



**FACULTY
OF ELECTRICAL
ENGINEERING
CTU IN PRAGUE**

**Fakulta elektrotechnická
Katedra radioelektroniky**

Podpora cvičení

B0B37NSI

Návrh systémů IoT

**Jan Kočí
Stanislav Vitek**

28. února 2022

Obsah

| | | |
|----------|---|-----------|
| 1 | Úvod | 2 |
| 2 | VS Code a plugin PlatformIO | 3 |
| 2.1 | Instalace programu VS Code | 3 |
| 2.2 | Uživatelské rozhraní – základní rozložení | 4 |
| 2.3 | Instalace pluginů PlatformIO a C/C++ | 4 |
| 2.4 | Popis desky | 5 |
| 2.5 | Aplikace pro vyzkoušení funkčnosti desky | 6 |
| 2.6 | Založení projektu | 8 |
| 2.7 | Stavová lišta | 9 |
| 2.8 | Kompilace programu a využití hlavičkových souborů | 10 |
| 3 | První programy | 12 |
| 3.1 | Hello World! | 12 |
| 3.1.1 | Užitečné funkce VS Code | 13 |
| 3.1.2 | Debugování programu ve VS Code | 14 |
| 3.2 | Uživatelská tlačítka | 16 |
| 3.3 | Vstupně výstupní piny | 17 |
| 3.4 | Sériová komunikace | 19 |
| 3.4.1 | Třída BufferedSerial | 19 |
| 3.4.2 | Příklad – ovládání LED příkazy přes UART | 20 |
| 4 | Přerušení | 22 |
| 4.1 | Funkce třídy InterruptIn | 23 |
| 4.1.1 | Příklad – obsluha tlačítek s využitím přerušení | 23 |
| 4.2 | Třída UnbufferedSerial | 24 |
| 5 | Čítače | 26 |

| | | |
|----------|---|-----------|
| 5.1 | Třída Timeout | 26 |
| 5.2 | Třída Ticker | 27 |
| 5.3 | Třída Timer | 28 |
| 5.3.1 | Příklad – měření reakční doby | 29 |
| 5.4 | Třída PwmOut | 30 |
| 5.4.1 | Motivační příklad | 30 |
| 5.4.2 | PWM – Pulsní šířková modulace | 32 |
| 5.4.3 | Funkce třídy PwmOut | 32 |
| 5.4.4 | Příklad | 33 |
| 5.4.5 | Příklad | 34 |
| 6 | Analogový svět | 36 |
| 6.1 | Analogově-digitální převodník | 36 |
| 6.2 | Třída AnalogIn | 37 |
| 6.3 | Příklad – generování náhodných čísel | 37 |
| 7 | Sériové sběrnice | 39 |
| 7.1 | UART | 39 |
| 7.2 | I ² C | 40 |
| 7.3 | SPI | 41 |
| 8 | Komunikace s periferiemi | 43 |
| 8.1 | Zapojení senzorů na desce Thunderboard | 43 |
| 8.1.1 | Si7021 Relative Humidity and Temperature Sensor | 44 |
| 8.1.2 | Si1133 UV Index and Ambient Light Sensor | 44 |
| 8.1.3 | BMP280 Barometric Pressure Sensor | 44 |
| 8.1.4 | Si7210 Hall Effect Sensor | 45 |
| 8.1.5 | CCS811 Indoor Air Quality Gas Sensor | 45 |
| 8.1.6 | ICM-20648 6-Axis Inertial Sensor | 46 |
| 8.2 | Příklad – získání dat ze senzorů | 46 |
| 8.2.1 | Inicializace | 47 |
| 8.2.2 | Inicializace a čtení senzoru CCS811 | 48 |
| 8.2.3 | Inicializace a čtení senzoru BMP280 | 49 |
| 8.2.4 | Inicializace a čtení senzoru Si7021 | 49 |
| 8.2.5 | Inicializace a čtení senzoru Si7210 | 50 |
| 8.2.6 | Inicializace a čtení senzorů Si1133 a ICM20648 | 51 |

1 | Úvod

Cílem tohoto textu je seznámit čtenáře s principy fungování a programování mikrokontrolérů a možnostmi jejich využití v rámci IoT systémů. Mikrokontroléry lze programovat různými způsoby, které se liší komplexitou, nároky na porozumění periferiím mikrokontrolérů a dostupnými vývojovými nástroji. Je třeba zdůraznit, že programování mikrokontrolérů není v případě předmětu Návrh systémů IoT cílem, ale pouze prostředkem, takže je třeba co nejvíce využít předchozích znalostí studentů a nerozebírat do příliš velkých podrobností detaily obvodových zapojení a komunikačních protokolů.

