Bakalárska práca

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra měření

Optimální využití mbed IDE v laboratorní výuce

Lukáš Bielesch
Kybernetika a robotika

Máj 2019
Vedúci práce: doc. Ing. Jan Fischer, CSc.

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Bielesch** Jméno: **Lukáš** Osobní číslo: **465960**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Kybernetika a robotika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Optimální využití mbed IDE v laboratorní výuce

Název bakalářské práce anglicky:

The Optimal Use of the mbed IDE in Laboratory Teaching

Pokyny pro vypracování:

Analyzujte možnosti optimálního využití mbed IDE v laboratorní výuce průmyslové elektroniky a senzorů. Vytvořte podrobné popisy jednotlivých tříd mbed s vysvětlením jejich vlastní funkce i využití obvodových prostředků mikrořadiče včetně podrobného popisu pro uživatele. Vypracujte metodiku ladění programu vytvořeného pomocí mbed včetně výpisu ladicích informací tak, aby je bylo možno využít při ladění studentských projektů. Navrhněte metodu ladění po úsecích s vkládáním testovacích bodů a signalizací pomocí LED i pomocí výpisů pomocných informací prostřednictvím kanálu UART. Doplňte další funkce pro využití čítačů i pro doplňkovou konfiguraci periferií mikrořadiče. Při návrhu se orientujte na použití mikrořadičů řady STM32Fxxx, především na STM32F042 a STM32F303.
Navrhněte metodu spolupráce programu s moduly v jazyce assembler.

Seznam doporučené literatury:

[1] Yiu, J.: The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ processors
[2] Toulson, R. & Wilmshurst, T.: Fast and Effective Embedded Systems Design - Applying the ARM mbed; 2016
[3] STMicroelectronics: RM0091 STM32F0x2 Reference manual

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: **31.01.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce:
do konce letního semestru 2019/2020

doc. Ing. Jan Fischer, CSc. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. 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

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval samostatne a že som uviedol všetky použité informačné zdroje v súlade s Metodickým pokynom o dodržovaní etických princípov pri príprave vysokoškolských záverečných prác. Táto práca vznikla v laboratóriu videometrie na katedre meraní na FEL - ČVUT v Prahe pod vedením
doc. Ing. Jana Fischera, CSc.

V Prahe dňa 24. 5. 2019 .....

Poďakovanie

Hlavné poďakovanie patrí vedúcemu tejto práce doc. Ing. Janu Fischerovi, CSc. za jeho trpezlivosť, ochotu a čas, ktorý mi venoval počas tvorby tejto bakalárskej práce. Ďalej by som sa chcel poďakovať mojej rodine a priateľom za podporu počas celého štúdia.

Abstrakt

Táto práca sa zaoberá optimálnym využitím vývojového prostredia mbed v laboratórnej výuke a zamieriava sa špeciálne na mikrokontroléry STM32F042F6P6 a NUCLEO-F303RE. Sú v nej popísané základné triedy prostredia a funkcionalita tried je demonštrovaná na príkladových programoch. V rámci práce je vytvorená sada nástrojov umožňujúca jednoduché ladenie programov s využitím LED a tlačidla alebo pomocou výpisov prostredníctvom kanálu UART. Práca sa zaoberá aj pokročilým použitím prostredia, volaním funkcií písaných v jazyku assembler a manuálnou konfiguráciou periférií mikrokontroléra.

Kľúčové slová: STM32; arm mbed; NUCLEO-F303RE; STM32F042F6P6; príručka mbed IDE; ladiace nástroje; konfigurácia periférií v mbed IDE

Abstract

The aim of this bachelor thesis is optimal usage of mbed IDE especially with the use of microcontrollers STM32F042F6P6 and NUCLEO-F303RE. The main features of core classes are described and their functionality is demonstrated on example programs. Different debugging tools are created within the work. This thesis also deals with advanced possibilities of mbed IDE such as calling functions in assembly language or direct configuration of peripherals.

Keywords: STM32; arm mbed; NUCLEO-F303RE; STM32F042F6P6; mbed IDE manual; debugging tools; peripheral configuration in mbed IDE

Title translation: The Optimal Use of the mbed IDE in Laboratory Teaching

Obsah

Obrázky

Tabuľky

Kapitola 1
Úvod

Témou tejto práce je optimálne využitie vývojového prostredia mbed pri programovaní mikrokontrolérov s jadrom ARM v laboratórnej výuke. Motiváciou tohto zadania je skutočnosť, že po ročnom používaní v predmete LPE sa ukázalo, že aj keď je mbed vhodný nástroj na prvotné zoznámenie sa s programovaním mikrokontrolérov, prostrediu chýba systematický popis a súčasne sa prejavili nedostatky a chyby v implementácii jednotlivých tried prostredia.

Prostredie mbed využíva objektovo-orientovaný prístup pre jednoduché programovanie mikrokontrolérov. Okrem toho však ponúka aj pokročilejšie metódy, volanie funkcií písaných v assembleri a priamu konfiguráciu periférií procesora. V kombinácii so vstavanými mbed triedami môže použitie týchto metód prispieť k lepšiemu pochopeniu štruktúry procesoru a umožniť plynulý prechod na nizkoúrovňové programovanie mikrokontrolérov. Problémom aj v tomto prípade sú chýbajúce materiály.

Optimálne využitie prostredia mbed teda znamená, že študenti majú k dispozícii všetky materiály potrebné pri vytváraní programu. O prípadných chybách a obmedzeniach v implementácii sú informovaní, aby nestrácali čas s hľadaním chýb, ktoré nespôsobili. Pre jednoduché ladenie vytvorených programov majú k dispozícii ladiace nástroje a sú oboznámení aj o pokročilých možnostiach, ktoré mbed ponúka.

Cieľom bakalárskej práce je vytvoriť materiály a vyvinúť nástroje, ktoré umožnia efektívne využitie vývojového prostredia mbed.

Kapitola 2
Rozbor a stanovenie cieľov práce

Ako už bolo v úvode spomenuté, mbed je vývojové prostredie na programovanie mikrokontrolérov. Tento vývoj prebieha vo webovom prostredí, a preto nie je potrebné inštalovať žiadne knižnice, kompilátory ani sa viazať na určitý operačný systém. Používateľ si na začiatku založí účet, na ktorom má neskôr prístup ku všetkým vytvoreným programom. Vďaka tejto vlastnosti môže študent na programe pracovať kdekoľvek, kde je internetové pripojenie.

Vývojové prostredie používa C++ kompilátor, konkrétne ARM Real View Development Suite. Vďaka tomu je možné okrem jazyka C využiť aj objektový prístup jazyka C++. V mbede je definovaná sada tried, ktorá umožňuje jednoduché použitie periférií mikrokontroléra. Počiatočná konfigurácia ale vyžaduje pokročilú znalosť hardvéra daného mikrokontroléra a býva časovo náročná. Hlavnou výhodou objektového prítupu je, že konfigurácia periférie sa prevedie automaticky pri vytváraní objektu prislúchajúcej triedy. Vďaka tomu je mbed vhodný aj pre študentov bez týchto pokročilých znalostí. Ďalšou výhodou je aj jednotný prístup k rôznym procesorom, takže rovnaký program je možné použiť na rôznych zariadeniach. V prípade vývojových prostredí využívajúcich iba jazyk C to nie je možné.

2.1 Chýbajúca dokumentácia prostredia

V procese výuky sa však vyskytli problémy, ktoré bránia optimálnemu využitiu mbed IDE. V súčasnosti neexistujú materiály popisujúce kompletnú funkcionalitu tried prostredia. Na oficiálnej stránke sú k dispozícii iba všeobecné popisy funcií, takže programátor nepozná konkrétnu konfiguráciu periférií procesora. Jednotlivé mikrokontrléry sa ale odlišujú rôznou funkcionalitou periférií, preto aj výsledná konfigurácia môže byť rozdielná pri použití rovnakej triedy. Pri pokročilejšom použití periférie môže pri neznalosti jej konfigurácie dôjsť k chybnému vykonávaniu programu.

Pre získanie úplnej informácie je nutné nahliadnuť do implementačných súborov triedy konkrétneho zariadenia. Druhou možnosťou je kontrolovať konfiguračné registre v mikrokontroléri počas behu programu. To však odporuje prvotnej motivácii, ktorou je jednoduché vytváranie programov.

Cieľom práce je preto vytvoriť príručku s podrobnými popismi jednotlivých tried a vysvetliť ich vlastnú funkcionalitu.

2.2 Chyby v implemenácii tried

Mbed nie je profesionálny nástroj, v jeho implementácii sú rôzne chyby, ktoré môžu spôsobovať problémy pri vytváraní programu. Ku každému procesoru, ktorý je možné v mbede programovať, existuje diagram pinov. V ňom je pri každom pine zoznam jeho možných konfigurácií. Počas výuky predmetu LPE sa zistilo, že piny v niektorých konfiguráciách nie je možné použiť napriek tomu, že v diagrame boli zmienené. Ďalej sa ukázalo, že niektoré funkcie nepracujú tak, ako by mali a pri ich použití dochádza k chybe programu.

Ďalším cieľom práce je preto overiť správnosť implementácie tried pre mikrokontroléry používané pri výuke. Nájdené chyby budú zdokumentované a chybné diagramy mikrokontrolérov budú opravené.

2.3 Absencia ladiacich nástrojov

Pri vytváraní programov často dochádza k chybám. Pre ich odstránenie sa používajú rôzne debugovacie metódy. Profesionálne vývojové prostredia majú implementované nástroje, ktoré umožňujú debugovať mikrokontrolér v reálnom čase. V prostredí mbed ladiace nástroje nie sú dostupné, preto jedinou možnosťou debugovania je metóda pokus-omyl alebo výpis do konzoly.

Z tohto dôvodu je potrebné vytvoriť metodiku ladenia programov s využitím periférií procesora. Jednotlivé metódy budú implementované v triedach, ktoré umožnia ich jednoduché použitie pri ladení študenských projektov. Triedy budú využívať rôznorodé prístupy, k ladeniu bude možné použiť ich v ľubovoľných mikrokontroléroch, ktoré podporuje mbed.

2.4 Použitie pokročilých nástrojov

V prostredí mbed je možné volať aj funkcie písané v assembleri. Pri programovaní na najnižšej úrovni býva problém s počiatočým nastavením periférií. Kombinácia objektovo-orientovaného prístupu s volaním assemblerovských funkcií môže v mbede tento problém odstrániť a umožniť tak plynulý prechod na nizkoúrovňové programovanie mikrokontrolérov. Na oficiálnych stránkach však tieto postupy nie sú dostatočne popísané.

Ďalším cieľom je preto zdokumentovať postup vytvárania funkcií písaných v assembleri. Vytvorené programy budú využívať inštrukčnú sadu pre procesory s jadrom Cortex-M4. Budú priblížovať prácu s registrami pre lepšie pochopenie štruktúry procesoru.

Pomocou tried mbed je možné použiť periférie len v určitej konfigurácii. Okrem nich je však možné použiť aj priamy prístup ku konfiguračným registrom s využitím štruktúr definovaných v prostredí.

Cieľom je pomocou priameho prístupu k registrom doplniť funkcionalitu periférií, ktorú triedy mbed neponúkajú. Súčasťou práce bude tiež program pre radič STM32F042F6P6, ktorý umožní využite čítača v konfigurácii počítania pulzov. Ďalej budú vytvorené funkcie pre konfiguráciu HSE a HSI48 ako zdroja hodinového signálu mikrokontroléra.

Kapitola 3
Použité mikrokontroléry

V tejto príručke budeme pracovať predovšetkým s dvomi mikrokontrolérmi NUCLEO-F303RE a STM32F042F6P6 V jednotlivých úlohách budeme používať piny mikrokontroléra v rôznych konfiguráciách. Voľne dostupné diagramy vývojových kitov NUCLEO sú chybné, predovšetkým pri funkcionalite PWM. Po dôkladnej kontrole konfiguračných registrov boli tieto chyby odstránené a na nasledujúcich obrázkoch sú opravené diagramy pre mikrokontroléry mikrokontrolérov STM32F042F6P6, NUCLEO-F303RE a NUCLEO-F042K6. Pri každom pine je zoznam jeho možných konfigurácií, pričom v jednom programe môže byť pin iba v jednej konfigurácii (napr. DigitalOut). Niektoré periférie (napr. Serial2) vyžadujú použitie viacerých pinov naraz (Serial1 TX, Serial1 RX), iné požadujú iba 1(napr. AnalogIn).

DigitalOut, DigitalIn, InterruptIn

Piny konfigurovateľné ako digitálny výstup, digitálny vstup, alebo digitálny vstup s možnosťou hardverového prerušenia, sú označené svetlomodrou farbou.

AnalogIn

Piny, ktoré je možné konfigurovať ako analógový vstup, sú označené svetlohnedou farbou s popisom AnalogIn.

AnalogOut

Piny s možnosťou konfigurácie ako analógový výstup, sú označené svetlohnedou farbou s popisom AnalogOut. Môžeme si všimnúť, že mikrokontrolér STM32F042F6P6 nemá ani jeden pin s takouto funkcionalitou, na aproximovanie analógovej hodnoty napätia na výstupe sa používajú piny v konfigurácii PwmOut.

PwmOut

Piny, ktoré môžu byť konfigurované ako generátory PWM, sú označené fialovou farbou. Generovanie pulzne šírkovo modulovaného signálu funguje za pomoci timerov. Čísla x a y v názve PWM x/y znamenajú, že kanál y timera x zbezpečuje generovanie PWM. Platí tu však obmedzenie, že na všetkých kanáloch jedného timera musí mať PWM signál rovnakú periódu. Navyše, ak sa v názve zhodujú timer aj číslo kanálu a jeden z kanálov je negovaný, súčasné generovanie PWM nie je možné ani na rovnakej frekvencii.

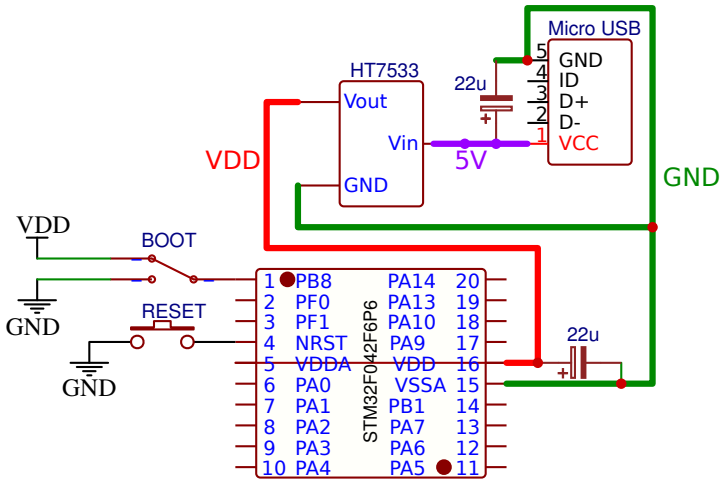
Kompatibilita vývodov NUCLEO-F303RE a NUCLEO-F042K6 s Arduinom

Vývojová doska NUCLEO-F303RE má piny vyvedené rovnako ako mikrokontrolér Arduino UNO a NUCLEO-F042K6 s Arduinom Nano, aby rôzne nástavce určené pre Arduino mohli byť použité aj pre NUCLEO a naopak. Z tohto dôvodu môžu byť piny NUCLEA pomenované značením, ktoré používa Arduino (tmavozelená farba). V tejto príručke Arduino značenie používať nebudeme s výnimkou inicializácie vstavanej LED diódy a vstavaného tlačidla na vývojovej doske NUCLEO-F303RE.

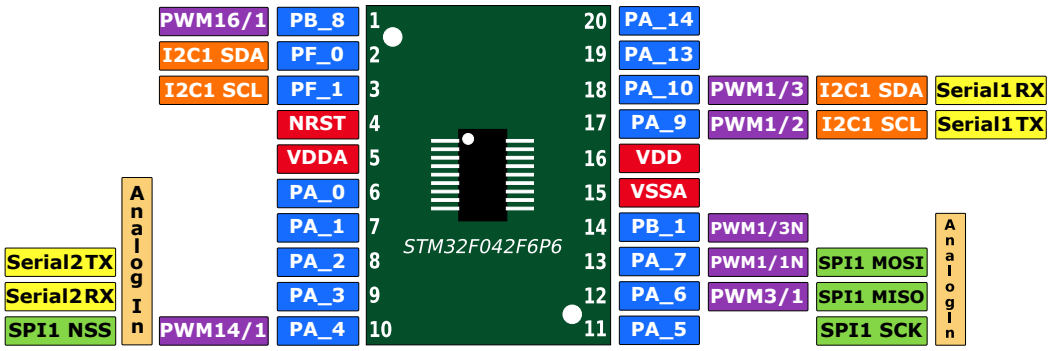
3.1 STM32F042F6P6

Mikrokontrolér STM32F042F6P6 sa pri výuke predmetu LPE používa v zapojení podľa obrázka 3.1. Dôvodom jeho používania je jeho cenová dostupnosť a fakt, že je možné ho

programovať v prostredí mbed aj keď nie je priamo podporovaný. Vývojový kit NUCLEO-F042K6, ktorý je podorovaný v Mbed IDE, obsahuje procesor STM32F042K6P6, ktorý je takmer identický s STM32F042F6P6. Jediný rozdiel medzi nimi je, že STM32F042F6P6 má vyvedených 20 pinov, zatiaľ čo STM32F042K6P6 má až 32 pinov. Problém nastane iba v prípade USB komunikácie, ktorý je takisto možné odstrániť premapovaním pinov(kapitola 4.9.3). V ostatných prípadoch je možné bez problémov vytvárať v Mbede program pre NUCLEO-F042K6 a použiť ho pre STM32F042F6P6. Po vytvorení sa program skompiluje a uloží sa ako *.bin* súbor. Program je možné do mikrokontroléra nahrať pomocou programu Cube Programmer po prepnutí posuvného bootovacieho tlačidla do bootovacieho režimu(na zem). Mikrokontrolér je možné použiť aj ako meracie zariadenie alebo osciloskop, firmvér a PC aplikácie je možné stiahnúť z [1].



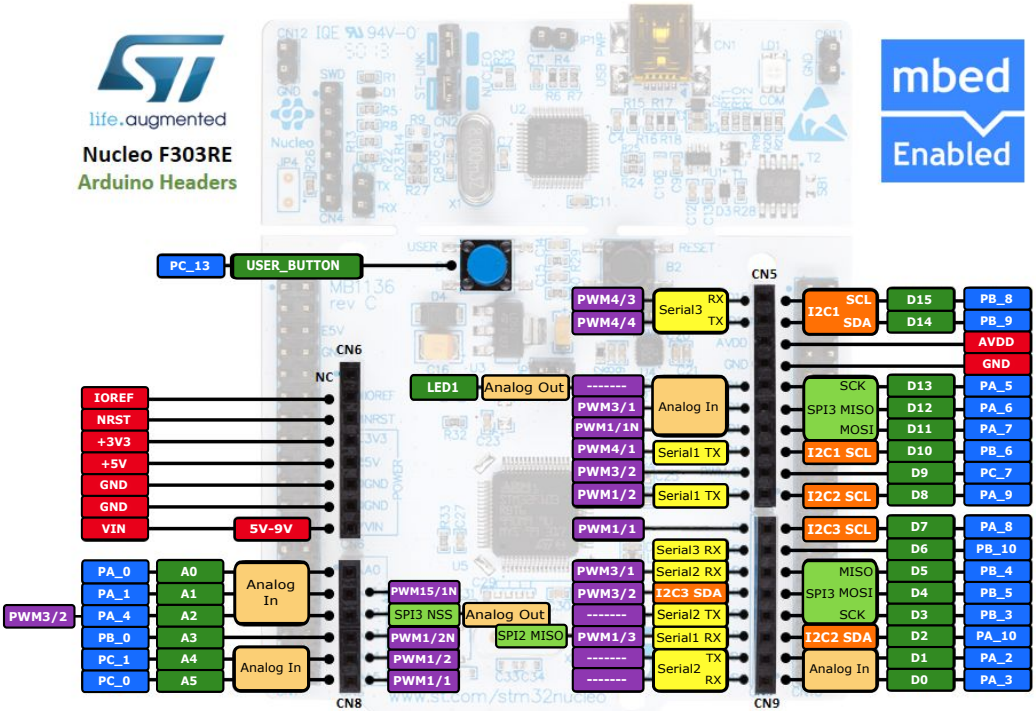
Obrázok 3.1. Zapojenie mikrokontroléru STM32F042F6P6 podľa [1].



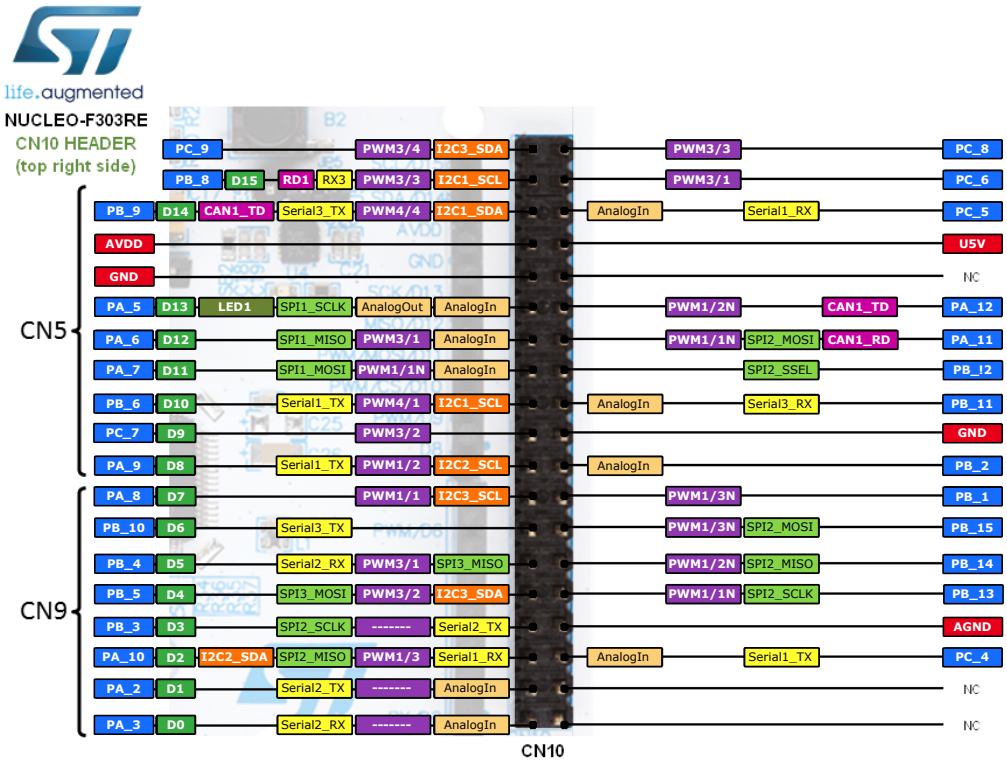
Obrázok 3.2. Pinout diagram procesoru STM32F042F6P6

3.2 NUCLEO-F303RE

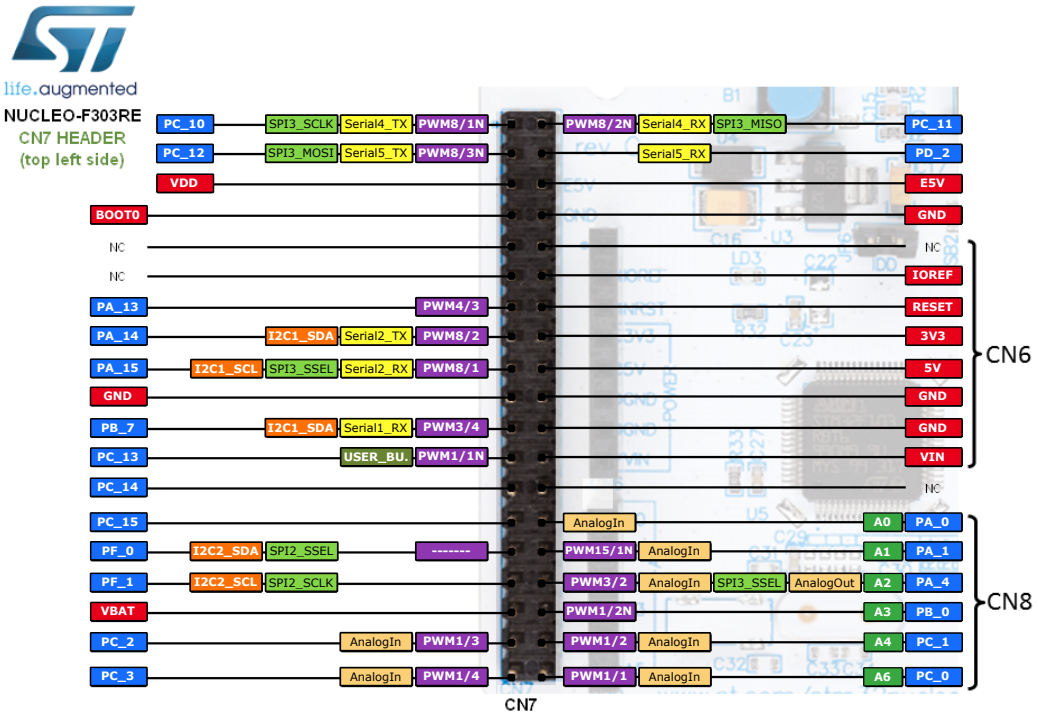
Mikrokontrolér NUCLEO-F303RE sa používa pri tvorbe zložitejších projektov v predmete LPE, pretože má viac vyvedených pinov a periférií v porovnaní s STM32F042F6P6. Vďaka tomu, že vývojový kit obsahuje ST-Link, sa po pripojení do PC správa ako jednotka USB flash, preto je možné priamo *.bin* súbor doň skopírovať bez použitia akéhokoľvek programu. Mikrokontrolér je možné použiť aj ako univerzálne meracie zariadenie a osciloskop(LEO), firmvér a PC aplikáciu je možné stiahnúť z [1].



Obrázok 3.3. Upravený pinout strednej časti pre NUCLEO-F303RE z [2].



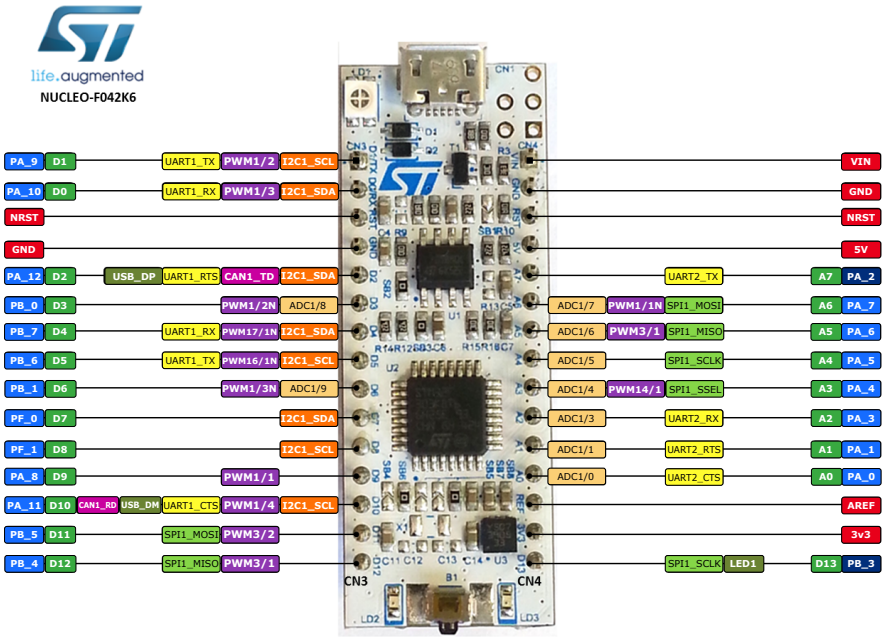
Obrázok 3.4. Upravený pinout pravej časti pre NUCLEO-F303RE z [2].



Obrázok 3.5. Upravený pinout ľavej časti pre NUCLEO-F303RE z [2].

3.3 NUCLEO-F042K6

Pre úplnú informáciu bol upravený aj pinout diagram vývojového kitu NUCLEO-F042K6.



Obrázok 3.6. Upravený pinout pre NUCLEO-F042K6 z [3].

Kapitola 4
Príručka vývojového prostredia mbed

Keďže má vývojové prostredie mbed C++ kompilátor, je možné plnohodnotne používať jazyk C a navyše využívať objektovo-orientovaný prístup. V prostredí je vytvorená sada tried, ktoré je možné použiť na konfiguráciu a používanie periférií mikrokontroléra. Táto sada tried je do každej zložky programu automaticky pridaná ako knižnica mbed. Na začiatku programu je nutné ju doňho zahrnúť pomocou

```
#include "mbed.h"
```

Rovnakým spôsobom je možné používať aj externe vytvorené triedy, ktoré treba do programovej zložky najskôr importovať. Pri importovaní knižníc sa nesmie povoliť ich aktualizácia, pretože knižnice boli vytvorené v nižších verziách knižnice mbed a nemusia byť kompatibilné s najnovšou verziou. Rovnako ako pri knižnici mbed, aj externé knižnice sa do programu najčastejšie pridávajú pomocou

```
#include "hlavickovy_subor.h"
```

kde *hlavickovy_subor* je názov súboru s príponou *.h*, ktorý sa nachádza v danej knižnici. Samotná konfigurácia periférie sa prevedie už pri vytvorení objektu danej triedy. Vo väčšine prípadov je potrebné ako parameter zadať pin (resp. piny), na ktorom(resp. ktorých) sa bude daná periféria používať.

```
nazov_triedy nazov_objektu(vstupne_parametre);
```

Objekty vytvárame pred funkciou *main*(), pretože potom majú globálnu platnosť a môžeme k nim pristupovať v ľubovoľnej časti programu. Každá trieda má funkcie (metódy), pomocou ktorých sa nastavujú a menia parametre objektu a môžu sa volať kedykoľvek po vytvorení objektu danej triedy. K funkciám danej triedy pristupujeme pomocou bodkovej notácie, teda

```
navratova_hodnota = názov_objektu.funkcia(vstupne_parametre);
```

Pri volaní funkcie je nutné zadať všetky povinné argumenty. Voliteľné argumenty nemusia byť použité, v takom prípade sa použije predvolená hodnota daného parametra.

4.1 Trieda Wait

Funkcie triedy wait slúžia ku generácii časového oneskorenia s maximálnym rozlíšením na mikrosekundy.

4.1.1 wait

Funkcia *wait()* generuje časové oneskorenie v sekundách s rozlíšením na mikrosekundy. Parametrom funkcie je premenná typu float, časové oneskorenie v sekundách.

```
wait(float time_in_s); // parametrom funkcie wait je desatinne cislo
wait(1);               // cakanie 1 sekundu
wait(0.345);           // cakanie 345 milisekund
```

V prípade, ak ako parameter funkcie zadáme desatinné číslo s dlhším desatinným rozvojom ako 6 miest(rozlíšenie na us), číslo sa nezaokrúhli, ale ďalší desatinný rozvoj sa neberie do úvahy, preto nasledujúce funkcie budú čakať rovnakú dobu.

```
float time1 = 0.123456;  // cas v sekundach s rozlisenim na us
wait(time1);             // funkcia caka 123456us
float time2 = 0.1234569; // cas v sekundach s vyssim rozlisenim ako us
wait(time2);             /* posledne cislo rozvoja sa neberie do uvahy,
                            funkcia caka  123456us */
float time3 = 0.1234561; // cas v sekundach s vyssim rozlisenim ako us
wait(time3);             /* posledne cislo rozvoja sa neberie do uvahy,
                            funkcia caka  123456us */
```

4.1.2 wait ms

Funkcia *wait_ms*() generuje oneskorenie v milisekundách. Parametrom funkcie je premenná typu int, časové oneskorenie v milisekundách. Výsledné časové oneskorenie môže byť o 1ms menšie ako zadané, preto sa pre malé oneskorenia odporúča pužiť funkciu *wait*(), lebo 1ms by spôsobila veľkú relatívnu chybu. Funkcia *wait_ms*() sa nesmie používať vo funkcii obsluhujúcej prerušenie.

```
wait_ms(int time_in_ms); // parametrom funkcie wait_ms je cele cislo
wait(756);               // cakanie 756 milisekund
wait(0.001); // pre kratke useky (jednotky ms) je lepsie pouzit wait
```

4.1.3 wait us

Funkcia *wait_us*() generuje oneskorenie v mikrosekundách. Parametrom funkcie je premenná typu int, časové oneskorenie v mikrosekundách. Výsledné časové oneskorenie môže byť o 1us menšie ako zadané, preto sa pre malé oneskorenia odporúča pužiť funkciu *wait*(), lebo 1us by spôsobila veľkú relatívnu chybu. Funkcia *wait_us*() sa nesmie používať vo funkcii obsluhujúcej prerušenie.

```
wait_ms(int time_in_us); // parametrom funkcie wait_us je cele cislo
wait(1250);               // cakanie 1250 mikrosekund
wait(0.000001); // pre kratke useky (jednotky us) je lepsie pouzit wait
```

4.2 Trieda DigitalOut

Funkcie triedy DigitalOut používame pri konfigurácii pinu ako výstupného s dvomi možnými logickými úrovňami napätia na výstupe a na zmenu logickej úrovne napätia na výstupe. Logickej úrovni 1 odpovedá kladné napájacie napätie procesora (V_{DD}) a logickej úrovni 0 odpovedá napätie 0V(GND). Výstup pinu ale nesmie byť zaťažovaný väčším odberom prúdu ako 25mA, pretože by mohlo dôjsť k poškodeniu procesoru [4]. Pre aplikácie s vysokým odberom prúdu je potrebné použiť zapojenie s tranzistorom alebo operačným zosilovačom.

