         pMc->speedControl.RPM = (((float)(pMc->encoder.prevPosition-pMc->
               ↪ encoder.position)*60000000.0)/((float)SINGLE_ROTATE*(float)
               ↪ SPEED_UPDATE_TIME));
17       }
18      if(pMc->encoder.prevPosition == pMc->encoder.position)
19       {
20         pMc->speedControl.RPM = 0;
21       }
22   }
```

Pro samotné ovládání motoru je implementováno několik funkcí, které motor zastaví, spustí, nebo změní jeho rychlost. Kompletní funkce a jejich danou implementaci lze opět nalézt v souboru „dc_library.c“. Zde si pouze řekneme k čemu funkce slouží.

- Funkce motor_start(int direct,int speed, motorControl_HandleTypeDef *pMc) spustí motor, kde vstupní parametry jsou směr otáčení, rychlost které má motor dosáhnout a hlavní struktura pro řízení motoru (definice viz projekt v soboru „dc_library.h“
- Funkce motor_stop(motorControl_HandleTypeDef *pMc), která motor zastaví má na vstupu pouze strukturu pro řízení motoru
- Funkce change_speed(int newSpeed, motorControl_HandleTypeDef *pMc) změní rychlost otáčení motoru za běhu a na vstupu má požadovanou novou rychlost motoru a opět hlavní strukturu pro řízení motoru

Nyní, když máme definované funkce pro řízení motoru, se zaměříme na ovládání celého běhu pomocí USART. Ve výchozím nastavení máme periférii USART_1 již spuštěnou, máme vyvedený pin Rx, pro příjem dat a zapnutý a nakonfigurované přerušení od periférie po přijetí nového znaku. Funkce pro zpracování přijatých dat bude velmi podobná jako v příkladech 4.7, nebo 4.8, kde při přijímání nových dat data vždy ukládáme do pole bajtů a pokud byl přijat znak „“ (line-feed), tak vyčteme, jaká data byla přijata a podle toho spustíme danou funkci. Nejprve si zadefinujeme jednotlivé příkazy pro ovládání motoru.

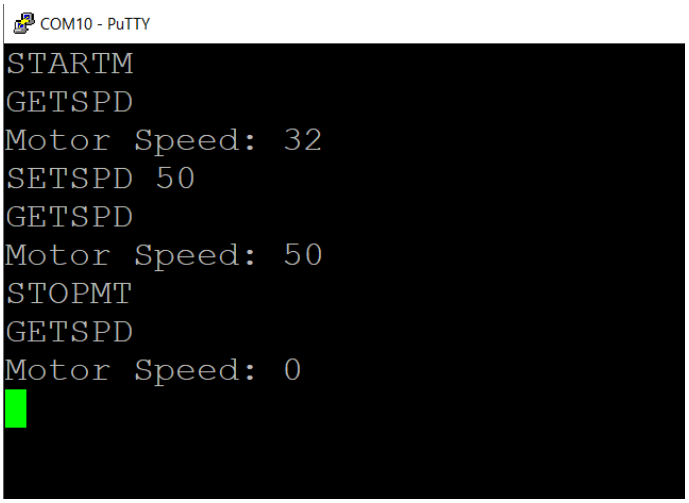
- Příkaz „STARTM“ spustí motor
- Příkaz „STOPMT“ zastaví motor

- Příkaz „GETSPD“ nám vrátí rychlost motoru
- Příkaz „SETSPD xxx“ nám nastaví novou rychlost za běhu motoru, kde xxx značí novou rychlost, výchozí rychlost pro každý start je definována v souboru „dc_library_config.h“
- Příkaz „ACCELE xxx“ nám nastaví akceleraci motoru, kde xxx značí zadanou hodnotu
- Příkaz „DIRECT CW/CCW“ nám změní směr otáčení motoru, podle toho zda uživatel zadá CW nebo CCW

Implementace všech příkazů bude velmi podobná. Při přijetí všech (s výjimkou „GETSPD“ pouze změníme nějakou proměnnou v algoritmu a ten se tomu přizpůsobí, nebo zavolá některou z implementovaných funkcí pro řízení motoru. Při příkazu „GETSPD“ však potřebujeme vrátit hodnotu změřené rychlost. Vzhledem k tomu, že nemáme na pouzdře pin pro Tx (odesílání) funkce bloku USART_1, ale pouze Rx (příjem), je potřeba pomocí bitu SWAP v USART1_CR2 registru navzájem zaměnit Tx a Rx piny mezi sebou. Na pozici tohoto bitu lze však zapisovat pouze ve chvíli kdy, je v bitu UE v registru USART1_CR1 zapsána log.0. Tudíž nejprve zapíšeme logickou 0 na jeho pozici, poté zaměníme piny mezi sebou, poté na pozici bitu UE opět zapíšeme log.1, tím periférii znovu spustíme a můžeme data odesílat. Díky obvodu viz obr. 4.22 se nám data přenesou k druhému zařízení bez jakéhokoliv problému (akorát v tu chvíli není schopen náš mikrokontrolér data přijímat). Po odeslání veškerých dat opět nastavíme vše zpět a mikrokontrolér bude opět v režimu, kdy čeká na další příkaz. Ukázka kódu pod textem znázorňuje způsob prohození pinů při vyhodnocení přijatého příkazu v přerušení od USART_1 periférie.