Jako celkem dobrá volba pro začátečníky a mírně pokročilé programátory se jeví Mbed OS¹, který se daří úspěšně využívat ve výuce na ČVUT FEL[1]. Mbed OS poskytuje C++ API², relativně jednoduchý a pochopitelný abstraktní přístup k periferiím procesoru. Periferie jsou automaticky inicializovány (třeba konstruktorem třídy) a lze udržovat jejich perzistentní stav. Programátor se tak může více soustředit na funkcionality větších celků bez nutných počátečních znalostí detailů.

Při programování v Mbed OS lze využít hned několik možností:

- online prostředí³, které ovšem vyžaduje internetové připojení a nedisponuje nástroji na debuggování. Chybějící ladicí nástroje řešit instrumenálně např. blikáním LED nebo posíláním kontrolních výpisů po sériové lince do PC; nejedná se o optimální řešení.
- nástroje pro příkazový řádek (CLI)⁴, nástroje jsou dodávány i v podobě Docker obrazu
- integrované vývojové prostředí (IDE) Mbed Studio⁵, které je ale poměrně objemné a nedá se využít k jiným účelům.

Pro účely tohoto textu a předmětu využijeme editor Microsoft Visual Studio Code (dále jen VS Code) s podporou platformy PlatformIO, díky které je možné programy pohodlně ladit. Je to flexibilní editor, který lze rozšiřovat pomocí zásuvných modulů a používat k řadě dalších účelů.

¹<https://os.mbed.com/>

²<https://os.mbed.com/docs/mbed-os/v6.15/apis/index.html>

³<https://ide.mbed.com/compiler/> – je třeba registrace

⁴<https://os.mbed.com/docs/mbed-os/v6.15/build-tools/mbed-cli-2.html>

⁵<https://os.mbed.com/studio/>

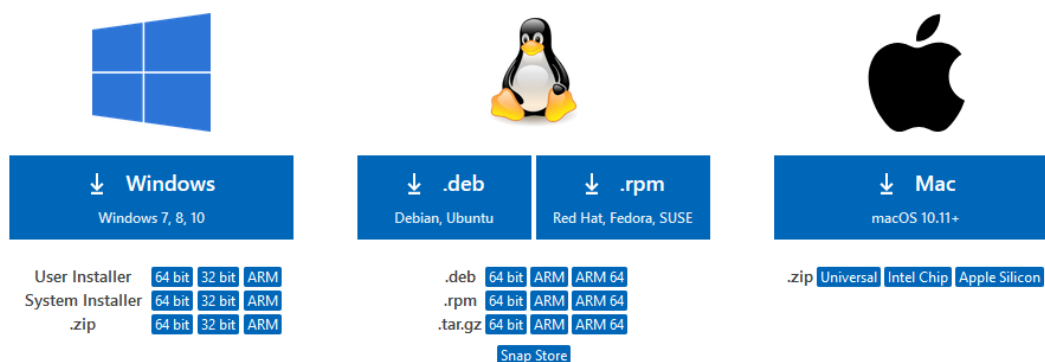
2 | VS Code a plugin PlatformIO

Před samotným programováním mikrokontroléru je zapotřebí si stáhnout a nainstalovat VS Code, seznámit se se samotným prostředím a dále nainstalovat plugin PlatformIO a s ním se také obeznámit. Zmíněná témata jsou předmětem této kapitoly.

Cílem samozřejmě není vysvětlovat veškerá možná nastavení, ale získat určitý náhled nad tím, co přibližně VS Code nabízí. Rozsáhlý zdroj informací najdeme zde¹, přičemž základní seznámení s VS Code je shrnuto v osmi videích².

2.1 Instalace programu VS Code

VS Code je editor zdrojového kódu dostupný pro operační systémy Windows, Linux a macOS, proto jsou na výběr instalační balíčky pro všechny tři operační systémy (viz obr 2.1). Stáhněte³ si patřičný balíček a program nainstalujte.