4.2.1 Konfigurácia pinu ako výstupného

Pin mikrokontroléru (napr. PA_5) si nakonfigurujeme ako digitálny výstup tak, že si vytvoríme objekt triedy DigitalOut s nejakým názvom (napr. led) a priradíme ho k tomuto pinu. Prvým parametrom je názov pinu, ktorý ideme konfigurovať.

```
DigitalOut led(PA_5); // vytvorenie objektu triedy DigitalOut s nazvom led
```

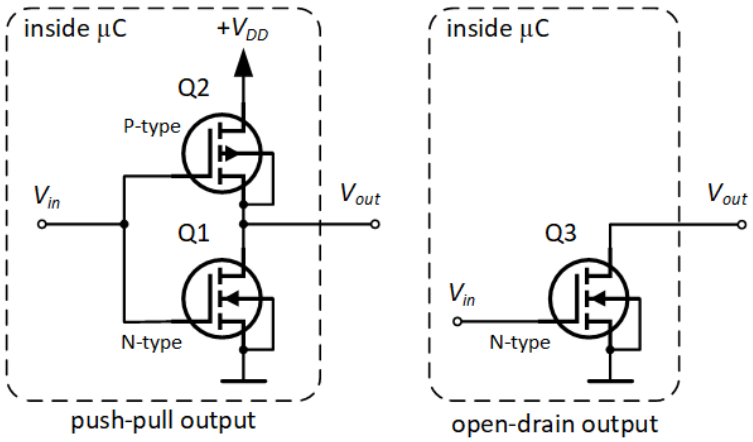
Druhý, voliteľný parameter je premenná typu int, logická hodnota na výstupe pinu po vytvorení objektu. Ak sa parameter nepoužije, na výstup sa zapíše log.0.

```
DigitalOut  led(PA_5, 1) // zapis log.1 pri vytvarani objektu;
```

Výstupný mód pinu

V procesoroch STM32 je možné použiť jeden z dvoch výstupných módov pinu:

- push pull: Zapojenie obsahuje 2 MOSFET tranzistory, ak je vstup v log.1, vrchný N-MOSFET aktívne tlačí výstup pinu do úrovne log.1 a ak je vstup log.0, spodný P-MOSFET aktívne tlačí výstup pinu do 0.
- open drain: Zapojenie obsahuje iba jeden MOSFET tranzistor, takže v prípade log.0 je MOSFET zopnutý a aktívne tlačí výstup do 0. V prípade log.1 nie je MOSFET zopnutý a na výstupe nedefinovaná hodnota napätia. Z tohto dôvodu sa väčšinou používa zapojenie s vnútorným pull-up rezistorom, ktorý v prípade nezopnutého MOSFETU zapezpečí na výstupe pinu napäťovú úroveň log.1.



Obrázok 4.1. Zapojenie pinu v móde push-pull a open drain dostupné z [5].

V triede DigitalOut je mód pinu pevne daný: Output push pull. V prípade, že chceme použiť pin digitálny výstup v móde Open drain, použijeme triedu DigitalInOut (kapitola 4.4).

4.2.2 is connected

Funkcia *is_connected*() slúži na kontrolu správnosti konfigurácie pinu ako digitálneho výstupu. Funkcia nemá vstupné parametre a jej návratová hodnota je premenná typu int s hodnotou 0 alebo 1:

- 1, ak bol pin správne nakonfigurovaný
- 0 v prípade chyby konfigurácie.

Táto funkcia sa väčšinou volá na začiatku funkcie *main*() a v prípade neúspešnej inicializácie neúspešnej inicializácie sa program ukončí alebo sa oznámi chyba.

```
if ( led.is_connected() == 0 ){
    return; // ak sa pin nenakonfiguroval spravne, program sa ukonci
}
```

Funkcia ale nerieši problém, ak viacnásobne konfigurujeme jeden pin, preto sa táto funkcia v praxi veľmi nepoužíva.

```
...
DigitalOut out(PA_4);
DigitalIn in(PA_4); //viacnasobna konfiguracia PA5
int main{
    if ( out.is_connected() == 0 ){
     return; // program sa neukonci
        }
}
```

4.2.3 write

Po vytvorení objektu je logická hodnota napätia na výstupe brány predvolene log.0. Túto hodnotu je možné meniť volaním funkcie *write*(). Vstupným parametrom funkcie je premenná typu int a podľa jej hodnoty je na výstupe brány napätie s logickou úrovňou log.1 alebo log.0.

Zápis log.1

Zápis logickej 1 sa môže previesť pomocou nenulovej konštanty:

```
  led.write(1);
```

alebo sa použije premenná typu int s hodnotou rozdielnou od 0.

```
int state = 1;
led.write(state);
```

Pre zápis log.1 na výstup brány sa väčšinou premennej priradí hodnota 1, ale rovnaký výsledok sa docieli, ak sa premennej priradí nenulová hodnota, napríklad -8.

```
int state = -8;
led.write(state); //led sa rozsvieti
```

Zápis log.0

Zápis logickej 0 sa môže previesť pomocou konštanty:

```
  led.write(0);
```

alebo sa použije premenná typu int s hodnotou 0.

```
int state = 0;
led = state;
```

Skrátený zápis funkcie write

V triede DigitalOut je operátor "=", ktorý priraďuje hodnotu danému objektu, preťažený a namapovaný na funkciu write, teda miesto *led.write(state);* sa častejšie používa skrátený zápis:

```
led = state; //state je premenna typu int
```

4.2.4 read

Funkcia *read*() slúži na čítanie logickej hodnoty napätia na výstupe brány. Funkcia nečíta logickú hodnotu na vstupe, ale na výstupe pinu. Tento rozdiel lepšie pochopíme pri práci s registrami procesora (kapitola 10), pretože hodnota na vstupe a výstupe sa číta z rôznych registrov. Funkcia *read*() vráti premennú typu int a jej hodnota je:

- 1, ak je na výstupe brány log.1
- 0, ak je na výstupe brány log.0

```
int state = led.read();
```

Funkciu použijeme napríklad, ak chceme invertovať logickú hodnotu napätia na výstupe pinu (kapitola 4.2.5).

Skrátený zápis funkcie read

V triede DigitalOut je operátor "=", ktorý priraďuje hodnotu objektu premennej, preťažený a namapovaný na funkciu *read*(), teda miesto *int state = led.read();* môžeme použiť skrátený zápis:

```
int state = led;
```

4.2.5 Invertovanie logickej hodnoty na výstupe

Pre zmenu logickej hodnoty na výstupe, bez ohľadu na aktuálnu hodnotu, použijeme podmienku *if-else*.

```
if (led == 1){ //ak je log. hodnota na vystupe log.1
    led = 0;   // na vystup sa zapise log.0
}else{         // ak je log. hodnota na vystupe log.0
    led = 1;   // na vystup sa zapise log.0
}
```

Pre lepšie pochopenie je sú funkcie *read*() a *write*() použité aj v dlhej podobe.

```
if (led.read() == 1){
    led.write(0);
}else{
    led.write(1);
```

Efektívnejšie, ako podmienku *if-else*, je použiť unárny operátor "!", ktorý zmení logickú premennej[1] a tým sa program skráti na jeden riadok.

```
led = !led; // led.write( !led.read() );
```

4.2.6 Vstavaná LED na NUCLEO-F303RE

Na vývojom kite NUCLEO-F303RE je možné použiť vstavanú LED diódu, ktorá je pripojená na pin PA_5. Inicializovať ju môžeme ako digitálny výstup na pine PA_5 alebo použijeme tmavozelené Arduino značenie (kapitola 3.2), ktoré je v tomto prípade zrozumiteľnejšie:

```
DigitalOut  led(LED1);
```

4.3 Trieda DigitalIn

Funkcie triedy DigitalIn slúžia na konfiguráciu pinu ako digitálneho vstupu a na čítanie logickej hodnoty napätia na vstupe pinu. V elektrickom obvode je každá logická hodnota reprezentovaná elektrickým napätím,Mikrokontroléry môžu byť napájané napätím v rozsau $V_{DD} \in < 2.4\,V\,,\ 3.6\,V >$, v prevažnej väčšine je to však 3.3V.

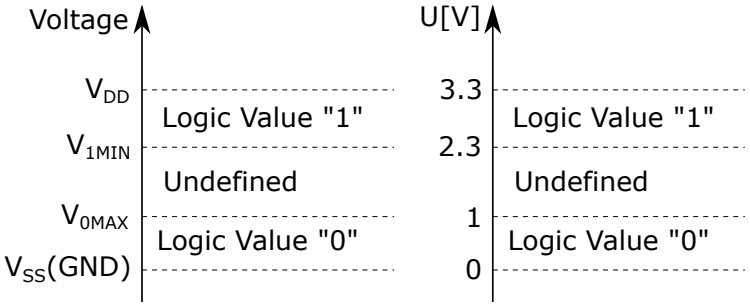
Log.0

Logická 0 je pri digitálnom vstupe reprezentovaná ako rozsah napätí
$V_{log0} \in < 0\,,\ 0.3V_{DD} >$, v prípade napájacieho napätia 3.3V je to interval $< 0\,V\,,\ 1\,V >$.

[1] logická hodnota premennej je log.0 ak je hodnota premennej 0, pre nenulovú hodnotu premennej je jej logická hodnota log.1

Log.1

Logická 1 je pri digitálnom vstupe reprezentovaná ako rozsah napätí $V_{log1} \in < 0.7V_{DD}, V_{DD} >$, v prípade napájacieho napätia 3.3V je to interval $< 2.3\,V\,, 3.3\,V >$.



Obrázok 4.2. Vstupné napäťové úrovne mikrokontroléra podľa [6] .

V prípade, že sa hodnota vyskytuje v intervale $V_{log1} \in < 0.7V_{DD}\,,\, V_{DD} >$, jedná sa o nedefinovanú logickú hodnotu, to znamená, že raz sa môže prečítať, ako log.0 a druhýkrát ako log.1. Takýto problém môže nastať napríklad pri nezapojenom vstupe(vysoká imedancia, floating input), preto sa používajú vnútorné pull rezistory mikrokontroléra, aby na vstupe pinu zabezpečili definovanú logickú úroveň napätia.

4.3.1 Konfigurácia pinu ako vstupného

Pin mikrokontroléru (napr. PA4) nakonfigurujeme ako digitálny vstup tak, že vytvoríme objekt triedy DigitalIn s nejakým názvom(napr. button) a priradíme mu tento pin. Prvým vstupným parametrom je názov pinu, ktorý ideme konfigurovať.

```
DigitalIn  button(PA_4);  //konfiguracia pinu PA4 ako digitalny vstup
```

Druhý, voliteľný parameter je vstupný mód pinu (PullUp, PullDown, PullNone), viac informácií je v kapitole 4.3.3. Ak sa parameter nepoužije, predvolený mód je PullNone.

```
DigitalIn  button(PA_4, PullUp) // nastavenie PullUp modu;
```

4.3.2 is connected

Funkcia *is_connected*() slúži na kontrolu správnosti konfigurácie pinu ako digitálneho vstupu. Funkcia nemá vstupné parametre a jej návratová hodnota je premenná typu int s hodnotou 0 alebo 1:

- 1, ak bol pin správne nakonfigurovaný
- 0 v prípade chyby konfigurácie.

Táto funkcia sa väčšinou volá na začiatku funkcie *main*() a v prípade neúspešnej inicializácie neúspešnej inicializácie sa program ukončí alebo sa oznámi chyba.

```
if ( button.is_connected() == 0 ){
    return; // ak sa pin nezinicializoval spravne, program sa ukonci
}
```

Funkcia ale nerieši problém, ak viacnásobne konfigurujeme jeden pin, preto sa táto funkcia v praxi veľmi nepoužíva, viac v kapitole 4.2.2

4.3.3 mode

Mód pinu sa mení volaním funkcie mode(), vstupným parametrom je jeden zo štyroch možných módov:

- PullUp
- PullDown
- PullNone
- OpenDrain[1].

Mikrokontroléry STM32 majú integrované programovateľné pull rezistory, konkrétne pull-up a pull-down rezistory. Hlavná úloha pull rezisotrov je zabezpečiť na vstupe pinu definovanú logickú úroveň v prípade logickej nedefinovanej hodnoty napätia (kapitola 4.3). Jenotlivé módy predstavujú použitie(resp. nepoužitie) jednotlivých pull rezistorov a odlišujú sa v logickej hodnote, ktorá sa prečíta pri ich použití(resp. nepoužití), ak je na vstupe nedefinovaná logická úroveň napätia, teda pin je pripojený "do vzduchu".

Pull-none

Pin nie je pripojený na zabudovaný programovateľný rezistor. V prípade nezapojeného vstupu (vysoká impedancia) môže prečítaná log. hodnota napätia ľubovoľne kmitať medzi log.0 a log.1

```
button.mode(PullNone);
```

Pull-down

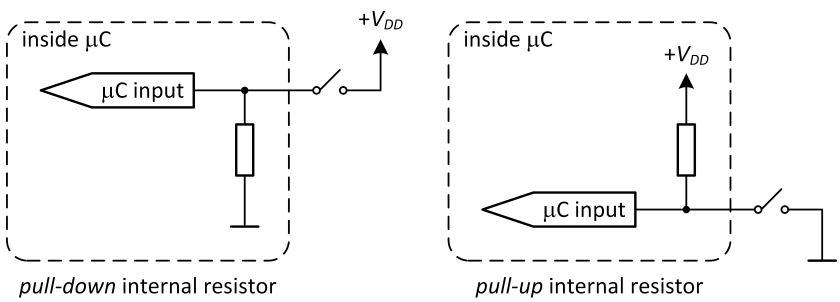
Medzi pin a zem mikrokontroléru(GND) je pripojený programovateľný pull-down rezistor. V prípade nezapojeného vstupu sa prečíta log.0.

```
button.mode(PullDown);
```

Pull-up

Medzi pin a napájanie mikrokontroléru(V_{DD}) je pripojený programovateľný pull-up rezistor. V prípade nezapojeného vstupu sa prečíta log.1.

```
button.mode(PullUp);
```



Obrázok 4.3. Zapojenie pinu v móde pull-up a pull-down dostupné z [5].

[1] Jedná sa o chybu implementácie, pretože mód open drain sa používa pri pine nakonfigurovanom ako digitálny výstup

4.3.4 read

Logická hodnota napätia na vstupe brány sa prečíta pomocu funkcie *read*(), ktorá vracia premennú typu int s hodnotou:

- 1, ak je na vstupe log.1
- 0, ak je na vstupe log.0

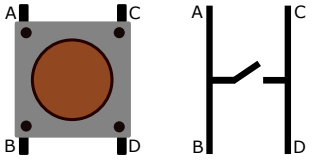
```
int state = button.read();
```

V triede DigitalIn je operátor "=" preťažený a je namapovaný na funkciu *read*(), preto väčšinou používame skrátený tvar

```
int state = button;
```

4.3.5 Tlačidlo

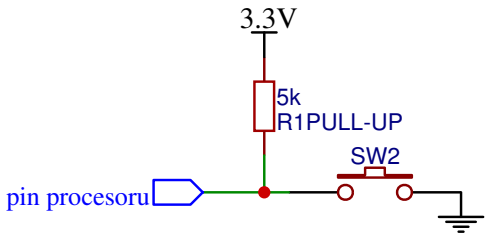
Tlačidlo je mechanická súčiastka, ktorá pozostáva z dvoch kontaktov ktoré sa spoja pri stlačení tlačidla. Na nasledujúcom obrázku je možné vidieť typické rozloženie vývodov tlačidla, vývody A a B tvoria prvý kontakt tlačidla a vývody C a D tvoria druhý kontakt tlačidla.



Obrázok 4.4. Kontakty tlačidla.

Spínanie do log.0

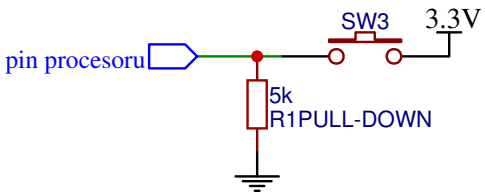
Jeden kontakt tlačidla je zapojený na zem a druhý na vstupný pin mikrokontroléra. Po stlačení tlačidla bude preto na vstupe log.0. Aby bolo možné detekovať stlačenie, musí byť na vstupe pinu log.1, keď tlačidlo nie je stlačené. Pri nestlačenom tlačidle je na vstupe pinu nedefinovaná hodnota napätia preto je potrebné použiť pull up rezistor s hodnotou rádovo v kΩ alebo nakonfigurovať pin ako vnútorný pull-up.



Obrázok 4.5. Zapojenie tlačidla so spínaním do log.0.

Spínanie do log.1

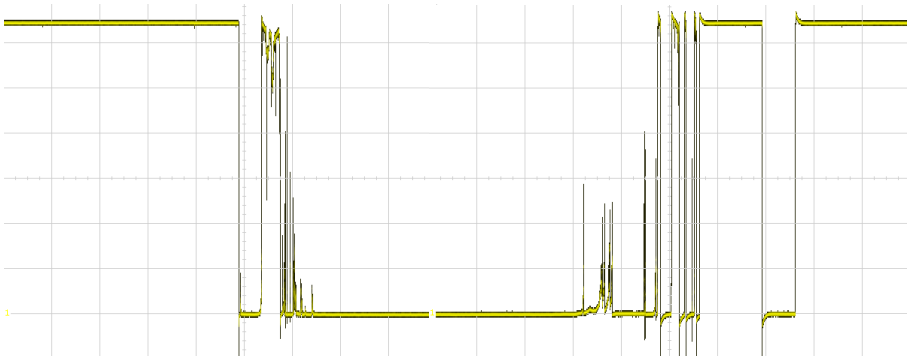
Jeden kontakt tlačidla je zapojený na V_{DD} a druhý na vstupný pin mikrokontroléra. Pre správne fungovanie je potrebné použiť pull down rezistor alebo nakonfigurovať pin ako vnútorný pull-down.



Obrázok 4.6. Zapojenie tlačidla so spínaním do log.1.

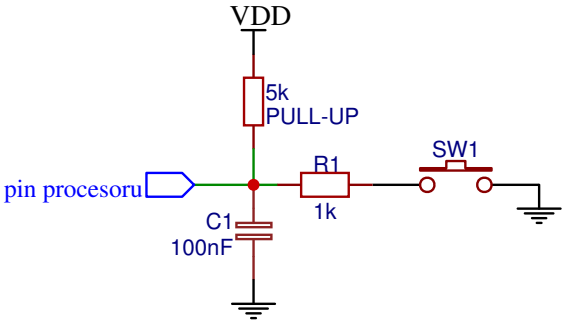
Odskakovanie kontaktov

Tlačidlo je mechanický spínač, pri jeho stlačení dochádza k spojeniu dvoch kovových častí. Pri stlačení však nedojde k okamžitému kontaktu, ale niekoľkokrát dôjde k zopnutiu a rozopnutiu kontaktu, kým sa kontakt ustáli, rovnaký jav nastane aj pri púčťaní tlačidla (obrázok 4.7). Tento jav sa nazýva bouncing, podľa stupňa opotrebovania tlačidla môže trvať až jednotky milisekúnd a môže spôsobiť detekciu viacerých stlačení mikrokontrolérom pri jednom stlačení tlačidla.

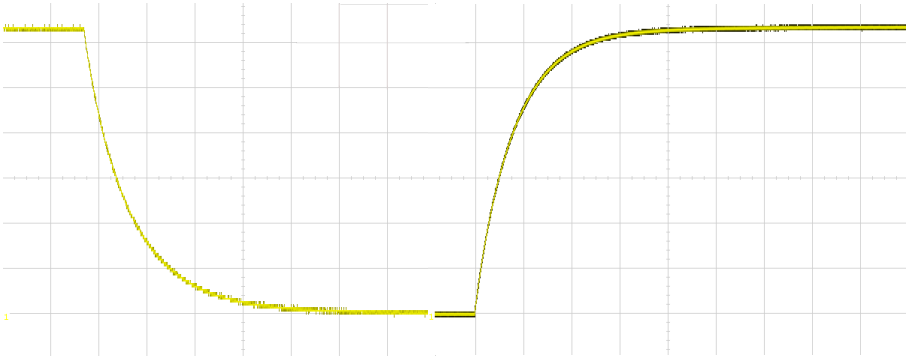


Obrázok 4.7. Odskakovanie kontaktov pri stlačení a pustení tlačidla spínaného do log.0.

Tento jav je možné odstrániť hardvérovo, použitím kapacitora. Pri zapojení na obrázku Pri zapojení na obrázku 4.8, pri stlačení sa bude správať ako integračný RC článok s odporom $R = 1\,k\Omega$ a kapacitou $C = 100\,nF$, teda kapacitor sa bude vybíjať s konštantou $\tau_{push} = 100\,\mu s$ a po rozopnutí sa bude správať ako integračný RC článok s odporom $R_{pullup} = 5\,k\Omega$ a kapacitou $C = 100\,nF$, teda kapacitor sa bude nabíjať s konštantou $\tau_{rel} = 500\,\mu s$. Na obrázku 4.9 je možné vidieť priebeh napätia na kapacitore pri stlačení a následne pri pustení tlačidla, obrázok je ilustračný, pretože priebeh pri stlačení je v skutočnosti päťkrát rýchlejší ako pri pustení tlačidla.



Obrázok 4.8. Hardvérové potlačenie bouncingu pomocou kapacitora.



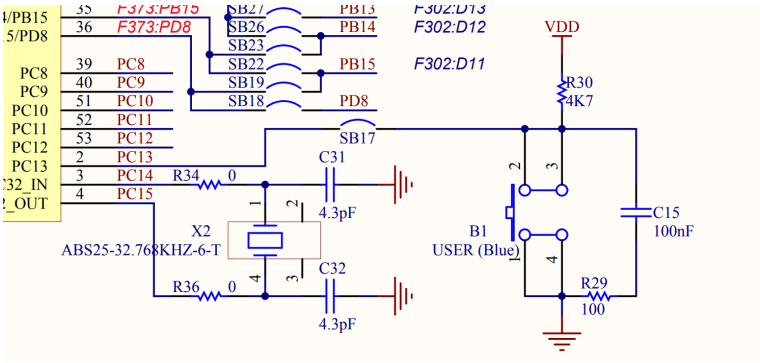
Obrázok 4.9. Priebeh napätia na kapacitore C_1 pri stlačení a pustení tlačidla.

Druhá možnosť je softvérové potlačenie tohto javu, najjednoduchší spôsob je pri prvej detekcii zopnutia kontaktu počkať nejaký čas (typicky 5ms), kým boucing prejde a až potom pokračovať v programe ďalej.

4.3.6 Vstavané tlačidlo na NUCLEO-F303RE

Na mikroprocesore NUCLEO-F303RE môžeme použiť vstavané tlačidlo, ktoré je pripojené na pin PC13. Inicializovať ho je možné ako digitálny vstup na pine PC_13 alebo sa použije tmavozelené Arduino značenie, ktoré je v tomto prípade zrozumiteľnejšie. Na obrázku 4.10 môžeme vidieť, že tlačidlo je pripojené na V_{DD} cez pull-up reziztor, preto nie je potrebné používať vnútorný pull-up a tlačidlo sa spína do log.0. V obvode je použitý aj kapacitor s hodnotou $C = 100nF$ pre obmedzenie efektu odskakovania kontaktov(bouncing).

```
DigitalIn  button(USER_BUTTON);
```



Obrázok 4.10. Zapojenie vstavaného tlačidla na NUCLEO-F303RE dostupné z [7].

4.4 Trieda DigitalInOut

Trieda DigitalInOut sa používa v prípade, že v rámci jedného programu je potrebné použiť pin v konfigurácii digitálneho vstupu aj výstupu alebo ak je potrebné použiť pin ako digitálny výstup v móde Open Drain.

4.4.1 Konfigurácia pinu ako vstupne-výstupného

Pin mikrokontroléru (napr. PA1) nakonfigurujeme ako vstupne-výstupný tak, že vytvoríme objekt triedy DigitalInOut s nejakým názvom (napr. alternate) a priradíme mu tento pin. Funkcia má 4 vstupné parametre, prvý povinný a ostatné nepovinné.
Prvým parametrom je názov pinu, ktorý ideme konfigurovať.

```
DigitalInOut  alternate (PA_1); /* inicializacia pinu PA1 ako vstupne-
                                        vystupneho s nazvom alternate */
```

Druhý, tretí a štvrtý parameter sú nepovinné, buď sa použijú všetky alebo ani jeden.

- 2. parameter určuje, či sa po vytvorení objektu pin správa ako vstupný alebo výstupný(PIN_INPUT, PIN_OUTPUT).
- 3. parameter určuje mód pinu po vytvorení objektu(PullUp, PullDown, PullNone, OpenDrain).
- 4. parameter je premenná typu int, logická hodnota na výstupe pinu vo výstupnom režime (0 -> log.0, nenulová hodnota -> log.1). Ak je pin vo vstupnom režime, logická hodnota sa na výstup zapíše po prepnutí do výstupného režimu.

Ak sa voliteľné parametre nepoužijú, po vytvorení objektu sa pin správa ako digitálne-vstupný bez použitia vnútorného programovateľného rezistora(PullNone) a po prepnutí do výstupného režimu sa na výstup zapíše log.0.

```
DigitalInOut  alternate (PA_1, PIN_INPUT, PullUp, 1); /* inicializacia
vstupneho pinu,mod pullup,log.1 na vystupe po prepnuti do rezimu vystupu*/
```

4.4.2 is connected

Funkcia *is_connected*() slúži na kontrolu správnosti konfigurácie pinu ako vstupne-výstupného. Funkcia nemá vstupné parametre a jej návratová hodnota je premenná typu int s hodnotou 0 alebo 1:

- 1, ak bol pin správne nakonfigurovaný
- 0 v prípade chyby konfigurácie.

Táto funkcia sa väčšinou volá na začiatku funkcie *main*() a v prípade neúspešnej inicializácie neúspešnej inicializácie sa program ukončí alebo sa oznámi chyba.

```
if ( alternate.is_connected() == 0 ){
    return; // ak sa pin nezinicializoval spravne, program sa ukonci
}
```

Funkcia ale nerieši problém, ak viacnásobne konfigurujeme jeden pin, preto sa táto funkcia v praxi veľmi nepoužíva (viac v kapitole 4.2.2).

4.4.3 output

Funkcia *output*() nastaví pin do režimu digitálneho výstupu, nemá vstupné parametre a nevracia žiadnu hodnotu. Na výstup brány sa zapíše logická hodnota naposledy volanej funkcie *write*().

```
alternate.output();
```

4.4.4 input

Funkcia *input*() nastaví pin do režimu digitálneho vstupu, nemá vstupné parametre a nevracia žiadnu hodnotu.

```
alternate.input();
```

4.4.5 mode

Mód pinu sa mení volaním funkcie *mode*(), vstupným parametrom je jeden zo štyroch možných módov:

- PullNone (predvolený)
- PullUp
- PullDown
- OpenDrain.

Ak je pin v režime vstupu, môže funkgovať v troch módoch: *pull* – *none*, *pull* – *up*, *pull* – *down* líšiace sa v logickej hodnote, ktorá sa prečíta na vstupe pinu, ak je pin pripojený "do vzduchu" (viac v kapitole 4.3.3). Mód OpenDrain nastaví výstupný mód pinu do režimu OpenDrain a pripojí pin na vnútorný pull-up rezistor. Mód nemení režim pinu zo vstupného na výstupný ani naopak.

PullNone

V režime vstupu v prípade, že pin je pripojený "do vzduchu" (vysoká impedancia), môže prečítaná log. hodnota napätia ľubovoľne kmitať medzi log.0 a log.1 V režime výstupu sa pin správa ako *push* – *pull*.

```
alternate.mode(PullNone);
```

PullUp

V režime vstupu sa pin správa ako vnútorný *pull* – *up*, v prípade nezapojeného vstupu (vysoká impedancia) prečíta sa log.1. V režime výstupu sa pin správa ako *push* – *pull*.

```
alternate.mode(PullUp);
```

PullDown

V režime vstupu sa pin správa ako vnútorný *pull* – *down*, v prípade nezapojeného vstupu (vysoká impedancia) prečíta sa log.0. V režime výstupu sa pin správa ako *push* – *pull*.

```
alternate.mode(PullDown);
```

OpenDrain

V režime výstupu sa pin správa ako *open* – *drain* so zapojeným *pull* – *up* rezistorom. V režime vstupu sa pin správa ako vnútorný *pull* – *up*, v prípade nezapojeného vstupu (vysoká impedancia) prečíta sa log.1.

```
alternate.mode(OpenDrain);
```

4.4.6 write

Logická hodnota napätia na výstupe brány sa mení volaním funkcie *write*(). Vstupným parametrom funkcie je premenná typu int podľa jej hodnoty sa na výstup brány zapíše log.1 alebo log.0:

- log.0 ak je hodnota premennej rovná 0
- log.1 ak je premenná nenulová

V prípade, že je pin nastavený ako výstupný, sa hodnota priamo zapíše na výstup brány. Ak je pin nastavený ako vstupný, hodnota sa zapíše na výstup brány pri najbližšom volaní funkcie *output*().

```
alternate.write(logic_value); // logic_value je premenna typu int
```

Skrátený zápis funkcie write

V triede DigitalInOut je operátor "=", ktorý priraďuje hodnotu danému objektu, preťažený a namapovaný na funkciu *write*(), teda miesto *alternate.write*(*logic_value*); väčšinou používame skrátený tvar

```
alternate = logic_value; // logic_value je premenna typu int
```

4.4.7 read

Pin v režime vstupu

V prípade, že pin je vo vstupnom režime, funkcia *read*() prečíta logickú hodnotu napätia na vstupe brány a jej návratová hodnota je premenná typu int s hodnotou 0 alebo 1:

- 1, ak je na vstupe brány log.1
- 0, ak je na vstupe brány log.0

```
int input_state = alternate.read();
```

Pin v režime výstupu

V prípade, že pin je vo výstupnom režime, funkcia *read*() prečíta logickú hodnotu napätia na výstupe brány a jej návratová hodnota je premenná typu int s hodnotou 0 alebo 1:

- 1, ak je na výstupe brány zapísaná log.1
- 0, ak je na výstupe brány zapísaná log.0

```
int output_state = alternate.read();
```

Skrátený zápis funkcie read

V triede DigitalInOut je operátor "=", ktorý priraďuje hodnotu objektu premennej, preťažený a namapovaný na funkciu *read*(), teda miesto $intstate = alternate.read()$; môžeme použiť skrátený zápis:

```
int state = alternate;
```

4.5 Trieda AnalogIn

Funkcie triedy AnalogIn slúžia na konfiguráciu pinu ako analógového vstupu a na čítanie hodnoty úmernej analógovej hodnote napätia na vstupe pinu. Prevod z analógovej hodnoty napätia do číslicovej vykonáva 12-bitový AD prevodník s postupnou aproximáciou. Jeho rozlišovacia schopnosť je $\frac{V_{DD}}{2^n} = \frac{V_{DD}}{2^{12}}$, pre hodnotu napájacieho napätia $V_{DD} = 3.3\,V$ to je $\frac{3.3}{2^{12}} \approx 0.8\,mV$. Mbed natvrdo priradí mikrokontroléru dobu odberu vzorku aj konfiguráciu hodín AD prevodníka, pre STM32F042F6P6 je doba odberu 41.5 hodinového cyklu hodín AD prevodníka, frekvencia hodín AD prevodníka je 12 MHz, teda doba odberu je

$$T_s = \frac{41.5}{12000000} s \approx 3.5\,\mu s$$

Problém pevne nastavenej doby odberu môže nastať pri meraní napätia v vysokým vstupným odporom, pretože odber vzorku prebieha tak, že po dobu T_s sa nabíja vzorkovací kondenzátor $C_s = 8pF$ z približne polovice napájacieho napätia $V_{DD}/2$. Ak je na vstupe napätie V_{in} so vstupným dporom R_{in}, odpor spôsobí, že obvod sa správa ako integračný RC článok.

$$U_{C_s}(T_s) = (V_{DD}/2 - V_{in})\, e^{\frac{-T_s}{R_{in} C_s}} + U_{in}$$

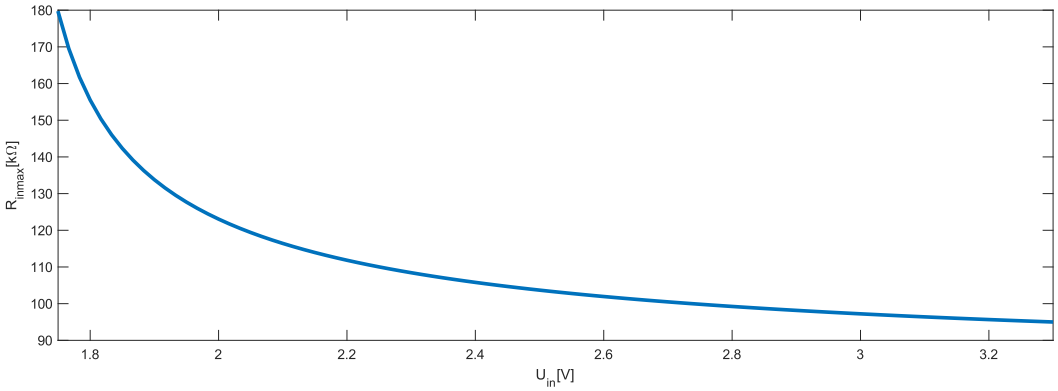
Čím väčšia je hodnota odporu, tým je väčšia hodnota RC konštany τ, čo spôsobí väčšiu chybu merania. Ak by sme chceli, aby chyba merania bola do 0.5%, pre vstupné napätie $U_{in} > \frac{V_{DD}}{2}$ platí:

$$0.995U_{in} = (V_{DD}/2 - U_{in})\, e^{\frac{-t}{R_{inmax}C_s}} + U_{in}$$

Pre $U_{in} > \frac{V_{DD}}{2} \wedge U_{in} > \frac{V_{DD}}{2*0.995}$ platí

$$R_{inmax} = -\frac{T_s}{C_s \log(\frac{0.005U_{in}}{U_{in}-V_{DD}/2})}$$

Po dosadení za $T_s \approx 3.5\,\mu s$, $V_{DD} = 3.3\,V$, $C_s = 8\,pF$ dostávame funkčnú závislosť maximálneho vstupného odporu R_{inmax} na vstupnom napätí U_{in} pre chybu merania 0.5%

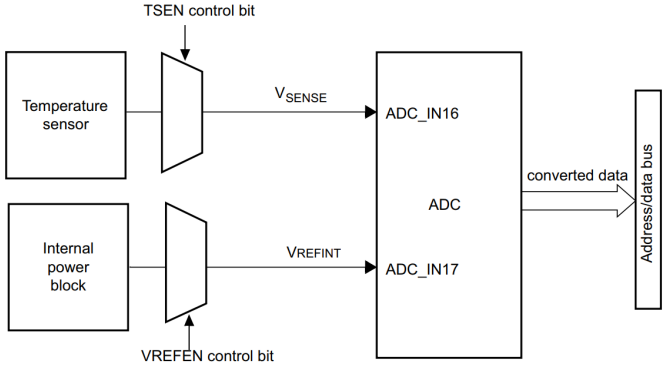


Obrázok 4.11. Závislosť odporu R_{inmax} od meraného napätia U_{in} pre chybu merania do 0.5%

Nameraná hodnota je závislá aj na hodnote napájacieho napätia, mbed umožňuje merať vnútornú referenciu (pre STM32F042F6P6 na 17. kanále ADC) a pomocou kalibračnej konštanty uloženej v pamäti mikrokontroléra (pre STM32F042F6P6 na adrese $0x1FFFF7BA - 0x1FFFF7BB$) je možné vypočítať skutočnú hodnotu napájacieho napätia a meranej hodnoty napätia

$$U_{real} = \frac{3.3\,V * UREF_{CAL} * ADC_{read\_u16}}{UREF_{read\_u16} * 2^n}$$

kde $UREF_{CAL}$ je kalibračná konštanta, ADC_{read_u16} je hodnota úmerná napätiu na pine prečítaná pomocou funkcie $read\_u16()$, $UREF_{read\_u16}$ je hodnota vnútornej referencie prečítaná pomocou funkcie $read\_u16()$ a n je počet bitov ADC.
Existuje aj možnosť merať teplotu čipu (pre STM32F042F6P6 na 16. kanále ADC). Hodnotu získanú z ADC je ešte potrebné prepočítať pomocou dvoch kalibračných konštánt uložených v pamäti mikroprocesoru podľa [4].