```
1  USART1->CR1 &= ~(USART_CR1_UE);
2  USART1->CR2 |= (USART_CR2_SWAP);
3  USART1->CR1 |= USART_CR1_UE;
4  char myTxData[30];
5  uint8_t size = sprintf(myTxData, "Motor Speed: %d \r\n",(int) pMc->
       ↪ speedControl.filteredRpm);
6  for(uint8_t i = 0;i<size;i++)
7  {
8   USART1->TDR = myTxData[i];
9   while ((USART1->ISR & USART_ISR_TC) != USART_ISR_TC)
10  {
11  }
12 }
13 USART1->CR1 &= ~(USART_CR1_UE);
14 USART1->CR2 &= ~(USART_CR2_SWAP);
15 USART1->CR1 |= (USART_CR1_UE);
16 USART1->ICR |= USART_ICR_TCCF;
```

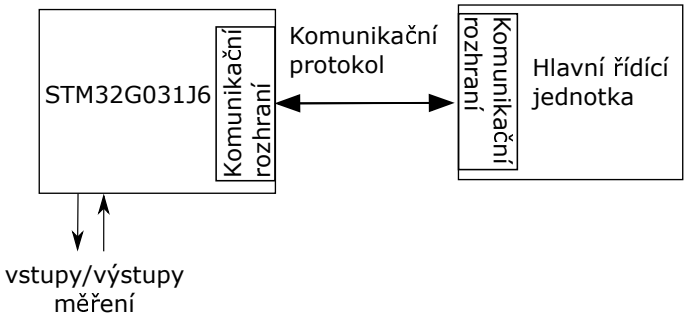
Nyní, když máme vše správně spuštěno, připojeno a otevřený terminál pro sériovou komunikaci na PC, nebo připojený nějaký jiný mikrokontrolér pro ovládání, můžeme motor spustit, zastavit a nastavit jeho parametry pro běh. Kompletní zdrojový kód s implementací všech nutných částí kódu pro běh motoru naleznete v příloze mojí BP.



Obr. 4.24: Ukázka výstupu komunikace na sériovém terminálu v PC

5 Další možnosti využití procesoru STM32G031J6, jakožto obvod typu „Smart-circuit“

Zadáním mé bakalářské práce je zhodnotit možnosti využití mikrokontroléru STM32G031J6, jakožto obvod typu „Smart-circuit“. Tím je myšleno využít mikrokontrolér jakožto „sateltiní“ čip připojený pomocí komunikačního rozhraní, ať už USART, SPI, či I^2C. Můžeme jej následně připojit k zařízení jako je například Raspberry Pi, Arduino, či jakémukoliv jinému výkonnějšímu procesoru, který disponuje některým z výše uvedených komunikačních rozhraní. Tento čip zde bude zastupovat jednoduché funkce jako jsou čítač impulzů, převodník ADC s možností průměrování a násobení několika hodnot, nebo jednoduchý obvod na zpracování dat z enkodéru, neboť jednotky jako Raspberry Pi jsou sice výkonné na výpočetní výkon, ale neintegrují ve svém pouzdře mnohé periférie, jako jsou čítače nebo ADC převodníky. Tímto způsobem můžeme tedy velmi ulehčit určité úlohy pro nadřazený procesor tím, že je v takto malém pouzdře zpracujeme, přepočítáme a následně je budeme pomocí nějakého komunikačního rozhraní odesílat nadřazeném systému.



Obr. 5.1: Myšlenka obvodu typu Smart-circuit

V kapitole 4 jsme si ukázali postupy pro realizaci některých možností využití obvodu v praxi, ale existuje jich mnohem více. V této kapitole si nastíníme, které piny a funkce mikrokontroléru nám zůstanou dostupné, když vždy využijeme některého z níže uvedených komunikačních rozhraní, a tím si ukážeme dostupné možnosti využití celého mikrokontroléru, jakožto obvod typu „Smart-circuit“.

5.1 Princip funkcionality komunikačního rozhraní

Když člověk řekne slovo komunikační rozhraní, jedná se o velmi široký pojem. V této sekci si shrneme výhody a nevýhody použití různých rozhraní dostupných v našem mikrokontroléru. Komunikační rozhraní lze realizovat pomocí několika komunikačních protokolů dostupných v našem zařízení.

- USART - Universal Synchronous/Asynchronous Receiver and Transmitter
- I^2C - Inter-Integrated Circuit
- SPI - Serial Peripheral Interface

Každé z výše uvedených rozhraní má jisté výhody oproti jiným. Rozhraní USART má výhodu ve své jednoduchosti, možnosti komunikace po 2 vodičích a možnosti snadného propojení s PC pomocí dostupných USART-USB převodníků, avšak nelze pomocí něj adresovat data na více zařízení. Rychlost přenosu dat pomocí USART může dosahovat až 8Mbit/s. Oproti tomu má rozhraní SPI vyšší přenosové rychlosti (až 32Mbit/s) a uživatel u něj může připojit více zařízení typu „Slave“ a volit přímo dané zařízení, ale komunikace probíhá po 3 vodičích + N vodičů podle počtu připojených zařízení, pokud je v režimu „master“. Avšak v našich aplikacích nejspíše nebude potřeba adresovat více zařízení, neboť náš mikrokontrolér bude „slave“ zařízení a „master“ zařízení bude nadřazený mikrokontrolér a pokud ten nebude mít na rozhraní SPI připojeno více zařízení, bude možné komunikovat po 3 vodičích. V poslední řadě lze využít protokol I^2C, který nabízí přenosovou rychlost až 1Mbit/s, ale s možností komunikace po 2 vodičích se všemi ostatními zařízeními, na rozdíl od SPI, kde s každým dalším zařízením přibyde potřeba dalšího pinu navíc.