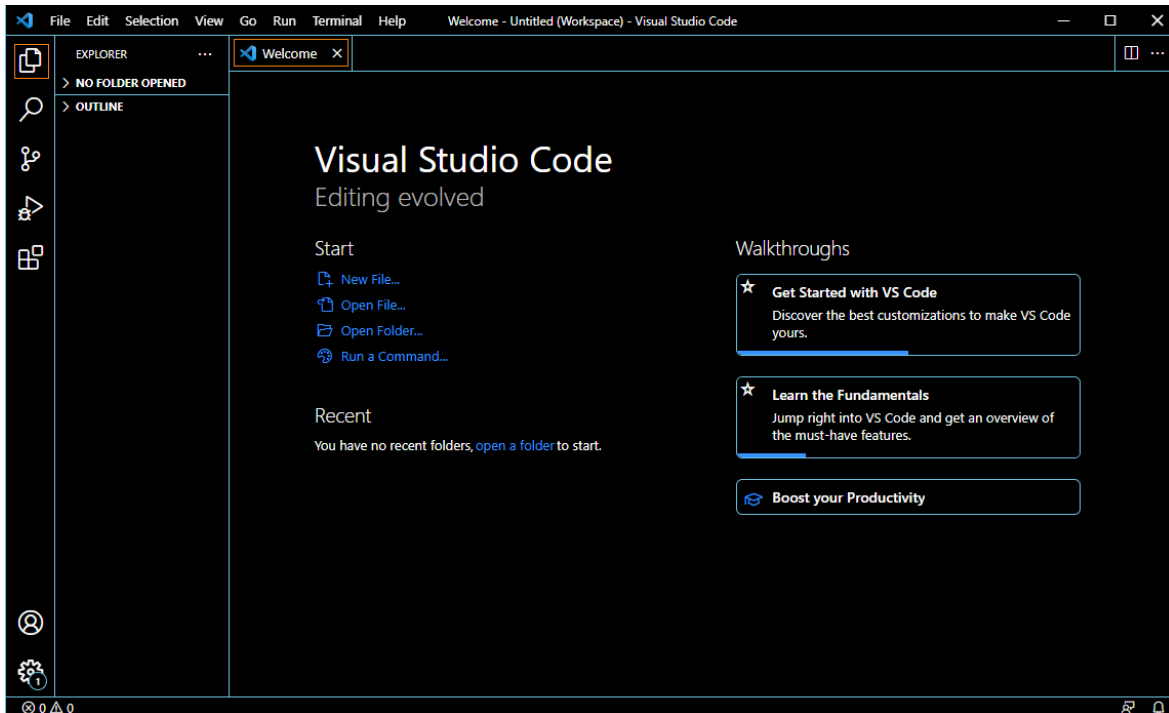
Obrázek 2.1: Instalační programy pro různé OS

Po úspěšném nainstalování a spuštění programu byste měli vidět úvodní stránku, která vypadá přibližně stejně jako ta na obrázku 2.2.

¹<https://code.visualstudio.com/docs>

²<https://code.visualstudio.com/docs/getstarted/introvideos>

³<https://code.visualstudio.com/Download>



Obrázek 2.2: Úvodní stránka VS Code

2.2 Uživatelské rozhraní – základní rozložení

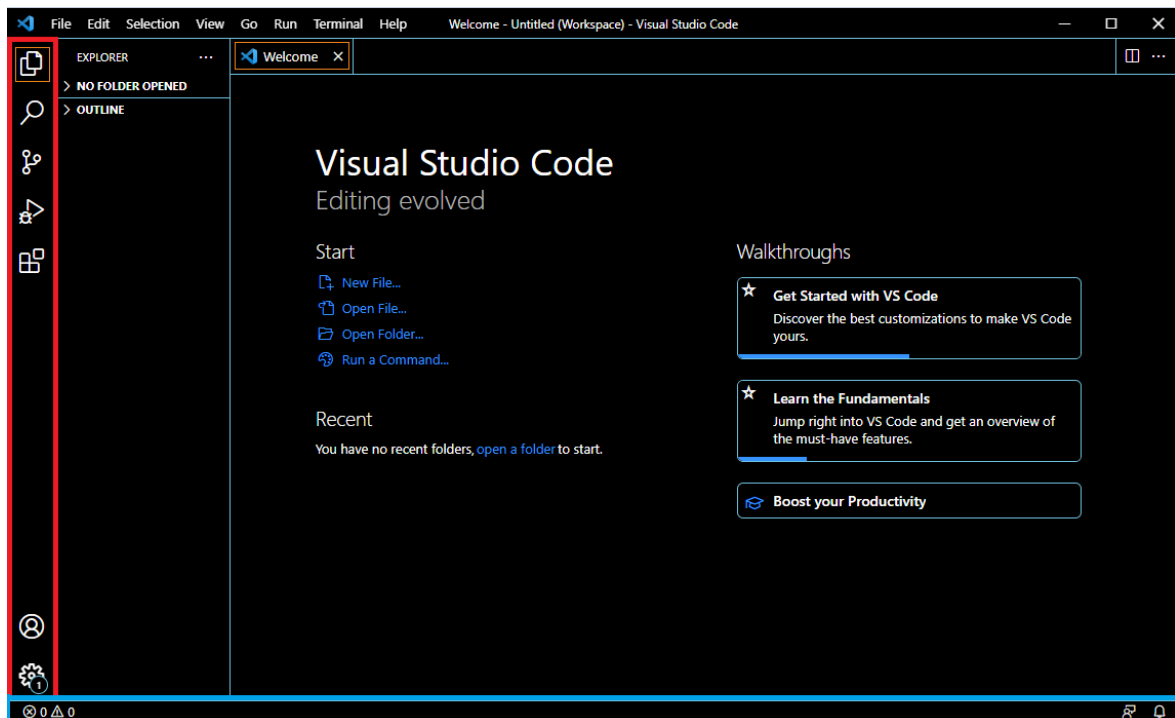
U úvodní stránky ještě na chvíli zůstaneme a popíšeme si základní rozložení uživatelského rozhraní (viz obr. 2.3). V levé části je zvýrazněna červeně lišta aktivit (z angl. Activity Bar) a dole stavová lišta (z angl. Status Bar). Se stavovou lištou se blíže seznámíme v kap. 2.7.

