Bakalářská práce

České vysoké učení technické v Praze

Fakulta elektrotechnická

Katedra měření

Výuka vestavných systémů s využitím mbed

Jan Kočí

Kybernetika a robotika

Květen 2020

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

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kočí** Jméno: **Jan** Osobní číslo: **466179**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Kybernetika a robotika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Výuka vestavných systémů s využitím mbed

Název bakalářské práce anglicky:

Embedded Systems Teaching Using Mbed

Pokyny pro vypracování:

Pro výuku problematiky vestavných systémů s mikrořadiči s jádrem ARM Cortex M ve verzi STM32 na FEL i na středních školách novou, netradiční metodou s využitím on – line IDE mbed vytvořte výukové materiály, demo programy, popisy a experimenty. Metoda bude spočívat v obráceném přístupu "shora", kdy student sice může rychle začít tvořit programy bez potřeby znát problematiku logických obvodů a mikrořadičů a jejich periferií, avšak prostřednictvím vhodných úloh a projektů se k těmto poznatkům postupně dostane. Postup bude od programování mikrořadiče v C++ pod mbed s využitím jeho tříd, přes programování s využitím knihoven HAL, dále přes práci s registry procesoru a výklad základu HW mikrořadiče a jeho periferií až po programování na nejnižší úrovni v jazyce assembler. Následně pak bude ukázán přestup na použití standardních nástrojů, jako je např. IDE Keil.
Použití metody dokumentujte na návrhu a realizaci vestavných systémů pro jednoduché přístroje a regulátory.

Seznam doporučené literatury:

[1] Yiu, J.: The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors,
[2] STMicroelectronics: RM0316, STM32F3 Reference manual
[3] STMicroelectronics: DS10362 - STM32F303 Data

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: **30.01.2020** Termín odevzdání bakalářské práce: **22.05.2020**

Platnost zadání bakalářské práce:
do konce letního semestru 2020/2021

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.

Datum převzetí zadání	Podpis studenta

Prohlášení

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.

V Praze dne …… ……...

Poděkování

V první řadě bych chtěl poděkovat vedoucímu této práce doc. Ing. Janu Fischerovi, CSc. za velkou trpělivost, ochotu, čas a obecně za veškerou spolupráci v rámci tvorby tohoto díla. Dále bych chtěl poděkovat rodině a přítelkyni za podporu v průběhu celého studia.

Abstrakt

Bakalářská práce se zabývá programováním mikrokontroléru STM32F303RE a slouží jako doplnění již existující bakalářské práce s názvem „Optimální využití mbed IDE v laboratorní výuce“. Cílem bakalářské práce je vytvořit výukový materiál spolu s názornými programy pro výuku vestavných systémů s využitím on-line vývojového prostředí IDE mbed. Důraz je kladen zejména na srozumitelnost a názornou demonstraci probíraných témat na příkladech, jelikož cílovou skupinu tvoří nejen studenti FEL, ale i studenti středních škol. Metoda spočívá v obráceném přístupu „shora“, kdy se student seznamuje s jednotlivými částmi mikrokontroléru postupně na vhodných příkladech. Díky tomu student nepotřebuje žádné vědomosti týkající se programování mikrokontrolérů, pouze základní znalosti z programování v programovacím jazyce C. Výsledná podoba bakalářské práce představuje srozumitelný výukový materiál pro programování mikrokontrolérů. Práce bude využita jako výukový materiál v rámci předmětů na FEL, které se zabývají programováním mikrokontrolérů. Dále v projektech zaměřených na popularizaci elektroniky na úrovni středních škol a také na kurzech praktické elektroniky organizovaných katedrou měření ČVUT FEL.

Klíčová slova: STM32; STM32F303RE; ARM Cortex M; IDE mbed; výukový materiál

Abstract

This bachelor thesis deals with the programming of the STM32F303RE microcontroller and serves as a supplement to the existing bachelor thesis entitled "The Optimal Use of mbed IDE in Laboratory Teaching". The aim of this bachelor thesis is to create study material with illustrative programs for teaching embedded systems using an online development environment IDE mbed. Emphasis is placed especially on the comprehensibility and illustrative demonstration of the topics discussed on examples, as the target group consists not only of FEE students, but also secondary school students. The method consists in the reverse approach "from above" – the student gets acquainted with individual parts of the microcontroller gradually on suitable examples. Thanks to this approach, the student does not need any knowledge related to programming microcontrollers, only basic knowledge of programming in the C programming language. The final form of the bachelor thesis is a clear teaching material for programming microcontrollers. The work will be used as a study material for FEE subjects, which deal with the programming of microcontrollers.Furthermore, in projects focused on the popularization of electronics at the level of secondary schools and also on practical electronics courses organized by the Department of Measurement, CTU FEE.

Keywords: STM32; STM32F303RE; ARM Cortex M; IDE mbed; study material

Obsah

Obrázky

Seznam nejčastěji používaných zkratek

ACR	Access control register
ADC	Analog to Digital Converter
AHB	Advanced High Performance Bus
AHBENR	AHB peripheral enable register
ALU	Arithmetic Logic Unit
APB	Advanced Peripheral Bus
BRR	Bit reset register
BSRR	Bit set/reset register
CAN	Controller Area Network
CCM RAM	Core Coupled Memory RAM
CFGR	Configuration register
CPU	Central Processing Unit
CR	Control register
DAC	Digital to Analog Converter
DRAM	Dynamic Random Access Memory
FT	5 V tolerant I/O
FTf	5 V tolerant I/O, I2C FM+ option
GND	Ground
GPIO	General Purpose Input Output
HAL	Hardware abstraction layer
HSE	High speed external
HSI	High speed internal
I/O	Input/Output
I2C	Inter-Integrated circuit
IC	Integrated circuit
IDR	Input data register
LED	Light-Emitting Diode
LEO	Little embedded oscilloscope
LL	Low layer
LR	Link register
LSE	Low speed external

LSI	Low speed internal
MISO	Master In Slave Out data
MODER	Mode register
MOSI	Master Out Slave In data
NSS	Slave select
ODR	Output data register
OSC	Oscilloscope/Oscillator (podle kontextu)
OSPEEDR	Output speed register
OTYPER	Output type register
PC	Program counter
PCB	Printed Circuit board
PLL	Phase Locked loop
PUPDR	Pull-up/pull-down register
PWM	Pulse-width Modulation
RAM	Random Access Memory
SCK	Serial Clock output
SP	Stack pointer
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SYSCLK	System clock
TS	Temperature sensor
TTa	3.3 V tolerant I/O
UART	Universal Asynchronous Receiver Transmitter.
USART	Universal Synchronus/Asynchronous Receiver Transmitter
USB	Universal Serial Bus

1 Úvod

Motivací pro tvorbu této práce byla absence výukového materiálu, který by zpřístupňoval velké množství témat týkajících se práce s mikrokontroléry řady STM32 s jádrem ARM Cortex M v dostatečně srozumitelné formě i pro studenty středních škol. Prostřednictvím výukového materiálu bych rád pomohl studentům FEL v předmětech, ve kterých se pracuje s mikrokontroléry. Současně bych chtěl pomoci popularizovat elektroniku na středních školách.

Bakalářská práce slouží jako doplnění již existující bakalářské práce s názvem „Optimální využití mbed IDE v laboratorní výuce“. Zmíněné dílo představuje základní zdroj informací a to zejména v první části této práce, ve které se student seznámí s mbed třídami. Využití mbed tříd poskytuje značnou míru abstrakce, která je velmi vhodná pro začátečníky. Na druhou stranu umožňuje používat i pokročilé metody programování, kterým jsou věnovány další části práce. Po jejich seznámení student získá určitý nadhled nad programováním mikrokontrolérů, neboť bude vědět, jak zacházet s mikrokontroléry obecně, bez závislosti na vývojovém prostředí. Programuje se v jazyce C a na závěr částečně v jazyce assembler ARM Cortex M3.

Jelikož se jedná o výukový materiál obsahující mnoho různých témat, je nezbytné pracovat s jinými zdroji, na které je často odkazováno. Po uvedení zdroje se předpokládá dostudování vybraného tématu, jelikož jednotlivé kapitoly na sebe často navazují. Pro ulehčení práce s jinými zdroji bývá vysvětleno, jak s daným zdrojem pracovat a čemu věnovat pozornost.

Jedinečnost práce spočívá v obráceném přístupu „shora“, kdy se student seznamuje s jednotlivými částmi mikrokontroléru postupně na vhodných příkladech. Vychází se proto pouze ze základních vědomostí z oblasti programování. Veškeré znalosti z elektroniky potřebné pro tuto práci student získá v rámci probíraných kapitol.

2 Rozbor a stanovení cílů práce

K programování mikrokontrolérů s jádrem ARM Cortex M ve verzi STM32 lze použít mnoho vývojových prostředí (např. STM32CubeMX, Atollic TrueSTUDIO, Keil uVision, mbed), které se liší možnostmi a samozřejmě i cenou. K používání některých prostředí je potřeba koupit licenci. Jiná jsou zdarma v plné verzi bez omezení nebo v demo verzi s různými omezeními. Kupříkladu program Keil uVision lze používat zdarma s omezením na velikost výsledného kódu, kdy po překročení 32 *kB* je nutné koupit licenci.

Mbed nabízí plnou verzi zdarma a navíc je implementovaný kombinací programovacích jazyků C/C++, což představuje jeho hlavní výhody. Spojení zmíněných dvou programovacích jazyků má pro programování mikrokontrolérů přínosné vlastnosti, protože umožňuje využít struktury, tzv. mbed třídy. Tato vlastnost je vhodná zejména pro začátečníky. Na druhou stranu mbed vyžaduje internetové připojení a neposkytuje dostatečně obsáhlou dokumentaci, kterou však může zastupovat bakalářská práce s názvem „Optimální využití mbed IDE v laboratorní výuce“ (viz [1]).

Způsob vysvětlování spočívá v obráceném přístupu „shora“, kdy se student seznamuje s jednotlivými částmi mikrokontroléru postupně na vhodných příkladech. Od opačného přístupu se očekává přijatelnější forma pro čtenáře, jelikož nové informace jsou podrobně vysvětleny (případně je odkázáno na kvalitní zdroj) a navíc ihned otestovány na příkladech. Zároveň se klade důraz na vysvětlování souvislostí.

V místech, kde je potřeba názorná ukázka kódu, bývá tento kód přiložen v modrém poli. Pokud je program rozsáhlý, jsou přiloženy a vysvětleny pouze jeho nové části. Celý program se v takovém případě nachází na konci práce v přílohách. Kromě modrých polí se v práci vyskytují také pole oranžová, ve kterých jsou uvedeny zajímavosti, doporučení, nové pojmy a vysvětlení souvislostí.

2.1 Seznámení s mikrokontroléry prostřednictvím mbed tříd

Mbed třídy umožňují používat periferie bez jejich inicializace (nainicializují se „samy“). Tato skutečnost představuje hlavní důvod, proč je vhodné začít seznamovat studenty s programováním mikrokontrolérů právě prostřednictvím zmíněných tříd. Protože tato práce má za cíl být použitelná jako samostatný pomocný výukový materiál, zahrnuje i některé citované části z práce [1], avšak jednotně v češtině.

Cílem je vytvořit velmi srozumitelnou a vhodně strukturovanou kapitolu seznamující studenta s problematikou programování mikrokontrolérů pomocí mbed tříd (konkrétně STM32F303RE), přičemž se klade důraz na názornou demonstraci na příkladech.

2.2 Náhled do vnitřní struktury mikrokontrolérů

Pro získání určitého nadhledu nad fungováním mikrokontrolérů je nezbytné obeznámit studenta s jejich vnitřní strukturou. Tyto znalosti jsou obecně platné a využívají se v jakémkoli vývojovém prostředí při programování mikrokontrolérů.

Dalším cílem proto je seznámit studenta s problematikou, která zahrnuje orientaci v jednotlivých částech mikrokontroléru a pochopení jejich funkcí. Stěžejní témata jsou: blokové schéma mikrokontroléru, adresní prostor, sběrnice, paměti, hodiny, GPIO a práce s registry. Po získání těchto znalostí by měl student vědět, jak si sám nakonfigurovat hodiny a GPIO.

2.3 Jiné způsoby programování v mbedu

Mbed poskytuje několik pokročilých nástrojů. K registrům lze v mbedu přistupovat také pomocí struktur, které zpřehledňují výsledný kód. Dále umožňuje používat HAL a LL knihovny, které představují profesionální nástroj pro programování mikrokontrolérů. Mikrokontroléry lze také programovat na nejnižší úrovni prostřednictvím funkcí psaných v programovacím jazyku ARM assembler.

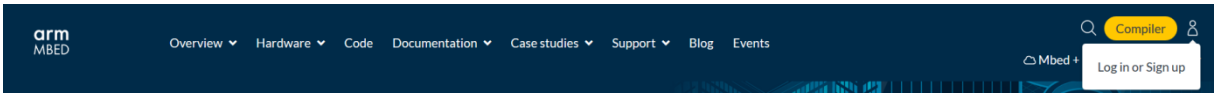
Mezi další cíle patří seznámení studenta s uvedenými pokročilými nástroji a tím ukázat jiné způsoby programování mikrokontrolérů. Posledním cílem je demonstrace výhod programování v ARM assembleru na tvorbě vstupního bloku logického analyzátoru zaznamenávajícího data.

3 Programování mikrokontrolérů s využitím mbed tříd

3.1 Úvodní seznámení s prostředím mbed

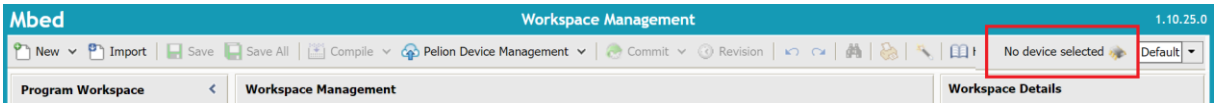
V kapitole 2 byly nastíněny důvody, proč je vhodné začít s programováním mikrokontrolérů[1] prostřednictvím vývojového prostředí mbed. V následujících deseti krocích se popisuje, jak vytvořit první program. Jedná se o program, který ještě nebudeme sami programovat. Využijeme šablony[2], která vytvoří program na blikání pomocí uživatelské LED.

1. Založit účet na http://www.mbed.com – viz *Log in or Sign up* na obr. 3.1.



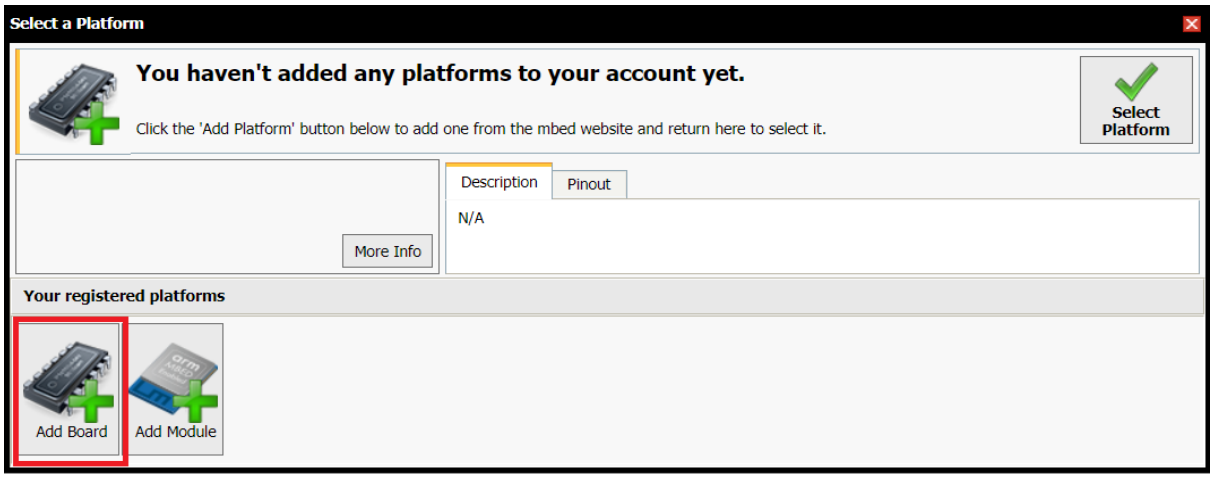
Obr. 3.1 Základní okno webu (převzato z [3])

2. Přihlásit se na vytvořený účet.
3. Otevřít *Compiler* – žluté políčko vpravo na obr. 3.1.
4. Otevřít *No device selected* – červený rámeček na obr. 3.2.



Obr. 3.2 Základní okno Compileru (převzato z [6])

5. Přidání desky pomocí *Add Board* – červený rámeček na obr. 3.1. Otevře se okno s podporovanými mikrokontroléry.

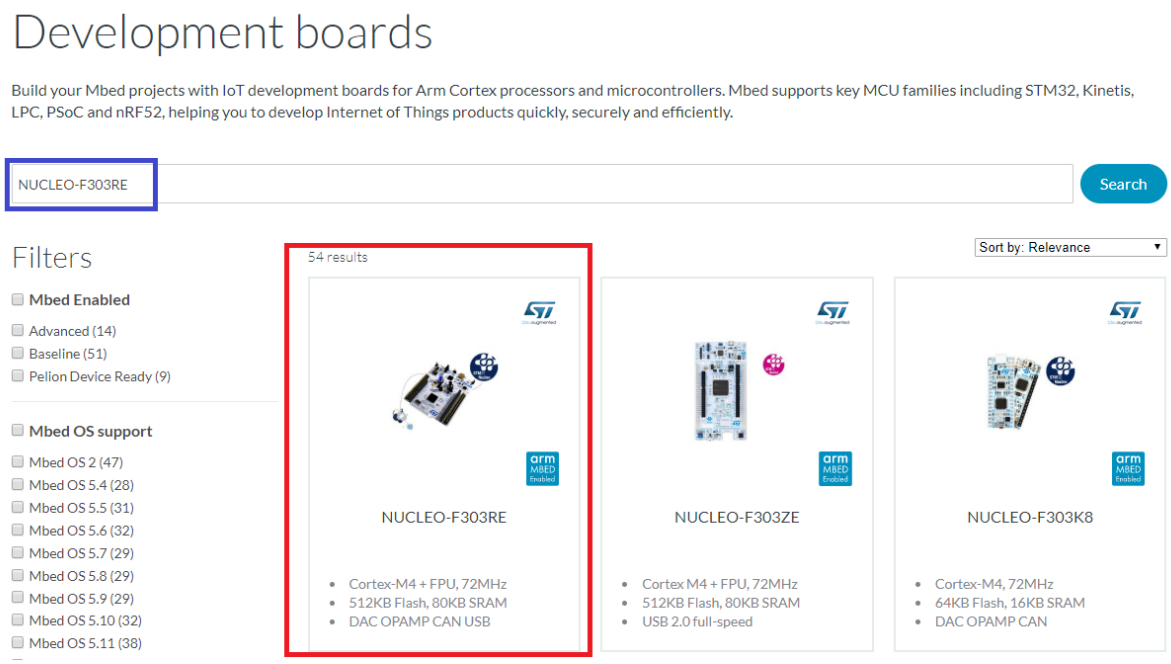


Obr. 3.3 Přidání desky (převzato z [6])

[1] Mikrokontrolér (angl. Microcontroller Unit = MCU), jednočipový počítač, deska či Nucleo se dále používá pro označení kitu **NUCLEO-F303RE**, na kterém se programování mikrokontrolérů demonstruje. Tento kit obsahuje **mikrokontrolér STM32F303RE**, což je tedy pouze jedna součástka na tomto kitu (viz obr. 5.11). Dále se však tento rozdíl zakrývá a označením mikrokontrolér se myslí celý kit NUCLEO-F303RE.

[2] Šablona je již vytvořený program, který mbed nabízí za účelem seznámení s danou problematikou.

6. Vyhledání desky s využitím vyhledávácího pole – modrý rámeček na obr. 3.4. Nucleo, které se bude používat, je zvýrazněno v červeném rámečku.

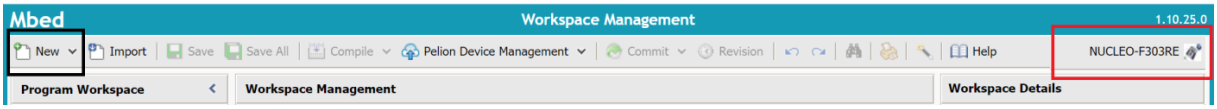


Obr. 3.4 Přehled podporovaných desek (převzato z [7])

7. Přidání zařízení do **Mbed Compileru** (dále jen Compiler) pomocí tlačítka *Add to your Mbed Compiler* (červený rámeček vpravo dole na obr. 3.5).

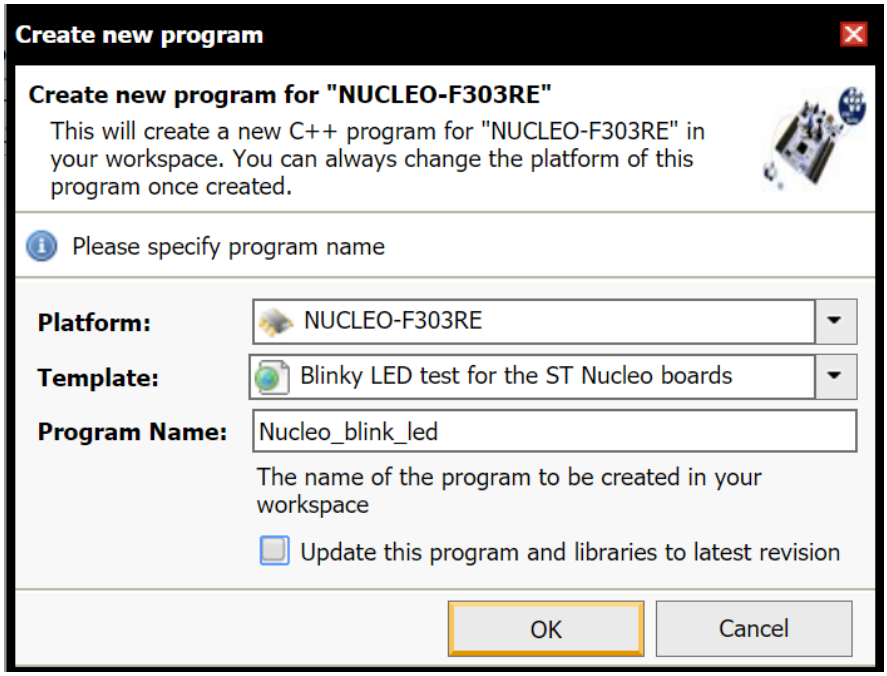
Obr. 3.5 Informace o desce NUCLEO-F303RE (převzato z [8])

8. Kontrola, zda se Nucleo skutečně přidalo do **Compileru**. Místo *No device selected* (viz obr. 3.2) se musí v **Compileru** objevit přidaná deska (viz obr. 3.6).



Obr. 3.6 Přidaná deska v Compileru (převzato z [6])

9. Vytvoření nového projektu pomocí *New* – černý rámeček vlevo na obr. 3.6.
10. Použitá platforma je přidaný mikrokontrolér z minulých dvou kroků. Jako šablonu (angl. template) zvolíme „Blinky LED test for the ST Nucleo boards“. Jméno programu se může nechat stejné, avšak pole „Update this program and libraries to latest revision“ je nutné nechat **nezaškrtnuté**, protože třídy, které si budeme ukazovat, by nemusely být kompatibilní s poslední verzí [1]. Výsledné okno je na obr. 3.7.

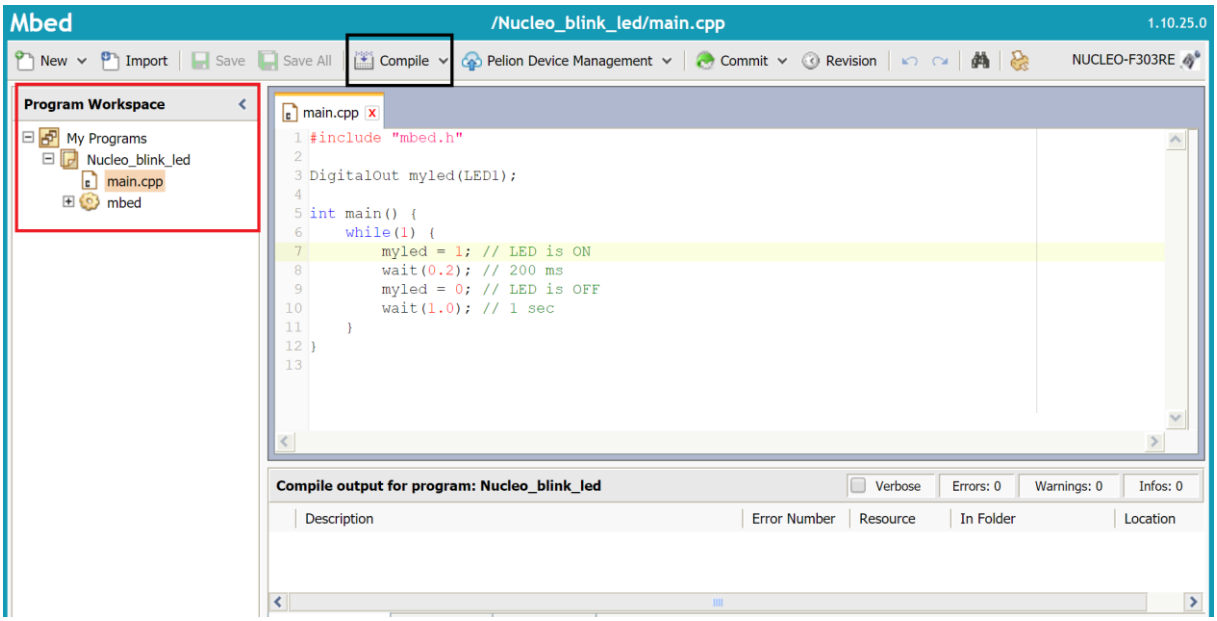


Obr. 3.7 Vytvoření prvního programu (převzato z [6])

První program je hotov. V levé části **Compileru** se nachází „Program Workspace“ (viz červený rámeček na obr. 3.8), jenž obsahuje složku „My Programs“, která sdružuje všechny námi vytvořené programy. Po provedení všech výše zmíněných kroků se v této složce nachází pouze program „Nucleo_blink_led“. Tento program obsahuje *main.cpp* a mbed knihovnu, jejíž dokumentace není připravena. Jak bylo uvedeno v kapitole 2, místo dokumentace využijeme práci [1], která obsahuje informace o většině definovaných tříd.

Na hlavní liště se mmj. nachází *Compile* (viz černý rámeček na obr. 3.8). Pokud se *main.cpp* takto zkompiluje[3], **Compiler** z něj vytvoří binární soubor. Pomocí kabelu připojíme desku s PC. V PC se po připojení objeví složka, do které vložíme vygenerovaný binární soubor a tím ho do mikrokontroléru nahrajeme.

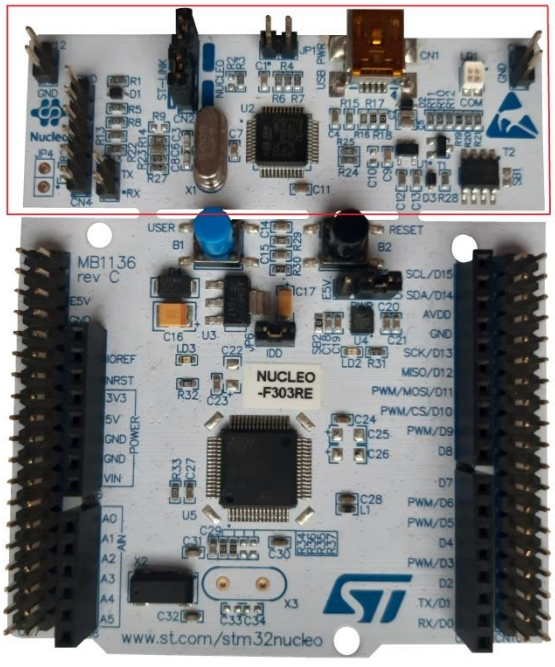
[3] Kompilace programu je proces, při němž kompilátor převede program/zdrojový kód do binární podoby, tedy do sekvencí jedniček a nul. Mikroprocesor umí vykonávat pouze takto převedený program.



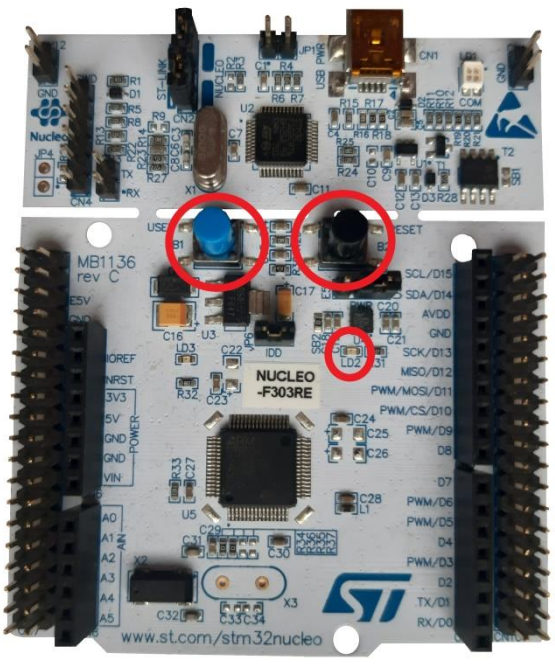
Obr. 3.8 Vytvořený první program (převzato z [6])

3.2 Vytvoření prvních programů

Jak již bylo v úvodních kapitolách nastíněno, budeme se zabývat programováním mikrokontroléru **STM32F303RE**, který je na obr. 3.9. Deska se skládá ze dvou částí. V červeném rámečku je zvýrazněn blok **ST-Link**, což je zařízení sloužící k nahrávání programu (vygenerovaného binárního souboru) z PC do mikrokontroléru, tedy do té „zbývající“ části. Jednotlivé části desky budou postupně vysvětlovány.



Obr. 3.9 Vývojový kit NUCLEO-F303RE



Obr. 3.10 Kit s označenými tlačítky a LED

Začneme s nejzákladnějšími uživatelskými komponentami (viz červeně zakroužkované části na obr. 3.10). Jedná se o uživatelské tlačítko (modré), resetovací tlačítko (černé) a uživatelskou LED[4]. Resetovací tlačítko slouží k resetování programu. Stlačíme-li ho tedy v průběhu chodu programu, spustí se od začátku.

3.2.1 Uživatelská LED

Následuje vysvětlení činnosti prvního programu *main.cpp* z kap. 3.2:

```
#include "mbed.h"
DigitalOut myled(LED1);
int main() {
   while(1) {
      myled = 1; // LED is ON
      wait(0.2); // 200 ms
      myled = 0; // LED is OFF
      wait(1.0); // 1 sec
   }
}
```

Na začátku každého programu je potřeba importovat knihovnu *mbed.h*. Tato knihovna obsahuje třídy, které při programování mikrokontrolérů značně ulehčují práci, protože před používáním dané periferie[5] samy zajistí její konfiguraci.

Konfigurace je obecně proces, při kterém se nastavuje pin nebo periferie. Konkrétně v řešeném programu se procesoru „říká“, že se použije dioda. Mbed třída *DigitalOut* zajistí potřebné nastavení. Blikání diody je podstata prvního programu. Blikání LED je proces, při kterém procesor zhasíná a rozsvěcí diodu. Jedná se proto o výstup (angl. output) mikrokontroléru. Za tímto účelem je v kódu druhý řádek:

```
DigitalOut myled(LED1);
```

Tento řádek vytvoří **objekt** *myled* (pojmenování je při použití klasických znaků v podstatě volitelné), který nakonfiguruje diodu (pro zvolení diody se píše *LED1*) jako *DigitalOut*. Za konstantou *LED1* se ve skutečnosti skrývá označení pinu, na kterém je dioda připojena. Problematikou značení pinů se zabývá kap. 3.3. *DigitalOut* se označuje **název třídy**. Tímto způsobem se procesoru „řekne“, co chceme používat (*LED1*) a na co to budeme používat (jako digitální výstup, tedy *DigitalOut*). Digitální proto, že LED je buď vypnutá, nebo zapnutá (více v kap. 3.5.1). Toto vysvětlení prozatím postačí.

Vytvoření objektu má obecný tvar (dle kap. 4 v [1]) :

```
název_třídy název_objektu (vstupní parametry);
```

[4] LED je zkratka z angl. Light-Emitting Diode, je tedy nesmyslné psát „LED dioda“. Nucleo má tři LED, avšak pouze LED2 (na desce označená jako LD2) je uživatelská. V textu bude označována jako LED, led, případně dioda, i když je dioda širší pojem. V případě nejasností ohledně LED si stáhněte volně dostupnou knihu [28] (u citace je uveden i odkaz na stažení) určenou nejen pro začátečníky v „bastlení“.

[5] Periferie je rozhraní, pomocí něhož umí mikrokontrolér komunikovat s „vnějším světem“, např. uživatelské tlačítko a uživatelská dioda patří mezi periferie.

Z obecného tvaru je patrné, že k vytvoření objektu bývají (ne vždy tomu tak je) potřeba vstupní parametry. Vstupní parametry většinou určují, jaký pin se bude používat, případně nastavují/specifikují další parametry. Názvem třídy se určuje, za jakým účelem se bude objekt používat.

Program se tradičně nachází ve funkci *main()*. Následuje nekonečná smyčka, ve které se vykonávají čtyři operace: rozsvícení diody, čekání 200 *ms*, zhasínání diody a čekání 1 *s*. Využití nekonečné smyčky je při programování mikrokontrolérů velmi časté, jelikož program tak neustále běží tzn. nikdy neskončí.

Při programování někdy bývá potřeba vložit prodlevu (čekat určitou dobu). Pro tento účel lze využít funkce *wait()*, *wait_ms()* a *wait_us()*. Zmíněné funkce se liší pouze v jednotkách, ve kterých se doba specifikuje. Výklad k těmto funkcím je v kap. 4.1 v [1]. Zmíněná kapitola se zabývá obecně třídou *Wait*. Její přečtení je doporučeno zejména začátečníkům.

Samotné rozsvícení/zhasnutí diody se provádí jednoduše zapsáním 1/0 do objektu *myled* (viz vytvořený program).

V této kapitole bylo vysvětleno, jak funguje a „co dělá“ vytvořený program z předešlé kapitoly. Naučili jsme se, co je to třída a objekt a co znamená pojem konfigurace. Konkrétně došlo k seznámení a využití tříd *Wait* a *DigitalOut*, na základě kterých jsme schopni zhasínat a rozsvěcet diodu, případně vložit dobu čekání do programu. Následuje seznámení s další části Nuclea, konkrétně s uživatelským tlačítkem.

3.2.2 Uživatelské tlačítko

Prostřednictvím diody procesor „dává“ zprávy vnějšímu světu (rozsvícením/zhasínáním), proto je dioda výstup. Nyní je naopak žádoucí opačný proces, tedy předání zprávy z okolí do procesoru, proto se jedná o vstup (angl. input) procesoru. Tlačítko může být stlačené či nikoli, jedná se proto opět o informaci s dvěma stavy, jinak řečeno o digitální vstup.

Za tímto účelem je naimplementovaná třída *DigitalIn*. Objekt této třídy se vytvoří takto:

```
DigitalIn mybutton(USER_BUTTON);
```

Za konstantou *USER_BUTTON* se opět skrývá označení pinu, na kterém je tlačítko připojeno. U diody se do objektu zapisovalo, u tlačítka se naopak z objektu čte. Když je objekt *mybutton* roven 0, pak je tlačítko stlačené. Poznamenejme, že to je **naopak** než bychom nejspíše čekali.

K napsání vlastní programu je nejdříve potřeba jeho vytvoření. Nový program lze vytvořit stejným postupem jako v kap. 3.1, avšak se začátkem od kroku 9. Nový program vhodně pojmenujte (např. UserButton). Nový program lze také vytvořit prostřednictvím klonování. Po kliknutí pravým tlačítkem na jméno již vytvořeného programu se objeví okno s možností *Clone*. Tímto způsobem se vytvoří nový stejný program, který je vhodné přejmenovat.

Vytvořme program, ve kterém se stlačením tlačítka rozsvítí dioda. Do vytvořeného programu (viz minulý odstavec) se přidá vytvoření objektu *mybutton* (viz výše). V nekonečné smyčce bude program, který při stlačeném tlačítku rozsvítí diodu. Zdůrazněme, že před nahlédnutím na řešení, byste si měli sami zkusit program naprogramovat. Vytvoření vlastního řešení přinese mnohem více, než pouhé zkoumání nalezených řešení.

Výsledný program:

```
#include "mbed.h"
DigitalOut myled(LED1);
DigitalIn mybutton(USER_BUTTON);
int main() {
   while (1) {
      if (mybutton == 0){ // The user button is pressed
         myled = 1; // LED on
      } else { // The user button is released
         myled = 0; // LED off
      }
   }
}
```

Vytvořený program demonstruje základní použití uživatelského tlačítka. Dále následuje připomenutí třídy *DigitalOut* a bližší seznámení s třídou *DigitalIn*. Ještě před tím je však nutné seznámit se s piny.

Blikání na začátku programu

Pro lepší představu o tom, co se v desce právě odehrává, je vhodné přidat několik zablikání diody na začátku programu. Uživatel tak získá informaci o tom, že program právě začal. Obecně se několika zablikáním dá signalizovat dosažení daného bodu programu. V kap. 3.7.1 je ukázán způsob, jak na začátku programu nejen blikat, ale samotný program také pozastavit.

3.3 Vstupně výstupní piny

V kap. 3.2.1 a 3.2.2 bylo řečeno, že za konstantami *LED1* a *USER_BUTTON* se ve skutečnosti skrývá označení pinu, na kterém se daná periferie nachází. Co to vlastně piny jsou a jak s nimi pracovat?

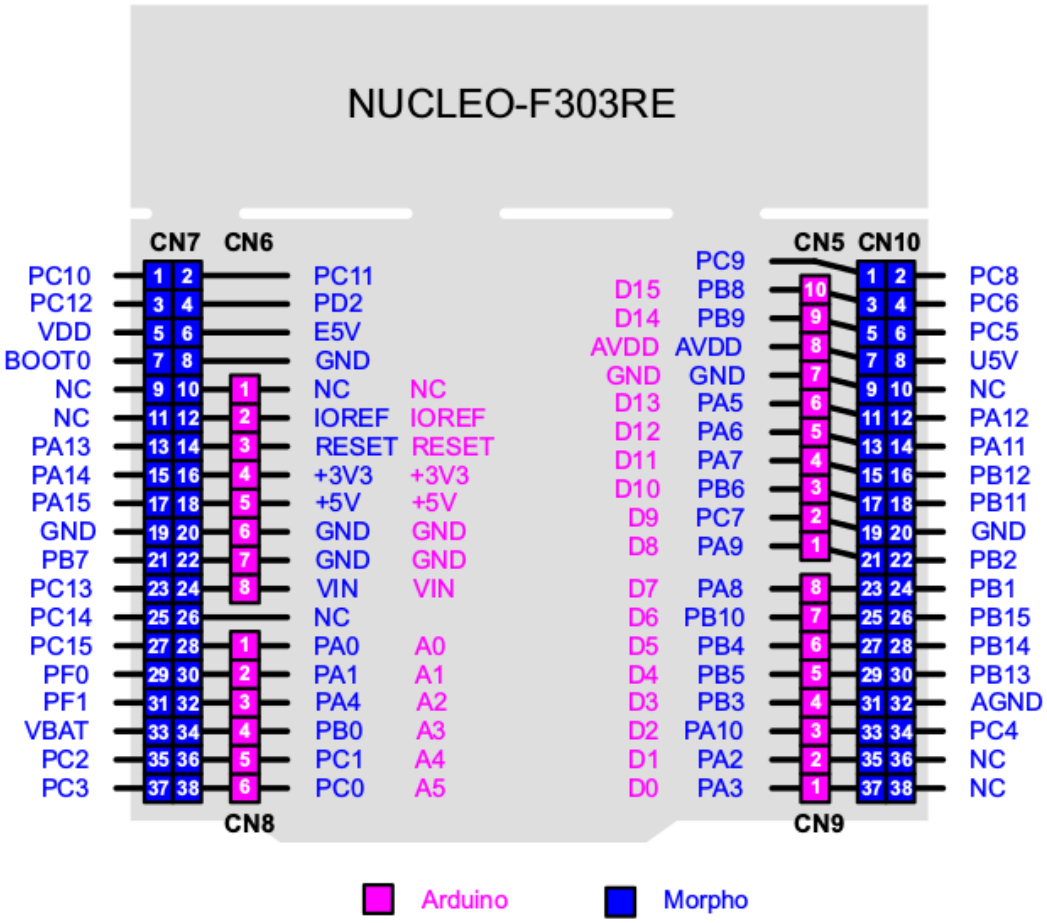
Pin obecně označuje vývod elektronické součástky. V podstatě se jedná o vodivé kontakty, které vedou do mikrokontroléru. Na Nucleu se piny rozdělují do dvou skupin: Arduino a Morpho konektory (viz obr. 3.11). Na piny jsou vnitřně „nataktovány“ periferie, proto se veškerá práce s periferiemi odehrává prostřednictvím pinů. Doposud jsme se setkali s využitím pinů jako digitálního vstupu a digitálního výstupu. Na desce se také nachází piny, které nelze využít pro práci s periferiemi. Kupříkladu se na Nucleo nacházejí piny jako je zem (angl. GND), zdroj napětí 3,3 *V* a 5 *V* (na desce značeny +3V3 a +5V), nepřipojené piny se značkou NC (angl. not connected), které nemají žádnou funkcionalitu a další.

Součástí práce [1] bylo také upravit pinout diagramy mikrokontroléru NUCLEO-F303RE, protože, jak je v úvodu kap. 3 v [1] uvedeno, ve volně dostupných diagramech bývají chyby. Zmíněný úvod si přečtěte.

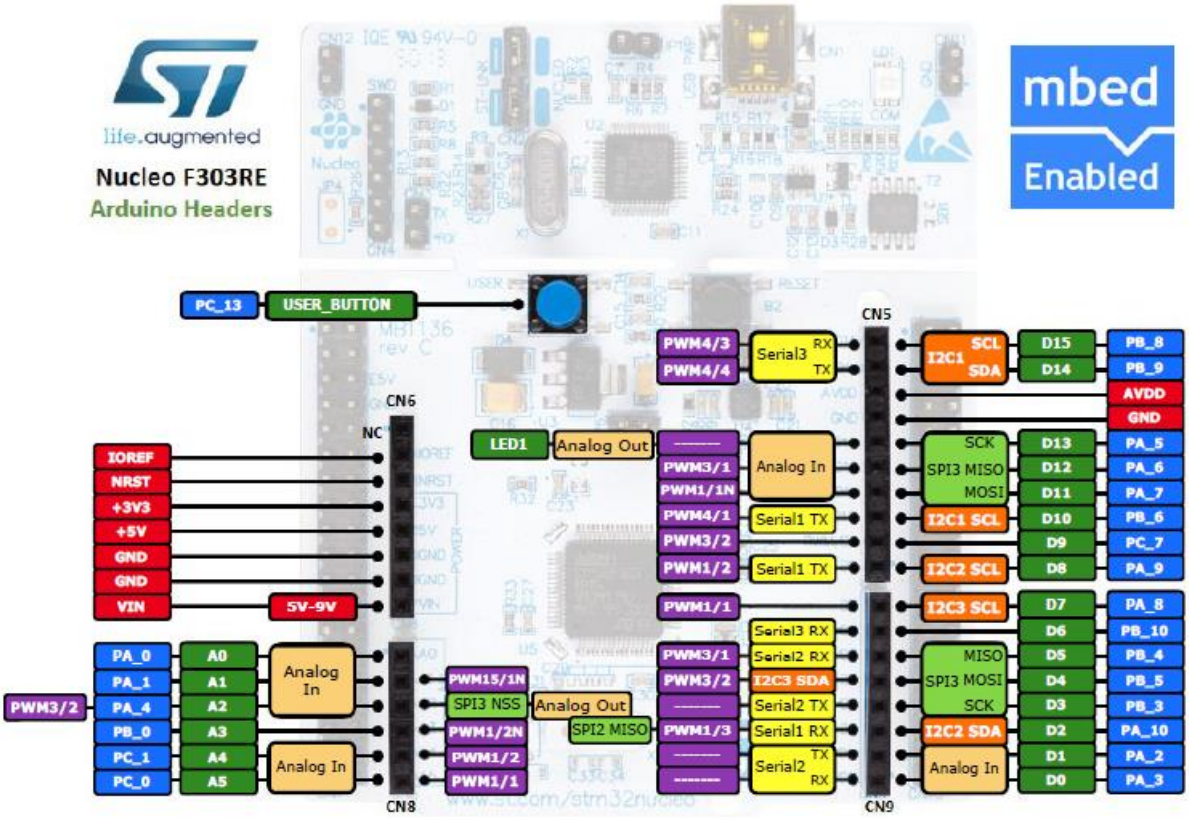
Upravený diagram najdeme v téže práci v kap. 3.2. Pro naše potřeby postačí obr. 3.11, avšak pro samostatnou činnost a pozdější část práce je nutné myslet na to, že každý pin nejde

konfigurovat pro každou periferii. Za tímto účelem jsou přidány i upravené pinout diagramy z [1] (viz obr. 3.12, obr. 3.13 a obr. 3.14). Poznamenejme, že Arduino piny jsou skutečně propojené s Morpho piny, tedy např. +5 *V* je na CN6 na pinu číslo 5, ale také na CN7 na pinu číslo 18.

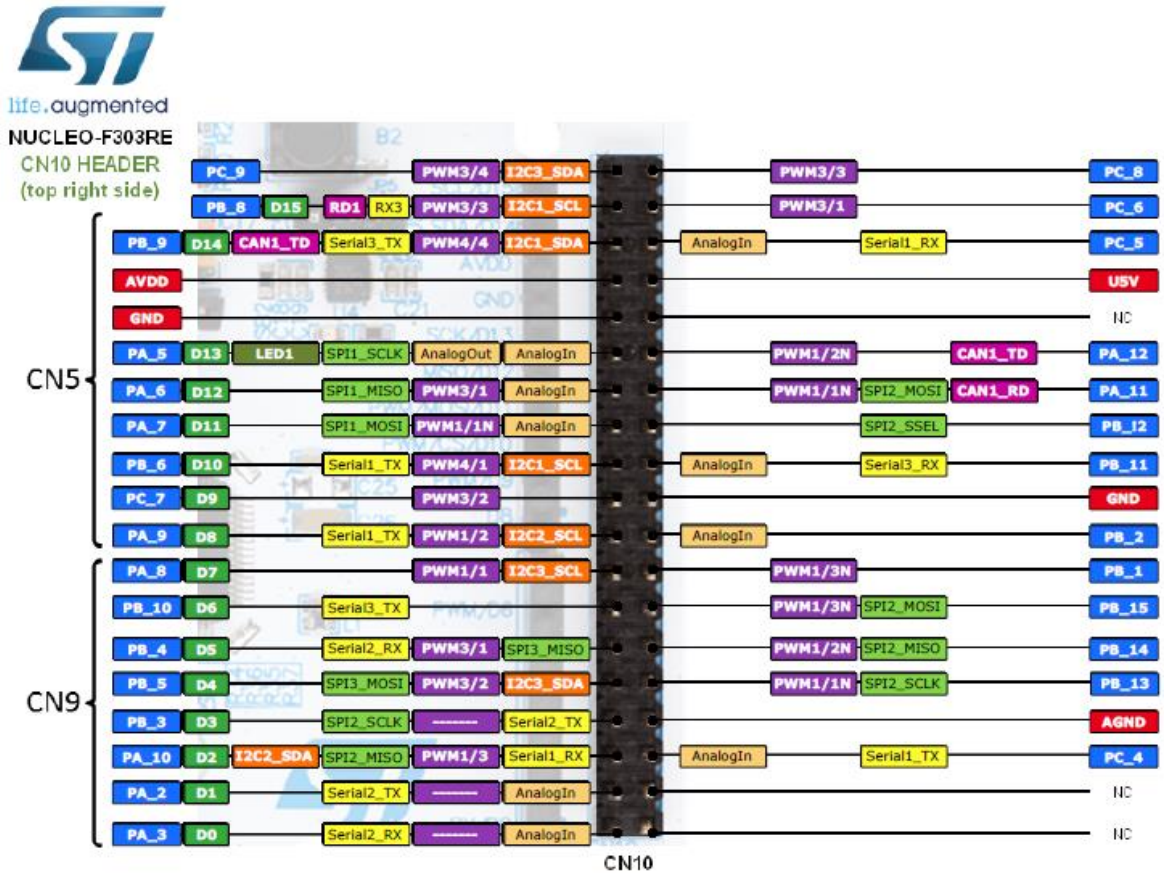
Uživatelské tlačítko ani uživatelská dioda na obr. 3.11 nejsou. Tlačítko je připojeno k pinu PC13 a dioda k pinu PA5. Písmeno P je zkratka slova pin a písmeno za ním specifikuje GPIO bránu, o tom však později. V mbedu se v označení pinu odděluje číslo od zbytku podtržítkem, proto se pin tlačítka značí PC_13 a pin diody PA_5. Je však možno použít i zmíněné konstanty USERBUTTON a LED1. Zkuste si v již vytvořených programech tyto konstanty nahradit.



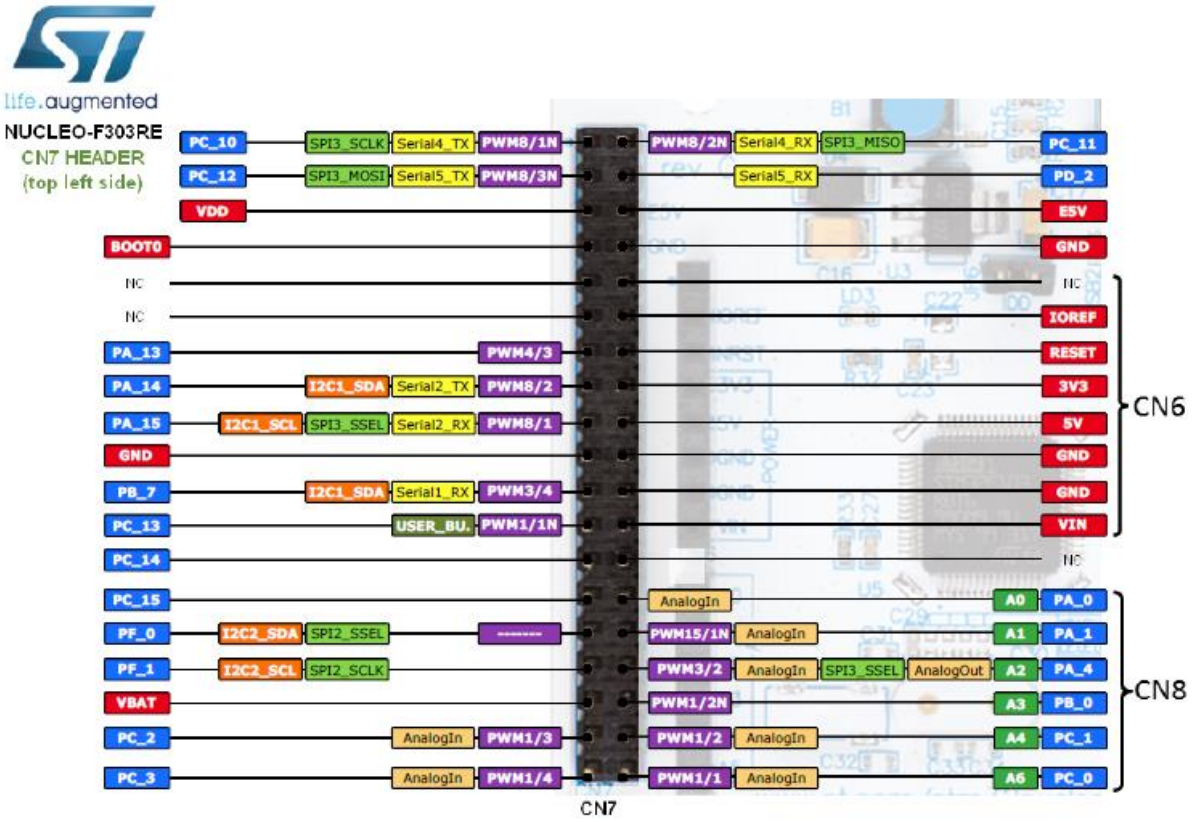
Obr. 3.11 Pinout diagram (upravený obr. z [3])



Obr. 3.12 Pinout střední části STM32F303RE (převzatý obrázek 3.3 z [1])



Obr. 3.13 Pinout pravé části STM32F303RE (převzatý obrázek 3.4 z [1])



Obr. 3.14 Pinout levé části STM32F303RE (převzatý obrázek 3.5 z [1])

3.4 Třída DigitalOut

Tato kapitola se zabývá třídou *DigitalOut*, o které pojednává kap. 4.2 v [1]. Po přečtení zmíněné kapitoly jsme schopni:

- nastavit mód pinu,
- nastavit výstupní úroveň pinu po jeho vytvoření (tzn. nastavit výstup ihned po inicializaci na logickou 1/0),
- velmi jednoduše invertovat logickou hodnotu (viz následující příklad).

Poznamenejme, že pod tabulkou 13 v [10], jejíž část se nachází na obr. 3.20, je uvedena důležitá informace týkající se použití pinů **PC13**, **PC14** a **PC15**. Tyto piny mají jiný druh napájení než ostatní, proto při jejich použití jako výstupních pinů je omezení na max. velikost proudu, které může externí obvod z pinu čerpat, konkrétně 3 *mA*. Za tímto účelem **nepoužívejme** tyto piny **jako výstupní**. Na desce je mnoho jiných pinů, které mohou být nastaveny jako výstupní bez omezení.

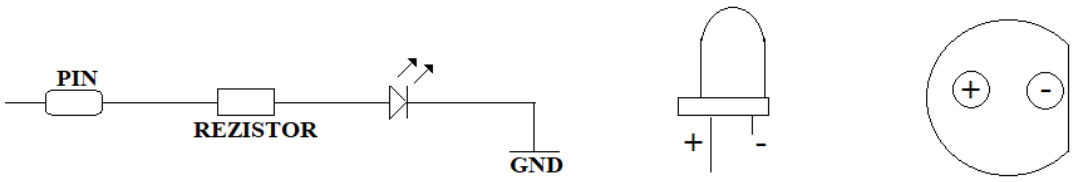
Je na místě vyzkoušet získané poznatky v praxi. Za tímto účelem byl vytvořen následující program, jenž ihned po vytvoření objektu třídy *DigitalOut* nastaví výstup do logické 1 (dioda svítí) a po stlačení tlačítka se změní její stav (tzn. zhasne/rozsvítí). Nekonečná smyčka řeší situaci, kdy je tlačítko stisknuté, tedy ještě není uvolněno:

```
#include "mbed.h"
#define PIN_LED PA_5
#define PIN_BUTTON PC_13
DigitalOut led(PIN_LED, 1);
DigitalIn button(PIN_BUTTON);
int main(){
   while(1){
      if (!button){
         while (!button){} //is the button still pressed?
         led = !led;
      }
   }
}
```

3.4.1 Zapojení externí LED

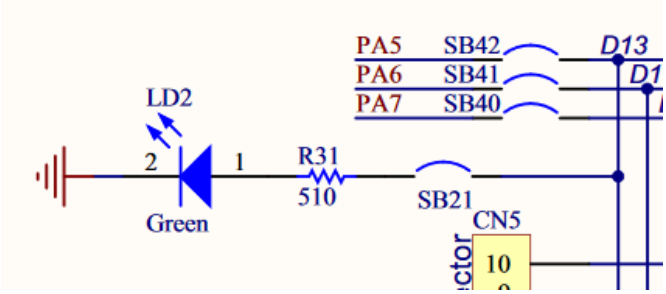
První externí obvod, který si zapojíme, je s diodou. Zopakujme, že pojem „dioda“ ve skutečnosti představuje širší pojem než „LED“, avšak přesto je v této práce používán jako synonymum. Dioda má dva vývody. Delšímu vývodu se říká anoda, kratšímu katoda. Katoda se připojuje na zem, anoda přes rezistor k napájení. Rezistor omezuje proud obvodem, jeho velikost zvolíme stovky Ω, max. pár desítek *k*Ω (záleží na samotné LED), klasicky 480 Ω. Zopakujme, že je nesmyslné psát „LED dioda“, protože písmeno **D** ve zkratce přestavuje slovo **Diode**, tedy dioda (viz pozn. pod čarou 4).

Napájení obvodu může zastupovat i pin nakonfigurovaný jako výstupní, viz obr. 3.15. Na tomto obrázku je vidět externí zapojení, samotná dioda s vyznačenou katodou a anodou a spodní pohled na diodu. Pohled zespoda ukazuje, že katoda je u rovné strany. Jedná se o způsob, jak rozpoznat vývody, pokud došlo k jejich zkrácení na stejnou délku. V případě nejasností spojených s LED se obraťte na [28].