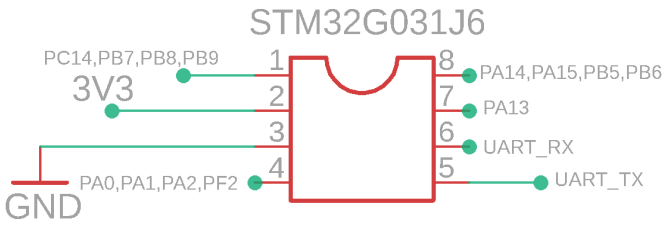
Nyní když si uživatel zvolí vybrané komunikační rozraní, je potřeba naimplementovat softwarovou část celého rozhraní („interface“). To by mělo mít funkce jako „Vyčti daný parametr“, „Nastav daný parametr“, popř. jiné, dle možností cílené aplikace. Tímto vytvoříme samostatně fungující blok, kterému budeme posílat pouze příkazy, co má a co nemá dělat a můžeme pomocí nadřazeného procesoru monitorovat jeho chod.

5.2 Seznam dostupných funkcionalit po volbě komunikačního protokolu

V této sekci si shrneme, jaké všechny periférie a jejich vstupy/výstupy máme k dispozici, když využijeme různé druhy komunikačních protokolů. Tím nastíníme možnosti realizace dalších obvodů typu „Smart-circuit“, které jsme nestihli zmínit v sekci 4. Budeme předpokládat, že máme k dispozici 6 fyzických pinů na pouzdře mikrokontroléru, tedy máme odstavenou funkci NRST pinu (viz sekce 3.3).

5.2.1 Použití komunikačního rozhraní USART_1

V tomto případě využijeme pro komunikaci USART piny PA9 a PA10, na fyzickém pouzdře se jedná o piny č. 5 a č. 6 (viz obr. 5.2). V tomto případě nám zbývají k dispozici následující funkce (viz tabulka pod). V řádku k danému fyzickému pinu jsou vždy uvedeny periférie, které lze na skupině GPIO pinů, vyvedených na fyzický výstup, nalézt. Funkci základního vstupu/výstupu u všech pinů bude vynechána, neboť je dostupná na všech pinech, zároveň nebudou zmíněny všechny funkce, jen ty nejdůležitější.



Obr. 5.2: Schéma zapojení pro USART_1

Fyzický pin	Skupina GPIO pinů	Funkce 1	Funkce 2	Funkce 3	Funkce 4
8	PA14,PA15, PB5,PB6	USART_TX	ADC_CH18	TIM1_CH3 TIM2_CH1 TIM3_CH2	I2C_SCL
7	PA13	ADC_CH17	SWD_SWDIO		
4	PA0,PA1, PA2,PF2	MCO	ADC_CH0 ADC_CH1 ADC_CH2	TIM2_CH1 TIM2_CH2 TIM2_CH3	
1	PC14,PB7, PB8,PB9	USART_RX	ADC_CH11	TIM16_CH1 TIM17_CH1	I2C_SDA

Tabulka 2: Tabulka dostupných periférií při použití komunikačního rozhraní USART_1

Když si projdeme seznam funkcí dostupných na jednotlivých pinech (viz tabulka 2), zjistíme že jsme schopni navrhnout hned několik různých obvodů typu „Smart-circuit“. Jednotlivé funkce dostupné na různých fyzických pinech můžeme libovolně kombinovat a tím vytvořit velmi komplexní uživatelský obvod, který odpovídá požadavkům aplikace, kde by se obvod využíval.

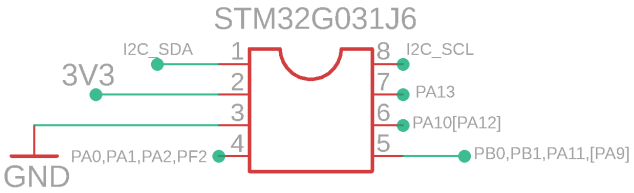
Z oblasti využití časovačů můžeme například využít piny č. 8 a 4 pro vstup signálu z enkodéru a zpracovávat tento signál pomocí 32-bitového časovače. Dalším možným využitím může být jednoduchý obvod pro generovaní PWM signálu za využití výstupu časovače TIM1, TIM2, nebo TIM3, například pro nastavování svítivosti diody, nebo na modulování frekvenčního výstupu z piezo bzučáku.

Kdybychom chtěli využívat obvod jako měřící zařízení, mohli bychom využít například kombinaci všech pinů a zhotovit komplexní systém na měření pomocí ADC převodníku. Můžeme jej využít například na měření výkonu, kde můžeme za pomocí převodníku proud napětí měřit proud a napětí hned dvou zařízení naráz a následně z těchto dat vypočítat výkon.

Toto rozvržení zahrnuje pouze jednu ze čtyř možností, jak nastavit piny pro komunikaci pomocí bloku USART_1. Pokud bychom použili jiné rozložení těchto pinů, měli bychom k dispozici jiné složení dostupných funkcí na jednotlivých pinech a tím i změnili celkové možnosti využití.

5.2.2 Použití komunikačního rozhraní I²C_1

V tomto případě si naznačíme, jaké varianty dostupných periférií mám k dispozici po zvolení komunikačního rozhraní I²C_1. Rozhraní samotné pro svoji funkci využívá dva piny mikrokontroléru, viz obr. 5.3.