Obrázok 4.12. Vnútorná štruktúra ADC pre STM32F042F6P6 prevzatá z [8].

4.5.1 Konfigurácia pinu ako analógového vstupu

Pin mikrokontroléru (napr. PB1) nakonfigurujeme ako analógový vstup tak, že si vytvoríme objekt triedy AnalogIn s nejakým názvom(napr. trimmer) a priradíme ho k tomuto pinu. Vstupným parametrom je názov pinu, ktorý ideme konfigurovať, ktorý má funkcionalitu analógového vstupu (kapitola 3).

```
AnalogIn trimmer(PB_1);
AnalogIn vrefint(ADC_VREF);// objekt pre meranie vnutornej referencie
AnalogIn tempint(ADC_TEMP);//objekt pre meranie teploty cipu
```

4.5.2 Čítanie analógovej hodnoty napätia na vstupe

Po vytvorení objektu je doba odberu vzorku T_s nastavená na minimálnu hodnotu a až pri prvom meraní sa nastaví na vyššiu hodnotu(u STM32F042F6P6 je to 41.5 cyklov ADC hodín) Po vytvorení objektu je možné prečítať hodnotu úmernú analógovému napätiu na vstupe pinu pomocou funkcií *read*() alebo *read_u*16().

read

Funkcia *read*() vracia premennú typu float v rozsahu <0,1> , pričom číslo 0 odpovedá napätiu V_{SS} (GND)a číslo 1 odpovedá napätiu V_{DD} (napájacie napätie procesoru).

```
  float hodnota = trimmer.read();
```

V triede AnalogIn je operátor "=" preťažený a je namapovaný na funkciu *read*(), preto väčšinou používame skrátený tvar

```
  float hodnota = trimmer;
```

read_u16

Funkcia *read_u*16() nevracia priamo výstup AD prevodníka, ale premapuje ho na 16-bitové číslo teda na celé číslo v rozsahu <0 ; 65535>.

```
int hodnota = trimmer.read_u16();
int internal_reference = vrefint.read_u16();
int internal_temp = tempint.read_u16();
```

4.5.3 Generovanie pseudonáhodných sekvencií

Na generovanie pseudonáhodných čísel slúži funkcia *rand*(), ak by sme, napríklad, chceli vygenerovať 10 náhodných čísel do medzi 0 a 100, mohli by sme použiť nasledujúci program:

```
int rand_val[20];
for (int i = 0; i < 20; i++){
    rand_val[i] = rand()\%100;
}
```

Samotná funkcia *rand*() generuje čísla od 0 do 32767, preto sme použili operátor modulo, ktorý vráti zvyšok po delení číslom 100. Ak by sme si ale program prehrali viackrát, zistili by sme, že sekvencia bude vždy obsahovať rovnaké čísla (prvý riadok tabuľky 4.1). Dôvod je ten, že funkcia *rand*() vracia postupne čísla z danej pseudonáhodnej sekvencie čísel, a ak používateľ nenastaví inak, funkcia vracia čísla od začiatku danej sekvncie. Preto je treba použiť funkciu *srand*(), kotrá pomocou parametru, náhodného čísla, do pseudonáhodnej sekvencie zasadí počiatočný index, od ktorého sa budú náhodné čísla čítať. Ak na analógovom vstupe nie je nič zapojené, niekoľko najnižších bitov nameranej hodnoty

bude šumieť, čo môže poslúžiť ako **náhodné** číslo, parameter funkcie *srand*(). Na 2., 3. a 4. riadku tabulky 4.1 je vidieť, že pseudonáhodné sekvencie sú pri opakovanom spustení programu rôzne, teda pre nenáročné aplikácie môžeme použiť meranie nezapojeného analógového vstupu ako pseudonáhodný generátor.

```
#include "mbed.h"
AnalogIn floating_input(PA_0);
int main(){
    int rand_val[20];
    srand(floating_input.read_u16());
    for (int i = 0; i < 20; i++){
     rand_val[i] = rand()\%100;
    }
}
```

bez srand()	68	67	9	60	83	41	60	67	7	64
1. priebeh	19	97	94	10	44	99	33	39	46	51
2. priebeh	69	42	47	16	72	7	62	83	47	48
3. priebeh	22	91	97	84	64	71	84	76	15	41

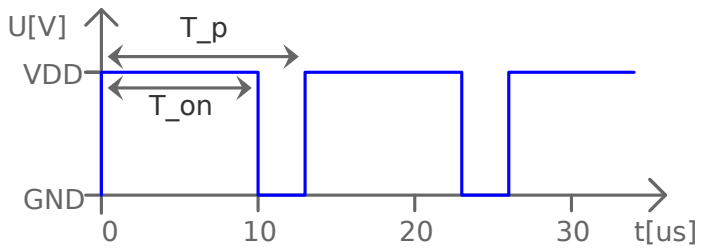
Tabuľka 4.1. Tabuľka pseudonáhodných sekvencií.

4.6 Trieda PwmOut

Funkcie triedy PwmOut slúžia k inicializácii pinu ako generátora PWM signálu a k zmene parametrov generovaného signálu.

4.6.1 PWM signál

Pulzne-šírkovo-modulovaný signál je periodický signál so základnými parametrami



Obrázok 4.13. Pulzne modulovaný signál

Perióda T_p

Perióda T_p je doba trvania jedného cyklu. V mbede je možné nastavovať periódu PWM signálu s maximálnym rozlíšením na mikrosekundy.

Šírka pulzu T_{on}

Šírka pulzu T_{on} je doba vrámci jednej periódy T_p, kedy je výstup v log.1. V mbede je možné nastavovať šírku pulzu s rozlíšením na mikrosekundy.

Strieda

Strieda PWM signálu je definovaná ako pomer šírky pulzu T_{on} ku perióde signálu T_p

$$stride = \frac{T_{on}}{T_p}, T_{on} < T_p$$

teda je to číslo v intervale $< 0; 1 >$ Strieda je nepriamy parameter PWM signálu. To znamená, že ak v mbede zmeníme striedu, v skutočnosti sa zmení šírka pulzu T_{on} tak, aby bol pomer $\frac{T_{on}}{T_p}$ čo najbližšie k zadanému číslu. Na obrázku 4.13 je je signál s periódou $T_p = 13\,\mu s$, teda strieda pracuje s rozlíšením $\frac{1}{13}$. Ak zadáme striedu napríklad 0.5, v mbede sa nastaví šírka pulzu na $T_{on} = 7\,\mu s$(max. rozlíšenie je na μs) a skutočná strieda bude $\frac{7}{13} \approx 0.54$ (najbližšia požadovanej hodnote). V extrémnom prípade, ak $T_p = 1\mu s$, strieda môže nadobúdať iba hodnoty 0 alebo 1.

4.6.2 Konfigurácia pinu

Pin mikrokontroléru(napr. PA6) je možné nakonfigurovať ako generátor PWM signálu, ak je v pinout diagrame označený fialovou farbou s označením PWM (kapitola 3). Oficiálne diagramy museli byť upravené, pretože na určitých pinoch boli zamenené čísla timerov a kanálov a niektoré piny nie je možné použiť, lebo sú predvolene v takej konfigurácii, že pri zmene ich konfigurácie na PwmOut dôjde k chybe programu. Konfiguráciu prevedieme vytvorením objektu triedy PwmOut s nejakým názvom (napr. led_pwm) a priradením pinu tomuto objektu. Vstupným parametrom je názov pinu, ktorý ideme konfigurovať.

```
PwmOut  led_pwm(PA_6);
```

Generovanie pulzne šírkovo modulovaného signálu funguje za pomoci timerov. Čísla x a y v názve PWM x/y v pinout diagrame znamenajú, že kanál y timera x zbezpečuje generovanie PWM, Platí tu však obmedzenie, že na všetkých kanáloch jedného timera musí mať PWM signál rovnakú periódu. Navyše, ak sa v názve zhodujú timer aj číslo kanálu a jeden z kanálov je negovaný, súčasné generovanie PWM nie je možné ani na rovnakej frekvencii.

4.6.3 Nastavenie periódy PWM signálu

Po vytvorení objektu je potrebné nastaviť prvý parameter, periódu PWM signálu. Na to slúžia funkcie *period*(), *period_ms*(), *period_us*(). V mbede je možné nastaviť periódu s rozlíšením maximálne na mikrosekundy. Pri zmene periódy PWM signálu sa automaticky zmení hodnota šírky pulzu T_{on} tak, aby sa zachovala reála hodnota striedy PWM signálu. Reálnou hodnotou striedy sa myslí pomer $\frac{T_{on}}{T_p}$, nie zadaná hodnota striedy, pretože tieto hodnoty sa môžu líšiť z dôvodov popísaných v kapitole 4.6.1. Perióda T_p musí byť kladné číslo, preto už pri vytváraní objektu triedy PwmOut sa nastaví nejaká hodnota(pre rôzne mikrokontroléry môže byť rôzna) a šírka pulzu T_{on} sa nastaví na 0(strieda je 0), aby sa žiaden signál negeneroval. Odporúča sa preto najskôr nastaviť periódu a až potom druhý parameter PWM signálu.

period

Funkcia *period*() nastaví periódu generovaného PWM signálu na hodnotu danú vstupným parametrom, premennou typu float vyjadrujúcu čas v sekundách s maximálnym rozlíšením na mikrosekundy. Pri číslach s rozlíšením väčším ako mikrosekundy, sa ďalší desatinný rozvoj zanedbá. V nasledujúcom príklade sa z tohto dôvodu perióda nastaví na rovnakú hodnotu v oboch prípadoch.

```
float period1 = 0.000345;  // cislo s rozlisenim na mikrosekundy: 345us
led_pwm.period(period1);   // perioda sa nastavi na hodnotu 345us
float period2 = 0.0003459; // cislo s vyssim rozlisenim: 345.9us
led_pwm.period(period2);   // perioda sa nastavi na hodnotu 345us
```

period_ms

Funkcia *period_ms*() nastaví periódu generovaného PWM signálu na hodnotu danú vstupným parametrom, premennou typu int vyjadrujúcu čas v milisekundách.

```
int  period = 3450;         // cele cislo:  perioda v milisekundach
led_pwm.period_ms(period); // perioda sa nastavi na 3450ms = 3.45s
```

period_us

Funkcia *period_us*() nastaví periódu generovaného PWM signálu na hodnotu danú vstupným parametrom, premennou typu int vyjadrujúcu čas v mikrosekundách.

```
int  period = 345;          // cele cislo: perioda v mikrosekundach
led_pwm.period_us(period); // perioda sa nastavi na hodnotu 345us
```

4.6.4 Nastavenie druhého parametru PWM signálu

Druhý parameter PWM signálu je možné nastaviť pomocou šírky pulzu(priamy parameter) alebo striedy(nepriamy parameter) PWM signálu.

Nastavenie šírky pulzu T_{on}

Doba šírky pulzu T_{on} sa mení pomocou funkcií *pulsewidth*(), *pulsewidth_ms*() alebo *pulsewidth_us*(). V mbede je možné nastaviť šírku pulzu s maximálnym rozlíšením na mikrosekundy. Pri zmene periódy PWM signálu sa automaticky zmení hodnota šírky pulzu T_{on} tak, aby sa zachovala reála hodnota striedy PWM signálu, teda pomer $\frac{T_{on}}{T_p}$.

```
led_pwm.period(1);       // perioda sa nastavi na hodnotu 1s
led_pwm.pulsewidth(0.5); // strieda je 0.5/1=0.5
led_pwm.period_us(2);    /* zmena periody na 2s, sirka pulzu sa
                            automaticky zmeni na 1s, strieda je stale 0.5 */
```

pulsewidth

Funkcia *pulsewidth*() nastaví šírku pulzu T_{on} PWM signálu na hodnotu danú vstupným parametrom, premennou typu float vyjadrujúcu čas v sekundách. Funkcia nastaví šírku pulzu s rozlíšením na mikrosekundy. Pri číslach s rozlíšením väčším ako mikrosekundy, sa ďalší desatinný rozvoj zanedbá.

```
float pwm_period = 0.000345;
led_pwm.period(pwm_period); // perioda sa nastavi na 345us
float pulse1 = 0.00033;     // cislo s rozlisenim na mikrosekundy: 330us
led_pwm.pulsewidth(pulse1); // sirka pulzu sa nastavi na hodnotu 330us
float pulse2 = 0.0001234;   // cislo s vyssim rozlisenim: 123.4us
led_pwm.pulsewidth(pulse2); // sirka pulzu sa nastavi na hodnotu 123us
```

pulsewidth_ms

Funkcia *pulsewidth_ms*() nastaví šírku pulzu PWM signálu na hodnotu danú vstupným parametrom, premennou typu int vyjadrujúcu čas v milisekundách.

```
int  pulse = 34;              // cele cislo, sirka pulzu v milisekundach
led_pwm.pulsewidth_ms(pulse); // sirka pulzu sa nastavi na hodnotu 34ms
```

pulsewidth_us

Funkcia *pulsewidth_us*() nastaví šírku pulzu PWM signálu na hodnotu danú vstupným parametrom, premennou typu int vyjadrujúcu čas v mikrosekundách.

```
int  pulse = 197;             // cele cislo, sirka pulzu v mikrosekundach
led_pwm.pulsewidth_ms(pulse); // sirka pulzu sa nastavi na hodnotu 197us
```

Nastavenie striedy PWM signálu

Strieda PWM signálu sa mení pomcou funkcie *write*(). Vstupným parametrom funkcie je premenná typu float v rozsahu <0 , 1>, požadovaná strieda signálu.

```
led_pwm.write(0.32);
```

Pri zmene periódy PWM signálu sa automaticky zmení hodnota šírky pulzu T_{on} tak, aby sa zachovala hodnota striedy.

```
led_pwm.period(1);   // perioda sa nastavi na 1s
led_pwm.write(0.23); // strieda je 23%
led_pwm.period(2);   // zmena periody na 2s, strieda je stale 23%
```

Strieda nie je priamy parameter PWM signálu, volaním funkcie *write*() sa v skutočnosti mení priamy parameter, šírka pulzu T_{on} tak, aby pomer $\frac{T_{on}}{T_p}$ bol čo najbližšie zadanému číslu. Keďže oba priame parametre T_p aj T_{on} je možné zadať s rozlíšením maximálne na mikrosekundy, tak čím menšia je perióda PWM signálu, tým s menším rozlíšením striedy pracujeme. V extrémnom prípade, ak je dĺžka periódy $T_p = 1\mu s$, šírka pulzu T_{on} môže mať iba $1\mu s$ alebo $0\mu s$, a strieda môže nadobúdať iba hodnoty 1 alebo 0.

```
led_pwm.period_us(3); // tp=3us, strieda moze byt iba 0, 1/3, 2/3 alebo 1
led_pwm.write(0.38);  //realna hodnota striedy bude 0.33
led_pwm.write(0.5);   //realna hodnota striedy bude 0.6667
```

Skrátený zápis funkcie write

V triede PwmOut je operátor "=", ktorý priraďuje hodnotu danému objektu, preťažený a namapovaný na funkciu *write*(), teda miesto *led_pwm.write*(*stride*); sa môže použiť skrátený zápis:

```
led_pwm = stride; //stride: premenna float v rozsahu <0;1>
```

4.6.5 read

Funkcia *read*() slúži na čítanie aktuálnej hodnoty striedy, jej návratatová hodnota je premenná typu float v rozsahu <0,1>.

```
float stride = led_pwm.read();
```

Skrátený zápis funkcie read

V triede PwmOut je operátor "=", ktorý priraďuje hodnotu daného objektu premennej, preťažený a namapovaný na funkciu *read*(), teda miesto *floatstride* = *led_pwm.read*(); sa môže použiť skrátený zápis:

```
float stride = led_pwm;
```

4.7 Trieda InterruptIn

Funkcie triedy InterruptIn slúžia na konfiguráciu pinu ako digitálneho vstupu, ale navyše s možnosťou hardvérového prerušenia pri určitej udalosti, konkrétne v mbede môže dôjsť k prerušeniu pri nábežnej alebo spádovej hrane logickej hodnoty napätia na vstupe pinu. Po vyvolaní prerušenia sa pozastaví behu programu, vykoná sa funkcia obsluhy prerušenia a následne program pokračuje tam, kde došlo k jeho prerušeniu.

4.7.1 Funkcia obsluhy prerušenia

Funkcia obsluhy prerušenia nesmie mať vstupné argumenty a nevracia žiadnu hodnotu. Nesmie obsahovať nekonečné cykly, časovo náročné funkcie (*wait_ms*(), *wait_us*(), *printf*()), ktoré by blokovali beh programu, a zakázané je používať aj funkcie, ktoré nemôžu byť volané viacnásobne(*new*() a *malloc*()), pretože by mohlo dôjsť k nedefinovanému chovaniu programu. Ak funkcia pristupuje ku globálnym premenným, tieto premenné musia byť pri inicializácii označené kľúčovým slovom *volatile*, ktoré zabezpečí, že pri kompilácii programu nedôjde k optimalizácii použitia danej premennej a ku premennej bude možné pristupovať kedykoľvek počas behu programu [9].

4.7.2 Konfigurácia pinu

Pin mikrokontroléru (napr. PA5) nakonfigurujeme ako digitálny vstup s možnosťou hardvérového prerušenia vytvorením objektu triedy InterruptIn s nejakým názvom (napr. button_interrupt) a priradením pinu tomuto objektu. Prvým vstupným parametrom je názov pinu, ktorý ideme konfigurovať.

```
InterruptIn button_interrupt (PA_5);
```

Druhý, voliteľný parameter je vstupný mód pinu (PullUp, PullDown, PullNone), viac informácií je v triede 4.3.3. Ak sa parameter nepoužije, predvolený mód je PullNone.

```
DigitalIn  button_interrupt(PA_5, PullUp) // nastavenie PullUp modu;
```

Po vytvorení objektu je funkcionalita prerušenia predvolene povolená.

4.7.3 mode

Po vytvorení objektu triedy InterruptIn je možné meniť mód pinu rovnako, ako v triede DigitalIn (kapitola 4.3). Predvolený mód pinu je PullNone.

PullUp

```
button_interrupt.mode (PullUp);   // pripojenie pinu na pull-up rezistor
```

PullDown

```
button_interrupt.mode (PullDown); // pripojenie pinu na pull-down rezistor
```

PullNone

```
button_interrupt.mode (PullNone); // ziadny vnutorny rezistor
```

4.7.4 read

Logická hodnota napätia na vstupe pinu sa prečíta pomocou funkcie *read*() rovnako, ako v triede DigitalIn (kapitola 4.3). Návratová hodnota funkcie je premenná typu int s hodnotou 0 alebo 1:

- 1, ak je na vstupe pinu log.1
- 0, ak je na vstupe pinu log.0

```
int state = button_ interrupt.read ();
```

V triede InterruptIn je operátor "=" preťažený a je namapovaný na metódu *read*() , preto sa častejšie používa

```
int state = button_ interrupt;
```

4.7.5 rise

Volaním funkcie *rise*() sa objektu sa priradí funkcia obsluhy prerušenia (napr. *catch_rising_edge*), ktorá sa zavolá vždy po vyvolaní hardvérového prerušenia nábežnou hranou (rising edge) logickej hodnoty napätia na vstupe daného pinu. Typicky sa táto funkcia volá na začiatku funkcie *main*(), pri jej viacnásobnom volaní prerušenie obsluhuje naposledy priradená funkcia.

```
InterruptIn button_interrupt(PA_4, PullDown); // pin v mode pull-down
volatile int count = 0; /* globalna premenna pouzita pri preruseni, je
                           oznacena  volatile */
void catch_rising_edge(){ // obsluha prerusenia pocita nabezne hrany
    count++;
}
main(){
    ...
    button_interrupt.rise(&catch_rising_edge); /* priradenie adresy
              funkcie, ktorá sa vola pri preruseni na nabeznu hranu */
    ...
}
```

4.7.6 fall

Volaním funkcie *fall*() sa objektu sa priradí funkcia obsluhy prerušenia (napr. *catch_falling_edge*), ktorá sa zavolá vždy po vyvolaní hardvérového prerušenia spádovou hranou (falling edge) logickej hodnoty napätia na vstupe daného pinu. Typicky sa táto funkcia volá na začiatku funkcie *main*(), pri jej viacnásobnom volaní prerušenie obsluhuje naposledy priradená funkcia.

```
InterruptIn button_interrupt(PA_4, PullUp); // pin v mode pull-up
volatile int count = 0; /* globalna premenna pouzita pri preruseni, je
oznacena  volatile */
void catch_falling_edge(){ // obsluha prerusenia pocita spadove hrany
    count++;
}
main(){
    ...
    button_interrupt.fall(&catch_falling_edge); /* priradenie adresy
              funkcie, ktorá sa vola pri preruseni na spadovu hranu */
    ...
}
```

4.7.7 disable irq

Obsluhu hardvérového prerušenia pre daný pin je možné zakázať volaním funkcie *disable_irq*(). Po vytvorení objektu triedy InterruptIn je hardvérové prerušenie predvolene povolené.

```
button_interrupt.disable_irq();
```

4.7.8 enable irq

Funkcia *enable_irq*() opätovne povolí obsluhu prerušenia na danom pine.

```
button_interrupt.enable_irq();
```

4.8 Trieda Serial

Mikrokontrolér môže komunikovať s inými zariadeniami pomocou sériovej komunikácie. V mbede je možné použiť sériovú komunikáciu v asynchrónnom režime (UART). Aby zariadenia mohli spolu komunikovať cez sériový port, musia byť splnené nasledovné podmienky:

- Zariadenia pracujú na rovnakých logických napäťových úrovniach.
- Zeme oboch zariadení sú prepojené.
- TX(vysielaci) pin prvého zariadenia je pripojený na RX(prijímací) pin druhého zariadenia.
- RX pin prvého zariadenia je pripojený na TX pin druhého zariadenia.
- Zariadenia majú rovnakú rýchlosť komunikácie (baudrate).
- Formát správ je u oboch zariadení rovnaký. Vo väčšine prípadov bude mikrokontrolér komunikovať s počítačom. Moderné počítače už nepoužívajú sériový port, preto je nutné použiť prevodník UART-USB, ktorý v počítači vytvorí virtuálny sériový port a pomocou terminálového programu je možné s mikrokontrolérom komunikovať(vývojový kit NUCLEO-F303RE má zabudovaný UART-USB prevodník).

4.8.1 Inicializácia sériového portu

Sériový port môžeme nainicializovať na dvojici pinov, ktoré sú v pinout diagrame v kapitole 3 označené žltou farbou s označením Serialx TX a Serialx RX. Inicializáciu prevedieme vytvorením objektu triedy Serial s nejakým názvom (napr. device) a priradením v poradí TX(napr. PA2) a RX(PA3) pinov tomuto objektu.
Prvými dvomi parametrami sú v poradí TX a RX piny sériového portu.

```
Serial device(PA_2, PA_3) ; // v poradi TX pin a RX pin
```

Tretím, nepovinným parametrom je premenná typu int, rýchlosť komunikácie(baudrate). Ak sa parameter nepoužije, predvolená rýchlosť je 9600 Bd/s

```
Serial device(PA_2, PA_3, 115200); // v poradi TX pin, RX pin a baudrate
```

4.8.2 baud

Rýchlosť komunikácie, baudrate, je možné nastaviť volaním funkcie *baud*(). Vstupným paramentrom je premenná typu int, požadovaná hodnota baudrate, najčastejšie používame jednu z nasledujúcich hodnôt: 9600, 14400, 19200, 38400, 57600, 115200. Predvolená hodnota baudrate je 9600 Bd/s.

```
device.baud(115200);
```

4.8.3 format

Každá správa, ktorá sa posiela cez sériový port, má presne danú štruktúru. Začína start bitom, za ním nasledujú dátové bity, ďalej nepovinný paritný bit a správa končí jedným alebo viacerými stop bitmi. Predvolený formát správy v mbede je: 1 start bit, 8 dátových bitov, žiaden paritný bit a 1 stop bit.

```
device.format(8, SerialBase::None,1);
```

Nasledujúce informácie boli získané z implementačného súboru triedy Serial z [10].

Počet dátových bitov

Prvý parameter, počet dátových bitov je prvý parameter funkcie *format*(), je premenná typu int s možnými hodnotami 7, 8 alebo 9:

- 7 s podmienkou, že je použitý paritný bit
- 8, paritný bit môže, ale nemusí byť použitý
- 9 s podmienkou, ža paritný bit nie je použitý

Pri zadaní iného čísla alebo pri porušení podmienky dôjde k chybe programu.

Paritný bit

Druhý parameter funkcie *format*(), paritný bit s nasledujúcimi možnosťami:

- SerialBase::None(predvolený), správa neobsahujuhe paritný bit.
- SerialBase::Odd, paritný bit sa nastaví na takú hodnotu, aby počet jednotiek v správe bol nepárny.
- SerialBase::Even, paritný bit sa nastaví na takú hodnotu, aby počet jednotiek v správe bol párny.

Počet stopbitov

Počet stopbitov je tretí parameter funkcie *format*(). Je to premenná typu int s možnáými hodnotami 1 alebo 2, pričom predvolená hodnota je 1. Pri zadaní iného čísla sa počet stopbitov nastaví na 1.

```
device.format(9, SerialBase::None,2);
device.format(8, SerialBase::Odd, 1);
device.format(8, SerialBase::None,2);
device.format(7, SerialBase::Even,1);
```

Ak použijeme 7 dátových bitov, musíme pridať aj paritný bit. Ak použijeme 8 dátových bitov, môžeme, ale nemusíme použiť paritný bit. Ak použijeme 9 dátových bitov, nesmieme použiť paritný bit. Počet stopbitov môže byť 1 alebo 2, ak použijeme akékoľvek iné číslo, automaticky sa počet stopbitov nastaví na 1.

4.8.4 getc

Na prečítanie jedného znaku zo sériového portu sa používa metóda *getc*(), návratová hodnota je premenná typu char, prečítaný znak.

```
char sign = device.getc();
```

4.8.5 readable

Funkcia *readable*() slúži na zistenie, či je nejaký znak na prečítanie zo sériového portu. Návratová hodnota je premenná typu bool s hodnotou true alebo false:

- true, ak je na aspoň jeden znak pripravený na prečítanie zo sériového portu.
- false v opačnom prípade

. Väčšinou sa používa v kombinácii s funkciou *getc*().

```
if(device.readable()) {
    char sign = device.getc();
}
```

4.8.6 putc

Pomocou funkcie *putc*() sa do sériového portu zapíše jeden znak. Vstupným parametrom je premenná typu char, znak ktorý posielame. Návratová hodnota je premenná typu int, ASCII hodnota zapísaného znaku alebo záporné číslo v prípade chyby zápisu.

```
char sign = 'a';
device.putc(sign);
```

4.8.7 writable

Funkcia *writable*() slúži na zistenie, či je možné do sériového portu zapísať znak. Návratová hodnota je premenná typu bool s hodnotou true alebo false:

- true, ak je možné do sériového portu zapísať znak
- false v opačnom prípade Najčastejšie použitie je v kombinácii s funkciou *putc*()

```
if(device.writable()) {
    char sign = device.putc();
}
```

4.8.8 printf

Formátovaný reťazec je možné poslať poocou funkcie *printf*().

Formát premenných

Reťazec môže obsahovať hodnoty premenných, ktoré sú do reťazca vložené pomocou formátovacej direktívy, názvy premenných sú zoradené a oddelené čiarkou za reťazcom.

- char: `%c`
- int: `%d`
- float: `%f`
- string: `%s`

Vstupným argumentom je reťazec, ktorý chceme poslať a za ním nasledujú premenné, ktoré sme do reťazca zapisovali. Návratová hodnota funkcie je premenná typu int, počet znakov zapísaných do sériového portu alebo záporné číslo v prípade chyby zápisu.

```
float pi = 3.14159265359;
char sign =  'A';  // jednoduche anglicke uvodzovky
int val_A = 65;{\tt '}
device.printf("pi = \%f, ASCII hodnota \%c je \%d\n", pi, sign, val_A);
```

4.8.9 scanf

Na prečítanie reťazca zo sériového portu sa používa funkcia *scanf*(). Vstupný argument je reťazec, ktorý očakávame a adresy premenných, ktorým sa priradia prečítané hodnoty. Funkcia číta znaky až do doby, kedy prichádzajúci znak porušuje očakávaný formát.

```
int num_int = 0;
float num_float = 0;
int read_params = 0;
device.printf("zadaj cele a desatinne cislo oddelene medzerou\n\r")
read_params = device.scanf("\%d \%f", &num_int, &num_float); // pred nazvom
premennych je &
```

Návratová hodnota funkcie je premenná typu int a jej hodnota je počet úspešne priradených hodnôt premenným alebo záporné číslo v prípade chyby čítania. Ak by sme

do sériového portu zapísali napríklad “-24 2E-3a”, funkcia priradí premennej *num_int* hodnotu -24 a premennej *num_float* hodnotu 0.003, pretože znak ’a’ už porušuje očakávaný formát a funkcia prestane čítať ďalšie znaky. Funkcia váti číslo 2.
Ak by sme do sériového portu zapísali “123456m34.56”, funkcia priradí premennej *num_int* hodnotu 123456. Premennej *num_float* ale nepriradí žiadnu hodnotu, pretože znak ’m’ už porušuje očakávaný formát a prestane čítať ďalšie znaky. Funkcia váti číslo 1. Zvyšné znaky sa prečítajú pri najbližšom volaní *getc*() alebo *scanf*().

4.8.10 attach

Čítanie znakov zo sériového portu je možné riešiť aj hardvérovo tak, že po prijatí znaku sa automaticky vyvolá prerušenie a zavolá sa funkcia obsluhujúca prerušenie, v ktorej sa znak prečíta. Vstupným parametrom funkcie *attach*() je adresa funkcie, ktorá bude obsluhovať prerušenie, preto sa pred jej názov ešte pridá znak ”&”. Funkcia obsluhy prerušenia nesmie mať vstupné argumenty a nevracia žiadnu hodnotu. Nesmie obsahovať nekonečné cykly, časovo náročné funkcie (*wait_ms*(), *wait_us*(), *printf*()), ktoré by blokovali beh programu, a zakázané je používať aj funkcie, ktoré nemôžu byť volané viacnásobne(*new*() a *malloc*()), pretože by mohlo dôjsť k nedefinovanému chovaniu programu. Ak funkcia pristupuje ku globálnym premenným, tieto premenné musia byť pri inicializácii označené kľúčovým slovom *volatile*, ktoré zabezpečí, že pri kompilácii programu nedôjde k optimalizácii použitia danej premennej a ku premennej bude možné pristupovať kedykoľvek počas behu programu [9].

```
Serial device (PA_2, PA_3, 115200);
volatile char sign;
void read_character(){
    sign = device.getchar();
}
main(){
    ...
    device.attach(&read_character); // priradenie obsluhy prerusenia
    ...
}
```

4.8.11 Komunikácia s PC

Počítač v dnešnej dobe nie je vybavený sériovým portom, preto aby mikrokontrolér mohol komunikovať s počítačom, musíme použiť adaptér USB to UART. Mikrokontrolér komunikuje pomocou sériovej komunikácie s prevodníkom, ktorý preposiela prijaté správy do počítača pomocou USB komunikácie a naopak. Aby komunikácia správne fungovala, pri prevodníku typu CH340 je potrebné:

- prepojiť zem mikrokontroléra a prevodníka,
- skratosvorkou prepojiť piny V_{CC} a 3V3 na prevodníku, aby prevodník s mikrokontrolérom komunikoval na 3.3-voltovej logike,
- pripojiť TX(vysielaci) pin mikrokontroléru na RX(prijímací) pin prevodníka,
- pripojiť RX pin mikrokontroléru na TX pin prevodníka.