Obr. 3.15 Zapojení LED

Stejným způsobem je zapojená i interní LED (viz obr. 3.16): GND, LED, R31 a pin PA5.



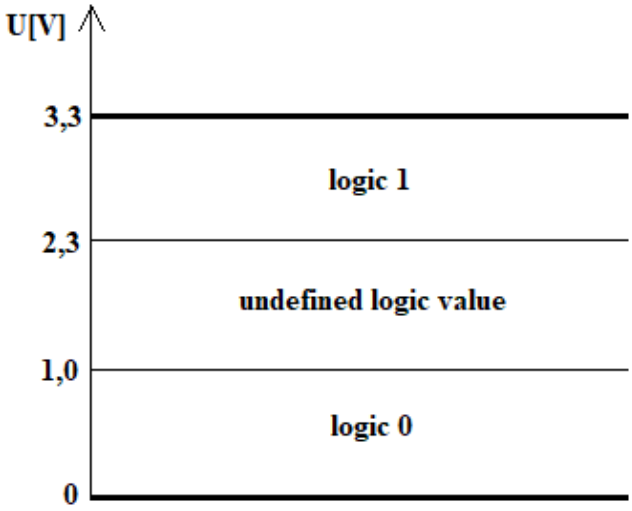
Obr. 3.16 Zapojení interní diody (převzato z [4])

3.5 Třída DigitalIn

Přehled základních funkcí třídy *DigitalIn* se nachází v kap. 4.3 v [1]. Opět se jedná o třídu velmi zásadní, proto je vhodné si ji pozorně přečíst. V úvodu výše zmíněné kapitoly je řečeno, že každá logická hodnota je reprezentována elektrickým napětím (více v kap. 3.5.1). Logické hodnoty si i experimentálně ověříme, avšak setkáme se s „podivným“ chováním, které nás přivede k programovatelným pull rezistorům (viz kap. 3.5.4). Na závěr se blíže seznámíme s interním a nově i externím tlačítkem a jevem zvaným bouncing (viz kap. 3.5.5).

3.5.1 Logické úrovně

Pokud zapojíme pin jako *DigitalIn* a přiložíme k němu napětí 0 *V*, bude na pinu logická (dále jen log.) 0. Pokud naopak přiložíme napětí 3,3 *V*, bude na pinu logická 1. Co když ale na vstup připojíme napětí mezi těmito dvěma mezními hodnotami? Jestliže je napájecí napětí 3,3 *V* a na pinu čteme log. 1, pak je na pin přiloženo napětí v intervalu 2,3 *V* až 3,3 *V*. Log. 0 čteme v případě, kdy je přiložené napětí v rozsahu 0 *V* až 1 *V*. Přiložíme-li napětí mezi 1 *V* a 2,3 *V*, pak je na pinu nedefinovaná logická hodnota, viz obr. 3.17.



Obr. 3.17 Vstupní napětí log. úrovní

Právě získané poznatky si experimentálně vyzkoušíme. Vytvořte program, ve kterém budete číst libovolný vstupní pin. Pokud bude na tomto pinu log. 1/0, pak rozsvítíte/zhasnete led a počkáte daný čas (např. na 50 *ms*). Diodu v daném stavu ponechte. K experimentu je potřeba rezistor a kabel. Pokud si nejste jisti vytvořeným programem, najdete ho v příloze 12.1. Po vytvoření programu je vhodné jeho otestování (viz následující odstavec).

Přes rezistor o velikosti několika set Ω až několika desítek *k*Ω připojíme pin s +3V3 (hledejte na CN6 na obr. 3.11) k libovolnému vstupnímu pinu (např. PC0), dioda pak bude svítit. Připojíme-li pin PC0 s GND (se zemí), pak led svítit nebude. Pomocí napěťového děliče bychom mohli otestovat reakci vytvořeného programu pro vstupní napětí v intervalu 1 *V* až 2,3 *V*. Zdůrazněme, že kvůli ochraně bychom měli vždy připojovat vstupní pin přes **rezistor** (viz kap. 3.5.3). Nedělá řešení přesně to, co jsme chtěli? Příčinu si vysvětlíme dále.

3.5.2 Stanovení velikosti přípustného napětí na připojeném pinu

Jelikož by čtenář měl získat i náhled na to, jak pracovat s manuály, je důležité ukázat, jak takovou informaci dohledat. Vzhledem k nově nabitým informacím ze začátku

této kapitoly se nabízí důležitá otázka, jak velké napětí se vlastně může na pin připojit? Na obr. 3.17 jsou vyznačeny hodnoty 0 *V* a 3,3 *V*, jsou to ale skutečně krajní hodnoty?

Odpověď na tyto otázky se nachází v manuálu daného mikrokontroléru (pro nás v [10]). Pomocí hesla „input voltage“ najdeme tabulku napěťové charakteristiky (viz obr. 3.18). Hledaná proměnná je V_{IN}, ta má ovšem definované min. a max. pro 4 různé skupiny pinů. Pouze pin Boot0 má minimální hodnotu napětí 0 *V*. Na ostatní piny je možno připojit napětí o minimální velikosti (V_{SS}-0,3) *V*, kde $\mathbf{V_{SS}}$ představuje zem (**0 *V***). Maximální hodnota napětí pro pin BooT0 je 9 *V*, pro FT a FTf piny (V_{DD}+4,0) *V* a pro zbylé piny 4 *V*, kde $\mathbf{V_{DD}}$ označuje napájecí napětí (**3,3 *V***). Co ale tyto zkratky znamenají a jak poznat, do jaké skupiny konkrétní pin patří?

Symbol	Ratings	Min	Max	Unit
V_{DD}–V_{SS}	External main supply voltage (including V_{DDA}, V_{BAT} and V_{DD})	-0.3	4.0	V
V_{DD}–V_{DDA}	Allowed voltage difference for $V_{DD} > V_{DDA}$	-	0.4	
V_{REF+}–V_{DDA} (2)	Allowed voltage difference for $V_{REF+} > V_{DDA}$	-	0.4	
V_{IN} (3)	Input voltage on FT and FTf pins	V_{SS} –0.3	V_{DD} + 4.0	V
	Input voltage on TTa pins	V_{SS} –0.3	4.0	
	Input voltage on any other pin	V_{SS} –0.3	4.0	
	Input voltage on Boot0 pin	0	9	
$\lvert\Delta V_{DDx}\rvert$	Variations between different V_{DD} power pins	-	50	mV
$\lvert V_{SSX} - V_{SS}\rvert$	Variations between all the different ground pins	-	50	
$V_{ESD(HBM)}$	Electrostatic discharge voltage (human body model)	see *Section 6.3.13: Electrical sensitivity characteristics*		-

1. All main power (V_{DD}, V_{DDA}) and ground (V_{SS}, V_{SSA}) pins must always be connected to the external power supply, in the permitted range. The following relationship must be respected between V_{DDA} and V_{DD}:
V_{DDA} must power on before or at the same time as V_{DD} in the power up sequence.
V_{DDA} must be greater than or equal to V_{DD}.
2. V_{REF+} must be always lower or equal than V_{DDA} ($V_{REF+} \leq V_{DDA}$). If unused then it must be connected to V_{DDA}.
3. V_{IN} maximum must always be respected. Refer to *Table 17: Current characteristics* for the maximum allowed injected current values.

Obr. 3.18 Napěťová charakteristika (převzatá tabulka 16 v [10])

K odpovědi na tyto otázky je potřeba dohledat další tabulky. Pro zjištění významů zkratek tabulku s legendou (viz obr. 3.19). Parametry FT a FTf tedy označují 5 *V* tolerantní piny I/O. Zmíněná tolerance je pouze za podmínky připojeného napájení. Do jaké skupiny „I/O structure“ konkrétní pin patří, se zjistí v tabulce 13 v [10]. Část zmíněné tabulky je na obr. 3.20. Poznamenjme, že přehled základních informací o Nucleu je uvedena v tabulce 2 v [10].

Na obr. 3.21 je tabulku shrnující obecné provozní informace. V této tabulce je opět V_{IN}, avšak s jinými hodnotami max. napětí. U pinu Boot0 je rozdíl max. napětí 4,5 *V*. Tento pin však používat nebudeme, proto rozdílu nevěnujme pozornost. Rozdíl v max. napětí u FT a FTf pinů je 3,7 *V*, avšak ten vysvětluje poznámka 1 pod tabulku, ze které plyne, že na daný pin lze připojit napětí v rozmezí (V_{DD}+0,3) *V* až (V_{DD}+4,0) *V* pouze pokud se nepoužije vnitřní pull-up/pull-down rezistor.

Výsledkem této kapitoly je skutečnost, že při čtení manuálů je potřeba obezřetnosti, jelikož tabulky často obsahují výjimky. Navíc samotné informace z jedné tabulky se nemusí shodovat s informacemi v tabulce druhé. Ohledně velikosti vstupního napětí je vhodné provést jedno shrnující obecně platné pravidlo: na piny připojovat pouze **nezáporné** napětí do **max.** velikosti **3,3 *V***.

Name		Abbreviation	Definition
Pin name		Unless otherwise specified in brackets below the pin name, the pin function during and after reset is the same as the actual pin name	
I/O structure		FT	5 V tolerant I/O
		FTf	5 V tolerant I/O, I^2C FM+ option
		TTa	3.3 V tolerant I/O
		TC	Standard 3.3V I/O
		B	Dedicated to BOOT0 pin
		RST	Bi-directional reset pin with embedded weak pull-up resistor
Notes		Unless otherwise specified by a note, all I/Os are set as floating inputs during and after reset	
Pin functions	Alternate functions	Functions selected through GPIOx_AFR registers	
	Additional functions	Functions directly selected/enabled through peripheral registers	

Obr. 3.19 Legenda k pinům (převzatá tabulka 12 v [10])

Pin number					Pin name (function after reset)	Pin type	I/O structure	Notes	Alternate functions	Additional functions
LQFP64	LQFP100	UFBGA100	WLCSP100	LQFP144						
37	63	E12	F4	96	PC6	I/O	FT	-	EVENTOUT, TIM3_CH1, TIM8_CH1, I2S2_MCK, COMP6_OUT	-
38	64	E11	F2	97	PC7	I/O	FT	-	EVENTOUT, TIM3_CH2, TIM8_CH2, I2S3_MCK, COMP5_OUT	-
39	65	E10	F1	98	PC8	I/O	FT	-	EVENTOUT, TIM3_CH3, TIM8_CH3, COMP3_OUT	-
40	66	D12	F3	99	PC9	I/O	FTf	-	EVENTOUT, TIM3_CH4, I2C3_SDA, TIM8_CH4, I2SCKIN, TIM8_BKIN2	-

Obr. 3.20 Definice pinů (převzatá část tabulky 13 v [10])

Symbol	Parameter	Conditions	Min	Max	Unit
f_{HCLK}	Internal AHB clock frequency	-	0	72	MHz
f_{PCLK1}	Internal APB1 clock frequency	-	0	36	
f_{PCLK2}	Internal APB2 clock frequency	-	0	72	
V_{DD}	Standard operating voltage	-	2	3.6	V
V_{DDA}	Analog operating voltage (OPAMP and DAC not used)	Must have a potential equal to or higher than V_{DD}	2	3.6	V
	Analog operating voltage (OPAMP and DAC used)		2.4	3.6	
V_{BAT}	Backup operating voltage	-	1.65	3.6	V
V_{IN}	I/O input voltage	TC I/O	-0.3	V_{DD}+0.3	V
		TTa I/O	-0.3	V_{DDA}+0.3	
		FT and FTf I/O(1)	-0.3	5.5	
		BOOT0	0	5.5	
P_D	Power dissipation at T_A = 85 °C for suffix 6 or T_A = 105 °C for suffix 7(2)	LQFP144	-	606	mW
		WLCSP100	-	454	
		LQFP100	-	476	
		UFBGA100	-	339	
		LQFP64	-	435	
T_A	Ambient temperature for 6 suffix version	Maximum power dissipation	-40	85	°C
		Low power dissipation(3)	-40	105	
	Ambient temperature for 7 suffix version	Maximum power dissipation	-40	105	°C
		Low power dissipation(3)	-40	125	
T_J	Junction temperature range	6 suffix version	-40	105	°C
		7 suffix version	-40	125	

1. To sustain a voltage higher than V_{DD}+0.3 V, the internal pull-up/pull-down resistors must be disabled.
2. If T_A is lower, higher P_D values are allowed as long as T_J does not exceed T_{Jmax} (see *Section 7.7: Thermal characteristics*).
3. In low power dissipation state, T_A can be extended to this range as long as T_J does not exceed T_{Jmax} (see *Section 7.7: Thermal characteristics*).

Obr. 3.21 Všeobecné provozní podmínky (převzatá tabulka 19 v [10])

3.5.3 Stanovení velikosti přípustného proudu na připojeném pinu

Velikost proudu, jenž může procházet pinem konfigurovaném jako výstupní, nezávisí na typu pinu. To znamená, že je tato hranice pro všechny I/O piny stejná. Podle obr. 3.22 je max. vstupní/výstupní proud 25 *mA*, **avšak** součet proudů všech I/O pinů musí být max. 80 *mA*. To znamená, že lze použít najednou max. čtyři piny po 20 *mA*. Navíc součet všech proudů, které je Nucleo schopno najednou přijmout/vydat, je **maximálně** 160 *mA*. Zaveďme opět jedno shrnující pravidlo, které se může zdát jako přehnaně omezující, avšak v praxi běžné: z/do pinu vždy **odebírejte/přivádějte** proud **maximálně** blízký hodnotě **5 *mA***.

Symbol	Ratings	Max.	Unit
ΣI_{VDD}	Total current into sum of all VDD_x power lines (source)	160	mA
ΣI_{VSS}	Total current out of sum of all VSS_x ground lines (sink)	-160	
I_{VDD}	Maximum current into each V_{DD_x} power line (source)(1)	100	
I_{VSS}	Maximum current out of each V_{SS_x} ground line (sink)(1)	100	
$I_{IO(PIN)}$	Output current sunk by any I/O and control pin	25	
	Output current source by any I/O and control pin	-25	
$\Sigma I_{IO(PIN)}$	Total output current sunk by sum of all IOs and control pins(2)	80	
	Total output current sourced by sum of all IOs and control pins(2)	-80	
$I_{INJ(PIN)}$	Injected current on FT, FTf, and B pins(3)	-5/+0	
	Injected current on TC and RST pin(4)	±5	
	Injected current on TTa pins(5)	±5	
$\Sigma I_{INJ(PIN)}$	Total injected current (sum of all I/O and control pins)(6)	±25	

1. All main power (V_{DD}, V_{DDA}) and ground (V_{SS} and V_{SSA}) pins must always be connected to the external power supply, in the permitted range.
2. This current consumption must be correctly distributed over all I/Os and control pins.The total output current must not be sunk/sourced between two consecutive power supply pins referring to high pin count LQFP packages.
3. Positive injection is not possible on these I/Os and does not occur for input voltages lower than the specified maximum value.
4. A positive injection is induced by $V_{IN} > V_{DD}$ while a negative injection is induced by $V_{IN} < V_{SS}$. $I_{INJ(PIN)}$ must never be exceeded. Refer to *Table 16: Voltage characteristics* for the maximum allowed input voltage values.
5. A positive injection is induced by $V_{IN} > V_{DDA}$ while a negative injection is induced by $V_{IN} < V_{SS}$. I_{INJ}(PIN) must never be exceeded. Refer also to *Table 16: Voltage characteristics* for the maximum allowed input voltage values. Negative injection disturbs the analog performance of the device. See note (2) below *Table 81*.
6. When several inputs are submitted to a current injection, the maximum $\Sigma I_{INJ(PIN)}$ is the absolute sum of the positive and negative injected currents (instantaneous values).

Obr. 3.22 Proudová charakteristika (převzatá tabulka 17 v [10])

3.5.4 Programovatelné pull rezistory

Řešení minulého programu (z kap. 3.5.1) funguje, avšak nedostatečně. Pokud ke vstupnímu pinu připojíme 0 *V* (+3,3 *V*), dioda nesvítí (svítí). Co když ale k vstupnímu pinu není nic připojeného? Pokud Váš program vypadá podobně jako výsledné řešení v kap. 3.5.1, pak se při nezapojeném vstupu dioda chová „zvláštně“. Chvíli svítí, chvíli ne a občas dokonce bliká. Proč tomu tak je a co s tím můžeme dělat?

Výše uvedeným problémem se zabývá kap. 4.3.3 v[1]. Jak je v této kapitole uvedeno, programovatelné pull rezistory slouží k tomu, aby byla vždy na vstupním pinu definovaná logická úroveň. V případě nepřipojeného vstupního pinu je pin tzv. připojený „do vzduchu“, tedy má nedefinovanou logickou úroveň.

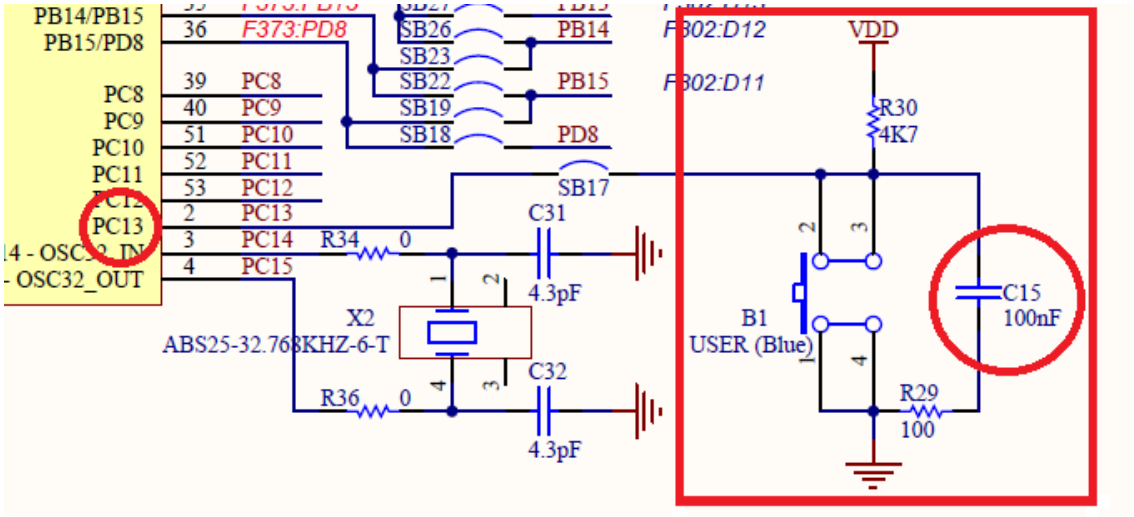
V kap. 3.5.1 jsme mód pinu neurčili, implicitně je nastaven PullNone. Využitím PullDown rezistoru se vyřeší „náš problém“ a program bude fungovat tak, jak jsme si ho v kap. 3.5.1 zadali. Využijeme-li PullUp rezistor, program bude fungovat opačně. Obě varianty si sami vyzkoušejte. Změna módu pinu:

```
DigitalIn input_pin(PC_0, PullDown);
```

3.5.5 Zapojení externího a interního tlačítka a odskakování kontaktů

Způsoby zapojení externího tlačítka jsou velmi přehledně vysvětleny v kap. 4.3.5 v [1]. Následuje vysvětlení jevu označovaného jako odskakování kontaktů (angl. bouncing), který je s použitím tlačítka spojen. Tato problematika je také přehledně popsána, proto si ji přečtěte tam.

Zdůrazněme, že bouncing se dá odstranit dvěma způsoby a to buď hardwarově, nebo softwarově. Hardwarově to jde přidáním kapacitoru zapojeným paralelně k tlačítku. Tento způsob je využit i u uživatelského tlačítka, viz obr. 3.23. V červeném obdélníku je celé zapojení tlačítka, v červeném pravém kruhu je připojený paralelní kondenzátor a v levém červeném kruhu je zvýrazněn pin PC13, ke kterému je tlačítko skutečně připojeno.



Obr. 3.23 Zapojení uživatelského tlačítka (převzato z [4])

Třída DigitalInOut

Pokud deska disponuje malým počtem pinů (nebo z jiných důvodů), je možné počet použitých pinů snížit využitím třídy *DigitalInOut*. Tato třída, jak již název napovídá, umožňuje použít jeden pin jako vstupní a zároveň výstupní. Pin samozřejmě nemůže obě možnosti zastupovat najednou, avšak je možné mezi těmito volbami přepínat v průběhu programu. Třídou *DigitalInOut* se zabývá kap. 4.4 v [1].

3.6 Třída Serial

Doposud byl uveden pouze jeden způsob, jak získat informaci z Nuclea a to rozsvícením/zhasnutím interní (kap. 3.2.1) nebo externí (kap. 3.4.1) diody. Tato informace je v mnoha případech nedostačující. Na řadu přichází třída *Serial*, pomocí které lze posílat znaky mezi deskou a PC prostřednictvím sériové komunikace v asynchronním režimu (UART). Informace o používání této třídy jsou v kap. 4.8 v [1]. Dále je uveden výtah z této kapitoly.

Na inicializování sériového portu jsou potřeba dva piny a to přijímací a vysílací. Tentokrát se ovšem nejedná o jakékoli dva piny, ale pouze o definované dvojice. Tyto dvojice nejsou ve výše uvedeném pinout diagramu (obr. 3.11), ale na obr. 3.12, obr. 3.13 a obr. 3.14 vyznačené žlutou barvou. Cílem je najít dvojici se stejnou číslovkou za názvem Serial, jeden s dovětkem RX a druhý s TX. Jedna z těchto dvojic je např. PA2 a PA3. Samotná inicializace vypadá takto:

```
Serial device(PA_2, PA_3);
```

Rychlost komunikace (tzv. baudrate) je implicitně nastavena na 9600 Bd/s. To znamená, že sériový port je schopen přenést maximálně 9600 bitů za sekundu. Rychlost komunikace se nedá nastavit libovolně.

Pro naše účely je nejdůležitější, jak přečíst/zapsat znak ze/do sériového portu (dále jen s. portu). Pro čtení znaku ze s. portu jsou potřeba funkce *readable()* a *getc()*. Pokud *readable()* vrátí true, pak teprve čteme ze s. portu pomocí *getc()*. Využitím funkcí *writeable()* a *putc()* se znak naopak do s. portu zapisuje. Nově je skutečně *writeable()* a ne *writable()*, jak je psáno v [1]. Zdůrazněme, že tímto způsobem se přečte/zapíše ze/do s. portu pouze **jeden** znak. K přečtení/zápisu formátovaného řetězce se používají funkce *printf()* a *scanf()*.

Mbed umožnuje použít funkce *print()* a *scanf()* bez vytvoření objektu třídy *Serial*. Funkčnost, i když se to při vyzkoušení na jednoduchém příkladu může zdát, není stejná. Mohou nastat problémy s grafickou úpravou v terminálu, příp. celková nefunkčnost. Proto bychom **vždy** měli nejdříve vytvořit objekt třídy *Serial* a s uvedenými funkcemi pracovat přes tento vytvořený objekt. Např. v příkladu v kap. 3.6.3 se posílá úvodní hláška tímto způsobem:

```
serial.printf("\n\rWrite your name and press ENTER.\r\n"); //introductory message
```

Nikoli takto, i když by mbed nehlásil žádnou chybu:

```
printf("\n\rWrite your name and press ENTER.\r\n"); //introductory message
```

3.6.1 Terminálový emulátor

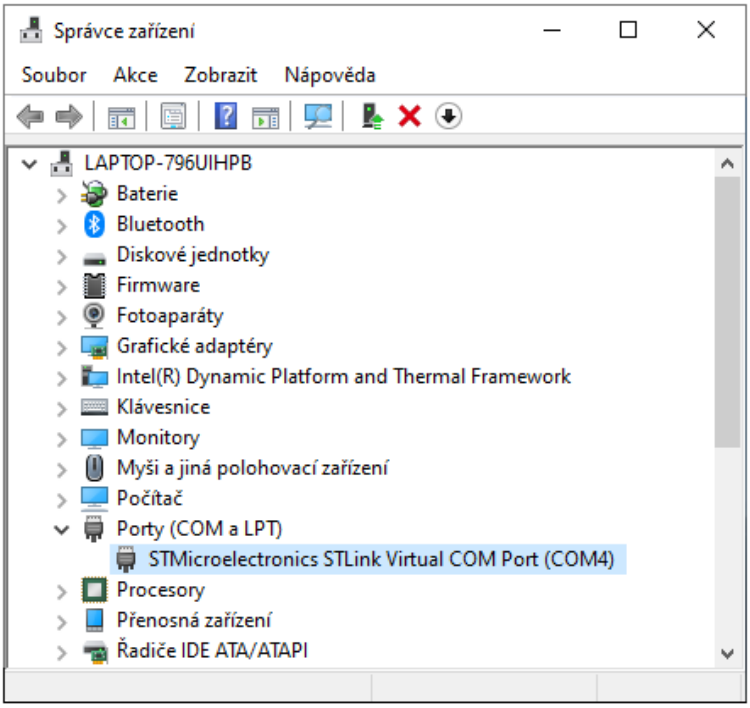
Než si přijímání a zapisování znaků/řetězců do sériového portu vyzkoušíme, je potřeba nejdříve nainstalovat terminálový emulátor, který zastupuje PC část komunikace (viz kap. 4.8.11 v [1]). Takových emulátorů je velké množství, např. PuTTy, Tera Term a Xterm. Nainstalovat si můžete oba.

Pokud je Nucleo připojené přes kabel do PC a je spuštěný emulátor Tera Term, objeví se okno *New connection*. V tomto okně přepneme komunikaci na *Serial* a spustíme. V emulátoru PuTTy je také potřeba přepnout typ komunikace na *Serial*, avšak zde musíme navíc zadat správný COM, na kterém je Nucleo připojeno. Tato informace se zjistí ve *Správci zařízení*. Kupříkladu na obr. 3.24 je připojeno na COM4.

3.6.2 Příklad: Příjem a vyslání jednoho znaku přes sériový port

Vytvořme program, který bude vracet znak přijatý z PC z terminálového emulátoru. Program obsahuje všechny funkce pro příjem a vyslání **jednoho** znaku (viz kap. 3.6). Celý kód je uveden v příloze 12.2. Vnitřek nekonečné smyčky může vypadat takto:

```
if (serial.readable()){
        sign = serial.getc();
        if (serial.writeable()){
                serial.putc(sign);
        }
}
```



Obr. 3.24 Nalezení čísla COMu ve Správci zařízení

3.6.3 Příklad: Práce s řetězci (samostatná činnost)

Tento program je prvním „větším“ projektem, ve kterém využijeme doposud získané znalosti z této práce a také z programování. Věnujme mu tedy náležitou pozornost. Zopakujme, že je potřeba postupovat systematicky a být si jistý každým řádkem. Za tímto účelem je vhodné program často kompilovat a ověřovat, že skutečně dělá to, co předpokládáme. Doporučení: žádné háčky, proměnné i komentáře psát anglicky.

Vytvořme program, který pošle přes s. port jméno uživatele po znacích. Načítání jména bude ukončeno vhodnou klávesou (např. enterem). Po načtení jména program skončí. Jako informaci o začátku/ukončení programu je vhodné přidat několik zablikání diody. V průběhu komunikace by mělo Nucleo znaky kontrolovat (kontrola, zda se jedná skutečně o písmena, příp. enter). Funkce vykonávající uvedené kontroly jsou implementovány např. v knihovně *ctype.h*.

Načtené znaky se dají ukládat do pole konstantní délky (např. o velikosti 25), nebo do dynamicky alokovaného pole, jehož velikost se dá měnit. Co se týče vstupu od uživatele, je důležité pokrýt všechny možné případy a reagovat na ně vhodnou chybovou hláškou, případně i program ukončit. Kupříkladu pokud se použije na ukládání písmen pole konstantní délky a uživatel bude chtít napsat delší řetězec, než je velikost tohoto pole, program skončí s vhodnou chybovou hláškou. Řešení je v příloze 12.3.

Přehlednost v programu

Pro lepší orientaci v programu a rychlejší začátek tvorby nového projektu je vhodné vytvořit si přehlednou univerzální šablonu, kterou můžeme při tvorbě nového projektu naklonovat. Dodržováním jednotnému způsobu oddělování důležitých částí programu získáme výše zmíněné benefity. Šablona může vypadat např. takto:

```
//----------------------------------------//
//What does the program below do?

//----------------------------------------//
//libraries
#include "mbed.h"

//----------------------------------------//
//constants

//----------------------------------------//
//init classes
DigitalOut led(LED1);

//----------------------------------------//
int main() {
        //flashing at the beginning
        for (int i = 0; i < 5*2; ++i) {
                led  =  !led;
                wait(0.05);
        }
        //continue
}
```

3.7 Ladění programu – jak najít chybu?

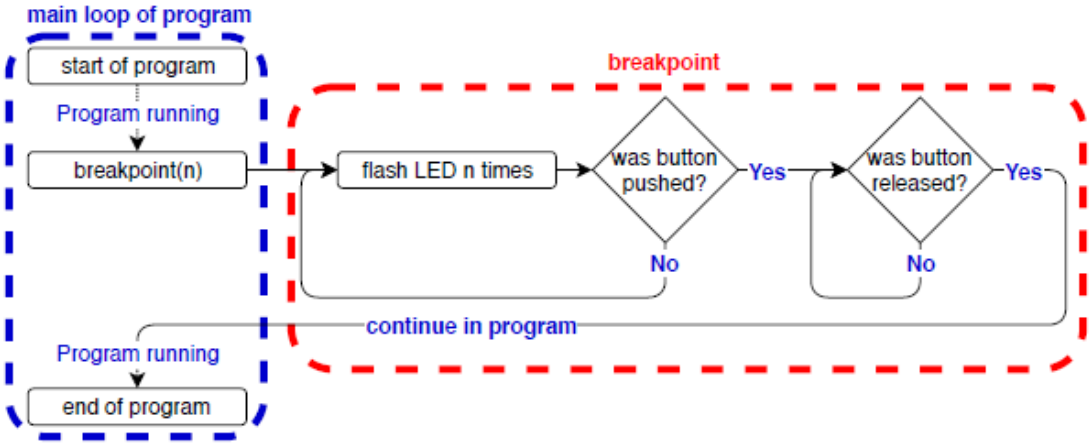
Obecný náhled na ladění programů je v kap. 5.1 v [1]. O tom, co se dá konkrétně dělat při hledání chyb v mbedu, pojednává kap. 5.2 v [3]. Kap. 5.2.3 v [1] přináší způsob poslání čísla řádku programu a čísla proměnné spolu s jejím názvem. Dále jsou v této kapitole uvedeny informace týkající se registrů, ty si číst nemusíte, k nim se vrátíme později. Uvedené kapitoly je vhodné před pokračováním přečíst.

Abychom zjistili, která část programu se momentálně vykonává, můžeme blikat diodou či posílat informace s. portem prostřednictvím funkce *printf()*. Součástí práce [1] bylo také vytvoření knihovny pro ladění programů, jenž dostala název DEBUG UNIVERSAL (viz kap. 6 v [1]). Tato knihovna obsahuje čtyři třídy. První dvě si probereme nyní, k zbývajícím dvěma se vrátíme později v kap. 5.2. Jak importovat knihovnu do programu je v kap. 6.0.2 v [1].

Zdůrazněme, že krokování programu se nedá takto jednoduše použít u programů závislých na časování, jelikož krokování přidá programu zpoždění, které může pozitivně ale i negativně změnit chování programu.

3.7.1 Třída Debug_led

O této třídě pojednává kap. 6.1 v [1]. Třída slouží na krokování programu v procesorech STM32 pomocí LED a tlačítka. Krokování je realizováno pomocí funkce *breakpoint()*, která je definována v této třídě. Funkci můžeme vložit na jakákoli místa v programu. Až program na toto místo dojde, zastaví se a dioda začne blikat. Program bude pokračovat až po stlačení a následném uvolnění tlačítka. Tento cyklus je graficky znázorněn na obr. 3.25 (v [1] obr. 6.5). [1]



Obr. 3.25 Stuktura breakpointu v třídě Debug_led (převzato z [1])

Jak třídu používat je v kap. 6.1.1 v [1]. Teorie je popsána obecně k užití s diodou a tlačítkem. Jak zapojit externí LED a externí tlačítko víme z kap. 3.4.1 a 3.5.5. Následuje ukázka použití s interní LED a interním tlačítkem. Vyzkoušení programu také s externími součástkami je na místě zejména pro ty, kteří si s nimi nejsou stále jisti. Interní tlačítko je připojené na zem, to lze vidět na obr. 3.23, proto se objekt vytvoří takto:

```
Debug_led my_debug(PA_5, PC_13, "BUTTON_TO_GND");
```

Poznamenejme, že podle obr. 3.25 by smyčka *breakpoint()* měla být ukončena až po stlačení a následném uvolnění tlačítka. Experimentálně bylo ověřeno, že se tímto způsobem smyčka *breakpoint()* ve skutečnosti nechová. Po stlačení tlačítka nečeká na jeho uvolnění, ale pokračuje dále v programu. Pokud je tato vlastnost žádoucí, lze za *breakpoint()* přidat následující smyčku, která zmíněnou vlastnost plní:

```
while(!my_btn){} //was button released?
```

Použití třídy si ukážeme na jednoduchém příkladu, ve kterém se pošle zpráva „Hello World“. Bez využití funkce *breakpoint()* by se zpráva po nahrání binárního souboru odeslala a program by skončil. S využitím této funkce se však program zastaví na začátku, po stlačení tlačítka odešle zprávu a pak opět zastaví na konci. Navíc můžeme pomocí argumentu zadávat

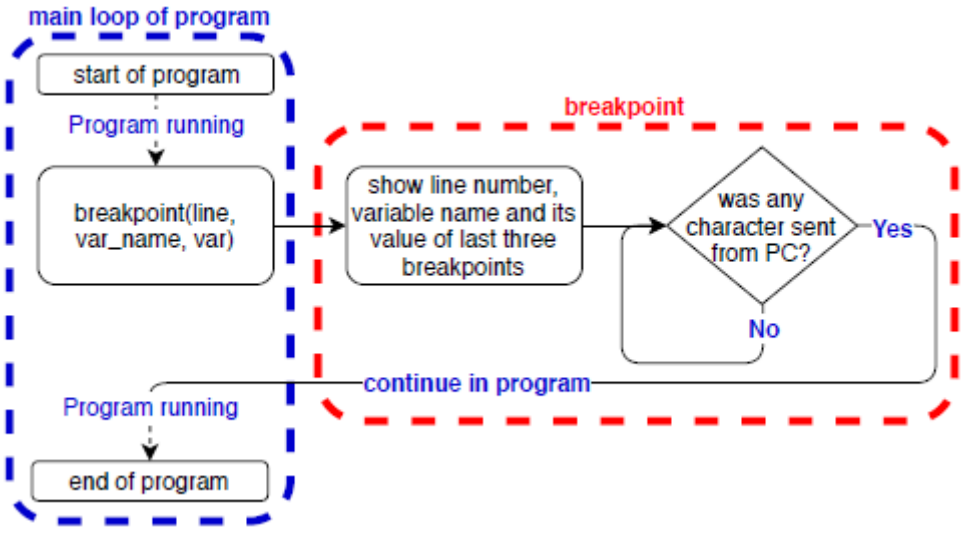
počet bliknutí v periodě a tím tak od sebe rozlišit jednotlivé breakpointy [1]. Jméno objektu se nemůže jmenovat pouze *debug*, jelikož se tak již jmenuje funkce v *mbed_debug.h*.

Řešení popsaného příkladu:

```
#include "mbed.h"
#include "Debug.h"
Serial serial(PA_2, PA_3);
DigitalIn my_btn(PC_13);
Debug_led my_debug(PA_5, PC_13, "BUTTON_TO_GND");
int main(){
        my_debug.breakpoint(2);
        while(!my_btn){} //was the button released?
        serial.printf("Hello World\r\n");
        my_debug.breakpoint(3);
}
```

3.7.2 Třída Debug_serial

Informace o třídě Debug_serial jsou uvedeny v kap. 6.2 v[1]. Použití této třídy je podobné třídě Debug_led z kap. 3.7.1 (porovnejte obr. 3.26 a obr. 3.25). Při použití funkce *breakpoint() se* nebliká, ale posílá zpráva s informací zahrnující číslo řádku, případně jméno a hodnotu určité proměnné. Program pokračuje po stlačení jakékoli klávesy, nikoli tlačítka. [1]



Obr. 3.26 Stuktura breakpointu v třídě Debug_serial (převzato z [1])

Následuje jednoduchý příklad na základní použití této třídy. V programu se do proměnné datového typu *int* několikrát za sebou přičte jednička. Mezi jednotlivými součty jsou vložené zmíněné breakpointy. Řešení popsaného příkladu:

```
#include "mbed.h"
#include "Debug.h"
Debug_serial pc(PA_2, PA_3);
int main() {
    int a=0;
    pc.breakpoint(__LINE__, name(a), a);
    a+=1;
    pc.breakpoint(__LINE__, name(a), a);
    a+=1;
    pc.breakpoint(__LINE__, name(a), a);
    a+=1;
    pc.breakpoint(__LINE__, name(a), a);
}
```

Zdůrazněme, že předvolená hodnota baudrate při použití třídy *Debug_*serial baudrate je 115 200 *Bd/s*. [1]

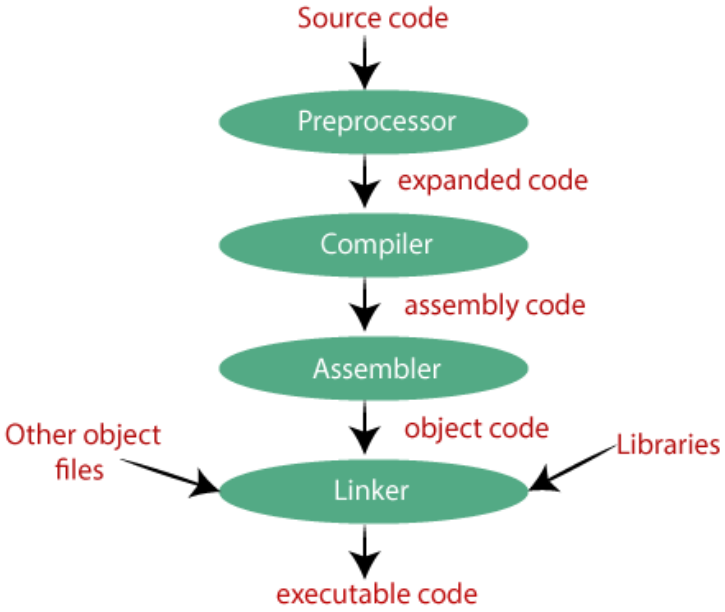
Kompilace programu a využití hlavičkových souborů

Při tvorbě větších projektů v C/C++ je vhodné projekt rozdělit na menší části a ty realizovat v samostatných souborech. Na jednotlivých částech projektu tak kupříkladu mohou pracovat různí lidé. Další výhodou tohoto přístupu je zlepšení čitelnosti kódu a jeho údržby. Celý projekt je pak nutné „poskládat dohromady“ a vytvořit z něj (v našem případě) binární soubor, který lze nahrát do mikrokontroléru. Tento proces se označuje jako kompilace.

Kompilace projektu probíhá v několika fázích, jak je naznačeno na obr. 3.27. Nejdříve dojde v *Preprocessoru* k fázi *preprocessingu*, při které se nahradí makra, naincludují požadované soubory, zkrátka se vyhodnotí všechny direktivy preprocesoru, tzn. všechny řádky začínající znakem # (mřížka). [32]

Výsledný kód přejde přes *Compiler*, na jehož výstupu je assembler kód složený z instrukcí (koncovka **.s**). Z něj se dále vytvoří objektový soubor (koncovka **.o**), který vstupuje do *Linkeru*. Ten spojí všechny objektové soubory do jednoho a udělá z něj spustitelný soubor, což je v kontextu práce s mikrokontroléry soubor binární.

S procesem kompilace souvisí také využití hlavičkových souborů (angl. header files). Při programování větších projektů je vhodné rozdělit kód do hlavičkových (s koncovkou **.h**) a implementačních (s koncovkou **.c**) souborů. Hlavičkové soubory obsahují pouze deklaraci. Říkají tedy pouze to, že někde existuje určitá funkce (struktura, třída apod.). Tím dávají ostatním souborům informace o tom, co je v jiných implementačních souborech obsaženo. [33]



Obr. 3.27 Proces kompilace programu (převzato z [35])

3.8 Třídy využívající čítače

V této kapitole se seznámíme s novou periferií zvanou čítač a s některými třídami, které tuto periferii využívají. Poznamenejme, že čítače pro svou činnost využívá i třída *PwmOut* (viz kap. 3.9).

3.8.1 Seznámení s periferií čítač

Periferie zvaná čítač je angl. označovaná jako **Timer**. Názvosloví je komplikované, jelikož čítač (angl. counter) také značí funkci, kterou čítač vykonává. Druhá funkce, kterou čítač plní, je časovač (angl. timer).

Obecně mikrokontroléry STM32 obsahují několik čítačů (viz obr. 3.28), které se liší svými vlastnostmi (viz obr. 3.29). Kupříkladu se dají čítače použít ke generování PWM signálu (viz kap. 3.9), pro měření časových úseků (o tomto použití pojednává tato kapitola) a pro počítání náběžných/sestupných (nebo obou) hran sign. [2].

Peripheral		STM32F303Rx		STM32F303Vx		STM32F303Zx	
Flash (Kbytes)		384	512	384	512	384	512
SRAM (Kbytes) on data bus		64					
CCM (Core Coupled Memory) RAM (Kbytes)		16					
FMC (flexible memory controller)		NO		YES			
Timers	Advanced control	2 (16-bit)(1)		3 (16-bit)			
	General purpose	5 (16-bit) 1 (32-bit)					
	PWM channels (all) (2)	31		40		40	
	Basic	2 (16-bit)					
	PWM channels (except complementary)	22		28		28	

Obr. 3.28 Čítače v STM32F303xD/E (převzatá část tabulky 2 v [10])

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/ compare channels	Complementary outputs
Advanced	TIM1, TIM8, TIM20	16-bit	Up, Down, Up/Down	Any integer between 1 and 65536	Yes	4	Yes
General-purpose	TIM2	32-bit	Up, Down, Up/Down	Any integer between 1 and 65536	Yes	4	No
General-purpose	TIM3, TIM4	16-bit	Up, Down, Up/Down	Any integer between 1 and 65536	Yes	4	No
General-purpose	TIM15	16-bit	Up	Any integer between 1 and 65536	Yes	2	1
General-purpose	TIM16, TIM17	16-bit	Up	Any integer between 1 and 65536	Yes	1	1
Basic	TIM6, TIM7	16-bit	Up	Any integer between 1 and 65536	Yes	0	No

Obr. 3.29 Vlastnosti jednotlivých čítačů STM32F303RE (převzatá tabulka 5 v [10])

3.8.2 Třída Ticker

O třídě *Ticker* se v [1] uvádí pouze přehled formou tabulky (viz obr. 3.30). Oficiální zdroj informací v mbedu je [20].

Ticker			
	funkcia	*popis, vstupné parametre, návratová hodnota*	*príklad*
	Ticker	Vytvorenie objektu triedy Ticker (napr. flipper).	Ticker flipper;

	attach	Priradenie funkcie prerušenia a časového intervalu, v ktorom sa bude prerušenie vyvolávať
		1: adresa funkcie obsluhujúcej prerušenie
		2: premenná typu float, interval v sekundách s rozlíšením na µs, v ktorom sa bude vyvolávať prerušenie
			void func(){
			...
			}
			int main(){
			...
			flipper.attach(&func, 1.2);
			...
			}

	attach_us	Priradenie funkcie prerušenia a časového intervalu, v ktorom sa bude prerušenie vyvolávať
		1: adresa funkcie obsluhujúcej prerušenie
		2: premenná typu int, interval v µs, v ktorom sa bude vyvolávať prerušenie
			void func(){
			...
			}
			int main(){
			...
			flipper.attach_us(&func, 135);
			...
			}

	detach	Zakázanie prerušenia.	flipper.detach();

Obr. 3.30 Třída Ticker (převzato z [1])

Podle obr. 3.30 třída *Ticker* poskytuje tři funkce: *attach(), attach_us() a detach()*. První dvě funkce mají stejný úkol. Periodicky volají funkci předávanou v argumentu. Liší se pouze způsobem předání času definující tuto periodu. Ve funkci *attach()* vstupuje jako druhý

argument počet sekund, zatímco ve funkci *attach_us()* počet mikrosekund. Rozlišení je v obou případech na μs. Poslední funkcí je funkce *detach()*, která periodické volání zruší. [1]

Jinak řečeno, třída umožňuje periodicky volat nějakou funkci v průběhu vykonávání programu. Stačí tedy vytvořit objekt třídy *Ticker*, jednou ho použít a on se bude „sám" vykonávat. Jedná se o zcela novou a velmi užitečnou vlastnost, protože jsme nově schopni vykonávat více činností „zároveň" (nejedná se o úplné paralelní vykonávání instrukcí).

Poznamenejme, že volaná funkce by neměla obsahovat žádné časově náročné operace. To znamená žádné čekání pomocí funkcí *wait()*, nekonečné smyčky, dynamické alokování paměti, posílání znaků přes sériový port apod. [20]

3.8.2.1 Příklad: Základní použití třídy Ticker

Výše popsanou vlastnost si vyzkoušíme na jednoduchém příkladu, ve kterém třída *Ticker* obstarává blikání diodou, zatímco program v mainu obsluhuje sériový port stejným způsobem jako v kap. 3.6.2. Nápad na tento příklad byl převzat z [20]. Program tedy nově nejen posílá znak, který přijde Nucleu po sériovém portu, ale mezitím také bliká danou periodou.

Do programu vytvořeném v kap. 3.6.2 se přidá vytvoření objektu třídy *Ticker* a definice funkce měnící stav LED. Na začátku funkce *main* pak stačí zavolat funkci vykonávající přerušení[6]:

```
  …
Ticker ticker;
  …
void flip(){
   led = !led;
}
  …
int main(){
        ticker.attach(&flip, 0.2); //the LED flips every 0.2 seconds
        …
}
```

Celý program je v příloze 12.4. Pro názornost je navíc vhodné čekat např. 1 *s* po přijetí znaku z PC. Demonstruje se tak skutečnost, že blikání LED opravdu nezávisí na právě vykonávaném programu, ani když se právě vykonává funkce *wait()*.

3.8.3 Třída Timer

V řadě aplikací je potřeba přesné měření krátkých časových úseků. Za tímto účelem mbed nabízí využití třídy *Timer*, o které pojednává kap. 4.10 v [1]. Jméno třídy je stejné jako periferie, která přesné měření času vykonává (viz kap. 3.8.1). Třída obsahuje několik jednoduchých funkcí: *start(), stop()* a *reset()*, jejichž smysl je z názvu zřejmý. Timer si lze

[6] Přerušením prozatím rozumějme proces, při kterém je procesor „přerušen" v jeho chodu. Ten pak zkontroluje, „kdo" ho přerušil a na základě toho vykoná požadované akce (tzv. obsluhu přerušení). V tomto příkladě procesor periodicky přerušuje ticker, který po něm chce, aby změnil logickou úroveň na pinu PA_5, neboli zhasnul/rozsvítil diodu. Více o přerušení v kap. 3.13.

představit jako stopky, pomocí kterých lze měřit krátké časové úseky v řádu *ms*, avšak i delší. Příklad v následující kapitole názorně demonstruje jeho použití.

3.8.3.1 Příklad: Měření doby reakce

Cílem tohoto příkladu je vytvořit program, který bude měřit dobu reakce v řádu *ms*. K tomuto účelu využijeme interní LED a interní tlačítko, i když by se daly stejným způsobem použít i externí součástky. Podstata programu je, aby se rozsvítila LED po náhodně vygenerované době. Po rozsvícení testovaná osoba musí co nejdříve stlačit tlačítko. Čas mezi rozsvícením LED a stlačením tlačítka nazveme **reakční dobou**.

Takovýchto měření provedeme několik. Program může provádět měření, dokud bude reakční doba pod určitou maximální hranicí. Situace, kdy testovaná osoba stlačila tlačítko před rozsvícením, je nepřijatelná a musí být ošetřena. Výsledky posílejme do PC. Reagujme pouze na stlačení tlačítka, nikoli na jeho uvolnění. Dobu mezi stlačením a uvolněním tlačítka ponechme testované osobě na prohlédnutí dosaženého výsledku. Program tedy bude pokračovat až po uvolnění tlačítka. Příklad názorně demonstruje využití třídy Timer. Jelikož se jedná o další „delší“ příklad, ukážeme si názorně, jak takové programy řešit. Příklad rozdělíme na několik částí.

Nejdříve je potřeba zjistit, jak čekat proměnnou dobu. K získání „náhodného“ čísla slouží funkce *rand()* popsaná v kap. 4.5.3 v [1]. Náhodného v uvozovkách proto, že funkce vrací postupně čísla z pseudonáhodné sekvence [1]. Z těchto čísel navíc nevybírá „na přeskáčku“, ale od začátku [1]. Funkce nevybírá čísla od začátku, ale od určitého indexu, který zvolíme funkcí *srand()*[1]. Náhodné číslo v intervalu 300 až 2300, které v další části převedeme na milisekundy, můžeme získat např. takto:

```
srand(time(NULL));
uint32_t num = (rand()%2000)+300;
```

Jak již bylo řečeno, číslo převedeme na *ms*. Nemůžeme však pasivně čekat num *ms*, protože je potřeba kontrolovat, zda testovaná osoba nestlačila předčasně tlačítko. Tato kontrola může být např. každou *ms*. Čekání náhodnou dobu s kontrolou stlačení tlačítka:

```
while(num){ //wait approximately num ms but watch the button
        num--;
        if (!button){
                quit = true; //the button pressed before the LED lights up
                num = 0;
        }
        wait_ms(1);
}
```

Tímto se vyřešila první část: čekání „náhodné“ doby. Pokud nebylo tlačítko předčasně stlačeno, pak následuje rozsvícení LED a změření reakční doby:

```
led = !led; //the LED lights up
trial.start(); //start to measure time
while (button){} //loop until the button is not pressed
trial.stop(); //the button is pushed
```

Následuje pouze vyhodnocení měření. Pokud se nedosáhlo lepšího výsledku, než je daná hranice, v programu je označena jako MAX_REACTION_TIME, pak se program ukončí. V opačném případě se pokračuje dalším měřením. Vyhodnocení měření:

```
trial_time_in_ms = trial.read_ms(); //get the result in ms
if (trial_time_in_ms <= MAX_REACTION_TIME){ //the button was pressed in time
        serial.printf("\t%d ms\r\n", trial_time_in_ms);
        trial.reset(); //DO NOT FORGET TO RESET
} else {
        serial.printf("\tYour time is %d ms. -> GAME OVER!\n\r", trial_time_in_ms);
        quit  = true;
}
```

Zbývá všechny části vložit do jedné smyčky a vhodně okomentovat nezmíněné případy. Celý program najdete v příloze 12.5.

3.8.4 Třída Timeout

Poslední třídou v mbedu využívající čítače je třída *Timeout*, o které je v [1] opět zmíněno pouze formou tabulky (viz obr. 3.31). Zdrojem informací se tak stává pouze oficiální přehled v mbedu (viz [21]). Třída *Timeout* je uvedena pouze pro úplnost, i když s ohledem na její obtížnost poslouží jako jednoduché zopakování.

Z obr. 3.31 je zřejmé, že třída *Timeout* nabízí stejné funkce jako třída *Ticker*. Nově funkce *attach()* a *attach_us()* periodicky nevolají funkci předávanou v argumentu, ale zavolají ji pouze jednou a to po čase, který do zmíněných funkcí vstupuje jako druhý argument. Opět se liší pouze jednotka tohoto času. Pokud přerušení doposud nenastalo, je možné ho zrušit pomocí funkce *detach()*. [21]

Ve volané funkci v přerušení opět nesmí být časově náročné operace stejně jako u třídy *Ticker* v kap. 3.8.2.

Smyslem třídy *Timeout* je tedy vykonání funkce se zpožděním, zatímco se může vykonávat něco jiného. Tento požadavek pro jednoduché programy není častý, proto tuto třídu v další části nevyužijeme. Příklad na použití této třídy může být stejný jako v 3.8.2, avšak s použitím třídy *Timeout* nikoli třídy *Ticker*. V takovém případě se pouze po dané době LED rozsvítí.

Timeout			
	funkcia	*popis, vstupné parametre, návratová hodnota*	*príklad*
	Timeout	Vytvorenie objektu triedy Timeout.	Timeout delay;

	attach	Priradenie funkcie prerušenia a času, po uplynutí ktorého sa prerušenie vyvolá.
		1: adresa funkcie obsluhujúcej prerušenie
		2: premenná typu float, čas v sekundách s rozlíšením na mikrosekundy, po uplynutí ktorého sa vyvolá prerušenie.
			void func(){
			…
			}
			int main(){
			…
			delay.attach(&func, 1.2);
			…
			}

	attach_us	Priradenie funkcie prerušenia a času, po uplynutí ktorého sa prerušenie vyvolá.
		1: adresa funkcie obsluhujúcej prerušenie
		2: premenná typu int, čas v mikrosekundách, po uplynutí ktorého sa vyvolá prerušenie
			void func(){
			…
			}
			int main(){
			…
			delay.attach_us(&func, 1500);
			…
			}

	detach	Zakázanie prerušenia.	cntdwn.detach();

Obr. 3.31 Třída Timeout (převzato z [1])

3.9 Třída PwmOut

Než si ukážeme, co dělá třída *PwmOut*, začněme s motivačním příkladem. Uživatelská LED figurovala doposud téměř v každém příkladu. Rozsvítit a zhasnout, nic víc neumí. Znamená to ale, že již nemůže překvapit?

3.9.1 Motivační příklad

Vraťme se k úvodnímu příkladu z kap. 3.2.1:

```
#include "mbed.h"
DigitalOut myled(LED1);
int main() {
   while(1) {
      myled = 1; // LED is ON
      wait(0.2); // 200 ms
      myled = 0; // LED is OFF
      wait(1.0); // 1 sec
   }
}
```

Pro zopakování, program v nekonečném cyklu rozsvítí LED, čeká 200 *ms*, zhasne ji a čeká 1 *s*. To znamená, že poměr dob, kdy LED svítí a nesvítí, je v poměru 1:5. Tento poměr zachováme, avšak základní dobu, která je v tomto případě 1 *s*, změníme o řád na 100 *ms*:

```
myled = 1; // LED is ON
wait(0.02); // 20 ms
myled = 0; // LED is OFF
wait(0.1); // 100 ms
```

Co se stane? Dioda bude rychle blikat. Změní-li se základní doba o další řád (na 10 *ms*), pak LED již (zdánlivě) blikat nebude. Proč? Lidské oko není schopné rozlišit tak rychlou změnu (rozsvíceno/zhasnuto). Někdo si však mohl všimnout drobné změny. Dioda nesvítí s takovou intenzitou, jako když je klasicky zapnuta po celou dobu. Došlo tedy k změně svítivosti diody. Aby byl tento efekt znatelnější, vytvoříme program, ve kterém se bude měnit výše zmíněný poměr. Změna může kupříkladu nastat po stisknutí tlačítka.

Perioda jednoho děje (rozsvícení, čekání, zhasnutí, čekání) je označena *BASE_TIME*, počet kroků *NUM_STEPS* (kolikrát se bude odečítat od *BASE_TIME*) a rozdíl mezi jednotlivými kroky *STEP*:

```
#define BASE_TIME 5000 //is us
#define NUM_STEPS 20
#define STEP BASE_TIME/NUM_STEPS
```

Velikost tohoto kroku se získá ze vzorce

$$STEP\ [us] = \frac{BASE\_TIME}{NUM\_STEPS}. \tag{3.1}$$

Program dělá to samé, avšak doba čekání po rozsvícení a zhasnutí LED se po stisku tlačítka mění. Začíná se ve stavu, kdy LED 100 % času svítí (0 % nesvítí). Pokud je *NUM_STEPS* rovno 20 (viz výše), pak se každým stiskem (a uvolněním) tlačítka zmenší doba, kdy LED svítí o 5 %. Po prvním kroku LED svítí 95 % času a 5 % nesvítí. Tímto způsobem se svítivost diody postupně snižuje, až nakonec dosáhne stavu: 0 % času svítí a 100 % času nesvítí.