Obr. 5.3: Schéma zapojení pro komunikační protokol I^{2}C_1

Fyzický pin	Skupina GPIO pinů	Funkce 1	Funkce 2	Funkce 3	Funkce 4
7	PA13	ADC_CH17	SWD_SWDIO		
6	PA10[PA12]	TIM1_CH3	ADC_CH16	SPI2_MOSI	
5	PB0, PB1 PA8, PA11[PA9]	ADC_CH8 ADC_CH9 ADC_CH15	TIM1_CH1 TIM1_CH2 TIM1_CH4 TIM3_CH3 TIM3_CH4 TIM14_CH1	USART_RX	SPI2_MISO
4	PA0,PA1, PA2,PF2	SPI2_SCK	ADC_CH0 ADC_CH1 ADC_CH2	TIM2_CH1 TIM2_CH2 TIM2_CH3	USART_TX

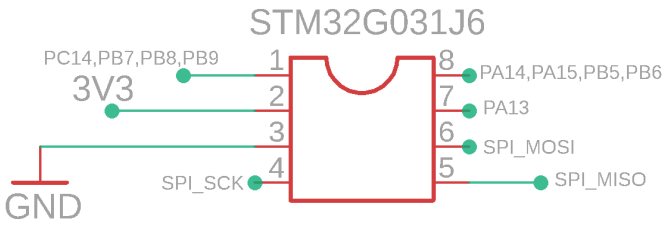
Tabulka 3: Tabulka dostupných periférií při použití komunikačního rozhraní I^{2}C_1

V následujícím rozvržení pinů (viz tabulka 3), při použití komunikačního rozhraní I^{2}C_1 jsme schopni realizovat podobné příklady jako v minulém případě, tedy různé měřící systémy za pomoci využití ADC, nebo zařízení, které bude generovat signál PWM pro buzení nějakého zařízení. V této konfiguraci pinů pro I^{2}C_1 však nejsme schopni zaznamenávat signál z enkodéru pomocí 32-bitového časovače TIM2. Jsme však schopni jednoduše realizovat například převodník I^{2}C→USART nebo I^{2}C→SPI, s možností rozšíření například o měření napětí pomocí ADC.

Nejedená se však o jedinou možnou konfiguraci dvojice pinů pro komunikaci. V tomto rozložení pinů využíváme I^{2}C_1, ale lze využít i druhý modul pro komunikaci, tedy I^{2}C_2, kde budeme namísto fyzických pinů č. 1 a 8 využívat piny č. 6 a 5. Z něj bychom již měli možnost realizovat čtení dat z enkodéru, opět bychom mohli realizovat různá měřící zařízení založená na převodníku ADC, nebo generátory PWM signálu. Možnost realizace převodníku I^{2}C→USART a I^{2}C→SPI nám také zůstane. Pro převodník I^{2}C→USART bude nutné využít fyzické piny č. 1 a 8, nebo pro verzi I^{2}C→SPI piny č. 1, 4 a 8.

5.2.3 Použití komunikačního rozhraní SPI_1

V tomto příkladu budeme komunikovat s nadřazeným mikrokontrolérem pomocí rozhraní SPI_1. V našem mikrokontroléru se nacházejí dva bloky pro komunikaci SPI, kde v tomto příkladu si zvolíme SPI_1 (viz obr. 5.4). V případě, že bychom využívali blok SPI_2, bude celkové rozložení pinů jiné a proto zbudou i jiné dostupné funkce na jednotlivých fyzických pinech mikrokontroléru.



Obr. 5.4: Schéma zapojení pro komunikační protokol SPI_1

Fyzický pin	Skupina GPIO pinů	Funkce 1	Funkce 2	Funkce 3	Funkce 4
8	PA14,PA15, PB5,PB6	USART_TX	ADC_CH18	TIM1_CH3 TIM2_CH1 TIM3_CH2	I2C_SCL
7	PA13	ADC_CH17	SWD_SWDIO		
1	PC14,PB7, PB8,PB9	USART_RX	ADC_CH11	TIM16_CH1 TIM17_CH1	I2C_SDA

Tabulka 4: Tabulka dostupných periférií při použití komunikačního rozhraní SPI_1

Vzhledem k tomu, že nyní využíváme komunikaci pomocí rozhraní SPI_1, zbývají nám k dispozici pouze 3 piny (viz tabulka 4). Pomocí nich můžeme opět realizovat některý z převodníku SPI→I^2C nebo SPI→USART, dále máme k dispozici 2 výstupy časovačů pro generování PWM signálu, nebo 3 kanály ADC pro měření různých elektrických veličin.

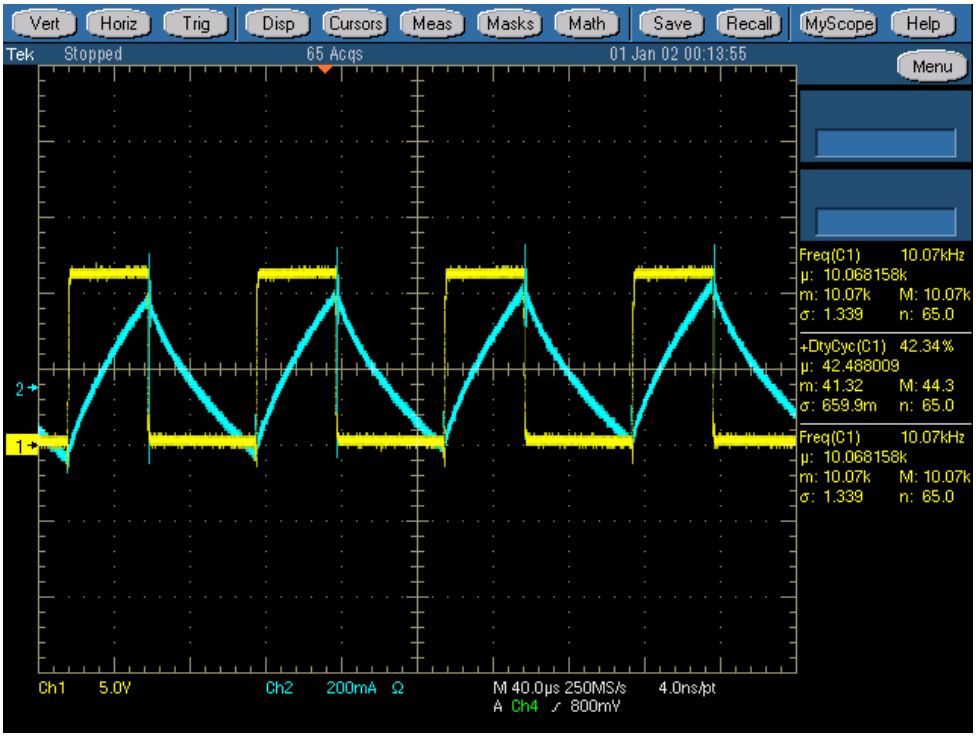
5.3 Další možnosti realizací obvodu typu „Smart-circuit“

V této kapitole si nastíníme různé projekty realizovatelní za pomoci obvodu STM32G031J6. Bude se jednat o jakýsi návod, či seznam nápadů, co by s obvodem, kromě již zmíněných úloh, šlo dělat a kam by šly rozšířit možnosti již hotových projektů

5.3.1 Měření proudu vinutím kotvy stejnosměrného motoru

Obvod by šel například využít jakožto chytrý obvod, který bude hlídat proud odebíraný DC motorem, kdy při příliš vysokém proudu pošle nadřazenému procesoru informaci, že je motor potřeba zastavit, nebo může naopak pravidelně odesílat data o průběhu proudu a tím může nadřazený mikrokontrolér monitorovat chod motoru. Zároveň bychom mohli motor i řídit signálem PWM (návod viz sekce 4.9).