Po zapojení prevodníka do počítača, prejdeme do Správcu zariadení a mali by sme vidieť nové zariadenie s číslom COM portu. Ak sa zobrazuje chyba, musíme manuálne nainštalovať driver k danému prevodníku.

Obrázok 4.14. Prevodník UART-USB typu CH340

Na vytvorený COM port sa pripojíme pomocou terminálového programu. Odporúčame použiť terminálový program Tera Term, pretože na rozdiel od iných programov (napr. Putty), stačí spojenie nadviazať iba raz a po prerušení spojenia nemusíme spojenie nanovo inicializovať. Program Tera Term je možné stiahnuť ako spustiteľný .exe súbor[1]. Pre zahájenie komunikácie sa pripojíme na daný COM port. Predvolené parametre komuikácie sú 9600 Bd/s baudrate, 8 dátových bitov, žiaden paritný bit a 1 stopbit. Tieto parametre je možné zmeniť zakaždým manuálne po kliknutí na $Setup- > Serial\ port$ alebo ich môžeme natrvalo zmeniť v konfiguračnom súbore s názvom $TERATERM$, ktorý sa nachádza v priečinku, kde je nainštalovaný program Tera Term (najčastejšie Program Files->teraterm)

```
;     Serial port parameters
;       Port number
ComPort=1
;       Baud rate
BaudRate=115200
;       Parity (even/odd/none/mark/space)
Parity=none
;       Data (7/8)
DataBit=8
;       Stop (1/1.5/2)
StopBit=1
```

Obrázok 4.15. Úprava konfiguračného súboru TERATERM

4.8.12 Aplikácia Serial plotter

Pri niektorých programoch pri opakovaných meraniach by bolo vhodné zobraziť priebeh daného merania v takmer reálnom čase. Na to môžeme použiť program Serial plotter, ktorý dokáže vyobrazovať až 3 priebehy premenných typu int. Serial plotter možné stiahnuť vo formáte .zip[2]. Po stiahnutí súbor odzipujeme a otvoríme aplikáciu. Následne sa pripojíme na správny COM port, nastavíme parametre komunikácie rovnaké ako v mikrokontroléri.

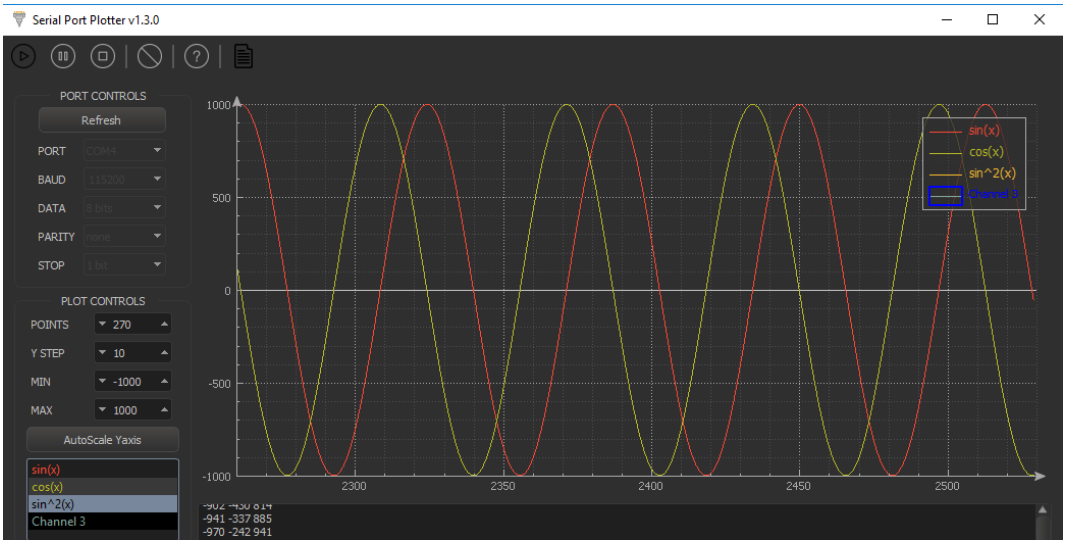
Formát správy

Do sériového portu môžeme posielať iba správy, ktoré majú pevne daný formát. Na začiatku je '$' potom nasledujú 1-3 premenné typu int, oddelené medzerou a na konci správy je ';'.

```
device.printf("\$\%d \%d \%d;", val_1, val_2, val_3); // 3 priebehy
```

V samotnej aplikácii je možné sa pripojiť na existujúci sériový port a upravovať formát odosielanej správy. V hornom paneli je možné začať a ukončiť vykresľovanie, prípadne zmazať vykreslený graf. Na bočnom paneli je možné dvojklikom na daný priebeh zakázať alebo povoliť vykresľovanie, nastaviť maximálny počet vykreslených hodnôt v jednom okne a nastaviť maximálnu a minimálnu hodnotu vykreslenú v okne. Po dvojkliku na legendu grafu je možné zmeniť názvy priebehov a graf uložiť do *.png* súboru.

[1] https://osdn.net/projects/ttssh2/releases/
[2] https://github.com/CieNTi/serial_port_plotter/releases



Obrázok 4.16. Program Serial plotter

4.9 Trieda USBSerial

Niektoré mikrokontroléry dokážu s počítačom komunikovať aj pomocou USB komunikácie. V nasledujúcej kapitole si ukážeme, ako môže mikrokontrolér F042F6P6 komunikovať s počítačom s využitím triedy USBSerial. Aby komunikácia mohla prebiehať, je potrebné mať v PC nainštalovaný VCP driver[1].

4.9.1 Komunikácia na strane PC

Po začatí programu sa vytvorí viruálny COM port, na ktorý sa pripojíme pomocou terminálového programu. Pri každom reštartovaní programu sa spojenie preruší a virtuálny COMport sa vytvorí nanovo. Odporúčame preto použiť program Tera Term, pretože na rozdiel od iných terminálových stačí spojenie nadviazať iba raz a pri reštartovaní programu nemusíme spojenie nanovo inicializovať, ako napríklad v programe Putty. Po spustení programu v mikrokontroléri otvoríme Tera Term a pripojíme sa novo-vytvorený COM port. Keďže sa jedná iba o emuláciu sériovej komunikácie, je jedno, aký baudrate si zvolíme.

4.9.2 Stiahnutie knižnice

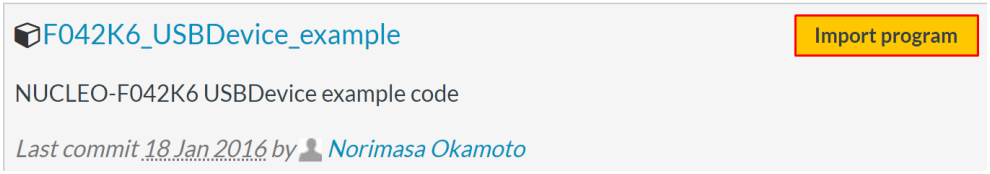
Triedy, ktoré sme doteraz v programoch použili, sú definované v knižnici mbed, ktorú do programu zahŕňame na začiatku každého programu pomocou

```
#include \"mbed.h";
```

Trieda USBSerial je definovaná v samostatne v knižnici F042K6_USBDevice, preto musíme túto knižnicu do projektu vložiť. Najjednoduchší spôsob je stiahnuť priamo ukážkový program pre triedu USBSerial[2].

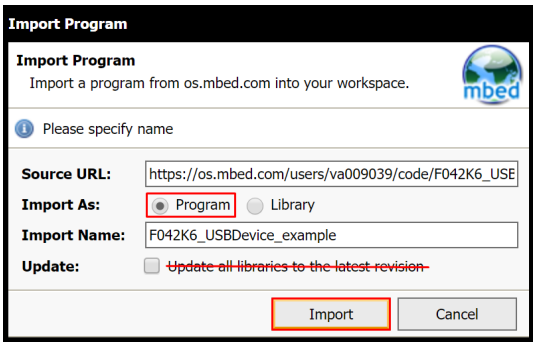
[1] https://embedded.fel.cvut.cz/kurzy/lpe_sw

[2] https://os.mbed.com/users/va009039/code/F042K6_USBDevice/file/6030a12b6c62/USBSerial/USBSerial.h/



Obrázok 4.17. Ukážkový program triedy USBSerial pre NUCLEO-F042K6.

Projekt pomenujeme a označíme ho ako program (obrázok 4.18). Aktualzáciu knižníc nesmieme označiť, pretože knižnice písané v starších verziach mbedu nemusia byť kompatibilné s najnovšími aktualizáciami knižnice mbed.



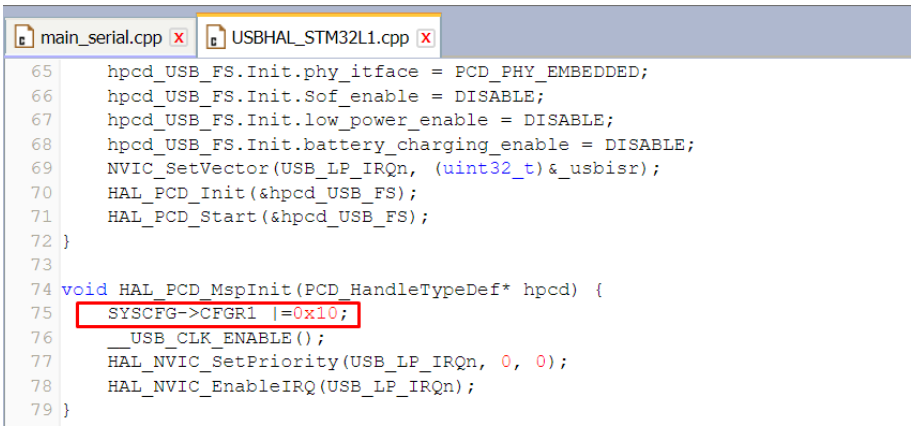
Obrázok 4.18. Importovanie ukážkového programu triedy USBSerial.

Po importovaní zo zložky programu odstránime súbory *main_mouse.cpp*, *main_serial.cpp* a vytvoríme nový súbor($New- > New\ File...$), ktorý pomenujeme *main.cpp*

4.9.3 Úprava knižnice

Keďže importovaná knižnica je určená pre vývojový kit NUCLEO-F042K6 a my používame mikrokontrolér STM32F042F6P6, ich procesory nie sú totožné a preto je potrebné premapovať piny určené ku komunikácii cez USB [4]. Premapovanie pinov prevedieme v samotnej knižnici, konkrétne v súbore $USBHAL\_STM32L1.cpp(F042K6\_USBDevice- > USBDevice- > USBHAL\_STM32L1.cpp)$ V tomto súbore pred riadok 75 pridáme

```
SYSCFG->CFGR1 |=0x10;  // premapovanie pinov USB
```

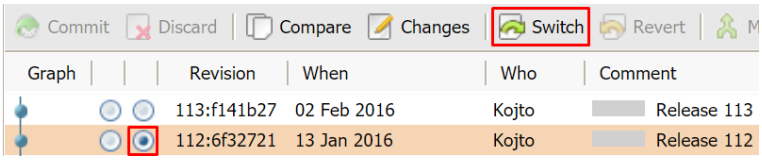


Obrázok 4.19. Úprava knižnice USBSerial.

Takto upravenú knižnicu môžeme používať aj v iných programoch tak, že po kliknutí na ňu pravým tlačidlom ju skopírujeme a vložíme do zložky iného programu.

4.9.4 Downgrade knižnice mbed

Knižnica USBDevice bola písaná za použitia staršej verzie knižnice mbed. Pri použití aktuálnej verzie dochádza k chybe pri kompilácii, preto je potrebné previesť downgrade knižnice mbed na verziu 112. V priečinku programu klikneme pravým tlačidlo na knižnicu mbed, otvoríme *Revisions*, označíme verziu 112 a zmenu potvrdíme kliknutím na *Switch*.



Obrázok 4.20. Downgrade knižnice mbed.

Ak zmena verzie knižnice mbed prebehla úspešne, po obnovení stránky zmení ikona knižnice, ktorá signalizuje, že je možné knižnicu aktualizovať.

4.9.5 Inicializácia virtuálneho sériového portu

Keďže trieda USBSerial je definovaná v samostatne v knižnici F042K6_USBDevice, musíme túto knižnicu na začiatku programu vložiť

```
#include "USBSerial.h" //zahrnutie kniznice F042K6_USBDevice do programu
```

Virtuálny sériový port inicializujeme tak, že vytvoríme objekt triedy USBSerial s nejakým názvom(napr. device).

```
USBSerial device; // vytvorenie virtualneho serioveho portu
```

V počítači sa vytvorí virtuálny COM port a zariadenie sa z pohľadu počítača tvári ako sériový port, aj keď sa jedná o USB komunikáciu. Na začiatku funkcie *main*() musí program čakať aspoň 1 sekundu, kým sa vytvorí spojenie medzi mikrokontrolérom a druhým zariadením, aby nedošlo k strate prvých poslaných znakov.

4.9.6 putc, getc, printf, scanf

V triede USBSerial je možné použiť funkcie *putc*(), *getc*(), *printf*() a *scanf*(), ktoré presmerujú štandardný výstup mikrokontroléru do virtuálneho sériového portu a prijaté znaky na štandardný vstup mikrokontroléru. Ich funkcionalita je popísaná v kapitole 4.8.

4.9.7 available

Počet znakov pripravených na prečítanie z virtuálneho COM portu sa zisťuje pomocou funkcie *available*(). Návratová hodnota funkcie je premenná typu uint8_t, počet prijatých neprečítaných znakov.

```
int number_of_available_signs = device.available();
```

Funkcia *available*() sa používa hlavne v kombinácii s *getc*(), kedy sa skontroluje, či je na vstupe neprečítaný znak, a ak je, prečíta sa.

```
if ( device.available() > 0 ){ // ak je nejaky znak neprecitany
    sign = device.getc();      // sign je premenna typu char
}
```

4.9.8 writeBlock

Pre zápis bloku dát typu uint8_t sa využíva funkcia *writeBlock*(). Jej vstupnými parametrami sú sú:

- premenná typu `uint8_t*`, ukazateľ na pole dát typu `uint8_t`,
- premenná typu `uint16_t`, dĺžka bloku dát, ktoré posielame.

```
uint8_t array[] = "Toto je blok dat \n\r";
device.writeBlock( array, 18 );
```

Návratová hodnota je premenná typu bool, a jej hodnota je:

- true, ak bol blok dát úspešne poslaný,
- false, v prípade neúspešného odoslania.

Ak používame ako blok dát napríklad pole premenných typu char, vo funkcii musíme pretypovať prvý parameter funkcie na ukazateľ na premennú uint8_t.

```
char array[] = "Toto je blok dat \n";
device.writeBlock( (uint8_t *)& array, 18 );
```

4.9.9 attach

To, či do mikrokontroléru bol poslaný nejaký znak je možné kontrolovať softvérovo volaním funkcie *available*() alebo hardvérovo tak, že po prijatí znaku sa vyvolá prerušenie a zavolá sa funkcia obsluhujúca prerušenie. Vstupným parametrom funkcie *attach*() je adresa funkcie, ktorá bude obsluhovať prerušenie, preto sa pred jej názov ešte pridá znak "&". Funkcia obsluhy prerušenia nesmie mať vstupné argumenty a nevracia žiadnu hodnotu. Nesmie obsahovať nekonečné cykly, časovo náročné funkcie (*wait_ms*(), *wait_us*(), *printf*()), ktoré by blokovali beh programu, a zakázané je používať aj funkcie, ktoré nemôžu byť volané viacnásobne(*new*() a *malloc*()), pretože by mohlo dôjsť k nedefinovanému chovaniu programu. Ak funkcia pristupuje ku globálnym premenným, tieto premenné musia byť pri inicializácii označené kľúčovým slovom *volatile*, ktoré zabezpečí, že pri kompilácii programu nedôjde k optimalizácii použitia danej premennej a ku premennej bude možné pristupovať kedykoľvek počas behu programu.

```
void func(){
    ...
}
main(){
    ...
    device.attach(&func); // priradenie obsluhy prerusenia funkcii func
    ...
}
```

Problém funkcie attach

Hlavným využitím funkcie *attach*() je, že vo funkcii prerušenia sa daný znak prečíta pomocou funkcie *getc*(), ale v prípade použitia *getc*() vo funkcii obsluhy prerušenia v triede USBSerial sa beh programu zasekne. Z tohto dôvodu funkcia *attach*() v triede USBSerial prakticky nemá využitie.

4.10 Trieda Timer

Trieda Timer slúži na presné meranie krátkych časových úsekov(do 30 minút).

4.10.1 Inicializácia časovača

Časovač nakonfigurujeme vytvorením objektu triedy Timer s nejakým názvom(napr. stopwatch). V prostredí mbed je možné používať naraz neobmedzený počet objektov triedy Timer [5].

```
Timer stopwatch;
```

4.10.2 start

Volaním funkcie *start*() sa spustí odmeriavanie času. Aj v prípade viacnásobného volania funkcie *start*() sa čas meria od miesta jej prvého výskytu.

```
stopwatch.start(); // spustenie casovaca
```

4.10.3 stop

Volaním funkcie *stop*() za zastaví odmeriavanie času.

```
stopwatch.stop(); // zastavenie casovaca
```

4.10.4 reset

Funkcia *reset*() vynuluje hodnotu časovača. V prípade, že časovač beží a dôjde k resetu, hodnota časovača sa vynuluje a časovač beží ďalej.

```
stopwatch.reset(); // vynulovanie casovaca
```

4.10.5 Prečítanie aktuálnej hodnoty časovača

Časovač pracuje na báze 32-bitového mikrosekundového čítača, teda dokáže merať čas s rozlíšením na mikrosekundy až do

$$(2^{32} - 1)\mu s \approx 35\ min.$$

Aktuálnu hodnotu časovača je možné prečítať pomocou funkcií *read*(), *read_us*() alebo *read_high_resolution_us*(). Pre čo najpresnejšie odmeranie časového úseku sa nepoužíva funkcia *stop*(), pretože vykonanie funkcie trvá určitý čas a nameraný čas s použitím funkcie *stop*() je dlhší (rádovo o jednotky μs), ako bez jej použitia.

read

Funkcia *read*() vráti premennú typu float, aktuálnu hodnotu časovača v sekundách s rozlíšením na mikrosekundy.

```
float timestamp = stopwatch.read();
```

V triede Timer je operátor "=" preťažený a je namapovaný na metódu *read*(), teda jednoduchšie je použiť

```
float timestamp = stopwatch;
```

read_us

Funkcia *read_us*() vráti premennú typu int, aktuálnu hodnotu časovača v mikrosekundách.

```
int timestamp = stopwatch.read_us();
```

Prečítanie hodnoty časovača vo vysokom rozlíšení

Funkcia *read_high_resolution_us*() vráti premennú typu uint64_t, aktuálny čas v mikrosekundách s maximálnou hodnotou

$$(2^{64} - 1)\mu s \approx 580000\ rokov$$

```
uint64_t timestamp = stopwatch.read_us();
```

Kapitola 5
Ladenie programov v prostredí Mbed

5.1 Profesionálne vývojové prostredia

Pri vytváraní programov často dochádza k chybám. Pre ich odstránenie sa používajú rôzne debugovacie metódy. Profesionálne vývojové prostredia, napríklad μVision Keil®5, majú implementované nástroje, ktoré umožňujú debugovať mikrokontrolér v reálnom čase. Beh programu je možné zastavovať pomocou breakpointov, kedy užívateľ dostane informácie o behu programu. Okrem toho umožňujú aj:

- označovanie breakpointov za behu programu,
- vykonanie jedného kroku programu,
- zobrazenie poradia breakpointu, funkcie a čísla riadku, na ktorej sa breakpoint nachádza,
- zobrazenie aktuálnej hodnoty premenných,
- zobrazenie a možnosť zmeny konfiguračných registrov periférií,
- možnosť stiahnuť informácie o breakpointoch do súboru.

Prostredie mbed nemá v sebe implementované ladiace nástroje, z tohto dôvodu je momentálne možné ladiť programy iba systémom *pokus-omyl* alebo s použitím základných tried mbed. V nasledujúcej kapitole sa oboznámime s možnosťami, ako je možné dosiahnuť aspoň čiastočnú funcionalitu profesionálnych debugovacích nástrojov len s použitím tried mbed. Nakoniec budú popísané novo-vytvorené triedy, v ktorých sú implementované ladiace postupy z nasledujúcej kapitoly a ktoré umožnia jednoduché ladenie programov na rôznej úrovni, od jednoduchého krokovania programu pomocou LED a tlačidla až po kompletné výpisy konfigurácií určitých periférií mmikrokontroléra.

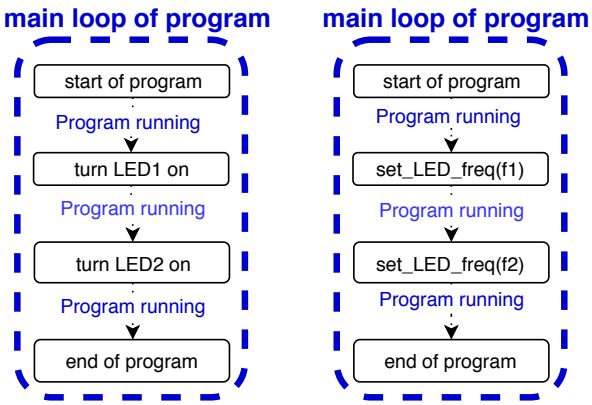
5.2 Metodika ladenia v prostredí mbed

V 70. a 80. rokoch minulého storočia, na počiatku softvérového vývoja v mikrokontroléroch, boli debugovacie nástroje vzácne a pre novo-vzniknuté architektúry neexistovali. Ladenie a debugovanie bolo vykonávané pomocou meracích nástrojov(napr. logický analyzátor) alebo pomocou výstupných súčiatok, najčastejšie LED diód. Mikrokontroléry väčšinou obsahovali niekoľko LED diód, pomocou ktorých bolo možné indikovať miesto, na ktoré program dobehol. Tieto debugovacie techniky sú stále dôležité v situáciach, keď nie je k dispozícii žiaden profesionálny debugovací nástrojc čo je aj náš prípad. Aby sa minimalizovala možnosť výskytu v programe, je veľmi dôležité písať prehľadný, ľahko pochopiteľný kód. Z často opakujúcich sa sekvencií treba vytvoriť samostatné funkcie a ešte počas písania kódu je potrebné doplniť komentár k jednotlvým úkonom v programe. Ďalej je nutné vytvoriť schému zapojenia obvodu a podľa diagramu procesoru overiť, že použité piny mikrokontroléru požadovanú funkcionalitu naozaj ponúkajú. Aj napriek dodržaniu všetkých predošlých pravidiel je veľká pravdepodobnosť, že v programe sa budú vyskytovať chyby. Chybu programu je možné odhaliť tak, že sa program otestuje pri rôznych

podmienkach a pozorované správanie programu sa nezhoduje s očakávaným správaním. Takýto druh testovania sa nazýva black box testing. Na odstránenie chyby je potrebné uistiť sa, či program vôbec beží a ak áno, je potrebné lokalizovať danú chybu.

5.2.1 Indikácia behu programu

Chyba v programe v mnohých prípadoch spôsobí, že počas behu sa program zacyklí alebo "spadne". Chyba pri nahrávaní programu do mikrokontroléra môže spôsobiť, že program sa ani nespustí. Prvou požiadavkou je preto indikácia, že počas behu program dobehol na určité miesto. Jednoduchý spôsob je použiť jednu alebo viacero LED diód. Po dobehnutí na dané miesto sa LED rozsvieti alebo začne blikať s určitou frekvenciou. Ak napríklad použijeme 3 LED, takýmto spôsobom je možné postupne skontrolovať 3 miesta, či na ne program dobehol. Ak je k dispozícii pin s funkcionalitou generovania PWM, rovnaký výsledok je možné dosiahnuť použitím jednej LED zmenou frekvencie blikania pri stálej striede PWM signálu. Pri použití viacerých LED by bolo možné binárne zobraziť číslo breakpointu, na ktorý program dobehol. Pri použití 3 LED by bolo možné odlíšiť až 8 breakpointov. Zásadná nevýhoda takéhoto prístupu je, že pri vetvení programu alebo pri použití cyklov nevieme presne identifikovať miesto breakpointu, preto na sofistikovaniejšie ladenie programov je potrebné iné metódy.



Obrázok 5.1. Kontrola behu programu pomocou LED.

5.2.2 Krokovanie programu pomocou tlačidla a LED

Predstavme si nasledujúcu úlohu: Máme roztočiť unipolárny krokový krokový motor, ale nevieme rozlíšiť jednotlivé cievky motora. Aby sme mohli roztočiť unipolárny krokový motor, musíme na jednotlivé cievky priviesť napäťové pulzy v správnom poradí 5.1, v opačnom prípade sa motor neroztočí.

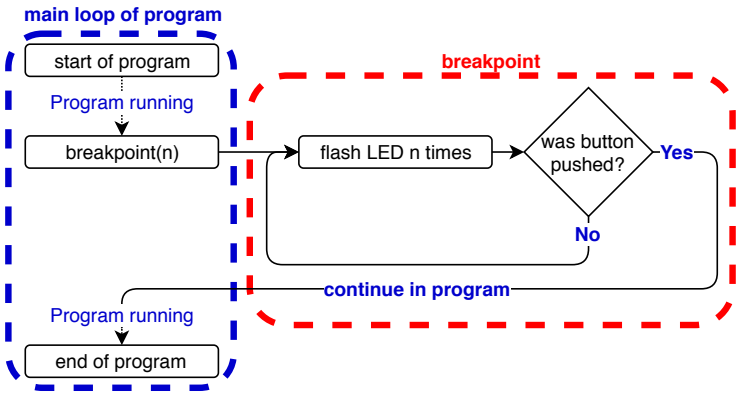
cievka	a	b	c	d
1. krok	1	0	0	0
2. krok	0	1	0	0
3. krok	0	0	1	0
4. krok	0	0	0	1

Tabuľka 5.1. Séria krokov na roztočenie unipolárneho krokového motora.

```
#include mbed.h
DigitalOut coil_a(PA_0, 0);
DigitalOut coil_b(PA_1, 0);
```

```
DigitalOut coil_c(PA_2, 0);
DigitalOut coil_d(PA_3, 0);
int main(){
    while(1){
        a  =  1;
        d = 0;
        wait_ms(10);
        //breakpoint
        b = 1;
        a = 0;
        wait_ms(10);
        //breakpoint
        c = 1;
        b = 0;
        wait_ms(10);
        //breakpoint
        d = 1;
        c = 0;
        wait_ms(10);
        //breakpoint
    }
}
```

Z predošle zmienených dôvodov prirodzene vychádza ďalšia požiadavka, ktorou je možnosť pozastaviť beh hlavného cyklu programu, aby bolo možné aj odkrokovať aj rýchle sekvencie, ktoré nedokážeme iným spôsobom odsledovať. Takým prístupom by bolo možné prísť na to, či jednotlivé cievky motora sú dobre zapojené a ak nie sú, v ktorom kroku došlo k chybe. Jednoduchý prípad realizácie spomenutého prístupu je použitie tlačidla v kombinácii s indikačnou LED diódou. Medzi jednotlivými krokmi krokového motora sa pozastaví hlavný cyklus programu, LED dióda blikaním indikuje, že program dobehol do breakpointu a čaká sa na stlačenie tlačidla, po ktorom hlavný cyklus programu pokračuje. Takýmto spôsobom je možné program krok za krokom a správne zapojiť vývody motora.

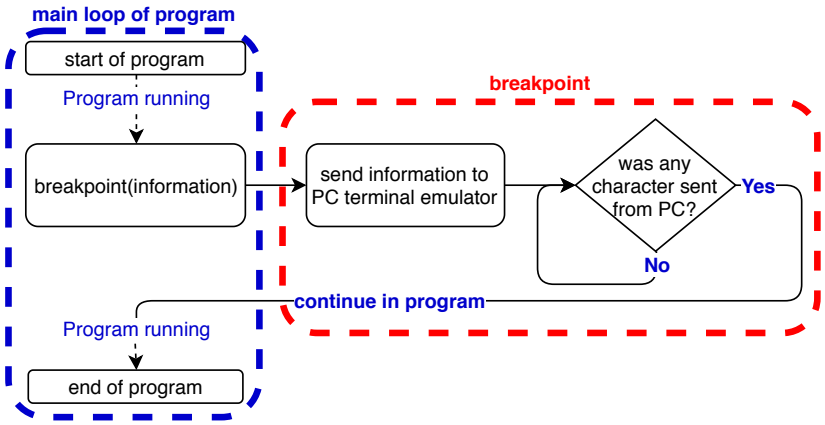


Obrázok 5.2. Krokovanie programu pomocou tlačidla a LED.

Je však potrebné brať do úvahy, že krokovacie meetódy spôsobujú zmeny správania programu, nie je možné testovať správne načasovanie, komunikačné systémy sa môžu dostať do timeout stavu a podobne [11]. Pri použití tlačidla musíme taktiež brať do úvahy "bouncing", ktorý je bližšie popísaný v kapitole 4.3.5, a softvérovo ho potlačiť, čo značne zneprehľadňuje program. Pri reálnom použití je pre správne pokračovanie programu po stlačení tlačidla potrebné počkať aj na pustenie tlačidla, pretože pri jednom stlačení by

inak mohlo dôjsť k preskočeniu niekoľkých breakpointov naraz. Z týchto dôvodov bola vytvorená trieda Debug_led, ktorá je popísaná v kapitole 6.1.

5.2.3 Ladenie programov s využitím výpisov prostredníctvom kanálu UART



Obrázok 5.3. Krokovanie programu s využitím sériového portu.

Ďalšou možnosťou ladenia je použiť PC na komunikáciu s mikrokontrolérom. Po dobehnutí na miesto breakpointu by boli do PC odoslané informácie o behu programu a podľa nich by mohol užívateľ odosielať do mikrokontroléra znaky, ktoré by ovplyvňovali beh programu, hodnotu premenných alebo registrov. V prípade, že má mikrokontrolér voľný UART, je možné do PC posielať takmer neobmedzené množstvo dát v užívateľsky príjemnej forme a rovnako aj z PC prijímať znaky a formátované reťazce. Rýchlosť komunikácie vždy volíme najväčšiu možnú: 115200 Bd/s, aby odosielanie a prijímanie znakov trvalo najkratšiu možnú dobu a tým došlo k čo najmenšiemu ovplyvneniu behu programu.
V nasledujúcich ukážkach budeme pracovať s objektom triedy Serial s názvom pc.

```
Serial pc(PA_2, PA_3, 115200); //v poradi tx a rx pin, baudrate
```

Odoslanie polohy breakpointu

Pre jednoznačnú identifikáciu breakpointu je potrebné do PC odoslať číslo riadku, kde sa program nachádza a takisto aj poradové číslo breakpointu, aby v prípade použitia breakpointu v cykle bolo možné určiť, v ktorej iterácii sa program aktuálne nechádza. Ako poradové číslo stačí inkrementovať hodnotu jednej premennej typu int od nuly pri každom výskyte breakpointu.
Pre univerzálne zobrazenie čísla riadku sa používa makro $\_\_LINE\_\_$ definované v mbede, ktoré sa pri preklade kódu automaticky nahradí premennou typu int, číslom riadku, na ktorom sa makro nachádza.

```
pc.printf("\%d",__LINE__);
```

Odoslanie premennej a jej názvu

Hlavnou výhodou UARTu v porovnaní s predošlými metódami debugovania, je možnosť odosielať a prijímať formátovaný reťazec, teda aj rôzne premenné. Zapísať reťazec do sériového portu je možné funkciou *printf*() pomocou formátovacích direktív popísaných v kapitole 4.8.8. S hodnotou premennej do sériového portu chceme väčšinou odoslať aj jej názov, pretože pri väčšom počte premenných by bol problém rozoznať jednotlivé premenné. Tento problém rieši makro (napr. `name(var)`) s jedným argumentom, premennou (napr. var), ktoré vráti reťazec s názvom danej premennej.

```
#define name(var) #var
```

V nasledujúcej ukážke je do sériového portu odoslaný názov a hodnota premennej typu float .

```
float pi = 3.14;
pc.printf("float \%s = \%f",name(pi), pi);
```

Prečítanie hodnoty registra na určitej adrese

Konfigurácie jednotlivých periférií sú uložené v 4-bajtových(32-bitových) registroch v adresnom priestore procesora. Každý register má priradenú adresu, ak chceme prečítať hodnotu určitého registra, musíme prečítať 4-bajtové číslo z adresy daného registra. Každá adresa, na ktorú pristupujeme musí byť deliteľná 4-mi, v opačnom prípade by sme sa pokúšali čítať z dvoch registrov naraz a došlo by k chybe. Na niektoré adresy nie je možné pristupovať a čítanie niektorých registrov môže zmazať hodnotu hodnotu takzvaných flag bitov registra, ktoré sa nastavujú hardvérovo po vykonaní nejakej operácie a čítaním daného registra sa ich hodnota zmaže. Z týchto dôvodov pri pristupovaní k registrom na určitých adresách môže spôsobiť prerušenie sériovej komunikácie alebo Hard fault programu.

```
uint32_t address;
int num_args = pc.scanf("\%u", &address);
if (num_args == 1){
    uint32_t reg_value = *((volatile unsigned int *)address);
    pc.printf("register value on address \%u is \%u", address, reg_value);
}
```

Zmena hodnoty registra

V prípade, že chceme zmeniť hodnotu určitého registra, užívateľ musí do mikrokontroléra odoslať dve 32-bitové čísla, adresu registra a požadovanú hodnotu registra. Na to je možné použiť funkciu *scanf*(), ktorá prijíma znaky do tej doby, kým prijatý zank neporušený očakávaný formát správy. Návratová hodnota funkcie *scanf*() je počet úspešne priradených hodnôt premenným, v tomto prípade sa priraďujú 2 hodnoty premenným, takže návratová hodnota pri úspešnom priradení je 2. Pri zmene hodnoty registra je potrebné riadiť sa manuálom k danému procesoru, aby nedošlo k chybe programu.

```
uint32_t address;
uint32_t reg_value;
int num_args = pc.scanf("\%u \%u", &address, &reg_values);
if (num_args == 2){
    *((volatile unsigned int *)address) = reg_value; //priradenie hodnoty
    pc.printf("value of register on address \%u is \%u", address,
                                *((volatile unsigned int *)address));
}
```

Ukončenie breakpointu

Pre ukočenie breakpointu musí užívateľ typicky poslať do sériového portu nejaký znak. V prípade, že chceme program iba krokovať, stačí použiť funkciu *getc*(), ktorá čaká na prijatie ľubovoľného znaku a po prijatí program beží ďalej.

```
pc.getc();
```

V prípade, že niektoré klávesy sú už používané na iné účely, je potrebné si iba vybrať konkrétny znak (napr. Enter, ASCII = 13), po ktorého odoslaní do mikrokontroléru sa breakpoint ukončí a program beží ďalej.