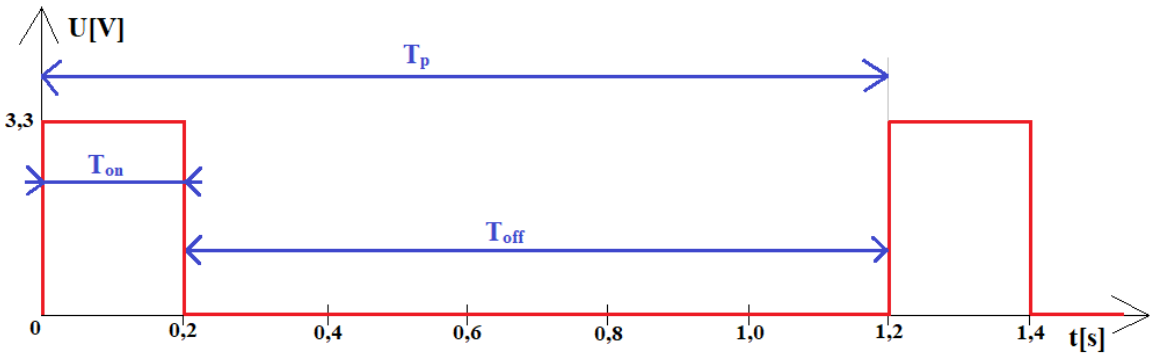
Je vhodné si také vyzkoušet změnu parametru *BASE_TIME*. Pokud např. dojde ke zvýšení o řád (tedy na 50 000 = 50 *ms*), pak lidské oko změnu sign. opět rozliší. Poznamenejme, že pokud je tlačítko stlačené, pak LED nesvítí. Jelikož se jedná o motivační příklad, není toto chování nežádoucí. Dá se však odstranit využitím třídy *Ticker* (viz kap. 3.8.2). V takovém případě by LED „blikala“ stejně i po dobu, kdy by tlačítko bylo stlačené. Celý příklad je uveden v příloze 12.6, přičemž hlavní smyčka vypadá takto:

```
if(!button){
        time -= STEP;
        while(!button){}
}
led = 1;
wait_us(time);
led = 0;
wait_us(BASE_TIME-time);
```

Shrnutí příkladu je tedy takové, že rychlým rozsvěcením a zhasínáním diody a vhodným poměrem mezi trváním obou stavů se dá nastavit její svítivost. V podstatě se tak mění energie dodaná diodě.

3.9.2 PWM signál

Opět se vrátíme k prvnímu vytvořenému programu (kap. 3.2.1). Jak vypadá výstupní napětí na pinu PA_5 by měl každý vědět. Ideální grafické znázornění je na obr. 3.32. Proč je popisek tohoto obrázku „PWM sign.“? Protože jsme takto jednoduše sestrojili pulzně-šířkově modulovaný sign. (angl. Pulse Width Modulation, zkráceně PWM).



Obr. 3.32 PWM signál

V motivačním příkladu (kap. 3.9.1) byl zmíněn poměr, kdy LED svítí a nesvítí, což by podle obr. 3.32 odpovídalo poměru T_{on} : T_{off}. Tato hodnota se standardně neuvádí. Místo ní se však používá pojem **střída** (angl. duty cycle), zn. D. Pro střídu sign. platí

$$D\ [\%] = 100\frac{T_{on}}{T_p} = 100\frac{T_{on}}{T_{on} + T_{off}}, \tag{3.2}$$

kde T_{on} je šířka pulzu a T_{off} čas, po který je pulz v 0 (viz obr. 3.32). Ze vzorce (3.2) je zřejmé, že střída může být minimálně 0 % a maximálně 100 %. Těmto krajním hodnotám odpovídají případy, kdy se pulz nachází celou periodu v log. 0 a log. 1.

O stejném tématu také pojednává kap. 4.6.1 v [1]. Na konci zmíněné kapitoly je uvedeno, že pokud v mbedu změníme střídu, změní se T_{on} tak, aby byl poměr podle vzorce (3.2) co nejblíže k zadanému číslu. Největší rozlišení je na μs, proto v případě nastavení T_p na 7 μs a střídy 50 % mbed nenastaví T_{on} na 3,5 μs, nýbrž na 3 μs. [1]

3.9.3 Funkce třídy PwmOut

Využitím třídy *PwmOut* lze na daném pinu vytvořit podobný sign. jako na obr. 3.32. Způsob, kterým mbed sign. vytvoří, je však odlišný. Pro svoji činnost využívá čítač (viz kap. 3.8.2) stejně jako třídy v kap. 3.8. Jelikož několik tříd používá stejnou periferii, musí se o ni „nějak podělit“. Jak?

Soubor, v němž se v mbedu definují piny pro práci s periferiemi ADC, DAC, I2C, PWM, UART (Serial), SPI a CAN, se nazývá *PeripheralPins.c* (viz [13]). V uvedeném souboru se pod nadpisem „***PWM***“ nachází komentář „TIM2 cannot be used because already used by the us_ticker“. To znamená, že pro generování PWM sign. se nepoužívá čítač 2. Jelikož Nucleo obsahuje čítačů několik (viz kap. 3.8.2), zbývá jich stále mnoho. Konkrétně mbed pro generování PWM sign. může použít čítače 1, 3, 4, 8, 15, 16 a 17 [13].

Třída *PwmOut* nabízí 10 funkcí (viz [22]):

- *period()*, *period_ms()* a *period_us()* pro nastavení periody signálu,
- *pulsewidth()*, *pulsewidth_ms()* a *pulsewidth_us()* pro nastavení šířky pulzu T_{on},
- *write()* a *read()* pro zápis a čtení střídy,
- *suspend()* pro zastavení generování PWM sign.,
- *resume()* pro pokračování v generování PWM sign.

které jsou velmi obsáhle popsány v kap. 4.6 v [1], proto si zde uvedeme jenom krátké shrnutí. V případě nejasností se tedy obraťte na zmíněný zdroj.

Podstatné je, aby se na začátku programu **nejdříve** nastavila perioda signálu a až potom šířka pulzu T_{on}. Maximální rozlišení v mbedu je na μs. Jelikož je střída sign. nepřímý parametr, při využití funkce *write()* se ve skutečnosti mění šířka pulzu T_{on} jakožto přímý parametr. [1]

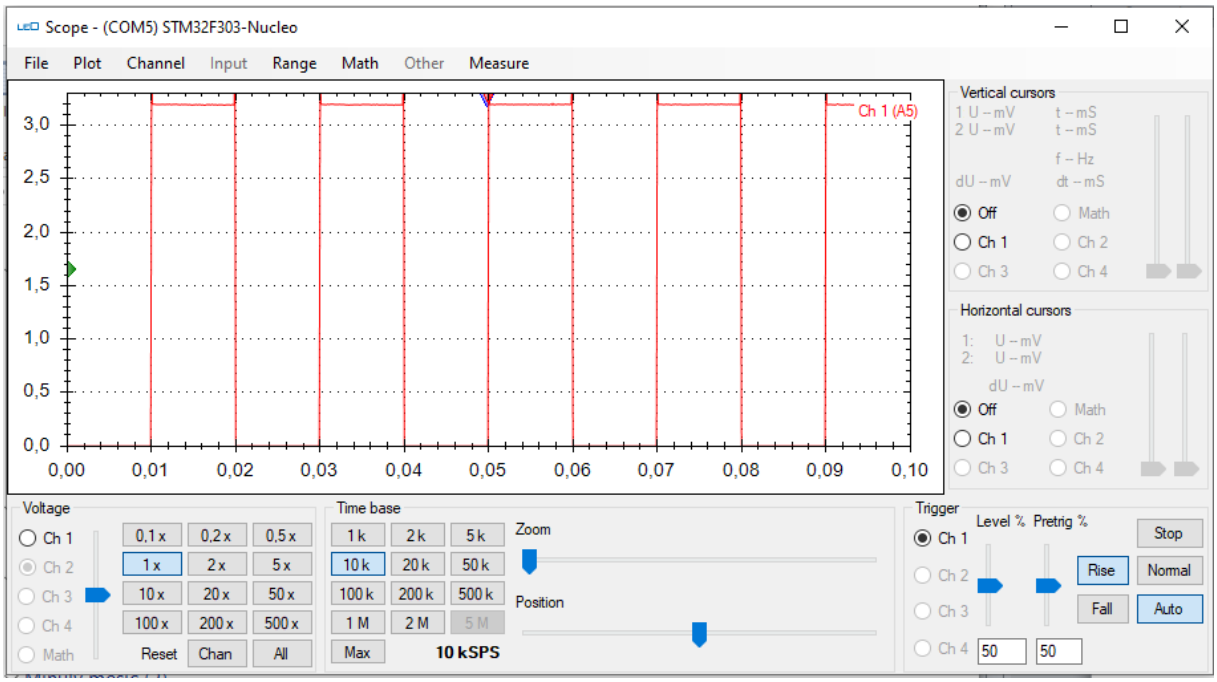
3.9.4 Příklad: Základní použití třídy PwmOut

Problematikou výběru pinu se zabývá kap. 3.9.5. Jelikož se pin PA5 nedá pro třídu *PwmOut* použít, k ověření správnosti generovaného PWM sign. je potřeba připojit externí diodu nebo osciloskop[7]. První případ je obdoba úvodního motivačního příkladu (viz kap. 3.9.1), avšak s využitím externí LED. Ověří se však pouze skutečnost, že se na pinu „něco generuje“. První případ je vhodné použít pro ověření řešení následujícího příkladu. Dále však budeme požadovat lepší představu o generovaném sign., kterou obstará již zmíněný osciloskop (viz kap. 3.9.7).

Následuje jednoduchý program, v němž se na pinu PC0 generuje PWM sign. o periodě 20 *ms* a střídě 50 %. Výsledný průběh napětí generovaný na pinu PC0 je zobrazen na obr. 3.33.

```
#include "mbed.h"
#define PWM_PIN PC_0
#define PWM_PERIOD 20 //in us
#define PWM_PULSEWIDTH 10 //in us
PwmOut my_pwm(PWM_PIN);
int main(){
   my_pwm.period_us(PWM_PERIOD); //set period
   my_pwm.pulsewidth_us(PWM_PULSEWIDTH); //set pulsewidth
   while(1){}
}
```

[7] Osciloskop je přístroj vykreslující časový průběh měřeného napěťového signálu.



Obr. 3.33 Generovaný signál na pinu PC0

3.9.5 Piny použitelné pro generování signálu PWM

Piny, které lze pro generováni PWM sign. nakonfigurovat, jsou vyznačeny fialovou barvou na obr. 3.12, obr. 3.13 a obr. 3.14. V políčku u takového pinu se nachází nápis „PWMX/Y“, kde písmeno X značí číslo čítače a Y číslo kanálu, které pin pro generování používá.

U pinu PA7 je za takovýmto označením navíc písmeno „N“ (tedy „PWM1/1N“). Písmeno „N“ značí negovaný sign., tzn. že sign. opačný tomu, co je na pinu s označením „PWM1/1“. Na základě experimentů však bylo zjištěno, že pokud generujeme dva PWM sign. na takto závislých pinech, pak jeden z nich nepracuje správně. Kupříkladu generováním PWM sign. s periodou 20 *ms* a střídou 50 % na pinech PC1 („PWM1/2“) a PB0 („PWM1/2N“) vznikne průběh vyobrazený na obr. 3.34, což není požadovaný průběh. Tento závěr potvrzuje i kap. 4.6.2 v [1], tedy **nelze** zároveň generovat PWM sign. na dvou vzájemně negovaných pinech.

Ve fialovém políčku se také může nacházet označení „-----“, které značí, že na základě experimentů provedené v [1] bylo zjištěno, že daný pin nelze pro tuto konfiguraci použít.

Při generování více PWM sign. je nutné řídit se **posledním pravidlem**. Pokud chceme použít dva piny, které používají stejný čítač, ale jiný kanál, pak generované sign. **musí** mít stejnou frekvenci, střídu však měnit lze. Pravidlo je natolik důležité, že mu je věnována následující kapitola.

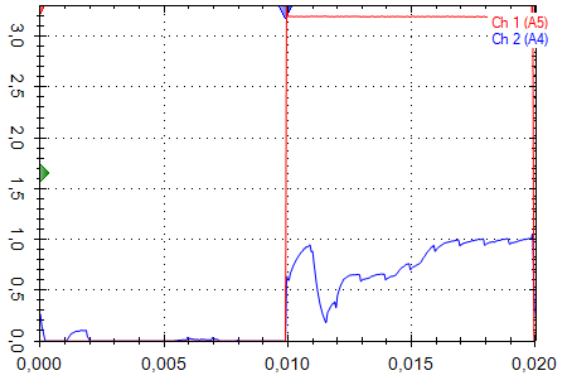
3.9.6 Příklad: Generování dvou PWM signálů s využitím stejného čítače

Pravidlo si demonstrujme na pinech PC0 („PWM1/1“) a PC1 („PWM1/2“), které, jak je z názvu patrné, pro svou činnost používají čítač 1 a kanál 1 a 2. Pokud na PC1 generujeme PWM sign. poloviční PC0, pak je výsledek zcela chybný (viz obr. 3.35).

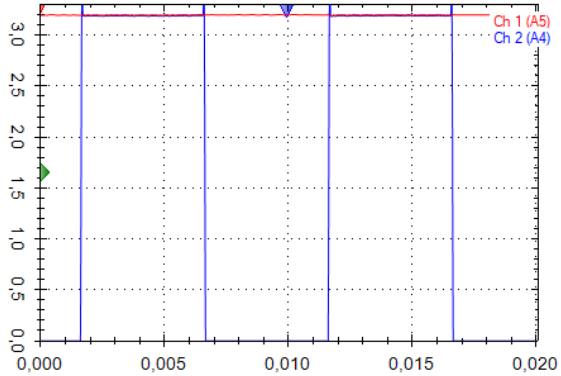
Případ pro dvojnásobnou velikost periody na PC1 je na obr. 3.36. Průběh vypadá v pořádku, na PC0 (červený průběh) je PWM sign. s periodou 20 *ms* a střídou 50 %, zatímco na PC1

(modrý průběh) je PWM sign. s periodou 40 *ms* a stejnou střídou. Průběh na PC1 je posunutý (měl by být od 0,0 do 0,2 nulový a od 0,2 do 0,4 aktivní), ale to na „funkci“ nemá vliv.

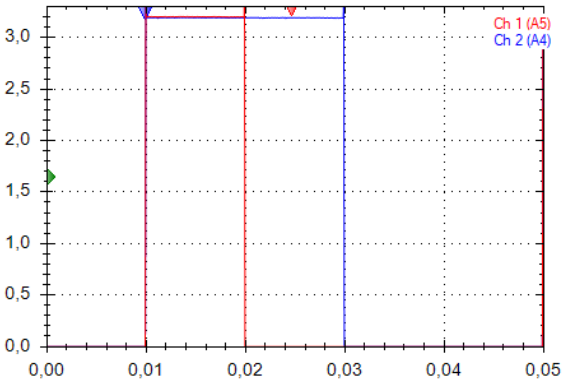
Na posledním obrázku (obr. 3.37) je na PC1 trojnásobná perioda oproti PC0. V tomto případě je průběh na PC1 (modrý) správný (i když opět posunutý), na rozdíl od průběhu na PC0 (červený).



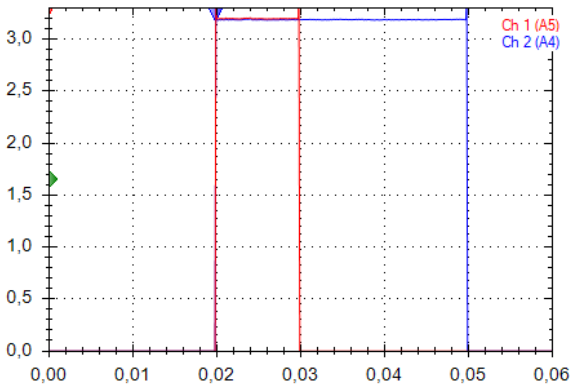
Obr. 3.34 Negovaná dvojice pinů



Obr. 3.35 Poloviční perioda na PC1



Obr. 3.36 Dvojnásobná perioda na PC1



Obr. 3.37 Trojnásobná perioda na PC1

3.9.7 Little embedded oscilloscope

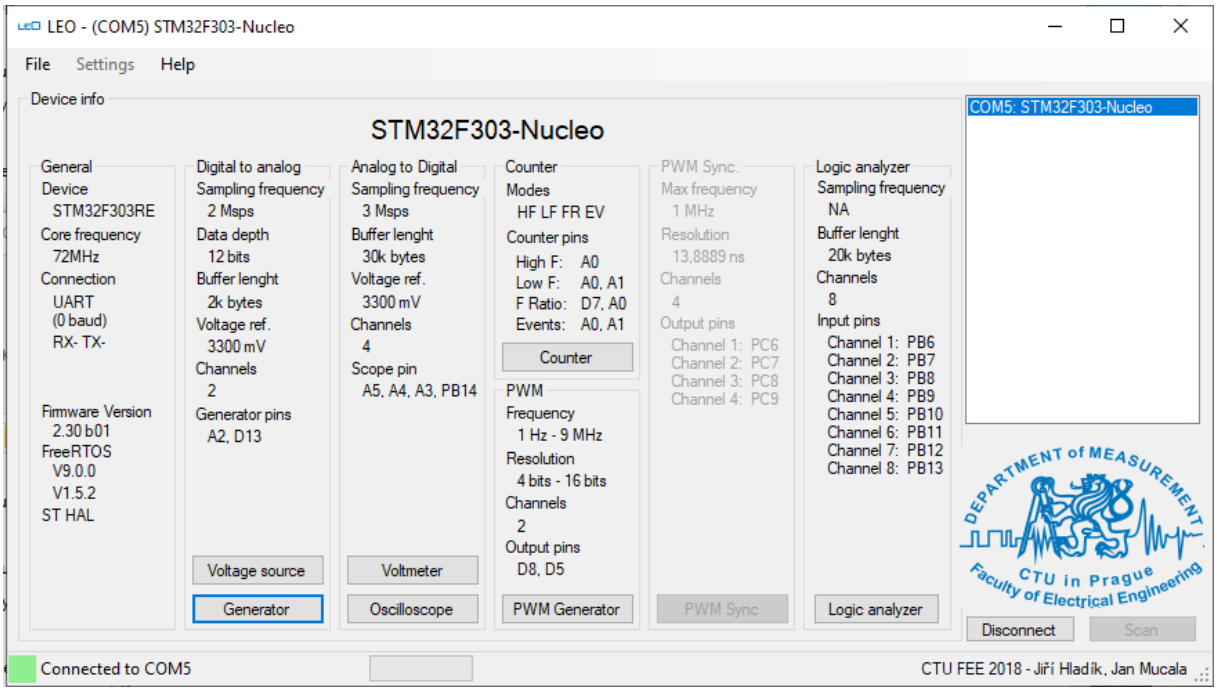
Jak již bylo zmíněno, osciloskop (dále jen OSC) je zařízení zobrazující napěťový průběh měřeného napěťového sign. Zcela běžné je měření hned několika průběhů najednou. Pomocí OSC byly vygenerovány všechny předešlé průběhy. Pokud disponujete mikrokontrolérem **STM32F303RE**, pak si takové průběhy můžete sami naměřit.

V laboratoři **videometrie na katedře měření ČVUT FEL** byl vyvinut software **LEO** (Little embedded oscilloscope), který zastane nejen funkci OSC, ale i zdroje napětí, voltmetru, generátoru, PWM generátoru, čítače a logického analyzátoru. Souhrnně se těmto aplikacím říká **přístrojové aplikace**. V kap. 8 si sami vytvoříme logický analyzátor. Veškeré informace o zmíněném softwaru LEO jsou v [29]. Poznamenejme, že pro měření jsou potřeba dvě desky, jelikož do jedné se nahraje LEO a do druhé vytvořený program.

Po stažení aktuální verze softwaru LEO stačí pouze nahrát binární soubor do Nuclea. Zobrazí se okno, které momentálně vidíme na obr. 3.38. V pravé části je okno, kde se zobrazují připojené desky. K využití OSC se stačí k dané desce připojit a kliknout na *Oscilloscope*. Okno OSC je podobné jako na obr. 3.33.

Účelem této kapitoly není podrobně vysvětlovat, jak LEO používat. Veškeré informace jsou uvedeny v [29]. Cílem je pouze seznámení s tímto softwarem. Zdůrazněme však, že vstupní rozsah napětí bez využití odporového děliče se může pohybovat v rozmezí **0 až +3,3 *V***. Pro ochranu je vhodné připojovat piny přes rezistory o odporu alespoň 470 **Ω**.

Nyní je vhodné přečíst si web, stáhnout software LEO a seznámit se s ním. Dále vyzkoušet OSC na pinech GND příp. +3,3 *V* a získat PWM průběhy kupříkladu stejné jako na předešlých obrázcích této kapitoly. Dále si můžete vytvořit program, ve kterém se na nějakém pinu generuje PWM sign. s určitou střídou a po stlačení tlačítka se tato střída zvýší/sníží. Průběh samozřejmě sledujte na OSC.



Obr. 3.38 Grafické rozhraní pro LEO

3.9.8 Příklad: Generování PWM signálu pomocí třídy Ticker

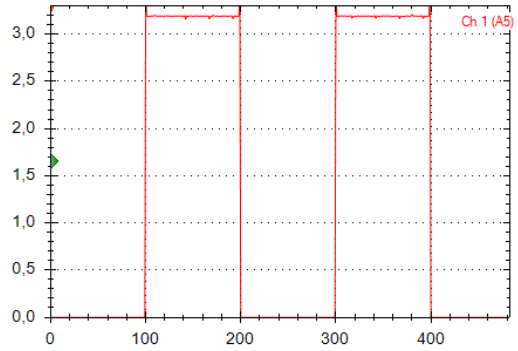
V této kapitole si ukážeme další způsob generování PWM sign. a to pomocí třídy *Ticker*. Třídy *PwmOut* a *Ticker* jsou implementovány pomocí čítačů. Cílem je ukázat, že ačkoli lze u třídy *Ticker* volat nějakou funkci s rozlišením na μs, nedosáhne se stejných výsledků jako při použití třídy *PwmOut*.

Příklad je analogický příkladu 3.8.2.1, v němž periodicky blikala LED, zatímco funkce main obsluhovala sériový port. V tomto příkladu se mění hodnota na pinu *PWM_PIN* s periodou *PWM_PULSEWIDTH*. Program je vidět dále.

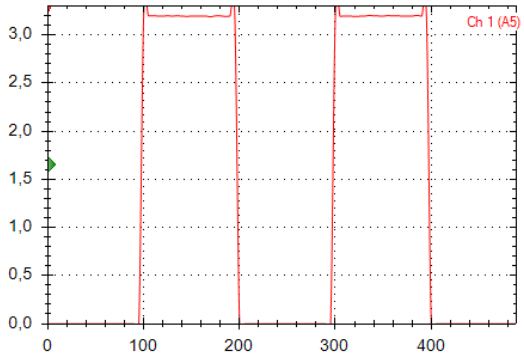
Pokud se takto vytvořeným programem generuje PWM sign. s periodou do cca 100 μs, pak je sign. v podstatě identický s tím, který je generovaný pomocí třídy *PwmOut* (porovnejte obr. 3.39 a obr. 3.40). Ve skutečnosti se nejedná o klasický PWM sign., ale o impulsní sign. se střídou 50 %.

Generujeme-li ale PWM sign. s rozlišením na jednotky/desítky μs, pak je potřeba využít třídu *PwmOut*, nikoli třídu *Ticker*. Porovnejte obr. 3.41 a obr. 3.42.

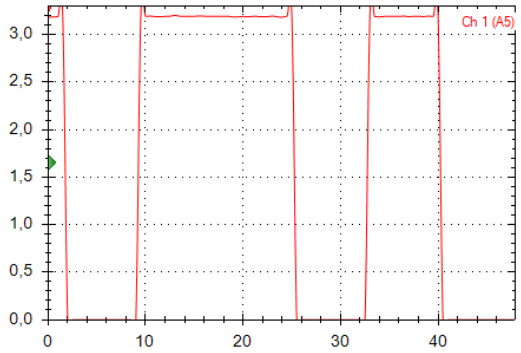
```
#include "mbed.h"
#define PWM_PIN PC_0
#define PWM_PULSEWIDTH 0.00001 //in s
DigitalOut pin(PWM_PIN);
Ticker ticker;
void flip(){
    pin = !pin;
}
int main(){
    ticker.attach(&flip, PWM_PULSEWIDTH);
    while(1){}
}
```



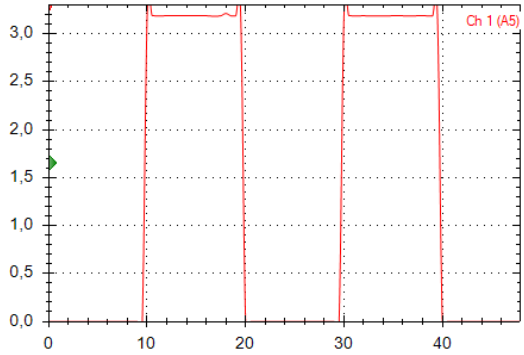
Obr. 3.39 Třída *Ticker*, perioda 100 *µs*



Obr. 3.40 Třída *PwmOut*, perioda 100 µs



Obr. 3.41 Třída *Ticker*, perioda 10 *µs*



Obr. 3.42 Třída *PwmOut*, perioda 10 µs

PWM Input

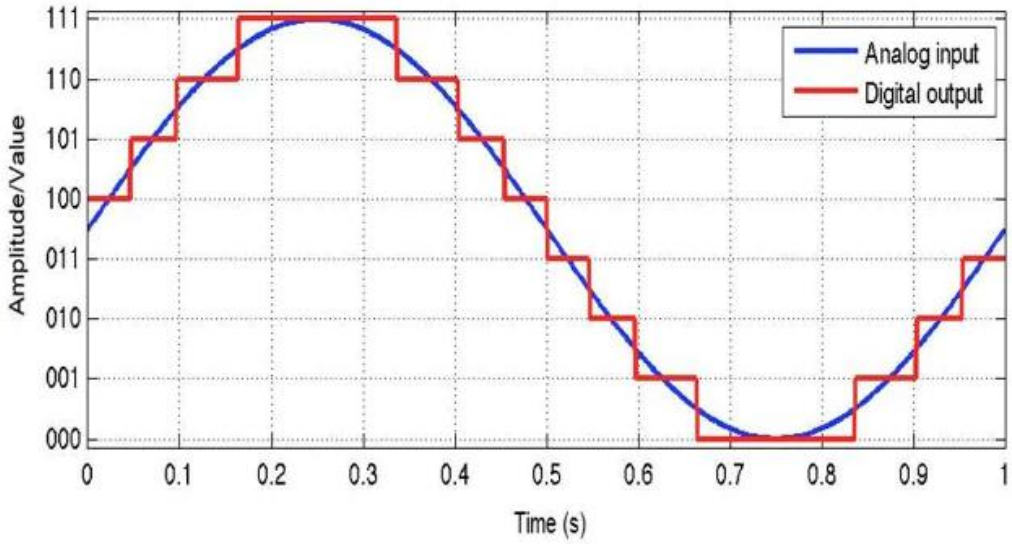
Na Nukleu je také možné nakonfigurovat pin pro příjem PWM sign. Při tomto procesu se využívá čítačů. Část čítače zvaná capture unit se dá pro tento účel nastavit do tzv. PWM Input módu. Může tak být měřena střída i perioda externího signálu. Mbed však třídou pro měření PWM sign. nedisponuje. [39]

3.10 Úvod do analogového signálu

Doposud jsme pracovali s digitálním vstupem a výstupem prostřednictvím tříd *DigitalIn* a *DigitalOut*. Každý pin patřící k nějaké bráně (viz kap. 3.3) lze nastavit jako digitální vstup/výstup. Následovalo seznámení se sériovým portem (kap. 3.6) a ladícími nástroji (kap. 3.7). V kap.3.5.1 je přesně řečeno, co se digitálním vstupem a výstupem rozumí (log. 1, nebo log. 0).

Svět okolo nás je spojitý. Výstupem senzoru měřícího nějakou fyzikální veličinu je spojité (tzv. analogové) napětí. Ne, že by konstantní napětí o velikosti 0 *V* nebo +3,3 *V* nebylo spojité, ale nabývá pouze diskrétní hodnoty (zmíněných 0 *V* nebo +3,3 *V*). Příkladem analogového sign. je sinusový sign. Jak takový sign. převést do počítače (do diskrétního světa)? Pomocí AD převodníku (angl. ADC). Obr. 3.43 názorně ukazuje, jak se analogový sign. převede do „jedniček a nul“ (viz osa y). O ADC i DAC pojednává následující kapitola.

Abychom takový signál mohli připojit na vstup Nuclea, musí se jednat o signál, jehož minimální a maximální hodnota (tzv. amplituda) leží v přípustných mezích (viz kap. 3.5.2). Uvažujme sinusový sign., který tomuto omezení odpovídá. Jeho časový průběh reprezentuje modrý sign. (Analog input) na obr. 3.43. Výstupem ADC je pak posloupnost čísel (viz ose y). ADC s rozlišovací schopností 3-bity tak rozliší 8 (2^3) možností. To už dá uživateli lepší orientaci o velikosti vstupního napětí.



Obr. 3.43 Vstupní a výstupní signál ADC (převzato z [9])

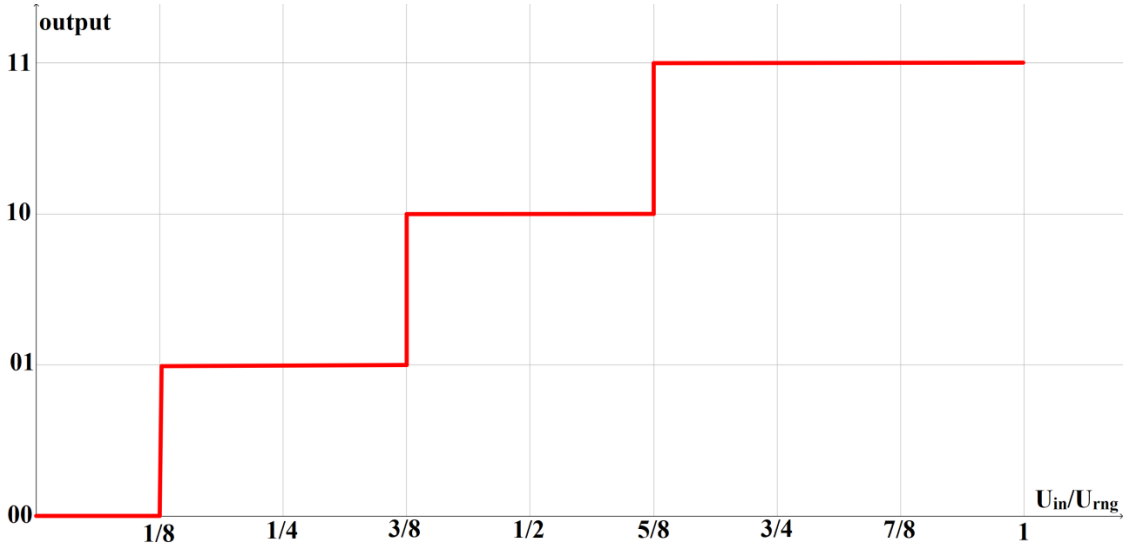
Pokud naopak chceme převést digitální sign. na analogový, pak Nucleo obsahuje dva 12-bitové DAC (Digital to Analog Converter) [10]. Mbed umožňuje s ADC a DAC pracovat prostřednictvím tříd *AnalogIn* a *AnalogOut* (viz dále). Nejdříve se blíže seznámíme s ADC a DAC.

3.10.1 Interní převodníky ADC a DAC

AD převodník (angl. AD converter, nebo jen ADC) slouží k převedení analogového (spojitého) sign. na sign. digitální (diskrétní). Tento převod probíhá ve dvou fázích. Zpočátku se provede diskretizace an. s. v čase: **vzorkování**. Následuje diskretizace an. s. v amplitudě: **kvantování**.

ADC může odebrat za daný čas pouze konečný počet vzorků. Tento údaj vyjadřuje vzorkovací frekvence a je to jeden z hlavních parametrů AD převodníku. Čím větší vzorkovací frekvence, tím sofistikovanější převodník, a to se samozřejmě projeví i na jeho ceně. Je však nutné zohlednit aplikaci, ve které se ADC použije. Např. není potřeba rychlý převodník, pokud nás zajímá měření teploty v místnosti, kde se napětí ze senzoru teploty zpracovává pouze několikrát za sekundu.

Rozdíl dvou sousedních hodnot je konstantní. Hodnota vzorku se tedy dá vyjádřit v kvantech, proto název kvantování. Po odebrání vzorku se mu musí přiřadit nějaká hodnota. Tato hodnota záleží na počtu bitů převodníku a vstupním rozsahu napětí. Např. pokud máme 2-bitový ADC se vstupním rozsahem napětí 0-1 *V*, pak výstup ADC je zobrazen na obr. 3.44. Na ose x je poměr vstupního napětí k rozsahu napětí, v tomto případě k 1. Vzorec platí pro ADC s rozsahem napětí od 0 do nějakého kladného napětí.



Obr. 3.44 výstup 2-bit ADC

STM32F303RE obsahuje čtyři 12-bitové ADC s postupnou aproximací, což představuje rozlišovací schopnost cca 0,8 *mV* [10]. Poznamenejme, že většina STM32 disponuje pouze jedním AD převodníkem. Výstup ADC je možné číst na 22 kanálech. Jeden pin se dá nastavit na jeden kanál. Jaký pin je možné nastavit na jaký kanál je uvedeno v tabulce 13 v [10] ve sloupci „Additional functions“. Doba odběru vzorku je nastavitelný parametr, který se dá nastavit od 1,5 hodinového cyklu[8] po 601,5 cyklu [12]. ADC nabízí značné množství parametrů, které se dají nastavovat. V mbedu však tyto parametry nastavovat nemůžeme, jsou zadány „napevno“ [1].

Pokud naopak chceme převést digitální sign. na analogový, pak STM32 obsahuje dva 12-bitové DAC [10].

3.11 Třída AnalogIn

Jak název napovídá, třída umožňuje nakonfigurovat pin pro příjem analogové signálu. O třídě *AnalogIn* pojednává kap. 4.5 v [1]. ADC nabízí značné množství parametrů, které se dají

[8] Aby periferie fungovala, je do ní potřeba pustit hodinový sign. Stejně tak je tomu i u ADC. Vysvětlení podstaty hodinového sign. se nachází v kap. 5.3.

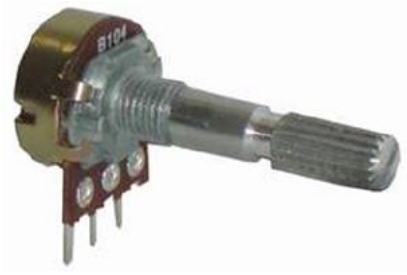
nastavovat. V mbedu však tyto parametry nastavovat nemůžeme, jsou zadány „napevno“ [1]. Kupříkladu doba odběru vzorku je v mbedu nastavena na 19,5 cyklu. Pevné nastavení parametrů má své nevýhody, např. problém s měřením napětí s vysokým vstupním odporem v úvodu kap. 4.5 v [1].

Třída *AnalogIn* obsahuje pouze dvě funkce: *read()* a *read_u16()* [1]. První vrací typ *float* v intervalu 0 až 1. Krajní hodnota 0 značí, že je na vstup připojeno napětí rovné GND tedy 0. Naopak 1 znamená, že je na vstup připojeno napětí rovné napájecímu napětí. Funkce *read_u16()* vrací číslo *uint16_t*, které odpovídá výstupu ADC přemapovaného na 16-bitové číslo.

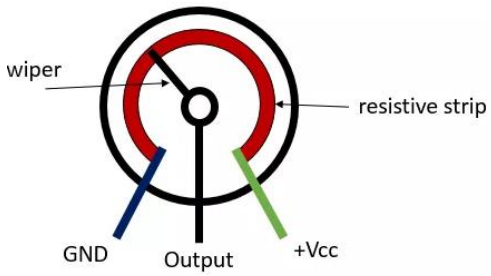
Jaký pin se může nastavit jako an. vstup? To už není tak jednoduché jako v digitálním případě. Je potřeba nahlédnout do upravených pinout diagramů, které jsou označené světlehnědou barvou a popisem *AnalogIn* na obr. 3.12, obr. 3.13 a obr. 3.14. Podle těchto obrázku se za tímto účelem dají nastavit piny PA_0 až PA_7, PB_2, PB_11 a PC_0 až PC_5. Poznamenejme, že v mbedu se dají nakonfigurovat pro analogový vstup ještě piny PB_0, PB_1 a PB_12 až PB_15, které však ve zmíněných obrázcích ani v oficiálním popisu desky NUCLEO_F303RE v [7] nejsou k tomuto účely vyznačeny [13]. Na základě experimentů bylo zjištěno, že hodnoty přečtené z těchto pinů bývají chybné, proto je nepoužívejme jako analogové vstupy.

Avšak **pozor**, je nutné myslet na již použité piny. V předešlé kapitole je psáno, že se na analogový vstup dají nakonfigurovat i piny PA_2, PA_3 a PA_5. Kde je problém? Piny PA_2 a PA_3 mohou být již použité na komunikaci po sériovém portu (viz kap. 3.6). Pokud si tohoto rozporu nevšimneme, mbed nás pravděpodobně neupozorní. Při nahrávání binárního souboru do Nuclea se v terminálu objeví MbedOS Error s hláškou „pinmap not found for peripheral“. Pin PA_5 je na Nucleu připojen na diodu, proto při měření může nastat situace, kdy by LED měla svítit, ale nesvítí a naopak. Pokud je toto chování nežádoucí, pak pin PA_5 není taktéž dobrá volba pro analogový vstup.

Nejjednodušší případ použití analogového vstupu je čtení výstupního napětí potenciometru[9]. Výstupní napětí se v případě otočného potenciometru reguluje pomocí otočného knoflíku (viz obr. 3.45). Kupříkladu uvažujme otočný potenciometru s výstupním napětím v rozsahu 0 až +3,3 *V*. Abychom mohli číst jiné hodnoty než krajní, je potřeba připojit výstup potenciometru do Nuclea jako an. vstup, nikoli digitální.



Obr. 3.45 Otočný potenciometr (převzato z [13])



Obr. 3.46 Dělič napětí (převzato z [31])

[9] Potenciometr je elektrotechnická součástka, která slouží jako proměnný dělič napětí. Více informací o této součástce se nachází v kap. 5.7 v [28].

3.11.1 Příklad: Základní použití třídy AnalogIn

Aby program neobsahoval pouze čtení daného pinu, ukážeme si na tomto příkladu, jaká hodnota se v mbedu skrývá za konstantnou označující daný piny. Jakou hodnotu tedy zastupuje např. konstanta PB_11? Piny jsou očíslovány popořadě. Začíná se bránou GPIOA. Piny PA_0 až PA_15 odpovídají číslům 0 až 15. Následuje brána GPIOB. Piny PB_0 až PB_15 odpovídají číslům 16 až 31 apod. Pomocí této konstanty tedy lze zjistit označení brány a číslo pinu. Tento převod děláme kvůli tomu, aby se ve výpisu dalo měnit označení pinu. Později však bude potřeba k důležitější funkci. Převod na označení brány a čísla pinu z konstanty ANALOG_INPUT vypadá:

```
char pin_port;
uint8_t pin_num = ANALOG_INPUT%16;
if (ANALOG_INPUT >= 32){
        pin_port = 'C';
} else {
        pin_port = ANALOG_INPUT >= 16 ? 'B' : 'A';
}
```

Celý program je v příloze 12.7. Výpis na procvičení funkcí *read()* a *read_u16()* stačí:

```
serial.printf("Voltage on pin P%c%lu:\tread()=%f\tread_u16()=%lu\r\n",
                pin_port, pin_num, an_in.read(), an_in.read_u16());
```

3.11.2 Příklad: Měření napětí vnitřní napětové reference

Přepočet získaných hodnot z kap. 3.11.1 na volty je

$$V_{IN} = V_{DDA}\frac{V_x}{FULL\_SCALE}, \tag{3.3}$$

kde V_x je napětí získané pomocí funkce *read_u16()*, V_{DDA} referenčního napětí ve voltech a *FULL_SCALE* maximální digitální hodnota výstupu ADC [12]. Např. pro ADC s 12-bitovým výstupem je *FULL_SCALE* roven 2^{12}-1=4095. Znamenáte to tedy, že pro nás je FULL_SCALE roven 4095? **Nikoli**! Funkce *read_u16()* nevrací maximálně číslo 4095, ale 65 535. Proč? 12-bitový výstup ADC je přemapován na 16-bitů (viz kap. 3.11). Proto je pro nás FULL_SCALE roven číslu 65 535. Počet výstupních bitů ADC je mimochodem také parametr, který se dá nastavovat [10]. Nastavení tohoto parametru však mbed neumožnuje, proto je FULL_SCALE konstantní.

Vzorec (3.3) obsahuje napájecí napětí V_{DDA}, které slouží pro napájení AD převodníku (tradičně 3,3 *V)*. Velikost napájecího napětí však může kolísat. Pokud chceme získat V_{IN} co nejpřesnější, je vhodné určit i toto napětí přesně. Skutečná hodnota V_{DDA} lze určit změřením napětí na vnitřní napětové referenci. Získání co nejpřesnějšího referenčního napětí je cílem tohoto příkladu. K tomuto účelu slouží vzorec

$$V_{DDA} = 3{,}3\frac{V_{REF\_CAL}}{V_{REF\_DATA}}, \tag{3.4}$$

kde V_{REF_CAL} je kalibrační konstanta a V_{REF_DATA} referenční napětí změřené pomocí ADC funkce *read_u16()*, avšak převedené na rozlišení ADC [12]. Funkce *read_u16()* tedy vrátí 16-bitové číslo, které je nutné binárně (konkrétně aritmeticky) posunout o 4 pozice vpravo:

```
uint16_t VREF_DATA = an_vref.read_u16();
VREF_DATA = VREF_DATA >> 4;
```

Zbývá zjistit kalibrační konstantu V_{REF_CAL}. Ta je uložena výrobcem na specifickém místě v paměti mikrokontroléru. Paměť obecně se nachází v adresním prostoru, který je rozdělen (odborně fragmentována) na adresy (angl. memory address). Vše si důkladně vysvětlíme v kap. 4.5. Kalibrační konstanta se tedy nachází na nějaké adrese. Na které zjistíme z obr. 3.47.

Calibration value name	Description	Memory address
V_{REFINT_CAL}	Raw data acquired at temperature of 30 °C V_{DDA}= 3.3 V	0x1FFF F7BA - 0x1FFF F7BB

Obr. 3.47 Kalibrační konstanta v STM32F303RE (převzatá tabulka 24 v [10])

Kalibrační konstanta je uložena na adrese 0x1FFF F7BA (podrobněji později). Momentálně stačí vědět, že se její hodnota přečte pomocí příkazu:

```
#define VREF_CAL *(uint16_t *)0x1FFFF7BA
```

Referenční napětí se v programu čte v nekonečném cyklu každé 2 *s*. Výsledný program:

```
#include "mbed.h"
#define VREF_CAL *(uint16_t *)0x1FFFF7BA
AnalogIn an_vref(ADC_VREF);
Serial serial(PA_2, PA_3);
int main() {
   uint16_t VREF_DATA;
   float V_DDA;
   while(1){
      VREF_DATA = an_vref.read_u16();
      VREF_DATA = VREF_DATA >> 4; //right shift by 4
      V_DDA = 3.3f*VREF_CAL/VREF_DATA; //equtation
      serial.printf("Reference voltage: %1.4f\n\r", V_DDA);
      wait(2);
   }
}
```

3.11.3 Příklad: Zpřesnění měřeného napětí

V kap. 3.11.1 bylo procvičeno použití funkcí *read()* a *read_u16()*. Výsledkem však bylo pouze číslo mezi 0 a 1 a *uint16_t*. Žádoucí je získat výsledek ve voltech (viz rovnice (3.3)). Pro přesné měření dosadíme za V_{DDA} rovnost (3.4), dostaneme

$$V_{in} = 3{,}3\frac{V_{REF\_CAL} \times V_x}{V_{REF\_DATA} \times (2^n - 1)}. \tag{3.5}$$

Rovnice (3.5) představuje vzorec, podle kterého lze vypočítat velikost vstupního napětí s ohledem na aktuální velikost napájecího napětí. Výsledný program vznikl spojením programů z kap. 3.11.1 a 3.11.2 (viz příloha 12.8).

3.11.4 Příklad: Měření teploty čipu

V této části změříme teplotu čipu pomocí vestavěného teplotního senzoru (angl. temperature sensor, dále TS), kterým mikrokontrolér disponuje. Výstupem TS je napětí, které lze měřit AD převodníkem. Výsledná teplota čipu se získá ze vzorce

$$TS\,[°C] = \frac{V_{25} - V_{TS}}{Avg\_Slope} + 25, \tag{3.6}$$

kde V_{25} je napětí při $25°C$, V_{TS} napětí získané ze senzoru a *Avg_Slope* průměrný sklon charakteristiky teplotního senzoru [12]. Charakteristika teplotního senzoru je téměř lineární, tedy změna napětí v čase je konstantní [12]. Hodnoty neznámých parametrů jsou na obr. 3.48.

Symbol	Parameter	Min	Typ	Max	Unit
T_L(1)	V_{SENSE} linearity with temperature	-	±1	±2	°C
Avg_Slope(1)	Average slope	4.0	4.3	4.6	mV/°C
V_{25}	Voltage at 25 °C	1.34	1.43	1.52	V
t_{START}(1)	Startup time	4	-	10	µs
T_{S_temp}(1)(2)	ADC sampling time when reading the temperature	2.2	-	-	µs

1. Guaranteed by design, not tested in production.
2. Shortest sampling time can be determined in the application by multiple iterations.

Obr. 3.48 Charakteristika teplotního senzoru (převzatá tabulka 89 z [10])

Úkolem tedy je vypočítat vzorec (3.6) v mbedu. Konstanty V_{25} a *Avg_Slope* jsou na obr. 3.48. Napětí V_{TS} se získá změřením napětí TS pomocí ADC. **Pozor** na jednotky napětí V_{TS}. Program, který každé 2 *s* měří teplotu TS a posílá ji přes sériový port do PC:

```
#include "mbed.h"
#define V_25 1430 //Min: 1.34; Typ: 1.43; Max: 1.52 [mV]
#define Avg_Slope 4.3f //Min: 4.0; Typ: 4.3; Max: 4.6 [mV/°C]
AnalogIn an_temp(ADC_TEMP);
Serial serial(PA_2, PA_3);
int main() {
   float TS_voltage, result;
   while(1){
      wait(2);
      TS_voltage = an_temp.read()*3300; //in ms!
      result = ((V_25-TS_voltage)/Avg_Slope)+25;
      serial.printf("Temperature: %.3f degree celsius\r\n", result);
   }
}
```

3.11.5 Možnosti generování náhodných čísel

V příkladu na měření reakční doby (viz kap. 3.8.3.1) jsme použili pro získání náhodného čísla funkci *rand()*. Pro zopakování, tato funkce vrací čísla z pseudonáhodné sekvence čísel a to ještě ze stejné pozice [1]. Vracená čísla tedy nejsou vůbec náhodná. Řešení pak nabízela funkce *srand()*, která umožnuje nastavení počátečního indexu.

Ke zvolení počátečního indexu je však opět potřeba náhodné číslo. S nově nabitými znalostmi z analogového světa víme, že lze takové náhodné číslo získat pomocí nezapojeného vstupu (viz kap. 4.5.3 v [1]). Po přečtení kap. 3.5.4 by měl čtenář tušit, že v případě nezapojeného vstupu se nenaměří 0 *V*. Na pinu bude díky šumu nějaké malé náhodné napětí, které našim účelům postačí [1].

3.12 Třída AnalogOut

Převod mezi digitální a analogovou podobou je potřeba oběma směry. Převod z analog-digital je náročnější úkol [18]. Tato část byla probrána v minulé kapitole, je tedy čas seznámit se s převodem digital-analog. Pin se nastaví jako analogový výstup třídou *AnalogOut*. Teorie o této třídě není uvedena v [1] jako doposud, ale v nové dokumentaci mbedu (viz [19]).

Jaké piny se dají nakonfigurovat jako analogový výstup? Každý by měl nyní vědět, kde takové informaci hledat. Zdůrazněme možnosti. Na prvním místě jsou upravené pinout diagramy (viz obr. 3.12, obr. 3.13 a obr. 3.14). Další možností je datasheet [10] a již zmíněná obsáhlá tabulka s definicemi pinů (viz obr. 3.20). V tomto případě odpověď najdeme hledáním pojmu DAC v poli Additional functions. Poznamenejme, že tento způsob je správný, avšak konkrétně v našem případě, kdy využíváme mbed, je lepší využít první způsob, jelikož tam je (měla by být) funkčnost pinu ověřena.

Poslední způsob, který byl již uveden, je náhled přímo do implementace v mbedu, v tomto případě konkrétně do souboru *PeripheralPins.c* (viz [13]). Zopakujme, že ve zmíněném souboru jsou definované piny, které se dají v mbedu využít k práci s periferiemi. Jako analogový výstup se dají nakonfigurovat pouze piny PA_4 a PA_5.

Třída AnalogOut poskytuje 3 funkce: *write()*, *write_u16()* a *read()*. První dvě jsou analogické funkcím *read()*, *read_u16()* z kap. 3.11. Funkci *read()* tedy předáváme číslo mezi 0 a 1 odpovídající procentům z referenčního napětí. Funkci *read_u16()* naopak číslo od 0x0 po 0xFFFF. [19]

3.12.1 Příklad: Program pro řízení svítivosti LED pomocí DAC

Úkolem příkladu je měnit svítivost interní LED. Tato změna může být realizována pomocí dvou tlačítek, kde jedno zvětšuje a druhé zmenšuje výstupní napětí, nebo pomocí sériové komunikace, kde stejnou funkci zastupují dvě klávesy. Je vhodné vyzkoušet si obě možnosti. Další možností je kombinace obou zmíněných, tzn. použití interního/externího tlačítka a nějakého znaku klávesnice, případně využití potenciometru. Pokud čtenář má k dispozici potenciometr, určitě by si měl zkusit i tuto alternativu.

Následuje varianta se zvětšováním/zmenšováním výstupního napětí na uživatelské LED pomocí tlačítek „s“ a „d“, kde tlačítko „s“ napětí přidává a tlačítko „d“ napětí ubírá a to

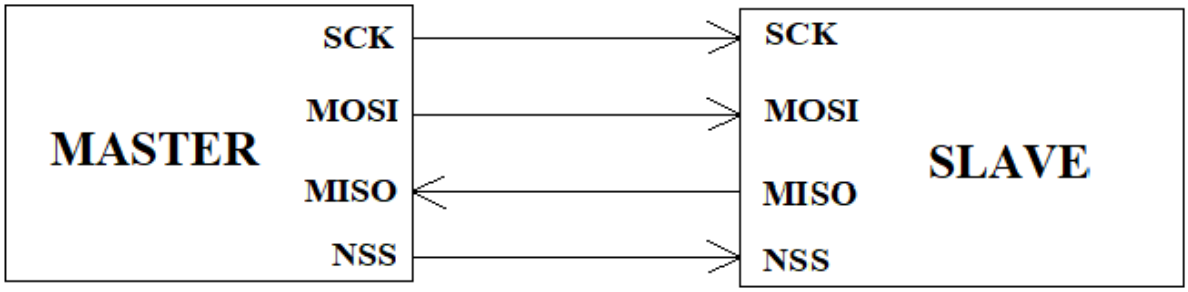
o fixní krok v programu definovaný jako konstanta STEP. Celý program je uveden v příloze 12.9, hlavní smyčka vypadá takto:

```
if (serial.readable()){
    sign = serial.getc();
    if (sign == 's' || sign == 'd'){
        if (sign == 's'){ //increase output voltage
            value += STEP;
        } else { //'d' -> decrease output voltage
            value -= STEP;
        }
        led.write(value);
        serial.printf("\rOutput voltage: %1.3f V.", led.read()*3.3f);
    }
}
```

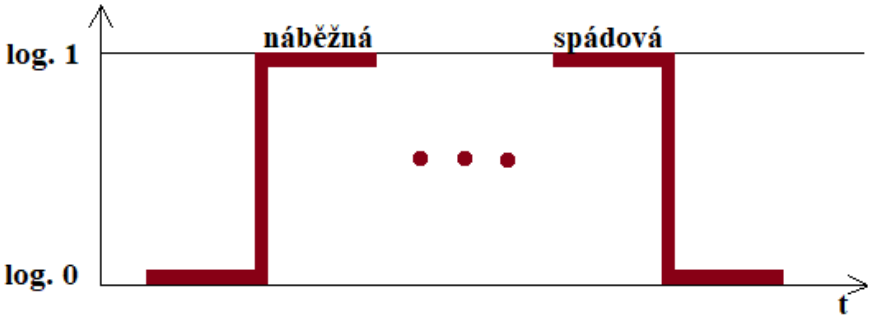
Poznamenejme, že na příkladu lze názorně demonstrovat chování diody. Pokud je napětí na LED cca 1,6 *V*, pak začne slabě svítit. Tato hranice se nazývá prahové napětí. Zvyšováním napětí z této hranice svítivost rychle roste. V případě zvětšování výstupní napětí DAC nad 2,7 *V* se záření LED téměř nemění. Toto napětí je rozděleno mezi LED a rezistor. To znamená, že není pouze na LED (to by tato součástka nevydržela, shořela by).

Komunikační periferie

STM32 disponuje pěti typy komunikačních periferií. Pro komunikaci mezi integrovanými obvody na stejném plošném spoji (angl. PCB) má SPI a I^2C. Jelikož základní princip SPI není složitý, seznámíme se s ním v další zajímavosti. Ke komunikaci mezi různými moduly slouží sběrnice CAN a pro komunikaci s PC periferie USB a USART. [39]



Obr. 3.49 Standardní rozhraní SPI



Obr. 3.50 Ideální náběžná a spádová hrana

SPI

SPI (Serial Peripheral Interface) je rychlá komunikační periferie určená k synchronní sériové komunikaci mezi integrovanými obvody po společné sběrnici. Základní zjednodušené schéma full-duplex[1] komunikace mezi dvěma integrovanými obvody, kde jeden plní roli Mastera a druhý roli Slave je na obr. 3.49. [12]

Význam zkratek jednotlivých pinů (dle [12]):

- SCK – výstupní hodiny (Serial Clock output) pro Mastera a vstupní hodiny (Serial Clock input) pro zařízení typu Slave,
- MOSI – Master data vysílá a Slave data přijímá (Master Out / Slave In data),
- MISO – Slave data vysílá a Master data přijímá (Master In / Slave Out data),
- NSS – výběr daného zařízení (Slave select), případně může být použit i pro synchronizaci dat nebo detekci konfliktu mezi několika Mastery.

Princip komunikace je tedy následující. Master řídí celou komunikaci mezi zařízeními. Do všech zařízení typu Slave generuje hodinové sign. (SCK). Pokud chce s daným Slavem komunikovat, pak ho pomocí pinu NSS určí, Slave to pozná a začíná komunikace. Master pak data posílá po pinu MOSI, zatímco Slave po pinu MISO.

Nucleo F303RE obsahuje 4 SPI jednotky, pomocí nichž lze komunikovat rychlostí až 18 *Mbit/s* v režimu full-duplex nebo half-duplex. Rychlost jednotlivých SPI jednotek je závislá na sběrnici, na které je připojena, protože některé jsou připojené na APB2, zatímco jiné na APB1. Rychlost na APB2 je max. 72 *MHz*, kdežto na APB1 36 *MHz*. Tuto vlastnost lze pozorovat i na hlavním obrázku s vnitřní strukturou (viz obr. 4.6). [10]

3.13 Třída InterruptIn

Poslední třída uvedená v [1] je třída *InterruptIn*, konkrétně se jí zabývá kap. 4.7. V této práci bylo již o přerušení (angl. interrupt) také zmíněno v kap. 3.8. Konkrétně např. třída *Ticker* (viz kap. 3.8.2) vyvolává přerušení (angl. interrupt) procesoru po určité periodě (viz pozn. p. č. 6).

Zopakujme, že přerušení je proces, při kterém je procesor „přerušen“ v jeho chodu. Ten následně přeruší vykonávání dané části programu a zkontroluje, „kdo“ ho přerušil a na základě toho vykoná požadované akce (tzv. obsluhu přerušení). Třída *InterruptIn* umožňuje nakonfigurovat pin jako digitální vstup s vlastností hardwarového přerušení při náběžné nebo spádové hraně log. hodnoty napětí [1]. Ideální náběžná a spádová hrana je vyobrazena na obr. 3.50.