Pro měření proudu budeme potřebovat jeden vstup na mikrokontroléru, který na sobě má vstup pro ADC. Dále budeme potřebovat převodník proud→napětí a poté můžeme již číst proud motorem. Pro čtení využijeme ADC v režimu externího spouštění, neboť budeme potřebovat získat „špičkovou“ hodnotu proudu, kterou můžeme následně filtrovat, například pomocí matematického filtru klouzavých průměrů. Pokud by chtěl totiž uživatel měřit proud motoru, který je spínán pomocí PWM (pulzně-šířková modulace) signálu, bude potřeba spouštění konverze synchronizovat se samotným PWM signálem, neboť průběh proudu motorem není lineární přímka (viz obr. 5.5).



Obr. 5.5: Příklad možného průběhu napětí a proudu u DC motoru

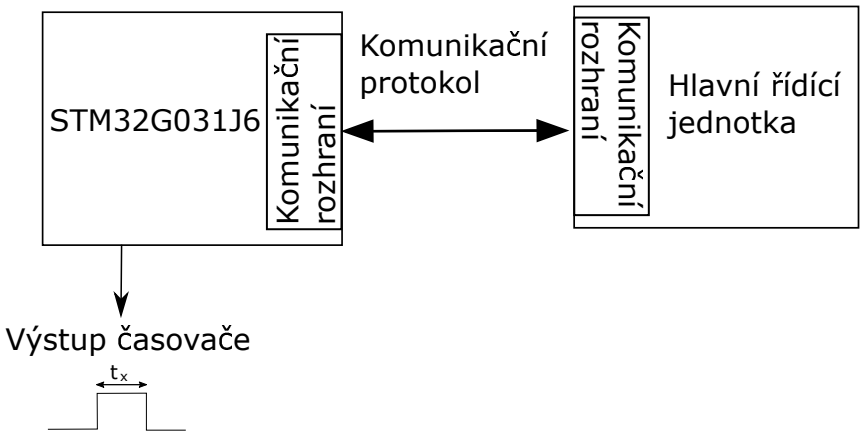
Na obrázku můžeme vidět, že proud (modrý průběh) dosahuje maximální hodnoty ve chvíli, kdy se mění stav napětí (žlutý průběh) ze „sepnuto“ na „vypnuto“, následně začne proud vinutím zase klesat. Tudíž bude potřeba nastavit spouštění ADC zároveň s klesající hranou PWM signálu, kterým budíme motor. V případě, že budíme motor pomocí časovače jednoho mikrokontroléru, můžeme využít generování signálu při „capture-compare“ eventu. V tu chvíli budeme mít k dispozici referenční signál pro spuštění nové konverze, která se provede v okamžiku, kdy je proud nejvyšší, neboť v tu chvíli dojde k „caputre-compare“ eventu, který udává délku střídy našeho PWM signálu. V případě, že by byl buzen motor z jiného zdroje, bude potřeba využít externího spouštěcího signálu, který bude přiveden na některý z volných pinů našeho mikrokontroléru

5.3.2 Generátor impulzů přesně definované délky

Pro některé aplikace se může hodit generovat impulz, který má přesně definovanou délku stavu „high“, tedy zapnuto. Toho jsme softwarovým zapínáním a vypínáním pinů těžko schopni dosáhnout, ale můžeme k tomu využít časovač nastavený v „One-pulse“ režimu, kdy je odeslán jeden pulz přesně definované délky na výstup mikrokontroléru. Jedná se o obdobnou aplikaci, jako je generování PWM signálu, ale jde pouze o jediný impulz, nikoliv periodický signál. Tímto způsobem si můžeme vytváře vlastní „komunikační protokoly“ pro jednoduché ovládání různých zařízení, která nepodporují běžné protokoly, jako je SPI, či I^2C.

Pro tuto aplikaci bychom potřebovali komunikační piny pro jedno z uvedených rozhraní, tedy 2 piny v případě využití I^2C_1 nebo USART_1, popř. tři piny v případě využití SPI_1. Následně bychom potřebovali jeden pin, na nějž jsme schopni namapovat alternativní funkci časovače, který disponuje „capture-compare“ kanálem s možností výstupu signálu. Tento časovač bychom spustili v režimu „One-pulse“ módu a mohli bychom

tak na základě přijatého příkazu skrze komunikační rozhraní generovat daný pulz definované délky t_x na výstupu. Podobný příklad, ale za pomoci využití softwarového spínání. jsme si ukázali v sekci 4.6.