Kapitola 6
Knižnica DEBUG UNIVERSAL

Pre jednoduché ladenie programov v procesoroch, ktoré môžu byť programované v mbede(mbed enabled), bola vytvorená knižnica DEBUG_UNIVERSAL, v ktorej sú definované triedy s rôznym prístupom k ladeniu a s rôznymi pamäťovými požiadavkami. Niektoré triedy sú implementované tak, aby neprerušovali beh programu, ktoré by mohlo zmeniť správanie a časovanie programov. Iné triedy, práve naopak, slúžia na odkrokovanie programov a umožňujú zmeniť konfiguračné registre procesora. Hlavnou výhodou použitia tried je jednoduchá konfigurácia periférií vytvorením objektu danej triedy a rovnako aj použitie breakpointov volaním funkcie *breakpoint*() definovaných v jednotlivých triedach. V triedach, ktoré používajú sériový port, sú pri vypisovaní použité ANSI escape sekvencie pre zobrazenie informácií v užívateľsky príjemnej podobe. Okno terminálového programu nefunguje v rolovacom režime, ako sme zvyknutí, ale jednotlivé informácie sa prepisujú a užívateľ má lepší prehľad o stave programu.

Použitá sekvencia	popis
\e[1m	Turn bold mode on
\e[22m	Turn bold and faint mode off
\e[Line;ColumnH	Move cursor to screen location v,h
\e[f	Move cursor to upper left corner
\e[K	Clear line from cursor right
\ee	Reset terminal to initial state
\e[B;Fm	Set background(B) and font(F) color
\e[s	Save cursor position
\e[u	Restore cursor position

Tabuľka 6.1. Použité ANSI escape sekvencie z [12].

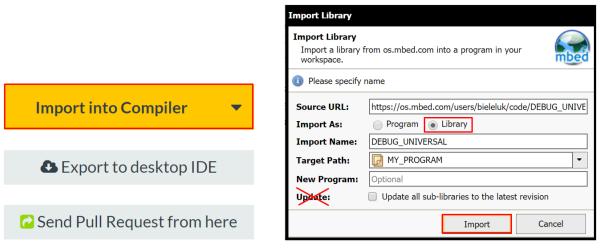
6.0.1 Triedy knižnice DEBUG UNIVERSAL

- Debug_led - slúži na krokovanie programu pomocou LED a tlačidla.
- Debug_serial - slúži na krokovanie programu pomocou sériového portu a na zobrazenie hodnoty jednej premennej.
- Debug_register - je určená na krokovanie programov pomocou sériového portu a na zobrazovanie a zmenu hodnoty konfiguračných registrov.
- Debug_register_print - umožňuje výpis konfiguračných registrov v definovanom formáte za behu programu bez zastavovania behu programu s možnosťou uloženia výpisov do súboru.

6.0.2 Importovanie knižnice do programu

Knižnica je voľne dostupná na stránke:
`https://os.mbed.com/users/bieleluk/code/DEBUG_UNIVERSAL/`
Pre importovanie knižnice do programu na pravej strane okna klikneme na

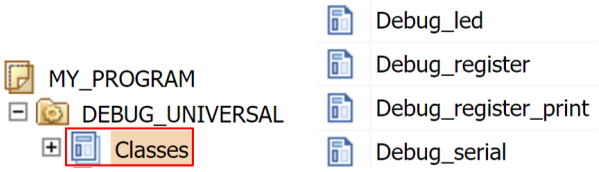
Import into Compiler, následne vyberieme cieľový program a knižnicu importujeme. V mbede všeobecne nesmieme povoliť update knižnice pri importovaní, pretože daná knižnica môže byť vytvorená v staršej verzii mbed a v novšej verzii by mohlo dôjsť k chybe pri kompilácii programu. V súčasnej dobe je knižnica v štádiu testovania a preto dochádza k odstraňovaniu chýb v implementácii.



Obrázok 6.1. Importovanie knižnice DEBUG_UNIVERSAL do programu

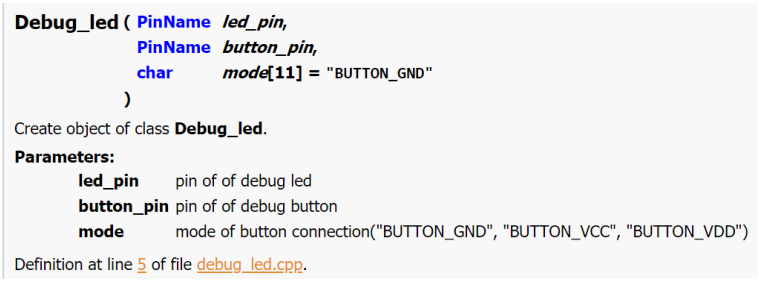
6.0.3 Dokumentácia knižnice

Po rozkliknutí zložky knižnice môžeme v priečinku *Classes* zobraziť dokumentáciu *Doxygen* jednotlivých tried.



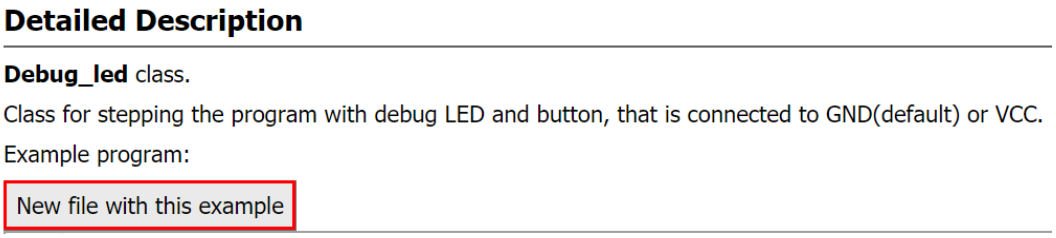
Obrázok 6.2. Dokumentácia knižnice DEBUG_UNIVERSAL

Dokumentácia každej triedy obsahuje detailný popis použitia triedy, popisy jednotlivých funkcií, vstupných a návratových argumentov jednotlivých funkcií.



Obrázok 6.3. Popis funkcií jednotlivých tried v knižnici DEBUG_UNIVERSAL

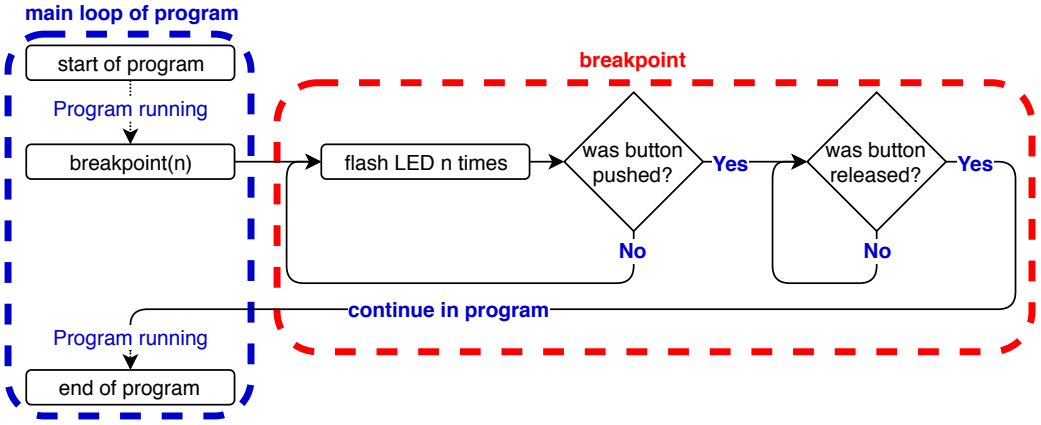
Okrem toho obsahuje program, ktorý ukazuje možnosti danej triedy a je možné zahrnúť ho do programu jedným klikom. Tento program je funkčný pre procesory STM32F042F6P6 a NUCLEO-F303RE, pri ostatných zariadeniach je potrebné skontrolovať, či obsahuje použité piny a či dané piny majú požadovanú funkcionalitu.



Obrázok 6.4. Ukážkový program v jednotlivých triedach knižnice DEBUG_UNIVERSAL

6.1 Trieda Debug led

Trieda Debug_led slúži na krokovanie programov v procesoroch STM32 pomocou LED diódy a tlačidla. V triede je definovaná funkcia *breakpoint*(), ktorá preruší beh programu v hlavnom cykle, LED dióda definovaným periodickým blikaním indikuje, na ktoré miesto program dobehol a následne čaká na stlačenie debugovacieho tlačidla. Po stlačení a následnom pustení tlačidla LED dióda prestane blikať a program beží až do miesta ďalšieho breakpintu. Pre použitie triedy Debug_led je potrebná LED dióda, k nej rezistor do série a tlačidlo pripojené na zem (resp. napájacie napätie procesoru). K tlačidlu sa nemusí pripájať pull-up (resp. pul-down) rezistor, pretože pin, ku ktorému je tlačidlo pripojené, je nakonfigurovný ako vnútorný pull-up (resp. pull-down).



Obrázok 6.5. Štruktúra breakpointu v triede Debug_led

6.1.1 Vytvorenie objektu

Piny procesoru nakonfigurujeme v poradí ako debugovaciu LED (napr. PA5) a debugovacie tlačidlo (napr. PA4) tak, že si vytvoríme objekt triedy Debug_led s nejakým názvom (napr. debug) a priradíme mu tieo piny. Prvé dva parametre sú povinné, sú to názvy pinov v poradí debugovacej LED(PA5) a debugovacieho tlačidla(PA4). Tretí, nepovinný parameter, je reťazec popisujúci zapojenie tlačidla. Ak sa parameter nepoužije, tlačidlo je predvolene pripojené na zem.

Tlačidlo pripojené na zem (GND)

Ak je tlačidlo pripojené na zem, nemusíme použiť tretí parameter, alebo použijeme reťazec: “BUTTON_GND”

```
Debug_led debug (PA_4, PA_5); /* v poradi debug_led a debug_button,
                                    tlacidlo je pripojene na GND */
Debug_led debug (PA_4, PA_5, "BUTTON_TO_GND");
```

Tlačidlo pripojené na napájacie napätie mikroprocesoru(VDD)

Ak je tlačidlo pripojené na VDD, ako tretí parameter použijeme jeden z nasledujúcich reťazcov: “BUTTON_VDD”,“BUTTON_VCC”.

```
 Debug_led debug (PA_4, PA_5, "BUTTON_TO_VDD"); /* v poradi debug_led a
                              debug_button,tlacidlo je pripojene na VDD */
Debug_led debug (PA_4, PA_5, "BUTTON_TO_VCC");
```

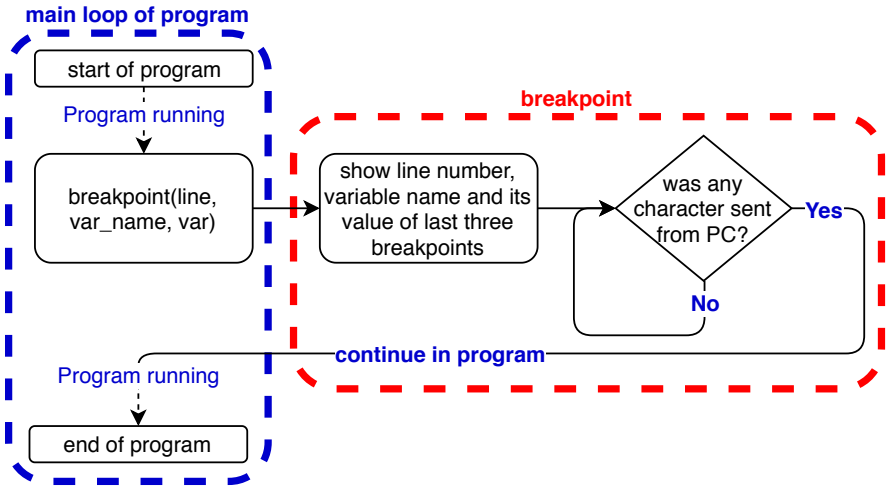
6.1.2 breakpoint

Funkcia *breakpoint*() zastaví beh hlavného cyklu programu a debugovacia LED začne periodicky blikať podľa zadaného parametru. Program ďalej čaká na stlačenie a následné pustenie debugovacieho tlačidla, po ktorom beží až do miesta ďalšieho brakpointu. Počas behu programu LED nesvieti.
Voliteľným vstupným parametrom je premenná typu int, ktorá udáva periodický počet bliknutí LED po prerušení behu programu, podľa ktorého je možné identifikovať miesto breakpointu. Bez použitia parametru bliká LED konštantne.

```
debug.breakpoint(); /* debugovacia LED bude periodicky blikat,
                       kym nedojde k stlaceniu a pusteniu tlacidla */
debug.breakpoint(3); /* debug. LED bude blikat periodicky po 3 razy az do
                        momentu stlacenia a nasledneho pustenia tlacidla */
```

6.2 Trieda Debug serial

Trieda Debug_serial je možné použiť na krokovanie programov pre procesory STM32 pomocou komunikácie s počítačom cez sériový port a umožňuje vypísanie aktuálnej hodnoty jednej premennej alebo registra v mieste breakpointu. V triede je definovaná funkcia *breakpoint*(),ktorá preruší beh programu v hlavnom cykle a do sériového portu odošle informácie o posledných troch breakpointoch: poradové číslo breakpointu, číslo riadku, na ktorom sa funkcia nachádza, a hodnotu jednej ľubovoľnej premennej typu *int*, *float*, *char* alebo *char*∗. Následne program čaká na prijatie ľubovoľného znaku zo sériového portu. Po prijatí znaku program beží až do miesta ďalšieho breakpintu. Debug_serial má aj čiastočnú funkcionalitu triedy Serial (kapitola 4.8), pretože sú v nej definované aj funkcie *printf*(), *scanf*(), *putc*(), *getc*(), *readable*() a *writable*(). Okno terminálového programu je vizuálne rozdelené na dve časti: debugovaciu časť na zobrazenie informácií o breakpointoch a sériový port na odosielanie a prijímanie správ mimo funkcie *breakpoint*(). Pre rozdelenie okna na dve časti a zobrazenie informácii v užívateľsky príjemnom formáte boli použité ANSI Escape sekvencie. Znaky sa správne vykresľujú iba do momentu kedy dôjde k "pretečeniu" okna, peto okno terminálového programu musí byť vo full screen režime a vypisovanie do sériového portu treba používať ohľadom na dané obmedzenie.
Pre použitie triedy Debug_serial je potrebný prevodník UART-USB na komunikáciu mikrokontroléra s počítačom.



Obrázok 6.6. Štruktúra breakpointu v triede Debug_serial

6.2.1 Vytvorenie objektu

Piny procesoru nakonfigurujeme v poradí ako TX pin (napr. PA2) a RX pin (napr. PA3) debugovacieho sériového portu tak, že vytvoríme objekt triedy Debug_serial s nejakým názvom (napr. pc) a priradíme mu tieto piny. Prvé dva parametre sú povinné, sú to názvy pinov v poradí TX pinu (PA2) a RX pinu (PA3) debugovacieho sériového portu. Tretí, nepovinný parameter, je premenná typu int, rýchlosť sériovej komunikácie (baudrate). Ak sa parameter nepoužije, predvolená hodnota baudrate je 115200 Bd/s.

```
Debug_serial pc (PA_2, PA_3); //tx a rx pin, baudrate je 115200 Bd/s
Debug_serial pc (PA_2, PA_3, 57600);//tx a rx pin, baudrate: 57600 Bd/s
```

6.2.2 breakpoint

Funkcia breakpoint zastaví beh hlavného cyklu programu, zobrazí informáciu o posledných troch breakpointoch, voliteľne aj hodnotu jednej ľubovoľnej premennej typu *int*, *float*, *char*, *char** (reťazec) alebo hodnoty jedného registra a čaká na prijatie ľubovoľného znaku zo sériového portu. Po prijatí znaku program beží až do miesta ďalšieho breakpintu.

Breakpoint bez zobrazenia premennej

Ak chceme program iba krokovať, bez vypisovania premenných, použijeme funkciu *breakpoint*() s jedným parametrom, ktorým je je premenná typu int, číslo riadku, na ktorom sa breakpoint nachádza. Odporúča sa použiť makro *__LINE__* definované v mbede, ktorá sa pri preklade programu automaticky nahradí číslom riadku, v ktorom sa nachádza.

```
pc.breakpoint(__LINE__);
```

Breakpoint so zobrazením premennej typu int, float, char alebo char *

Prvým parametrom je premenná typu int, číslo riadku, na ktorom sa breakpoint nachádza(*__LINE__*). Druhým parametrom je reťazec s maximálnou dĺžkou 20 znakov, názov premennej (napr. var), ktorej hodnota sa v breakpointe zobrazí. Odporúča sa použiť makro *name*(*var*) definované v knižnici Debug_universal, ktoré sa pri preklade programu automaticky nahradí názvom premennej (napr. `var`). Tretím parametrom je samotná premenná (napr. var) typu *int*, *float*, *char* alebo *char** (reťazec).

```
float pi = 3.141;          // premenna typu float
char one_character =  'A'; // premenna typu char
int var_int = 10;    // premenna typu int
char my_string[] = "toto je retazec"; // premenna typu char* (string)
pc.breakpoint(__LINE__, name(pi), pi); // breakpoint so zobrazenim float-u
pc.breakpoint(__LINE__, name(var_int), var_int); // zobrazenie int-u
pc.breakpoint(__LINE__, name(my_string), my_string); /* breakpoint so
                                                    zobrazenim retazca */
pc.breakpoint(__LINE__, name(one_character), one_character); /* breakpoint
                                                     so zobrazenim znaku */
```

Breakpoint so zobrazením hodnoty jedného registra

Prvým parametrom je premenná typu int, číslo riadku, na ktorom sa breakpoint nachádza(*__LINE__*). Druhým parametrom je premenná typu uint32_t, adresa registra, ktorého hodnota sa prečíta, viac informácií o konfiguračných registroch v kapitole 10.

```
pc.breakpoint(__LINE__, 0x48000000 ); /* breakpoint so zobrazenim registra
                                                    na adrese 0x48000000 */
```

6.2.3 printf, scanf, putc, getc, readable, writable

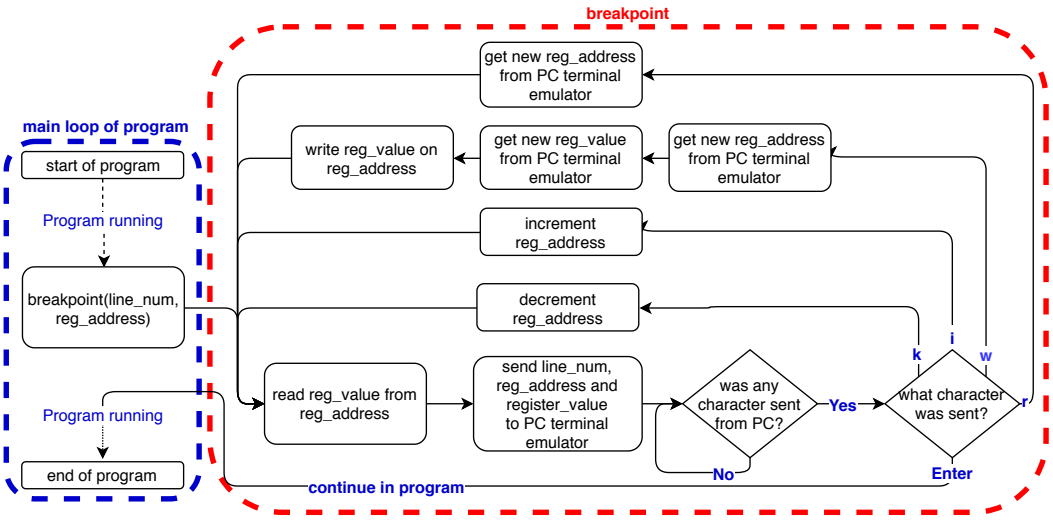
Funkcie ponúkajú čiastočnú funkcionalitu triedy Serial a sú popísané v kapitole 4.8.

6.3 Debug register

Trieda Debug_register je určená na krokovanie programov pre procesory STM32, na výpis miesta breakpointu a hodnoty registrov do sériového portu v mieste breakpointu a na zmenu hodnoty registrov v mieste breakpointov. V triede je definovaná funkcia *breakpoint*(), ktorá preruší beh programu v hlavnom cykle, do sériového portu odošle informácie o breakpointe: poradové číslo, číslo riadku, na ktorom sa breakpoint nachádza, a hodnotu jedného registra. Následne program čaká na prijatie znaku a podľa neho može:

- zobraziť hodnotu susedného registra a čakať na prijatie ďalšieho znaku z PC,
- zobraziť hodnotu registra z ľubovoľnej adresy a čakať na prijatie ďalšieho znaku z PC,
- zapísať hodnotu ľubovoľného registra a čakať na prijatie ďalšieho znaku z PC,
- ukončiť breakpoint.

Debug_serial má aj čiastočnú funkcionalitu triedy Serial (kapitola 4.8), pretože sú v nej definované aj funkcie *printf*(), *scanf*(), *putc*(), *getc*(), *readable*() a *writable*(). Okno terminálového programu je vizuálne rozdelené na dve časti: debugovaciu časť na zobrazenie informácií o breakpointoch a sériový port na odosielanie a prijímanie správ mimo funkcie *breakpoint*(). Pre rozdelenie okna na dve časti a zobrazenie informácii v užívateľsky príjemnom formáte boli použité ANSI Escape sekvencie. Znaky sa správne vykresľujú iba do momentu kedy dôjde k "pretečeniu" okna, peto okno terminálového programu musí byť vo full screen režime a vypisovanie do sériového portu treba používať ohľadom na dané obmedzenie.
Pre použitie triedy Debug_serial je potrebný prevodník UART-USB na komunikáciu mikrokontroléra s počítačom.



Obrázok 6.7. Štruktúra breakpointu v triede Debug_register

6.3.1 Vytvorenie objektu

Piny procesoru nakonfigurujeme v poradí ako TX pin (napr. PA2) a RX pin (napr. PA3) debugovacieho sériového portu tak, že vytvoríme objekt triedy Debug_register s nejakým názvom (napr. pc) a priradíme mu tieto piny. Prvé dva parametre sú povinné, sú to názvy

pinov v poradí TX pinu (PA2) a RX pinu (PA3) debugovacieho sériového portu. Tretí, nepovinný parameter, je premenná typu int, rýchlosť sériovej komunikácie (baudrate). Ak sa parameter nepoužije, predvolená hodnota baudrate je 115200 Bd/s.

```
Debug_register pc (PA_2, PA_3); // tx a rx pin, baudrate je 115200 Bd/s
Debug_register pc (PA_2, PA_3, 57600); //tx a rx pin, baudrate: 57600 Bd/s
```

6.3.2 breakpoint

Funkcia *breakpoint*() zastaví beh hlavného cyklu programu a od sériového portu zapíše poradové číslo breakpointu, číslo riadku, na ktorom sa funkcia volá, a hodnotu jedného registra. Funkcia má 2 parametre, prvým je premenná typu int, číslo riadku, na ktorom sa breakpoint nachádza(*__LINE__*).
Druhý parameter je premenná typu uint32_t, adresa registra, ktorého hodnota sa prečíta. Procesory STM32 majú 4-bajtové(32-bitové) registre, to znamená, že adresa registra musí byť deliteľná 4-mi. V prípade,že nie je, adresa sa zaokrúhli smerom dole na najbližšie číslo deliteľné 4-mi. Na niektoré adresy nie je možné pristupovať a čítanie niektorých registrov môže zmazať hodnotu hodnotu takzvaných flag bitov registra, ktoré sa nastavujú hardvérovo po vykonaní nejakej operácie a čítaním daného registra sa ich hodnota zmaže.
Z týchto dôvodov pri pristupovaní k registrom na určitých adresách môže spôsobiť prerušenie sériovej komunikácie alebo Hard fault programu.
Následne program čaká na prijatie znaku a podľa jeho hodnoty:

- 'i' – prečíta a odošle hodnotu susedného registra na vyššej adrese a opäť čaká na prijatie znaku,
- 'k' - prečíta a odošle hodnotu susedného registra na nižšej adrese a opäť čaká na prijatie znaku,
- 'r' – užívateľ zadá hexadecimálne adresu, z ktorej chce prečítať hodnotu registra, pri zadávaní adresy užívateľ prepisuje jednotlivé číslice naposledy zobrazenej adresy, kurzor sa posúva pomocou znakov 'j' a 'l', v prípade chyby užívateľa je možné pomocou klávesy *esc* vymazať zadanú hodnotu. Následne užívateľ stlačí *Enter*(prípadne 'p'), funkcia prečíta a zobrazí hodnotu daného registra a opäť čaká na prijatie znaku,
- ' w' – užívateľ zadá hexadecimálne adresu, na ktorej chce zmeniť hodnotu registra, pri zadávaní adresy užívateľ prepisuje jednotlivé číslice naposledy zobrazenej adresy, kurzor sa posúva pomocou znakov 'j' a 'l', v prípade chyby užívateľa je možné pomocou klávesy *esc* vymazať zadanú hodnotu. Následne užívateľ stlačí *Enter*(prípadne 'p'), to isté zopakuje pre nastavenie požadovanej hodnoty registra a po druhom stlačení *Enter*(prípadne 'p') sa prepíše hodnota registra na zadanej adrese na požadovanú hodnotu. Nakoniec sa prečíta hodnota registra na zadanej adrese, zobrazí sa a opäť sa čaká na prijatie znaku.
- Enter – ukončí sa breakpoint a program v hlavnom cykle pokračuje.

```
pc.breakpoint(__LINE__,0x48000008); //register na adrese 0x48000008
```

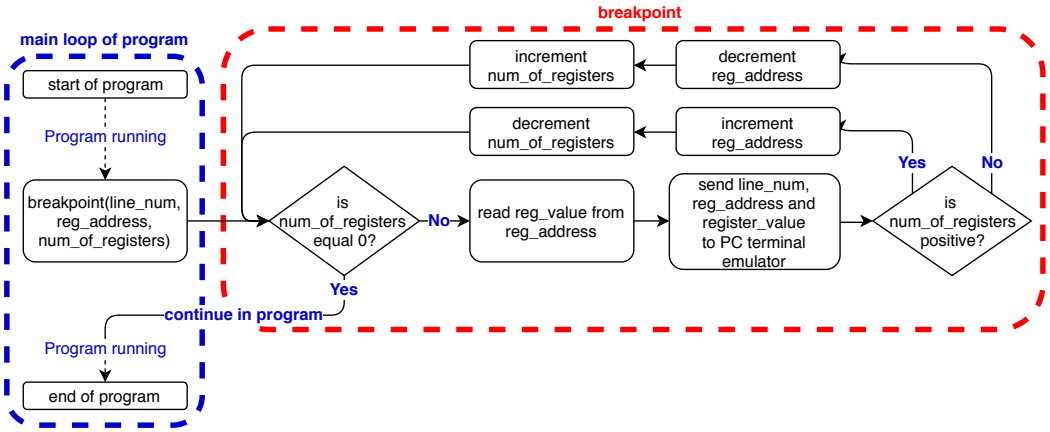
6.3.3 printf, scanf, putc, getc, readable, writable

Funkcie ponúkajú čiastočnú funkcionalitu triedy Serial a sú popísané v kapitole 4.8.

6.4 Debug register print

Trieda Debug_register_print, na rozdiel od triedy Debug_register, nekrokuje program, ale za behu programu v určitom formáte vypisuje hodnotu jedného alebo viacerých po sebe

nasledujúcich registrov. Z tohto dôvodu je vhodná aj na debugovanie programov kritických na časovanie, pretože, na rozdiel od krokovacích tried, funkcia *breakpoint*() pozastaví beh programu iba na nevyhnutný čas(rádovo jednotky ms) na odoslanie informácie o polohe breakpointu a hodnoty registra do debugovacieho sériového portu.
Pre použitie triedy Debug_serial_print je potrebný prevodník UART-USB na komunikáciu mikrokontroléra s počítačom.



Obrázok 6.8. Štruktúra breakpointu v triede Debug_register_print

6.4.1 Vytvorenie objektu

Piny procesoru nakonfigurujeme v poradí ako TX pin (napr. PA2) a RX pin (napr. PA3) debugovacieho sériového portu tak, že vytvoríme objekt triedy Debug_register_print s nejakým názvom (napr. device) a priradíme mu tieto piny.
Prvé dva parametre sú povinné, sú to názvy pinov v poradí TX pinu (PA2) a RX pinu (PA3) debugovacieho sériového portu.
Tretí, nepovinný parameter, je premenná typu int, rýchlosť sériovej komunikácie (baudrate). Ak sa parameter nepoužije, predvolená hodnota baudrate je 115200 Bd/s.

```
Debug_register_print device(PA_2, PA_3);//tx a rx pin, baudrate:115200 Bd/s
Debug_register_print device(PA_2, PA_3, 57600); //tx a rx pin, baudrate
```

6.4.2 format

Funkcia *format*() nastavuje formát správy odosielanej do sériového portu pri breakpointe. Má 4 voliteľné parametre, premenné typu int, ktoré, určujú, v akom tvare sa do sériového portu odšlú v poradí: poradové číslo breakpointu, číslo riadku, na ktorom sa breakpoint nachádza, adresa registra, hodnota registra. Ak je hodnota daného parametra

- 0, informácia sa neodosiela,
- 1, daná hodnota sa odošle v hexadecimálnom tvare,
- 2, daná hodnota sa odošle v decimálnom tvare,
- 3, daná hodnota sa odošle v binárnom tvare (možné použiť iba pre hodnotu registra).

Predvolene sa poradové číslo breakpointu a číslo riadku odosielajú v decimálnom tvare, adresa registra v hexadecimálnom tvare a hodnota registra v binárnom tvare.

```
device.format(2, 2, 1, 1); /*cislo breakpointu a riadok decimalne,
                             adresa s hodnotou registra hexadecimalne */
```

6.4.3 breakpoint

Funkcia *breakpoint*() odošle jednu alebo niekoľko správ do sériového portu vo formáte danom funkciou *format*() a program v hlavnom cykle beží ďalej. Funkcia má 3 parametre, prvým je premenná typu int, číslo riadku, na ktorom sa breakpoint nachádza(*__LINE__*). Druhý parameter je premenná typu uint32_t, adresa registra, ktorého hodnota sa prečíta ako prvá. Procesory STM32 majú 4-bajtové(32-bitové) registre, to znamená, že adresa registra musí byť deliteľná 4-mi. V prípade,že nie je, adresa sa zaokrúhli smerom dole na najbližšie číslo deliteľné 4-mi. Na niektoré adresy nie je možné pristupovať a čítanie niektorých registrov môže zmazať hodnotu hodnotu takzvaných flag bitov registra, ktoré sa nastavujú hardvérovo po vykonaní nejakej operácie a čítaním daného registra sa ich hodnota zmaže. Z týchto dôvodov pri pristupovaní k registrom na určitých adresách môže spôsobiť prerušenie sériovej komunikácie alebo Hard fault programu.
Tretí, voliteľný, parameter je premenná typu int, počet po sebe nasledujúcich registrov, ktorých hodnota sa prečíta. Ak je číslo kladné, postupne sa zobrazujú hodnoty registrov na vyšších adresách ako adresa z 2. parametra, ak je číslo záporné, postupne sa zobrazujú hodnoty registrov na nižších adresách.

```
device.breakpoint(__LINE__,0x48000008); //1 register na adrese 0x48000008
device.breakpoint(__LINE__,0x48000008, 3); /* 3 registre na adresach
                                    0x48000008,0x4800000C,0x48000010 */
device.breakpoint(__LINE__,0x48000008, -2);/* 2 registre na adresach
                                              0x48000008,0x48000004 */
```

Kapitola 7
Knižnica DEBUG USB F042F6P6

Pre jednoduché ladenie programov pomocou USB komunikácie pre mikrokontrolér STM32F042F6P6 bola knižnica z [11] doplnená o debugovacie nástroje. Výsledkom je knižnica DEBUG_USB_F042F6P6, v ktorej je definovaná trieda Debug_usb. Jej hlavnou výhodou sú nulové požiadavky na dodatočný hardvér, v PC ale musí byť nainštalovaný VCP driver. Informácie o breakpointe sa do terminálového programu vypisujú pomocou ANSI escape sekvencií. Z tohto dôvodu okno terminálového programu nefunguje v rolovacom režime, ako sme zvyknutí, ale jednotlivé informácie sa prepisujú a užívateľ má lepší prehľad o stave programu. Knižnica DEBUG_USB_F042F6P6 je určená špeciálne pre mikrokontrolér
STM32F042F6P6 aj keď sa program v mbede vytvára pre vývojový kit NUCLEO-F042K6, ktorý obsahuje mikroprocesor STM32F042K6P6. Rozdiel medzi týmto procesorom a STM32F042F6P6, ktorý používame, je, že STM32F042K6P6 má vyvedených 32 pinov, zatiaľ čo STM32F042F6P6 má vyvededených iba 20 pinov. Na USB komunikáciu sa používaju piny piny PA11 a PA12, ktoré na STM32F042F6P6 nie sú vyvedené. Je však možné namapovať ich na miesto pinov PA9 a PA10 pomocou

```
SYSCFG->CFGR1 |=0x10;  // premapovanie pinov USB
```

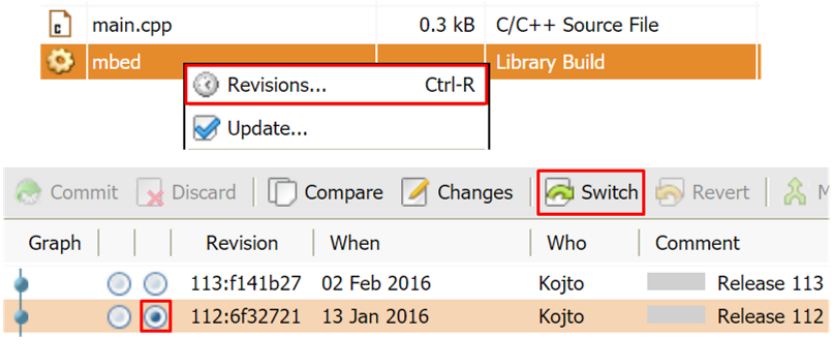
Premapovanie pinov je použité v knižnici a z tohto dôvodu nie je možné knižnicu použiť na vývojový kit NUCLEO-F042K6, pre ktoré sa v mbede program vytvára.