V horní části lišty aktivit se nachází pět záložek, které si dále popíšeme:

1. Explorer – Vytvořené soubory a složky.
2. Search – Hledání v rámci aktuálně otevřeného souboru.
3. Source Control – Zálohování prostřednictvím verzovacího nástroje Git.
4. Run and Debug – Debuggování programu (blíže se seznámíme v kap. 3.1.2).
5. Extensions – Umožňuje nainstalovat a spravovat velkou řadu různých rozšíření (využijeme v následující kapitole).

2.3 Instalace pluginů PlatformIO a C/C++

Abychom mohli pro programování mikrokontroléru využít mbed knihovnu, je nutné nainstalovat rozšíření (tzv. plugin) s názvem PlatformIO IDE. S jeho využitím získáme možnost programovat za pomoci mbed tříd, ale bez nutnosti využít online Mbed překladač nebo Mbed IDE.



Obrázek 2.3: Úvodní stránka VS Code s vyznačenými částmi

Instalace pluginu PlatformIO:

1. Přepněte na záložku Extensions.
2. Do vyhledávacího pole zadejte PlatformIO IDE.
3. Nainstalujte PlatformIO IDE (po úspěšném nainstalování by se měla přidat na liště aktivit další záložka).

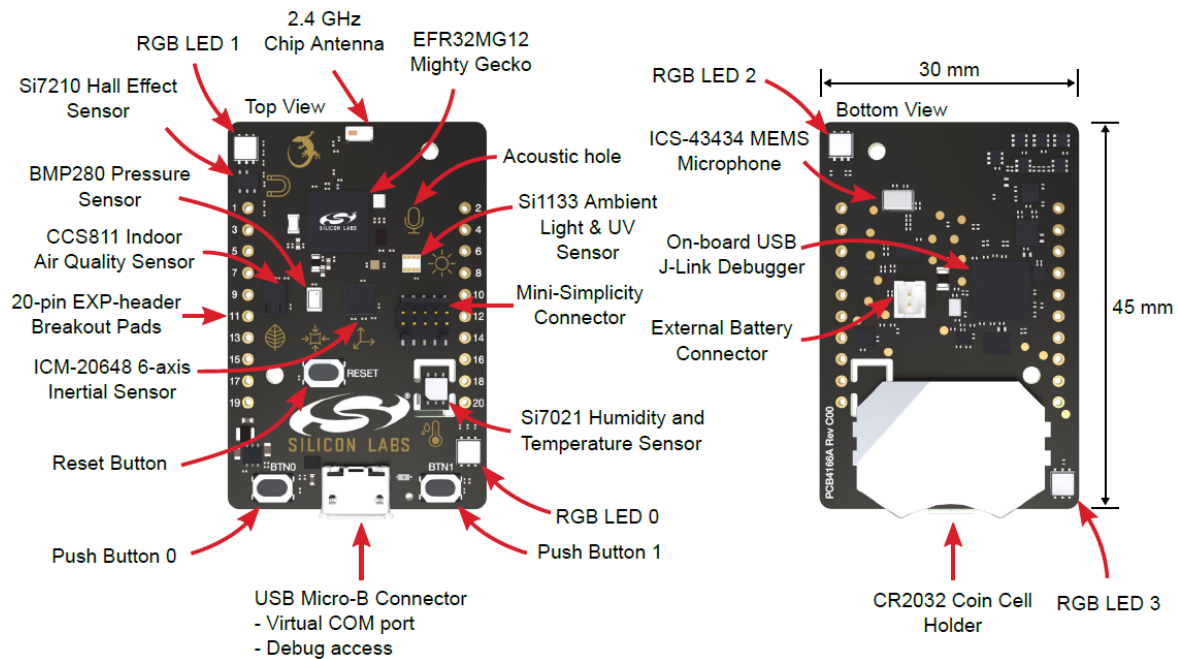
Stejným postupem si nainstalujte i plugin s názvem C/C++ (žádná nová záložka v liště aktivit se neobjeví).