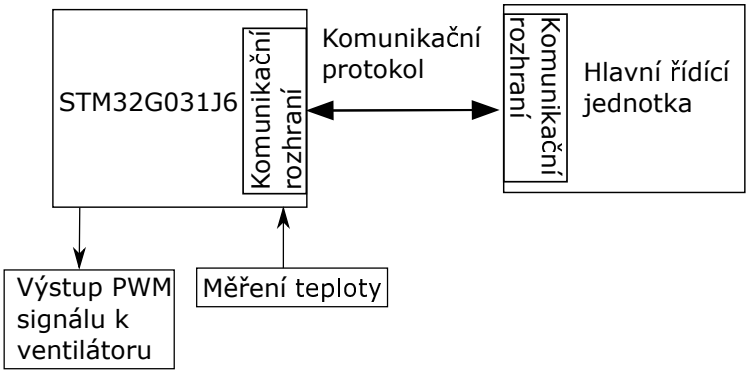


Obr. 5.6: Blokové schéma satelitního generátoru pulzů definované délky

5.3.3 Regulace teploty

Mikrokontrolér by bylo možné využít například jako regulátor teploty nějakého zařízení. Například při použití teplotního čidla a připojeného DC ventilátoru k chladiči můžeme jednoduše řídit otáčky ventilátoru v závislosti na změřené teplotě. Teplota by mohla být vyčtena např. z termistoru pomocí ADC a zpracována na žádanou hodnotu. Následně je potřeba naimplementovat regulátor, který bude na základě teploty upravovat střídu PWM signálu, pomocí nějž se budou řídit otáčky ventilátoru.

Pro tento příklad bychom opět potřebovali 2 - 3 piny pro zvolení komunikační rozhraní, následně jeden pin na čtení teploty a jeden pin na regulaci otáček. Vzhledem k odporovému charakteru termistorů, kdy termistor mění svůj odpor v závislosti na své teplotě, by bylo možné realizovat měření teploty obdobným způsobem jako v příkladu 4.5.4, kde bychom místo neznámého odporu zapojili termistor. Na základě změření hodnoty bychom odpor přepočítali na teplotu a tu se následně snažili otáčkami ventilátoru regulovat na referenční (žádanou) hodnotu. Orientační blokové schéma zapojení je znázorněno v obrázku 5.7.

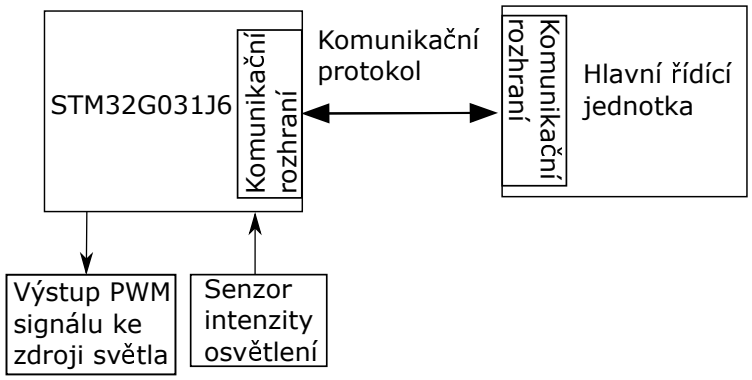


Obr. 5.7: Blokové schéma satelitního regulátoru teploty

5.3.4 Regulace intenzity osvětlení

Uživatel může vytvořit aplikaci, která bude pomocí fototranzistoru měřit intenzitu osvětlení a podle ní měnit intenzitu doplňujícího osvětlení. Čtení intenzity osvětlení lze realizovat například pomocí čtení napětí na odporu při použití fototranzistoru jako zdroje proudu, nebo měření vlastního odporu fotorezistoru. Následně intenzitu doplňkového osvětlení může uživatel měnit nastavenou střídou PWM signálu.

Fotorezistor má odporový charakter, tudíž v závislosti na intenzitě okolního osvětleni mění vlastní odpor a tím převádí míru osvětlení na elektrickou veličinu (napětí). Měření by bylo možné zapojit stejně, jako v případě 4.5.4, kde využíváme poměrového měření odporů. Tedy měli bychom nějaký referenční odpor a druhý, neznámý odpor, by byl vložený fotorezistor. Z následných hodnot změřeného odporu bychom dopočetli intenzitu osvětlení a dle ní mohli například regulovat svit osvětlovací diody, nebo žárovky. V případě využití fototranzistoru bychom měřili úbytek napětí vytvořený na odporu známé velikosti. Fototranzistor se totiž chová jako zdroj proudu v závislosti na intenzitě okolního osvětlení. Tudíž se bude měnit proud protékající odporem známé velikosti a dle něj se bude měnit úbytek napětí na něm. Ten následně změříme a dopočteme z něj intenzitu okolního osvětlení. Orientační blokové schéma zapojení viz obr. 5.8.



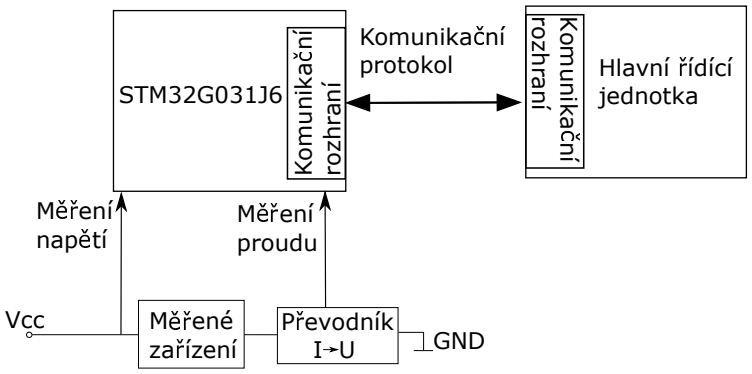
Obr. 5.8: Blokové schéma satelitního regulátoru osvětlení

5.3.5 Měření příkonu

Uživatel může naimplementovat měření příkonu nějakého zařízení, například opět motoru, kdy si bude měřit jednak napětí přivedené k zařízení a následně proud jím protékající (viz měření proudu motorem 5.3.1), kde následně obě změřené hodnoty mezi sebou vynásobí a výslednou vypočtenou hodnotu může odeslat nadřazenému zařízení, které se nebude o samotné měření a výpočet muset starat.