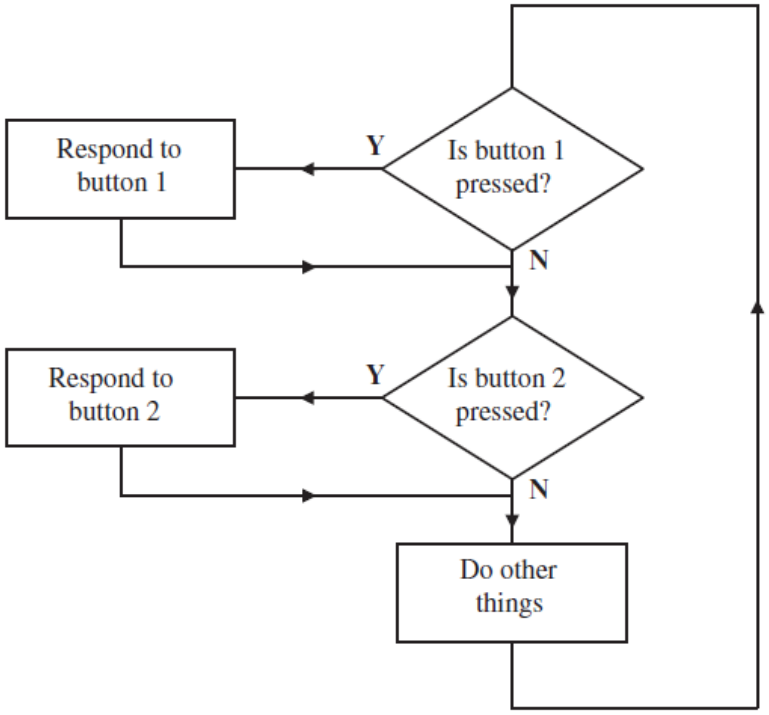
3.13.1 Motivační příklad

Příklad vychází z kap. 9.2.1 v [18]. Proč je vhodné používat přerušení? Doposud nebyl ukázán žádný příklad, ve kterém by se četlo několik vstupů zároveň, protože se pro zjednodušení používalo pouze jediné tlačítko. Se dvěma vstupy by se v projektu pracovalo, pokud by se využívalo tlačítka a sériové komunikace, nebo jednodušeji dvou tlačítek. Za účelem zjišťování stavu obou tlačítek by docházelo k jejich čtení a to v každém

cyklu nekonečné smyčky (viz obr. 3.51). Tento způsob se nazývá **polling**. V jednoduchých projektech, jako jsou ty naše, může být dostačující, avšak pro větší projekty jeho použití není vhodné.

Představme si situaci, kdy za pomoci pollingu obsluhujeme 20 tlačítek. Ve většině případů, uvažujeme-li rychlé vykonávání hlavní smyčky, se žádná vstupní hodnota nezmění, tedy velké množství smyček proběhne „zbytečně“. Navíc program může strávit čas reagováním na stisk tlačítka, která nejsou tolik důležitá (např. button 1, button 2 a button 3), zatímco je aktivní také tlačítko signalizující velkou poruchu (např. button 20). K důležitému tlačítku se tedy program dostane až se značným zpožděním. Další nevýhodou je skutečnost, že procesor značnou část nekonečné smyčky věnuje obsluze vstupů a nikoli vykonávání programu.

S využitím přerušení lze všechny zmíněné negativní vlastnosti odstranit. Přerušení a jejich obsluha se nastaví pouze jednou. Procesor se tak o vstupy nemusí starat (neztrácí čas). Navíc je u přerušení možné zvolit jejich prioritu. Zopakujme, že mbed umožňuje nastavit přerušení pomocí třídy *InterruptIn* a to na spádovou nebo náběžnou hranu, prioritu (ani jiné vlastnosti) však nastavovat neumožňuje. Pro účely této práce to však ani není potřeba.



Obr. 3.51 Čtení dvou tlačítek s využitím pollingu (převzato z [18])

3.13.2 Funkce třídy InterruptIn

Použití třídy *InterruptIn* je podobné třídě *Ticker* (viz kap. 3.8.2) a třídě *DigitalIn* (viz kap. 3.5). Veškeré informace o použití této třídy jsou uvedeny v kap. 4.7 v [1]. Shrňme bodově význam jednotlivých funkcí:

- *mode()* – nastavení módu pinu,
- *read()* – čtení aktuální log. úrovně na daném pinu,
- *rise(), fall()* – stanovení, zda se má reagovat na náběžnou nebo spádovou hranu,

- *disable_irq(), enable_irq()* – zrušení či opětovné povolení obsluhy přerušení.

Zdůrazněme, že obsluha přerušení nesmí mít vstupní argumenty ani naopak vracet nějakou hodnotu. Navíc nemůže obsahovat, stejně jako obsluha přerušení u třídy *Ticker* či *Timeout*, nekonečné cykly, časově náročné operace či dynamickou alokaci paměti. Při použití globálních proměnných je vhodné jejich inicializace s klíčovým slovem **volatile**. [1]

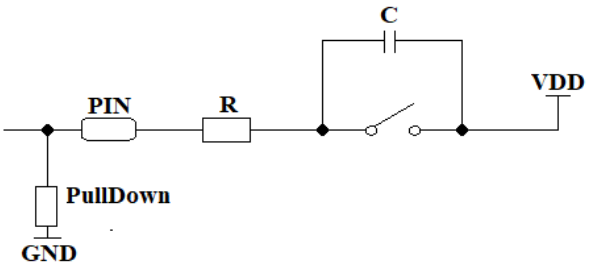
3.13.3 Příklad: Základní použití třídy InterruptIn s využitím dvou externích tlačítek

Vytvořme program, ve kterém se využijí dvě externí tlačítka. Tlačítka budou využívat třídu *InterruptIn*, přičemž jedno bude na náběžnou hranu přičítat jedničku do nějaké proměnné (např. *number*) a druhé na spádovou hranu jedničku odečítat. Hodnota se bude periodicky (např. s periodou 100 *ms*) aktualizovat v terminálu s využitím s. portu.

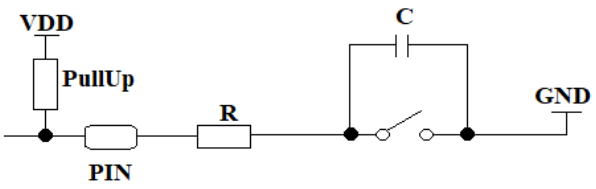
Kvůli debouncingu (viz kap. 3.5.5) je nutné paralelně přidat kapacitory o velikosti řádově stovek *nF* (např. 100 *nF*), tedy o stejné velikosti jako je u interního tlačítka (viz obr. 3.23). Jelikož se jedná o tlačítko, je také nutné využít pull rezistory (viz kap. 3.5.4), konkrétně PullUp a PullDown.

Na obr. 3.52 je zapojení, ve kterém se standartně na pinu nachází log. 0 a po stlačení tlačítka log. 1. Proto se na vstupu objeví náběžná hrana a po uvolnění spádová hrana. V případě druhého zapojení z obr. 3.53 se standartně na pinu nachází log. 1 a po stlačení tlačítka log. 0. Jedná se tedy o inverzní chování. Obě tlačítka je možné zapojit oběma možnostmi, avšak záleží na tom, zda chceme reagovat na stlačení tlačítka, nebo na jeho následné uvolnění.

Následuje program, jenž plní zadanou úlohu pro případ zapojení prvního tlačítka (inkrementujícího na náběžnou hranu) jako na obr. 3.52 a druhého tlačítka (dekrementujícího na spádovou hranu) jako na obr. 3.53. Jedná se tedy o zapojení, kdy se hodnota inkrementuje/dekrementuje ihned po stisku tlačítka. Poznamenejme, že mezera ve funkci *printf()* před „\r“ je nutná. Proč? Vyzkoušejte si a sledujte chování.



Obr. 3.52 PullDown rezistor a tlačítko připojené na V_{DD}



Obr. 3.53 PullUp rezistor a tlačítko připojené na GND

```
#include "mbed.h"
InterruptIn in1(PC_0, PullDown); //PC_0 as an incremental pin
InterruptIn in2(PC_1, PullUp); //PC_1 as an decremental pin
Serial serial(PA_2, PA_3);
volatile int number = 0; //global variable
void increment(){
   number++;
}
void decrement(){
   number--;
}
int main() {
   in1.rise(&increment);
   in2.fall(&decrement);
   serial.printf("PRESS THE EXTERNAL BUTTONS TO INCREMENT/DECREMENT
         the number\n\r");
   while(1){
      serial.printf("number: %d \r", number);
      wait(0.1);
   }
}
```

4 Vnitřní struktura mikrokontrolérů

Doposud byly probrány všechny třídy začleněné do práce [1]. Nejednalo se o veškeré třídy, které mbed poskytuje, avšak o jakýsi souhrn „základních“ tříd. Mbed navíc nabízí hlavně třídy pro práci s komunikačními periferiemi. Veškerý přehled tříd je v [23].

Tímto tedy skončilo vysvětlování poskytovaných tříd. Mbed, jak se postupně ukazovalo, některé věci zkrátka neumí, avšak jako nástroj pro seznámení s mikrokontroléry je velice vhodný. Abychom věci, které mbed nepodporuje (nebo i ty, které podporuje), mohli sami naimplementovat, je nutná orientaci ve vnitřní struktuře mikrokontroléru. Nejen touto otázkou se zabývá tato kapitola.

Nová kapitola začíná ukázkou, jak hledat zdrojové soubory přímo na stránkách mbedu a seznámením s vestavěnými systémy. Následuje vnitřní struktura mikrokontroléru, což zahrnuje sběrnice, adresní prostor a paměti. Na závěr se blíže seznámíme s pojmem periferie, na který navážeme v další kapitole.

4.1 Zdrojové soubory v mbedu

Občas se může naskytnout otázka, jak má vlastně mbed určité záležitosti definované. Za tímto účelem je vhodné být schopen dohledat určité zdrojové soubory. Kupříkladu již byla zmíněna práce se souborem *PeripheralPins.c* (viz [13]), ve kterém jsou nadefinované piny pro práci s periferiemi ADC, DAC, I^2C, PWM, USART, SPI a CAN.

Základní adresář je [24], v něm pokračujeme na „mbed-dev“ a dále „targets“. V otevřeném adresáři se nacházejí všechny mikrokontroléry, které mbed podporuje. Cílem je najít **STM32F303RE**, proto se pokračuje do adresáře „TARGET_STM“. Součástí následujících adresářů jsou také .c a .h soubory, ve kterých jsou hledané informace. Následuje adresář, v němž jsou mimo .c a .h souborů také různé typy podporovaných mikrokontrolérů firmy STMicroelectronics, volíme „TARGET_STM32F3“.

V adresáři „device“ se nachází HAL a LL programy, jenž mbed umožňuje využít. Více o těchto programech si řekneme v kap. 6.2. Pokračováním do „TARGET_STM32F303xE“ a dále „TARGET_STM32F303RE“ se objeví adresář stejný jako na obr. 4.1.

V cílovém adresáři se nachází soubor *PeripheralNames.h* obsahující symbolické rozčlenění pomocí datového typu *enum* a již několikrát zmíněný soubor *PeripheralPins.c*. Dále soubor *PinNames.h,* v němž se mmj. přiřadí symbolickým názvům pinů jejich skutečné označení (např. LED1 je PA_5, USER_BUTTON PC_13). Poslední je velmi důležitý soubor *system_clock.c* obsahující inicializaci hodin, což, jak se dále ukáže, je klíčové pro fungování jakékoli periferie.

Name	Size	Actions
[up]		
PeripheralNames.h	2892	Revisions Annotate
PeripheralPins.c	23115	Revisions Annotate
PinNames.h	6349	Revisions Annotate
system_clock.c	8802	Revisions Annotate

Obr. 4.1 Práce se zdrojovými soubory (převzato z [27])

4.2 Vestavěné systémy

Celá kapitola vychází z kap. 1.1 z [18]. Stolní počítač nebo notebook není potřeba nikomu představovat. V samotném srdci těchto zařízení se nachází **mikroprocesor** (angl. microprocessor), též označovaný jako centrální procesorová jednotka (angl. central processing unit, zkratka **CPU**). Jedná se o velmi komplikovaný elektrický obvod, který obstarává základní funkce počítače. CPU se nachází na jednom integrovaném obvodu (angl. integrated circuit, zkratka **IC**). CPU může vykonávat aritmetické a logické operace v digitálním elektrickém obvodu zvaném aritmetická logická jednotka (angl. arithmetic logic unit, zkratka **ALU**). Tato jednotka se nachází uvnitř CPU.

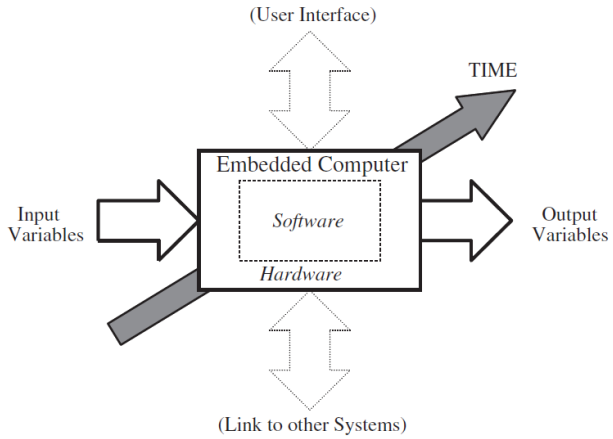
Méně známé užití mikroprocesorů je v zařízeních, které s počítáním nemají „nic společného“. Jedná se o domácí spotřebiče jako je myčka, pračka, lednice, ale i mixér či toustovač. Mikroprocesor v nich plní funkci řízení. Takováto zařízení se souhrnně označují jako **vestavěné systémy**[10] (angl. embedded systems).

Užití vest. syst. je všeobecně velmi široké. Rozšířené jsou v domácnostech, v automobilovém průmyslu a obecně v průmyslu, kde je potřeba řízení, automatizace, robotika apod. Příkladem vest. syst. je bankomat, jehož grafické znázornění je na obr. 4.4. Obecné schéma vest. syst. vidíme na obr. 4.2.

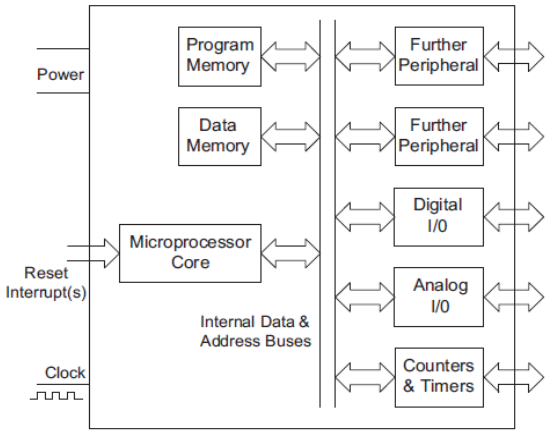
Vest. syst. vykonává program, jenž je permanentně uložen v paměti. Jistě si každý všiml vlastnosti, že pokud odpojíme Nucleo od napájení (od PC) a následně opět připojíme, program se znovu vykonává (např. zase bliká LED apod.). Více o paměti mikrokontroléru v kap. 4.6. Na rozdíl od stolních počítačů či notebooků, ve vest. syst. běží najednou pouze jeden program.

Další významná proměnná je čas, proto prochází v obr. 4.2 skrz blok reprezentující vest. syst. Proč je tak významný? Jeho význam není pouze pro měření času, ale hlavně umožňuje práci s periferiemi a komunikaci v rámci vnitřního (i vnějšího prostřednictvím komunikačních periferií) prostředí. Více se dozvíme dále v kap. 5.3.

[10] Dále jen zkráceně vest. syst.



Obr. 4.2 Obecné schéma vest. syst. (převzato z [18])

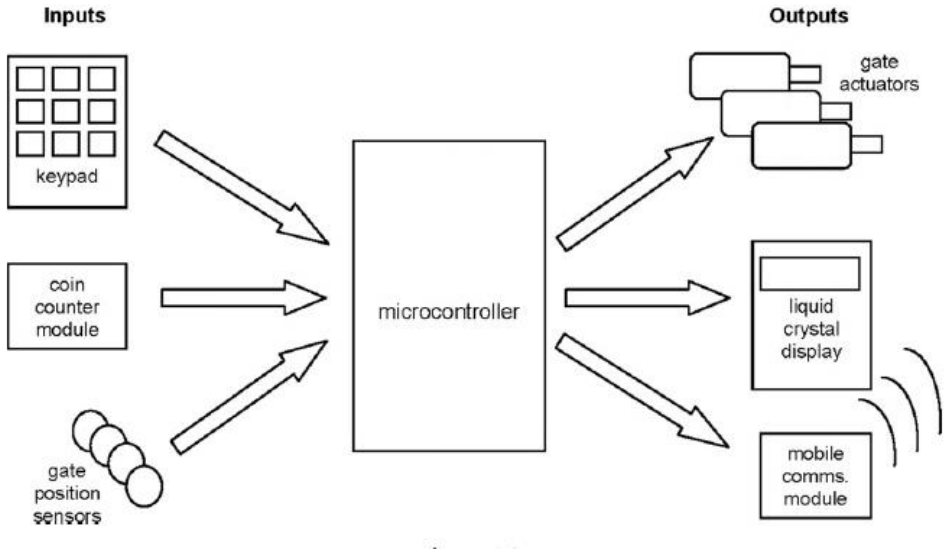


Obr. 4.3 Schéma mikrokontroléru (převzato z [18])

Rozdíl mezi mikroprocesorem a mikrokontrolérem

Pro práci s třídami jsme využívali mikrokontrolér STM32F303RE. Jaký je ale mezi mikrokontrolérem a mikroprocesorem rozdíl?

V rámci kap. 4.2 bylo řečeno, že mikroprocesor je IC s CPU. Mikrokontrolér je také IC s CPU, ale má **navíc** paměť a periferie viz schéma mikrokontroléru na obr. 4.3. O paměti mikrokontroléru později v kap. 4.6. Periferie zahrnují digitální a analogové vstupy/výstupy, sériový port, čítače, časovače atd. S těmito periferiemi jsme se již v rámci práce s mbed třídami seznámili.



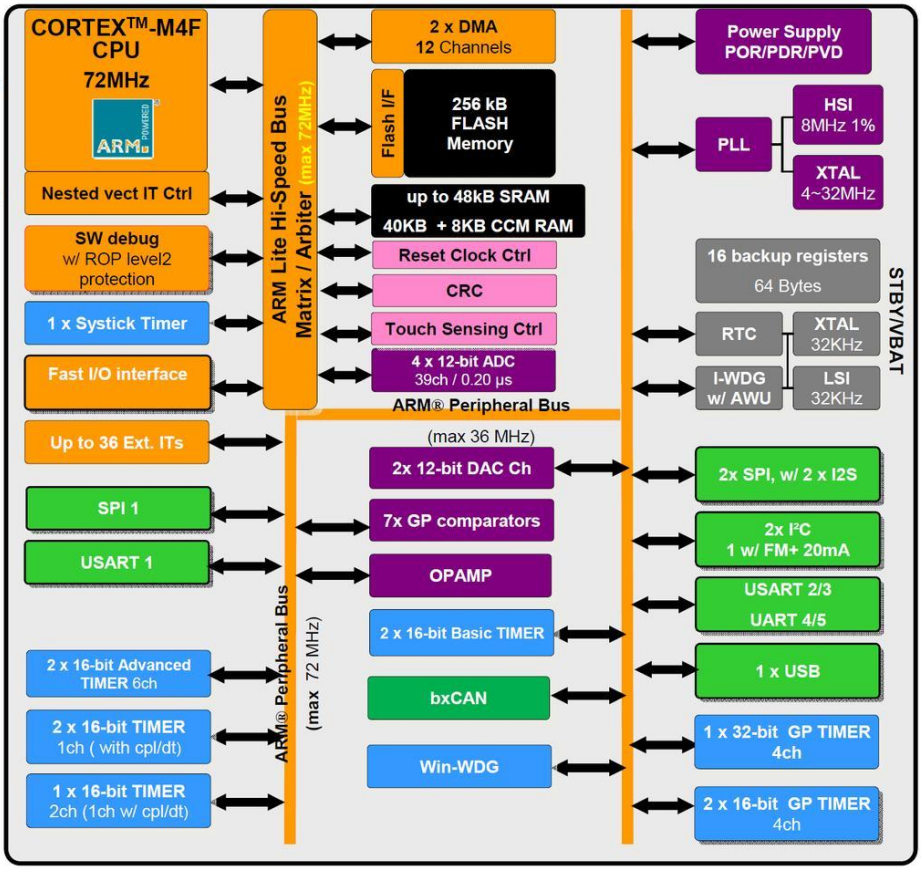
Obr. 4.4 Bankomat jako příklad vest. syst. (převzato z [18])

4.3 Vnitřní struktura mikrokontroléru STM32F303RE

Přesný název jádra **STM32F303RE** je „ARM®32-bit Cortex®-M4 CPU“, tzn. že patří do rodiny jader M4. Připomeňme, že SMT32F303RE je pro účely této práce označováno jako Nucleo. Na obr. 4.5 je blokové schéma mikrokontroléru z této rodiny, i když se nejedná přímo o blokové schéma **STM32F303RE** (např. velikost paměti by měla být 512 *kB* nikoli 256 *kB*). Až na některé hodnoty se však jedná o stejné schéma.

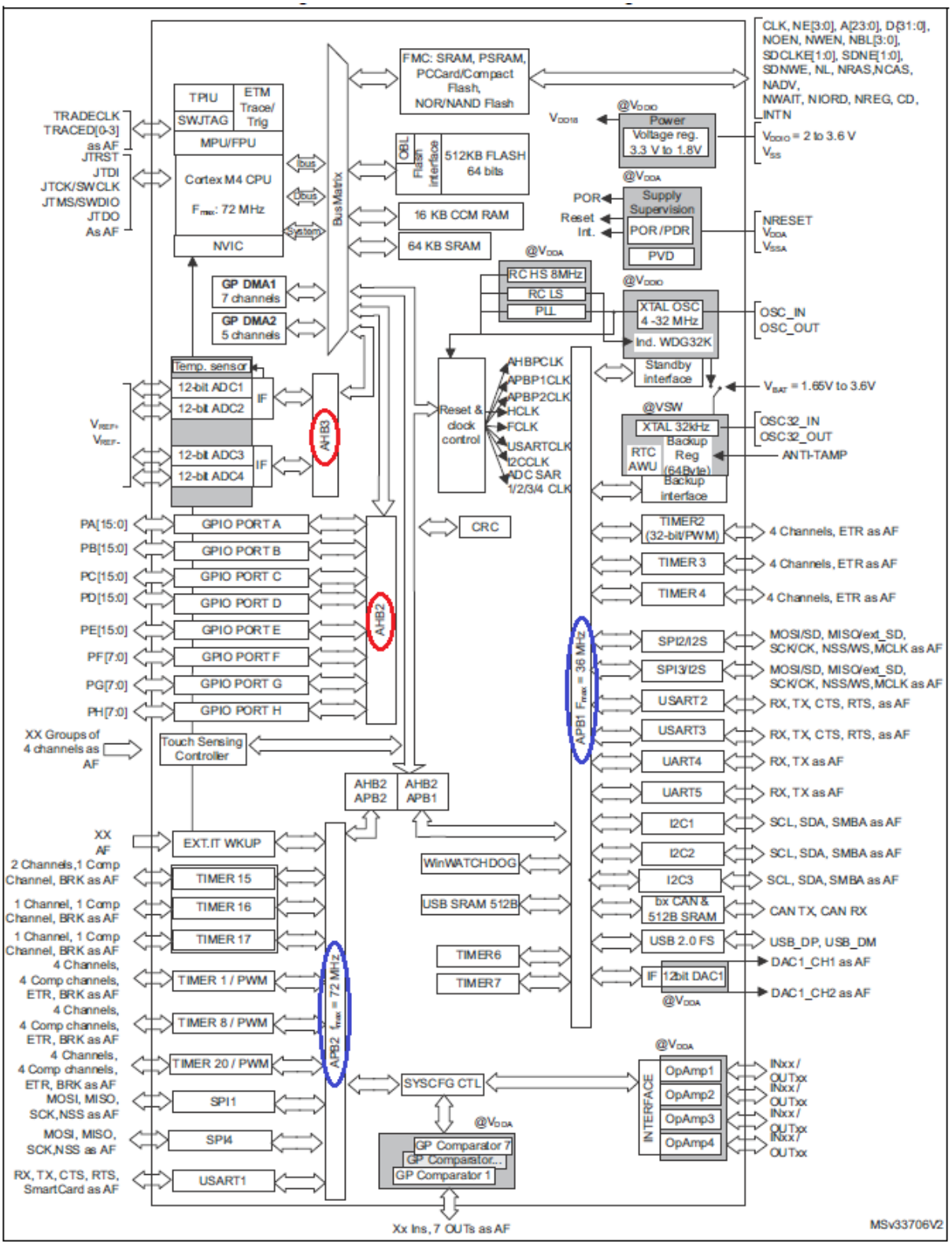
S většinou bloků jsme se již nějakým způsobem setkali, avšak na obr. 4.5 se nachází také doposud neznámé bloky. Následuje zběžný přehled některých z nich:

- PLL, HSI, XTAL, RTC, I-WDGw/AWU a LSI: jedná se obvody obecně pracující s hodinovým signálem (více o hod. sign. v kap. 5.3),
- OPAMP (operational amplifier): operační zesilovač,
- COMP (comparator): komparátor,
- Touch sensing controller: umožňuje připojit kapacitní zařízení,
- CRC (cyclic redundancy check): jedná se o jednotku umožňující rychlý výpočet tzv. kontrolního součtu používaného pro validaci dat,
- Reset Clocl Ctrl: obvod kontrolující generované hodiny,
- Flash, SRAM, CCM RAM: tři druhy pamětí (viz kap. 4.6),
- DMA: (direct memory access) velmi sofistikovaná část umožňující posílat/přijímat data z/do paměti,
- Ext. ITs (external interrupts): více o interruptech v kap. 3.13,
- Nested vect IT Ctrl: jednotka sloužící k práci s interrupty.



Obr. 4.5 Blokový diagram pro Cortex M4 (převzato z [36])

Smyslem obr. 4.5 je získání nadhledu nad vnitřní strukturou, jelikož představení přímo následujícího obrázku by „mohlo dojít k panice“. Na obr. 4.6 je zobrazena konkrétní vnitřní struktura mikrokontroléru STM32F303RE.



Obr. 4.6 Blokový diagram mikrokontroléru SMT32F303RE (převzato z [10])

4.4 Interní sběrnice

Sběrnice (angl. bus) je soustava vodičů, která přenáší data nebo signál stejného charakteru. Vodiče mohou mít několik funkcí, např. slouží k adresaci či přenosu dat). Sběrnice musí mít řídící jednotku tzv. řadič (master), kterému jsou ostatní části podřízeny (slave). Obecně se sběrnice rozdělují podle provozu (synchronní a asynchronní), uspořádání (sériová, paralelní či sérioparalelní) podle směru přenosu (jednosměrné a obousměrné) apod. [34]

Hlavní systém Nucleo SMT32F303RE se skládá z pěti bus masterů:

- Cortex® -M4 core **I-bus**,
- Cortex® -M4 core **D-bus**,
- Cortex® -M4 core **S-bus**,
- GP-DMA1 a GP-DMA2 (general-purpose DMAs),

které lze nalézt na obr. 4.6 v levém horním rohu. Všechny tyto řadiče jsou připojeny na sběrnici BusMatrix. **BusMatrix** tedy spojuje procesor s externími sběrnicemi. Následná komunikace s externí sběrnicí je uskutečněna s využitím jednoho z výše uvedených řadičů (masterů). [12]

Procesory Cortex M4 mají **Harvardskou architekturu**. To znamená. že mají fyzicky oddělenou paměť pro program a data a navíc pro obě části mají vlastní sběrnici (I-bus a D-bus). Program se tedy do jádra přenáší ve formě **instrukcí** (viz obr. 3.27) pomocí sběrnice **I-bus**. Odkud tyto instrukce jdou? Z paměti, konkrétně z paměti Flash, SRAM nebo CCM RAM. **Data** jsou ze stejných pamětí přenášena pomocí sběrnice **D-bus**. Zmíněná spojení vidíme na obr. 4.7. [12]

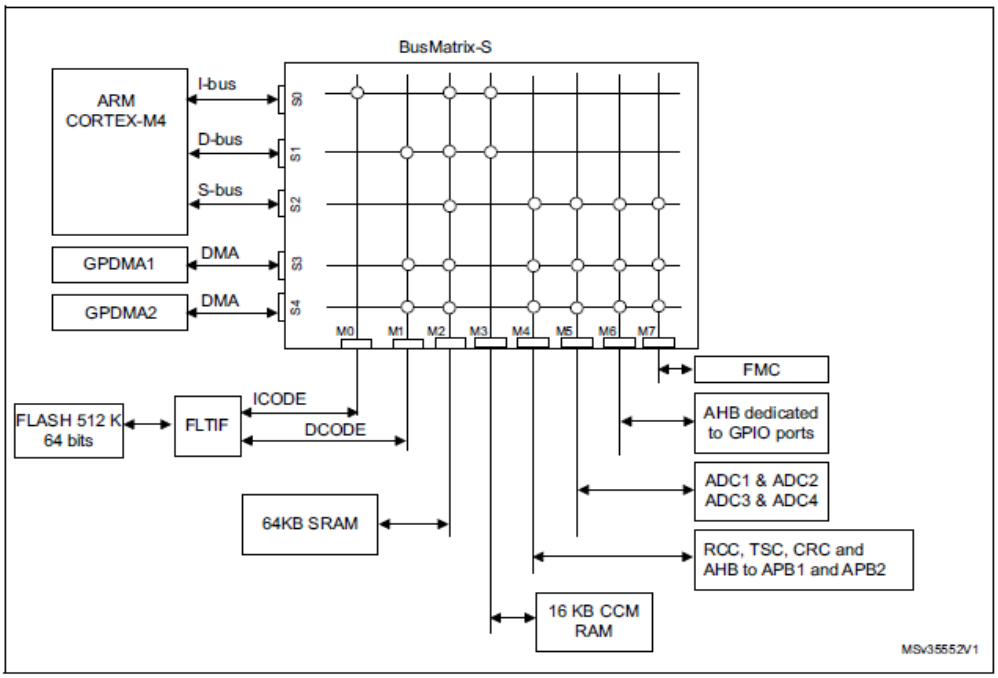
Další velmi důležitou sběrnicí je **systémová** sběrnice **S-bus**, která se používá k přístupu k datům umístěným v **periferní oblasti** nebo paměti SRAM (viz obr. 4.7). Sběrnice S-bus je tedy velmi důležitá pro práci s periferiemi. Ta se realizuje pomocí sběrnic **AHB** a **APB**, které jsou na obr. 4.6 zvýrazněny červeně a modře. [12]

Zdůrazněme, že práce s příslušnými periferiemi prostřednictvím APB2 může být rychlejší, protože maximální frekvence hodin na **APB2** je **72 *MHz***, zatímco na **APB1** pouze **36 *MHz***. Proto se APB2 označuje jako „high speed APB“ a APB1 jako „low speed APB“. [12]

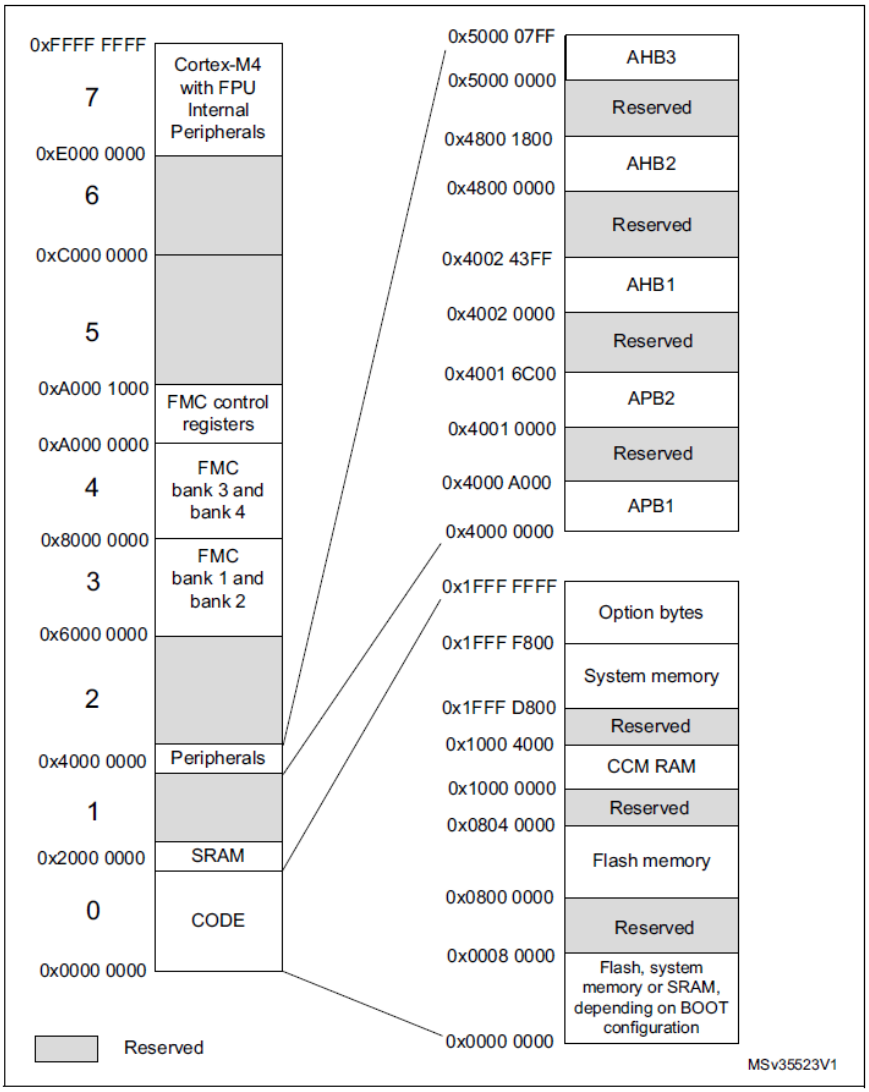
4.5 Adresní prostor

Následuje zcela nový pohled na práci s mikrokontroléry. Veškeré programování mikrokontrolérů je úzce spjato s adresním prostorem. Program, data, registry (o nich více v kap.) a I/O porty jsou organizovány do stejného lineárního 2^{32} ~ **4 *GB*** adresového prostoru, jehož základní rozčlenění vidíme na obr. 4.8. Tento prostor je fragmentován do 8 částí podle toho, za jakým účelem je daná část použita (např. část 0 v rozsahu adres 0x0000 0000 až 0x2000 0000 je vyhrazena sekci „CODE“). Uvedeny jsou vždy hraniční adresy (angl. boundary addresses). [12]

Adresa je tedy 32 bitové číslo (8 hexadecimálních číslic), které ukazuje v adresním prostoru na jeden **bajt**. Na adresu je možné zapisovat, případně z ní číst, to však neplatí pro každou adresu. Možné jsou všechny varianty, tzn. povolené obě, pouze jedna nebo ani jedna možnost.



Obr. 4.7 Architektura sběrnice BusMatrix (převzato z [12])



Obr. 4.8 Paměťový prostor SMT32F303RE (převzato z [10])

Přehledné rozdělení paměťového prostoru podle hraničních adres se v **STM32F303RE** nachází v tabulce 3 v [12], jejíž část vidíme na obr. 4.9. Tato tabulka je explicitně uvedena pro úplnost a také kvůli odkazu na ni v následující kapitole. V případě hledání rozsahu hraničních adres určité části paměťového prostoru je tato tabulka správná volba.

Bus	Boundary address	Size (bytes)	Peripheral	Peripheral register map
	0x1000 0000 - 0x1000 3FFF	16 K	CCM SRAM	-
	0x0808 0000 - 0x0FFF FFFF	~128 M	Reserved	
	0x0800 0000 - 0x0807 FFFF	512 K	Main Flash memory	-
	0x0008 0000 - 0x07FF FFFF	~128 M	Reserved	
	0x0000 000 - 0x0007 FFFF	512 K	Main Flash memory, system memory or SRAM depending on BOOT configuration	-

Obr. 4.9 Hraniční adresy SMT32F303RE (převzatá část tabulky 3 v [12])

4.5.1 Příklad: Práce s adresami

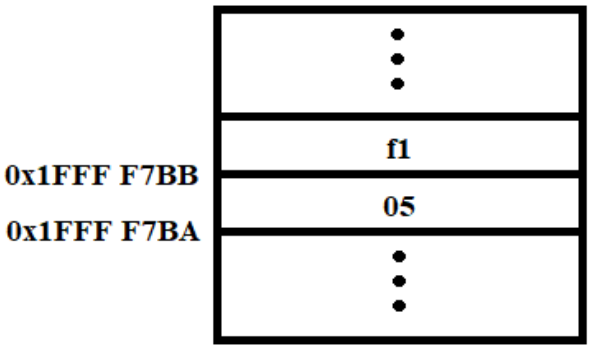
S pojmem adresa jsme se již setkali v kap. 3.11.2, kde byla na „určitém místě“ uložena kalibrační konstanta (dále jen kal. konst.). Na tomto příkladu si konkrétně demonstrujeme, jak se s adresami pracuje a na co si dát při práci s nimi pozor.

Z obr. 3.47 je zřejmé, že se kal. konst. nachází na adrese 0x1FFF F7BA - 0x1FFF F7BB. Tyto adresy jsou uvnitř sekce 0 označené jako „CODE“, konkrétně se jedná o adresy v intervalu 0x1FFF D800 až 0x1FFF F800, což je „System memory“ (viz obr. 4.8). Jedná se o dvě adresy, každá ukazuje na jeden bajt, proto má kal. konst. 2 bajty. V kap. 3.11.2 byl také ukázán způsob, jak ji z této adresy přečíst:

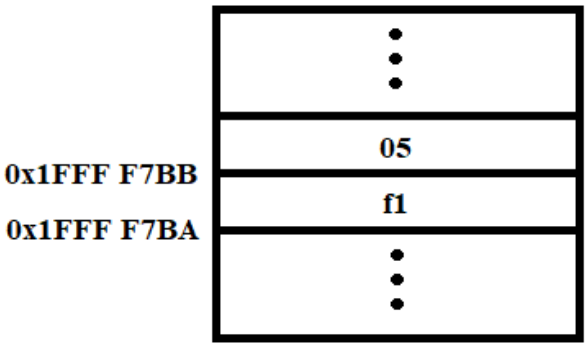
```
#define VREF_CAL *(uint16_t *)0x1FFFF7BA
```

Kal. konst. se tak uloží do konstanty (makra) *VREF_CAL*. Typ *uint16_t* určuje, kolik bitů se přečte (16 bitů jsou 2 bajty) od dané adresy směrem nahoru. Pokud si sériovým portem pošleme její hodnotu do terminálu v dekadické a hexadecimální podobě, dostaneme čísla **1521** a **5f1**. Dobře, tedy kal. konst je v paměti uložena tak, jak je znázorněno na obr. 4.10? Nikoli…

Bajty jsou v paměti Nuclea kódovány ve formátu **Little-endian**. V tomto formátu je na **nejnižší adrese** uložený **nejméně významný bajt** a naopak na nejvyšší adrese je uložený nejvíce významný bajt. Kal. konst. má tedy na nejnižší adrese „f1“ a na nejvyšší „05“ (viz obr. 4.11). Opačný způsob kódování bajtů v paměti je ve formátu Big-endian, ve kterém je na nejnižší adrese uložen nejvíce významný bajt a na nejnižší adrese nejméně významný bajt (kal. konst. by v tomto formátu byla kódována tak, jak je znázorněno na obr. 4.10). [10]



Obr. 4.10 Uložení kal. konst. v paměti (formát Big-endian)



Obr. 4.11 Uložení kal. konst v paměti (formát Little-endian)

Výsledek si ověříme pomocí následujícího programu:

```
#include "mbed.h"
#define VREF_CAL *(uint16_t *)0x1FFFF7BA
#define add1 *(uint8_t *)0x1FFFF7BA
#define add2 *(uint8_t *)0x1FFFF7BB
Serial serial(PA_2, PA_3);
int main() {
        serial.printf("VREF_CAL: %x\t0x1FFFF7BA: %x\t0x1FFFF7BB: %x\n\r",
                VREF_CAL, add1, add2);
}
```

4.6 Elektronické paměti

Na co je paměť vlastně potřeba? Někde v mikrokontroléru musí být uložen program (ve formě instrukcí v binární podobě, viz obr. 3.27) a data, se kterými pracuje. Paměť, ve které je uložen program se nazývá *program memory* a paměť s daty *data memory*.

Základní paměť mikrokontrolérů tvoří **Flash** a **RAM** (Random Access Memory). RAM se dělí na SRAM a DRAM (rozdíl vysvětlíme dále). SMT32F303RE má Flash velikost **512 *kB*** a RAM **80 *kB***. RAM je navíc rozdělena na SRAM (64 *kB*) a CCM (Core Coupled Memory) SRAM (16 *kB*), která se označuje jako paměť s rychlým přístupem a slouží pro vykonávání časově náročných operací (v kontextu této práce tuto vlastnost potřebovat nebudeme). [12]

Po přečtení minulé kapitoly by mělo být jasné, že zmíněné paměti mají svoje místo v paměťovém prostoru (mají přiřazené určité rozsahy adres):

- Flash – 0x0800 0000 – 0x0807 FFFF,
- SRAM – 0x2000 0000 – 0x2000 FFFF,
- CCM SRAM – 0x1000 0000 – 0x1000 3FFF.

Jednotlivé adresy lze najít v již zmíněné tabulce 3 v [12]. Konkrétně hraniční adresy pro Flash a CCM RAM jsou přímo vidět na obr. 4.9. Poznamenejme, že značení pro paměť s rychlým přístupem není vždy stejné (viz CCM RAM na obr. 4.8 a CCM SRAM na obr. 4.9). Touto pamětí se však dále zabývat nebudeme. V následujícím textu se pojmem **RAM** myslí paměť SRAM (konkrétní typ rozlišovat nebudeme).

Rozdílů mezi pamětí Flash a RAM je několik. Flash je na napájení nezávislá (angl. **non-volatile**) paměť, která se používá pro uložení **programu** a **konstantních globálních proměnných** (ty s klíčovým slovem *const*) [25]. Program se za pomoci ST-Linku ukládá právě do této paměti. Zápis do ní tedy probíhá při nahrávání binárního souboru, v průběhu běhu programu se z ní pouze čte. Skutečnost, že se jedná o non-volatile paměť umožňuje, aby si Nucleo „pamatovalo“ nahraný program i po odpojení napájení. Zápis do paměti Flash je v porovnání se zápisem do RAM více komplexní.

RAM je naopak paměť závislá na napájení (angl. **volatile**), ve které jsou uloženy **všechny proměnné** používané při běhu programu [25]. To zahrnuje statické proměnné, zásobník (angl. stack) a haldu (angl. heap), o nich více později [25]. Zásadní výhoda RAM je doba přístupu k datům (angl. access time), která může být až několika násobně **rychlejší** v porovnání s přistupováním k datům v paměti Flash. To je důvod, proč se používá při vykonávání programu. Do paměti RAM je možné přistupovat jako k bajtu, polovičnímu slovu (angl. **halfword**) a plnému slovu (angl. **full word**[11]). Bajt má 8 bitů, halfword 16 bitů a word 32 bitů [12].

Stack a Heap

Stack se používá pro uložení lokálních proměnných, což jsou proměnné s tzv. fixní životností. Naopak Heap se používá pro proměnné vzniklé dynamickou alokací paměti (např. s využitím funkce *malloc*). [25]

DRAM a SRAM

SRAM (Static Random Access Memory), neboli statická paměť, je tvořená bistabilním klopným obvodem sestaveným ze čtyř nebo šesti tranzistorů. Bistabilní znamená, že je stabilní ve dvou stavech (to je např. i mince, kde jeden stabilní stav představuje panna a druhý orel). Informace je tedy uložena ve stavu paměťové buňky. [18]

DRAM (Dynamic Random Access Memory), neboli dynamická paměť, pro uchování informace nevyužívá stav obvody, nýbrž malý náboj uložený v kondenzátoru. Pro zajímavost velikost této kapacity je v řádech desetin pikofaradů. [37]

Vzhledem k malé kapacitě kondenzátoru se vlivem nedokonalostí reálného světa (prostředí není zcela nevodivé) kondenzátor samovolně vybíjí. Pokud je tedy kondenzátor v log. jedničce (DRAM buňka uchovává 1), postupně dojde k přechodu do log. nuly. Tímto způsobem DRAM „zapomíná“ informace a proto je nutná jejich periodická obnova (řádově desítky mikrosekund). Čtení buňky DRAM je destruktivní operace (kondenzátor se vybije), po které je také zapotřebí informaci obnovit. [37]

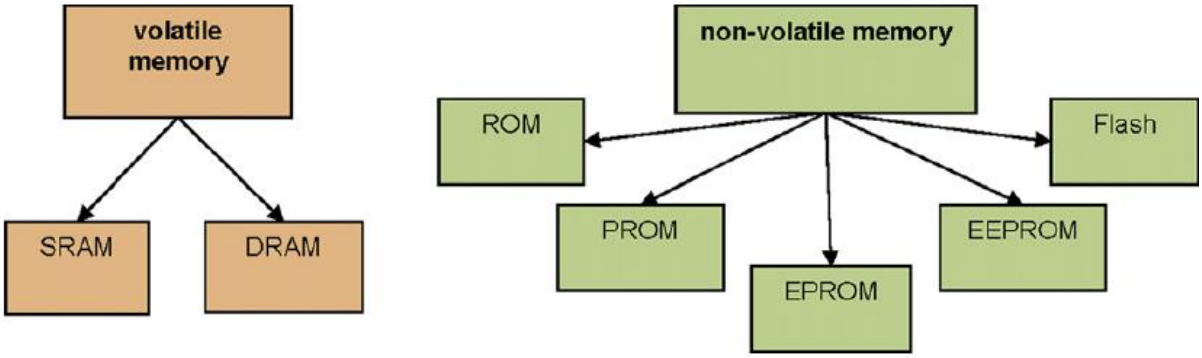
Jelikož se jedna paměťová buňka SRAM skládá ze čtyř nebo šesti tranzistorů, velikost a cena takovéto buňky je v porovnání s DRAM vyšší. Kupříkladu DRAM může mít čtyřikrát větší kapacitu na stejné ploše a to za stejnou cenu. Naopak výhoda SRAM je nízká spotřeba (angl. low power). [18]

[11] Dále místo full word se používá pouze výraz **word**.

Přehled elektronických pamětí

Na obr. 4.12 je graficky znázorněno rozdělení elektronických pamětí podle závislosti na napájení. Tento obrázek je uveden pro shrnutí a také kvůli seznámení se zkratkami některých dalších non-volatile pamětí.

ROM (*Read Only Memory*) je paměť, do které zapíše výrobce a uživatel z ní může pouze číst. Do paměti **PROM** (*Programmable ROM*) se na rozdíl od paměti ROM dá zapsat, avšak **pouze** jednou. Opětovné zapisování je možné do pamětí **EPROM** (*Eraseable PROM*) a **EEPROM** (*Electrically Erasable PROM*), které se liší způsobem zápisu. [18]



Obr. 4.12 Rozdělení elektronických pamětí podle závislosti na napájení (převzato z [18])

4.6.1 Příklad: Ověření získaných poznatků

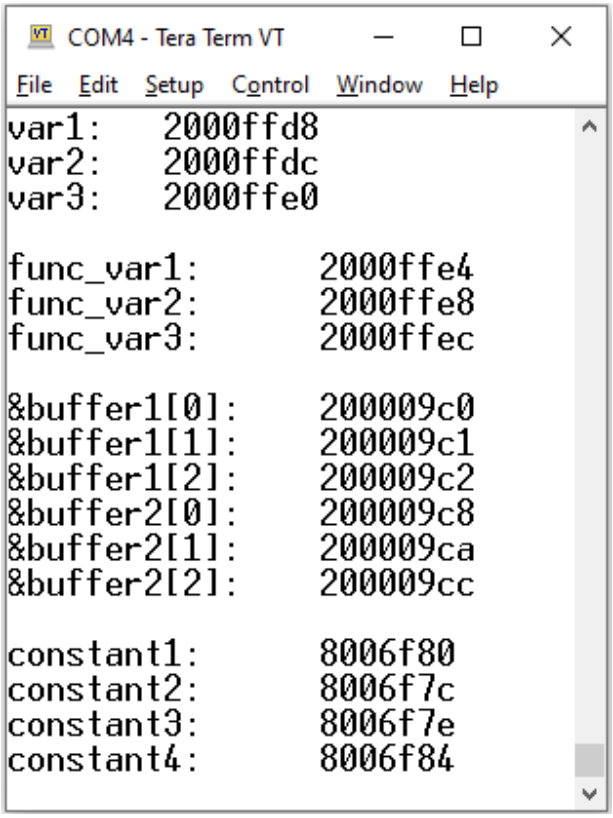
V kap 4.6 bylo řečeno, že program a konstantní globální proměnné (ty s klíčovým slovem *const*) jsou uloženy v paměti Flash, které jsou přiděleny adresy v intervalu 0x0800 0000 až 0x0807 FFFF. Proměnné, které se používají za běhu programu (všechny statické a globální proměnné), jsou naopak uloženy v paměti RAM, která se nachází v rozsahu adres 0x2000 0000 až 0x2000 FFFF. Tyto poznatky v této části ověříme.

Program pouze vytváří určité proměnné a posílá po s. portu jejich adresy. Pro demonstraci další (zatím nezmíněné) vlastnosti budou všechny druhy vytvořených proměnných různých velikostí (datových typů), konkrétně *uint8_t, uint16_t, uint32_t* a jednou i datového typu *int*. Jelikož jsme se s adresním prostorem seznámili teprve v kap. 4.5, bude vše názorně graficky vysvětleno.

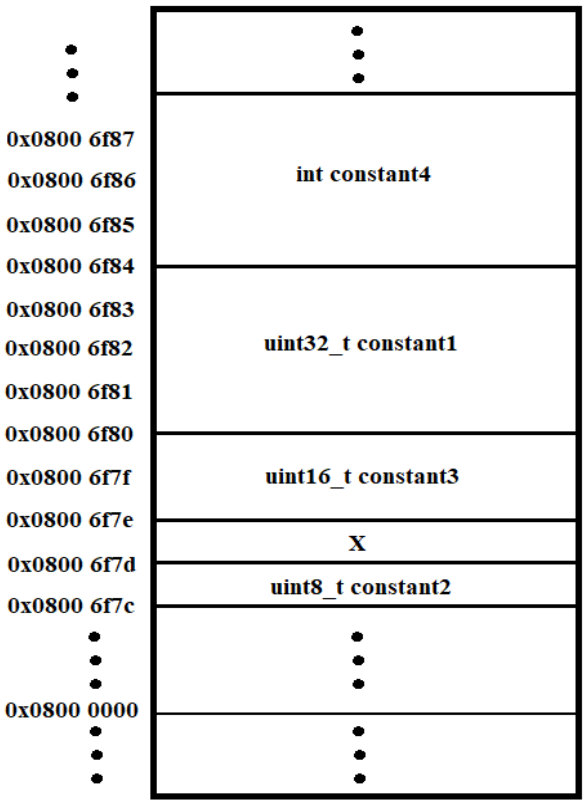
Nejprve se zaměřme na globální proměnné s klíčovým slovem *const* (statické se ukládají také na RAM, vyzkoušejte):

```
const int constant1 = 1;
const uint8_t constant2 = 2;
const uint16_t constant3 = 3;
const uint32_t constant4 = 4;
```

Zopakujme, že by měly být uloženy v paměti Flash (0x0800 0000 až 0x0807 FFFF), což podle obr. 4.13 platí. Tato skutečnost je graficky znázorněna na obr. 4.14. Proč jsou však adresy rozděleny právě takto? Jako první se definovala proměnná *constant1*, proto by měla být také definována jako první, tzn. mít nejnižší adresu v paměti Flash?



Obr. 4.13 Příchozí adresy



Obr. 4.14 Paměť Flash

Jelikož se do paměti RAM může přistupovat jako k bajtu, halfwordu nebo wordu, je možné vytvářet tzv. zarovnané paměti (angl. *aligned memory*). Využitím této vlastnosti získává kompilátor určitou „svobodu“ při ukládání programových dat do paměti. Kompilátor by měl uložit data tak, aby to bylo co nejvhodnější pro CPU. Žádoucí je, aby čtyřbajtové (*uint32_t* a *int)* a dvoubajtové datové typy začínaly na adrese dělitelné čtyřmi, příp. dvěma pro dvoubajtové datové typy.

V případě dodržení uvedeného pořadí by taková situace nastala, jelikož by proměnná *constant4* začínala na adrese 0x0800 6f83. Proč? Proměnná *constant1* by začínala na 0x0800 6f7c, *constant2* na 0x0800 6f80 a *constant3* na 0x0800 6f81. Důvod uložení dat přesně tak, jak je znázorněno na obr. 4.14, není zřejmý. Nicméně kompilátor by měl sám vědět, jak data co nejvhodněji uspořádat…

Dále otestujme dynamicky alokovanou paměť:

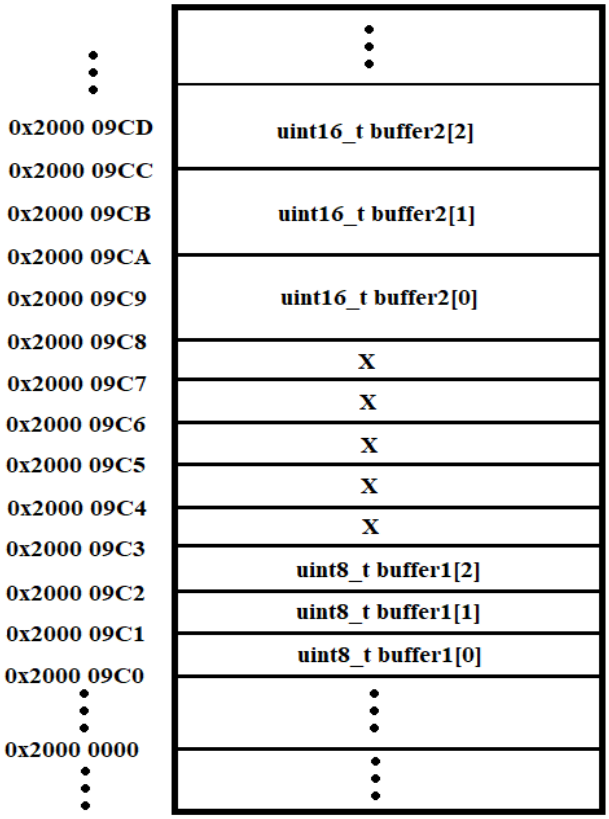
```
uint8_t* buffer1 = (uint8_t*) malloc(sizeof(uint8_t)*3);
uint16_t* buffer2 = (uint16_t*) malloc(sizeof(uint16_t)*3);
```

Alokovala se tak paměť o velikosti tří bajtů pro *buffer1* a šesti bajtů pro *buffer2*. Přidělené adresy lze opět najít na obr. 4.13 a jejich grafické znázornění na obr. 4.15. V tomto případě se kompilátor (z nejasných příčin) rozhodl mezi *buffer1* a *buffer2* nechat 5 volných bajtů označených symbolem „X“. Vynechání adresy 0x2000 09C3 za účelem stanovení začátku proměnné *buffer2* na adrese dělitelné čtyřmi je pochopitelné, avšak důvod vynechání následujících dvou halfwordů není zřejmý. Poznamenejme, že přidělené adresy jsou blízko počáteční hraniční adrese pro RAM (0x2000 0000), tedy **Heap** se ukládá na začátku RAM.

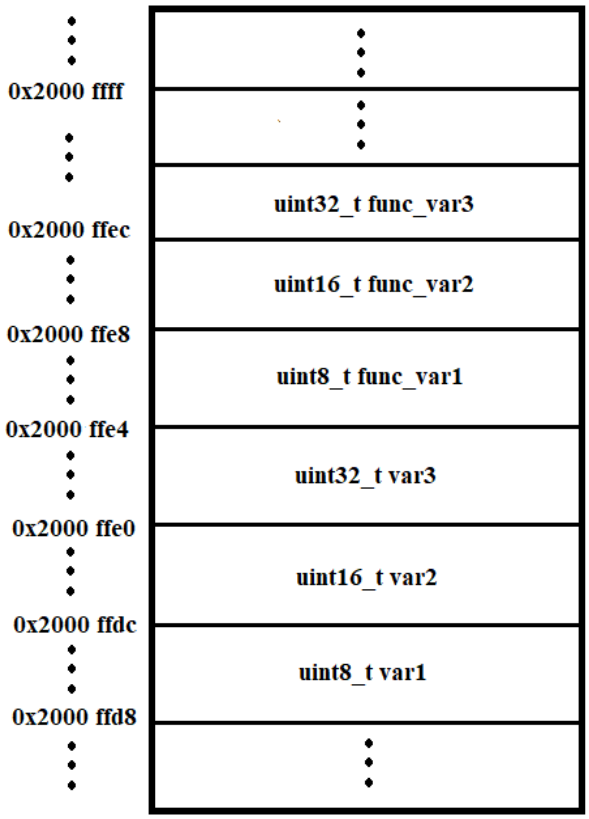
Zbývá otestovat statické proměnné, které jsou jednak v hlavní funkci *main* (s názvem „var“), ale i uvnitř jiné vytvořené funkce (s názvem „func_var“):

```
uint8_t func_var1;
uint16_t func_var2;
uint32_t func_var3;
uint8_t var1;
uint16_t var2;
uint32_t var3;
```

Přidělené adresy jsou opět vidět na obr. 4.13 a jejich grafické znázornění na obr. 4.16. Je patrné, že v rámci Stacku kompilátor „nešetří místem“ a všem použitým proměnným přiděluje adresu dělitelnou čtyřmi. Poznamenejme, že přidělené adresy jsou blízko koncové hraniční adrese pro RAM (0x2000 ffff), tedy **Stack** se ukládá na konec RAM. Celý program je uveden v příloze 12.10.