Importovanie knižnice do programu

Knižnica je voľne dostupná na stránke:
`https://os.mbed.com/users/bieleluk/code/DEBUG_USB_F042F6P6/`
Návod na importovanie knižnice do programu je popísaný v kapitole 6.0.2.

Downgrade knižnice mbed

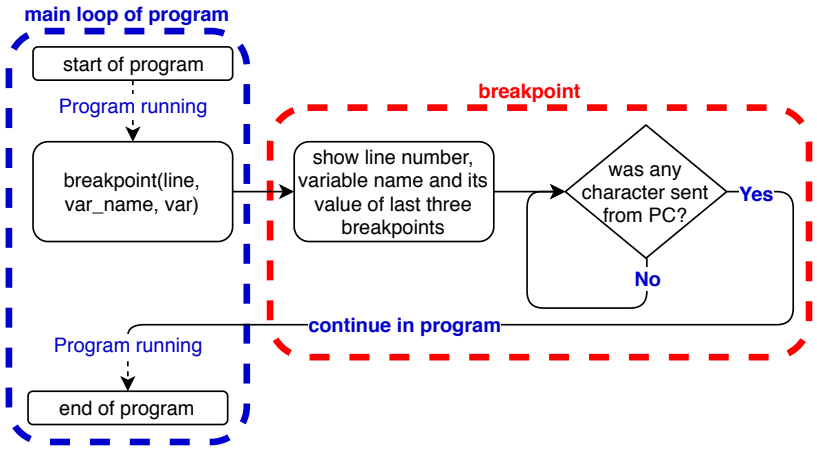
Pri vytváraní nového programu sa knižnica mbed automaticky aktualizuje do najnovšej verzie. Pre použitie knižnice DEBUG_USB_F042F6P6 je potrebné previesť downgrade knižnice mbed na verziu 112, pretože pri použití novších verzií dochádza pri kompilácii programu k chybe. V priečinku programu klikneme pravým tlačidlo na knižnicu mbed, otvoríme Revisions, označíme verziu 112 a zmenu potvrdíme kliknutím na Switch.



Obrázok 7.1. Downgrade knižnice mbed

7.1 Trieda Debug usb

Knižnica DEBUG_USB_F042F6P6 obsahuje triedu Debug_usb , ktorá slúži na krokovanie programu a vypisovanie pozície breakpointu a aktuálnej hodnoty jednej premennej pomocou USB komunikácie do virtuálneho sériového portu v počítači. V triede je definovaná funkcia *breakpoint*(), ktorá preruší beh programu v hlavnom cykle a do virtuálneho sériového portu odošle informácie o posledných troch breakpointoch: poradové číslo breakpointu, číslo riadku, na ktorom sa funkcia nachádza, a hodnotu jednej ľubovoľnej premennej typu *int*, *float*, *char* alebo *char**. Následne program čaká na prijatie ľubovoľného znaku z COM prtu. Po prijatí znaku program beží až do miesta ďalšieho breakpintu. Debug_usb má aj plnú funkcionalitu triedy USBSerial (kapitola 4.9), pretože sú v nej definované aj funkcie *printf*(), *scanf*(), *putc*(), *getc*(), a *available*(). Okno terminálového programu je vizuálne rozdelené na dve časti: debugovaciu časť na zobrazenie informácií o breakpointoch a sériový port na odosielanie a prijímanie správ mimo funkcie *breakpoint*(). Pre rozdelenie okna na dve časti a zobrazenie informácii v užívateľsky príjemnom formáte boli použité ANSI Escape sekvencie. Znaky sa správne vykresľujú iba do momentu kedy dôjde k "pretečeniu" okna, peto okno terminálového programu musí byť vo full screen režime a vypisovanie do sériového portu treba používať ohľadom na dané obmedzenie.
Hlavnou výhodou triedy Debug_usb je, že na jej použitie nie je potrebný žiadny dodatočný hardvér. Na druhej strane, samotná implementácia triedy zaberá 23kB z celkových 32kB flash pamäti, preto je možné triedu použiť iba pri kratších a pamäťovo nenáročných programoch.



Obrázok 7.2. Štruktúra breakpointu v triede Debug_usb

7.1.1 Vytvorenie objektu

Ešte pred nahratím programu je potrebné spojiť pin 17 procesoru s pinom D- usb adaptéra a pin 18 procesoru s pinom D+ usb adaptéra pre umožnenie USB komunikácie s počítačom (kapitola 3.1). Virtuálny sériový port inicializujeme vytvorením objektu triedy Debug_usb s nejakým názvom (napr. pc).

```
Debug_usb pc; // vytvorenie virtualneho serioveho portu
```

Použitie vhodného terminálového programu

Pri každom resetovaní programu sa spojenie s počítačom preruší a virtuálny sériový port sa vytvára nanovo. V programe Tera Term, na rozdiel od ostatných terminálových programov, po resetovaní programu nie je potrebné spojenie nanovo inicializovať, preto ho odporúčame používať. Po nahraní programu do mikrokontroléru otvoríme Tera Term a

pripojíme sa novo-vytvorený COM port. Keďže sa jedná iba o emuláciu sériovej komunikácie, je jedno, aký baudrate si zvolíme.

7.1.2 breakpoint

Funkcia breakpoint zastaví beh hlavného cyklu programu, zobrazí informáciu o posledných troch breakpointoch, voliteľne aj hodnotu jednej ľubovoľnej premennej typu *int*, *float*, *char*, *char** (reťazec) alebo hodnoty jedného registra a čaká na prijatie ľubovoľného znaku zo sériového portu. Po prijatí znaku program beží až do miesta ďalšieho breakpintu.

Breakpoint bez zobrazenia premennej

Ak chceme program iba krokovať, bez vypisovania premenných, použijeme funkciu *breakpoint*() s jedným parametrom, ktorým je je premenná typu int, číslo riadku, na ktorom sa breakpoint nachádza. Odporúča sa použiť makro *__LINE__* definované v mbede, ktorá sa pri preklade programu automaticky nahradí číslom riadku, v ktorom sa nachádza.

```
pc.breakpoint(__LINE__);
```

Breakpoint so zobrazením premennej typu int, float, char alebo char *

Prvým parametrom je premenná typu int, číslo riadku, na ktorom sa breakpoint nachádza(*__LINE__*). Druhým parametrom je reťazec s maximálnou dĺžkou 20 znakov, názov premennej (napr. var), ktorej hodnota sa v breakpointe zobrazí. Odporúča sa použiť makro *name*(*var*) definované v knižnici Debug_universal, ktoré sa pri preklade programu automaticky nahradí názvom premennej (napr. `var`). Tretím parametrom je samotná premenná (napr. var) typu *int*, *float*, *char* alebo *char** (reťazec).

```
float pi = 3.141;          // premenna typu float
char one_character =  'A'; // premenna typu char
int var_int = 10;    // premenna typu int
char my_string[] = "toto je retazec"; // premenna typu char* (string)
pc.breakpoint(__LINE__, name(pi), pi); // breakpoint so zobrazenim float-u
pc.breakpoint(__LINE__, name(var_int), var_int); // zobrazenie int-u
pc.breakpoint(__LINE__, name(my_string), my_string); /* breakpoint so
zobrazenim retazca */
pc.breakpoint(__LINE__, name(one_character), one_character); /* breakpoint
so zobrazenim znaku */
```

Breakpoint so zobrazením hodnoty jedného registra

Prvým parametrom je premenná typu int, číslo riadku, na ktorom sa breakpoint nachádza(*__LINE__*). Druhým parametrom je premenná typu uint32_t, adresa registra, ktorého hodnota sa prečíta, viac o konfiguračných registroch v kapitole 10.

```
pc.breakpoint(__LINE__, 0x48000000 ); /* breakpoint so zobrazenim registra
                                                 na adrese 0x48000000 */
```

7.1.3 printf, scanf, putc, getc, available

Funkcie ponúkajú čiastočnú funkcionalitu triedy USBSerial a sú popísané v kapitole 4.9.

Kapitola 8
Knižnice DEBUG F042F6P6 a DEBUG F303

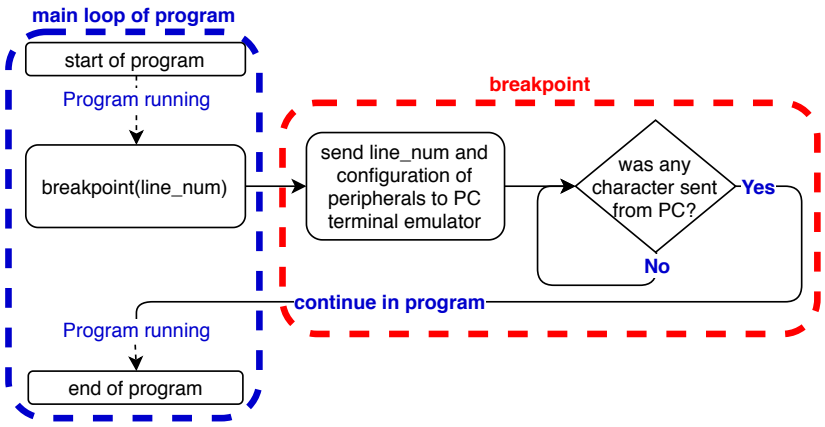
Pre jednoduché ladenie programov pomocou výpisu konfigurácie periférií do sériového portu pre mikrokontrolér STM32F042F6P6 a NUCLEO-F303RE boli vytvorené knižnice DEBUG_F042F6P6 a DEBUG_F303, v ktorých je definovaná trieda Debug_complete. Jej hlavnou výhodou je komplexný výpis konfigurácií pinov a ďalších periférií, informácie sa do terminálového programu vypisujú pomocou ANSI escape sekvencií. Z tohto dôvodu okno terminálového programu nefunguje v rolovacom režime, ako sme zvyknutí, ale jednotlivé informácie sa prepisujú a užívateľ má lepší prehľad o stave programu.

8.0.1 Importovanie knižnice do programu

Knižnica DEBUG_F042F6P6 je voľne dostupná na stránke:
`https://os.mbed.com/users/bieleluk/code/DEBUG_F042F6P6/`
Knižnica DEBUG_F303 je voľne dostupná na stránke:
`https://os.mbed.com/users/bieleluk/code/DEBUG_F303/`
Návod na importovanie knižníc do programu je popísaný v kapitole 6.0.2

8.1 Trieda Debug complete

Trieda Debug_complete slúži na krokovanie programu pomocou sériového portu a na výpis konfigurácie periférií procesoru v mieste breakpointu. Keď program dobehne na miesto breakpointu, beh programu v hlavnom cykle sa preruší, zobrazí sa konfigurácia hodín mikrokontroléra, AD prevodníka, timerov a pinov mikrokontroléra. Následne sa čaká na prijatie ľubovoľného znaku zo sériového portu a po jeho prijatí program beží až do miesta ďalšieho breakpintu. Pre mikrokontrolér STM32F042F6P6je použitie triedy je náročné na pamäť, zaberá 27kB z celkových 32kB flash pamäti, ale ponúka ucelený balík informácií o konfigurácii periférií procesoru v užívateľky príjemnej forme. Pre použitie triedy Debug_complete je potrebný prevodník UART-USB pre komunikáciu mikrokontroléru s počítačom.



Obrázok 8.1. Štruktúra breakpointu v triede Debug_complete

8.1.1 Vytvorenie objektu

Piny procesoru nakonfigurujeme v poradí ako TX pin (napr. PA2) a RX pin (napr. PA3) debugovacieho sériového portu tak, že vytvoríme objekt triedy Debug_complete s nejakým názvom (napr. pc) a priradíme mu tieto piny.
Prvé dva parametre sú povinné, sú to názvy pinov v poradí TX pinu (PA2) a RX pinu (PA3) debugovacieho sériového portu.
Tretí, nepovinný parameter, je premenná typu int, rýchlosť sériovej komunikácie (baudrate). Ak sa parameter nepoužije, predvolená hodnota baudrate je 115200 Bd/s.

```
Debug_complete pc (PA_2, PA_3); // tx a rx pin, baudrate je 115200 Bd/s
Debug_complete pc (PA_2, PA_3, 57600); //tx a rx pin, baudrate: 57600 Bd/s
```

8.1.2 breakpoint

Funkcia breakpoint zastaví beh hlavného cyklu programu, zobrazí informácie o konfigurácii hodín mikrokontroléra, AD prevodníka, timerov, pinov mikrokontroléra a následne čaká na prijatie ľubovoľného znaku zo sériového portu. Po prijatí znaku program beží až do miesta ďalšieho breakpintu. Parametrom funkcie je premenná typu int, číslo riadku, na ktorom sa breakpoint nachádza($\_\_LINE\_\_$).

```
pc.breakpoint(__LINE__); // vypise sa konfiguracia vybranych periferii
```

Kapitola 9
Použitie jazyka ARM assembler v prostredí mbed pre procesory STM32 s jadrom Cortex-M4

Vysokoúrovňové programovacie jazyky vyžadujú kompilátor, ktorý proloží program do strojového kódu, do binárnej postupnosti inštrukcií, ktoré procesor priamo vykonáva. Písať program v strojovm kóde by bolo príliš zložité, preto sa na programovanie na najnižšej úrovni používa jazyk assembler, ktorý umožňuje používať inštrukcie v textovej forme. Následne sa textové inštrukcie preložia a výsledkom je efektívnejší a kratší strojový kód ako pri použití programovacieho jazyka vyššej úrovne.

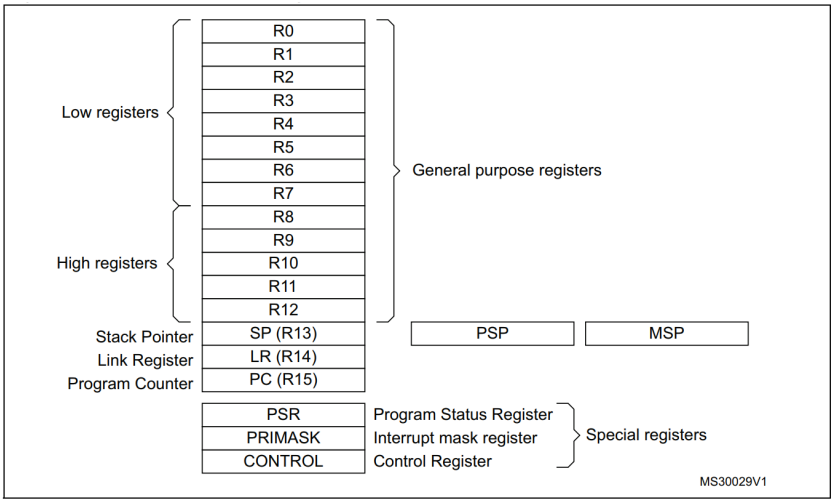
Procesory sa navzájom líšia samotnou architektúrou alebo rôznym stupňom funkcionality, preto sa pre rôzne procesory používajú odlišné inštrukčné sady. V nasledujúcej kapitole budeme pracovať s mikrokontrolérom NUCLEO-F303RE, na programovanie ktorého je možné použiť inštrukčnú sadu pre procesory s jadrom Cortex®-M4 dostupnú z [13].

Hlavným rozdielom oproti vysokoúrovňovým programovacím jazykom je, že programátor priamo pristupuje a zapisuje do registrov procesora a môže používať iba operácie(inštrukcie), ktoré sú priamo implementované v procesore. Z tohto dôvodu sa jazyk assembler využíva hlavne v prípade, keď programátor potrebuje mať úplný prehľad o behu programu priamo v procesore.

Študenti bez pokročilej znalosti hardvéru majú pri prechode na nízkoúrovňové programovacie jazyky najväčší problém so zložitou konfiguráciou periférií na začiatku programu. Tento problém je možné odstrániť použitím IDE Mbed, pretože okrem použitia tried mbed je takisto možné volať fukcie písané v assembleri. Hlavnou ideou je, že na konfiguráciu periférií a niektoré operácie sa použijú triedy mbed a pomocou jazyka assembler by boli vytvorené funkcie na vykonávanie operácií, kde potrebuje mať programátor celý priebeh pod kontrolou. Takto študent rýchlejšie pochopí ideu programovania pomocou registrov a podľa nadobudnutých znalostí môže plynule prejsť na nízkoúrovňové programovanie.

9.1 Registre procesora

Na programovanie procesorov s jadrom Cortex-M4 v assembleri je možné použiť základné registre procesora R0-R15(obrázok 9.1). Register je pamäťové miesto v procesore, ktoré uchováva 32-bitové číslo. Registre sa používajú na uchovávanie čísel, na aritmeticko-logické operácie medzi sebou alebo poskytujú informácie o behu programu a o výsledkoch predošlých operácií (inštrukcií).



Obrázok 9.1. Základné registre procesora s jadrom Cortex®-M4, prebraté z [14].

Základné registre

Základné registre R0-R12 sa používajú na načítanie hodnôt z pamäti procesora a na aritmeticko-logické operácie medzi sebou.

Ukazateľ na vrchol zásobníka

Zásobník je časť pamäti procesora, ktorá sa používa na uloženie návratových adries alebo hodnôt registrov počas volania podprogramov alebo pri obsluhe prerušení. Do zásobníka je možné ukladať položky pomocou inštrukcie *PUSH* alebo ich z neho odoberať inštrukciou *POP*. Zásobník, ako dátová štruktúra, funguje tak, že naposledy pridaná položka bude odobratá ako prvá. Dôležitou hodnotou je preto adresa vrchola zásobníka, ktorá je uložená registri s názvom stack pointer(SP). Zásobník začína na konci dátovej pamäti, po každom pridaní položky hodnota SP klesne o 4 a po každom odobratí položky sa hodnota SP zvýši o 4.

Link register(LR)

Link register je register R14, ktorý obsahuje návratovú adresu z naposledy volaného podprogramu alebo funkcie. V prípade vnoreného volania podprogramov sa jeho hodnota ukladá do zásobníka, aby po vykonaní vnorených funkcií program pokračoval na mieste ich volania.

Program counter(PC)

V registri s názvom Program counter(PC) je uložná adresa nasledujúcej vykonávanej inštrukcie

9.1.1 Program status register (PSR)

Väčšina inštrukcií môže podľa výsledku operácie aktualizovať takzvané flag bity, ktoré sa nachádzajú v program status registri (PSR). Základné flag bity sú:

- N - je automaticky nastavený do 1, ak je výsledok operácie záporný, v opačnom prípade je jeho hodnota vynulovaná.
- Z - je automaticky nastavený do 1, ak je výsledok nula, v opačnom prípade je jeho hodnota vynulovaná.
- C - je automaticky nastavený do 1, ak je výsledok sčítania väčší alebo rovný 2^{32}, ak výsledok odčítania je nezáporný a v ďalších prípadoch popísaných v [13], v opačnom prípade je vynulovaný.

- V - je automaticky nastavený do 1, ak operácia spôsobí pretečenie hodnoty v registri, v opačnom prípade je vynulovaný.

Inštrukcie typicky aktualizujú flag bity, ak obsahujú príponu S, napríklad inštrukcia ADD neaktualizuje flag bity, ale $ADDS$ áno.

9.1.2 Podmienečné vykonanie inštrukcií

Ak nejaká inštrukcia aktualizuje flag bity registra, na základe výsledku danej inštrukcie (podľa hodnoty flag bitov) je možné pomocou konštrukcie If-Then podmienečne vykonať až 4 inštrukcie.

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.

Obrázok 9.2. Podmienkové prípony pre inštrukcie, prebraté z [13].

Konštrukcia If-Then má presne daný tvar:

- Ako prvý je použitý tvar konštrukcie (napr. ITE) a za ním ľubovoľná testovaná podmienka z obrázka 9.2 (napr. GE).
- Počet písmen T(then) v konštrukcii určuje počet inštrukcií, ktoré sa vykonajú v prípade, že podmienka je splnená, jedna inštrukcia typu T je povinná. Za inštrukciu typu T je potrebné doplniť testovanú podmienkovú príponu (napr. ADDGE R1, R2).
- Počet písmen E(else) v konštrukcii určuje počet inštrukcií, ktoré sa vykonajú v prípade, že testovaná podmienka nie je splnená. Po prvej inštrukcii typu E môžu nasledovať iba ďalšie inštrukcie typu E. Na koniec každej inštrukcie typu E je potrebné doplniť opačnú podmienkovú príponu, teda že testovaná podmienka neplatí (napr. SUBLT R1, R2).

```
ITE GE
ADDGE R1, R2
SUBLT R1, R2
```

Správne konštrukcie:IT, ITEEE, ITTTT, ...
Nesprávne konštrukcie: IE, ITET, ITTTEE, ...
Konštrukcie nesmú byť do seba vnorené, inštrukcie meniace hodnotu PC registra musia byť mimo konštrukcie alebo ako posledná inštrukcia v bloku(B, BL, POP{PC}, ...), každá inštrukcia v bloku musí obsahovať podmienkovú príponu a použité inštrukcie v bloku nesmú aktualizovať flag bity.

9.2 Konfiguračné registre periférií

Konfigurácie jednotlivých periférií procesora sú uložené v konfiguračných registroch v pamäti procesora, napríklad konfiguračné registre periférie GPIOA v procesoroch STM32F303xD/E začínajú na adrese $0x48000000$ (obrázok 9.3).

Boundary address	Size (bytes)	Peripheral
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

Obrázok 9.3. Miesto onfiguračných registrov niektorých periférií pre STM32F303xD/E.

Jedným z konfiguračných registrov periférií GPIO je aj outupt data register(ODR), ktorý obsahuje logické hodnoty na výstupe jednotlivých pinov daného portu v prípade, že pin je vo výstupnom režime(obrázok 9.4). Offset $0x14$ znamená, že register sa nachádza na adrese o $0x14$ bajtov posunutej od počiatočnej adresy konfiguračných registrov danej periférie. Pre periférium GPIOA sa bude register ODR nachádzať na adrese $GPIOA + offset = 0x48000014$.

8.4.6 GPIO port output data register (GPIOx_ODR) (x = A..F)

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

Obrázok 9.4. Output data register.

9.3 Štruktúra funkcie písanej v assembleri

Funkcia písaná v assembleri (napr. my_func) musí byť vytvorená v súbore s príponou $.s$ (napr. $my\_func.s$,) ktorý musí byť v rovnakom priečinku ako súbor $main.cpp$, aby pri kompilácii programu došlo k zlinkovaniu týchto súborov. V súbore $main.cpp$ je potrebné ešte pred začiatkom funkcie $main()$ zadeklarovať všetky použité funkcie ako externé [15].

```
extern "C" void my_func();
```

V súbore s príponou $.s$ je potrebné dodržať pevne danú štruktúru.

- Na začiatku sú zadefinované konštanty, ktoré budú vo funkcii použité, (napr. konštanta PIN s hodnotou 5)

```
PIN EQU 5
```

- Po nich nasleduje povinný riadok.

```
    AREA asm_func, CODE, READONLY
```

- Ďalej je potrebné previesť export funkcie pre úspešné zlinkovanie so súborom *main.cpp* pri kompilácii.

```
    EXPORT my_func
```

- Samotné telo funkcie začína jej názvom, ktorý nie je odsadený, ako ostatné inštrukcie. Potom je potrebné vložiť hodnotu LR do zásobníka, aby po skončení funkcie program správne pokračoval za niestom volania funkcie.

```
my_func
    PUSH {LR}
```

- Potom nasledujú inštrukcie funkcie.
- Na konci funkcie je potrebné zo zásobníka odstrániť hodnotu PC, do PC sa uloží návratová adresa a program pokračuje za miestom volania danej funkcie [16].

```
    POP {PC}
```

- V prípade potreby nasledujú ďalšie funkcie volané z $my\_func()$.
- Na konci súboru sa nachádzajú dva povinné riadky.

```
    ALIGN
    END
```

- Na zakomentovanie zvyšku riadku sa používa znak ;

9.4 Použitie funkcie bez vstupných parametrov a návratovej hodnoty

Úlohou je rozblikať LED na pine PA5 tak, že na konfiguráciu pinu a generovanie oneskorenia budú použité triedy mbed, LED sa bude rozsvecovať a zhasínať pomocou dvoch funkcií písaných v assembleri.

9.4.1 Súbor main.cpp

Vo súbore *main.cpp* je potrebné deklarovať funkcie písané v assembleri ako externé, bez vstupných a návratových parametrov. Vo funkcii *main*() je možné funkcie písané v assembleri používať rovnako, ako ostatné funkcie.

```
#include "mbed.h"
//konfiguracia digitalneho vystupu na PA5 s nazvom myled
DigitalOut myled(PA_5);
//deklaracia funkcie turn_on_pa5 pisanu v assembleri (v turn_on_pa5.s)
extern "C" void turn_on_pa5();
//deklaracia funkcie turn_off_pa5 pisanu v assembleri (v turn_off_pa5.s)
extern "C" void turn_off_pa5();
int main() {
    while(1) {
        turn_on_pa5();  // volanie funkcie turn_on_pa5
        wait(0.5);      // cakanie 0.5s
        turn_off_pa5(); // volanie funkcie turn_off_pa5
        wait(0.5);      // cakanie 0.5s
    }
}
```

9.4.2 Súbor turn on pa5.s

V súbore *turn_on_pa5.s* je definovaná funkcia *turn_on_pa5* písaná v assembleri, ktorá zmení logickú hodnotu výstupu pinu PA5 na log.1. Výstupná logická hodnota jednotlivých pinov portu *A* je uložená v ODR na adrese *GPIOA* posunutej o offset $0x14$. Hodnotu ODR registra je potrebné načítať do univerzálneho registra (napr. R2), nastaviť hodnotu piateho bitu do 1 a takto upravenú hodnotu opäť zapísať do na adresu *GPIOA* posunutú o offset $0x14$. Piaty bit registra môžeme do 1 nastaviť napríklad tak, že do iného registra zapíšeme číslo $0x20$ (5. bit je 1, ostatné 0) a prevedieme logický súčet týchto registrov.

```
;miesto pre definovanie konstant
GPIO_ODR_Offset  EQU 0x14; adresovy offset GPIO ODR register
LED EQU 5; cislo pouziteho pinu
GPIOA EQU 0x48000000; pociatocna adresa GPIOA konfiguracnych registrov
    ;zaciatok funkcneho bloku v assembleri
    AREA asm_func, CODE, READONLY
    ;export funkcie turn_on_pa5, funkcia bude viditelna pre compiler
    EXPORT turn_on_pa5
;zaciatok funkcie
turn_on_pa5
    ; vlozenie LR do zasobnika pre uspesny navrat po vykonani funkcie
    PUSH {LR}
    ; ulozenie cisla 0x1 pousunuteho o 5 bitov dolava(0x20) do R0
    MOV R0, #(0x01 :SHL: LED)
    ; nacitanie pociatocnej adresy konfiguracnych registrov GPIOA do R1
    LDR R1, =GPIOA
    ; nacitanie hodnoty konfiguracneho registra ODR z adresy GPIOA
    ; posunutej o GPIO_ODR_Offset (0x48000014) do R2
    LDR R2, [R1,#GPIO_ODR_Offset]
    ; logicky sucet R2 a R0 ulozeny do R2 (nastavenie 5. bitu do 1)
    ORR R2, R0
    ; ulozenie hodnoty R2 do ODR na adresu 0x48000014
    STR R2,[R1,#GPIO_ODR_Offset] ;rozsvieti sa LED
    ;odstranenie PC zo zasobnika pre navrat z funkcie
    POP {PC}
    ;koniec funkceho bloku v assembleri
    ALIGN
    END
```

9.4.3 Súbor turn off pa5.s

V súbore $turn\_off\_pa5.s$ je definovaná funkcia $turn\_off\_pa5$, ktorá zmení logickú hodnotu výstupu pinu PA5 na log.0. Výstupná logická hodnota jednotlivých pinov portu *A* je uložená v ODR na adrese *GPIOA* posunutej o offset $0x14$. Hodnotu ODR registra je potrebné načítať do univerzálneho registra (napr. R2), nastaviť hodnotu piateho bitu do 0 a takto upravenú hodnotu opäť zapísať do na adresu *GPIOA* posunutú o offset $0x14$. Piaty bit registra je možné nastaviť do 0 napríklad tak, že do iného registra sa zapíše číslo $0xFFFFFFDF$ (5. bit je 0, ostatné 1) a následne sa prevedie logický súčin týchto registrov. Číslo $0xFFFFFFDF$ môžeme dostať napríklad tak, že prevedieme vylučovacie alebo čísla $0x20$ (5. bit je 1, ostatné 0) s $0xFFFFFFFF$.

```
GPIO_ODR_Offset  EQU 0x14 ;adresovy offset pre GPIO ODR register
LED EQU 5 ;cislo pouziteho pinu
```

```
GPIOA EQU 0x48000000 ;pociatocna adresa GPIOA konfiguracnych registrov
    AREA asm_func, CODE, READONLY ;zaciatok funkcneho bloku v assembleri
    EXPORT turn_off_pa5 ;export funkcie turn_off_pa5
turn_off_pa5 ;zaciatok funkcie turn_off_pa5
    PUSH {LR} ; vlozenie LR do zasobnika
    ; ulozenie cisla 0x1 pousunuteho o 5 bitov dolava(0x20) do R0
    MOV R0, #(0x01 :SHL: LED)
    ; pouzitie vylucujuceho alebo na R0 s konstantou 0xFFFFFFFF
    ; a ulozenie do R0 , 0x20 XOR 0xFFFFFFFF -> 0xFFFFFFDF
    EOR R0, #0xFFFFFFFF
    ; nacitanie pociatocnej adresy konfiguracnych registrov GPIOA do R1
    LDR R1, =GPIOA
    ; nacitanie hodnoty konfiguracneho registra ODR do R2
    LDR R2, [R1,#GPIO_ODR_Offset]
    ; logicky sucet R2 a R0 ulozeny do R2 (nastavenie 5. bitu do 0)
    AND R2, R0
    ; ulozenie zmenenej hodnoty do registra na adresu 0x48000014
    STR R2,[R1,#GPIO_ODR_Offset] ; LED zhasne
    POP {PC} ;odstranenie PC zo zasobnika
    ALIGN ;koniec funkceho bloku v assembleri
    END
```

9.5 Použitie funkcie so vstupným argumentom

Funkcia písaná v assembleri môže mať až štyri vstupné parametre, tieto parametre sa uložia do registrov R0-R3. Registre sú 4-bajtové, preto argument funkcie nesmie mať väčšiu veľkosť, v drvivej väčšine sa používa 4-bajtové celé číslo bez znamienka *uint32_t*. Úlohou je rozblikať LED na ľubovoľnom pine na porte *A* tak, že na konfiguráciu pinu a generovanie oneskorenia budú použité triedy mbed, LED s bude rozsvecovať a zhasínať pomocou jednej funkcie písanej v assembleri, jej vstupným parametrom bude premenná typu *uint32_t*, číslo pinu na porte *A*, na ktorom je pripojená LED.