2.4 Popis desky

Před samotným programováním mikrokontroléru bychom se s ním měli alespoň rámcově seznámit. Budeme pracovat s mikrokontrolérem třídy ARM Cortex M4, který je osazen na vývojové desce Thunderboard Sense 2 Sensor-to-Cloud Advanced IoT (dále bude označován jen jako Thunderboard). Na oficiální webové stránce⁴ se nachází nejen přehled základních parametrů, ale také několik videí o senzorech a obecně možnostech desky. Tuto sérii krátkých videí by si měl každý čtenář pustit, jelikož se jedná o základní přehled o možnostech mikrokontroléru (videa se nachází ve spodní části webové stránky).

⁴<https://www.silabs.com/development-tools/thunderboard/thunderboard-sense-two-kit>

Obecně se Thunderboard prezentuje jako kompaktní vývojářská platforma nabitá funkcemi, která nabízí nejrychlejší cestu k vývoji IoT produktů, jako jsou např. bezdrátové low power senzory uzly napájené z baterie. Deska nabízí velké množství senzorů a rádiové rozhraní, umožňující využít standardizované protokoly jako je Zigbee, Thread a Bluetooth low energy, pomocí kterých lze realizovat cloudové připojení. Dále kupříkladu mikrokontrolér disponuje čtyřmi RGB LED, jednou RG LED, resetovacím tlačítkem, dvěma uživatelskými tlačítky a pouzdem na baterii. Hlavní části mikrokontroléru jsou zvýrazněny na obr. 2.4.



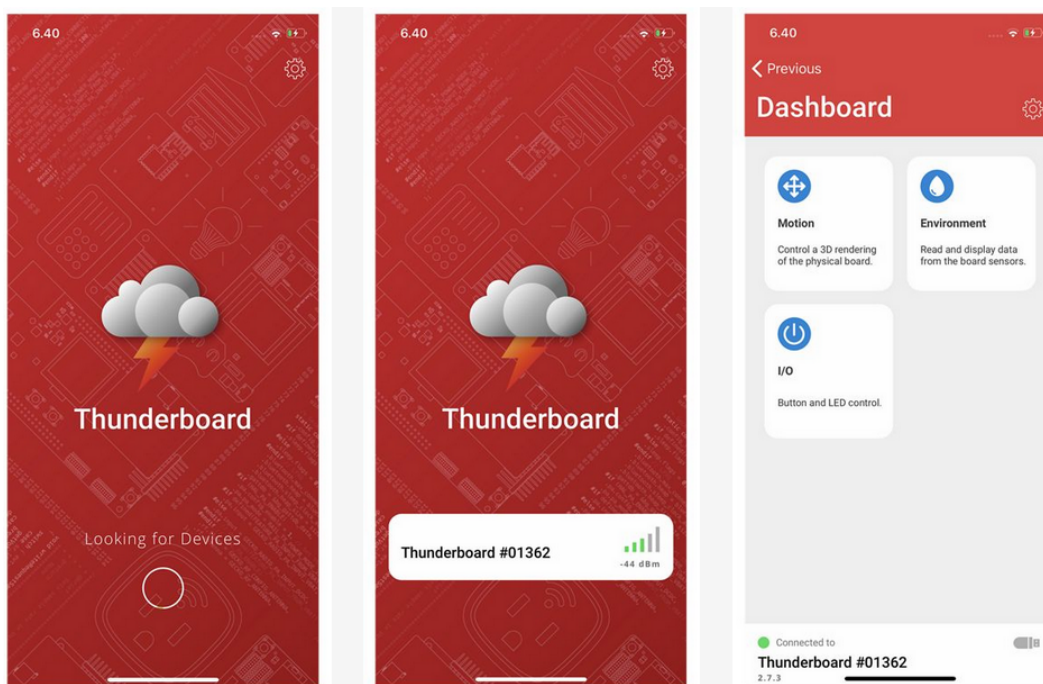
Obrázek 2.4: Popis mikrokontroléru Thunderboard Sense 2 (převzato z [3])

2.5 Aplikace pro vyzkoušení funkčnosti desky

Výrobce desky Thunderboard (Silicon Labs) nabízí svým zákazníkům si zdarma stáhnout mobilní aplikaci s názvem Thunderboard, která je dostupná v Google Play i v Apple Store. Postup, jak si aplikaci vyzkoušet, je popsán v dokumentu [4] v sekci GETTING STARTED. Pokud je Thunderboard nový, demo aplikace je v desce nahraná, pokud byl již Thunderboard přeprogramován, je třeba nahrát demo aplikaci prostřednictvím prostředí Simplicity Studio⁵. Pokud máte v telefonu aplikaci, zapněte si Bluetooth a připojte se k zařízení (viz obr. 2.5) (pozn. BLE advertisement trvá 30s po restartu).