Obr. 4.15 Paměť RAM - Heap



Obr. 4.16 Paměť RAM - Stack

4.7 Rozdělení periferií

S pojmem periferie jsme se seznámili na samotném počátku této práce. Přesněji byl definován v poznámce pod čarou 5, kde se periferie definovala jako rozhraní, pomocí něhož umí mikrokontrolér komunikovat s „vnějším světem“. V této kap. si pro přehled periferie rigorózně rozdělíme.

Periferie se na mikrokontrolérech STM32 dají rozdělit do dvou skupin: periferie s univerzálním účelem (angl. **general purpose peripherals**) a komunikační periferie (**communications peripherals**). Všechny jsou vysoce sofistikované. Každá z nich má mnoho „chytrých funkcí“, které slouží hlavně k minimalizování množství času potřebného od CPU k jejich řízení. [39]

4.7.1 Periferie s obecným účelem

Periferie s univerzálním účelem se skládají z univerzálních vstupně výstupních pinů (general purpose IO, zkráceně **GPIO**), kontroléru pro externí přerušení (external interrupt controller), ADC, časovačů (general purpose/advanced timer units) a záložních registrů (backup registers) [39]. Pro účely této práce je nejdůležitější periferie GPIO, proto se jí budeme věnovat v další kapitole. Přehled základního použití ostatních periferií s univerzálním účelem je přehledně zpracován v kap. 5 v [39], méně názorně v datasheetu ([10]) a obsáhle se všemi podrobnostmi v referenčním manuálu ([12]).

4.7.2 Komunikační periferie

O komunikačních periferiích již bylo psáno na konci kap. 3.12. Připomeňme, že mikrokontroléry STM32 disponují pěti typy komunikačních periferií: SPI, I^2C, CAN, USB a USART.

5 GPIO a hodinový signál

V předchozí kapitole došlo k základnímu seznámení s vnitřní strukturou mikrokontroléru. Součástí této struktury jsou také GPIO brány a hodinový signál, které jsou však natolik obsáhlé, že jim je věnována samostatná kapitola. V této části se také seznámíme s dalšími ladícími nástroji z práce [1], jež jsou užitečné při práci s registry (viz dále). Na konci kapitoly budeme schopni **sami** nakonfigurovat daný pin jako digitální vstup/výstup bez závislosti na třídách *DigitalIn*/*DigitalOut* a to dokonce bez celkové závislosti na knihovně *mbed.h*.

5.1 GPIO

Na začátku této práce (v kap. 3.2.1) bylo pro tu nejjednodušší manipulaci s mikrokontrolérem (blikání diodou) potřeba zavést termín **konfigurace**. Připomeňme, že tento termín byl definován jako proces, při kterém se procesoru „řekne“, co a za jakým účelem se bude používat. S využitím tříd v mbedu se tento proces značně zjednodušil, protože např. pro použití digitálního výstupu stačilo vytvořit objekt třídy *DigitalOut* s argumentem určujícím daný pin a byla konfigurace hotová. Postupně však bylo zmiňováno, že se s periferiemi dá dělat víc, než mbed nabízí. Za tímto účelem je potřeba, abychom si periferie konfigurovali sami. Cílem této kapitoly je osvojení schopnosti nakonfigurovat GPIO, konkrétně si ukážeme blikání LED bez využití třídy *DigitalOut*.

Samotná konfigurace probíhá vždy zápisem do příslušných registrů přístupných na daných adresách. Obecně se musí najít adresy přidělené dané periferii sloužící pro manipulaci s ní. Místa v paměti vyhrazená pro samotnou konfiguraci se nazývají **konfigurační registry**. Konfigurační registry však nejsou jediné registry, které má periferie k dispozici (viz dále), proto pod pojmem registr obecně chápejme místo v paměti určené pro práci s danou periferií (také existují registry procesoru, o nich více v kap. 7). Pro používání určité periferie je tedy zapotřebí dohledat její registry.

Každá GPIO brána (angl. port), připomeňte si jejich umístění na obr. 4.6, má čtyři 32-bitové konfigurační registry (GPIOx_MODER, GPIOx_OTYPER, GPIOx_OSPEEDR a GPIOx_PUPDR), dva 32-bitové datové registry (GPIOx_IDR a GPIOx_ODR) a jeden 32-bitový set/reset registr (GPIOx_BSRR). Dále jeden 32-bitový reset registr (GPIOx_BRR), jeden 32-bitový zamykací registr (GPIOx_LCKR) a dva 32-bitové registry pro nastavení alternativní funkce (GPIOx_AFRH a GPIOx_AFRL). Dohromady se tedy jedná o **jedenáct registrů** pro práci s každou bránou GPIO. [12]

Kde tyto registry nalézt? **Každý** GPIO port má svůj adresní prostor, ve kterém se zmíněné registry nacházejí, viz obr. 5.1. Jelikož jsou GPIO brány připojeny na sběrnici AHB2, najdeme hraniční adresy jednotlivých GPIO bran v sektoru AHB2 (viz obr. 4.6). Pro naše účely jsou nejdůležitější brány GPIOA, GPIOB a GPIOC, proto jsou na obr. 4.6 tyto brány zvýrazněny červeným rámečkem. Uživatelská dioda je připojena na pinu PA5, tedy na bráně GPIOA. Začátek adresního prostoru této brány je na adrese **0x4800 0000**.

V manuálu se neuvádí hraniční adresa každého registru, nýbrž pouze tzv. **address offset**. Jedná se o číslo, které přičteme k začátku adresního prostoru dané brány, což je u brány GPIOA již zmíněná adresa 0x4800 0000. Přehled všech registrů je v kap. 11.4 v [12]

Bus	Boundary address	Size (bytes)	Peripheral
AHB4	0xA000 0000 - 0xA000 0FFF	4 K	FSMC control registers
	0x8000 0000 - 0x9FFF FFFF	512 M	FSMC Banks 3 and 4
	0x6000 0000 - 0x7FFF FFFF	512 M	FSMC Banks 1 and 2
-	0x5000 0800 - 0x5FFF FFFF	384 M	Reserved
AHB3	0x5000 0400 - 0x5000 07FF	1 K	ADC3 - ADC4
	0x5000 0000 - 0x5000 03FF	1 K	ADC1 - ADC2
-	0x4800 2000 - 0x4FFF FFFF	~132 M	Reserved
AHB2	0x4800 1C00 - 0x4800 1FFF	1 K	GPIOH
	0x4800 1800 - 0x4800 1BFF	1 K	GPIOG
	0x4800 1400 - 0x4800 17FF	1 K	GPIOF
	0x4800 1000 - 0x4800 13FF	1 K	GPIOE
	0x4800 0C00 - 0x4800 0FFF	1 K	GPIOD
	0x4800 0800 - 0x4800 0BFF	1 K	GPIOC
	0x4800 0400 - 0x4800 07FF	1 K	GPIOB
	0x4800 0000 - 0x4800 03FF	1 K	GPIOA
-	0x4002 4400 - 0x47FF FFFF	~128 M	Reserved
AHB1	0x4002 4000 - 0x4002 43FF	1 K	TSC
	0x4002 3400 - 0x4002 3FFF	3 K	Reserved
	0x4002 3000 - 0x4002 33FF	1 K	CRC
	0x4002 2400 - 0x4002 2FFF	3 K	Reserved
	0x4002 2000 - 0x4002 23FF	1 K	Flash interface
	0x4002 1400 - 0x4002 1FFF	3 K	Reserved
	0x4002 1000 - 0x4002 13FF	1 K	RCC
	0x4002 0800 - 0x4002 0FFF	2 K	Reserved
	0x4002 0400 - 0x4002 07FF	1 K	DMA2
	0x4002 0000 - 0x4002 03FF	1 K	DMA1

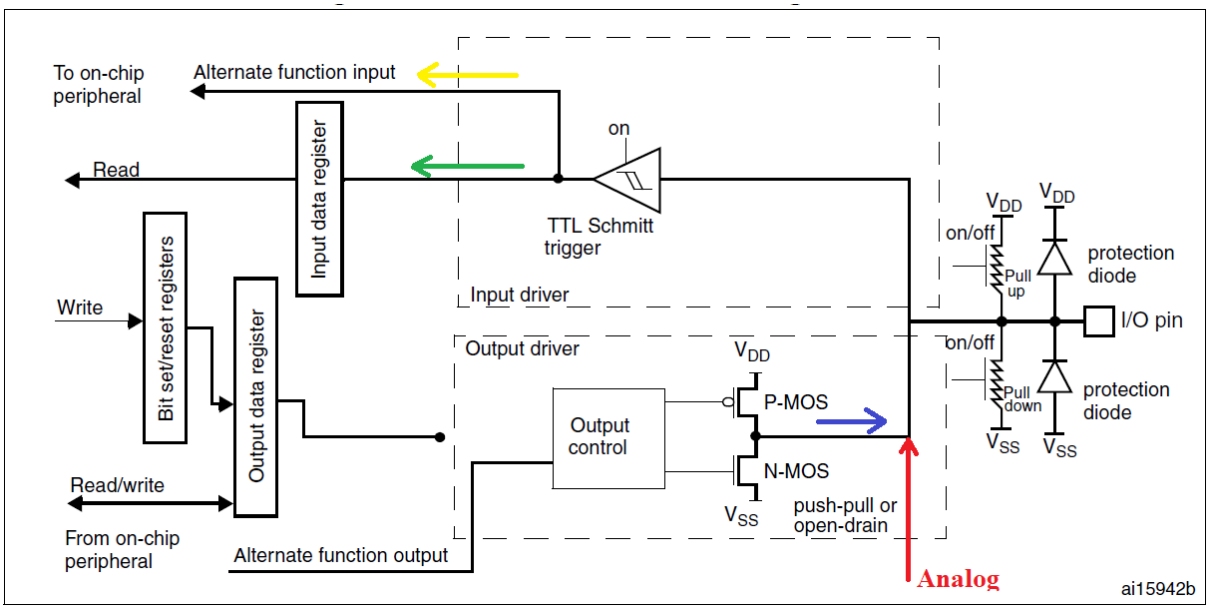
Obr. 5.1 Adresní prostor sběrnic AHBx (převzatá část tabulky 15 z [10])

5.1.1 Konkrétní realizace GPIO

Před vysvětlováním významu jednotlivých registrů je vhodné uvést konkrétní schéma vstupně výstupního pinu (viz obr. 5.2), jelikož úzce souvisí s významem dále probíraných registrů. Jinými slovy každý registr má na obr. 5.2 svoje místo a plní na něm danou funkci.

Základní rozdělení GPIO je na **dvě části** (viz obr. 5.2). V první se pin využívá jako vstup (horní obdélník vyznačený přerušovanou čarou) a v druhé jako výstup (spodní obdélník vyznačený přerušovanou čarou). Obě části potřebují řídící obvody (input driver a output driver), které představují zmíněné obdélníky.

Každý vstup také disponuje ochranou proti statické elektřině ve formě dvou ochranných diod připojených nejblíže samotnému konci vstupně výstupního pinu (najděte si je na obrázku).



Obr. 5.2 Konfigurace vstupního výstupního pinu (převzato z [12])

5.1.2 Konfigurační registry

5.1.2.1 Registr GPIOx_MODER

Vysvětlení tohoto registru bude obsáhlé a názorné, protože se jedná o první probíraný registr. V levém horním rohu je necháno číslo kapitoly z referenčního manuálu [12] (viz obr. 5.3). Následuje jméno registru a jeho zkratka, což je v tomto případě „GPIOx_MODER“. Znak „x“ za „GPIO“ znamená, že tento registr existuje pro více bran. Pro které je definováno na konci řádku řetězcem „x=A..H“, existuje tedy pro vaše GPIO brány.

Na druhém řádku se definuje již zmíněný „address offset“, který je v tomto případě 0x00. Registr GPIOA_MODER se tudíž nachází na adrese 0x4800 0000 (0x4800 0000+0x00). Další řádek definuje hodnotu, kterou registr nabývá po resetu Nuclea, Připomeňme, že hexadecimální číslo je zakódováno do čtyř bitů, proto např. hodnota 0xA800 0000 je skutečně 32 bitů.

Následuje grafické znázornění segmentace registru. Čísla reprezentují daný bit (začíná se od 0 a končí 31). Pod čísly se nachází tabulka o velikosti 16 bitů a její rozdělení. Kupříkladu na bitech 16 a 17 se nachází oblast zvaná „MODER8“. Co do této oblasti doplnit si povíme v dalším odstavci. U každého bitu je také uvedeno, co je s daným bitem povoleno dělat. V tomto případě je pod každým bitem řetězec „rw“ reprezentující *read-write*, což znamená, že se do daného bitu může zapisovat příp. z něj číst.

Pod grafickou částí je uvedeno, co se do jednotlivých oblastí zapisuje. Do registru GPIOx_MODER se zapisují dva bity definující danou konfiguraci pinu (dané brány). Tedy v případě, kdy chceme pin PA5 nakonfigurovat jako výstupní, musíme do oblasti „MODER5“ zapsat dvojici bitů **01**, která odpovídá módu „general purpose output“.

Na obr. 5.3 se dále vyskytuje poznámka k bitům 10 a 11, která se ovšem netýká Nuclea, s nímž pracujeme, jelikož ten disponuje mikrokontrolérem STM32F303RE, nikoli žádným z uvedených. V referenčním manuálu se často vyskytují podobné poznámky příp. různé

podmínky, avšak vždy je nutné **prvně** zjistit, k jakému mikrokontroléru se daná poznámka/podmínka vztahuje.

Význam registru GPIOx_MODER na obr. 5.2 je ten, že určuje, jakým „směrem“ se bude daný pin používat. Tyto směry jsou na zmíněném obrázku vyznačeny čtyřmi šipkami (žlutá, zelená, modrá a červená). Červená šipka je přimalována kvůli tomu, že tento obrázek odpovídá zapojení pro alternativní funkci, nikoli pro všechny funkce současně (viz kap. 11.3 v [12]), avšak připojení analogového obvodu se v těchto místech skutečně vyskytuje. Pokud se pin nakonfiguruje do analogového módu (tomu odpovídá dvoje bitů 11), signál opravdu směřuje ve směru této šipky.

11.4.1 GPIO port mode register (GPIOx_MODER) (x =A..H)

Address offset:0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[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
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y+1:2y **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O mode.
00: Input mode (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Note: In STM32F303xB/xC and STM32F358x devices, bits 10 and 11 of GPIOF_MODER are reserved and must be kept at reset state.

Obr. 5.3 Registr GPIOx_MODER (převzato z [12])

5.1.2.2 Registr GPIOx_OTYPER

Druhý konfigurační registr GPIOx_OTYPER určuje, zda se v případě použití GPIO ve výstupním módu použije výstup v módu push-pull nebo open-drain (viz obr. 5.4). Pro účely této práce stačí vědět, že GPIO ve výstupním módu se chová „přirozeně“, pokud je v módu push-pull. Pojmem „přirozeně“ se myslí, že pin ve stavu log. 1 má na výstupu napájecí napětí a v stavu log. 0 nulové napětí. Analogie s třídou *DigitalOut* je ta, že pokud pomocí této třídy nastavíme pin jako výstupní, pak se od něj očekává, že po zapsání nuly do objektu této třídy bude na výstupu daného pinu 0 *V*, zatímco po zapsání jedničky do objektu této třídu bude na daném pinu 3,3 *V*. Popsané chování je ekvivalentní použití módu push-pull.

Mód push-pull se definuje zapsáním 0 na dané místo v GPIOx_OTYPER. U tohoto registru se však zapisuje pouze na prvních 16 bitů, nikoli na celých 32 bitů. Důvodem je dostatečnost

16 bitů pro definování výstupního módu 16 pinů (pro každý pin je potřeba jeden bit, nikoli dva jako u registru GPIOx_MODER). Do zbylých pinů se nesmí zapisovat ani z nich číst.

Je nutné zapisovat 0 definující mód push-pull, když je hodnota registru po resetování 0x0…0? **Není**, tudíž konfigurace registru GPIOx_OTYPER za účelem použití GPIO v módu push-pull je hotova „bez práce“. V případě přehlédnutí „Reset value“ je na tuto skutečnost také upozorněno v definici významu 0 a 1.

11.4.2 GPIO port output type register (GPIOx_OTYPER) (x = A..H)

Address offset: 0x04

Reset value: 0x0000 0000

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

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OTy:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O output type.
0: Output push-pull (reset state)
1: Output open-drain

Obr. 5.4 Registr GPIOx_OTYPER (převzato z [12])

5.1.2.3 Registr GPIOx_OSPEEDR

V tomto registru se **nenastavuje** rychlost přepínání GPIO (nakonfigurovaném ve výstupním módu) z jednoho stavu do druhého, ale **strmost** hrany výstupního sign. při tomto přepínání (viz obr. 5.5). Jedná se tedy o speciální požadavek, který pro účely této práce není potřeba více specifikovat. Pro nás stačí ta nejnižší rychlost (**low speed**), která je navíc pro většinu pinů nastavena ihned po resetu. U brány GPIOA a GPIOB jsou dohromady dva piny, které mají po resetu jinou hodnotu než 00, které to jsou? Jedná se o piny PA13 a PB3 a jejich hodnota po resetu je nastavena na 11 (high speed). Pro použití LED se tedy v tomto registru nemusí nic nastavovat.

5.1.2.4 Registr GPIOx_PUPDR

Posledním konfiguračním registrem je registr GPIOx_PUPDR. Jak název „GPIO port pull-up/pull-down register“ napovídá (viz obr. 5.6), slouží k nastavení pull-up/pull-down rezistoru. Význam těchto rezistorů byl vysvětlen v kap. 3.5.4. Na obr. 5.2 se pomocí registru GPIOx_PUPDR jednoduše aktivuje pull-up, pull-down nebo žádný z nich (nachází si vpravo vedle ochranných diod).

Adress offset je u tohoto registru 0x0C. Hodnoty po resetování nejsou u všech pinů „00“ (no pull-up/pull-down). Po resetování jsou na daných pinech uvedené hodnoty: na PA15 „01“, na PA14 „10“ a na PA13 „01“. Pro ujasnění následuje vysvětlení, jak se na tyto hodnoty přišlo. Resetovací hodnota na bráně GPIOA je 0x64, což binárně znamená 0110 0100.

První dvoje „01“ odpovídá pinu PA15, druhá dvojice „10“ odpovídá pinu PA14 atd. Na pinu PA5 je hodnota „00“, proto se v tomto registru nic nastavovat nemusí (pokud nechceme ani pull-up, ani pull-down rezistor).

11.4.3 GPIO port output speed register (GPIOx_OSPEEDR) (x = A..H)

Address offset: 0x08

Reset value:

- 0x0C00 0000 for port A
- 0x0000 00C0 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15 [1:0]		OSPEEDR14 [1:0]		OSPEEDR13 [1:0]		OSPEEDR12 [1:0]		OSPEEDR11 [1:0]		OSPEEDR10 [1:0]		OSPEEDR9 [1:0]		OSPEEDR8 [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
OSPEEDR7 [1:0]		OSPEEDR6 [1:0]		OSPEEDR5 [1:0]		OSPEEDR4 [1:0]		OSPEEDR3 [1:0]		OSPEEDR2 [1:0]		OSPEEDR1 [1:0]		OSPEEDR0 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y+1:2y **OSPEEDRy[1:0]**: Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O output speed.
x0: Low speed
01: Medium speed
11: High speed
Note: Refer to the device datasheet for the frequency specifications and the power supply and load conditions for each speed.

Obr. 5.5 Registr GPIOx_OSPEEDR (převzato z [12])

11.4.4 GPIO port pull-up/pull-down register (GPIOx_PUPDR) (x = A..H)

Address offset: 0x0C

Reset values:

- 0x6400 0000 for port A
- 0x0000 0100 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[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
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y+1:2y **PUPDRy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

Obr. 5.6 Registr GPIOx_PUPDR (převzato z [12])

5.1.3 Shrnutí konfiguračních registrů

Před vysvětlováním dalších registrů je vhodné pro zopakování připomenout význam konfiguračních registrů:

- GPIOx_MODER – za jakým účelem se pin použije („00“ pro vstupní mód, „01“ pro výstupní mód),
- GPIOx_OTYPER – nastavuje se **pouze** pro GPIO nakonfigurovaný v předchozím registru jako výstupní a smyslem je specifikace módu výstupní pinu (push-pull nebo open-drain),
- GPIOx_OSPEEDR – rychlost přepínání GPIO (pro naše účely nemusíme nastavovat),
- GPIOx_PUPDR – použití pull-up, pull-down rezistoru.

V mbedu u třídy *DigitalOut* bylo také možné nastavit mód výstupního pinu případně použít pull-up/pull-down rezistor (viz kap. 4.2 v [1]). Uvnitř mbedu se tyto vlastnosti nastavují **stejným způsobem**, tzn. prostřednictvím zmíněných registrů.

5.1.4 Bitové operace

V kap. 5.1.2 bylo řečeno, co se má zapsat do konfiguračních registrů GPIO, aby se konkrétní vstupně výstupní pin mohl používat daným účelem. Ve zmíněné kapitole však nebylo uvedeno, **jak** zapsat požadovanou (binární) hodnotu do registru, což je právě náplní této kapitoly.

Nejdříve si ukážeme, jak se vlastně čte/zapisuje z/do paměti. Tato metoda se nazývá **přímá** a je zdlouhavější, avšak pro prvotní pochopení je ideální. V mbedu je také možné využít jinou metodu, se kterou se seznámíme dále (viz kap. 6.1). Čtení z paměti z adresy 0x4800 0000:

```
uint32_t value = *((volatile uint32_t *)0x48000000);
```

Zapsání proměnné *value* na stejnou adresu:

```
*((volatile uint32_t *)0x48000000) = value;
```

Bitovými (dále bit.) operacemi se obecně rozumí operace, jež pracují na binární úrovni, tzn. manipulují s čísly jako s posloupnostmi nul a jedniček. Pro účely této práce budou nejdůležitější následující bit. operace: bit. součet, bit. součin, bit. exkluzivní součet, bit. negace a bit. posuny.

Bit. součet (log. operace disjunkce neboli OR) vykonává (myšleno v jazyce C/C++) operátor „|“, **bit. součin** (log. operace konjunkce neboli AND) operátor „**&**“, **bit. exkluzivní součet** (log. operace exkluzivní disjunkce neboli XOR) operátor „^“, bit. negaci (log. operace negace neboli **NOT**) operátor „~“ a bit. posun doleva/doprava operátor „>>“/„<<“.

V našem případě budeme tyto operace využívat pro tři činnosti: nastavení konkrétního bitu do 0 nebo 1 a pro přepnutí hodnoty bitu. **Nastavení bitu do 1** se vykoná pomocí bit. součtu a bit. posunu doleva. Kupříkladu chceme nastavit hodnotu „01“ do registru GPIOA_MODER do oblasti „MODER5“ (předpokládá se, že je na začátku v této oblasti sekvence „00“). Předpona „0b“ indikuje, že je číslo uvedeno v binární formě. Posouvá se o 10 míst vlevo

(2*5), protože pro každý pin jsou potřeba dva bity, tzn. dohromady 10 míst před oblastí „MODER5“:

```
*((volatile uint32_t *)0x48000000) |= 0b01 << 2*5; // 0b - binary format
```

K **přepnutí hodnoty bitu** se používá log. operace XOR. Tato akce se tedy používá, pokud chceme bit přepnout z jednoho log. stavu do druhého, což je žádoucí operace např. při práci s registrem GPIOA_ODR, jehož význam se objasní v kap. 5.1.6.2. Přepnutí hodnoty pátého bitu by se vykonalo takto:

```
*((volatile uint32_t *)0x48000014) ^= 0b1 << 5;
```

Nastavení hodnoty bitu do 0 je z uvedených tří akcí tou nejsložitější operací, jelikož jsou pro ni potřeba tři bitové operace: bit. součin, bit. negace a bit. posun doleva. Uvažujme situaci, kdy se požaduje zpětné zapsání sekvence „00“ (prvně se zapsalo např. „01“) do zmíněné oblasti „MODER5“ v registru GPIOA_MODER. Tato akce se nedá vykonat následujícím způsobem, jelikož pomocí bit. součtu **nelze** vynulovat žádný bit (z podstaty této log. operace):

```
*((volatile uint32_t *)0x48000000) |= 0b00 << 2*5;
```

Požadovanou operaci lze vykonat následujícím způsobem:

```
*((volatile uint32_t *)0x48000000) &= ~(0b11 << 2*5);
```

V případě nejasností u kterékoli z výše uvedených činností silně doporučuji rozepsání operace na papír, jelikož se dané akce budou po zbytek této práce velmi často používat (obzvláště při práci s registry).

5.1.5 Nastavení konfiguračních registrů pro práci s LED

S informacemi o konfiguračních registrech (z kap. 5.1.2) a bitových operacích (z kap. 5.1.4) jsme schopni nakonfigurovat pin PA5 pro práci s diodou. Za tímto účelem je **pouze** potřeba nastavit sekvenci „01“ do oblasti „MODER5“ v registru GPIOA_MODER. Tato operace již byla ukázána v předchozí kapitole:

```
*((volatile uint32_t *)0x48000000) |= 0b01 << 2*5; // 0b - binary format
```

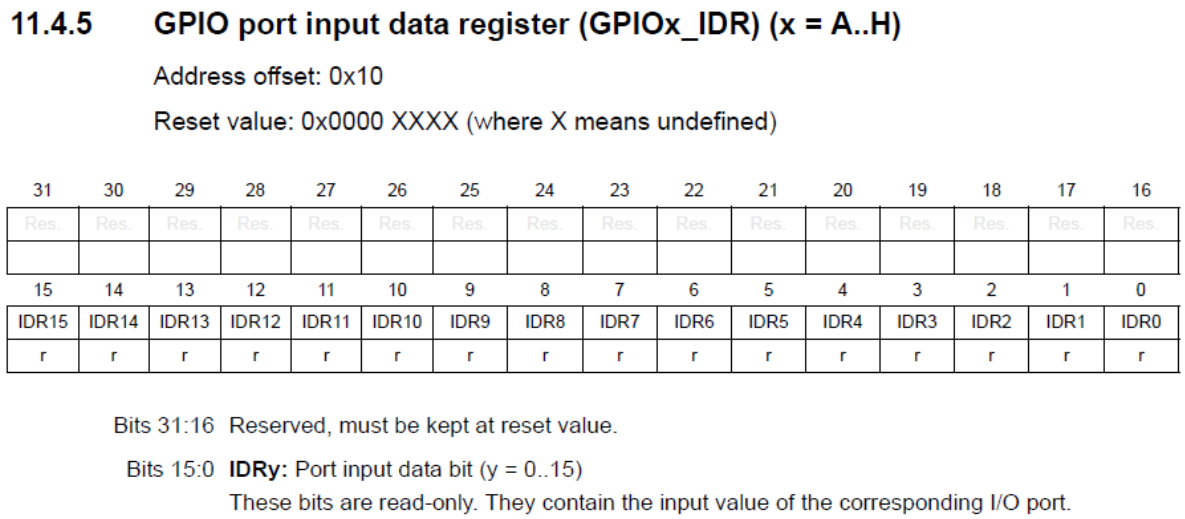
5.1.6 Další GPIO registry

V této části se seznámíme s dalšími GPIO registry, konkrétně s dvěma 32-bitovými datovými registry (GPIOx_IDR), jedním 32-bitovým set/reset registrem (GPIOx_BSRR) a jedním 32-bitovým reset registrem (GPIOx_BRR). Zbývající GPIO registry nebudou probírány. Mezi tyto registry patří zamykací registr GPIOx_LCKR, pomocí něhož lze zamknout konfigurační registry a tak znemožnit jejich překonfigurování. Dále GPIO disponuje dvěma registry pro nastavení alternativní funkce (GPIOx_AFRH a GPIOx_AFRL).

5.1.6.1 Registr GPIOx_IDR

První datový registr je GPIOx_IDR neboli „input data register“ (viz obr. 5.7). V tomto registru se čte log. úroveň pinu nakonfigurovaného v GPIOx_MODER ve vstupním módu (viz kap. 5.1.2.1). Na obr. 5.2 je probíraný registr označen ve formě bloku.

Zdůrazněme dvě skutečnosti. První je, že jednotlivé bity se dají pouze číst (viz řetězec „r“ na obr. 5.7). Druhá skutečnost se týká doposud **utajené zásadní souvislosti** mezi funkčností GPIO a hodinovým signálem. Tím se myslí, že momentálně není vhodné testovat funkčnost GPIO ve vstupním ani výstupním módu, ale pouze se s registry seznámit. Vysvětlení této souvislosti je v kap. 5.1.7.



11.4.5 GPIO port input data register (GPIOx_IDR) (x = A..H)

Address offset: 0x10

Reset value: 0x0000 XXXX (where X means undefined)

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
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **IDRy:** Port input data bit (y = 0..15)
These bits are read-only. They contain the input value of the corresponding I/O port.

Obr. 5.7 Registr GPIOx_IDR (převzato z [12])

5.1.6.2 Registr GPIOx_ODR

Datový registr GPIOx_ODR (viz obr. 5.8) slouží pro nastavení log. úrovně pinu nakonfigurovaného v GPIOx_MODER ve výstupním módu. Jedná se tedy o zásadní registr pro blikání diodou. Zmíněný registr je opět označen na obr. 5.2 formou bloku.

5.1.6.3 Registry GPIOx_BSRR a GPIOx_BRR

Registr GPIOx_BSRR neboli „bit set/reset register“ (viz obr. 5.9) slouží k definování výstupní úrovně daného vstupně výstupního pinu (nakonfigurovaného v GPIOx_MODER ve výstupním módu). Pokud zapíšeme „1“ do oblasti „set bit“ (bity 0 až 15), pak se nastaví výstupní úroveň konkrétního pinu na log. 1. Zapsání „1“ do oblasti „reset bit“ (bity 16 až 31) naopak GPIO resetuje a tím definuje jeho výstupní úroveň na log. 0. Registr GPIOx_BRR má stejnou funkci jako GPIOx_BSRR pro bity 16 až 31, tedy nastavuje výstupní úroveň konkrétního pinu do log. 0 (viz obr. 5.10).

Registry GPIOx_BSRR a GPIOx_BRR **pouze** zjednodušují práci, jelikož jejich funkci zastává samotný GPIOx_ODR. Jak by se v tomto registru nastavil daný bit do 0 nebo 1 bylo uvedeno v kap. 5.1.4. Poznamenejme, že do všech (nerezervovaných) bitů lze v obou registrech pouze zapisovat (viz „w“ na obr. 5.9 a obr. 5.10).

11.4.6 GPIO port output data register (GPIOx_ODR) (x = A..H)

Address offset: 0x14

Reset value: 0x0000 0000

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
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy:** Port output data bit (y = 0..15)

These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and/or reset by writing to the GPIOx_BSRR or GPIOx_BRR registers (x = A..F).

Obr. 5.8 Registr GPIOx_ODR (převzato z [12])

11.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A..H)

Address offset: 0x18

Reset value: 0x0000 0000

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

Bits 31:16 **BRy:** Port x reset bit y (y = 0..15)

These bits are write-only. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

1: Resets the corresponding ODRx bit

Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BSy:** Port x set bit y (y= 0..15)

These bits are write-only. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

1: Sets the corresponding ODRx bit

Obr. 5.9 Registr GPIOx_BSRR (převzato z [12])

11.4.11 GPIO port bit reset register (GPIOx_BRR) (x =A..H)

Address offset: 0x28
Reset value: 0x0000 0000

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
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

Bits 31:16 Reserved

Bits 15:0 **BRy:** Port x Reset bit y (y= 0..15)
These bits are write-only. A read to these bits returns the value 0x0000
0: No action on the corresponding ODx bit
1: Reset the corresponding ODx bit

Obr. 5.10 Registr GPIOx_BRR (převzato z [12])

5.1.7 Příklad: Blikání LED s využitím GPIO registrů

Tímto je seznamování s GPIO registry u konce a následuje praktické vyzkoušení získaných poznatků. V této části je úkolem vytvořit program pro blikání LED s využitím GPIO registrů, tzn. bez použití třídy *DigitalOut*. Nastavení konfiguračních registrů pro tento účel bylo uvedeno již v kap. 5.1.5. Zbývá tedy měnit výstupní úroveň na pinu PA5, což lze provést pomocí registrů GPIOx_ODR, GPIOx_BSRR a GPIOx_BRR (viz předchozí kapitola). Pro procvičení je žádoucí vyzkoušet všechny možnosti, tzn. rozsvěcet prostřednictvím GPIOx_ODR i GPIOx_BSRR a zhasínat prostřednictvím všech tří registrů.

Před samostatnou realizací výše zmíněného programu je potřeba upozornit na dvě skutečnosti. **První** se týká samotného programování v mbedu. Pro přehlednost by bylo žádoucí uchovávat hodnoty jednotlivých adres v makrech:

```
#define GPIOA 0x48000000
#define MODER_offset 0x00
#define ODR_offset 0x14
#define GPIOA_MODER GPIOA + MODER_offset
#define GPIOA_ODR GPIOA + ODR_offset
```

Tento způsob však (z neznámých důvodů) **není funkční**, i když je hodnota makra *GPIOA_ODR* správná (0x4800 0014), což lze ověřit posláním její hodnoty přes s. port. Nefunkční je pouze zapisování do takovéto adresy. Následující zápis také nefunguje.

```
#define GPIOA_ODR 0x48000000+0x14
```

Proto jediné správné použití maker pro uchování adres (alespoň v mbedu) je přímé napsání adresy. Pro přehlednost je vhodné rozepsat alespoň do komentáře, jak tato adresa vznikla:

```
#define GPIOA_ODR 0x48000014 //GPIOA(0x48000000) + ODR_offset (0x14)
```

Druhá zásadní věc je vytvoření objektu třídy *Serial* v tomto programu z důvodu aktivace hodinového sign. pro bránu GPIOA (stačí pouze tento objekt vytvořit). Pro sériovou komunikaci se předpokládá použití pinů **PA2 a PA3**. Bez této aktivace by periferie GPIO nefungovala. Této „finty“ je potřeba využít, jelikož s hodinovým sign. se seznámíme až v kap. 5.3, kde si také mimo jiné ukážeme, jak aktivovat hodiny pro GPIO.

Nyní je prostor pro samostatné řešení. Výsledné řešení je uvedeno v příloze 12.11. Pro lepší orientaci v programu (např. při ladění programu) je vhodné vypisovat hodnoty registrů před a po jejich modifikaci. K tomuto účelu lze využít buď vlastní výpis, nebo další ladící nástroje vytvořené v práci [1], s nimiž se seznámíme v následující kapitole. Vlastní výpis např. registru GPIOA_ODR v hexadecimálním tvaru může vypadat:

```
serial.printf("GPIOA_ODR: %x\n\r", *((volatile uint32_t *)0x48000014));
```

Pro procvičení práce s registry je vhodné dále vytvořit program zastávající funkci třídy *DigitalIn*, tzn. nakonfigurovat si vlastní pin ve vstupním módu a číst jeho vstupní log. hodnotu. Vytvořený objekt třídy *Serial* však pouští hodinový sign. pouze do brány **GPIOA**, nikoli do bran GPIOB ani GPIOC. Z tohoto důvodu je nutné použít pin na bráně GPIOA. V kap. 5.3 se naučíme pouštět hodinový sign. do jakékoli periferie.

5.2 Ladící nástroje pro práci s registry

V kap. 3.7 jsme se seznámili s ladícími nástroji *Debug_led* a *Debug_serial*, které umožňovali krokování programu. V knihovně *DEBUG_UNIVERSAL* popisované v kap. 6 v [1] se také vyskytují dvě třídy (*Debug_register* a *Debug_register_print*), které jsou určené k práci s registry. Zmíněným registrům je věnována tato kapitola.

Třída *Debug_register* slouží ke krokování programu, výpisu aktuálních hodnot registrů a také umožňuje tyto hodnoty změnit. Jedná se tedy o poměrně sofistikovanou třídu. Její použití je podrobně popsáno v kap. v 6.3 v [1].

Druhá třída *Debug_register_print* je určená pouze k výpisu aktuální hodnoty daného registru. Podrobný přehled probírané třídy je v kap. 6.4 v [1]. Oproti předchozí třídě lze zvolit, v jakém formátu se vypíše hodnota registru, což je pro nastavování registrů velmi užitečné, jelikož programátor nemusí rozepisovat hexadecimální čísla do binární podoby. K definování formátu, v jakém se budou vypisovat hodnoty adres, se využije funkce *format()*. Kupříkladu výpis hodnoty registru GPIOA_ODR v binární podobě vypadá takto (připomeňme, že nejdříve je potřeba importovat knihovnu *DEBUG_UNIVERSAL*, viz kap. 3.7):

```
  …
#include "Debug.h"
  …
#define GPIOA_ODR 0x48000014 //GPIOA(0x48000000) + ODR_offset(0x14)
  …
Debug_register_print device(PA_2, PA_3, 9600);
  …
int main(){
```

```
    device.format(3);
      …
    device.breakpoint(__LINE__, GPIOA_ODR);
      …
}
```

Upozornění: pokud se v programu používá jedna ze zmíněných dvou tříd pro práci s registry a zároveň třída *Serial*, pak musí být u obou používaných tříd nastavena **stejná** přenosová rychlost.

5.3 Hodinový signál

Hodinový signál (zkráceně hodiny) tvoří další nepostradatelnou součást mikrokontroléru. Proč? Vykonání instrukce v mikroprocesoru trvá určitý čas, po jehož uplynutí se přečte výstup a může se vykonávat další instrukce. K řízení tohoto cyklu je potřeba stanovit, kdy se může vykonávat další instrukce, kdy se může číst výstup apod. K tomuto účelu slouží hodinový (dále jen hod.) sign. Analogie: jestliže je mikroprocesor „srdcem“ mikrokontroléru, pak hod. sign. určuje rytmus.

Jak hod. sign. vypadá? Jedná se o elektrický sign. (nejčastěji obdélníkový) se střídou 50 %. Tento sign. generuje tzv. **hodinový generátor** neboli elektronický oscilátor[12]. Na Nucleu se nachází dva typy hod. generátorů: RC obvod a krystalový oscilátor[13]. Základní schéma hod. stromu (angl. clock tree) je na obr. 5.12.

Na začátek je potřeba vybrat hod. generátor, možnosti jsou na obr. 5.12 zvýrazněny červeným rámečkem. Pouze dva z nich jsou pro účely této práce důležité: **8 *MHz* HSI RC** a **4-32 *MHz* HSE OSC**. Zbylé dva, „LSE OSC 32.768 *kHz*“[14] a „LSI RC 49 *kHz*“[15], nebudou dále potřeba.

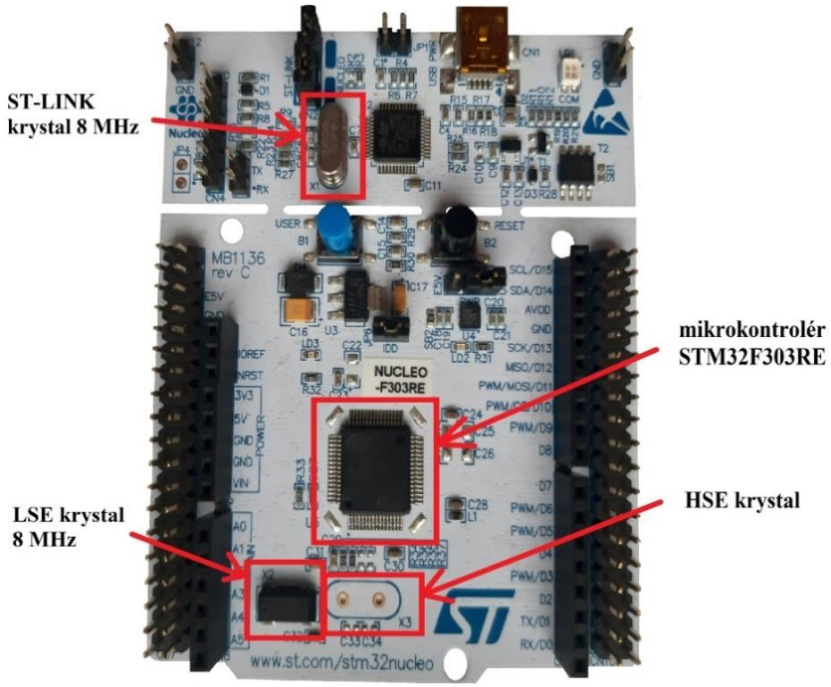
HSI (high speed internal) RC 8 *MHz* je elektronický oscilátor tvořený RC obvodem, jenž generuje hod. sign. o frekvenci 8 *MHz*. Interní znamená, že je uvnitř mikrokontroléru, proto ho najdeme na obr. 4.6. Která součástka na Nucleu je mikrokontrolér ukazuje obr. 5.11. Výhoda HSI oscilátoru je jeho malá cena a v porovnání s HSE oscilátorem rychlejší spouštěcí čas (tzv. startup time). To znamená, že se rychleji dostane do pracovního stavu [12].

[12] Elektronický oscilátor je elektronický obvod generující periodický signál, jehož základ často tvoří RC nebo LC obvod.

[13] Krystalový oscilátor je elektronická součástka využívající piezoelektrický jev, pomocí něhož generuje elektrický (hodinový) signál. Základ krystalového oscilátoru tvoří krystal, který negeneruje přímo obdélníkový sign., avšak obsahuje další elektronické obvody, které ho na obdélníkový sign. transformují. [12]

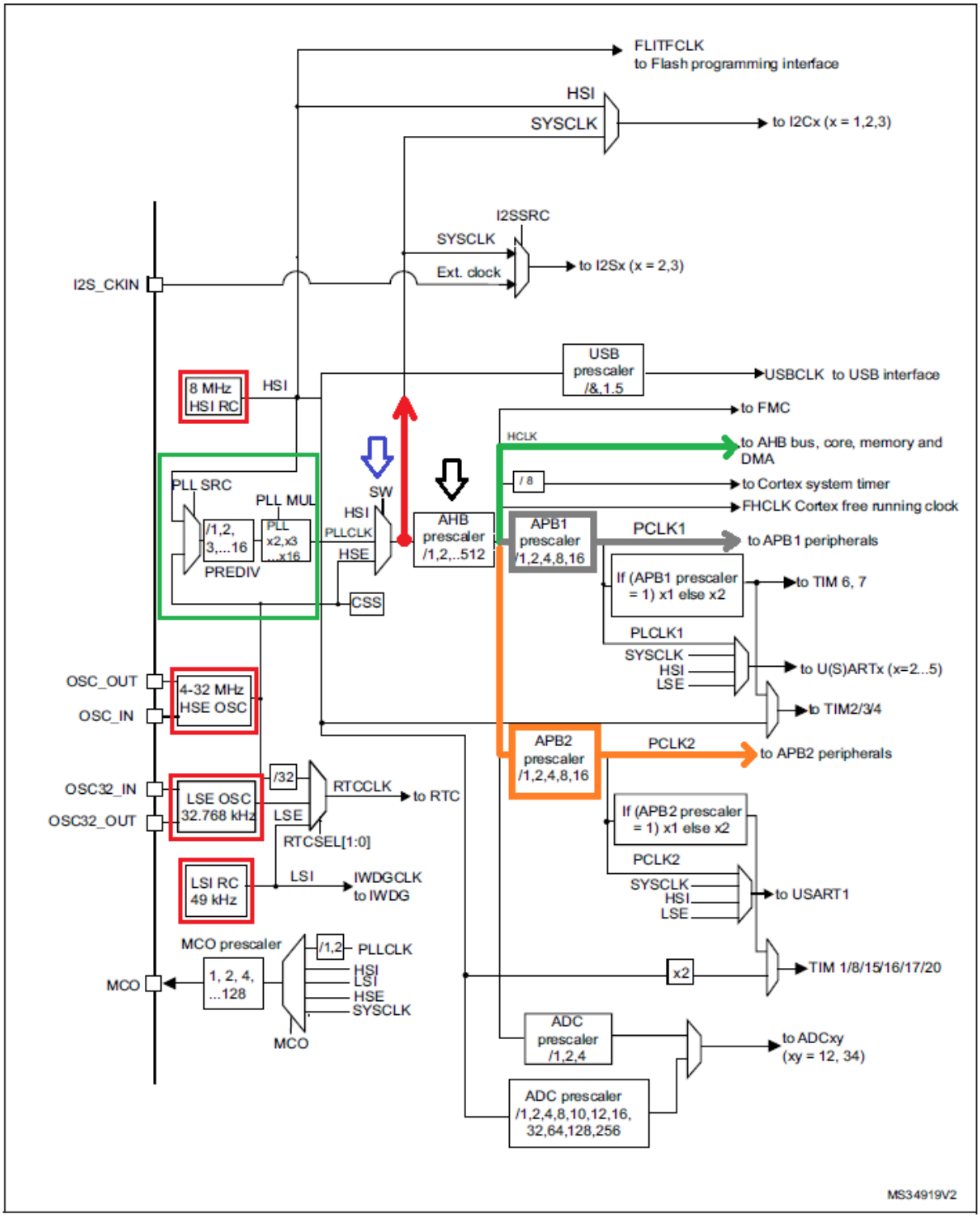
[14] LSE OSC 32.768 *kHz* (low speed external crystal) je low-power hodinový generátor s velkou přesností, vhodný zejména pro použití jako real-time clock. [12]

[15] LSI RC 49 *kHz* (low speed internal) se používá jako hodinový generátor pro watchdog (dohlížecí obvod), což je periferie, která dohlíží na určité procesy (např. v případě využití externího hodinového generátoru může dohlížet na správnost generovaného hodinového sign. a v případě detekce chyby vyvolat přerušení). [12]



Obr. 5.11 Nucleo STM32F303RE hodinové generátory

Přesnější hod. sign. generuje HSE (high speed external) OSC, což je krystalový oscilátor. Nevýhoda tohoto oscilátoru je, že se na Nucleu **nenachází**. Prostor na jeho osazení je připraven, avšak samotný oscilátor chybí (viz obr. 5.11). Existuje však možnost využít hod. sign. z 8 *MHz* krystalu nacházejícím se na ST-LINKu, tzv. **HSE bypass** (viz dále).



Obr. 5.12 STM32F303RE clock tree (převzato z [10])

5.3.1 Bližší popis hodinového stromu

V této části se blíže seznámíme s hod. stromem z obr. 5.12. Již bylo řečeno, že červeným rámečkem jsou zvýrazněny hod. generátory, přičemž pro účely této práce jsou podstatné pouze HSI RC a HSE OSC. Z HSE OSC využijeme konkrétně HSE bypass pro přístup k 8 *MHz* krystalu z ST-LINKu. Jedná se tedy o dva 8 *MHz* hod. generátory. U popisu Nuclea, např. v tom mbedu (viz [8]), se však udává údaj „72 *MHz* max CPU frequency“. Kde se

„vezme“ 72MHz hod. sign., když se na Nucleu nachází pouze 8MHz hod. generátory? Využije se bloku zvaného PLL, který je na obr. 5.12 zvýrazněn zeleným rámečkem.

PLL (phase-locked loop) neboli česky **fázový závěs** je řídicí systém, jenž umožňuje generovat sign., který má násobnou frekvenci než jeho vstupní sign. Přivedením hod. sign. o frekvenci 8 MHz na vstup fázového závěsu tedy lze získat 72 *MHz* hod. sign. Vstup fázového závěsu (**PLL SRC**) je buď HSI, nebo HSE (viz obr. 5.12). Tzv. předděličkou (**PREDIV**) a násobičkou (**PLL MUL**) se nastaví požadovaný násobek. Kupříkladu nastavením PLL SRC na HSI, PREDIV na 1 a PLL MUL na 9 bude fázový závěs generovat hod. sign. **PLLCLK** o frekvenci 72 *MHz* (8 *MHz* * 9). [12]

Hod. sign. PLLCLK vstupuje do bloku, ke kterému na obr. 5.12 směřuje modrá šipka. Jedná se o tzv. multiplexor, což je elektronická součástka vykonávající funkci přepínače, proto i název **SW** (system clock switch) neboli přepínač systémových hodin. V SW se tedy určí, jaký hod. generátor se má použít: HSI, PLLCLK nebo HSE (vstupy multiplexoru). [12]

Vybraný hod. sign. (výstup z multiplexoru SW) se nazývá **SYSCLK** (system clock) neboli systémové hodiny. Ten se pak rozvádí do různých částí mikrokontrolérů, příp. se ještě upravuje jeho frekvence pomocí děličů (angl. **prescaler**). SYSCLK vstupuje do bloku „AHB prescaler /1,2,..512“, ke kterému směřuje černá šipka a také směřuje vzhůru po červené šipce (viz obr. 5.12).

V tomto odstavci se zaměříme na druhý směr (směr vzhůru po červené šipce). SYSCLK vede na vstup do dvou multiplexorů, kde jeden je označen „I2SSRC“ (najděte si je oba ve zmíněném obrázku). V něm se vybírá zdroj hod. sign. pro práci s periférií I2Sx. U druhého multiplexoru, umístěného na obr. 5.12 nejvýše, se vybírá zdroj hod. sign. (HSI nebo SYSCLK) pro periférii I2Cx. Ani jedna z těchto periférií se dále používat nebude, avšak nyní víme, že pro jejich používání je potřeba prostřednictvím zmíněných multiplexorů vybrat zdroj hod. sign.

První směr systémových hodin byl do bloku „AHB prescaler /1,2,..512“. S pojmem AHB jsme se již setkali v kap. 4.4, jedná se totiž o sběrnici. Konkrétně se na Nucleu nachází čtyři sběrnice AHB (AHB1 až AHB4). Každá z nich zajišťuje přístup k určitým periferiím a každá z nich má svůj adresní prostor (viz obr. 5.1). Pro účely této práce je nejdůležitější sběrnice **AHB2**, pomocí které se komunikuje s **bránami** (GPIOA, GPIOB atd.). Dále např. pro práci s AD převodníky je potřeba sběrnice AHB3. Obě si znovu najděte na obr. 4.6.

Z „AHB prescaler /1,2,..512“ se hod. sign. šíří do celého mikrokontroléru. Pro účely této práce jsou vybrány tři nejdůležitější směry: zelený, šedý a oranžový. Zeleným směrem se šíří **HCLK**, což je SYSCLK vydělený AHB prescalerem (defaultně 1), do sběrnice AHB, jádra (**CPU**), paměti a DMA. Šedým/oranžovým směrem hod. sign. postupuje přes blok „APB1 prescaler /1,2,4,8,16“/„APB2 prescaler /1,2,4,8,16“ (vzniká PCLK1/PCLK2) k periferiím APB1/APB2. **Připomeňme**, že nejvyšší možná frekvence na sběrnici AHB2 je 72 *MHz,* zatímco na sběrnici AHB1 pouze 36 *MHz*. Pokud se tedy pomocí fázového závěsu generuje hod. sign. o vyšší frekvenci než 36 *MHz*, musí se příslušně upravit APB1 prescaler.

Na závěr této kap. je zapotřebí říci, že po resetování Nuclea běží jen HSI oscilátor.

5.3.2 Registry pro konfiguraci hodin

Předešlá kapitola obsahovala velké množství teoretických informací týkajících se hod. sign. V této kapitole se seznámíme s registry, ve kterých se uvedené záležitosti nastavují/konfigurují. Stejně jako GPIO brány (a obecně všechny periferie) i hodiny mají svůj adresní prostor. Všechny registry pro práci s hod. sign. jsou součástí adresního prostoru označeného jako „RCC“ neboli „Reset and clock control“. Rozsah adresního prostoru RCC je 0x4002 1000 až 0x4002 13ff (viz obr. 5.1). Počátek tohoto prostoru se tedy nachází na adrese **0x4002 1000** (důležité při programování na úrovni registrů). Částí „reset“ se zabývat nebudeme. Souhrnný název registrů pro nastavení/konfiguraci hodin je **RCC registry**.

Připomeňme, že hod. sign. je pro práci s mikrokontrolérem **zásadní**, protože ho každá periferie potřebuje pro svou činnost. Z toho vyplývá, že jistě existuje velké množství registrů, ve kterých se vše nastavuje/konfiguruje. Tento poznatek je také patrný z obr. 5.12, kde se vyskytuje velké množství bloků, tzn. velké množství parametrů pro nastavování. Konkrétně existuje **13** RCC registrů, avšak pouze se čtyřmi z nich (RCC_CR, RCC_CFGR, RCC_AHBENR a RCC_CFGR2) se seznámíme.

Před popisem prvního registru je vhodné zhruba vědět, co se obecně v RCC registrech nachází. Z obr. 5.12 je evidentní, že se musí vybrat vstupní sign. multiplexorů a nastavit předděličky (prescalers). Dále se ve zmíněných registrech velmi často povolují hodiny pro danou periferii.

Jelikož jsou popisky RCC registrů značně obsáhlé, budou na obrázcích pouze ty, které zastupují (v kontextu této práce) významné oblast. **Celistvý popis** je v kap. 9.4 v [12]. Oblasti s názvem „Res“ jsou, jak již víme z kap. 5.1, rezervované a musí být ponechány na hodnotu, která jim byla přidělena po resetování.

5.3.2.1 Registr RCC_CR

V RCC_CR se zapínají hod. generátory (viz *PPLON*, *HSEON* a *HSION* na obr. 5.13). Navíc je možné aktivovat **HSE bypass** (viz kap. 5.3) prostřednictvím *HSEBYP*. Tento bit však musí být nastaven společně s bitem *HSEON* (viz popisek u *HSEBYP* na obr. 5.13), to si ukážeme dále v kap. 5.4.2. V případě aktivace HSE bypassu se použije vnější hod. sign. přivedený na pin *OSC_IN*. Pokud se aktivuje HSE bez HSE bypassu, pak se použije hod. sign. z krystalového oscilátoru prostřednictvím pinů OSC_IN a OSC_OUT. **Najděte** si zmíněné piny na obr. 5.12.

Po resetování jsou bity *HSIRDY* a *HSION* aktivní (viz resetovací hodnota RCC_CR), tedy po resetování běží hod. generátor HSI 8 *MHz*. Ostatní (pro nás potřebné) bity jsou v nule. V případě aktivace jiného hod. generátoru (*PPLON* nebo *HSEON*) je **vždy** potřeba počkat na potvrzující bit (*PLLRDY* nebo *HSERDY*). Tyto bity jsou nastaveny hardwarově a značí, že je příslušný hod. generátor připraven na použití. Hod. generátory tedy **nejsou** připraveny hned, ale musí se chvíli čekat. V kódu se zmíněné čekání provádí jednoduše v nekonečném cyklu.

Jak ale číst hodnotu konkrétního bitu? Využije se k tomu tzv. **maskování**. To se provede přečtením hodnoty dané adresy a log. součinem s tzv. **maskou bitu**. Maska bitu je 1 posunutá o daný počet bitů (v kap. 5.1.4 jsme se s ní již setkali, avšak nebyla takto pojmenována).

Kupříkladu čekání v nekonečné smyčce na aktivaci hod. generátoru HSE pomocí maskování bitu *HSERDY* vypadá takto:

```
  …
#define RCC_CR 0x40021000  //RCC(0x40021000) + CR_offset(0x00)
#define MASK_HSERDY 1<<17 //mask of the HSERDY bit
  …
int main() {
    …
  //wait until the HSE is ready
  while(!(*((volatile uint32_t *)RCC_CR) & MASK_HSERDY)){}
    …
}
```

Stejným způsobem by se čekalo na potvrzující bit *PLLRDY*. V RCC_CR se tedy hod. generátor **pouze spouští**.

5.3.2.2 Registr RCC_CFGR

Druhý stěžejní registr je RCC_CFGR neboli „clock configuration register“, který slouží pro konfiguraci hodin (viz obr. 5.14 a obr. 5.15). Obecně se v registru nachází oblasti pro konfiguraci PLL, předděliček (AHB, APB1, APB2) a systémových hodin. Dále také oblasti pro konfiguraci MCO, I2S a USB, které však nejsou pro tuto práci důležité.