9.5.1 Súbor main.cpp

Vo súbore *main.cpp* je potrebné deklarovať funkciu *change_state* ako externú s jedným vstupným parametrom, bez návratovej hodnoty. Následne sa táto funkcia volá so vstupným parametrom 5(pin PA5) v nekonečnom cykle.

```
#include "mbed.h"
DigitalOut myled(PA_5);
extern "C" void change_state(uint32_t pin_number);
int main() {
    while(1) {
        change_state(5); // zmena log. hodnoty vystupu PA5
        wait(0.2);       // cakanie 200 ms
    }
}
```

9.5.2 Súbor change state.s

V súbore *change_state.s* je definovaná funkcia *change_state*, ktorá invertuje logickú hodnotu na výstupe pinu na porte *A*, ktorý je parametrom funkcie(v R0 na začiatku funkcie). Výstupná logická hodnota jednotlivých pinov portu *A* je uložená v ODR na

adrese $GPIOA$ posunutej o offset $0x14$. Hodnotu ODR registra je potrebné načítať do univerzálneho registra a do iného registra treba načítať masku x-tého pinu(x-tý bit je 1, ostatné 0). Ďalej sa prevedie logický súčin týchto registrov, ak je na výstupe x-tého pinu log.0, výsledok bude 0. Nakoniec sa pomocou If-Then konštrukcie invertuje hodnota x-tého bitu na základe výsledku predošlej operácie. Nakoniec sa musí upravená hodnota zapísať do na adresu $GPIOA$ posunutú o offset $0x14$.

```
GPIO_ODR_Offset  EQU 0x14 ; adresovy offset pre GPIO ODR register
GPIOA EQU 0x48000000 ; pociatocna adresa GPIOA konfiguracnych registrov
AREA asm_func, CODE, READONLY
EXPORT change_state ; export funkcie change_state
change_state ;zaciatok funkcie
    PUSH {LR}
    ; v R0 je cislo pinu x
    ; nacitanie 0x1 do R1
    LDR R1, =0x1
    ; posunutie hodnoty R1 o hodnotu R0 dolava a ulozenie do R1
    ; (hodnota x bitu bude 1, ostatne budu 0)
    LSL R1, R0
    ; nacitanie pociatocnej adresy konfiguracnych registrov GPIOA do R2
    LDR R2, =GPIOA
    ; nacitanie GPIOA ODR registra do R3
    LDR R3, [R2, #GPIO_ODR_Offset]
    ; logicky sucin(s aktualizovanim flag bitov) R3 a R1 ulozeny do R4
    ; ak bola na vystupe log.0 hodnota R4 bude 0
    ANDS R4,R3,R1
    ;konsrukcia ITEE na x-ty bit zapise hodnotu podla flag bitu
    ITEE EQ
    ;ak v R0 je 0, logicky sucet R3 a R1 ulozeny do R3 (x-ty bit do 1)
    ORREQ R3, R1
    ;ak v R0 nie je 0, pouzitie vylucujuceho alebo na R1 s 0xFFFFFFFF
    ; a ulozenie do R1 , (x-ty bit bude 0, ostatne 1)
    EORNE R1, #0xFFFFFFFF
    ;ak v R0 nie je 0, logicky sucet R3 a R1 ulozeny v R3 (x-ty bit do 0)
    ANDNE R3, R1
    ; ulozenie zmenenej hodnoty do registra na adrese 0x48000014
    STR R3,[R2,#GPIO_ODR_Offset]; invertovanie hodnoty na x-tom pine
    POP {PC}
    ALIGN
    END
```

9.6 Použitie funkcie s volaním vnorenej funkcie

Funkcia písaná v assembleri môže v jednom súbore volať aj ďalšie vnorené funkcie. Štruktúra vnorenej funkcie je nasledovná:

- Funkcia(procedúra) začína jej názvom a slovom $PROC$ označujúcim jej začiatok
- Následne sa do zásobníka vložia hodnota LR a aktuálne hodnoty registrov ktoré sa vo vnorenej funkcii budú používať, aby sa zachovali hodnoty týchto registrov a po vykonaní vnorenej funkcie ich bude možné použiť.
- Potom nasleduje telo funkcie.

- Na konci sa zo zásobníka odstráni hodnota PC pre návrat na miesto volania a uložené registrov vložených do zásobníka na začiatku sa napäť priradia týmto registrom. Koniec funkcie označuje slovo $ENDP$.

Skok do podprogramu sa prevedie pomocou inštrukcie BL s názvom volaného podprogramu.
Úlohou je rozblikať LED na ľubovoľnom pine na porte A s ľubovoľnou hodnotou periódy danej v milisekundách tak, že na konfiguráciu pinu budú použité triedy mbed. Bliknutie LED s požadovnou periódou na požadovanom pine bude zabezpečené jednou funkciou písanej v assembleri, jej vstupnými parametrami budú:

- Premenná typu $uint32\_t$, číslo pinu na porte A, na ktorom je pripojená LED
- Premenná typu $uint32\_t$, prióda blikania v milisekudách

9.6.1 Súbor main.cpp

Vo súbore *main.cpp* je potrebné deklarovať funkciu $flash$ ako externú s dvomi vstupnými parametrami, bez návratovej hodnoty. Následne sa táto funkcia volá so vstupnými parametrami 5(pin PA5) a 1000(perioda v ms) v nekonečnom cykle.

```
#include "mbed.h"
DigitalOut myled(PA_5);
// deklaracia externej funkcie flash s 2 vstupnymi argumentmi
extern "C" void flash(uint32_t pin, uint32_t period_ms);
int main() {
    while(1) {
        flash(5, 1000); // bliknutie na PA5 s periodou 1000ms
    }
}
```

9.6.2 Súbor flash.s

V súbore $flash.s$ je definovaná funkcia $flash$, ktorá najskôr nastaví logickú hodnotu x-tého pinu na porte A na log.1. Následne sa zavolá vnorená funkcia $WAIT\_MS$, ktorá čaká počet milisekúnd uložený v R1. Potom sa logická hodnota x-tého pinu zmení na log.0 a opäť sa zavolá funkcia $WAIT\_MS$.
Výstupné logické hodnoty jednotlivých pinov portu sú uložené v registri ODR, môžeme ich prečítať alebo všetkým naraz priradiť novú hodnotu, čo môže spôsobovať problémy. Ak chceme vynulovať(resp. nastaviť do 1) iba niektoré bity, výhodnejšie je použiť register BRR(resp. BSRR). Tieto registre sú určené iba na zápis, nie je možné z nich čítať logické hodnoty jednotlivých pinov portu. Nastavením n-tého bitu registra BRR(reset) do 1 sa výstupná hodnota n-tého pinu zmení na log.0 a nastavením n-tého bitu registra BSRR(set) do 1 sa výstupná hodnota n-tého pinu zmení na log.1.
Frekvencia hodín mikrokontroléra je 72MHz, aby vo funkcii $WAIT\_MS$ vzniklo oneskorenie y ms, je potrebné aby funkcia trvala $y \cdot 72000$ hodinových cyklov mikrokontroléra. Do registra si preto uložíme číslo $y \cdot 72000$ a v cykle od tejto hodnoty budeme odpočítavať trvanie(počet hodinových cyklov) jedného cyklu odčítania. Na to potrebujeme poznať trvanie použitých inštrukcií, ktoré nájdeme v [14]. Až na posledný prechod cyklu bude jeho trvanie 6 hodinových cyklov procesora.

```
GPIO_BSRR_Offset  EQU 0x18; bit set reset register offset
GPIO_BRR_Offset   EQU 0x28; bit reset register offset
GPIOA EQU 0x48000000
    AREA asm_func, CODE, READONLY
```

```
    EXPORT flash
flash
    PUSH {LR}
    ; v R0 je cislo pinu x
    ; v R1 je perioda blikania v ms
    ; posunutie hodnoty R1 o 1 bit doprava, celociselne delenie 2
    LSR R1,#1 ; v R1 je polperioda
    ; nacitanie pociatocnej adresy konfiguracnych registrov GPIOA do R2
    LDR R2, =GPIOA
    ; nacitanie 0x1 do R3
    LDR R3, = 0x01
    ; posunutie hodnoty R3 o hodnotu R0 dolava a ulozenie do R3
    ; (hodnota x bitu bude 1, ostatne budu 0)
    LSL R3, R3, R0
    ; ulozenie hodnoty R3 na adresu 0x48000018
    ; nastavenie x-teho bitu BSRR do 1, vystup x-teho pinu do log.1
    STR R3,[R2,#GPIO_BSRR_Offset] ; rozsvieti LED
    ; vykonanie funkcie WAIT_MS so vstupnym argumentom R1
    BL WAIT_MS
    ; ulozenie hodnoty R3 na adresu 0x48000028, nastavenie
    ; nastavenie x-teho bitu BSRR do 1, vystup x-teho pinu do log.0
    STR R3,[R2,#GPIO_BRR_Offset] ;zhasne LED
    ; vykonanie funkcie WAIT_MS so vstupnym argumentom R1
    BL WAIT_MS
    POP {PC}

; zaciatok procedury WAIT_MS
WAIT_MS PROC
    ; v procedure sa pouzivaju R0 a R1 -> aktualne hodnoty R0 a R1 sa
    ; ulozia do zasobnika, aby sa po skonceni procedury mohli opat pouzit
    ; LR sa ulozi pre navrat do funcie flash po vykonani WAIT_MS
    PUSH {R0,R1,LR}
    ; v R1 je cas v ms
    ; nacitanie 72000 do R0, frekvencia hodin je 72MHz,
    ; pre cakanie 1ms musi cyklus trvat 72000 hodinovych cyklov
    LDR R0, =72000
    ; nasobenie hodnoty registra R1 s R0 a ulozenie do R1
    ; pre cakanie y ms musi cyklus trvat y*72000 hodinovych cyklov
    MUL R1,R0
; zaciatok cyklu s nazvom WAITLOOP
WAITLOOP
    ; odcitanie 6 od R1 s aktualizovanim flag bitov
    SUBS R1,#6 ; odcitanie poctu hodinovych cyklov v jednom cykle
    ;prazdna instrukcia
    NOP
    ; podmieneny skok na WAITLOOP, ak v R0 nie je 0
    BNE WAITLOOP
    ; priradenie R0 a R1 hodnot v case volania WAIT_MS, navrat do flash
    POP {R0,R1,PC}
    ENDP; koniec procedury
    ; koniec funkcneho bloku v assembleri
    ALIGN
    END
```

Kapitola 10
Použitie periférií v konfigurácii, ktoré knižnice mbed nepodporujú

Pri použití tried v prostredí mbed sa periférie konfigurujú pri vytváraní objektu danej triedy. To umožňuje jednoduché používanie periférií, ale obmedzeným spôsobom. Pre manuálnu konfiguráciu periférií je možné pristupovať aj priamo ku konfiguračným registrom mikrokontroléra. V nasledujúcej kapitole budú popísané dva možné prístupy ku týmto registrom. Ďalej budú tieto princípy použité na konfiguráciu kryštálového oscilátora ako zdroja hodín mikrokontroléra STM32F042F6P6 a na konfiguráciu timera TIM1 na počítanie pulzov z výstupu optického inkrementálneho senzora na pine PA9.

10.1 Priama metóda prístupu k registrom

Konfiguračné registre sú namapované do spoločného pamäťového priestoru. Priama metóda umožňuje pristupovať nielen k týmto konfiguračným registrom, ale aj do zvyšku pamäťového priestoru. Na nasledujúcich príkladoch je možné vidieť, akým spôsobom je prečítaný a zmenený obsah ODR registru periférie GPIOA.

```
uint32_t reg_value;
// precitanie ODR periferie GPIOA
reg_value = *((volatile unsigned int *)0x4800000);
// zapis reg_value do ODR periferie GPIOA
*((volatile unsigned int *)0x4800000) = reg_value;
```

Použitie priameho prístupu môže byť riskantné, pretože niektoré miesta v pamäti sú rezervované a nesmie sa na ne pristupovať (obrázok 10.1). V opačnom prípade dôjde k Hard faultu. Rovnaká situácia nastane, ak sa číta z registrov označených ako *write only*, teda smie sa do nich iba zapisovať. Čítať registre je možné čítať iba z adries zarovnaných na celé slová(4-bajtové), v prípade použitia nezarovnanej adresy (napr. 0x48000002) dôjde takisto k chybe programu.

```
// snaha o pristup do rezervovanej pamati, dojde k chybe programu
reg_value = *((volatile unsigned int *)(0x4800000-0x4));
// snaha o pristup na nezarovnanu adresu, dojde k chybe programu
reg_value = *((volatile unsigned int *)0x4800002);
```

Boundary address	Size (bytes)	Peripheral
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

Obrázok 10.1. Miesto konfiguračných registrov niektorých periférií pre STM32F303xD/E, prebraté z [17].

10.2 Použitie štruktúr pre prístup k registrom

Priamy prístup k registrom býva často neprehľadný, pretože programátor si musí pamätať všetky adresy konfiguračných registrov, ku ktorým pristupuje. Ďalší problém je, že priama metóda nie je univerzálna pre rôzne platformy, dva procesory môžu mať periférium s rovnakou funkcionalitou, ale ak majú konfiguračné registre danej periférie na rôznych adresách, je potrebné celý program prerábať. Pre odstránenie týchto problémov je možné použiť metódu prístupu k registrom s využitím štruktúr. Pre každý typ periférie je definovaná štruktúra, v ktorej sú premenné typu *uint32_t* s rovnakými názvami, ako konfiguračné registre danej periférie. K jednotlivým registrom sa pristupuje nasledovne:

```
uint32_t value = PERIFERIA->REGISTER; // citanie hodnoty registra
PERIFERIA->REGISTER = value;          // zmena hodnoty registra
```

Napríklad pre periférie typu GPIO (GPIOA, GPIOB, ...) je definovaná štruktúra, ktorá obsahuje premenné s názvami konfiguračných registrov(MODER, OTYPER, ODR, ...). Hodnotu registra ODR periférie GPIOA teda môžeme čítať alebo meniť pomocou:

```
uint32_t value = GPIOA->ODR;
GPIOA->ODR = value;
```

Okrem toho sa často využívajú bitové logické operátory na vymaskovanie určitých bitov registra, na nastavenie nastavenie určitého bitu do 1 (resp. 0) a podobne. V nasledujúcom programe bliká LED na pine PA5 tak, že sa 5. bit ODR registra periférie GPIOA nastavuje periodicky do 1 a 0 a tým sa mení aj logická hodnota napätia na výstupe pinu.

```
#include "mbed.h"
DigitalOut led(PA_5);
int main() {
    while(1){
        GPIOA->ODR |= 1 << 5; //5. bit GPIOA ODR sa nastavi do 1
        wait(1);
        GPIOA->ODR &= ~(1 <<5); //5. bit GPIOA ODR sa vynuluje
        wait(1);
    }
}
```

10.2.1 Použitie definovaných konštánt

Ďalšou možnosťou, ako ešte viac sprehľadniť program, je použiť definované koštanty, ktoré vymaskujú hodnoty určitých bitov konfiguračného registra. Všetky konštanty pre NUCLEO-F042K6 a teda aj STM32F042F6P6 sú definované v [18]. Najčastešia štruktúra konštanty je

```
TYPPERIFERIE_REGISTER_BIT
```

Napríklad na vymaskovanie 5. bitu registra ODR je možné použiť konštantu *GPIO_ODR_5*, a teda program na blikanie LED na PA5 môžeme napísať aj

```
#include "mbed.h"
DigitalOut led(PA_5);
int main() {
    while(1){
        GPIOA->ODR |= GPIO_ODR_5; //5. bit GPIOA ODR sa nastavi do 1
```

```
            wait(1);
            GPIOA->ODR &= ~GPIO_ODR_5; //5. bit GPIOA ODR sa vynuluje
            wait(1);
        }
    }
```

10.3 Konfigurácia externého HSE na STM32F042F6P6

Na mikrokontroléri STM32F042F6P6 sa ako zdroj hodín predvolene používa PLL, ktorej zdrojom je HSI oscilátor s frekvenciou 8MHz. PLL prenásobí frekvenciu HSI číslom 6 a výsledná frekvencia hodín procesora je 48MHz. Pre časovo kritické operácie je potrebné použiť externý kryštálový oscilátor (HSE), ktorý je presnejší ako HSI. Externý oscilátor je možné zapojiť na piny PF0 a PF1, pre správne fungovanie je potrebné použiť aj kapacitory, ktorých veľkosť sa vypočíta podľa [19]. V implementačnom súbore mbed [20] je napísané, že na začiatku programu sa najskôr otestuje, či je HSE pripojený a ak áno, použije sa ako zdroj hodín PLL. V skutočnosti sa otestuje iba možnosť použiť HSE cez BYPASS a nie externe zapojený HSE, preto je potrebná manuálna konfigurácia. Výslednú frekvenciu systémových hodín ale nechceme meniť. V našom prípade používame HSE s frekvenciou 8MHz, teda nie je potrebné meniť meniť multiplikatívny činiteľ PLL. Celý program aj s podrobným popisom a schémou zapojenia je v prílohe D. Pri vytváraní programu boli použité časti kódu prebraté z [8]. Pre overenie správnej funkcionality je možné použiť debugovaciu triedu Debug_complete (kapitola 8.1).

10.4 Hardvérové počtanie pulzov na STM32F042F6P6

Pri práci s DC motormi sa často využívajú optické inkrementálne snímače na určovanie rýchlosti otáčania alebo na určenie prejdenej vzdialenosti. Na výstupe senzora je pulzný signál, pri pomalších rýchlostiach (desiatky Hz) je možné detekovať nábežné alebo spádové hrany signálu pomocou harvérového prerušenia a podľa počtu pulzov dopočítať prejdenú dráhu. Pri vyšších rýchlostiach sa nezachytia všetky hrany, preto lepšia alternatíva by bola timer v konfigurácii počítania pulzov na určitom pine. Pomocou tried mbed je možné nakonfigurovať timer iba na generovanie PWM, preto je opäť nutná manuálna konfigurácia. V prvom rade je potrebné skontrolovať v datasheete alternatívne funkcie pinov a nájsť taký pin, na ktorom je mžné použiť timer. V našom prípade sme si vybrali pin PA9, na ktorom je budeme používať kanál 2 čítača TIM1 (obrázok 10.2).

Pin name	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7
PA9	-	USART1_TX	TIM1_CH2	TSC_G4_IO1	I2C1_SCL	MCO	-	-

Obrázok 10.2. Alternatívne funkcie pinu PA9 pre STM32F042x6, prebraté z [4].

Celý program, ktorý obsahuje konfiguráciu a aj simuláciu inkrementálneho optického senzora pomocou PWM, je podrobne popísaný v prílohe E. Pri vytváraní programu boli použité časti kódu prebraté z [8].

Kapitola 11
Zhodnotenie výsledkov práce

11.1 Používateľská príručka mbed IDE

V tejto práci bola vytvorená užívateľská príručka obsahujúca detailné popisy najpoužívanejších tried prostredia s teoretickým popisom použitia daných tried a podrobným návodom na vytvorenie objektov. Bola vysvetlená funkcionalita jednotlivých metód, pričom z dôvodu častého výskytu chýb v ich implementácii, bola správnosť konfigurácie kontrolovaná priamo čítaním konfiguračných registrov mikrokontroléra.

Pre praktickú demonštráciu funkcionality bolo vytvorených 40 ukážkových programov pre mikrokontroléry STM32F042F6P6 a NCLEO-F303RE spolu so schémami zapojenia, ktoré sa nachádzajú v priloženom CD.

Odhalené chyby v implementácií a ďalšie problémy súvisiace s použitím jednotlivých tried boli explicitne popísané. Na základe nájdených chýb bolo potrebné opraviť diagramy mikrokontrolérov NCLEO-F303RE a NUCLEO-F042K6. Príručka bola už počas semestra používaná v predmete LPE a na základe spätnej väzby študentov bola priebežne upravovaná.

11.2 Metodika a nástroje pre ladenie programov

Boli vysvetlené základné techniky ladenia a krokovania programov s využitím LED diódy a tlačidla a rôzne prístupy debugovania pomocou výpisov prostredníctvom kanálu UART.

Bola vyvinutá sada debugovacích tried, ktorá je voľne dostupná na stránke vývojového prostredia. Každá trieda obsahuje dokumentáciu Doxygen s podrobným popisom funkcií danej triedy a s ukážkovým programom na demonštráciu funkcionality pre platformy STM32F042F6P6 a NUCLEO-F303RE.

Pre jednoduché použitie popísaných techník nezávisle na zvolenom zariadení bola vytvorená knižnica DEBUG_UNIVERSAL, v ktorej sú implementované 4 triedy s rôznou pamäťovou náročnosťou umožňujúce rôznorodý prístup ladenia programov:

- Debug_led - na krokovanie programu pomocou LED.
- Debug_serial - na ladenie programu pomocou výpisov do sériového portu a na zobrazenie hodnoty jednej premennej s čiastočnou funkcionalitou triedy Serial.
- Debug_register - na krokovanie programov pomocou výpisov do sériového portu, na zobrazovanie a zmenu hodnoty konfiguračných registrov a s čiastočnou funkcionalitou triedy Serial.
- Debug_register_print - pre výpis konfiguračných registrov v užívateľom definovanom formáte bez zastavovania behu programu s možnosťou uloženia výpisov do súboru.

Ďalej bola realizovaná trieda Debug_complete určená špeciálne pre platformy NCLEO-F303RE a STM32F042F6P6 na ladenie programov pomocou podrobných

výpisov konfigurácie vybraných periférií do sériového portu. Trieda bola vyvinutá predovšetkým s cieľom jednoduchšieho a systematického odhaľovania chýb v implemenácii tried mbed pre mikrokontroléry používané v laboratórnej výuke.

Pre ladenie programov v mikrokontroléri STM32F042F6P6 bez použitia dodatočného hardvéru bola vytvorená trieda Debug_usb. Tá prevzala časť funcionality triedy USBSerial a následne bola doplnená o možnosť krokovania programu a zobrazenie hodnoty jednej premennej pomocou USB komunikácie s PC.

11.3 Pokročilé použitie prostredia mbed

Posledným cieľom bolo demonštrovať pokročilé možnosti použitia vývojového prostredia mbed. Pre mikrokontroléry s jadrom Cortex-M4 boli zdokumentované možnosti programovania pomocou jednoduchých modulov písaných v jazyku assembler v kombinácii s využitím tried mbed. Boli vytvorené ukážkové programy pre lepšie pochopenie štruktúry procesora a na vysvetlenie princípu prístupu ku konfiguračným registrom procesora.

S využitím definovaných štruktúr boli popísané možnosti prístupu ku konfiguračným registrom. Pomocou tejto metódy boli vytvorené programy pre mikrokontrolér STM32F042F6P6, ktoré umožňujú použiť periférie aj v takej konfigurácii, akú triedy mbed neponúkajú. Konkrétne boli vytvorené programy pre umožnenie použitia externého kryštálového oscilátora v aplikáciách kritických na časovanie a na konfiguráciu časovača TIM1 ako čítača pulzov.

11.4 Materiály vytvorené nad rámec zadania

Nad rámec zadania boli vytvorené prehľadné zoznamy funkcií tried s krátkou ukážkou použitia. Študenti preto nebudú musieť dohľadávať syntax funkcií v rozsiahlom manuále, ale budú ju mať k dispozícii na niekoľkých stranách, v ktorých sa jednoducho orientuje (príloha B). Rovnakým spôsobom boli zdokumentované aj vytvorené debugovacie triedy (príloha C). Počas vývoja ladiacich tried bola ich implementácia exportovaná do prostredia uVision Keil 5 a debugovaná s využitím ST-Link debuggera na vývojovom kite NUCLEO-F303RE. Všetky tieto úkony boli zdokumentované a nachádzajú sa v priloženom CD. Pri opakovanom použití rovnakého prístupu už nebude potrebné znovu riešiť technické problémy, ktoré sa pri exporte programu vyskytujú.

Kapitola 12
Záver

Cieľom bakalárskej práce bolo analyzovať vývojové prostredie mbed a na základe rozboru vytvoriť materiály, ktoré umožnia optimálne využitie vývojového prostredia mbed v laboratórnej výuke.

Bola vytvorená príručka vývojového prostredia, ktorá vysvetľuje a na vytvorených programoch demonštruje funkcionalitu ale aj obmedzenia jednotlivých tried prostredia (bližší popis je v kapitole 11.1).
Do budúcna by bolo vhodné doplniť príručku aj o ďalšie triedy a doplniť úplné základy programovania v prostredí mbed, a tak by sa z nej mohol stať komplexný výukový materiál vhodný aj pre študentov bez predošlej znalosti programovania. Aj napriek týmto možným vylepšeniam sa jedná o materiály, ktoré môžu výrazne zlepšiť úroveň výuky programovania mikrokontrolérov.

Ďalším cieľom bolo vytvoriť metodiku ladenia programov. Bola vyvinutá sada ladiacich nástrojov, ktorá implementuje metódy ladenia popísané vo vytvorenej metodike a ktorá umožňuje študentom ladenie programov s využitím vytvorených tried (bližší popis je v kapitole 11.2). Z dôvodu použitia aj mikrokontrolérov s menšou pamäťou flash (do 32kB), bol kladený dôraz na pamäťovú efektivitu, pretože pri ďalšom rozširovaní funkcionality by samotný debugovací objekt zabral väčšinu pamäti zariadenia. Pre mikrokontroléry s väčšou pamäťou flash by bolo možné túto funkcionalitu rozšíriť, avšak použitie by už nebolo univerzálne a obsluha koplexného ladiaceho nástroja cez sériový port by bola pre užívateľa zložitá.

Posledným cieľom bolo predstaviť pokročilé možnosti mbed IDE. Pre pochopenie štruktúry procesora boli zdokumentované a realizované programy s volaním jednoduchých modulov písaných v jazyku assebler v kombinácii s použitím tried mbed.

S využitím jazyka C boli popísané možnosti prístupu ku konfiguračným registrom a pre STM32F042F6P6 boli vytvorené programy umožňujúce použiť periférie aj konfiguráciu, akú triedy mbed neponúkajú.

V rámci tejto bakalárskej práce se podarilo splnit všetky body zadania a niektoré časti boli doplnené aj nad rámec zadania (kapitola 11.4). Vytvorené materiály a implementované ladiace nástroje môžu zlepšiť proces laboratórnej výuky a takisto môžu slúžiť ako podporný materál pre záujemcov o elektroniku a programovanie mikrokontrolérov s jadrom ARM.

Literatúra

[1] EMBEDDED FEL ČVUT. *Laboratoře z průmyslové elektroniky a senzorů (online)*. https://embedded.fel.cvut.cz/kurzy/lpe_sw. (cit. 2019-05-17).

[2] Mbed. *NUCLEO-F303RE (online)*. https://os.mbed.com/platforms/ST-Nucleo-F303RE/. (cit. 2019-05-21).

[3] Mbed. *NUCLEO-F042K6 (online)*. https://os.mbed.com/platforms/ST-Nucleo-F042K6/. (cit. 2019-05-17).

[4] STMicroelectronics. *STM32F042x4 STM32F042x6, Rev. 5*. 2017. https://www.st.com/resource/en/datasheet/stm32f042c6.pdf.

[5] Mbed. *Full API list (online)*. https://os.mbed.com/docs/mbed-os/v5.11/apis/index.html. (cit. 2019-05-17).

[6] T. Toulson, R. Wilmshurst. *Fast and Effective Embedded Systems Design - Applying the ARM mbed*. Druhé vydanie. Newnes , 2016 . ISBN 978-0-08-100880-5.

[7] STMicroelectronics. *STM32 Nucleo (64 pins) schematics, Rev. C.3*. 2014. https://www.st.com/resource/en/schematic_pack/nucleo_64pins_sch.zip.

[8] STMicroelectronics. *RM0091 Reference manual, Rev. 9*. 2017. https://www.st.com/resource/en/reference_manual/dm00031936.pdf.

[9] Michael Barr. *Programming Embedded Systems in C and C++*. Prvé vydanie. O'Reilly , 1999 . ISBN 1-56592-354-5. https://pdfs.semanticscholar.org/7f3c/3c7925eb00589f91036ab081549d695aba37.pdf.

[10] ARMmbed. *TARGET_STM Serial (online)*. 2018. https://github.com/ARMmbed/mbed-os/blob/master/targets/TARGET_STM/. (cit. 2019-05-21).

[11] Javier Orensanz. *Advanced Debugging for Cortex-M Microcontrollers (online)*. https://www.arm.com/files/pdf/AT_-_Advanced_Debug_of_Cortex-M_Systems.pdf. (cit. 2019-05-21).

[12] *ANSI Escape sequences (online)*. 2019. http://ascii-table.com/ansi-escape-sequences-vt-100.php. (cit. 2019-05-12).

[13] STMicroelectronics. *PM0214 STM32 Cortex-M4 MCUs and MPUs programming manual, Rev. 7*. 2019. https://www.st.com/resource/en/programming_manual/dm00046982.pdf.

[14] ARM Limited. *Cortex-M4 Technical Reference Manual*. 2009, 2010. https://developer.arm.com/docs/100166/0001.

[15] Mbed. *Calling an Assembly Language Function from C/C++ (online)*. https://os.mbed.com/cookbook/Assembly-Language. (cit. 2019-05-21).

[16] Joseph Yiu. *The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ Processors*. Druhé vydanie. Elsevier , 2015 . ISBN 978-0-12-803277-0.

[17] STMicroelectronics. *RM0316 Reference manual, Rev 8*. 2017.
https://www.st.com/resource/en/reference_manual/dm00043574.pdf.

[18] ARMmbed. *NUCLEO-F042K6 system constants (online)*. 2018.
https://github.com/ARMmbed/mbed-os/blob/master/targets/TARGET_STM/TARGET_STM32F0/TARGET_NUCLEO_F042K6/device/stm32f042x6.h. (cit. 2019-05-21).

[19] STMicroelectronics. *AN2867 Oscillator design guide for STM8AF/AL/S and STM32 microcontrollers, Rev. 11*. 2017.
https://www.st.com/resource/en/application_note/cd00221665.pdf.

[20] ARMmbed. *NUCLEO-F042K6 system clock (online)*. 2018.
https://github.com/ARMmbed/mbed-os/blob/master/targets/TARGET_STM/TARGET_STM32F0/TARGET_NUCLEO_F042K6/device/system_clock.c. (cit. 2019-05-21).

Príloha A
Skratky

AD prevodník	Analógovo digitálny prevodník.
APSR	Application program status register. Register mikrokontroléra, ktorý v sebe uchováva flag bity aritmeticko-logickej jednotky.
ARM	Advanced RISC Machine. Typ architektúry procesorov.
ASCII	ASCII American Standard Code for Information Interchange (americký štandardný kód pre výmenu informácií). Kódová tabuľka definujúca znaky anglickej abecedy a iné používané znaky.
BP	Bakalárska práca.
BRR	Bit reset register. Register typu $write-only$, nastavením bitov do 1 je možné resetovať výstupné logické hodnoty odpovedajúcich pinov.
BSRR	Bit set reset register. Register typu $write-only$, nastavením horných(resp. dolných) 16 bitov do 1 je možné nastaviť logické hodnoty odpovedajúcich pinov do log.0(resp. log.1).
COM port	Communication port (komunikačný port). Najbežnejší typ sériového rozhrania.
GND	Zem mikrokontroléra.
GPIO	General purpose input output (univerzálny vstupne-výstupný pin).
HSE	High speed external clock signal (vysokorýchlostný externý hodinový signál).
HSI	High speed internal clock signal (vysokorýchlostný vnútorný hodinový signál).
IDR	Input data register. Register, ktorý v sebe drží vstupné logické úrovne pinov jedného portu.
LED	Light-Emitting Diode (dióda emitujúca svetlo). Polovodičová súčiastka schopná vyžarovať svetlo.
LR	Link register. Register, ktorý v sebe drží návratovú adresu z volanej funkcie.
ODR	Output data register. Register, ktorý v sebe drží výstupné logické úrovne pinov jedného portu.
PC	Personal computer (osobný počítač).
PWM	Pulse width modulation (pulzne šírková modulácia).
SP	Stack pointer. Register mikrokontroléra, ktorý v sebe drží adresu vrchola zásobníka.
UART	Universal asynchronous receiver-transmitter. Zariadenie pre asynchrónnu sériovú komunikáciu.
USB	Universal Serial Bus (univerzálna sériová zbernica). Zbernica používaná hlavne na komunikáciu počítača s perifériami.
VCP	Virtuálny komunikačný port.
VDD	Kladné napájacie napätie mikrokontroléra.
VSS	Záporné napájacie napätie mikrokontroléra.