Aplikace umožňuje vyzkoušet funkčnost všech senzorů, diod i tlačítek. V části Motion se využívá šestiosý inerciální senzor a v části Environment jsou hodnoty ze všech ostatních senzorů, zamyslete se nad jejich hodnotami a otestujte je (inspirovat se můžete např. videem zde). Parametr VOCs (z angl. volatile organic compound) neboli

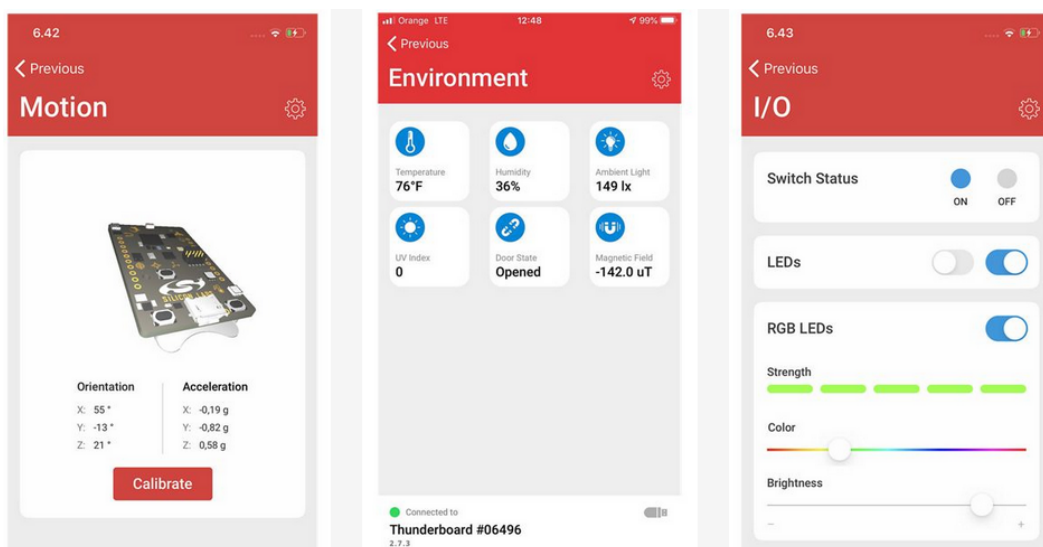
⁵https://cw.fel.cvut.cz/wiki/courses/b0b37nsi/hw/01#demo_aplikace



Obrázek 2.5: Mobilní aplikace – připojování, úvodní vzhled (převzato z [2])

těkavá organická látka měří senzor kvality ovzduší a jedná se o veličinu, která udává počet těchto látek v nějakém množství, proto je uváděna v jednotce ppb (z angl. parts per billion), což je počet částic na miliardu.

V poslední části I/O se pracuje (jak název napovídá) se vstupy a výstupy, konkrétně se dvěma uživatelskými tlačítky, RG LED a čtyřmi RGB LED, u kterých je možné měnit barvu a jas. Všechny zmíněné části vidíme na obr. 2.6.