V oblasti konfigurace PLL (bity 15 až 21) se nastavuje zdroj hod. sign. *PLLSRC* (bity 15-16), předdělička *PLLXTPRE* (bit 17) a násobička *PLLMUL* (bity 18-21). Mezi bity 4 až 13 se stanovuje hodnota předděličeк sběrnic AHB, APB1, APB2 (viz *HPRE*, *PRE1* a *PRE2* na obr. 5.15). Připomeňme, že maximální povolená frekvence na APB1 je 36 *MHz* a na APB2 72 *MHz*. V oblasti *SW* (bity 0-1) se vybírají systémové hodiny a v oblasti SWS (bity 2-3) lze ověřit, který hod. generátor je vybrán jako SYSCLK. Většinu ze zmíněných parametrů je možné dohledat na obr. 5.12.

Po nastavení zmíněných oblastí je puštěn hod. sign. do CPU, sběrnic a dalších částí mikrokontroléru. Dále ho je však potřeba pustit do jednotlivých periferií, což se pro GPIO odehrává v následujícím registru.

5.3.2.3 Registr RCC_AHBENR

Registr RCC_AHBENR neboli „AHB peripheral clock enable register“ (viz obr. 5.16) umožňuje pustit hod. sign. do periferií, jež se nacházejí na sběrnicích AHB (AHB1 až AHB4). Připomeňme, že právě na jedné z těchto sběrnic se nachází i GPIO brány. **Ověřte** si tyto informace na obr. 5.1 a také na obr. 4.6.

Pro použití nějakého pinu např. na bráně GPIOC se tedy v RCC_AHBENR musí pustit hod. sign. do této brány prostřednictvím *IOPCEN* (viz obr. 5.16). V případě, že tak neučiníme, **brána nebude fungovat**. Z toho vyplývá důležité **pravidlo** týkající se práce s periferiemi: každá periferie potřebuje pro svoji funkci povolit hodiny. Poznamenejme, že v kap. 5.1.7 se povolily hodiny na bráně GPIOA prostřednictvím vytvoření objektu třídy *Serial* (na pinech PA2 a PA3). Nyní se již bez této „finty“ obejdeme.

9.4.1 Clock control register (RCC_CR)

Address offset: 0x00

Reset value: 0x0000 XX83 where X is undefined.

Access: no wait state, word, half-word and byte access

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	Res	Res	Res	Res	PLL RDY	PLLON	Res	Res	Res	Res	CSS ON	HSE BYP	HSE RDY	HSE ON
						r	rw					rw	rw	r	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HSICAL[7:0]								HSITRIM[4:0]					Res	HSI RDY	HSION
r	r	r	r	r	r	r	r	rw	rw	rw	rw	rw		r	rw

●●●

Bit 25 **PLLRDY:** PLL clock ready flag

Set by hardware to indicate that the PLL is locked.

0: PLL unlocked

1: PLL locked

Bit 24 **PLLON:** PLL enable

Set and cleared by software to enable PLL.

Cleared by hardware when entering Stop or Standby mode. This bit can not be reset if the PLL clock is used as system clock or is selected to become the system clock.

0: PLL OFF

1: PLL ON

●●●

Bit 18 **HSEBYP:** HSE crystal oscillator bypass

Set and cleared by software to bypass the oscillator with an external clock. The external clock must be enabled with the HSEON bit set, to be used by the device. The HSEBYP bit can be written only if the HSE oscillator is disabled.

0: HSE crystal oscillator not bypassed

1: HSE crystal oscillator bypassed with external clock

Bit 17 **HSERDY:** HSE clock ready flag

Set by hardware to indicate that the HSE oscillator is stable. This bit needs 6 cycles of the HSE oscillator clock to fall down after HSEON reset.

0: HSE oscillator not ready

1: HSE oscillator ready

Bit 16 **HSEON:** HSE clock enable

Set and cleared by software.

Cleared by hardware to stop the HSE oscillator when entering Stop or Standby mode. This bit cannot be reset if the HSE oscillator is used directly or indirectly as the system clock.

0: HSE oscillator OFF

1: HSE oscillator ON

●●●

Bit 1 **HSIRDY:** HSI clock ready flag

Set by hardware to indicate that HSI oscillator is stable. After the HSION bit is cleared, HSIRDY goes low after 6 HSI oscillator clock cycles.

0: HSI oscillator not ready

1: HSI oscillator ready

Bit 0 **HSION:** HSI clock enable

Set and cleared by software.

Set by hardware to force the HSI oscillator ON when leaving Stop or Standby mode or in case of failure of the HSE crystal oscillator used directly or indirectly as system clock. This bit cannot be reset if the HSI is used directly or indirectly as system clock or is selected to become the system clock.

0: HSI oscillator OFF

1: HSI oscillator ON

Obr. 5.13 Registr RCC_CR (převzato z [12])

9.4.2 Clock configuration register (RCC_CFGR)

Address offset: 0x04

Reset value: 0x0000 0000

Access: 0 ≤ wait state ≤ 2, word, half-word and byte access

1 or 2 wait states inserted only if the access occurs during clock source switch.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PLLNODIV	MCOPRE[2:1]		MCOF/MCOPRE0	Res	MCO[2:0]			I2SSRC	USBPRE	PLLMUL[3:0]				PLLXTPRE	PLLSRC
rw	rw	rw	r / 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
PLLSRC(1)	Res	PPRE2[2:0]			PPRE1[2:0]			HPRE[3:0]				SWS[1:0]		SW[1:0]	
rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	r	r	rw	rw

1. STM32F303xD/E only

●●●

Bits 21:18 **PLLMUL:** PLL multiplication factor

These bits are written by software to define the PLL multiplication factor. These bits can be written only when PLL is disabled.

Caution: The PLL output frequency must not exceed 72 MHz.

0000: PLL input clock x 2
0001: PLL input clock x 3
0010: PLL input clock x 4
0011: PLL input clock x 5
0100: PLL input clock x 6
0101: PLL input clock x 7
0110: PLL input clock x 8
0111: PLL input clock x 9
1000: PLL input clock x 10
1001: PLL input clock x 11
1010: PLL input clock x 12
1011: PLL input clock x 13
1100: PLL input clock x 14
1101: PLL input clock x 15
1110: PLL input clock x 16
1111: PLL input clock x 16

Bit 17 **PLLXTPRE:** HSE divider for PLL input clock

This bits is set and cleared by software to select the HSE division factor for the PLL. It can be written only when the PLL is disabled.

Note: This bit is the same as the LSB of PREDIV in Clock configuration register 2 (RCC_CFGR2) (for compatibility with other STM32 products)

0000: HSE input to PLL not divided
0001: HSE input to PLL divided by 2

Bits 16:15 **PLLSRC**: PLL entry clock source (STM32F303xD/E and STM32F398xE only)

Set and cleared by software to select PLL clock source. These bits can be written only when PLL is disabled.

00: HSI/2 used as PREDIV1 entry and PREDIV1 forced to div by 2.
01: HSI used as PREDIV1 entry.
10: HSE used as PREDIV1 entry.
11: Reserved.

Obr. 5.14 Registr RCC_CFGR (převzato z [12])

●●●

Bits 13:11 **PPRE2:** APB high-speed prescaler (APB2)
Set and cleared by software to control the division factor of the APB2 clock (PCLK).
0xx: HCLK not divided
100: HCLK divided by 2
101: HCLK divided by 4
110: HCLK divided by 8
111: HCLK divided by 16

Bits 10:8 **PPRE1:** APB Low-speed prescaler (APB1)
Set and cleared by software to control the division factor of the APB1 clock (PCLK).
0xx: HCLK not divided
100: HCLK divided by 2
101: HCLK divided by 4
110: HCLK divided by 8
111: HCLK divided by 16

Bits 7:4 **HPRE:** HLCK prescaler
Set and cleared by software to control the division factor of the AHB clock.
0xxx: SYSCLK not divided
1000: SYSCLK divided by 2
1001: SYSCLK divided by 4
1010: SYSCLK divided by 8
1011: SYSCLK divided by 16
1100: SYSCLK divided by 64
1101: SYSCLK divided by 128
1110: SYSCLK divided by 256
1111: SYSCLK divided by 512
Note: The prefetch buffer must be kept on when using a prescaler different from 1 on the AHB clock. Refer to section Read operations on page 66 for more details.

Bits 3:2 **SWS:** System clock switch status
Set and cleared by hardware to indicate which clock source is used as system clock.
00: HSI oscillator used as system clock
01: HSE oscillator used as system clock
10: PLL used as system clock
11: not applicable

Bits 1:0 **SW:** System clock switch
Set and cleared by software to select SYSCLK source.
Cleared by hardware to force HSI selection when leaving Stop and Standby mode or in case of failure of the HSE oscillator used directly or indirectly as system clock (if the Clock Security System is enabled).
00: HSI selected as system clock
01: HSE selected as system clock
10: PLL selected as system clock
11: not allowed

Obr. 5.15 Registr RCC_CFGR pokračování (převzato z [12])

9.4.6 AHB peripheral clock enable register (RCC_AHBENR)

Address offset: 0x14

Reset value: 0x0000 0014

Access: no wait state, word, half-word and byte access

Note: When the peripheral clock is not active, the peripheral register values may not be readable by software and the returned value is always 0x0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	Res	ADC34 EN	ADC12EN	Res	Res	Res	TSCEN	IOPG EN(1)	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	IOPH EN(1)
		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	Res	Res	Res	CRC EN	FMC EN(1)	FLITF EN	Res	SRAM EN	DMA2 EN	DMA1 EN
									rw	rw	rw		rw	rw	rw

1. Only on STM32F303xDxE.

●●●

Bit 19 **IOPCEN:** I/O port C clock enable
Set and cleared by software.
0: I/O port C clock disabled
1: I/O port C clock enabled

Bit 18 **IOPBEN:** I/O port B clock enable
Set and cleared by software.
0: I/O port B clock disabled
1: I/O port B clock enabled

Bit 17 **IOPAEN:** I/O port A clock enable
Set and cleared by software.
0: I/O port A clock disabled
1: I/O port A clock enabled

●●●

Obr. 5.16 Registr RCC_AHBENR (převzato z [12])

5.3.2.4 Registr RCC_CFGR2

Registr RCC_CFGR2 je druhý konfigurační registr, ve kterém se definuje hodnota předděliček AD převodníků a hlavně PLL předděličky. Zmíněný registr uvádíme pouze kvůli této oblasti, která je na obr. 5.17 značena *PREDIV* (bity 0 až 3). Pod stejným názvem je zobrazena na obr. 5.12 a uvedena v kap. 5.3.1.

Na Nucleu (F303RE) lze zvolit „0000“, což znamená, že se vstupní hod. generátor HSE nebude dělit. Tato možnost však u jiných typů Nuclea (s jiným typem mikrokontroléru) není pravidlem.

9.4.12 Clock configuration register 2 (RCC_CFGR2)

Address: 0x2C

Reset value: 0x0000 0000

Access: no wait states, word, half-word and byte access

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	ADC34PRES[4:0]					ADC12PRES[4:0]					PREDIV[3:0]			
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

•••

Bits 3:0 **PREDIV**: PREDIV division factor

These bits are set and cleared by software to select PREDIV division factor. They can be written only when the PLL is disabled.

Note: Bit 0 is the same bit as bit17 in Clock configuration register (RCC_CFGR), so modifying bit17 Clock configuration register (RCC_CFGR) also modifies bit 0 in Clock configuration register 2 (RCC_CFGR2) (for compatibility with other STM32 products)

0000: HSE input to PLL not divided
0001: HSE input to PLL divided by 2
0010: HSE input to PLL divided by 3
0011: HSE input to PLL divided by 4
0100: HSE input to PLL divided by 5
0101: HSE input to PLL divided by 6
0110: HSE input to PLL divided by 7
0111: HSE input to PLL divided by 8
1000: HSE input to PLL divided by 9
1001: HSE input to PLL divided by 10
1010: HSE input to PLL divided by 11
1011: HSE input to PLL divided by 12
1100: HSE input to PLL divided by 13
1101: HSE input to PLL divided by 14
1110: HSE input to PLL divided by 15
1111: HSE input to PLL divided by 16

Obr. 5.17 Registr RCC_CFGR2 (převzato z [12])

5.3.2.5 Shrnutí RCC registrů

V předešlých čtyřech kapitolách došlo k seznámení s registry RCC_CR, RCC_CFGR, RCC_AHBENR a RCC_CFGR2. V RCC_CR se daný hod. generátor aktivuje a v RCC_CFGR konfiguruje. Registr RCC_AHBENR umožňuje zapnout hod. sign. do konkrétní periferie na sběrnicích AHB (AHB1 až AHB4) a v RCC_CFR2 lze předdělit vstupní hog. sign. z HSE.

Zbývá 9 RCC registrů. S některými z nich se v této části zběžně seznámíme. V RCC registrech se nachází další konfigurační registr RCC_CFGR3, v němž se vybírá zdroj hod. sign. komunikačních periferií a také některých Timerů.

Funkce registrů RCC_APB1ENR a RCC_APB2ENR je podobná funkci RCC_AHBENR, tedy povolují se v nich hodiny pro periferie, tentokrát nacházející se na sběrnicích APB1 a APB2. K resetování hodin určité periferie na sběrnicích AHB, APB1, APB2 slouží registry RCC_AHBRTSR , RCC_APB1RSTR a RCC_APB2RSTR.

Tímto byly probrány **téměř** všechny pro tuto práci potřebné registry. Zbývá seznámení s jedním registrem, bez kterého bychom se pravděpodobně neobešli.

5.3.3 Registr FLASH_ACR

FLASH_ACR, jak označení napovídá, patří do registrů pracující s paměti FLASH, tzv. FLASH registrů. Počátek adresního prostoru těchto registrů je **0x4002 2000** (viz oblast „Flash interface“ na obr. 5.1). Registr FLASH_ACR (viz obr. 5.18) je potřeba znát při programování prostřednictvím registrů, protože obsahuje oblast, ve které se definuje zpoždění. Přesněji se v této oblasti stanovuje, jak dlouho se po zadání požadované adresy musí čekat, než lze na výstupu paměti FLASH číst hodnotu z žádané adresy. Tato prodleva se počítá ve formě tzv. čekacích stavů (angl. wait states). [39]

Proč je potřeba vkládat prodlevu mezi zadáním požadované adresy a čtením? Zjednodušeně řečeno, paměť FLASH není tak rychlá, jak může být CPU. Pro názornost si lze představit paměť FLASH jako černou skříňku, které zadáme adresu a pak čekáme na výstupní hodnotu. Pokud je adresa zadávána příliš často (s velkou frekvencí), pak se musí na výstup ze skříňky čekat.

V případě, kdy je HCLK (viz kap. 5.3.1) do 24 *MHz* (viz obr. 5.18), pak se žádná prodleva vkládat nemusí (skříňka zvládá dostatečně rychle „dávat“ hodnoty daných adres na výstup). Pokud je frekvence HCLK vyšší než zmíněných 24 *MHz*, pak se prodleva vkládat musí (skříňka nestíhá).

4.5.1 Flash access control register (FLASH_ACR)

Address offset: 0x00
Reset value: 0x0000 0030

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.	PRFTBS	PRFTBE	HLFCYA	LATENCY[2:0]		
										r	rw	rw	rw	rw	rw

•••

Bits 2:0 **LATENCY[2:0]**: Latency
These bits represent the ratio of the HCLK period to the Flash access time.
000: Zero wait state, if 0 < HCLK ≤ 24 MHz
001: One wait state, if 24 MHz < HCLK ≤ 48 MHz
010: Two wait sates, if 48 < HCLK ≤ 72 MHz

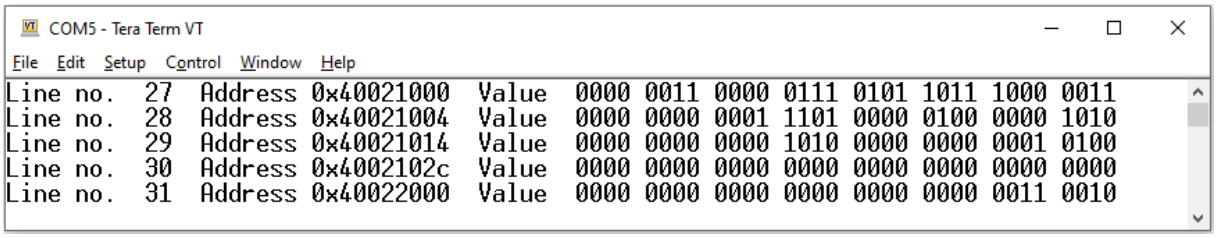
Obr. 5.18 Registr FLASH_ACR (převzato z [12])

5.3.4 Základní nakonfigurování hodin v mbedu

Tímto jsou probrány všechny registry (kromě registrů procesoru), které budou v této práci potřeba. Než si nastavíme vstupně výstupní pin **celkově** samostatně (bez použití knihovny *mbed.h*), podíváme se, jak jsou hodiny nastaveny v mbedu. Pro názornost vytvořme objekt třídy *DigitalIn* (na pinu brány GPIOC, např. PC10) a třídy *Debug_register_print* (na pinech PA2 a PA3):

```
DigitalIn my_PC10 (PC10);
Debug_register_print device(PA_2, PA_3, 9600);
```

Cílem je poslat hodnoty registrů RCC_CR, RCC_CFGR, RCC_AHBENR, RCC_CFGR2 a FLASH_ACR prostřednictvím třídy *Debug_register_print* a to přímo v binární podobě. Výsledný program je v příloze 12.12. Získané hodnoty zmíněných registrů jsou na obr. 5.19, těmi se budeme dále zabývat.



Obr. 5.19 Hodnoty vybraných registrů

V registru RCC_CR (0x40021000) jsou aktivní všechny bity probírané v kap. 5.3.1, tzn. je spuštěný HSI, HSE (i HSE bypass) a PLL. Zároveň jsou aktivní potvrzující sign. (*HSIRDY*, *HSERDY* a *PLLRDY*).

Z konfiguračního registru RCC_CFGR (0x4002 1004) je patrné, že pro funkci systémových hodin je vybrán fázový závěs. Předdělička sběrnice AHB (*HPRE*) a APB2 (*PPRE2*) je jedna (nepředděluje se), zatímco předdělička APB1 (PPRE1) je zvolena „100“, což odpovídá vydělení HCLK dvěma (viz obr. 5.15). Zdroj PLL (*PLLSRC*) je „01“ neboli HSE. Násobička PLLMULL je „0111“, což odpovídá vynásobení frekvence vstupního signálu **devíti**. Znamená to, že PLL generuje hod. sign. o frekvenci 72 *MHz* (9x8 *MHz*)? Musíme se ještě podívat na oblast *PREDIV* v registru RCC_CFGR2 (0x4002 102c). Ta je nulová, tedy skutečně PLL generuje hod. sign. PLLCLK o frekvenci 72 *MHz*.

V kap. 5.3.3 však bylo řečeno, že pokud je frekvence HCLK větší než 24 MHz, pak se musí nakonfigurovat FLASH_ACR (0x4002 2000) pro vložení „wait states“. Frekvence HCLK odpovídá frekvenci SYSCLK vydělené předděličkou AHB (též označovanou jako předdělička HCLK). Z minulé kapitoly víme, že tato předdělička je nastavena na 1 (nepředděluje se), proto je i frekvence HCLK 72 *MHz*. Z toho vyplývá potřeba vložení dvou „wait states“ (viz kap. 5.3.3). V registru FLASH_ACR je hodnota prvních třech bitů „010“ (viz obr. 5.19), což odpovídá přidání dvou „wait states“. Vše je tedy nastaveno podle očekávání.

Poslední doposud nezmíněný registr, který je zobrazen na obr. 5.19, je RCC_AHBENR (0x4002 1014). V tomto registru jsou aktivní čtyři bity: *SRAMEN*, *FLITFEN*, *IOPAEN* a *IOPCEN*. Povolení hod. sign. do GPIOC (*IOPAEN*) došlo ve třídě *DigitalIn* a do GPIOA (*IOPCEN*) ve třídě *Debug_register_print* (kvůli použití pinů PA2 a PA3 pro sériovou komunikaci). Zbylé dva bity jsou nastaveny defaultně (viz resetovací hodnota na obr. 5.16).

5.4 Blikání LED bez použití knihovny mbed.h

Na závěr této kapitoly shrneme získané znalosti do třech programů, které budou mít stejnou funkci, blikat uživatelskou LED, avšak použijí k tomu tři různé hod. generátory. K jejich implementaci však není povoleno použití knihovny *mbed.h*. V podstatě se jedná o „více nezávislý“ příklad řešeného v kap. 5.1.7.

Nepoužitím knihovny mbed.h vyvstanou nejméně dva problémy. První se týká nedefinovaného datového typu *uint32_t* (a jemu podobným), který se vyřeší jednoduše přidáním standartní knihovny s*tdint.h*. Druhý problém je s funkcí *wait()*, která je definována v knihovně *mbed.h*, tudíž ji nelze použít. Tuto záležitost vyřešíme „obklikou“ využitím funkce *HAL_Delay()*, která je součásti tzv. HAL knihovny *stm32f3xx_hal.h*, s níž se blíže seznámíme v kap. 6.2. Argument odpovídá počtu milisekund.

5.4.1 Příklad: Blikání LED s využitím hodinového generátoru HSI

První část je velmi jednoduchá, jelikož hod. generátor HSI je nastaven defaultně, tedy systémové hodiny **není** potřeba nastavovat. Hodiny jsou tedy aktivní, stačí je pouze pustit do brány GPIOA:

```
#include <stdint.h>
#include "stm32f3xx_hal.h"
#define RCC_AHBENR 0x40021014 //RCC(0x40021000) + AHBENR_offset(0x14)
#define GPIOA_MODER 0x48000000 //GPIOA(0x48000000) + MODER_offset(0x00)
#define GPIOA_ODR 0x48000014   //GPIOA(0x48000000) + ODR_offset(0x14)
int main() {
   *((volatile uint32_t *)RCC_AHBENR) |= 0b1 << 17; //enable clock to GPIOA
   *((volatile uint32_t *)GPIOA_MODER) |= 0b01 << 2*5;
   while(1){
      *((volatile uint32_t *)GPIOA_ODR) |=  0b1 << 5; //switch the LED on
      HAL_Delay(1000);
      *((volatile uint32_t *)GPIOA_ODR) &=  ~(0b1 << 5); //switch the LED off
      HAL_Delay(1000);
   }
}
```

5.4.2 Příklad: Blikání LED s využitím hodinového generátoru HSE bypass

Tento příklad je obdobný předchozímu příkladu, avšak jako systémové hodiny využijeme **HSE bypass**. Za tímto účelem stačí definovat konstanty:

```
#define RCC_CR 0x40021000 //RCC(0x40021000) + CR_offset(0x0)
#define RCC_CFGR 0x40021004 //RCC(0x40021000) + CFGR_offset(0x04)
#define MASK_HSERDY 1<<17 //Mask of the HSERDY bit
```

Dále se aktivují bity *HSEON* a *HSEBYP* v RCC_CR (viz kap. 5.3.2.1) a čeká se na potvrzující sign. *HSERDY* ve stejném registru. Nakonec se vyberou HSE jako systémové hodiny

v RCC_CFGR. Výsledný program vcelku je v příloze 12.13. Zmíněné úkony se provedou takto:

```
*((volatile uint32_t *)RCC_CR) |= 0b101 << 16; //set HSEBYP and HSEON
while(!(*((volatile uint32_t *)RCC_CR) & MASK_HSERDY)){} //is the HSE ready?
*((volatile uint32_t *)RCC_CFGR) |= 0b01; //select HSE as system clock
```

5.4.3 Příklad: Blikání LED s využitím hodinového generátoru PLL

Třetí a poslední případ je s využitím hod. generátoru PLL. V kap. 5.3.4 bylo ukázáno, jak vypadají probírané registry nakonfigurované pro generování SYSCLK o frekvenci 72 *MHz*. V této části zapíšeme hodnoty, které byly ve zmíněné kapitole zjištěny.

Před aktivací fázového závěsu se musí nakonfigurovat příslušné oblasti (*PLLMUL*, *PLLSRC* a může se rovnou i předdělička *PPRE1*) v RCC_CFGR případně v RCC_CFGR2 (*PREDIV*). Oblast *PREDIV* stačí nechat na resetovací hodnotě „0000“. Zmíněné operace se provedou takto:

```
//these bits can be written only when PLL is disabled:
*((volatile uint32_t *)RCC_CFGR) |= 0b0111 << 18; //set PLLMUL -> „0111“ = x 9
*((volatile uint32_t *)RCC_CFGR) |= 0b10 << 15; //set PLLSRC
*((volatile uint32_t *)RCC_CFGR) |= 0b100 << 8; //set PRE1
```

Po konfiguraci v RCC_CFGR následuje aktivace HSE a HSE bypassu v RCC_CR, kde se před pokračováním, stejně jako v předchozí kapitole, musí vyčkat na kontrolní sign. *HSERDY*. Poté se ve stejném registru aktivuje i hod. generátor PLL a opět se čeká na aktivaci kontrolního sign. *PLLRDY*.

Na závěr je potřeba nastavit vložení dvou „wait states“ do oblasti *LATENCY* v registru FLASH_ACR a zdroj systémových hodin na PLL:

```
*((volatile uint32_t *)FLASH_ACR) |= 0b010; //add two wait states
*((volatile uint32_t *)RCC_CFGR) |= 0b10; //select the PLL as system clock
```

Zbytek programu je stejný jako v kap. 5.4.1. Výsledný program je v příloze 12.14.

6 Programování s využitím struktur, HAL a LL knihovny a ARM assembleru

Cílem předchozí kapitoly bylo získání bližší představy o vnitřní struktuře mikrokontroléru a nadhledu nad způsobem, jak s touto strukturou pracovat. V této kapitole se seznámíme s jiným způsobem přístupu k registrům a HAL a LL knihovnami, jakožto zástupcem skutečného profesionálního prostředku pro programování mikrokontroléru.

6.1 Využití struktur pro přístup k registrům

V kap. 5.1.4 proběhlo seznámení s tzv. **přímou metodou**, která je sice „zdlouhavá“, avšak názornější. Této metodě se stručně věnuje i kap. 10.1 v [1]. Kupříkladu povolení hod. sign. v registru RCC_AHBENR vypadá:

```
#define RCC_AHBENR 0x40021014 //RCC(0x40021000) + AHBENR_offset(0x14)
  …
*((volatile uint32_t *)RCC_AHBENR) |= 0b1 << 17; //enable clock to GPIOA
```

Zkrácení stejné akce:

```
*((volatile uint32_t *)0x40021014) |= 0b1 << 17; //enable clock to GPIOA
```

První způsob je přehlednější. Právě kvůli **přehlednosti** jsme adresy uchovávali v makrech. Mbed nabízí další metodu, určenou speciálně pro přístup k registrům, která využívá struktur. Celkový přehled najdeme přímo na stránkách mbedu v [14]. Jedná se o velmi objemný soubor, ve kterém jsou definovány všechny struktury pro zmíněnou metodu.

Povolení hod. sign v registru RCC_AHBENR se vykoná:

```
RCC->AHBENR |= 0b1 << 17;
```

Tato metoda nabízí mnoho výhod, mezi které patří zvětšení přehlednosti kódu, jeho zmenšení a absence hledání a vypisování adres.

6.1.1 Příklad: Implementace příkladu z kap. 5.4.1 pomocí struktur

Úkolem je procvičit si využití struktur pro přístup k registrům na příkladu z kap. 5.4.1. Výsledný kód se značně zkrátí a zpřehlední. Poznamenejme, že u každého registry jsou také definovány jednotlivé možnosti, které se dají zapsat do každého registru. Kupříkladu povolení hod. sign. v registru RCC_AHBENR lze také vykonat:

```
RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
```

I když je soubor [14] značně rozsáhlý, dají se v něm hledat zmíněné možnosti prostřednictvím klávesové zkratky „ctrl+f“. Stačí vyhledávat název daného registru. Pro registr RCC_AHBENR jsou jednotlivé možnosti definovány pod nadpisem „Bit definition for RCC_AHBENR register“.

Program bez použití zmíněných bitových definic:

```
#include "stm32f3xx_hal.h"
int main() {
   RCC->AHBENR |= RCC_AHBENR_GPIOAEN; //enable clock to GPIOA
   GPIOA->MODER |= 0b01 << 2*5;
   while(1){
      GPIOA->ODR |=  0b1 << 5; //switch the LED on
      HAL_Delay(1000);
      GPIOA->ODR &=  ~(0b1 << 5); //switch the LED off
      HAL_Delay(1000);
   }
}
```

S využitím zmíněných definic:

```
#include "stm32f3xx_hal.h"
int main() {
   RCC->AHBENR |= RCC_AHBENR_GPIOAEN; //enable clock to GPIOA
   GPIOA->MODER |= GPIO_MODER_MODER5_0;
   while(1){
      GPIOA->ODR |=  GPIO_ODR_5; //switch the LED on
      HAL_Delay(1000);
      GPIOA->ODR &=  ~(GPIO_ODR_5); //switch the LED off
      HAL_Delay(1000);
   }
}
```

6.1.2 Příklad: Implementace příkladu z kap. 5.4.3 pomocí struktur

Pro procvičení metody přístupu k registrům pomocí struktur si sami naimplementujte příklad z kap. 5.4.3. Využijte co nejvíce bitových definic. Výsledný program je v příloze 12.15.

6.2 HAL a LL knihovny

Součástí mbedu jsou také HAL a LL knihovny, kterým je věnována tato kapitola. Jedná se o nástroj, který se využívá i na **profesionální** úrovni, zejména pro inicializaci/konfiguraci periferií. Tyto knihovny jsou však velmi obsáhlé a jejich použití vyžaduje praxi. Cílem tak **není** naučit se programovat pomocí HAL a LL knihoven, ale pouze **seznámit** se s nimi.

Mbed, jak již bylo uvedeno, **zdaleka** neumožňuje využít všechny dostupné funkce periferií. V případě, kdy potřebujeme využít něco (funkci nebo celou periferii), co mbed nepodporuje, máme dvě možnosti. První je kompletní doimplementování dané problematiky a druhá využití HAL nebo LL knihovny. Čas strávený na jedné či druhé variantě však nemusí být tak rozdílný, zejména pokud s ani jednou možností nemáme moc zkušeností.

HAL (**hardware** abstraction layer) knihovny jsou **vysokoúrovňové** knihovny, které poskytují vysokou míru abstrakce. Jejich hlavní výhodou je **multiplatformnost** (dobrá přenositelnost**)** v rámci **STM32**. To znamená, že program vykonávající danou funkci na jednom MCU se

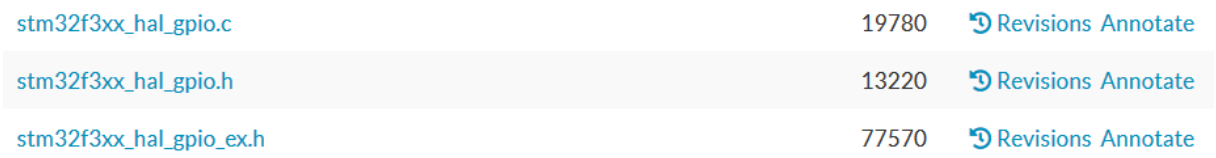
tolik neliší od kódu vykonávajícího stejnou funkci na jiném MCU (obě MCU z rodiny STM32). Jelikož se jedná o vysokoúrovňové knihovny, programátor nemusí řešit detaily pro zprovoznění periferií. Konfigurace se tak stává rychlejší. [38]

Vysoká míra abstrakce v rámci má však i své nevýhody. Díky ní jsou HAL knihovny velmi obsáhlé a kód se tak stává méně přehledný. Čas pro naučení konkrétní HAL knihovny tak může být srovnatelný s časem stráveným na pochopení, jak daný hardware funguje. Mezi další nevýhody patří občasný výskyt errorů a buggů. Kód se v takovém případě těžko debugguje, protože se neví, jestli je chyba v kódu programátora nebo v HAL knihovně.

HAL knihovny např. pro svoji činnost využívá program **STM32Cube-MX**, což je nástroj firmy STMicroelectronics pro programování mikrokontrolérů STM32. Poznamenejme, že tento program umožňuje velmi snadnou konfiguraci periferií.

Knihovny LL (low level) jsou **nízkoúrovňové** knihovny, které slouží pro konfiguraci periferií. Nepracují přímo s registry, ale zachovávají všechny funkce dané periferie. Díky blízkému vztahu k periferiím jsou však méně přenositelné, což je jeden z hlavních rozdílů oproti HAL knihovnám. V rámci této práce je však rozdíl mezi nimi nepatrný. [38]

Jak již bylo uvedeno v kap. 4.1, všechny HAL a LL knihovny pro Nucleo jsou v mbedu umístěny v [15]. Název jednotlivých knihoven vždy začíná specifikováním mikrokontroléru, to je v našem případě „stm32xx“. Následuje řetězec „hal“ oddělený podtržítkem. Za dalším podtržítkem se nachází název periferie, které se daná knihovna věnuje. Kupříkladu pro GPIO jsou jednotlivé programy vidět na obr. 6.1. Většinou se u periferie také nachází hlavičkový soubor s dovětkem „_ex“, ve kterém jsou definovány různá rozšíření.



Obr. 6.1 HAL knihovny v mbedu pro GPIO (převzato z [15])

Velikost HAL a LL knihoven vyžaduje také obsáhlý zdroj informací, jenž v našem případě představuje manuál [11].

6.2.1 Orientace v manuálu HAL a LL knihoven

V kap. 6.2.3 se zaměříme na blikání LED s využitím HAL knihovny *stm32fxx_hal_gpio*. Před tím je však potřeba vědět, jak se orientovat v manuálu [11], abychom tento příklad mohli realizovat.

V manuálu je GPIO popisováno v kap. 23. Začátek této kapitoly je na obr. 6.2. Tradičně se na začátku kapitoly nachází inicializační struktura, v tomto případě *GPIO_InitTypeDef*. Její proměnné by měly být po přečtení minulých kapitol zřejmé (viz kap. 5.1).

Dále bývá uveden popis a **návod**, jak danou periferii používat (viz obr. 6.3). Následuje přehled funkcí a jejich popis (např. popis funkce *HAL_GPIO_TogglePin* vidíme na obr. 6.4). Na závěr se tradičně uvádí definice maker, kupříkladu definice maker týkající se pull rezistorů jsou na obr. 6.5.

23 HAL GPIO Generic Driver

23.1 GPIO Firmware driver registers structures

23.1.1 GPIO_InitTypeDef

GPIO_InitTypeDef is defined in the stm32f3xx_hal_gpio.h

Data Fields

- ***uint32_t Pin***
- ***uint32_t Mode***
- ***uint32_t Pull***
- ***uint32_t Speed***
- ***uint32_t Alternate***

Obr. 6.2 Úvod kapitoly HAL GPIO (převzato z [11])

23.2.2 How to use this driver

1. Enable the GPIO AHB clock using the following function: __HAL_RCC_GPIOx_CLK_ENABLE().
2. Configure the GPIO pin(s) using HAL_GPIO_Init().
   - Configure the IO mode using "Mode" member from GPIO_InitTypeDef structure
   - Activate Pull-up, Pull-down resistor using "Pull" member from GPIO_InitTypeDef structure.
   - In case of Output or alternate function mode selection: the speed is configured through "Speed" member from GPIO_InitTypeDef structure.
   - In alternate mode is selection, the alternate function connected to the IO is configured through "Alternate" member from GPIO_InitTypeDef structure.
   - Analog mode is required when a pin is to be used as ADC channel or DAC output.
   - In case of external interrupt/event selection the "Mode" member from GPIO_InitTypeDef structure select the type (interrupt or event) and the corresponding trigger event (rising or falling or both).
3. In case of external interrupt/event mode selection, configure NVIC IRQ priority mapped to the EXTI line using HAL_NVIC_SetPriority() and enable it using HAL_NVIC_EnableIRQ().
4. To get the level of a pin configured in input mode use HAL_GPIO_ReadPin().
5. To set/reset the level of a pin configured in output mode use HAL_GPIO_WritePin()/HAL_GPIO_TogglePin().
6. To lock pin configuration until next reset use HAL_GPIO_LockPin().
7. During and just after reset, the alternate functions are not active and the GPIO pins are configured in input floating mode (except JTAG pins).
8. The LSE oscillator pins OSC32_IN and OSC32_OUT can be used as general purpose (PC14 and PC15U, respectively) when the LSE oscillator is off. The LSE has priority over the GPIO function.
9. The HSE oscillator pins OSC_IN/OSC_OUT can be used as general purpose PF0 and PF1, respectively, when the HSE oscillator is off. The HSE has priority over the GPIO function.

Obr. 6.3 Návod, jak danou periferii používat (převzato z [11])

HAL_GPIO_TogglePin

Function name	**void HAL_GPIO_TogglePin (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)**
Function description	Toggle the specified GPIO pin.
Parameters	• **GPIOx:** where x can be (A..F) to select the GPIO peripheral for STM32F3 family • **GPIO_Pin:** specifies the pin to be toggled.
Return values	• **None:**

Obr. 6.4 Popis funkce HAL_GPIO_TogglePin (převzato z [11])

GPIO pull

GPIO_NOPULL	No Pull-up or Pull-down activation
GPIO_PULLUP	Pull-up activation
GPIO_PULLDOWN	Pull-down activation

Obr. 6.5 Definice maker pro GPIO pull (převzato z [11])

6.2.2 Orientace v HAL a LL knihovnách v mbedu

Před samotnou realizací prvního programu je také vhodné zmínit, jak jsou jednotlivé HAL a LL knihovny implementované v mbedu. V mbedu je většina toho, co bylo probíráno v předchozí kapitole, implementováno v souborech *stm32fxx_hal_gpio.c* a *stm32fxx_hal_gpio.h* (viz obr. 6.1).

Konkrétní implementace jednotlivých funkcí v *stm32fxx_hal_gpio.c* není podstatná. Důležité je však zmínit, že na začátku těchto implementačních souborů (platí to pro ně obecně) se nachází dvě důležité sekce (v některých se nachází pouze druhá sekce).

První je „X Peripheral features“, ve které je uveden přehled některých základních poznatků periferie X. V *stm32fxx_hal_gpio.c* se tato sekce také nachází. Část vidíme na obr. 6.6. **Druhá** sekce se nazývá „How to use this driver“ a je v podstatě identická stejnojmenné kapitole z manuálu, což je pro GPIO obr. 6.3.

```
==============================================================================
                ##### GPIO Peripheral features #####
==============================================================================
[..]
  (+) Each port bit of the general-purpose I/O (GPIO) ports can be individually
      configured by software in several modes:
      (++) Input mode
      (++) Analog mode
      (++) Output mode
      (++) Alternate function mode
      (++) External interrupt/event lines
```

Obr. 6.6 Část sekce "GPIO Peripheral features" (převzato z [16])

Hlavičkový soubor je pro naše účely důležitější (v případě GPIO *stm32fxx_hal_gpio.h*). V něm jsou definovány důležité struktury, konstanty a makra. Na závěr jsou deklarovány funkce z implementačního souboru.

6.2.3 Příklad: Blikání LED s využitím HAL knihovny

Nyní již k samotné realizaci projektu. Jako knihovnu stačí přidat pouze soubor *stm32f3xx_hal.h*, nikoli soubory *stm32fxx_hal_gpio*.

Postup je uveden na obr. 6.3. Shrňme ho do několika bodů: povolení hodin, vytvoření a konfigurace třídy *GPIO_InitTypeDef*, inicializace GPIO s využitím *HAL_GPIO_Init()*. Změnu výstupní úrovně vykonává funkce *HAL_GPIO_TogglePin()*.

První krok je povolení hodin do příslušné brány, ale nikde nebyla zmíněna funkce pro jejich konfiguraci. Bylo tomu tak záměrně, jelikož hodiny konfigurovat **nemusíme**. Mbed totiž nakonfiguruje hodiny na začátku programu **sám** a to stejně jako v kap. 5.3.4 (72 *MHz*).

Výsledný program:

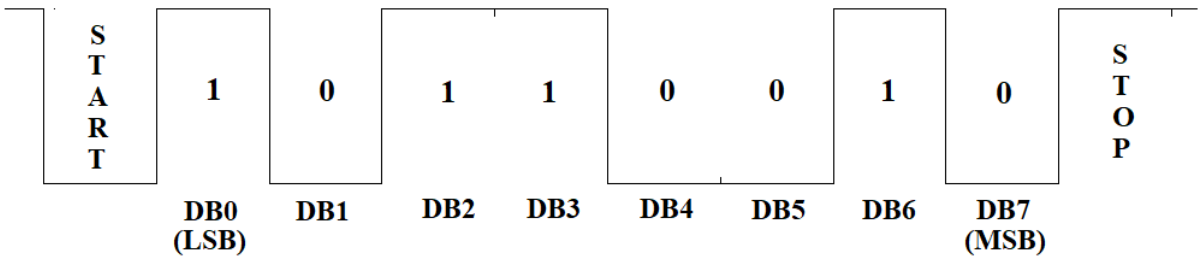
```
#include "stm32f3xx_hal.h"
int main() {
   __HAL_RCC_GPIOA_CLK_ENABLE();
   GPIO_InitTypeDef GPIOA_Init;
   GPIOA_Init.Pin = GPIO_PIN_5;
   GPIOA_Init.Mode = GPIO_MODE_OUTPUT_PP;
   GPIOA_Init.Pull = GPIO_NOPULL;
   GPIOA_Init.Speed = GPIO_SPEED_FREQ_LOW;
   HAL_GPIO_Init(GPIOA, &GPIOA_Init);
   while(1){
      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
      HAL_Delay(1000);
   }
}
```

6.2.4 Alternativní funkce

V kap. 6.2.6 se bude pracovat s jinou periferií než GPIO, konkrétně s **USART2**. Pin se může pro takovou činnost nakonfigurovat, pokud se nachází v módu alternativní funkce (viz definice registru GPIOx_MODER na obr. 5.3). Každý pin **nelze** nakonfigurovat pro práci se všemi periferiemi, ale pouze pro určitou množinu, která je specifikována ve sloupci „Alternate functions“ v tabulce 13 v [10]. Část této tabulky je na obr. 3.20.

Pro sériovou komunikaci je potřeba přijímací a vysílací pin. Tyto piny jsou označeny dovětkem RX a TX (viz kap. 3.6). U třídy *Serial* bylo možno použít žluté zvýrazněné odpovídající dvojice pinů z obr. 3.12, obr. 3.13 a obr. 3.14. Číslo za názvem „Serial“ odpovídalo jednotce USART, která byla pro tuto komunikaci použita. Ze zmíněných obrázků je zřejmé, že pro použití USART2 lze použít dvě dvojice pinů: PA2-PA3 a PB3-PB4. Použijeme např. dvojici PA2-PA3.

Jistější způsob nalezení této dvojice je z již zmíněné tabulky 13 v [10]. Část této tabulky se zvýrazněnými hledanými údaji je na obr. 6.8. Dvojice pinů je tedy nalezena. Zbývá zjistit, jaké číslo má alternativní funkce USART2_RX a USART2_TX. Tento údaj obsahuje tabulka 14 v [10], jejíž část je na obr. 6.9. Číslo alternativní funkce je tedy 7 (AF7).



Obr. 6.7 UART data

Pin number					Pin name (function after reset)	Pin type	I/O structure	Notes	Alternate functions	Additional functions
LQFP64	LQFP100	UFBGA100	WLCSP100	LQFP144						
11	18	K2	H10	29	PC3	I/O	TTa	-	EVENTOUT, TIM1_CH4, TIM1_BKIN2	ADC12_IN9
12	20	K1	H8	30	VSSA	S	-	(1)	-	-
-	-	-	-	31	VREF-	S	-	(1)	-	-
-	21	M1	J8	32	VREF+(4)	S	-	-	-	-
13	22	L1	J10	33	VDDA	S	-	-	-	-
14	23	L2	H9	34	PA0	I/O	TTa	-	TIM2_CH1/TIM2_ETR, TSC_G1_IO1, USART2_CTS, COMP1_OUT, TIM8_BKIN, TIM8_ETR, EVENTOUT	ADC1_IN1(3), COMP1_INM, RTC_TAMP2, WKUP1
15	24	M2	J9	35	PA1	I/O	TTa	-	RTC_REFIN, TIM2_CH2, TSC_G1_IO2, USART2_RTS, TIM15_CH1N, EVENTOUT	ADC1_IN2(3), COMP1_INP, OPAMP1_VINP, OPAMP3_VINP
16	25	K3	F7	36	PA2	I/O	TTa	(5)	TIM2_CH3, TSC_G1_IO3, USART2_TX, COMP2_OUT, TIM15_CH1, EVENTOUT	ADC1_IN3(3), COMP2_INM, OPAMP1_VOUT
17	26	L3	G7	37	PA3	I/O	TTa	-	TIM2_CH4, TSC_G1_IO4, USART2_RX, TIM15_CH2, EVENTOUT	ADC1_IN4(3), OPAMP1_VINM OPAMP,1_VINP

Obr. 6.8 Definice hledaných pinů PA2 a PA3 (druhá převzatá část tabulky 13 v [10])

Port	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7
	SYS_AF	TIM2/15/16/17/EVENT	I2C3/TIM1/2/3/4/8/20/15/GPCOMP1	I2C3/TIM8/20/15/GPCOMP7/TSC	I2C1/2/TIM1/8/16/17	SPI1/SPI2/I2S2/SPI3/I2S3/SPI4/UART4/5/TIM8/Infrared	SPI2/I2S2/SPI3/I2S3/TIM1/8/20/Infrared	USART1/2/3/CAN/GPCOMP3/5/6
PA0	-	TIM2_CH1/TIM2_ETR	-	TSC_G1_IO1	-	-	-	USART2_CTS
PA1	RTC_REFIN	TIM2_CH2	-	TSC_G1_IO2	-	-	-	USART2_RTS
PA2	-	TIM2_CH3	-	TSC_G1_IO3	-	-	-	USART2_TX
PA3	-	TIM2_CH4	-	TSC_G1_IO4	-	-	-	USART2_RX

Obr. 6.9 Mapování alternativních funkcí (převzatá část tabulky 14 v [10])

6.2.5 Podoba UART dat

Před realizací příkladu, ve kterém se operuje s periférií USART2, je vhodné vědět, jak vypadají data, která se pomocí této periferie posílají.

Na obr. 6.7 vidíme zakódovaný znak „M“, který má v ASCII tabulce přiřazené číslo 77, což je binárně „01001101“. První bit je tzv. **START** bit, který je povinný, za ním následují datové bity (DB). Jednotlivé datové bity jsou ve výsledné sekvenci zakódovány obráceně, tzn. **první** datový bit (MSB) je až na **konci** (před STOP bitem) a poslední bit (LSB) je první (ihned za **START** bitem).

Po datech by mohl být přidán ještě tzv. PARITNÍ bit, kterým se však zabývat nebudeme. Na konci je již zmíněný **STOP** bit, kterých může být i několik. Shrnutí: písmeno M s binární podobou „01001101“ se zakóduje do desetibitové sekvence „0101100101“, která se pošle po sériové lince zvolenou rychlostí (baudrate).

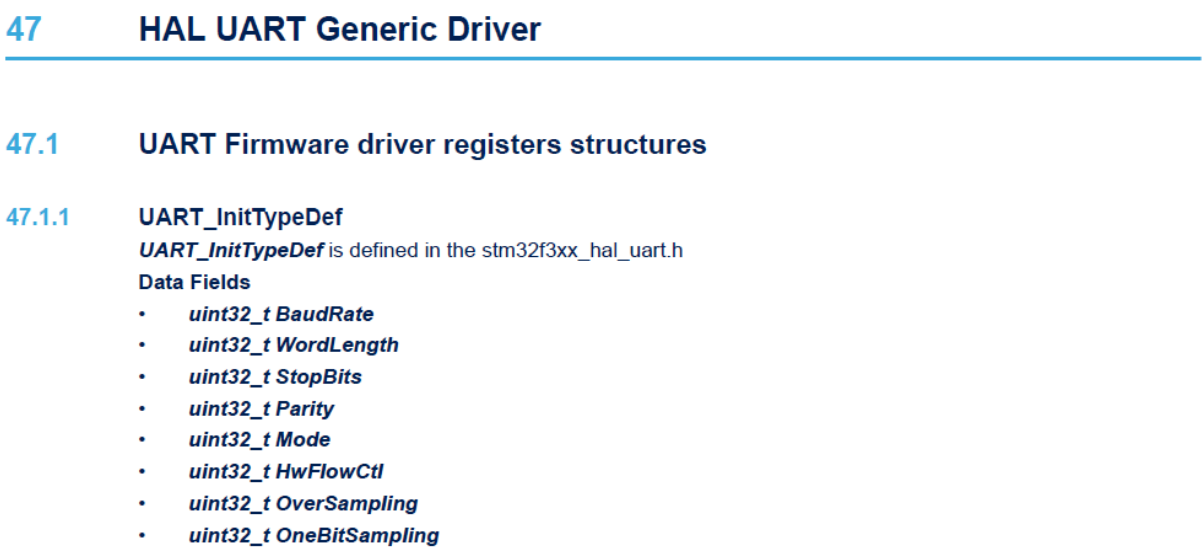
Poznamenejme, že pokud je kupříkladu baudrate 9600, pak doba trvání jednoho bitu je 1/9600 *s*, což je cca **104,2 *us***. Pokud se po sériové lince neposílají data, pak je na ni napětí log. 1, nikoli log. 0.

6.2.6 Příklad: Posílání znaků prostřednictvím USART2 s využitím HAL knihovny

Zdůrazněme, že výsledný program vznikl inspirací programů z [40]. V uvedeném zdroji je mimochodem velmi obsáhle demonstrováno použití STM32 UART, které je daleko nad rámec této práce.

V tomto příkladu si ukážeme, jak posílat znaky pomocí periferie USART2. V podstatě se jedná o stejný příklad jako v kap. 3.6.2, kdy se znak zapsaný do terminálového emulátoru pošle do Nuclea a z něj zpátky do PC. Celý proces tak vypadá jako psaní do terminálového emulátoru. Před samotnou realizací příkladu zdůrazněme, že jeho smyslem **není** detailní vysvětlení, proč to funguje, ale ukázka toho, že to tak lze naprogramovat.

O UART periferiích v manuálu [11] pojednává kap. 47, úvod této kapitoly je na obr. 6.10. Pojmům uvedeným v obrázku bychom po přečtení minulé kapitoly měli rozumět. Odpovídající knihovna v mbedu je *stm32f3xx_hal_gpio* (viz [17]).


47 HAL UART Generic Driver

47.1 UART Firmware driver registers structures

47.1.1 UART_InitTypeDef

UART_InitTypeDef is defined in the stm32f3xx_hal_uart.h

Data Fields

- *uint32_t BaudRate*
- *uint32_t WordLength*
- *uint32_t StopBits*
- *uint32_t Parity*
- *uint32_t Mode*
- *uint32_t HwFlowCtl*
- *uint32_t OverSampling*
- *uint32_t OneBitSampling*

Obr. 6.10 Úvod kapitoly HAL UART (převzato z [11])

V předešlé kapitole bylo zmíněno, že pokud se po sériové lince neposílají data, pak je na ni log. 1. Toho lze dosáhnou dvěma způsoby. První je konfigurací módu pinu „GPIO_MODE_AF_PP“, což odpovídá alternativní funkci s nastaveným push-pull módem (viz kap. 5.1.2.2). Druhý způsob je konfigurací módu pinu na „GPIO_MODE_AF_OD“, což je alternativní funkce s nastaveným výstupem open-drain, ale musí se přidat pull-up rezistor.

Na začátku programu je potřeba pustit hodiny do GPIOA a USART2:

```
__GPIOA_CLK_ENABLE();
__USART2_CLK_ENABLE();
```

Konfigurace a inicializace pinu PA2 (funguje i s „GPIO_SPEED_LOW“):

```
GPIO_InitTypeDef GPIO_PA2_InitStructure;
GPIO_PA2_InitStructure.Pin = GPIO_PIN_2;
GPIO_PA2_InitStructure.Mode = GPIO_MODE_AF_PP; //Push Pull Mode
GPIO_PA2_InitStructure.Pull = GPIO_NOPULL;
GPIO_PA2_InitStructure.Alternate = GPIO_AF7_USART2;
GPIO_PA2_InitStructure.Speed = GPIO_SPEED_HIGH;
HAL_GPIO_Init(GPIOA, &GPIO_PA2_InitStructure);
```

V případě využití „GPIO_MODE_AF_OD“ a pull-up rezistoru se změní „Mode“ a „Pull“ na:

```
GPIO_PA2_InitStructure.Mode = GPIO_MODE_AF_OD; //Open Drain mode
GPIO_PA2_InitStructure.Pull = GPIO_PULLUP;
```

Konfigurace pinu PA3 (funguje opět i s „GPIO_SPEED_LOW“):

```
GPIO_InitTypeDef GPIO_PA3_InitStructure;
GPIO_PA3_InitStructure.Pin = GPIO_PIN_3;
GPIO_PA3_InitStructure.Mode = GPIO_MODE_AF_OD;
GPIO_PA3_InitStructure.Pull = GPIO_NOPULL;
GPIO_PA3_InitStructure.Alternate = GPIO_AF7_USART2;
GPIO_PA3_InitStructure.Speed = GPIO_SPEED_HIGH;
HAL_GPIO_Init(GPIOA, &GPIO_PA3_InitStructure);
```

Následuje konfigurace periferie USART2:

```
UART_InitTypeDef myUART_Init;
myUART_Init.BaudRate = 9600;
myUART_Init.WordLength = UART_WORDLENGTH_8B;
myUART_Init.StopBits = UART_STOPBITS_1;
myUART_Init.Parity = UART_PARITY_NONE;
myUART_Init.Mode = UART_MODE_TX_RX;
myUART_Init.HwFlowCtl = UART_HWCONTROL_NONE;
```

```
UART_HandleTypeDef myUART_Handle;
myUART_Handle.Instance = USART2;
myUART_Handle.Init = myUART_Init;
```

Inicializace USART2 a nekonečná smyčka, ve které se čeká na přijetí znaku poslaného z PC a který se následně posílá zpátky do PC:

```
uint8_t buffer[1];
if (HAL_UART_Init(&myUART_Handle) == HAL_OK){
   while(1){
      HAL_UART_Receive(&myUART_Handle, buffer,
                              sizeof(buffer), HAL_MAX_DELAY);
      HAL_UART_Transmit(&myUART_Handle, buffer,
                               sizeof(buffer), HAL_MAX_DELAY);
   }
}
```

Celý program vcelku je uveden v příloze 12.16.

UART vs USART

V posledním příkladu byly použity pojmy UART a USART bez vysvětlení rozdílu mezi nimi. Této problematice se budeme dále věnovat. Zdrojem veškerých informací je [41].

UART (Universal Asynchronous Receiver Transmitter) je periferie mikrokontroléru, která transformuje přicházející/odcházející data z/do sériového bitového proudu. Stejná definice však platí i pro USART (Universal Synchronous/Asynchronous Receiver Transmitter). Přesto jsou mezi nimi značné rozdíly, my si povíme o dvou hlavních.

První zásadní rozdíl je **taktování**. Periferie UART si generuje svoje „datové hodiny" interně a synchronizuje je s datovým proudem pomocí START bitu. Neexistuje tedy žádný vstupní hod. sign. spojený s daty, proto UART potřebuje znát baudrate předem. USART může být nastaven na práci v tzv. synchronním módu, ve kterém periferie posílající data generuje také hod. sign., a tím tak přijímací část nepotřebuje znát baudrate předem. Generovaný hod. sign. může být buď externí (vlastní kabel) nebo zakódovaný v datovém proudu.

Druhý rozdíl je v **počtu protokolů**, které periferie podporuje. USART je oproti periferii UART mnohem komplikovanější. Umožňuje generovat data v mnoha formátech požadovaných v různých standardizovaných protokolech (např. IrDA, LIN, Smard Card atd.). UART oproti tomu nabízí pouze několik možností konfigurace (např. počet STOP bitů, volbu paritního bitu apod.).

USART v asynchronním módu se chová stejně jako UART. Umožňuje tedy generovat stejná data jako na obr. 6.7.

Shrnutí: USART zahrnuje stejné možnosti jako UART, avšak také mnoho dalších. USART proto lze používat jako UART bez využití dalších možností, které tato periferie poskytuje.

6.2.7 Příklad: Zjištění aktuální konfigurace hodin prostřednictvím HAL knihovny

V kap. 6.2.3 bylo uvedeno, že při použití HAL knihoven není v mbedu potřeba konfigurovat hodiny, jelikož si je mbed nakonfiguruje **sám**. V této části vytvoříme program, který z velké části vychází z minulého příkladu, avšak modifikuje ho pro posílání více bajtového datového typu. Pomocí funkce *HAL_UART_Transmit()* totiž lze posílat pouze jednobajtové buffery. Konkrétně je cílem poslat aktuální frekvence vybraných hod. sign. pomocí funkcí z další HAL knihovny, konkrétně z *stm32f3xx_hal_rcc*.