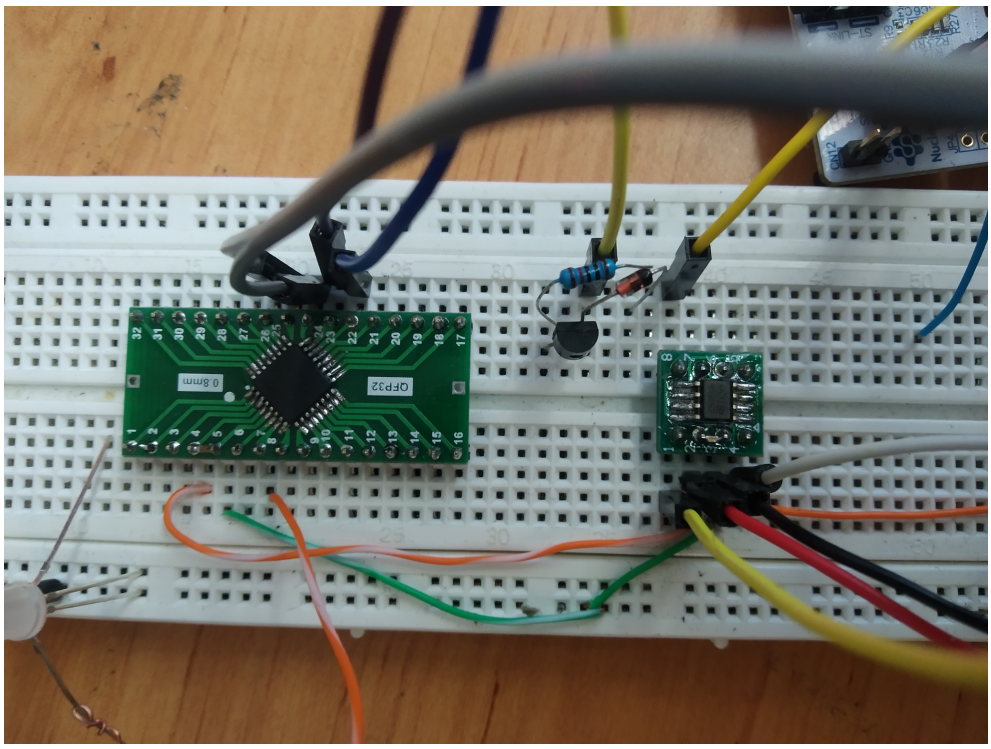
Pro realizace na měření výkonu jednoho zařízení bychom potřebovali 2-3 piny pro komunikační rozhraní (dle jeho volby) a následně dva piny mikrokontroléru s vyvedeným vstupem k periférii ADC. Na jednom kanálu bychom měřili napětí na zařízení (v případě, že by bylo vyšší než 3,3V bude potřeba jej snížit, například pomocí napěťového děliče) a na druhém kanálu ADC bychom měřili proud jím protékající. Pro měření proudu bychom potřebovali převodník proud→napětí, který bude mít na výstupu hodnotu mezi 0-3.3V. Následně by bylo potřeba změřená data průměrovat s nějakou periodou, abychom získali střední hodnotu. Pak bychom mezi sebou hodnoty vynásobili a převedli na správné jednotky a výslednou změřenou hodnotu odesílali řídící jednotce. Ta by se tak nemusela starat

o samotné měření a pouze když potřebuje znát hodnotu výkonu, tak by vyčetla poslední odeslanou hodnotu naším mikrokontrolérem. Orientační schéma zapojení viz obrázek 5.11.

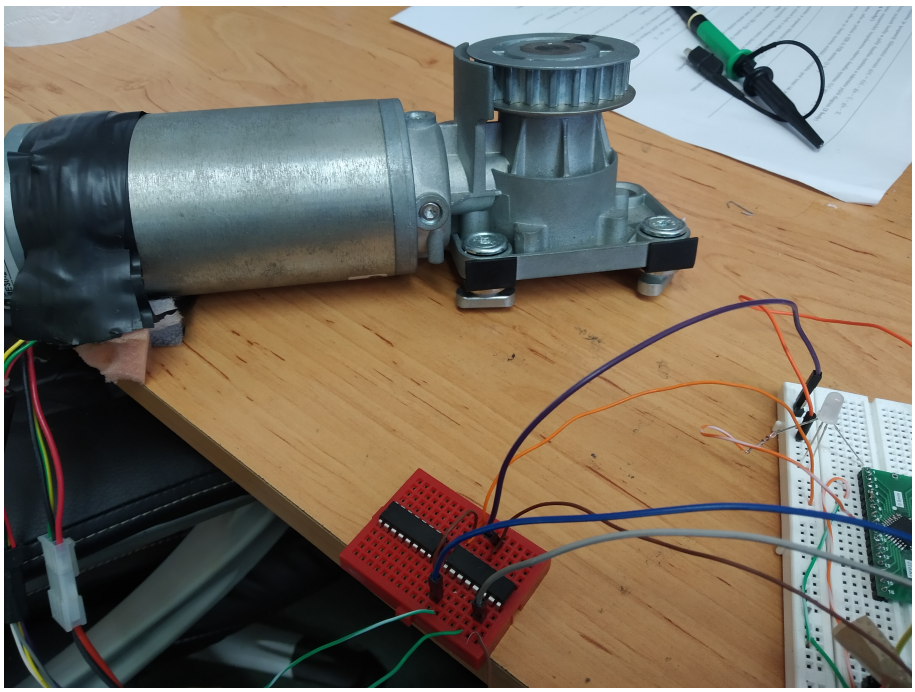


Obr. 5.9: Blokové schéma satelitního měřiče výkonu

5.4 Obrázky reálného zapojení mikrokontroléru na nepájivém poli



Obr. 5.10: Zapojení STM32G031 v pouzdrech SO8N a LQFP32



Obr. 5.11: Zapojený DC motor

6 Závěr

Zadáním mé práce bylo seznámit se s mikrokontroléry řady STM32G031, analyzovat možnost jejich využití jakožto obvod typu „smart-circuit“, kde obvod může zastávat funkci jednoduchých zákaznických obvodů. Dále bylo mým úkolem realizovat jednoduché návrhy realizace těchto zařízení a sepsat metodiku jejich návrhu, neboť text bude využit jako podpůrný studijní materiál pro studenty předmětu Vestavných systému na Fakultě elektrotechnické ČVUT.