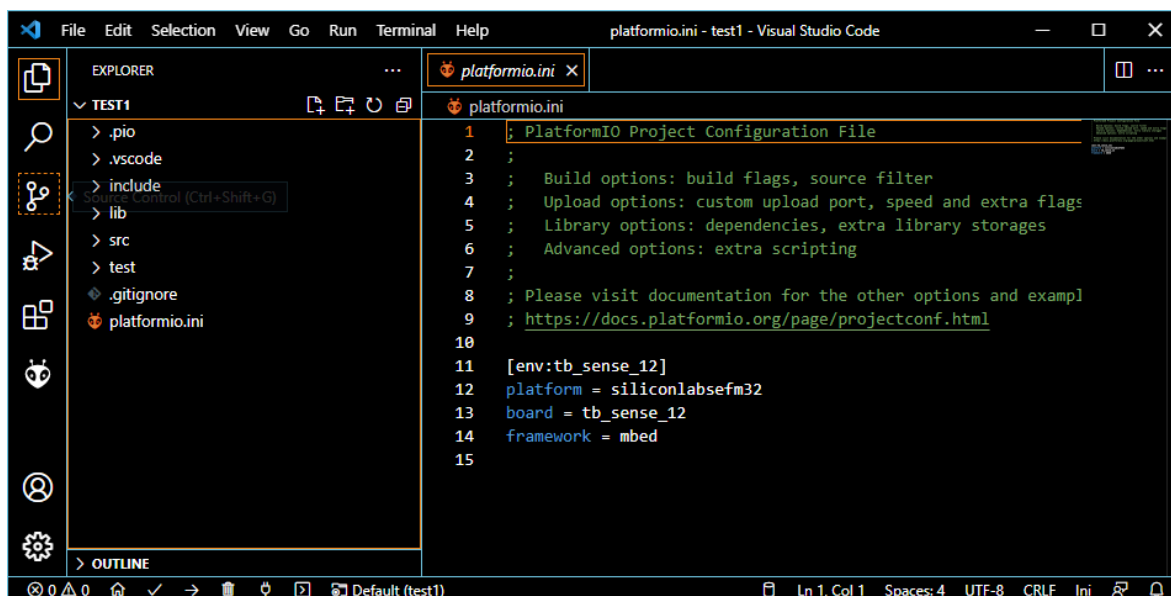


Obrázek 2.6: Mobilní aplikace – jednotlivé části (převzato z [2])

2.6 Založení projektu

První projekt vytvoříme podle následujícího postupu:

1. Na liště aktivit rozklikneme novou záložku PlatformIO.
2. PIO Home -> Open
3. New project
4. Zvolíme jméno projektu (např. test1).
5. Najdeme vhodnou desku, tzn. v našem případě Thunderboard Sense 2 Sensor-to-Cloud Advanced IoT.
6. Framework necháme Mbed.
7. Pole Use default location necháme označené, projekt se tak uloží do složky Dokumenty, kde se vytvoří složka s názvem PlatformIO.
8. Finish (vytváření projektu může chvíli trvat).



Obrázek 2.7: První vytvořený projekt

Po vytvoření programu by se měla otevřít složka platformio.ini (viz obr. 2.7), kterou si dále popíšeme. Každý PlatformIO (dále už jen zkráceně PIO) projekt má tento konfigurační soubor v kořenovém adresáři projektu. V tomto souboru je v základní verzi (té naší) pouze informace o platformě, použité desce a vybraném prostředí (framework). PIO umožňuje pracovat s mnoha prostředími⁶, přičemž mezi nejznámější patří STM32Cube a Arduino.

Podadresáře projektu a jejich význam:

⁶viz <https://docs.platformio.org/en/latest/frameworks/index.html>

- .pio** – nastavení komunikace s deskou a debugování
- .vscode** – nastavení kompilace
- include** – hlavičkové soubory
- lib** – knihovny specifické pro projekt (soukromé)
- src** – zdrojové kódy včetně main.cpp
- test** – umožňuje specifické testování projektů a tzv. PlatformIO Unit Testing⁷

2.7 Stavová lišta

Před programováním prvního programu se blíže seznámíme se stavovou lištou, kterou budeme používat. Jedná se o lištu na spodní části editoru. Význam jednotlivých ikon zleva doprava podle obr. 2.7

errors & warnings – první dvě čísla ukazují počet errorů a warningů.

PlatformIO: Home – otevře PIO úvodní stránku.

PlatformIO: Build – zkompiluje AKTUÁLNĚ otevřený PIO projekt. Jak poznat aktuálně otevřený PIO projekt si vysvětlíme dále.

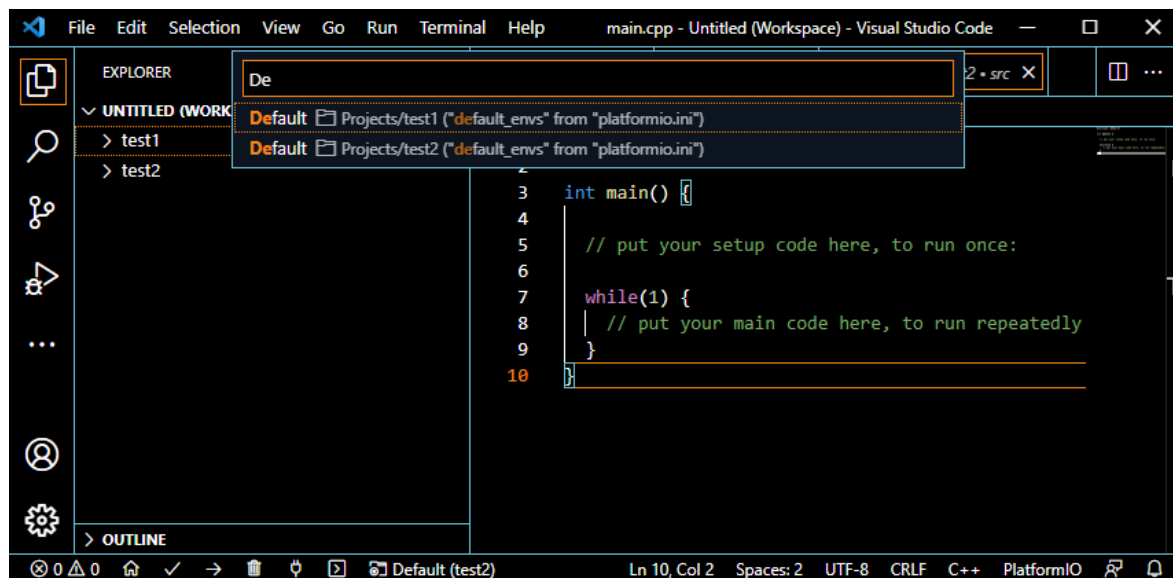
PlatformIO: Upload – nahraje úspěšně zkompilovaný program do připojené desky.