Konfigurace pinů a USART2 je stejná jako v předchozím příkladu. Ve zmíněné knihovně jsou definovány funkce *HAL_RCC_GetSysClockFreq(), HAL_RCC_GetHCLKFreq(), HAL_RCC_GetPCLK1Freq()* atd. Tyto funkce vrací číslo datového typu *uint32_t*, které je následně potřeba rozdělit na jednotlivé bajty a poslat pomocí funkce *HAL_UART_Transmit()*.

Rozdělení lze vykonat např. pomocí funkce *sprint()*. Následuje ukázka rozdělení a poslání aktuální frekvence hod. sign. systémových hodin SYSCLK. V příloze 12.17 je uveden příklad, ve kterém se pošlou frekvence SYSCLK, HCLK a PCLK1. Tímto způsobem lze rychle zjistit aktuální nastavení určitých typů hod. sign.

```
uint32_t new_var = HAL_RCC_GetSysClockFreq();
char buffer[10];
sprintf(buffer, "%d\n\r", new_var);
HAL_UART_Transmit(&myUART_Handle, (uint8_t *)buffer, sizeof(buffer),
                                HAL_MAX_DELAY);
```

7 ARM assembler v mbedu

V mbedu je možné k programování mikrokontrolérů využít kromě med tříd (kap. 3), struktur (kap. 6.1), HAL a LL knihoven (kap. 6.2), také nízkoúrovňový programovací jazyk **ARM assembler** (dále jen assembler), jemuž je věnována tato kapitola. Jelikož je assembler nízkoúrovňový programovací jazyk, nevyžaduje kompilátor. Kód v něm napsaný přímo odpovídá binárnímu kódu, který je pouze transformovaný do textové podoby. Napsaný kód v assembleru se tak lehce převede do binárního kódu.

Programování v assembleru v prostředí mbed se věnuje i kap. 9 v [1]. Jelikož je zmíněná kapitola obsáhle a přehledně zpracována, bude na ni v této kapitole velmi často odkazováno. Po odkazu na určitou pasáž se následně předpokládají znalosti získané z této pasáže. Nyní je vhodné přečíst si úvod kap. 9 z [1].

Zopakujme z uvedené kapitoly, že motivace pro naučení se programovat v assembleru je hlavně možnost **kompletně** řídit daný procesor. Samotné řízení se odehrává prostřednictvím registrů procesoru a instrukcí. Proto začneme právě těmito dvěma pojmy.

7.1 Registry procesoru

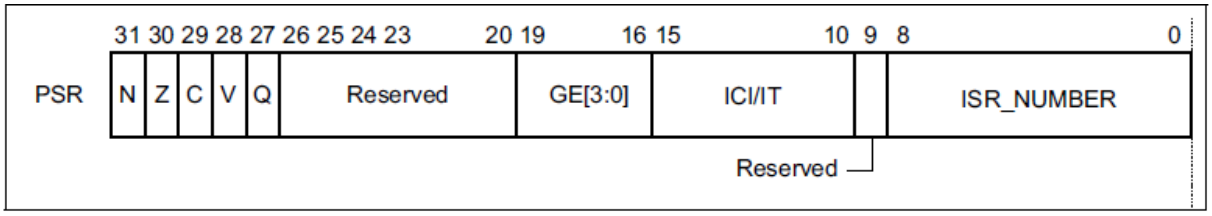
Registry procesoru představují prostor, na kterém se veškeré programování odehrává. K dispozici je 16 registrů R0 až R15 a další speciální registry (viz obr. 7.2). Pro účely této práce jsou podstatné pouze **základní** registry R0 až R15 a jeden speciální **PSR**. O registrech procesoru pojedává kap. 9.1 v [1] (přečtěte si ji). Zdůrazněme, že registr je **paměťové místo** v **procesoru**. Z toho důvodu je přístup k nim velmi rychlý.

Po přečtení uvedené kapitoly by měly být jasné odpovědi na následující otázky: na co se registry používají obecně, na co se používají základní registry, co je to zásobník, na co se používá a jak se s ním pracuje. Dále co je to link register (**LR**) a program counter (**PC**).

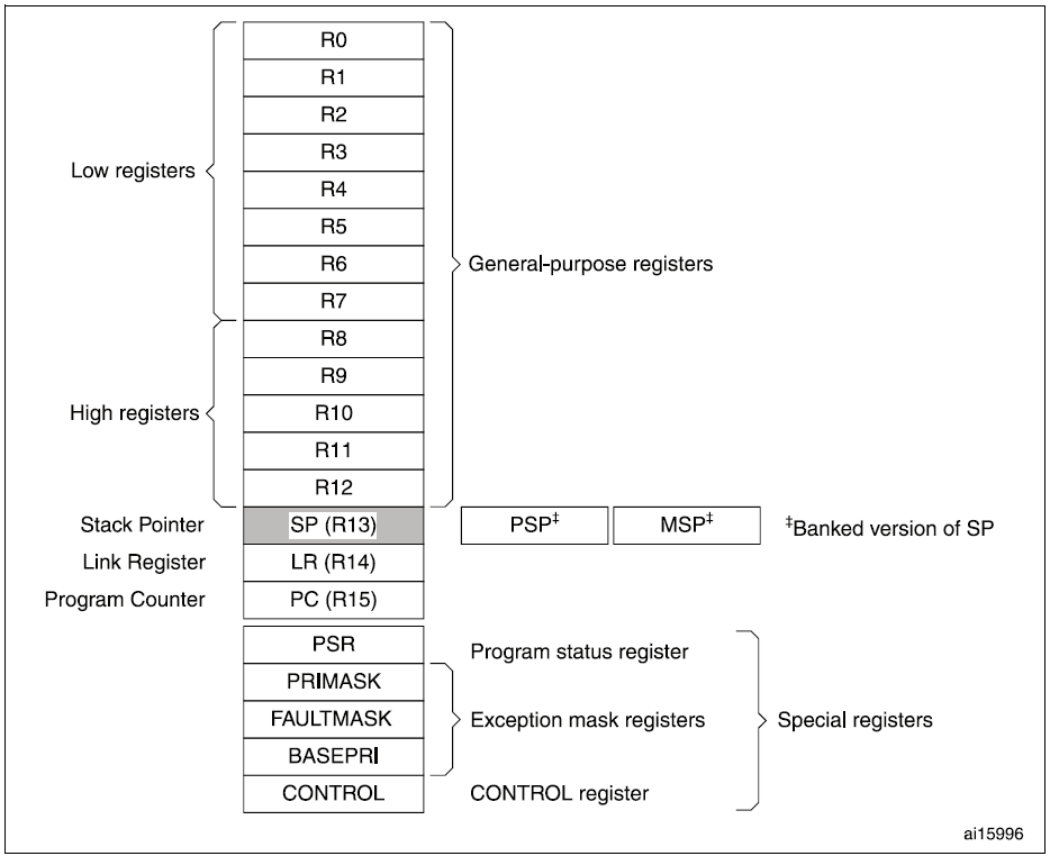
Zopakujme, že pomocí instrukce **PUSH** lze uložit obsah registru do zásobníku a to na adresu, na kterou ukazuje **SP** (stack pointer). Po tomto zápisu **klesne** hodnota SP o 4. Naopak odebrání hodnoty z adresy SP vykonává instrukce **POP**. [1]

7.1.1 Program Status Registr (PSR)

Program status register neboli **PSR** je speciální registr (viz obr. 7.2), z něhož využijeme pouze poslední čtyři bity: **N** (negative or less than flag), **Z** (zero flag), **C** (carry or borrow flag) a **V** (overflow flag). Uvedeným bitům, jak jejich název napovídá, se říká **flagy**. Význam těchto flagů je patrný z kap. 9.1.1 z [1] nebo z obr. 7.3. V kap. 7.5 si ukážeme praktické využití flagů na příkladu.



Obr. 7.1 Program status register (převzato z [42])



Obr. 7.2 Registry procesoru (převzato z [42])

Bits	Description
Bit 31	**N:** Negative or less than flag: 0: Operation result was positive, zero, greater than, or equal 1: Operation result was negative or less than.
Bit 30	**Z:** Zero flag: 0: Operation result was not zero 1: Operation result was zero.
Bit 29	**C:** Carry or borrow flag: 0: Add operation did not result in a carry bit or subtract operation resulted in a borrow bit 1: Add operation resulted in a carry bit or subtract operation did not result in a borrow bit.
Bit 28	**V:** Overflow flag: 0: Operation did not result in an overflow 1: Operation resulted in an overflow.
Bit 27	**Q:** DSP overflow and saturation flag: Sticky saturation flag. 0: Indicates that saturation has not occurred since reset or since the bit was last cleared to zero 1: Indicates when an SSAT or USAT instruction results in saturation, or indicates a DSP overflow. This bit is cleared to zero by software using an MRS instruction.
Bits 26:20	Reserved.
Bits 19:16	**GE[3:0]:** Greater than or Equal flags. See *SEL on page 105* for more information.
Bits 15:0	Reserved.

Obr. 7.3 Význam PSR (převzato z [41)

7.2 Základní instrukce procesoru

Z minulé kapitoly víme, že registr představuje místo, kde je uložené nějaké číslo. Operacím, které na těchto číslech (registrech) vykonávají různé funkce, se říká **instrukce**. Instrukcí existuje mnoho různých typů. Přehled všech instrukcí procesorů rodiny **STM32 CORTEX-M4** je uveden v tabulce 21 v [42]. Část této tabulky vidíme na obr. 7.4.

Mnemonic	Operands	Brief description	Flags	Page
ADC, ADCS	{Rd,} Rn, Op2	Add with carry	N,Z,C,V	*3.5.1 on page 83*
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V	*3.5.1 on page 83*
ADD, ADDW	{Rd,} Rn, #imm12	Add	N,Z,C,V	*3.5.1 on page 83*
ADR	Rd, label	Load PC-relative address	—	*3.4.1 on page 70*

Obr. 7.4 Část instrukční sady STM32 Cortex-M4 (převzatá část tabulky 21 z [42])

7.2.1 Instrukce přesunu

Mezi základní typ instrukcí patří **instrukce přesunu**: MOV, LDR, STR, PUSH a POP. Vysvětlení významu a použití těchto instrukcí je věnována následující kapitola.

7.2.1.1 Instrukce MOV

Na obr. 7.5 je znázorněno, jak vypadá popis instrukce **MOV** v tabulce 21 v [42]. Názvy jednotlivých sloupců jsou stejné jako na obr. 7.4. V prvním sloupci je název instrukce: **MOV**. Pokud se na konec instrukce (která to povoluje), přidá „S“, pak to značí, že má operace **aktualizovat flagy** (viz dále). Funkce instrukcí MOV a MOVS jsou tedy stejné, akorát MOVS navíc aktualizuje flagy.

V druhém sloupci se nachází formát toho, co lze za instrukci napsat (viz dále). Třetí sloupec stručně popisuje danou instrukci. Sloupec s názvem „Flags“ uvádí přehled flagů, které se aktualizují při použití instrukce MOVS. Poslední sloupec odkazuje na kapitolu v [42], jenž se danou instrukcí zabývá podrobně.

MOV, MOVS	Rd, Op2	Move	N,Z,C	*3.5.6 on page 89*

Obr. 7.5 Instrukce MOV (převzatá část tabulky 21 z [42])

Dále si za účelem zlepšení orientace v [42] ukážeme, jak je kap. 3.5.6 v [42], obsahující podrobný popis instrukce MOV, popsána. Úvod této kapitoly je na obr. 7.6. Jak název „MOV and MVN“ napovídá, kapitola se nezabývá pouze instrukcí MOV, ale také (díky podobnosti) instrukcí MVN.

Pod názvem je uvedený „Syntax“ a pod ním vysvětlivky. První dva řádky syntaxe si dále názorně rozebereme. V prvním řádku „{S}“ značí, že se dá za instrukci MOV přidat „S“ a tím tak aktualizovat flagy (viz výše). Řetězec „{cond}“ značí, že lze za „MOV“ (případně za „MOVS“) přidat podmínku (viz kap. 7.5).

Za označením instrukce následuje: cílový registr „Rd“ (destination register), čárka a „Operand2“. Podle vysvětlivky pod syntaxí je „Operand2“ tzv**. flexibilní druhý operand**, což je buď konstanta ve speciálním tvaru nebo registr (např. R1). Konstanty obecně se

zapisují za znak „#“. Speciální konstantou je myšleno číslo, které může vzniknout posunutím 8 bitového čísla (0-255) zbývajícím libovolným počtem bitů v rámci 32-bitového slova. Jedná se tedy např. o čísla: 0x000f f000, 0x8000 0000, 0x0000 0890 atd.

Druhý řádek syntaxe je velmi podobný, akorát místo flexibilního druhého operandu lze použít konstantu v rozmezi hodnot 0 až 65 535.

3.5.6 MOV and MVN

Move and Move NOT.

Syntax

```
MOV{S}{cond} Rd, Operand2
MOV{cond} Rd, #imm16
MVN{S}{cond} Rd, Operand2
```

Where:

- '*S*' is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation (see *Conditional execution on page 65*).
- '*cond*' is an optional condition code (see *Conditional execution on page 65*).
- '*Rd*' is the destination register.
- '*Operand2*' is a flexible second operand (see *Flexible second operand on page 60*) for details of the options.
- '*imm16*' is any value in the range 0—65535.

Obr. 7.6 Úvod kapitoly zabývající se instrukcemi MOV a MVN (převzato z [42])

Doposud nebylo zmíněno, co vlastně instrukce MOV dělá. Odpověď se skrývá pod nadpisem „Operation“, který následuje za nadpisem „Syntax“ (kvůli rozsáhlosti není uveden). Pomocí této instrukce lze zkopírovat hodnotu jednoho registru do druhého. Dále uložit speciální konstantu z flexibilního operandu (viz výše) nebo konstantu 0 až 65 535 do požadovaného registru Rd. Následují příklady na zmíněné funkce.

Zkopírování hodnoty R1 do R0:

```
MOV R0, R1
```

Uložení konstanty v rozsahu 0 až 65 535 do R5:

```
MOV R5, #18791
```

Uložení speciální konstanty v hexadecimálním tvaru (0x79000 je číslo 121 posunuté o 12 míst vlevo) do R7:

```
MOV R7, #79000
```

Na závěr jsou v každé kapitole zabývající se danou instrukcí uvedené příklady, jak probíranou instrukci použít. U instrukce MOV tyto příklady vidíme na obr. 7.7.

Example

```
MOVS R11, #0x000B     ; write value of 0x000B to R11, flags get updated
MOV R1, #0xFA05       ; write value of 0xFA05 to R1, flags not updated
MOVS R10, R12         ; write value in R12 to R10, flags get updated
MOV R3, #23           ; write value of 23 to R3
MOV R8, SP            ; write value of stack pointer to R8
MVNS R2, #0xF         ; write value of 0xFFFFFFF0 (bitwise inverse of 0xF)
                      ; to the R2 and update flags
```

Obr. 7.7 Příklady použití instrukce MOV (převzato z [42])

7.2.1.2 Instrukce LDR

Instrukce **LDR** obecně slouží k čtení dat z paměti do registru. Lze ale také využít pro uložení konstant do registru. Pomocí této instrukce lze uložit i jiné konstanty než ty, které lze uložit pomocí instrukce MOV (viz kap. 7.2.1.1). Konstanta se v tomto případě nezapisuje za znak „#“, ale za „=“. Uložení konstanty 0x1234 5678 do registru R3:

```
LDR R3, =0x12345678 ;a comment starts with a semicolon (or // in mbed)
```

Pomocí funkce LDR lze také načíst hodnotu z adresy uložené v nějakém registru. Uložení adresy 0x4800 0000 (základní adresa brány GPIOA) do R0 a následné načtení hodnoty z této adresy do R1:

```
LDR R0, =0x48000000
LDR R1, [R0]
```

Adresy je také možné ukládat jako konstanty na začátku programu:

```
GPIOA EQU 0x48000000
```

Instrukce LDR navíc umožňuje přidat **offset**, což je vhodné při práci s registry. Načtení hodnoty registru RCC_CFGR (address offset 0x04) do R1:

```
GPIOA EQU 0x48000000
RCC_CFGR_offset EQU 0x04
  …
LDR R0, =GPIOA
LDR R1, [R0, #RCC_CFGR_offset] ;load value from (GPIO+RCC_CFGR_offset) to R1
```

Offset je možné zapsat ještě jinými způsoby (viz [42]), např. lze také použít zápis:

```
LDR R1, [R0], #RCC_CFGR_offset
```

Poznamenejme, že LDR, následující instrukce STR (a některé další) umožňují specifikovat počet bajtů, se kterými se má pracovat. Možnosti jsou **B** (byte - jeden bajt), **H** (halfword - dva bajty) a **B** (word - čtyři bajty). Pomocí této instrukce tak jde např. načíst pouze jeden bajt z určité adresy.

7.2.1.3 Instrukce STR

Instrukce **STR** umožňuje zapsat hodnotu registru na adresu zapsanou v jiném registru. Jedná se tedy o velmi užitečnou instrukci, jelikož jejím prostřednictvím lze měnit hodnoty registrů, což lze využít např. při konfiguraci hodin, GPIO apod. **Zdůrazněme**, že tentokrát je na prvním místě registr, jehož hodnota se ukládá na adresu uloženou v druhém registru. Jedná se tak o opačný směr než u instrukce LDR (viz kap. 7.2.1.2).

Uložení hodnoty (VALUE) z R0 na adresu (ADDRESS) uloženou v R1:

```
LDR R0, =VALUE
LDR R1, =ADDRESS
STR R0, [R1] ;store VALUE to the ADDRESS
```

Instrukce STR také umožňuje přidat **offset**. Možnosti zápisu offsetu jsou stejné jako v předešlé kapitole.

7.2.1.4 Instrukce PUSH a POP

Poslední dvě základní instrukce přesunu jsou **PUSH** a **POP**. Jejich význam byl již vysvětlen v kap. 9.1 v [1] a následně shrnut v kap. 7.1. Zopakujme, že se tyto instrukce obecně používají pro uložení obsahu registrů.

Základní použití instrukce **PUSH** je pro uložení LR do zásobníku na začátku programu/funkce. Pro ty nejjednodušší případy ji budeme používat k uložení adresy pro vrácení se zpátky do *main.cpp* z assembler programu. Uložení LR do zásobníku:

```
PUSH {LR}
```

Na konci programu se pomocí instrukce **POP** nahraje uložená adresa na zásobníku do PC (program counter) a tím se skočí zpátky do *main.cpp*:

```
POP {PC}
```

V mbedu je navíc na začátku programu **potřeba** uložit hodnoty registrů R4-R8, R10, R11, pokud s nimi bude operováno. Na konci programu se pak tyto hodnoty obnoví zpět do příslušných registrů. Navíc je také nutné odebrat všechny vložené hodnoty do zásobníku a tím tam zachovat SP. [26]

Kupříkladu budou-li v assembler programu měněny hodnoty registrů R5 a R6, pak je musíme na začátku (spolu s LR) uložit pomocí instrukce PUSH a na konci opět obnovit (spolu s LR, který uložíme do PC) pomocí instrukce POP:

```
  …
PUSH {R5, R6, LR}
  …
POP { R5, R6, PC}
  …
```

7.2.2 Aritmetické instrukce

Prostřednictvím aritmetických instrukcí procesor vykonává aritmetické operace. Základ této skupiny instrukcí tvoří operace sčítání a odčítání, které realizují instrukce **ADD** a **SUB**. Syntax je pro obě instrukce stejná (viz obr. 7.8). Smyslem vkládání syntaxí z manuálu je osvojení si stylu psaní těchto manuálů. Případné dohledávání a dostudovávání dalších instrukcí se tak stane rychlejší. Přičtení jednoho registru do druhého:

```
ADD R2, R1, R0 ; R2=R1+R0
```

Poznamenejme, že následující zápisy jsou ekvivalentní:

```
ADD R1, R1, R0 ; R1=R1+R0
ADD R1, R0 ; R1=R1+R0
```

3.5.1 **ADD, ADC, SUB, SBC, and RSB**

Add, Add with Carry, Subtract, Subtract with Carry, and Reverse Subtract.

Syntax

```
op{S}{cond} {Rd,} Rn, Operand2
op{cond} {Rd,} Rn, #imm12; ADD and SUB only
```

Where:

- '*op*' is one of the following:
  ADD: Add
  ADC: Add with carry
  SUB: Subtract
  SBC: Subtract with carry
  RSB: Reverse subtract
- '*S*' is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation (see *Conditional execution on page 65*)
- '*cond*' is an optional condition code (see *Conditional execution on page 65*)
- '*Rd*' is the destination register. If Rd is omitted, the destination register is Rn
- '*Rn*' is the register holding the first operand
- '*Operand2*' is a flexible second operand (see *Flexible second operand on page 60* for details of the options)
- '*imm12*' is any value in the range 0—4095

Obr. 7.8 Úvod kapitoly zabývající se instrukcemi ADD a SUB (převzato z [42])

7.3 Struktura programu psaného v assembleru v prostředí mbed

S informacemi získanými v předešlých dvou kapitolách máme dostatečné teoretické znalosti k vytvoření prvního programu psaného v assembleru. V mbedu se assembler programy volají jako funkce, proto je lze také jako funkce označit. Zmíněné funkce mají danou strukturu, která se **musí** dodržet. Veškeré informace ohledně této struktury jsou uvedené v **kap. 9.3** v [1].

Získané informace z uvedené kapitoly shrneme do dvou skutečností. V mbedu se vytvořená funkce psaná v assembleru (v souboru *name.s*) deklaruje tímto způsobem:

```
extern “C“ void name();
```

Funkce *name.s* má přesně danou podobu, pro kterou si je vhodné vytvořit šablonu:

```
;-------------------------------------;
; description
;-------------------------------------;
; constants
;-------------------------------------;
; program
   AREA asm_func, CODE, READONLY
   EXPORT name
name
   PUSH{LR}
   ; WRITE YOUR CODE
   POP {PC}
   ALIGN
   END
```

Při následném použití této šablony je žádoucí příhodně pojmenovat vytvořený soubor (např. *flash.s*), stejným způsobem pojmenovat i dvě místa v tomto souboru (místo „name“ napsat nový název souboru) a název funkce v deklaraci v *main.cpp*.

7.4 První vytvořená funkce napsaná v assembleru

V této kapitole si vytvoříme první funkci napsanou v assembleru (dále označováno jako assembler funkce). Z předešlé kapitoly víme, jak funkci vytvořit. Zbývá doplnit kód do sekce „; WRITE YOUR CODE“.

Aby bylo zřejmé, že funkce „něco dělá“, je vhodné, aby vytvořená funkce vracela nějaké číslo. Toho lze dosáhnout např. upravením deklarace v *main.cpp* na:

```
extern “C“ uint32_t name();
```

V takovém případě bude hodnota uložená v R0 a při návratu zpět z assembler funkce vrácena do *main.cpp* jako návratová hodnota datového typu *uint32_t*.

Jelikož se jedná o první příklad na práci s assembler funkcemi a znalost instrukcí je minimální, spokojíme se s programem, který uloží do jednoho registru číslo jedna, do druhého číslo dva a vrátí jejich součet. Pro procvičení uložte jedno číslo pomocí MOV a druhé pomocí LDR. Na závěr pošlete návratovou hodnotu assembler funkce sériovým portem s využitím třídy *Serial*. Assembler funkci pojmenujte např. *first_program.s*.

Soubor *main.cpp*:

```
#include "mbed.h"
Serial serial(PA_2, PA_3);
extern "C" uint32_t first_program();
int main() {
   uint32_t return_value = first_program();
   serial.printf("return_value: %d\n\r", return_value);
}
```

Soubor *first_program.s*:

```
    AREA asm_func, CODE, READONLY
    EXPORT first_program
first_program
    PUSH{LR} ; save LR to the Stack
    MOV R1, #1 ; R1 = 1
    LDR R2, =2 ; R2 = 2
    ADD R0, R1, R2 ; R0 = R1 + R2
    POP {PC} ; load LR from the Stack to PC = jump back to main.cpp
    ALIGN
    END
```

7.5 Podmíněné vykonávání instrukcí

V kap. 7.6 budou vysvětleny nové typy instrukcí procesoru, jejichž použití bývá často spojováno s podmíněným vykonáváním. Z tohoto důvodu je vhodné se nejdříve seznámit s podstatou podmíněného vykonávání instrukce. Poznamenejme, že u starších procesorů (před řadou Cortex M3) lze pro podmíněné vykonávání instrukcí použít pouze skokové instrukce.

Podmíněné vykonávání instrukce je spojeno s instrukcí **IT** (if-then), kterou lze použít **po** instrukci aktualizující flagy (viz kap. 7.2.1.1). Připomeňme, že aktualizace flagů se tradičně vynutí přidáním „S“ za danou instrukci (např. z instrukce SUB vznikne SUBS). Tento úkon nemá smysl provádět u každé instrukce. Poznamenejme, že instrukci IT lze používat u jader rodiny Cortex M3 a vyšší.

Probíráné téma je obsáhle zpracováno v kap. 9.1.2 v [1] (přečtěte si). Zdůrazněme, že tvar instrukce IT lze různě měnit (**podle daných pravidel**) v závislosti na požadované funkci.

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned ≥
CC or LO	C = 0	Lower, unsigned <
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned >
LS	C = 0 or Z = 1	Lower or same, unsigned ≤
GE	N = V	Greater than or equal, signed ≥
LT	N != V	Less than, signed <
GT	Z = 0 and N = V	Greater than, signed >
LE	Z = 1 and N != V	Less than or equal, signed ≤
AL	Can have any value	Always. This is the default when no suffix is specified.

Obr. 7.9 Podmínkové přípony (převzato z [42])

7.5.1 Ukázka použití podmíněného vykonávání instrukcí na instrukci SUBS

Příklad začíná takto:

```
NUM EQU 1
AREA asm_func, CODE, READONLY
   EXPORT name
name
   PUSH{LR}
   MOV R0, #1
   SUBS R0, #NUM
      …
   POP {PC}
   ALIGN
   END
```

Cílem je dopsat kód do oblasti „…“ tak, aby se uložila pětka do R0, pokud je výsledek odečítání nulový, jinak desítka. K tomu lze využít podmíněné vykonávání instrukce MOV.

Zajímá nás podmínka na rovnost/nerovnost nule, proto se za MOV musí doplnit „EQ“/„NE“ (viz obr. 7.9). Následující instrukce představuje operaci, která uloží do R0 číslo pět, pokud je flag Z roven jedné. V takovém případě je výsledek operace „SUBS R0, #NUM“ nulový:

```
MOVEQ R0, #5
```

Uložení čísla deset do R0 v případě nenulovosti zmíněné operace (flag Z roven nule):

```
MOVNE R0, #10
```

Před obě instrukce se dopíše „ITE EQ“, protože je podmínka zaměřena na flagy „EQ“/„NE“. Tvar instrukce je „if-then-else“ (viz kap. 9.1.2 v [1]). Poznamenjme, že zápis „ITE EQ“ lze vynechat a že pomocí instrukce IT lze vždy testovat pouze **jeden** flag. Instrukce, u kterých není podmínka splněna, nejsou vynechány, ale jsou nahrazeny instrukcí **NOP**, což je instrukce nevykonávající nic. Čas se tedy na těchto instrukcích „neušetří“.

Správnost příkladu lze ověřit návratovou hodnotou assembler funkce stejně jako v předchozí kapitole.

7.6 Další instrukce procesoru

V této kapitole se seznámíme s novými třemi typy instrukcí procesoru, které budou pro další programování v assembleru potřeba.

7.6.1 Instrukce logické

V kap. 5 byla vysvětlena konfigurace GPIO bran a hodin prostřednictvím GPIO a RCC registrů. Čtení hodnoty z nějaké adresy vykonává LDR a zápis hodnoty na danou adresu STR (viz kap. 7.2.1).

Dále se však ke konfiguraci registrů využívaly také bitové (logické) operace, konkrétně bit. součet, bit. součin, bit. exkluzivní součet, bit. negace a bit. posun dolevo/doprava. Pomocí

těchto operací se nastavoval konkrétní bit do 1/0, případně měnila jeho hodnota. Stejné operace lze realizovat i v assembler funkcích s využitím log. operací ORR, BIC, a XOR, jejichž syntaxe je na obr. 7.10.

3.5.2 AND, ORR, EOR, BIC, and ORN

Logical AND, OR, Exclusive OR, Bit Clear, and OR NOT.

Syntax

```
op{S}{cond} {Rd,} Rn, Operand2
```

Where:

- '*op*' is one of:
  AND: Logical AND.
  ORR: Logical OR or bit set.
  EOR: Logical exclusive OR.
  BIC: Logical AND NOT or bit clear.
  ORN: Logical OR NOT.
- '*S*' is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation, see *Conditional execution on page 65*.
- '*cond*' is an optional condition code, see *Conditional execution on page 65*.
- '*Rd*' is the destination register.
- '*Rn*' is the register holding the first operand.
- '*Operand2*' is a flexible second operand, see *Flexible second operand on page 60* for details of the options.

Obr. 7.10 Úvod kapitoly zabývající se instrukcemi ORR, BIC a XOR (převzato z [42])

7.6.1.1 Instrukce ORR

Pomocí **ORR (**log. součet) se nastavuje konkrétní bit do 1. S využitím instrukce ORR tedy lze např. aktivovat pátý bit v registru GPIOA_ODR a tím tak rozsvítit diodu. Vytvořte program obsahující assembler funkci *switch_LED_on.s*, která rozsvítí diodu.

Poznamenejme, že stejný program je vytvořen i v kap. 9.4.2 v [1]. V případě problémů se obraťte na zmíněnou kapitolu, jelikož obsahuje i detailní popis jednotlivých řádků kódu.

Soubor *main.cpp*:

```
#include "mbed.h"
DigitalOut led(PA_5);
extern "C" void switch_LED_on();
int main() {
   switch_LED_on();
}
```

Soubor *switch_LED_on.s*:

```
GPIOA EQU 0x48000000
ODR_offset EQU 0x14
   AREA asm_func, CODE, READONLY
   EXPORT switch_LED_on
```

```
switch_LED_on
   PUSH{LR}
   LDR R0, =GPIOA ; R0 = GPIOA
   LDR R1, [R0, #ODR_offset] ; load value of R0 + ODR_offset to R1
   ORR R2, R1, #(0x1 :SHL: 5) ; R2 = R1 | (0x1 << 5)
   STR R2, [R0, #ODR_offset] ; save the R2 registry value to R0 + ODR_offset
   POP {PC}
   ALIGN
   END
```

7.6.1.2 Instrukce BIC

Instrukce **BIC** (bit clear) vykonává funkci, jež zahrnuje dvě bitové operace (logický součin a negaci). Pokud bychom k vyjádření významu operace „BIC R2, R1, R0“ použili bitové operace „&“ a „~“ z kap. 5.1.4, pak by význam odpovídal zápisu „R2 = R1 & (~R2)“.

Instrukce BIC realizuje opačnou operaci než předešlá instrukce ORR (viz kap. 7.6.1.1). Pomocí ORR se daný bit aktivoval, tedy pomocí BIC se konkrétní bit nastaví do 0. Toho lze využít při zpětném nastavení pátého bitu do 0 v GPIOA_ODR.

Rozšiřte program vytvořený v minulé kapitole o assembler funkci *switch_LED_off*.s, která LED opět zhasne. Pomocí funkcí *switch_LED_on*.s a *switch_LED_off*.s periodicky blikejte. V případě problému se obraťte na kap. 9.4 v [1], ve které je implementovaný stejný příklad.

7.6.1.3 Instrukce EOR

Pomocí instrukce **EOR** (exclusive or) lze změnit hodnotu daného bitu. S jejím využitím lze spojit obě vytvořené assembler funkce (*switch_LED_on*.s a *switch_LED_off*.s) do jedné (např. s názvem *change_state.s*), která mění hodnotu pátého bitu v GPIOA_ODR. To je zadání dalšího příkladu. Výsledné řešení je v příloze 12.18.

7.6.2 Instrukce skoku

Předposlední typ instrukcí jsou instrukce **přesunu**. Základním představitelem, ze kterého jsou další instrukce odvozeny, je instrukce **B** (branch). Pomocí instrukce B se přesouvá (skáče) na návěští.

Návěští je část programu obvykle psaná velkými písmeny, před kterými **nejsou** mezery. V podstatě se nejedná o nic jiného, než o adresu (hodnotu PC) daného místa v programu. Příkladem je „switch_LED_on“ v kap. 7.6.1.1 před instrukcí PUSH. Zmíněný řetězec sice není psaný velkými písmeny, ale jedná se také o návěští.

Jako další příklad návěští spolu s instrukcí B uvádíme nekonečnou smyčku:

```
NAVESTI
   B NAVESTI
```

Časté je využití instrukce B s podmíněným vykonáváním. Kupříkladu smyčka odečítající od registru R0 jedničku, dokud se tento registr nevynuluje, může vypadat takto:

```
NUM EQU 12345689
  …
  LDR R0, =NUM
LOOP
  SUBS R0, #1
  BNE LOOP ; branch to the label LOOP if R0 != 0
  ; R0 ==0, continue
  …
```

7.6.2.1 Vnořená funkce

V programu lze volat funkci respektive proceduru uvnitř assembler funkce. K tomu jsou potřeba instrukce přesunu. Základní struktura vnořených funkcí je v úvodu kap. 9.6 v [1].

Pro procvičení vnořených funkcí můžeme pozměnit příklad v kap. 7.6.1.1 tak, že LED rozsvítíme až uvnitř vnořené funkce. Soubor *main.cpp* je stejný.

Pozměněný soubor *switch_LED_on.s*:

```
GPIOA EQU 0x48000000
ODR_offset EQU 0x14
   AREA asm_func, CODE, READONLY
   EXPORT switch_LED_on
switch_LED_on
   PUSH{LR}
   BL SWITCH_LED_ON ; branch to the label SWITCH_LED_ON
   POP {PC}

SWITCH_LED_ON PROC
   PUSH{LR}
   LDR R0, =GPIOA
   LDR R1, [R0, #ODR_offset]
   ORR R2, R1, #(0x1 :SHL: 5)
   STR R2, [R0, #ODR_offset]
   POP{PC}
   ENDP ; end of procedure

   ; end of assembly function
   ALIGN
   END
```

Instrukce **BL** (branch with link) skočí na dané návěští a zároveň uloží návratovou adresu do LR. Tato návratová hodnota je následně v proceduře uložena do zásobníka. Na konci procedury se načtením této hodnoty ze zásobníku (řádek „POP{PC}“) skočí zpátky a to konkrétně na stejně vypadající řádek „POP{PC}“. Pro zopakování: v tomto řádku se opět načte hodnota ze zásobníku, ale skočí se do souboru *main.cpp*.

Poznamenejme, že v uvedeném příkladu by také šlo použít instrukci B (místo BL):

```
B SWITCH_LED_ON
```

Z procedury by se pak vracelo prostřednictvím instrukce **BX** (branch indirect), která umožňuje skočit na adresu umístěnou v LR:

```
SWITCH_LED_ON PROC
   LDR R0, =GPIOA
   LDR R1, [R0, #ODR_offset]
   ORR R2, R1, #(0x1 :SHL: 5)
   STR R2, [R0, #ODR_offset]
   BX LR ; branch to the address stored in LR
   ENDP ; end of procedure
     …
```

Nejjednodušším případem však zůstává první verze. Pokud bychom v průběhy druhé verze přidali skok do další procedory, pak by tato varianta nešla použít. **Shrnutí**: do procedur skákejte prostřednictvím BL a LR si na začátka procedury ukládejte do zásobníku. Pokud je navíc žádoucí udržovat obsah registrů v původní funkci, pak je na začátku procedur také ukládejte (a na konci obnovujte).

7.6.3 Instrukce porovnání

Poslední typ instrukcí, který zmíníme, jsou instrukce **porovnání**. Z této skupiny uvedeme instrukce **CMP** (compare positive) a **TST** (test bits), jejichž syntaxe jsou stejné (viz [42]). Syntaxe CMP je na obr. 7.11. Obě instrukce porovnávají hodnotu v „Rn“ s flexibilním druhým operandem (viz 7.2.1.1) a obě aktualizují flagy.

Rozdíl mezi instrukcemi CMP a TST je jejich implementace a z toho vyplývající použití. Instrukce CMP dělá to samé jako instrukce SUBS, akorát nezmění hodnotu registrů. To znamená, že odečte „Rn“ a „Operand2“ (viz obr. 7.11), aktualizuje flagy, ale jejich hodnoty zachová. Na druhou stranu instrukce TST je naimplementována jako ANDS (viz obr. 7.10), ale opět se zachováním hodnot „Rn“ a „Operand2“.

Pro porovnání dvou registrů (případně registru a čísla) se používá CMP. Instrukce TST se používá pro zjištění nastavení konkrétního bitu (bitů).

Porovnání registrů R0 a R1 pomocí CMP:

```
CMP R0, R1
BEQ R0_EQUAL_R1 ; R0 == R1 -> branch to the label R0_EQUAL_R1
BNE R0_NOT_EQUAL_R1 ; R0 != R1 -> branch to the label R0_NOT_EQUAL_R1
```

Testování nastavení pátého bitu v R0:

```
TST R0, #(1 :SHL: 5)
BEQ BIT5_SET ; bit 5 is 1 -> branch to the label BIT5_SET
BNE BIT5 _NOT_SET ; bit 5 is 0  -> branch to the label BIT5 _NOT_SET
```

3.5.5 CMP and CMN

Compare and Compare Negative.

Syntax

```
CMP{cond} Rn, Operand2
CMN{cond} Rn, Operand2
```

Where:

- '*cond*' is an optional condition code (see *Conditional execution on page 65*).
- '*Rn*' is the register holding the first operand.
- '*Operand2*' is a flexible second operand (see *Flexible second operand on page 60*) for details of the options.

Obr. 7.11 Úvod kapitoly zabývající se instrukcemi CMP a CMN (převzato z [42])

7.7 Příklad: Použití funkce se vstupním argumentem

Příklad na použití funkce se vstupním argumentem je uveden v kap. 9.5 v [1]. Jedná se o příklad, ve kterém se bliká diodou, avšak nově do assembler funkce vstupuje číslo specifikující pin. V programu se nachází nová instrukci **LSL** (logical shift left).

Poznamenejme, že následující část kódu lze vyměnit za poněkud čtivější.

```
ANDS R4,R3,R1
ITEE EQ
ORREQ R3, R1
EORNE R1, #0xFFFFFFFF
ANDNE R3, R1
```

Čitelnější kód:

```
TST R3, R1
ITE EQ
ORREQ R3, R1 ; GPIOA_ODR5 = 0 -> active ODR5
BICNE R3, R1 ; GPIOA_ODR5 = 1 -> reset ODR5
```

Zopakujme, že assembler funkce mohou mít až **čtyři** (max. 32 bitové) vstupní argumenty, které se popořadě uloží do registrů R0 až R4.

7.8 Příklad: Blikání LED pouze prostřednictvím assembleru

Na závěr této kapitoly si vytvoříme assembler funkce obstarávající celý proces blikání LED. To zahrnuje konfiguraci hodin a GPIO bran, implementaci funkce čekání apod. Příklad vychází z kap. 9.6 v [1], ze které bude převzata (a mírně upravena) procedure **WAIT_MS**.

Hodiny a GPIO brány nakonfigurujeme **stejně** jako v kap. 5.4.3 (výsledek je v příloze 12.14). Pro generování hodin tedy využijeme **PLL** se vstupním sign. z HSE bypassu. PLL nakonfigurujeme pro generování hod. sign. o frekvenci 72 *MHz*.

Jelikož se povětšinou jedná pouze o načtení hodnoty dané adresy, aktivaci několika bitů a uložení výsledku zpátky na danou adresu, což bylo již mnohokrát procvičeno, ukážeme si

jen malou část výsledného programu. Jedná se o část, ve které se aktivují bity HSEBYP a HSEON (viz kap. 5.3.2.1) a následně se ve smyčce čeká na potvrzující bit HSERDY:

```
RCC EQU 0x40021000
CR_offset EQU 0x0
        …
   ; set HSEBYP and HSEON
   LDR R0, =RCC
   LDR R1, [R0, #CR_offset]
   ORR R1, #(0x5 :SHL: 16)
   STR R1, [R0, #CR_offset]

   LDR R2, =(1 :SHL: 17) ; Mask of the HSERDY bit in RCC_CR
HSE_READY_LOOP ; is the HSE ready?
   LDR R1, [R0, #CR_offset]
   TST R1, R2
   IT EQ
   ; HSE not ready -> jump back to the label HSE_READY_LOOP
   BEQ HSE_READY_LOOP
        …
```

Výsledný program je v příloze 12.19.

8 Blok pro záznam dat v logickém analyzátoru

V poslední kapitole vytvoříme vstupní blok logického analyzátoru zaznamenávající data. Bude se jednat pouze o jednovstupový LA, avšak implementovaný různými způsoby. Navíc si vyzkoušíme různé dvě varianty toho bloku: bez triggeru a s tzv. post-triggerem.

Nejdříve je potřeba říci, co to vlastě LA je a k čemu slouží. LA je zařízení umožňující zobrazit digitální sign. Obecně LA zobrazují několik desítek/stovek digitálních sign. zároveň, to však není náš cíl. Cílem je vytvořit takový LA, aby byl schopen zaznamenat digitální signál s co možná největší frekvencí. Lze předpokládat, že nejrychlejší načítání bude na úrovni assembleru.

Zopakujme, že nevytváříme samostatný LA, ale pouze vstupní blok zaznamenávající data. Dále se zkráceně uvádí pouze tvorba logického analyzátoru. Pro zobrazení uložených hodnot lze využít software Serial Plot Plotter (viz dále).

8.1 Příklad: LA s využitím mbed tříd

První způsob implementace LA je s využitím mbed třídy *DigitalOut*.

Kolik vlastně lze uložit hodnot digitálního sign. do Nuclea? Pokud každou načtenou hodnotu (1 nebo 0) uložíme do jednoho bajtu, pak lze teoreticky zaznamenat max. 65 536 hodnot, což je přesně velikost paměťového prostoru SRAM mikrokontroléru STM32F303RE (viz kap. 4.6). Prakticky se však bude jednat o méně hodnot, jelikož část SRAM prostoru je využita pro proměnné programu. Z tohoto důvodu budeme ukládat pouze 49 152 hodnot (to je paměťový prostor o velikosti 48 kB). Poznamenejme, že 64 *kB* je 65 536 bajtů.

Hodnoty digitálního sign. by také šly ukládat vedle sebe, tzn. neobětovat jeden bajt jedné hodnotě. V takovém případě by se však musely využít bitové operace, které by zpomalovaly načítání. Tudíž k uložení jedné hodnoty je skutečně potřeba obětovat jeden bajt, i když by do něj šlo uložit 256 hodnot.

Zadání příkladu je tedy následující. Vytvořte program, ve kterém se po přijetí nějakého znaku z terminálu (např. „s“) začne ukládat jedna hodnota za druhou pinu *INPUT_PIN*, což může být např. pin PC0. Hodnoty ukládejte do pole datového typu „uint8_t“ o velikosti 16 384. Zároveň změřte čas, za který se pole zaplnilo, s využitím třídy *Timer* (viz kap. 3.8.3). Na závěr zjistěte, kolik se do pole uložilo 1/0 a případně jiných hodnot. Výslednou statistiku spolu s naměřeným časem pošlete do PC prostřednictvím s. portu.

Jelikož byly všechny záležitosti potřebné k implementaci již probrány, uvedeme pouze podobu posílaných řetězců. Poslání naměřeného času *time*:

```
serial.printf("DATA SAVING FINISHED! IT TOOK %f us. It means %f ns for saving
                ONE uint8_t\n\r", time, (double) (time/NUM_BYTES)*1000);
```

Informace o uložených datech, přičemž *num_ones*/*num_zeros* je počet bajtů rovných jedné/nule a *num_zeros* je počet bajtů neobsahujících číslo nula ani jedna:

```
serial.printf("NUMBER OF ONES:\t%d\tZEROS:\t%d\tUNSPECIFIED:\t%d!\n\r",
            num_ones, num_zeros, unspecified);
```

Výsledný program je uveden v příloze 12.20. Testovat ho lze připojením +3,3 *V* na daný pin (GND je připojena defaultně přes pull-down rezistor). V případě připojení na +3,3 *V* (0V) program načte 48 *kB* rovných jedné (nule). Průměrný čas načtení jedné hodnoty digitální sign. je cca **152,913 *ns***.

K testování by se hodilo použít generátor PWM signálu. Mbed třída PwmOut (viz kap. 3.9) umožňuje generovat PWM sign. s rozlišením na μs. V případě generování PWM sin. s šířkou pulzu 1 μs a střídou 50 % se jedná o digitální sign. pouze o frekvenci 500 *kHz*. Maximální frekvence vstupního sign. je

$$f = \frac{1}{T} = \frac{1}{152{,}913 \cdot 10^{-9}} \approx 6{,}539\ MHz. \tag{8.1}$$

PWM sign. o frekvenci v řádu *MHz* získáme pomocí softwaru LEO (viz kap. 3.9.7), který obsahuje PWM generátor s max. frekvencí 9 *MHz*. Využití softwaru LEO pro testování však předpokládá dvě desky Nukleo. Poznamenejme, že při testování programu na vysokofrekvenčním PWM sign. je potřeba počítat s nedokonalostmi.

Průběh sign. (uložená data) si lze zobrazit prostřednictvím grafické aplikace jako je např. Serial chart (viz [30]) nebo Serial plot plotter (viz [43]). Seznámení a práce s těmito programy **není** součástí této kapitoly. Zopakujme, že cílem je pouze implementace programů vykonávající co nejrychlejší uložení log. hodnoty daného pinu. Správná funkce programů tak pro jednoduchost stačí testovat na konstantním sign. o napětí +3,3 *V* (případně 0 *V)*.

8.2 Příklad: LA s využitím assembleru

V této části využijeme stejný program jako v předchozí části, ale s načítáním hodnot digitálního sign. v assembler funkci. Nechť má assembler funkce **tři vstupy**: základní adresu GPIO brány (viz obr. 5.1), počáteční adresu pole (pro uložení dat) a počet bajtů, jenž se má uložit.

V *main.cpp* přibudou dvě části. První pro zjištění základní adresy GPIO brány a čísla pinu:

```
//peripherals
#define GPIO_A 0x48000000
#define GPIO_B 0x48000400
#define GPIO_C 0x48000800
   …
//INPUT_PIN recognition
//0-15: PA_0 - PA_15;    16-31: PB_0 - PB_15;    32-47: PC_0 - PC_15
uint32_t pin_num=INPUT_PIN%16;
uint32_t peripheral_num;
char peripheral_char;
if (INPUT_PIN >= 32){
      peripheral_num = GPIO_C;
```

```
        peripheral_char = 67; //67 == "C"
} else {
        peripheral_num = (INPUT_PIN >= 16) ? GPIO_B : GPIO_A;
        peripheral_char = (INPUT_PIN >= 16) ? 66 : 65; // 65 == "A", 66 == "C"
}
```

Následně lze poslat označení pinu:

```
serial.printf("INPUT_PIN: P%c%d\n\r", peripheral_char, pin_num);
```

Druhá část se týká úpravy dat načtených v assembler funkci. Kvůli max. rychlosti načítání se totiž data z registru GPIOx_IDR (viz kap. 5.1.6.1) jen uloží (neupravují se). Navíc se z důvodu zvýšení rychlosti načítání ukládá pouze **první bajt**, tedy jako vstupní pin se dají použít piny s číslem 0 až 7 (brány GPIOA, GPIOB nebo GPIOC). Z toho důvodu je nutné dodatečné vymaskování konkrétního bitu (viz kap. 5.3.2.1). V případě použití dat by se takto upravila všechny data a uložila na stejné místo. V tomto případě však stačí upravit data v cyklu, ve kterém se zjišťuje počet načtených 1/0 příp. jiných hodnot. Samotná úprava může vypadat takto:

```
var = data[x];
var = var >> pin_num; //right shift
var = var & 0x1;  //mask first bit
```

Hlavní smyčka v assembler funkci (*memory.s*) vykonávající načítání:

```
//input arguments:
//  R0 = peripheral (address of GPIOA, GPIOB or GPIC),
//  R1 = MY_ADDRESS (address to SRAM - see main.cpp),
//  R2 = NUM_BYTES
    …
LOOP
    LDRB R3, [R0, #GPIO_IDR_Offset] //read from GPIOx_IDR just one byte
    STRB R3, [R1], #1 //save this byte to the R1 register (MY_ADRESS)
                      //and auto-increment
    SUBS R2, #1
    BNE LOOP
    …
```

Celý program je v příloze 12.21. Načítání a ukládání pouze prvního bajtu z registru GPIOx_IDR (díky omezení se na piny 0-7) proces značně urychlilo. Průměrný čas načtení jedné hodnoty se snížil na cca **125,244 *ns***, což umožňuje analyzovat digitální sign. o max. frekvenci blížící se **8 *MHz*** (viz vzorec (8.1)).

8.3 Příklad: LA s post-triggerem

V minulých dvou příkladech byl implementovaný ne zrovna v praxi použitelný LA, protože načítání dat po stisku klávesy z PC není moc užitečné. Užitečnější by bylo, kdyby se načítání

spustilo na náběžnou/spádovou hranu jiného signálu, což je přesně funkce LA s post-triggerem.

Příklad budeme implementovat přímo v assembleru, jelikož v mbedu by se nejednalo o velkou změnu. Soubor *main.cpp* je téměř identický jako v minulém příkladě. Nově přibude práce s triggerovacím pinem *TRIGGER_PIN*.

Do assembler funkce může vstoupit více argumentů než čtyři prostřednictvím předání adresy ukazující na místo v paměti, kde se požadované údaje nachází. Tímto způsobem se v tomto příkladě předává dohromady pět vstupů:

```
uint8_t data[NUM_BYTES];
uint32_t pins_info[3];
pins_info[0] = input_peripheral;
pins_info[1] = trig_peripheral;
pins_info[2] = trig_num;
memory(&pins_info[0], &data[0], NUM_BYTES);
```

V assembler funkci se na začátku programu čeká na změnu sign. triggerovacího pinu a až poté se začnou data ukládat. Celý program je v příloze 12.22.

9 Zhodnocení výsledků práce

Práce začíná kapitolou, ve které se student seznamuje s problematikou programování mikrokontroléru STM32F303RE prostřednictvím mbed tříd. V kapitole je důraz na srozumitelnost a názornou demonstraci na příkladech. Také se nepředpokládají žádné znalosti z oblasti elektroniky, proto se všechny pojmy před samotným používáním vysvětlují (případně se odkazuje na kvalitní zdroj).

Na poznatky získané v první kapitole se navazuje hlubším náhledem do samotného fungování mikrokontroléru. To zahrnuje představení blokového schématu mikrokontroléru, popis a vysvětlení funkcí jednotlivých částí (jádro, sběrnice, paměti, hodiny a periferie). V rámci získání nezávislosti na mbed třídách se student naučí sám si nakonfigurovat GPIO a hodiny. Součástí jsou proto také kapitoly objasňující použití bitových operací, práci s registry a orientaci v adresním prostoru.

Po obdržení veškerých základních a pokročilých znalostí o fungování mikrokontrolérů následuje seznámení s jinými způsoby programování v mbedu. První způsob je využití struktur pro práci s registry, které zpřehleňuje výsledný kód. Dále následuje seznámení s HAL a LL knihovnami, které mbed umožňuje využívat. Práce s HAL knihovnami je demonstrována na blikání LED a také posílání znaků přes sériový port prostřednictvím periferie UART.

Poslední vysvětlovaný způsob programování mikrokontrolérů v mbedu zahrnuje využití programovacího jazyka ARM assembler, pomocí něhož lze v mbedu psát funkce. Byla vytvořena příručka, která shrnuje základní použití zmíněného jazyka. Na závěr se získané znalosti využily při tvorbě vstupního bloku logického analyzátoru zaznamenávajícího data. Tento blok je vytvořen ve třech formách: bez triggeru s využitím mbed tříd, bez triggeru s načítací smyčkou psanou v assembleru a s post-triggerem s využitím assembleru. Porovnání prvních dvou forem demonstruje výhodu programování v assembleru, kterou je rychlost. S mbed třídami lze načítat signál o max. frekvenci cca 6,5 *MHz*, zatímco s využitím assembleru signál o max. frekvenci cca 8 *MHz*.

10 Závěr

Vytvořená práce představuje výukový materiál zabývající se programováním mikrokontrolérů s jádrem ARM Cortex M ve verzi STM32 (konkrétně mikrokontrolérem STM32F303RE) ve výjovém prostřed mbed IDE. Práce je koncipována tak, aby byla dostatečně srozumitelná i pro studenty středních škol, což zahrnuje názornou demonstraci vysvětlované teorie na příkladech. Také se nepředpokládají žádné počáteční znalosti z oblasti elektroniky. Problematika programování mikrokontrolérů je nejdříve vysvětlena na mbed třídách. Dále pokračuje hlubším vhledem do vnitřní struktury mikrokontrolérů a pokročilými metodami programování v mbedu, které zahrnují využití struktur a HAL a LL knihoven. Na závěr se student seznámí s nízkoúrovňovým programovacím jazykem ARM assembler, který lze v mbedu využít formou funkcí.

Získané znalosti nejsou použitelné pouze pro NUCLEO-F303RE, ale obecně pro moduly s mikrokontroléry s jádrem ARM Cortex M0, M3 a M4, které podporuje IDE mbed. Kromě firmy STMicroelectronics takové moduly nabízí např. NXP Semiconductors, Silicon Labs, Sigma Delta Technologies, SpeedStudio, AnalogDevices atd.

Na práci by se v budoucnu dalo navázet vysvětlením funkce čítačů a vytvořením demonstračních příkladů na toto téma. Dále by bylo vhodné ukázat, jak se pracuje v jiných vývojových prostředích, např. v IDE Keil. Přechod k zmíněnému prostředí měl být součástí této práce, avšak z důvodu nefunkčnosti exportování projektu z mbedu do IDE Keil nebyl realizován.

Výsledná práce bude využita jako výukový materiál v rámci předmětů na FEL, které se zabývají programováním mikrokontrolérů. Dále v projektech zaměřených na popularizaci elektroniky na úrovni středních škol a také na kurzech praktické elektroniky organizovaných katedrou měření ČVUT FEL. Nedokončená práce byla již využita při distanční výuce v předmětu B3B38LPE (Laboratoře z průmyslové elektroniky a senzorů) v rámci výjimečné situaci (koronavirus) v průběhu semestru (LS 2019/2020).

11 SEZNAM LITERATURY

[1] BIELESCH, Lukáš. *Optimální využití mbed IDE v laboratorní výuce* [online]. Praha, 2019 [cit. 2020-02-24]. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/82751/F3-BP-2019-Bielesch-Lukas-Optimalni_%20vyuziti_%20mbed_IDE_v_laboratorni_vyuce.pdf.pdf?sequence=-1&isAllowed=y. Bakalářská práce. České vysoké učení technické, Fakulta elektrotechnická. Vedoucí práce Jan Fischer.

[2] OBRK, Matúš. *Základní měřicí přístroje pro výukové laboratoře, realizované mikrořadičem* [online]. Praha, 2019 [cit. 2020-02-24]. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/82752/F3-BP-2019-Obrk-Matus-Zakladni_merici_pristroje_pro_vyukove_laboratore_realizovane_mikroradicem.pdf?sequence=-1&isAllowed=y. Bakalářská práce. České vysoké učení technické, Fakulta elektrotechnická. Vedoucí práce Jan Fischer.

[3] STM32 Nucleo-F303RE. In: *RIOT* [online]. Berlín: Hahm, c2013-2020 [cit. 2020-01-28]. Dostupné z: https://api.riot-os.org/group__boards__nucleo-f303re.html.