V práci byla probrána problematika práce s mikrokontroléry řady STM32G031, konkrétně s provedením v pouzdře SO8N (STM32G031J6), která se týkala popisu jednotlivých bloků mikrokontroléru a problémy spojenými s malým počtem dostupných pinů na pouzdře. Vzhledem k budoucímu možnému využití tohoto textu, jakožto studijních materiálů pro výuku vestavných systému byly kladen důraz na teoretický popis funkcionalit nejdůležitějších bloků mikrokontroléru, jako jsou vstupní/výstupní brány GPIO, analogově-digitální převodník ADC, časovače a komunikační periférie USART. Tyto periférie mají popis dostupný v referenčním manuálu k mikrokontroléru, ale ten je velmi obsáhlý a studenti se v něm těžko orientují.

Dále byla vybrána sada příkladů možných realizací chytrých obvodu. Ke všem zvoleným úlohám byly vypracovány názorné ukázky jejich implementace, popis zapojení a realizace celé úlohy. Byla vytvořena jednoduchá schémata zapojení úloh a popis nastavení jednotlivých bloků mikrokontroléru tak, aby to byli studenti schopni jednoduše uchopit a mohli se tím způsobem naučit s mikrokontroléry pracovat jednoduše a efektivně. Jednotlivé úlohy byly následně zpracovány a naimplementovány v programovacím jazyku C za využití integrovaného vývojového prostředí MDK-ARM Keil. Všechny zhotovené projekty jsou přiloženy v příloze BP a mohou sloužit jako další zdroj pro inspiraci k vlastním projektům pro studenty, ať už při výuce, nebo v individuálních projektech.

Celá práce byla završena souhrnem jiných dostupných možností využití mikrokontroléru STM32G031J6, jako obvodu typu „smart-circuit“. Zde byla popsána metodika a postup, který by byl potřeba k úspěšné realizaci těchto obvodů. Byly popsány dostupné periférie v pouzdře takto malých rozměrů a jejich možná implementace společně se zachováním dostupných pinů pro komunikaci s jiným zařízením. Dle mého názoru byla náplň práce, s ohledem na její zadání, splněna.

Reference

[1] STMicroelectronics. RM0444. STM32G0x1 advanced Arm®-based 32-bit MCUs. URL: https://www.st.com/resource/en/reference_manual/dm00371828-stm32g0x1-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

[2] STMicroelectronics. DS12992. Arm® Cortex®-M0+ 32-bit MCU, up to 64 KB Flash, 8 KB RAM, 2x USART, timers, ADC, comm. I/Fs, 1.7-3.6V. URL: https://www.st.com/resource/en/datasheet/stm32g031j6.pdf

[3] STMicroelectronics. RM0091. STM32F0x1/STM32F0x2/STM32F0x8 advanced ARM®-based 32-bit MCUs. URL: https://www.st.com/resource/en/reference_manual/dm00031936-stm32f0x1stm32f0x2stm32f0x8-advanced-armbased-32bit-mstmicroelectronics.pdf

[4] STMicroelectronics. STSW-LINK004. STM32 ST-LINK utility. URL: https://www.st.com/en/development-tools/stsw-link004.html

[5] STMicroelectronics. STM32CubeProg. STM32CubeProgrammer software for all STM32. URL: https://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-programmers/stm32cubeprog.html

[6] ARM Germany GmbH. MDK Microcontroller Development Kit. Keil uVision5. URL: https://www2.keil.com/mdk5

[7] STMicroelectronics. DS0046. PUSH-PULL FOUR CHANNEL DRIVER WITH DIODES. URL: https://www.st.com/resource/en/datasheet/l293d.pdf

[8] STMicroelectronics. DB2196. STM32 Nucleo-64 boards. URL: https://www.st.com/resource/en/data_brief/nucleo-f401re.pdf

[9] STMicroelectronics. STM32 Nucleo (64 pins) schematics. URL: https://www.st.com/resource/en/schematic_pack/nucleo_64pins_sch.zip

[10] Worldsemi. WS2812D-F5. URL: http://www.world-semi.com/DownLoadFile/135

[11] Yiu, J.: The Definitive Guide to ARM Cortex -M0 and Cortex-M0+ processors. Elsevier ScienceTechnology, 2013. 864 p. ISBN:978-0-12408-082-9.