Príloha B
Prehľad funkcií jednotlivých tried mbed

DigitalOut			
	Funkcia	*Popis, vstupné parametre, návratová hodnota*	*Príklad*

	DigitalOut	Inicializácia digitálneho výstupu na pine (napr. PA5) s nejakým názvom (napr. led)
		1: názov pinu, ktorý konfigurujeme ako DigitalOut,
		2 (voliteľný): premenná typu int, log. hodnota na výstupe (predvolene log.0)
			//s log.0 na vystupe
			DigitalOut led(PA_5);
			
			//s log.1 na vystupe
			DigitalOut led(PA_5, 1);

	write	Zápis logickej hodnoty na výstup brány
		premenná typu int, požadovaná logická hodnota napätia na výstupe
		0 -> log.0, ostatné-> log.1
			// standardny zapis
			led.write(1);
			
			// skrateny zapis
			led = 1;

int	read	Prečítanie logickej hodnoty, ktorá je na **výstupe** pinu
		
		premenná typu int s hodnotou 0 alebo 1 podľa log. hodnoty na výstupe pinu
			// standardny zapis
			int state = led.read();
			
			//skrateny zapis
			int state = led;

int	is_connected	kontrola správnosti inicializácie pinu ako DigitalOut
		
		premenná typu int s hodnotou 1, ak bol pin správne inicializovaný, inak 0
			if (led.is_connected() == 0){
			
			return;
			
			}

DigitalIn			
	Funkcia	*Popis, vstupné parametre, návratová hodnota*	*Príklad*

	DigitalIn	Inicializácia digitálneho vstupu na pine s nejakým názvom,
		1: názov pinu, ktorý konfigurujeme ako DigitalIn
		2 (voliteľný): požadovaný mód pinu (PullUp, PullDown, PullNone), PullNone je predvolený mód
			// konfiguracia bez pull rezistora
			DigitalIn button(PA_4);
			
			// konfiguracia s pull rezistorom
			DigitalIn button(PA_4, PullDown);

	mode	Nastavenie módu vstupného pinu požadovaný mód (PullUp, PullDown, PullNone), PullNone je predvolený mód	button.mode(PullUp);

int	read	Prečítanie logickej hodnoty napätia na vstupe pinu
		
		premenná typu int s hodnotou 0 alebo 1 podľa log. hodnoty na vstupe
			// standardny zapis
			int state = button.read();
			
			// skrateny zapis
			int state = button;

int	is_connected	kontrola správnej inicializácie pinu ako DigitalIn
		
		1, ak bol pin správne inicializovaný alebo 0 v opačnom prípade
			if (button.is_connected() == 0){
			return;
			}

DigitalInOut			
	funkcia	*popis, vstupné parametre, návratová hodnota*	*príklad*
	DigitalInOut	Inicializácia pinu ako vstupne-výstupného s nejakým názvom 1: názov pinu, ktorý konfigurujeme ako DigitalInOut Parametre 2, 3 a 4 sú voliteľné, buď sa použijú všetky alebo ani jeden. 2: typ pinu (PIN_INPUT, PIN_OUTPUT), predvolený je PIN_INPUT 3: požadovaný mód pinu (PullUp, PullDown, PullNone, OpenDrain), PullNone je predvolený mód 4: premenná typu int, log. hodnota na výstupe, predvolená je log.0	//input, pull-none, log.0 na vystupe DigitalInOut alternate(PA_3); /* pin PA3 nakonfigurovany ako vstupny s vnutornym pull-up rezistorom s log.1 na vystupe po prepnuti na vystupny */ DigitalInOut alternate(PA_3, PIN_INPUT, PullUp, 1);
	mode	Nastavenie módu pinu. požadovaný mód (PullUp, PullDown, PullNone, OpenDrain), PullNone je predvolený mód.	alternate.mode(PullUp);
	input	Nastavenie pinu na vstupný.	alternate.input();

int	read	Ak je pin nastavený ako vstupný: prečítanie logickej hodnoty napätia na vstupe pinu
		Ak je pin nastavený ako výstupný: prečítanie logickej hodnoty napätia na výstupe pinu
		premenná typu int s hodnotou 0 alebo 1 podľa log. hodnoty na vstupe (resp. výstupe) pinu
			// standardny zapis
			int state = alternate.read();
			//skrateny zapis
			int state = alternate;

	output	Nastavenie pinu na výstupný.	alternate.output();

	write	Ak je pin nastavený ako výstupný: zápis logickej hodnoty na výstup brány.
		V opačnom prípade sa hodnota zapíše na výstup po zavolaní funkcie *output().*
		premenná typu int, požadovaná logická hodnota napätia na výstupe (0 -> log.0, ostatné -> log.1)
			// standardny zapis
			alternate.write(1);
			//skrateny zapis
			alternate = 1;

int	is_connected	Kontrola správnej inicializácie pinu ako DigitalInOut
		1, ak bol pin správne inicializovaný alebo 0 v opačnom prípade
			if (alternate.is_connected()== 0){
			return;
			}

InterruptIn			
	funkcia	*popis, vstupné parametre, návratová hodnota*	*príklad*

	InterruptIn	Inicializácia digitálneho vstupu s možnosťou prerušenia na pine s nejakým názvom (napr. button). Prerušenie je predvolene povolené.
		1: názov pinu, ktorý konfigurujeme ako InterruptIn
		2 (voliteľný): požadovaný mód pinu (PullUp, PullDown, PullNone)
		PullNone je predvolený mód.
			// konfiguracia bez pull rezistora
			InterruptIn button(PA_4);
			// konfiguracia s pull rezistorom
			InterruptIn button(PA_4, PullDown);

	mode	Nastavenie vstupného módu pinu. požadovaný mód (PullUp, PullDown, PullNone, OpenDrain) PullNone je predvolený mód.	button.mode(PullUp);
int	read	Prečítanie logickej hodnoty napätia na vstupe pinu. Premenná typu int s hodnotou 0 alebo 1 podľa log. hodnoty napätia na vstupe	int state = button.read(); int state = button; //skrateny zapis

	rise	Priradenie funkcie prerušenia pri nábežnej hrane signálu na vstupe pinu.
		adresa funkcie obsluhujúcej prerušenie
			void func_rise(){
			...
			}
			int main(){
			...
			button.rise(&func_rise);
			...
			}

	fall	Priradenie funkcie prerušenia pri spádovej hrane signálu na vstupe pinu.
		adresa funkcie obsluhujúcej prerušenie
			void func_fall(){
			...
			}
			int main(){
			...
			button.fall(&func_fall);
			...
			}

	enable_irq	Povolenie prerušenia na pine.	button.enable_irq();
	disable_irq	Zakázanie prerušenia na pine.	button.disable_irq();

int	is_connected	Kontrola správnej inicializácie pinu ako InterruptIn.
		1, ak bol pin správne inicializovaný alebo 0 v opačnom prípade.
			if (button.is_connected() == 0){
			return;
			}

PwmOut			
	funkcia	*popis, vstupné parametre, návratová hodnota*	*príklad*
	PwmOut	Inicializácia generátora PWM na pine s PWM funkcionalitou s nejakým názvom (napr. led). názov pinu, ktorý konfigurujeme ako PWM generátor	PwmOut led(PA_6);
	period	Nastavenie periódy PWM signálu na hodnotu v sekundách s rozlíšením na mikrosekundy. premenná typu float, dĺžka periódy PWM v sekundách s max. rozlíšením na mikrosekundy	led.period(9.999);
	period_ms	Nastavenie periódy PWM signálu na hodnotu v milisekundách, strieda PWM sa zachová. premenná typu int, dĺžka periódy PWM v milisekundách	led.period_ms(246);
	period_us	Nastavenie periódy PWM signálu na hodnotu v mikrosekundách, strieda PWM sa zachová. premenná typu int, dĺžka periódy PWM v mikrosekundách	led.period_us(3000);
	pulsewidth	Nastavenie trvania pulzu PWM vrámci periódy na hodnotu v sekundách s rozlíšením na mikrosekundy. premenná typu float v rozsahu <0, perióda>	led.pulsewidth(3.333);
	pulsewidth_ms	Nastavenie trvania pulzu PWM vrámci periódy na hodnotu v milisekundách. premenná typu int, trvanie pulzu v milisekundách	led.pulsewidth_ms(12);
	pulsewidth_us	Nastavenie trvania pulzu PWM vrámci periódy na hodnotu v mikrosekundách. premenná typu int, trvanie pulzu v mikrosekundách	led.pulsewidth_us(15);

	write	Nastavenie striedy PWM signálu pri zachovaní dĺžky periódy.
		premenná typu float v rozsahu <0, 1>
			// standardny zapis
			led.write(0.6);
			// skrateny zapis
			led = 0.6;

float	read	Prečítanie aktuálnej hodnoty striedy PWM.
		premenná typu float v rozsahu <0, 1>
			// standardny zapis
			float stride = led.read();
			// skrateny zapis
			float stride = led;

Serial			
	funkcia	*popis, vstupné parametre, návratová hodnota*	*príklad*

	Serial	Vytvorenie objektu triedy Serial, inicializácia UARTU
		1: názov TX pinu
		2: názov RX pinu
		3: (voliteľný): premenná typu int, baudrate, predvolene 9600 Bd/s.
			//pouzitie predvoleneho baudrate
			Serial pc (PA_2, PA_3); // 9600Bd/s
			
			// pouzitie ineho baudrate
			Serial pc (PA_2, PA_3, 115200);

	baud	Nastavenie rýchlosti komunikácie. premenná int, požadovaný baudrate	pc.baud(19200);
	format	nastavenie formátu správy 1: premenná typu int, počet dátových bitov (7, 8 alebo 9, predvolene 8) 2: parita->SerialBase::None(predvolená) SerialBase::Even, SerialBase::Odd 3: počet stopbitov (1, 2, predvolene 1)	pc.format(8, SerialBase::Odd,1);
char	getc	Prečítanie jedného znaku zo COM portu premenná typu char, prečítaný znak	char sign = pc.getc();

bool	readable	Kontrola, či je možné z COM portu prečítať znak.
		
		true, ak je možné zo sériového portu prečítať znak, false v opačnom prípade
			if (pc.readable()){
			  char sign = pc.getc();
			}

int	putc	Zápis jedného znaku do COM portu premenná typu char, zapisovaný znak ASCII hodnota zapísaného znaku alebo záporné číslo v prípade chyby zápisu.	char sign = 'a'; pc.putc(sign);

bool	writable	Kontrola, či je možné do COM portu zapísať znak.
		
		true, ak je možné do sériového portu zapísať znak, false v opačnom prípade
			char sign = 'a';
			if (pc.readable()){
			  pc.putc(sign);
			}

	attach	Priradenie funkcie prerušenia vyvolanom po prijatí znaku.
		adresa funkcie obsluhujúcej prerušenie
			void func(){
			…
			}
			int main(){
			  …
			  pc.attach(&func);
			  …
			}

int	printf	Zápis formátovaného reťazca do sériového portu.
		formátovaný reťazec
		Počet zapísaných znakov alebo záporné číslo v prípade chyby zápisu.
			float value = 3.14;
			char name[] = "pi";
			pc.printf("hodnota %s je %f\n\r",name, value);

int	scanf	Prečítanie formátovaného reťazca z COM portu a priradenie prečítaných hodnôt premenným. Čítanie znakov sa ukončí pri prvom znaku porušujúcom očakávaný formát.
		tvar očakávaného reťazca
		Počet priradených hodnôt premenným.
			int age;
			char sex;
			pc.printf("zadaj svoje pohlavie(M/F), vek");
			int read_params = pc.scanf("%c %d",&sex,&age);

USBSerial			
	funkcia	*popis, vstupné parametre, návratová hodnota*	*príklad*
	USBSerial	Vytvorenie objektu triedy USBSerial, vytvorenie virtuálneho COM portu.	USBSerial pc; wait(2); /* pre spravne vytvorenie COM portu */
char	getc	Prečítanie jedného znaku z virtuálneho sériového portu (VCP). premenná typu char, prečítaný znak	char sign = pc.getc();

uint8_t	available	Kontrola počtu znakov, ktoré je možné z VCP prečítať
		
		premenná typu uint8_t, počet znakov, ktoré môžu byť prečítané
			if (pc.available() != 0){
			  char sign = pc.getc();
			}

int	putc	zápis jedného znaku do virtuálneho sériového portu zapisovaný znak ASCII hodnota zapísaného znaku alebo záporné číslo v prípade chyby	char sign = 'a'; pc.putc(sign);

	attach	Priradenie funkcie prerušenia vyvolanom po prijatí znaku. **Pri použití *getchar()* v ISR program 'spadne'**
		adresa funkcie obsluhujúcej prerušenie
			void func(){
			...
			}
			int main(){
			  ...
			  pc.attach(&func);
			  ...
			}

int	printf	Zápis formátovaného reťazca do VCP
		formátovaný reťazec
		Počet zapísaných znakov alebo záporné číslo v prípade chyby zápisu.
			float value = 3.14;
			char name[] = "pi";
			pc.printf("hodnota %s je %f\n\r",name, value);

int	scanf	Prečítanie formátovaného reťazca VCP a priradenie prečítaných hodnôt premenným. Čítanie znakov sa ukončí pri prvom znaku porušujúcom očakávaný formát.
		tvar očakávaného reťazca
		počet priradených hodnôt premenným
			int age;
			char sex;
			pc.printf("zadaj svoje pohlavie(M/F), vek");
			int read_params = pc.scanf("%c %d",&sex,&age);

bool	writeBlock	Zápis bloku znakov do VCP. 1: premenná typu uint8_t*, ukazateľ na pole znakov, ktoré sa 2: premenná typu uint16_t , počet znakov, ktore sa do virtuálneho sériového portu pošlú true v prípade úspešného odoslenia false v opačnom prípade	char array[] = "toto je pole znakov"; pc.writeBlock((uint8_t *)& array, 12);

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();

Timer			
	funkcia	*popis, vstupné parametre, návratová hodnota*	*príklad*
	Timer	Vytvorenie objektu triedy Timer.	Timer stopwatch;
	start	spustenie časovača	stopwatch.start();
float	read	Prečítanie aktuálnej hodnoty časovača. premenná typu float, aktuálna hodnota časovača v sekundách s rozlíšením na µs	float timestamp = stopwatch.read();
int	read_ms	Prečítanie aktuálnej hodnoty časovača. premenná typu int, hodnota časovača v ms	int timestamp = stopwatch.read_ms();
int	read_us	prečítanie aktuálnej hodnoty časovača premenná typu int, hodnota časovača v µs	int timestamp = stopwatch.read_us();
uint64_t	read_high_resolution_us	Prečítanie aktuálnej hodnoty časovača vo vysokom rozlíšení (pre dlhé časové úseky). premenná typu uint64_t, aktuálna hodnota časovača v mikrosekundách	uint64_t timestamp = stopwatch.read_high_resolution_us();
	stop	zastavenie časovača	stopwatch.stop();
	reset	vynulovanie časovača	stopwatch.reset();

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();

Príloha C
Prehľad funkcií vytvorených debugovacích tried

Debug_led			
#include "Debug.h" //zahrnutie kniznice DEBUG_UNIVERSAL do programu			
	Funkcia	*Popis, vstupné parametre, návratová hodnota*	*Príklad*

	Debug_led	Konfigurácia debugovacej LED a tlačidla.
		1: názov pinu s debugovacou LED
		2: názov pinu s debugovacím tlačidlom
		3 (voliteľný): zapojenie tlačidla v obvode "BUTTON_VDD", "BUTTON_VCC", "BUTTON_GND" (predvolene)
			//v poradi LED a tlacidlo
			// tlacidlo je zapojené na zem
			Debug_led debug(PA_5, PA_6);
			// tlacidlo je zapojene na VDD
			Debug_led debug(PA_5, PA_6, "BUTTON_VDD");

	breakpoint	Zastavenie behu programu a indikovanie breakpointu LED až do stlačenia a následného pustenia tlačidla
		(voliteľný) premenná typu int, periodický počet bliknutí LED pri breakpointe,
		Bez použitia parametra LED bliká konštantne.
			// konstantne blikanie LED
			debug.breakpoint();
			
			// periodicke blikanie LED po 4
			debug.breakpoint(4);

Debug_complete			
#include "Debug.h" //zahrnutie kniznice DEBUG_F303/DEBUG_F042F6P6 do programu			
	Funkcia	*Popis, vstupné parametre, návratová hodnota*	*Príklad*
	Debug_complete	Inicializácia debugovacieho sériového portu 1: názov TX pinu 2: názov RX pinu 3: (voliteľný): premenná typu int, požadovaná hodnota baudrate, predvolene 115200 Bd/s.	//baudrate 115200 Bd/s Debug_complete pc (PA_2, PA_3);
	breakpoint	Odoslanie informácií o konfigurácii určitých periférií do COM portu a zastavenie programu až do prijatia ľubovoľného znaku z COM portu. aktuálny riadok programu -> **__LINE__**	pc.breakpoint(__LINE__);

Debug_serial			
#include "Debug.h" //zahrnutie kniznice DEBUG_UNIVERSAL do programu			
	Funkcia	*Popis, vstupné parametre, návratová hodnota*	*Príklad*
	Debug_serial	Inicializácia debugovacieho sériového portu s čiastocnou funkcionalitou triedy Serial. 1: názov TX pinu 2: názov RX pinu 3: (voliteľný): premenná typu int, požadovaná hodnota baudrate, predvolene 115200 Bd/s.	//baudrate 115200 Bd/s Debug_serial pc (PA_2, PA_3);
	breakpoint	Odoslanie informácií o posledných 3 breakpointoch do COM portu a zastavenie behu programu až do prijatia ľubovoľného znaku z COM portu. aktuálny riadok programu -> **__LINE__**	pc.breakpoint(__LINE__);

	breakpoint	Odoslanie do COM portu informácií o posledných 3 breakpointoch, názvu a hodnoty premennej typu **char**, **char***, **float** alebo **int** a zastavenie programu až do prijatia ľubovoľného znaku z COM portu.
		1: aktuálny riadok programu -> **__LINE__**
		2: reťazec, názov premennej -> **name(var)**
		3: premenná typu **char**, **char***, **float** alebo **int**
			float pi = 3.14;
			pc.breakpoint(__LINE__, name(pi), pi);
			int val = 123;
			pc.breakpoint(__LINE__, name(val), val);

	breakpoint	Odoslanie do COM portu informácií o posledných 3 breakpointoch, hodnoty jedného registra a zastavenie programu až do prijatia ľubovoľného znaku z COM portu. 1: aktuálny riadok programu -> **__LINE__** 2: adresa registra	uint32_t gpioa_moder = 0x48000000; pc.breakpoint(__LINE__, gpioa_moder);
char	getc	Prečítanie jedného znaku z COM portu. premenná typu char, prečítaný znak	char sign = pc.getc();

bool	readable	Kontrola, či je možné z COM portu prečítať znak.
		true, ak je možné zo sériového portu prečítať znak, false v opačnom prípade
			if (pc.readable()){
			char sign = pc.getc();
			}

int	putc	Zápis jedného znaku do COM portu a zobrazenie v sériovej časti okna. premenná typu char, zapisovaný znak ASCII hodnota zapísaného znaku alebo záporné číslo v prípade chyby zápisu.	char sign = 'a'; pc.putc(sign);

bool	writable	Kontrola, či je možné do COM portu zapísať znak.
		true, ak je možné do sériového portu zapísať znak, false v opačnom prípade
			char sign = 'a';
			if (pc.readable()){
			pc.putc(sign);
			}

int	printf	Zápis formátovaného reťazca do COM portu a zobrazenie v sériovej časti okna. formátovaný reťazec Počet zapísaných znakov alebo záporné číslo v prípade chyby zápisu.	float value = 3.14; char name[] = "pi"; pc.printf ("hodnota %s je %f\n\r", name, value);

int	scanf	Prečítanie formátovaného reťazca z COM portu a priradenie prečítaných hodnôt premenným. Čítanie znakov sa ukončí pri prvom znaku porušujúcom očakávaný formát.
		tvar očakávaného reťazca
		počet priradených hodnôt premenným
			int age;
			char sex;
			pc.printf("zadaj svoje pohlavie(M/F), vek");
			int read_params = pc.scanf ("%c %d",&sex,&age);

Debug_register			
#include "Debug.h" //zahrnutie kniznice DEBUG_UNIVERSAL do programu			
	Funkcia	*Popis, vstupné parametre, návratová hodnota*	*Príklad*
	Debug_register	Inicializácia debugovacieho sériového portu s čiastocnou funkcionalitou triedy Serial. 1: názov TX pinu 2: názov RX pinu 3: (voliteľný): premenná typu int, požadovaná hodnota baudrate, predvolene 115200 Bd/s.	//baudrate 115200 Bd/s Debug_register pc (PA_2, PA_3);
	breakpoint	Odoslanie do COM portu informácií o posledných 3 breakpointoch, hodnoty jedného registra a zastavenie behu programu. Podľa prijatého znaku z COM portu sa buď - prečíta register na vyššej(nižšej) adrese: 'i' ('k') - prečíta register na manuálne zadanej adrese: 'r' - prepíše hodnota registra na zadanej adrese na zadanú hodnotu: 'w' - ukončí breakpoint: Enter alebo 'p' Pri manuálnom zadávaní čísla sa kurzor posúva pomocou 'j' a 'l' a hodnota sa potvrdí(vymaže) pomocou Enter(esc) Pri pristupovaní na rezervovanú adresu alebo zápise nepovolenej hodnoty dôje k Hard faultu. 1: aktuálny riadok programu -> __**LINE**__ 2: adresa registra	uint32_t gpioa_moder = 0x48000000; pc.breakpoint(__LINE__, gpioa_moder);
char	getc	Prečítanie jedného znaku zo COM portu premenná typu char, prečítaný znak	char sign = pc.getc();

bool	readable	Kontrola, či je možné z COM portu prečítať znak.
		true, ak je možné zo sériového portu prečítať znak, false v opačnom prípade
			if (pc.readable()){
			char sign = pc.getc();
			}

int	putc	Zápis jedného znaku do COM portu a zobrazenie v sériovej časti okna. premenná typu char, zapisovaný znak ASCII hodnota zapísaného znaku alebo záporné číslo v prípade chyby zápisu.	char sign = 'a'; pc.putc(sign);

bool	writable	Kontrola, či je možné do COM portu zapísať znak.
		true, ak je možné do sériového portu zapísať znak, false v opačnom prípade
			char sign = 'a';
			if (pc.readable()){
			pc.putc(sign);
			}

int	printf	Zápis formátovaného reťazca do COM portu a zobrazenie v sériovej časti okna. formátovaný reťazec Počet zapísaných znakov alebo záporné číslo v prípade chyby zápisu.	float value = 3.14; char name[] = "pi"; pc.printf ("hodnota %s je %f\n\r", name, value);

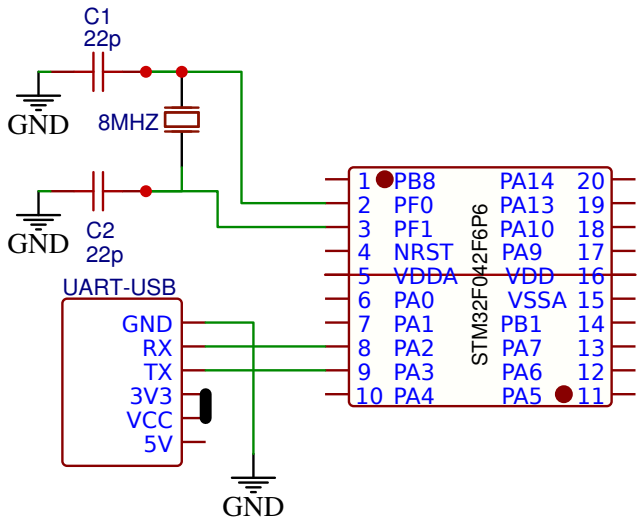
int	scanf	Prečítanie formátovaného reťazca z COM portu a priradenie prečítaných hodnôt premenným. Čítanie znakov sa ukončí pri prvom znaku porušujúcom očakávaný formát.
		tvar očakávaného reťazca
		počet priradených hodnôt premenným
			int age;
			char sex;
			pc.printf("zadaj svoje pohlavie(M/F), vek");
			int read_params = pc.scanf ("%c %d",&sex,&age);

Debug_register_print			
#include "Debug.h" //zahrnutie kniznice DEBUG_UNIVERSAL do programu			
	Funkcia	*Popis, vstupné parametre, návratová hodnota*	*Príklad*
	Debug_register_print	Inicializácia debugovacieho sériového portu. 1: názov TX pinu 2: názov RX pinu 3: (voliteľný): premenná typu int, požadovaná hodnota baudrate, predvolene 115200 Bd/s.	//baudrate 115200 Bd/s Debug_register_print pc (PA_2, PA_3);
	format	Nastavenie formátu odosielanej správy. 4 premenné typu int, v poradí formát čísla breakpointu, čísla riadku, adresy registra a hodnoty registra. Ak je hodnota daného parametra: 0, hodnota sa neodosiela 1, hodnota sa odošle v hexadecimálnom tvare 2, hodnota sa odošle v decimálnom tvare 3, (iba v prípade hodnoty registra): hodnota sa odošle v binárnom tvare Predvolený tvar je: číslo breakpointu a riadok decimálne, adresa registra hexadecimálne a hodnota registra binárne.	pc.format(2, 2, 1, 1); /*cislo breakpointu a riadok decimalne, adresa s hodnotou registra hexadecimalne */
	breakpoint	Odoslanie do COM portu informácií o posledných 3 breakpointoch, hodnoty jedného registra a zastavenie behu programu. Podľa prijatého znaku z COM portu sa buď - prečíta register na vyššej(nižšej) adrese: 'i' ('k') - prečíta register na manuálne zadanej adrese: 'r' - prepíše hodnota registra na zadanej adrese na zadanú hodnotu: 'w' - ukončí breakpoint: Enter alebo 'p' Pri manuálnom zadávaní čísla sa kurzor posúva pomocou 'j' a 'l' a hodnota sa potvrdí(vymaže) pomocou Enter(esc) Pri pristupovaní na rezervovanú adresu alebo zápise nepovolenej hodnoty dôje k Hard faultu. 1: aktuálny riadok programu -> __**LINE**__ 2: adresa registra	uint32_t gpioa_moder = 0x48000000; pc.breakpoint(__LINE__, gpioa_moder);

Príloha D
Zmena syst. hodín pre STM32F0

```
#include "Debug.h" // DEBUG_F042F6P6 library must imported to program

PwmOut led(PA_4); // PWM generator on PA4
Debug_complete pc(PA_2,PA_3, 115200);  // TX pin, RX pin, baudrate
// Function to turn PLL_HSE on, external HSE must be connected
void turn_PLL_HSE_on();
void turn_PLL_HSI_on(); // Function to turn PLL_HSI on
void turn_HSI48_on(); // Function to turn HSE48 on

int main() {
    led.period(1); // Set period of PWM on PA4 to 1 s
    led = 0.5; // Set duty cycle of PWM on PA4 to 50%
    pc.breakpoint(__LINE__); // breakpoint with actual line number
    turn_PLL_HSE_on(); // external HSE must be connected
    pc.breakpoint(__LINE__); // breakpoint with actual line number
    turn_HSI48_on();
    pc.breakpoint(__LINE__); // breakpoint with actual line number
    turn_PLL_HSI_on();
    pc.breakpoint(__LINE__); // breakpoint with actual line number
    while(1){}
}

void turn_PLL_HSE_on(){
    // Test if PLL is used as System clock
    if ((RCC->CFGR & RCC_CFGR_SWS) == RCC_CFGR_SWS_PLL){
        RCC->CFGR &= (uint32_t) (~RCC_CFGR_SW);//Select HSI as system CLK
        //Wait for HSI switched
        while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_HSI){}
        RCC->CR &= (uint32_t)(~RCC_CR_PLLON); // Disable the PLL
        // Wait until PLLRDY is cleared
        while((RCC->CR & RCC_CR_PLLRDY) != 0){}
        // Test if HSI48 is used as System clock
        }else if( (RCC->CFGR & RCC_CFGR_SWS) == RCC_CFGR_SWS_HSI48 ){
        RCC->CFGR &= (uint32_t) (~RCC_CFGR_SW);//Select HSI as system CLK
        // Wait for HSI switched
        while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_HSI){}
        RCC->CR2 &= (uint32_t)(~RCC_CR2_HSI48ON); // Disable the HSI48
        //Wait until HSI48RDY is cleared
        while((RCC->CR2 & RCC_CR2_HSI48RDY) != 0){}
    }
    RCC->CR|=RCC_CR_HSEON; // Enable the HSE clock
    while(!(RCC->CR&RCC_CR_HSERDY)){} //Wait until HSERDY is cleared
    // Set HSE/PREDIV as PLL source clock
    RCC->CFGR=(RCC->CFGR&(~RCC_CFGR_PLLSRC))|RCC_CFGR_PLLSRC_HSE_PREDIV;
    RCC->CR |= RCC_CR_PLLON; // Enable the PLL
    while((RCC->CR & RCC_CR_PLLRDY) == 0){} // Wait until PLLRDY is set
    RCC->CFGR |= (uint32_t) (RCC_CFGR_SW_PLL); //Select PLL as system CLK
    // Wait until the PLL is switched on
    while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL){}
}
```

```
void turn_PLL_HSI_on(){
    // Test if PLL is used as System clock
    if ((RCC->CFGR & RCC_CFGR_SWS) == RCC_CFGR_SWS_PLL){
        RCC->CFGR &= (uint32_t) (~RCC_CFGR_SW);//Select HSI as system CLK
        // Wait for HSI switched
        while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_HSI){}
        RCC->CR &= (uint32_t)(~RCC_CR_PLLON); // Disable the PLL
        // Wait until PLLRDY is cleared
        while((RCC->CR & RCC_CR_PLLRDY) != 0){}
        // Test if HSI48 is used as System clock
        }else if( (RCC->CFGR & RCC_CFGR_SWS) == RCC_CFGR_SWS_HSI48 ){
        // Select HSI as system clock
        RCC->CFGR &= (uint32_t) (~RCC_CFGR_SW);
        // Wait for HSI switched
        while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_HSI){}
        RCC->CR2 &= (uint32_t)(~RCC_CR2_HSI48ON); // Disable the HSI48
        //Wait until HSI48RDY is cleared
        while((RCC->CR2 & RCC_CR2_HSI48RDY) != 0){}
    }
    // Set HSI/PREDIV as PLL source clock
    RCC->CFGR=(RCC->CFGR&(~RCC_CFGR_PLLSRC))|RCC_CFGR_PLLSRC_HSI_PREDIV;
    RCC->CR |= RCC_CR_PLLON; //Enable the PLL
    while((RCC->CR & RCC_CR_PLLRDY) == 0){} //Wait until PLLRDY is set
    RCC->CFGR |= (uint32_t) (RCC_CFGR_SW_PLL); //Select PLL as system CLK
    //Wait until the PLL is switched on
    while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL){}
}

void turn_HSI48_on(){
    // Test if PLL is used as System clock
    if ((RCC->CFGR & RCC_CFGR_SWS) == RCC_CFGR_SWS_PLL){
        // Select HSI as system clock
        RCC->CFGR &= (uint32_t) (~RCC_CFGR_SW);
        // Wait for HSI switched
        while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_HSI){}
        RCC->CR &= (uint32_t)(~RCC_CR_PLLON); // Disable the PLL
        // Wait until PLLRDY is cleared
        while((RCC->CR & RCC_CR_PLLRDY) != 0){}
        //Test if HSI48 is used as System clock
    }else if( (RCC->CFGR & RCC_CFGR_SWS) == RCC_CFGR_SWS_HSI48 ){
        return;
    }
    RCC->CR2|=RCC_CR2_HSI48ON; // Enable the HSI48 clock
    // Wait until HSI48RDY is cleared
    while(!(RCC->CR2&RCC_CR2_HSI48RDY)){}
    // Select HSI48 as system CLK
    RCC->CFGR |= (uint32_t) (RCC_CFGR_SW_HSI48);
    // Wait until the HSI48 is switched on
    while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_HSI48){}
}
```



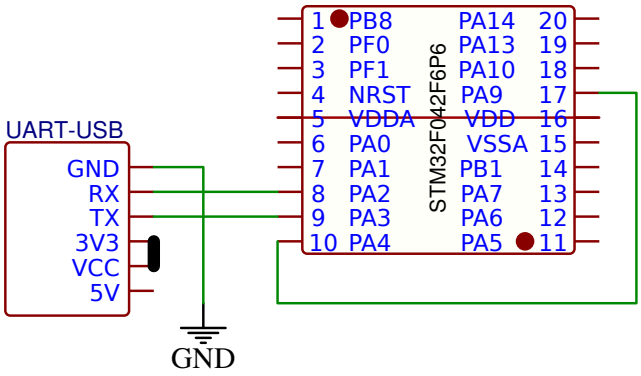
Obrázok D.1. Schéma zapojenia pre konfiguráciu HSE

Príloha E

Konfigurácia TIM1 CH2 na počítanie pulzov na vstupe pinu PA9 pre STM32F042F6P6

```
#include "mbed.h"
// Macro that returns max value of two input parameters
#define max(a, b) (((a) > (b)) ? (a) : (b))
PwmOut sensor_simulation(PA_4); // PWM generator on PA4
Serial pc(PA_2, PA_3, 115200); // TX pin, RX pin, baudrate
//Pulse counter on PA9 using TIM1_CH2
volatile float pwm_period = 0.3;
// ISR function to handle serial interrupt
void read_char(){
    char sign = pc.getc(); // get character
    if (sign == 'w' || sign == 'W'){ // Received character is w or W
        pwm_period += 0.01; // Increment value of pwm_period by 0.01
    }else if (sign == 's' || sign == 'S'){ //Received character is s or S
        // Decrement value of pwm_period by 0.01
        pwm_period = max(0.01, pwm_period-0.01);
    }
}
int main() {
    // Enable the peripheral clock of Timer 1
    RCC->APB2ENR |= RCC_APB2ENR_TIM1EN;
    // Enable the peripheral clock of GPIOA
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
    // Select AF2 on PA9 in AFRH for TIM1_CH2
    GPIOA->MODER=(GPIOA->MODER&~(GPIO_MODER_MODER9))|(GPIO_MODER_MODER9_1);
    // Select AF2 on PA9 in AFRH for TIM1_CH2
    GPIOA->AFR[1] |= 0x2 << GPIO_AFRH_AFSEL9_Pos;
    // Configure channel 2 to detect rising edges on the TI2
    TIM1->CCMR1 |=  TIM_CCMR1_CC2S_0;
    // Configure the input filter duration to 8
    TIM1->CCMR1 |= TIM_CCMR1_IC2F_0 | TIM_CCMR1_IC2F_1;
    // Select rising edge polarity by writing CC2P=0
    TIM1->CCER &= (uint16_t)(~TIM_CCER_CC2P);
    // Configure the timer in external clock mode 1 by writing SMS=111
    // Select TI2 as the trigger input source by writing TS=110
    TIM1->SMCR |= TIM_SMCR_SMS | TIM_SMCR_TS_2 | TIM_SMCR_TS_1;
    TIM1->CR1 |= TIM_CR1_CEN; // Enable the counter by writing CEN=1
    sensor_simulation.period(pwm_period); // Set period of PWM on PA4
    sensor_simulation = 0.5; // Set duty cycle of PWM on PA4 to 50\%
    // Attach read_char function to call in case of serial interrupt
    pc.attach(&read_char);
```

```
    while(1){
        pc.printf("\ec"); // Clear the whole screen
        wait_ms(10); // Wait until the screen is cleared
        pc.printf("SIMULATION OF  ENCODER\n\r");
        // Print counter value of TIM1
        pc.printf("TIMER VALUE: \%4d\n\r",TIM1->CNT);
        // Print value of variable pwm_period
        pc.printf("PWM PERIOD: \%4.2f\n\r",pwm_period);
        pc.printf("TO INCREASE(DECREASE) PWM PERIOD PRESS W(S)");
        sensor_simulation.period(pwm_period); // set PWM period of PA4
        wait(0.3); // wait 300 ms
    }
}
```



Obrázok E.2. Schéma zapojenia pre konfiguráciu TIM1 ako čítača pulzov na pine PA9