[4] *STM32 Nucleo-64 boards: UM1724* [online]. Rev 13. STMicroelectronics, c2019 [cit. 2020-02-12]. Dostupné z: https://www.st.com/content/ccc/resource/technical/document/user_manual/98/2e/fa/4b/e0/82/43/b7/DM00105823.pdf/files/DM00105823.pdf/jcr:content/translations/en.DM00105823.pdf

[5] IoT Device Development. *Arm MBED* [online]. c2020 [cit. 2020-02-05]. Dostupné z: https://www.mbed.com/en/

[6] Workspace Management. Arm *MBED* [online]. c2020 [cit. 2020-02-05]. Dostupné z: https://ide.mbed.com/compiler/

[7] Development boards. *Arm MBED* [online]. c2020 [cit. 2020-02-06]. Dostupné z: https://os.mbed.com/platforms/?q=NUCLEO-F303RE

[8] NUCLEO-F303RE. *Arm MBED* [online]. c2020 [cit. 2020-03-20]. Dostupné z: https://os.mbed.com/platforms/ST-Nucleo-F303RE/

[9] GUDINO, Miguel. Engineering Resources: Basics of Analog-to-Digital Converters. *Arrow* [online]. Arrow Electronics, c2020 [cit. 2020-02-12]. Dostupné z: https://www.arrow.com/en/research-and-events/articles/engineering-resource-basics-of-analog-to-digital-converters

[10] *STM32F303xD STM32F303xE* [online]. Rev 5. STMicroelectronics, c2016 [cit. 2020-02-12]. Dostupné z: https://www.st.com/resource/en/datasheet/stm32f303re.pdf

[11] *UM1786User manual: Description of STM32F3 HAL and low-layer drivers* [online]. Rev 8. STMicroelectronics, c2020 [cit. 2020-03-03]. Dostupné z: https://www.st.com/content/ccc/resource/technical/document/user_manual/a6/79/73/ae/6e/1c/44/14/DM00122016.pdf/files/DM00122016.pdf/jcr:content/translations/en.DM00122016.pdf

[12] *RM0316 Reference manual* [online]. Rev 8. STMicroelectronics, c2017 [cit. 2020-02-13]. Dostupné z: https://www.st.com/content/ccc/resource/technical/document/reference_manual/4a/19/6e/18/9d/92/43/32/DM00043574.pdf/files/DM00043574.pdf/jcr:content/translations/en.DM00043574.pdf

[13] PeripheralPins.c. *Arm MBED* [online]. c2020 [cit. 2020-02-14]. Dostupné z: https://os.mbed.com/users/mbed_official/code/mbed-

dev//file/f392fc9709a3/targets/TARGET_STM/TARGET_STM32F3/TARGET_STM32F303xE/TARGET_NUCLEO_F303RE/PeripheralPins.c/
[14] Stm32f303xe.h. *Arm MBED* [online]. c2020 [cit. 2020-04-03]. Dostupné z: https://os.mbed.com/users/mbed_official/code/mbed-dev//file/f392fc9709a3/targets/TARGET_STM/TARGET_STM32F3/TARGET_STM32F303xE/device/stm32f303xe.h/
[15] TARGET_STM32F3/device. *Arm MBED* [online]. c2020 [cit. 2020-04-07]. Dostupné z: https://os.mbed.com/users/mbed_official/code/mbed-dev//file/f392fc9709a3/targets/TARGET_STM/TARGET_STM32F3/device/
[16] Stm32f3xx_hal_gpio.c. *Arm MBED* [online]. c2020 [cit. 2020-04-07]. Dostupné z: https://os.mbed.com/users/mbed_official/code/mbed-dev//file/f392fc9709a3/targets/TARGET_STM/TARGET_STM32F3/device/stm32f3xx_hal_gpio.c/
[17] Stm32f3xx_hal_uart.h. *Arm MBED* [online]. c2020 [cit. 2020-04-09]. Dostupné z: https://os.mbed.com/users/mbed_official/code/mbed-dev//file/f392fc9709a3/targets/TARGET_STM/TARGET_STM32F3/device/stm32f3xx_hal_uart.h/
[18] TOULSON, Rob a Tim WILMSHURST. *Fast and Effective Embedded Systems Design - Applying the ARM mbed*. 2nd edition. Oxford: Newnes, c2012. ISBN 978-0-08-097768-3.
[19] AnalogOut. *Arm MBED* [online]. c2020 [cit. 2020-02-16]. Dostupné z: https://os.mbed.com/docs/mbed-os/v5.15/apis/analogout.html
[20] Ticker. *Arm MBED* [online]. c2020 [cit. 2020-02-25]. Dostupné z: https://os.mbed.com/docs/mbed-os/v5.15/apis/ticker.html
[21] Timeout. *Arm MBED* [online]. c2020 [cit. 2020-02-26]. Dostupné z: https://os.mbed.com/handbook/Timeout
[22] PwmOut. *Arm MBED* [online]. c2020 [cit. 2020-02-26]. Dostupné z: https://os.mbed.com/docs/mbed-os/v5.15/apis/pwmout.html
[23] Drivers overview. *Arm MBED* [online]. c2020 [cit. 2020-03-03]. Dostupné z: https://os.mbed.com/docs/mbed-os/v5.15/apis/drivers.html
[24] Mbed official. *Arm MBED* [online]. c2020 [cit. 2020-03-03]. Dostupné z: https://os.mbed.com/users/mbed_official/code/
[25] Memory Model. *Arm MBED* [online]. c2020 [cit. 2020-03-11]. Dostupné z: https://os.mbed.com/handbook/Memory-Model
[26] Assembly Language. *Arm MBED* [online]. c2020 [cit. 2020-04-15]. Dostupné z: https://os.mbed.com/cookbook/Assembly-Language
[27] TARGET_NUCLEO_F303RE. *Arm MBED* [online]. c2020 [cit. 2020-03-03]. Dostupné z: https://os.mbed.com/users/mbed_official/code/mbed-dev//file/f392fc9709a3/targets/TARGET_STM/TARGET_STM32F3/TARGET_STM32F303xE/TARGET_NUCLEO_F303RE/
[28] MALÝ, Martin. *HRADLA, VOLTY, JEDNOČIPY: Úvod do bastlení*. Praha: CZ.NIC, c2017. ISBN 978-80-88168-26-3. Kurz praktické elektroniky 2020: Příprava na kurz. *Embedded server: ČVUT v Praze - Fakulta elektrotechnická Katedra měření* [online]. c2019 [cit. 2020-02-20]. Dostupné z: https://embedded.fel.cvut.cz/kurzy/elektronika/elektronika2020
[29] HLEDÍK, Jiří. LEO - Little embedded oscilloscope. *Embedded server: ČVUT v Praze - Fakulta elektrotechnická Katedra měření* [online]. c2019 [cit. 2020-02-27]. Dostupné z: https://embedded.fel.cvut.cz/platformy/leo

[30] Materiály k seminářům BBC Micro:bit a LEO: Programy. *Embedded server: ČVUT v Praze - Fakulta elektrotechnická Katedra měření* [online]. c2019 [cit. 2020-04-17]. Dostupné z: https://embedded.fel.cvut.cz/platformy/bbc/seminar/materialy
[31] SANTOS, Rui a Sara SANTOS. Electronics Basics – How a Potentiometer Works: Voltage Divider. *RANDOM NERD TUTORIALS* [online]. Portugal, c2013-2020 [cit. 2020-02-23]. Dostupné z: https://randomnerdtutorials.com/electronics-basics-how-a-potentiometer-works/
[32] VALKOVIČ, Patrik. Lekce 1 - Kompilace v jazyce C a C++. *ITnetwork.cz* [online]. Praha, c2020 [cit. 2020-03-02]. Dostupné z: https://www.itnetwork.cz/cplusplus/cecko/pokrocile/kompilace-v-jazyce-c-a-cplusplus
[33] VALKOVIČ, Patrik. Lekce 2 - Kompilace v jazyce C a C++ pokračování. *ITnetwork.cz* [online]. Praha, c2020 [cit. 2020-03-02]. Dostupné z: https://www.itnetwork.cz/cplusplus/cecko/pokrocile/kompilace-v-jazyce-c-a-cplusplus-pokracovani
[34] JANČÍK, David. Sběrnice. *ITnetwork.cz* [online]. Praha, c2020 [cit. 2020-03-06]. Dostupné z: https://www.itnetwork.cz/hardware-pc/hardware/tvy-sbernice
[35] Compilation process in c. *JavaTpoint* [online]. India, c2011-2018 [cit. 2020-03-02]. Dostupné z: https://www.javatpoint.com/compilation-process-in-c
[36] Úvod. *Moodle Fakulta elektrotechnická* [online]. [cit. 2020-03-06]. Dostupné z: https://moodle.fel.cvut.cz/pluginfile.php/47322/course/section/10709/LPE_2018_Uvod_a_zaklady_v01.pdf
[37] TIŠNOVSKÝ, Pavel. Statické a dynamické paměti. *ROOT.CZ* [online]. c1998-2020, 12. 6. 2008 [cit. 2020-03-13]. Dostupné z: https://www.root.cz/clanky/staticke-a-dynamicke-pameti/
[38] DUDKA, Michal. Přístupy k programování STM32. *ROOT.CZ* [online]. c1998-2020, 3. 10. 2017 [cit. 2020-04-04]. Dostupné z: https://www.root.cz/clanky/pristupy-k-programovani-stm32/
[39] MARTIN, Trevor. *The Insider's Guide To The STM32 ARM®Based Microcontroller: An Engineer's Introduction To The STM32 Series Version 1.8* [online]. 2nd ed. United Kingdom: Hitex, 2009 [cit. 2020-03-21]. ISBN 0-95499888. Dostupné z: https://www.hitex.com/fileadmin/documents/tools/dev_tools/dt_protected/insiders-guides/stm32/isg-stm32-v18d-scr.pdf
[40] Using the STM32 UART interface with HAL. *VisualGDB* [online]. Sysprogs OÜ, c2012-2020 [cit. 2020-04-09]. Dostupné z: https://visualgdb.com/tutorials/arm/stm32/uart/hal/
[41] BENINGO, Jacob. USART vs UART: Know the difference. *EDN* [online]. AspenCore, c2020 [cit. 2020-04-10]. Dostupné z: https://www.edn.com/usart-vs-uart-know-the-difference/
[42] PM0214 Programming manual: STM32 Cortex®-M4 MCUs and MPUs programming manual. *STMicroelectronics* [online]. c2020 [cit. 2020-04-10]. Dostupné z: https://www.st.com/resource/en/programming_manual/dm00046982.pdf
[43] KEREZIEV, Borislav. Serial Port Plotter. *GitHub repository* [online]. c2020 [cit. 2020-04-17]. Dostupné z: https://github.com/CieNTi/serial_port_plotter

12 Přílohy

12.1 Výsledné řešení pro kap. 3.5.1

```
//---------------------------------------//
//switch LED on/off if the INPUT_PIN is high/low
//for WAIT_TIME ms.
//DON'T FORGET TO ADD OHM BEFORE CONNECTING!

//---------------------------------------//
//libraries
#include "mbed.h"

//---------------------------------------//
//constants
#define INPUT_PIN PC_0 //optional pin
#define WAIT_TIME 50 //in ms

//---------------------------------------//
//init
DigitalOut led(LED1);
DigitalIn input_pin(INPUT_PIN);

//---------------------------------------//
//program
int main() {
    while(1){
        if (input_pin){
            led = 1;
        } else {
            led = 0;
        }
        wait_ms(WAIT_TIME);
    }
}
```

12.2 Výsledné řešení pro kap. 3.6.2

```
//---------------------------------------//
//return a sign received from PC via UART

//---------------------------------------//
//libraries
#include "mbed.h"

//---------------------------------------//
//init
Serial serial(PA_2, PA_3);
DigitalOut led(LED1);

//---------------------------------------//
//program
int main() {
    //flashing at the beginning
    for (int i = 0; i < 5*2; ++i) {
        led = !led;
        wait(0.05);
    }

    char sign;
    while(1){
        if (serial.readable()){
            sign = serial.getc();
            if (serial.writeable()){
                serial.putc(sign);
            }
        }
    }
}
```

12.3 Výsledné řešení pro kap. 3.6.3

```
//---------------------------------------//
//Load the user's name to the array called name.
//After typing the name, press Enter.
//If an error occurs, send an appropriate message.
//---------------------------------------//
//libraries
#include "mbed.h"
#include "ctype.h" //to use the function isalpha()
//---------------------------------------//
//constants
#define MAX_LEN_NAME 25
//---------------------------------------//
//init
Serial serial(SERIAL_TX, SERIAL_RX);
DigitalOut led(LED1);
//---------------------------------------//
//program
int main() {
    //flashing at the beginning
    for (int i = 0; i < 5*2; ++i) {
        led = !led;
        wait(0.05);
    }
    serial.printf("\n\rWrite your name and press ENTER.\r\n"); //introductory message
    char name[MAX_LEN_NAME];
    char sign;
    bool quit=false; //do until the variable is false
    uint8_t index=0; //position of the last character in the name array
    while(!quit){
        if (serial.readable()){ //anything to read?
            sign=serial.getc();
            if (sign == 13){ //13=Enter; end of loading the user's name
                if (index){
                    serial.printf("\n\rHELLO! Nice to meet you, %s!\r\n", name);
                    quit=true;
                } else { //Enter pressed BUT no character has been loaded.
                    serial.printf("\n\rWARN: No character has been loaded. Write your name and
press ENTER.\r\n");
                }
            }
```

```
        else if (index==MAX_LEN_NAME-1){ //is there enough space to load another character?
            serial.printf("\n\rWARN: Loading completed. Not enough space to load another character.\r\n"
            "The string '%s' has been loaded so far. If your name has truly more than %d characters, "
            "modify the constant MAX_LEN_NAME.\r\n", name, MAX_LEN_NAME);
            quit=true;
        }

        else { //yes, there is...
            if (isalpha(sign)){ //is the loaded character in alphabet?
                name[index]=sign;
                name[index+1]='\0';
                index+=1;
                serial.putc(sign);
            } else { //no, it isn't
                serial.printf("\n\rERROR: The string '%s' has been loaded so far BUT the next character is '%c' (in ASCII '%d') - IT "
                "IS NOT IN ALPHABET! Reset circuit and try it again!\r\n", name, sign, sign);
                quit=true;
            }
        }
    }

    //flashing at the end
    for (int i = 0; i < 5*2; ++i) {
        led = !led;
        wait(0.05);
    }
}
```

12.4 Výsledné řešení pro kap. 3.8.2.1

```
//---------------------------------------//
//return a sign received from PC via UART
//while flipping the LED

//---------------------------------------//
//libraries
#include "mbed.h"

//---------------------------------------//
//init
Serial serial(PA_2, PA_3);
DigitalOut led(PA_5);
Ticker ticker;

//---------------------------------------//
void flip(){
    led = !led;
}

//---------------------------------------//
//program
int main() {
    ticker.attach(&flip, 0.2); //the LED flips every 0.2 seconds
    char sign;
    while(1){
        if (serial.readable()){
            sign = serial.getc();
            if (serial.writeable()){
                serial.putc(sign);
            }
        }
    }
}
```

12.5 Výsledné řešení pro kap. 3.8.3.1

```
//---------------------------------------//
//The program measures reaction times. The program waits for
//the button to be pressed at the beggining. Pressing the button
//starts the measurement. Trial results are sent to PC via serial port.
//If any trial result is worse than the MAX_REACTION_TIME, the program
//ends. The program also ends if the button is pressed before the LED lights up.

//---------------------------------------//
//libraries
#include "mbed.h"
#include "Debug.h"

//---------------------------------------//
//constants
#define MAX_REACTION_TIME 275 //in ms

//---------------------------------------//
//init classes
DigitalOut led(PA_5);
DigitalIn button(PC_13);
Debug_led my_debug(PA_5, PC_13);
Serial serial(PA_2, PA_3);
Timer trial;

//---------------------------------------//
int main() {
   srand(time(NULL)); //for non-repeating sequences
   bool quit=false;
   uint32_t trial_time_in_ms, num;

   //introductory information
   serial.printf("\tPUSH THE BUTTON TO BEGIN\r\n");
   serial.printf("You have to have less than %d ms otherwise the game is over.\r\n",
MAX_REACTION_TIME);
   serial.printf("Let's begin, good luck!\r\nYour times:\r\n");

   my_debug.breakpoint(3); //push the user button to start measurement
   while (!button){ } //is the button still pressed?
```

```
   while(!quit){
      num = (rand()%2000)+300; //how long to wait before the LED lights up

      while(num){ //wait approximately num ms but watch the button
         num--;
         if (!button){
            quit=true; //the button pressed before the LED lights up
            num=0;
         }
         wait_ms(1);
      }

      if (!quit){
         led=!led; //the LED lights up
         trial.start(); //start to measure time
         while (button){} //loop until the button is not pressed
         trial.stop(); //the button is pushed

         //evaluation of the result
         trial_time_in_ms=trial.read_ms(); //get the result in ms
         if (trial_time_in_ms<=MAX_REACTION_TIME){ //the button was pressed in
time
            serial.printf("\t%d ms\r\n", trial_time_in_ms);
            trial.reset(); //DO NOT FORGET TO RESET
         } else {
            serial.printf("\tYour time is %d ms. -> GAME OVER!\n\r", trial_time_in_ms);
            quit=true;
         }

         //time to see the result
         while(!button){} //loop until the button is pressed
         led=!led; //turn off the LED

      } else { //button was pressed before the LED lights up
         serial.printf("The button was pushed too quickly. -> GAME OVER!\r\n");
      }
   }
   my_debug.breakpoint(3); //end of the program
}
```

12.6 Výsledné řešení pro kap. 3.9.1

```
//---------------------------------------//
//Introductory example to PWM

//---------------------------------------//
//libraries
#include "mbed.h"
#include "Debug.h"

//---------------------------------------//
//constants
#define BASE_TIME 50000 //is us
#define NUM_STEPS 20
#define STEP BASE_TIME/NUM_STEPS

//---------------------------------------//
//init classes
DigitalOut led(PA_5);
DigitalIn button(PC_13);
Debug_led my_debug(PA_5, PC_13, "BUTTON_TO_GND");

//---------------------------------------//
int main() {
    uint16_t time = BASE_TIME;
    while(time){
        if(!button){
            time -= STEP;
            while(!button){}
        }
        led = 1;
        wait_us(time);
        led = 0;
        wait_us(BASE_TIME-time);
    }
    my_debug.breakpoint(3);
}
```

12.7 Výsledné řešení pro kap. 3.11.1

```
//---------------------------------------//
//Wait until the button is pressed, then measure
//the voltage at pin ANALOG_INPUT. Send the result
//of functions read() and read_u16().

//---------------------------------------//
//libraries
#include "mbed.h"
#include "Debug.h"

//---------------------------------------//
//constants
#define ANALOG_INPUT PB_11 //pay attention to the pin selection

//---------------------------------------//
//init
AnalogIn an_in(ANALOG_INPUT);
Serial serial(PA_2, PA_3);
Debug_led my_debug(PA_5, PC_13);

//---------------------------------------//
//program
int main() {
    char pin_port;
    uint8_t pin_num = ANALOG_INPUT%16;
    if (ANALOG_INPUT >= 32){
        pin_port = 'C';
    } else {
        pin_port = ANALOG_INPUT >= 16 ? 'B' : 'A';
    }
    while(1){
        serial.printf("Voltage on pin P%c%lu:\tread()=%f\tread_u16()=%lu\r\n",
                pin_port, pin_num, an_in.read(), an_in.read_u16());
    }
}
```

12.8 Výsledné řešení pro kap. 3.11.3

```
//---------------------------------------//
//Measure every 2 sec voltage on pin ANALOG_INPUT.
//Also measure the reference voltage for accurate
//measurement. Send the result in V to PC via serial port.

//---------------------------------------//
//libraries
#include "mbed.h"

//---------------------------------------//
//constants
#define VREF_CAL *(uint16_t *)0x1FFFF7BA //calibration constant
#define ANALOG_INPUT PB_11 //pay attention to the pin selection
#define FULL_SCALE 65535 //2^16-1

//---------------------------------------//
//init
AnalogIn an_in(ANALOG_INPUT);
AnalogIn an_vref(ADC_VREF);
Serial serial(PA_2, PA_3);

//---------------------------------------//
//program
int main() {
    float V_IN, V_DDA;
    char pin_port;
    uint8_t pin_num = ANALOG_INPUT%16;
    if (ANALOG_INPUT >= 32){
        pin_port = 'C';
    } else {
        pin_port = ANALOG_INPUT >= 16 ? 'B' : 'A';
    }
    while(1){
        V_DDA = 3.3f*VREF_CAL/(an_vref.read_u16()>>4); //get the reference voltage
        V_IN = V_DDA*((float) an_in.read_u16()/FULL_SCALE);
        serial.printf("Voltage on pin P%c%lu:\t%1.4f V\r\n", pin_port, pin_num, V_IN);
        wait(2);
    }
}
```

12.9 Výsledné řešení pro kap. 3.12.1

```
//-------------------------------------//
//Increase/decrease the output voltage at user LED (PA_5)
//by pressing 's'/'d' in a terminal.

//-------------------------------------//
//libraries
#include "mbed.h"

//-------------------------------------//
//constants
#define STEP 0.01f

//-------------------------------------//
//init classes
AnalogOut led(PA_5);
Serial serial(PA_2, PA_3);

//-------------------------------------//
int main() {
    float value = 0.5f;
    led.write(value);
    serial.printf("Press 's' / 'd' to increase / decrease the output voltage.\r\n"
            "Output voltage: %1.3f V.", led.read()*3.3f);
    wait(1);
    char sign;
    while(1){
        if (serial.readable()){
            sign = serial.getc();
            if (sign == 's' || sign == 'd'){
                if (sign == 's'){ //increase output voltage
                    value += STEP;
                } else { //'d' -> decrease output voltage
                    value -= STEP;
                }
                led.write(value);
                serial.printf("\rOutput voltage: %1.3f V.", led.read()*3.3f);
            }
        }
    }
}
```

12.10 Výsledné řešení pro kap. 4.6.1

```
//---------------------------------------//
//Introductory to memories on the chip

//---------------------------------------//
//libraries
#include "mbed.h"

//---------------------------------------//
//constants
const int constant1 = 1;
const uint8_t constant2 = 2;
const uint16_t constant3 = 3;
const uint32_t constant4 = 4;

//---------------------------------------//
//init classes
Serial serial(PA_2, PA_3);

//---------------------------------------//
//init functions
void func(){
    uint8_t func_var1;
    uint16_t func_var2;
    uint32_t func_var3;
    serial.printf("func_var1:\t%x\n\r", &func_var1);
    serial.printf("func_var2:\t%x\n\r", &func_var2);
    serial.printf("func_var3:\t%x\n\n\r", &func_var3);
}

//---------------------------------------//
int main() {
    uint8_t var1;
    uint16_t var2;
    uint32_t var3;
    serial.printf("var1:\t%x\n\r", &var1);
    serial.printf("var2:\t%x\n\r", &var2);
    serial.printf("var3:\t%x\n\n\r", &var3);

    func();
```

```
    uint8_t* buffer1 = (uint8_t*) malloc(sizeof(uint8_t)*3);
    uint16_t* buffer2 = (uint16_t*) malloc(sizeof(uint16_t)*3);
    serial.printf("&buffer1[0]:\t%x\n\r", &buffer1[0]);
    serial.printf("&buffer1[1]:\t%x\n\r", &buffer1[1]);
    serial.printf("&buffer1[2]:\t%x\n\r", &buffer1[2]);
    serial.printf("&buffer2[0]:\t%x\n\r", &buffer2[0]);
    serial.printf("&buffer2[1]:\t%x\n\r", &buffer2[1]);
    serial.printf("&buffer2[2]:\t%x\n\n\r", &buffer2[2]);

    serial.printf("constant1:\t%x\n\r", &constant1);
    serial.printf("constant2:\t%x\n\r", &constant2);
    serial.printf("constant3:\t%x\n\r", &constant3);
    serial.printf("constant4:\t%x\n\r", &constant4);
}
```

12.11 Výsledné řešení pro kap. 5.1.7

```
//---------------------------------------//
//Configure LED and use it to flashing.
//Do it using GPIO registers.

//---------------------------------------//
//libraries
#include "mbed.h"

//---------------------------------------//
//constants
#define GPIOA_MODER 0x48000000 //GPIOA(0x48000000) + MODER_offset(0x00)
#define GPIOA_ODR 0x48000014   //GPIOA(0x48000000) + ODR_offset(0x14)
#define GPIOA_BSRR 0x48000018  //GPIOA(0x48000000) + BSRR_offset(0x18)
#define GPIOA_BRR 0x48000028   //GPIOA(0x48000000) + BRR_offset(0x28)

//---------------------------------------//
//init classes
Serial serial (PA_2, PA_3); //be sure to create an object of the class Serial

//---------------------------------------//
int main() {
   *((volatile uint32_t *)GPIOA_MODER) |= 0b01 << 2*5; //0b - binary format
   while(1){
      //switch the LED on (use one option, comment the other)
//       *((volatile uint32_t *)GPIOA_ODR) |=  0b1 << 5; //using GPIOA_ODR
      *((volatile uint32_t *)GPIOA_BSRR) |=  0b1 << 5; //using GPIOA_BSRR
      wait(1);
      //switch the LED off (use one option, comment another)
//       *((volatile uint32_t *)GPIOA_ODR) &=  ~(0b1 << 5); //using GPIOA_ODR
//       *((volatile uint32_t *)GPIOA_BSRR) |=  0b1 << 16+5; //using GIOA_BSRR
      *((volatile uint32_t *)GPIOA_BRR)  |=  0b1 << 5; //using GPIOA_BRR
      wait(1);
   }
}
```

12.12 Výsledné řešení pro kap. 5.3.4

```
//---------------------------------------//
//Create an object of DigitalIn and Debug_register_print class
//and send values of registers RCC_CR, RCC_CFGR, RCC_AHBENR, RCC_CFGR2
//and FLASH_ACR in binary format using Debug_register_print class

//---------------------------------------//
//libraries
#include "mbed.h"
#include "Debug.h"

//---------------------------------------//
//constants
#define RCC_CR 0x40021000 //RCC(0x40021000) + CR_offset(0x0)
#define RCC_CFGR 0x40021004 //RCC(0x40021000) + CFGR_offset(0x04)
#define RCC_AHBENR 0x40021014 //RCC(0x40021000) + AHBENR_offset(0x14)
#define RCC_CFGR2 0x4002102C //RCC(0x40021000) + AHBENR_offset(0x2c)
#define FLASH_ACR 0x40022000 //FLASH(0x40022000) + ACR_offset(0x0);

//---------------------------------------//
//init classes
DigitalIn my_PC10 (PC_10);
Debug_register_print device(PA_2, PA_3, 9600);

//---------------------------------------//
int main() {
    device.format(3);
    device.breakpoint(__LINE__,RCC_CR);
    device.breakpoint(__LINE__,RCC_CFGR);
    device.breakpoint(__LINE__,RCC_AHBENR);
    device.breakpoint(__LINE__,RCC_CFGR2);
    device.breakpoint(__LINE__,FLASH_ACR);
}
```

12.13 Výsledné řešení pro kap. 5.4.2

```
//---------------------------------------//
//Configure clock and GPIO to flash user LED.
//Use the HSE bypass as system clock.

//---------------------------------------//
//libraries
#include <stdint.h>
#include "stm32f3xx_hal.h"

//---------------------------------------//
//constants
#define RCC_CR 0x40021000 //RCC(0x40021000) + CR_offset(0x0)
#define RCC_CFGR 0x40021004 //RCC(0x40021000) + CFGR_offset(0x04)
#define RCC_AHBENR 0x40021014 //RCC(0x40021000) + AHBENR_offset(0x14)
#define GPIOA_MODER 0x48000000 //GPIOA(0x48000000) + MODER_offset(0x00)
#define GPIOA_ODR 0x48000014   //GPIOA(0x48000000) + ODR_offset(0x14)
#define MASK_HSERDY 1<<17 //Mask of the HSERDY bit

//---------------------------------------//
int main() {
   *((volatile uint32_t *)RCC_CR) |= 0b101 << 16; //set HSEBYP and HSEON
   while(!(*((volatile uint32_t *)RCC_CR) & MASK_HSERDY)){} //is the HSE ready?
   *((volatile uint32_t *)RCC_CFGR) |= 0b01; //select HSE as system clock
   *((volatile uint32_t *)RCC_AHBENR) |= 0b1 << 17; //enable clock to GPIOA
   *((volatile uint32_t *)GPIOA_MODER) |= 0b01 << 2*5;
   while(1){
      *((volatile uint32_t *)GPIOA_ODR) |=  0b1 << 5; //switch the LED on
      HAL_Delay(1000);
      *((volatile uint32_t *)GPIOA_ODR) &=  ~(0b1 << 5); //switch the LED off
      HAL_Delay(1000);
   }
}
```

12.14 Výsledné řešení pro kap. 5.4.3

```
//---------------------------------------//
//Configure clock and GPIO to flash user LED.
//Use the PLL as system clock.

//---------------------------------------//
//libraries
#include <stdint.h>
#include "stm32f3xx_hal.h"

//---------------------------------------//
//constants
#define RCC_CR 0x40021000 //RCC(0x40021000) + CR_offset(0x0)
#define RCC_CFGR 0x40021004 //RCC(0x40021000) + CFGR_offset(0x04)
#define RCC_AHBENR 0x40021014 //RCC(0x40021000) + AHBENR_offset(0x14)
#define GPIOA_MODER 0x48000000 //GPIOA(0x48000000) + MODER_offset(0x00)
#define GPIOA_ODR 0x48000014   //GPIOA(0x48000000) + ODR_offset(0x14)
#define FLASH_ACR 0x40022000 //FLASH(0x40022000) + ACR_offset(0x0);
#define MASK_HSERDY 1<<17 //Mask of the HSERDY bit
#define MASK_PLLRDY 1<<25 //Mask of the PLLRDY bit

//---------------------------------------//
int main() {
   //these bits can be written only when PLL is disabled:
   *((volatile uint32_t *)RCC_CFGR) |= 0b0111 << 18; //set PLLMUL -> „0111“ = x 9
   *((volatile uint32_t *)RCC_CFGR) |= 0b10 << 15; //set PLLSRC
   *((volatile uint32_t *)RCC_CFGR) |= 0b100 << 8; //set PRE1
   *((volatile uint32_t *)RCC_CR) |= 0b101 << 16; //set HSEBYP and HSEON
   while(!(*((volatile uint32_t *)RCC_CR) & MASK_HSERDY)){} //is the HSE ready?
   *((volatile uint32_t *)RCC_CR) |= 0b1 << 24; //set PLLON
   while(!(*((volatile uint32_t *)RCC_CR) & MASK_PLLRDY)){} //is the PLL ready?
   *((volatile uint32_t *)FLASH_ACR) |= 0b010; //add two wait states
   *((volatile uint32_t *)RCC_CFGR) |= 0b10; //select the PLL as system clock
   *((volatile uint32_t *)RCC_AHBENR) |= 0b1 << 17; //enable clock to GPIOA
   *((volatile uint32_t *)GPIOA_MODER) |= 0b01 << 2*5;
   while(1){
      *((volatile uint32_t *)GPIOA_ODR) |=  0b1 << 5; //switch the LED on
      HAL_Delay(1000);
      *((volatile uint32_t *)GPIOA_ODR) &=  ~(0b1 << 5); //switch the LED off
      HAL_Delay(1000);
   }
}
```

12.15 Výsledné řešení pro kap. 6.1.2

```
//---------------------------------------//
//Configure clock and GPIO to flash user LED.
//Use registry access structures.
//Determine the PLL as system clock.

//---------------------------------------//
//libraries
#include "stm32f3xx_hal.h"

//---------------------------------------//
int main() {
   //these bits can be written only when PLL is disabled:
   RCC->CFGR |= RCC_CFGR_PLLMUL9; //set PLLMUL -> „0111“ = x 9
   RCC->CFGR |= RCC_CFGR_PLLSRC_HSE_PREDIV; //set PLLSRC
   RCC->CFGR |= RCC_CFGR_PPRE1_DIV2; //set PRE1

   RCC->CR |= RCC_CR_HSEBYP; //set HSEBYP
   RCC->CR |= RCC_CR_HSEON; //set HSEON
   while(!(RCC->CR & RCC_CR_HSERDY)){} //is the HSE ready?

   RCC->CR |= RCC_CR_PLLON; //set PLLON
   while(!(RCC->CR & RCC_CR_PLLRDY)){} //is the PLL ready?

   FLASH->ACR |= FLASH_ACR_LATENCY_1; //add two wait states

   RCC->CFGR |= RCC_CFGR_SW_PLL; //select the PLL as system clock

   RCC->AHBENR |= RCC_AHBENR_GPIOAEN; //enable clock to GPIOA
   GPIOA->MODER |= GPIO_MODER_MODER5_0;
   while(1){
      GPIOA->ODR |=  GPIO_ODR_5; //switch the LED on
      HAL_Delay(1000);
      GPIOA->ODR &=  ~(GPIO_ODR_5); //switch the LED off
      HAL_Delay(1000);
   }
}
```

12.16 Výsledné řešení pro kap. 6.2.6

```
//---------------------------------------//
//Receive and send back a character via USART2,
//use HAL libraries. Example inspired by
//https://visualgdb.com/tutorials/arm/stm32/uart/hal/

//---------------------------------------//
//libraries
#include <stm32f3xx_hal.h>

//---------------------------------------//
int main(void)
{
   __GPIOA_CLK_ENABLE();
   __USART2_CLK_ENABLE();

   GPIO_InitTypeDef GPIO_PA2_InitStructure;
   GPIO_PA2_InitStructure.Pin = GPIO_PIN_2;
   GPIO_PA2_InitStructure.Mode = GPIO_MODE_AF_PP; //Push Pull Mode
   GPIO_PA2_InitStructure.Pull = GPIO_NOPULL;;
   //alternate option to GPIO_MODE_AF_PP:
//    GPIO_PA2_InitStructure.Mode = GPIO_MODE_AF_OD; //Open Drain mode
//    GPIO_PA2_InitStructure.Pull = GPIO_PULLUP; //with Pull-up resistor
   GPIO_PA2_InitStructure.Alternate = GPIO_AF7_USART2;
   GPIO_PA2_InitStructure.Speed = GPIO_SPEED_HIGH;
   HAL_GPIO_Init(GPIOA, &GPIO_PA2_InitStructure);

   GPIO_InitTypeDef GPIO_PA3_InitStructure;
   GPIO_PA3_InitStructure.Pin = GPIO_PIN_3;
   GPIO_PA3_InitStructure.Mode = GPIO_MODE_AF_OD;
   GPIO_PA3_InitStructure.Pull = GPIO_NOPULL;;
   GPIO_PA3_InitStructure.Alternate = GPIO_AF7_USART2;
   GPIO_PA3_InitStructure.Speed = GPIO_SPEED_HIGH;
   HAL_GPIO_Init(GPIOA, &GPIO_PA3_InitStructure);
```

```
   UART_InitTypeDef myUART_Init;
   myUART_Init.BaudRate   = 9600;
   myUART_Init.WordLength = UART_WORDLENGTH_8B;
   myUART_Init.StopBits   = UART_STOPBITS_1;
   myUART_Init.Parity     = UART_PARITY_NONE;
   myUART_Init.Mode       = UART_MODE_TX_RX;
   myUART_Init.HwFlowCtl  = UART_HWCONTROL_NONE;

   UART_HandleTypeDef myUART_Handle;
   myUART_Handle.Instance = USART2;
   myUART_Handle.Init = myUART_Init;

   uint8_t buffer[1];
   if (HAL_UART_Init(&myUART_Handle) == HAL_OK){
      while(1){
         HAL_UART_Receive(&myUART_Handle,  buffer,  sizeof(buffer),
HAL_MAX_DELAY);
         HAL_UART_Transmit(&myUART_Handle,  buffer,  sizeof(buffer),
HAL_MAX_DELAY);
      }
   }
}
```

12.17 Výsledné řešení pro kap. 6.2.7

```
//-------------------------------------//
//Send frequency of SYSCLK, HCLK and PCLK1 via USART2

//-------------------------------------//
//libraries
#include <stm32f3xx_hal.h>

//-------------------------------------//
int main(void)
{
   __USART2_CLK_ENABLE();
   __GPIOA_CLK_ENABLE();

   GPIO_InitTypeDef GPIO_PA2_InitStructure;
   GPIO_PA2_InitStructure.Pin = GPIO_PIN_2;
   GPIO_PA2_InitStructure.Mode = GPIO_MODE_AF_PP; //Push Pull Mode
   GPIO_PA2_InitStructure.Pull = GPIO_NOPULL;
   GPIO_PA2_InitStructure.Alternate = GPIO_AF7_USART2;
   GPIO_PA2_InitStructure.Speed = GPIO_SPEED_HIGH;
   HAL_GPIO_Init(GPIOA, &GPIO_PA2_InitStructure);

   GPIO_InitTypeDef GPIO_PA3_InitStructure;
   GPIO_PA3_InitStructure.Pin = GPIO_PIN_3;
   GPIO_PA3_InitStructure.Mode = GPIO_MODE_AF_OD;
   GPIO_PA3_InitStructure.Pull = GPIO_NOPULL;
   GPIO_PA3_InitStructure.Alternate = GPIO_AF7_USART2;
   GPIO_PA3_InitStructure.Speed = GPIO_SPEED_HIGH;
   HAL_GPIO_Init(GPIOA, &GPIO_PA3_InitStructure);

   UART_InitTypeDef myUART_Init;
   myUART_Init.BaudRate   = 9600;
   myUART_Init.WordLength = UART_WORDLENGTH_8B;
   myUART_Init.StopBits   = UART_STOPBITS_1;
   myUART_Init.Parity     = UART_PARITY_NONE;
   myUART_Init.Mode       = UART_MODE_TX_RX;
   myUART_Init.HwFlowCtl  = UART_HWCONTROL_NONE;

   UART_HandleTypeDef myUART_Handle;
   myUART_Handle.Instance = USART2;
   myUART_Handle.Init = myUART_Init;
   HAL_UART_Init(&myUART_Handle);
```

```
    char sysclk_buff[] = "SYSCLK: ";
  HAL_UART_Transmit(&myUART_Handle, (uint8_t *)sysclk_buff,
                                sizeof(sysclk_buff), HAL_MAX_DELAY);

  uint32_t new_var = HAL_RCC_GetSysClockFreq();
  char buffer[10];
  sprintf(buffer, "%d\n\r", new_var);
  HAL_UART_Transmit(&myUART_Handle, (uint8_t *)buffer, sizeof(buffer),
                                HAL_MAX_DELAY);

  char hclk_buff[] = "HCLK: ";
  HAL_UART_Transmit(&myUART_Handle, (uint8_t *)hclk_buff, sizeof(hclk_buff),
                                HAL_MAX_DELAY);
  new_var = HAL_RCC_GetHCLKFreq();
  sprintf(buffer, "%d\n\r", new_var);
  HAL_UART_Transmit(&myUART_Handle, (uint8_t *)buffer, sizeof(buffer),
                                HAL_MAX_DELAY);

  char pclk1_buff[] = "PCLK1: ";
  HAL_UART_Transmit(&myUART_Handle, (uint8_t *)pclk1_buff, sizeof(pclk1_buff),
                                HAL_MAX_DELAY);
  new_var = HAL_RCC_GetPCLK1Freq();
  sprintf(buffer, "%d\n\r", new_var);
  HAL_UART_Transmit(&myUART_Handle, (uint8_t *)buffer, sizeof(buffer),
                                HAL_MAX_DELAY);
}
```

12.18 Výsledné řešení pro kap. 7.6.1.3

Soubor *main.cpp*:

```
//-----------------------------------//
//LED flashing using assembly function

//-----------------------------------//
//libraries
#include "mbed.h"

//-----------------------------------//
//init
DigitalOut led(PA_5);
```

```
//---------------------------------------//
//init extern functions
extern "C" void change_state();

//---------------------------------------//
//program
int main() {
    while(1) {
        change_state();
        wait(0.5);
    }
}
```

Soubor *change_state.s*:

```
;---------------------------------------;
; change the state of the 5th bit in GPIOA_ODR

;---------------------------------------;
; constants
GPIOA EQU 0x48000000
ODR_offset EQU 0x14

;---------------------------------------;
; program
    AREA asm_func, CODE, READONLY
    EXPORT change_state

change_state
    PUSH{LR}

    LDR R0, =GPIOA ; R0 = GPIOA
    LDR R1, [R0, #ODR_offset] ; load value of R0 + ODR_offset to R1
    EOR R2, R1, #(0x1 :SHL: 5) ; R2 = R1 ^ (0x1 << 5)
    STR R2, [R0, #ODR_offset] ; save the R2 registry value to R0 + ODR_offset

    POP {PC}
    ALIGN
    END
```

12.19 Výsledné řešení pro kap. 7.8

Soubor *main.cpp*:

```
//---------------------------------------//
//just initialize and call the assembly function flash.s

//---------------------------------------//
//libraries
#include <stdint.h> //because of the data type uint32_t

//---------------------------------------//
//init extern functions
extern "C" void flash(uint32_t period_ms);

//---------------------------------------//
//program
int main() {
   flash(500); //period of flashing is 500 ms
}
```

Soubor *flash.s*:

```
;---------------------------------------;
; set and configure everything needed for the LED flashing

;---------------------------------------;
;constants
RCC EQU 0x40021000
GPIOA EQU 0x48000000
FLASH EQU 0x40022000

;offsets
CR_offset EQU 0x0
CFGR_offset EQU 0x04
AHBENR_offset EQU 0x14
MODER_offset EQU 0x00
ODR_offset EQU 0x14
ACR_offset EQU 0x0

;---------------------------------------;
;program
   AREA asm_func, CODE, READONLY
   EXPORT flash
```

```
flash
   PUSH{LR}
   BL CLOCK_INIT
   BL GPIO_INIT

   ; number of miliseconds is stored in register R0
MAIN_LOOP
   BL CHANGE_LED_STATE
   BL WAIT_MS
   B MAIN_LOOP
   POP {PC}

;--------------------------------------;
; init SYSCLK as PLL (72 MHz)
CLOCK_INIT PROC
   PUSH{R0, R1, R2, LR}

   ; these bits can be written only when PLL is disabled:
   LDR R0, =RCC
   LDR R1, [R0, #CFGR_offset]
   ORR R1, #(0x7 :SHL: 18) ; set PLLMUL -> 0x7 = „0b111“ = x 9
   ORR R1, #(0x2 :SHL: 15) ; set PLLSRC
   ORR R1, #(0x4 :SHL: 8) ; set PRE1
   STR R1, [R0, #CFGR_offset]

   ; set HSEBYP and HSEON
   LDR R1, [R0, #CR_offset]
   ORR R1, #(0x5 :SHL: 16)
   STR R1, [R0, #CR_offset]

   LDR R2, =(1 :SHL: 17) ; Mask of the HSERDY bit in RCC_CR
HSE_READY_LOOP ; is the HSE ready?
   LDR R1, [R0, #CR_offset]
   TST R1, R2
   IT EQ
   BEQ HSE_READY_LOOP ; HSE not ready -> jump back to the label HSE_READY_LOOP

   ; set PLLON
   LDR R1, [R0, #CR_offset]
   ORR R1, #(0x5 :SHL: 24)
   STR R1, [R0, #CR_offset]
```

```
   LDR R2, =(1 :SHL: 25) ; ;Mask of the PLLRDY bit in RCC_CR
PLL_READY_LOOP ; is the PLL ready?
   LDR R1, [R0, #CR_offset]
   TST R1, R2
   IT EQ
   BEQ PLL_READY_LOOP ; PLL not ready -> jump back to the label HSE_READY_LOOP

   ; select the PLL as system clock
   LDR R1, [R0, #CFGR_offset]
   ORR R1, #2
   STR R1, [R0, #CFGR_offset]

   ; enable clock to GPIOA
   LDR R1, [R0, #AHBENR_offset]
   ORR R1, #(1 :SHL: 17)
   STR R1, [R0, #AHBENR_offset]

   ; add two wait states to FLASH_ACR
   LDR R0, =FLASH
   LDR R1, [R0, #ACR_offset]
   ORR R1, #0x2
   STR R1, [R0, #ACR_offset]

   POP{R0, R1, R2, PC}
   ENDP ; end of procedure CLOCK_INIT

;--------------------------------------;
; configure PA5 as output
GPIO_INIT PROC
   PUSH{R0, R1, LR}
   LDR R0, =GPIOA
   LDR R1, [R0, #MODER_offset]
   ORR R1, #(1 :SHL: 10)
   STR R1, [R0, #MODER_offset]
   POP{R0, R1, PC}
   ENDP ; end of procedure GPIO_INIT
```

```
;--------------------------------------;
; wait R0 miliseconds
; input: R0 - number of miliseconds
WAIT_MS PROC
    PUSH{R0, R1, LR}
    LDR R1, =72000
    MUL R1, R0
WAITLOOP
    SUBS R1, #6
    NOP
    BNE WAITLOOP
    POP{R0, R1, PC}
    ENDP ; end of procedure WAIT_MS

;--------------------------------------;
CHANGE_LED_STATE PROC
    PUSH{R0, R1, R2, LR}
    LDR R0, =GPIOA ; R0 = GPIOA
    LDR R1, [R0, #ODR_offset] ; load value of R0 + ODR_offset to R1
    EOR R2, R1, #(0x1 :SHL: 5) ; R2 = R1 ^ (0x1 << 5)
    STR R2, [R0, #ODR_offset] ; save the R2 registry value to R0 + ODR_offset
    POP{R0, R1, R2, PC} ; load the original values of R0, R1, R2
    BX LR
    ENDP ; end of procedure SWITCH_LED_ON

;--------------------------------------;
    ; end of assembly function
    ALIGN
    END
```

12.20 Výsledné řešení pro kap. 8.1

```
//---------------------------------------//
//if 's' is received -> NUM_BYTES values read on the pin INPUT_PIN is saved to memory;
//count the number of stored 1/0 and the time that took one uint8_t number to save;

//---------------------------------------//
//libraries
#include "mbed.h"

//---------------------------------------//
//constants
#define INPUT_PIN PC_0
#define NUM_BYTES 49152 //SRAM on F303RE has 64 kB -> use just 48 kB

//---------------------------------------//
//init
Serial serial(SERIAL_TX, SERIAL_RX);
DigitalOut led(LED1);
DigitalIn input(INPUT_PIN, PullDown);
Timer timer;

//---------------------------------------//
//program
int main() {
    char sign;
    serial.printf("PRESS 's' TO START SAVING DATA.\n\r");
    while(1){
        if (serial.readable()){
            sign=serial.getc(); //read it...
            if (sign=='s'){ //is it 's'?
                serial.printf("'%c' RECEIVED -> START SAVING DATA\n\r", sign);

                uint8_t savedData[NUM_BYTES]; //we will store data in this array

                timer.reset();//measure time during data storage
                timer.start();
                for (uint32_t x=0; x<NUM_BYTES; x++){ //storing data
                    savedData[x] = input;
                }
                timer.stop();
```

```
            double time = timer.read_us(); //get measured time
            serial.printf("DATA SAVING FINISHED! IT TOOK %f us. It means %f ns for
saving ONE uint8_t\n\r", time, (double) (time/NUM_BYTES)*1000);

            //count the number of stored 1/0
            uint32_t num_ones = 0;
            uint32_t num_zeros = 0;
            uint32_t unspecified = 0;
            for (uint32_t x=0; x<(NUM_BYTES); x++){
               if (savedData[x] == 1){
                  num_ones++;
               } else if (savedData[x] == 0){
                  num_zeros++;
               } else {
                  unspecified++;
               }
            }
            serial.printf("NUMBER OF ONES: \t%d\tZEROS: \t%d\tUNSPECIFIED:
                  \t%d!\n\r", num_ones, num_zeros, unspecified);
         } else {
            serial.printf("Char '%c' received\n\r", sign);
         }
      }
   }
}
```

12.21 Výsledné řešení pro kap. 8.2

Soubor *main.cpp*:

```
/----------------------------------------//
//if 's' is received -> NUM_BYTES values read on the pin INPUT_PIN is saved to the
memory;
//count the number of stored 1/0 and the time that took one uint8_t number to save;
//the assembler file uses input arguments;
//----------------------------------------//
//libraries
#include "mbed.h"

//----------------------------------------//
//constants
#define INPUT_PIN PB_5 //works ONLY for pins PA/B/C 0-7
#define NUM_BYTES 49152 //SRAM on F303RE has 64 kB -> use just 48 kB
```

```
//peripherals
#define GPIO_A 0x48000000
#define GPIO_B 0x48000400
#define GPIO_C 0x48000800

//-------------------------------------//
//init
Serial serial(SERIAL_TX, SERIAL_RX);
DigitalIn input(INPUT_PIN, PullDown);
DigitalOut led(LED1);
Timer timer; //used to measuring computing time

//-------------------------------------//
//init extern functions
extern "C" void memory(uint32_t peripheral, uint8_t* address, uint32_t size);

//-------------------------------------//
//program
int main() {
    //flashing at the beginning
    for (int i = 0; i < 5*2; ++i) {
        led = !led;
        wait(0.05);
    }

    //INPUT_PIN recognition
    //0-15: PA_0 - PA_15;     16-31: PB_0 - PB_15;    32-47: PC_0 - PC_15
    uint32_t pin_num=INPUT_PIN%16;
    uint32_t peripheral_num;
    char peripheral_char;
    if (INPUT_PIN >= 32){
        peripheral_num = GPIO_C;
        peripheral_char = 67; //67 == "C"
    } else {
        peripheral_num = (INPUT_PIN >= 16) ? GPIO_B : GPIO_A;
        peripheral_char = (INPUT_PIN >= 16) ? 66 : 65; // 65 == "A", 66 == "B"
    }

    serial.printf("INPUT_PIN: P%c%d\n\r", peripheral_char, pin_num);

    char sign;
    serial.printf("PRESS 's' TO START SAVING DATA.\n\r");
```

```
while(1){
      if (serial.readable()){ //is something to read?
         sign=serial.getc();
         if (sign=='s'){
            serial.printf("'%c' RECEIVED -> START SAVING DATA\n\r", sign);

            uint8_t data[NUM_BYTES];

            timer.reset(); //measure time during data storage
            timer.start();
            memory(peripheral_num, &data[0], NUM_BYTES);
            timer.stop();

            double time = timer.read_us();
            serial.printf("DATA SAVING FINISHED! IT TOOK %f us. It means %f ns for
saving ONE uint8_t\n\r", time, (double) (time/NUM_BYTES)*1000);

            //count the number of stored 1/0
            uint32_t num_ones = 0;
            uint32_t num_zeros = 0;
            uint32_t unspecified = 0;
            uint32_t var;
            for (uint32_t x=0; x<NUM_BYTES; x++){
               var = data[x];
               var = var >> pin_num; //right shift
               var = var & 0x1;  //mask first bit
               if (var == 1){
                  num_ones++;
               } else if (var == 0){
                  num_zeros++;
               } else {
                  unspecified++;
               }
            }
            serial.printf("NUMBER OF ONES: \t%d\tZEROS: \t%d\tUNSPECIFIED:
                        \t%d!\n\r", num_ones, num_zeros, unspecified);
         } else {
            serial.printf("Char '%c' received\n\r", sign);
         }
      }
   }
}
```

Soubor *memory.s*:

```
;---------------------------------------;
;save NUM_BYTES values read on the pin INPUT_PIN to SRAM (at the address MY_ADDRESS)
;input arguments:
;  R0 = peripheral (address of GPIOA, GPIOB or GPIC),
;  R1 = MY_ADDRESS (address to SRAM - see main.cpp),
;  R2 = NUM_BYTES

;---------------------------------------;
;constants
GPIO_IDR_Offset EQU 0x10

;---------------------------------------;
;program
   AREA asm_func, CODE, READONLY
   EXPORT memory

memory
   PUSH{LR}

LOOP
   LDRB R3, [R0, #GPIO_IDR_Offset] ; read just the first byte of the GPIOx_IDR
   STRB R3, [R1], #1 ;save this byte to the R1 register (MY_ADRESS)
                     ;and auto-increment
   SUBS R2, #1
   BNE LOOP

   POP{PC}
   ALIGN
   END
```

12.22 Výsledné řešení pro kap. 8.3

Soubor *main.cpp*:

```
//---------------------------------------//
//if change on TRIGGER_PIN occurs, start saving NUM_BYTES values on the pin INPUT_PIN
//to the memory at the address &data[0] (see below);
//count the number of stored 1/0 and send result to PC;
//pin_recognition converted to the function;
```

```
//---------------------------------------//
//libraries
#include "mbed.h"

//---------------------------------------//
//constants (constants INPUT_PIN, NUM_BYTES and MY_ADDRESS can be set)
#define INPUT_PIN PB_7 //works ONLY for pins PA/B/C 0-7
#define TRIGGER_PIN PA_10 //works for all pins on the port GPIOA/B/C
#define NUM_BYTES 49152 //SRAM on F303RE has 64 kB -> use just 48 kB

//---------------------------------------//
//init
Serial serial(SERIAL_TX, SERIAL_RX);
DigitalIn input(INPUT_PIN, PullDown);
DigitalIn trigger(TRIGGER_PIN, PullDown);

//---------------------------------------//
//init functions
extern "C" void memory(uint32_t *pins_info, uint8_t* address, uint32_t size);
void pin_recognition(uint32_t *peripheral, uint8_t *peripheral_char, uint8_t *pin_num,
uint32_t pin);

//---------------------------------------//
//program
int main() {
    //INPUT_PIN and TRIGGER_PIN recognition
    uint8_t input_num, trig_num, input_char, trig_char;
    uint32_t input_peripheral, trig_peripheral;
    pin_recognition(&input_peripheral, &input_char, &input_num, INPUT_PIN);
    pin_recognition(&trig_peripheral, &trig_char, &trig_num, TRIGGER_PIN);

    serial.printf("Data will be stored as soon as the logic value on "
            "the trigger pin changes.\n\r");
    serial.printf("INPUT_PIN:\tP%c%d\n\rTRIGGER_PIN:\tP%c%d\n\r",
            input_char, input_num, trig_char, trig_num);

    uint8_t data[NUM_BYTES];
    uint32_t pins_info[3];
    pins_info[0] = input_peripheral;
    pins_info[1] = trig_peripheral;
    pins_info[2] = trig_num;

    memory(&pins_info[0], &data[0], NUM_BYTES);
```

```
    //count the number of stored 1/0
    uint32_t num_ones = 0;
    uint32_t num_zeros = 0;
    uint32_t unspecified = 0;
    uint32_t var;
    for (uint32_t x=0; x<NUM_BYTES; x++){
        var = data[x];
        var = var >> input_num; //right shift
        var = var & 0x1; //mask
        if (var == 1){
            num_ones++;
        } else if (var == 0){
            num_zeros++;
        } else {
            unspecified++;
        }
    }
    serial.printf("NUMBER OF ONES:\t%d\tZEROS:\t%d\tUNSPECIFIED:\t%d!\n\r",
num_ones, num_zeros, unspecified);
}

//---------------------------------------//
void pin_recognition(uint32_t *peripheral, uint8_t *peripheral_char, uint8_t *pin_num,
uint32_t pin){
    //0-15: PA_0 - PA_15;     16-31: PB_0 - PB_15;    32-47: PC_0 - PC_15
    *pin_num = pin%16;
    if (pin >= 32){
        *peripheral = 0x48000800; //GPIOC
        *peripheral_char = 67; //67 == "C"
    } else if (pin >= 16){
        *peripheral = 0x48000400; //GPIOB
        *peripheral_char = 66; //66 == "B"
    } else {
        *peripheral = 0x48000000; //GPIOA
        *peripheral_char = 65; //65 == "A"
    }
}
```

Soubor *memory.s*:

```
//-------------------------------------//
//save NUM_BYTES values read on the pin INPUT_PIN to SRAM (at the address MY_ADDRESS);
//after the logic value on the pin TRIGGER_PIN has changed
//input arguments:
//      R0 = pins_info_ADDRESS (input_peripheral (GPIOA/B/C), trig_peripheral (GPIOA/B/C), trig_num),
//  R1 = MY_ADDRESS (address to SRAM - see main.cpp),
//  R2 = NUM_BYTES,

//-------------------------------------//
//constants
GPIO_IDR_Offset EQU 0x10

//-------------------------------------//
   AREA asm_func, CODE, READONLY
   EXPORT memory
memory
   PUSH {R4, R5, R6, R7, LR}
   LDR R3, [R0], #4 //R3 = input_peripheral
   LDR R4, [R0], #4 //R4 = trig_peripheral
   LDR R5, [R0] //R5 = trig_num

   LDR R0, [R4, #GPIO_IDR_Offset]
   MOV R6, #1 //R6 = 1
   LSL R6, R6, R5 //R6 = 1 << trig_num -> R6 = mask of the trigger pin
   AND R0, R0, R6 //R0 = the origin logic value of the trigger pin
MAIN_LOOP
   LDR R7, [R4, #GPIO_IDR_Offset] //read TRIGGER_PIN again
   AND R7, R7, R6 //R7 = new logic value of the trigger pin
   CMP R7, R0 //compare them
   BEQ MAIN_LOOP //R7==R0 -> nothing to do -> return to the label MAIN_LOOP

LOOP
   LDRB R0, [R3, #GPIO_IDR_Offset] //read just the first byte of the GPIOx_IDR
   STRB R0, [R1], #1 //save it to R1 (MY_ADRESS) and auto-increment
   SUBS R2, #1
   BNE LOOP

   POP {R4, R5, R6, R7, PC}
   ALIGN
   END
```