PlatformIO: Clean – první kompilace často trvá zpravidla velmi dlouho, jelikož se kompiluje mnoho souborů. Výsledkem jsou soubory s koncovkami .o a .d. Druhá kompilace trvá už jen zlomek původního času, jelikož změna v kódu se týká pouze malé části souborů, které je potřeba znova zkompilovat. Pokud aktivujeme zmíněnou ikonu, všechny soubory vygenerované v průběhu kompilace se smažou. Jedná se proto o ikonu, na kterou bychom si měli dávat pozor.

PlatformIO: Serial Monitor – slouží pro sériovou komunikaci s deskou, zastává tedy funkci terminálového emulátoru (viz kap. 3.4).

PlatformIO: New Terminal – otevře se nový terminál.

Switch PlatformIO Project Environment – poslední ikona z levé části stavové lišty a je to právě ta ikona, která dává informaci o aktuálně otevřeném PIO projektu. Pokud chceme volně přejít k práci na jiném projektu, nemusíme projekt zavírat a znovu otevírat. Stačí probíranou ikonu otevřít a ta zobrazí příkazovou řádku, ve které vybereme řádek se slovem Default a cestou k požadovanému projektu. Kupříkladu na obr. 2.8 je zachycena situace, ve které je otevřen projekt test2 a chceme dále pracovat na projektu test1, tím pádem otevřeme probíranou ikonu a zvolíme první řádek z příkazové řádky.



Obrázek 2.8: Přepínání mezi projekty

Další ikony zobrazují aktuální polohu kurzoru, počet mezer na odsazení, způsob kódování, sekvenci pro konec řádku, mód pro daný programovací jazyk, nebo rozhraní.

2.8 Kompilace programu a využití hlavičkových souborů

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

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

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

S procesem kompilace souvisí také využití hlavičkových souborů (angl. header files). Při programování větších projektů je vhodné rozdělit kód do hlavičkových (s koncovkou

⁷viz <https://docs.platformio.org/en/latest/plus/unit-testing.html>

.h) a implementačních (s koncovkou .c) souborů. Hlavičkové soubory obsahují pouze deklaraci. Říkají tedy pouze to, že někde existuje určitá funkce (struktura, třída apod.). Tím dávají ostatním souborům informace o tom, co je v jiných implementačních souborech obsaženo.

3 | První programy

3.1 Hello World!

Cílem je rozblikat malou RG LED, kterou již známe z kap. 2.4. Jelikož většina čtenářů pravděpodobně nemá žádné zkušenosti s Mbed OS ani s programováním mikrokontrolérů, ukážeme si rovnou výsledek. Finální kód detailně rozebereme.

```
1 | #include "mbed.h"
2 |
3 | DigitalOut myled(LED1);
4 |
5 | int main() {
6 |     while(1) {
7 |         myled = 1; // LED is ON
8 |         wait(0.2); // 200 ms
9 |         myled = 0; // LED is OFF
10 |        wait(1.0); // 1 sec
11 |    }
12 | }
```

Pro svou činnost a komunikaci s okolím používá mikrokontrolér nejrůznější interní periférie¹, v tomto případě GPIO (General Purpose Input Output). K jednotlivým pinům GPIO se pak připojují periférie externí, jako je třeba tlačítko nebo LED. Před použitím interní periférie je třeba provést konfiguraci, tj. zpravidla připojit danou periférii ke sběrnici zajišťující komunikaci s jádrem kontroléru a distribuci hodinového signálu. Konfigurace je v Mbed OS zajištěna v konstruktorech tříd, které jsou definovány v knihovně `mbed.h`.

Konfigurace GPIO pinu kontroléru, ke kterému je připojena LED, probíhá na druhém řádku kódu:

```
2 | DigitalOut myled(LED1); // R:LEDO = PD8, G:LED1 = PD9
```

Tento řádek vytvoří objekt `myled` jako instanci třídy `DigitalOut`, který nakonfiguruje GPIO pin, ke kterému je připojena zelená LED, v režimu digitální výstup. `LED1` (zelená) a `LEDO` (červená) jsou symbolickým označením pinu, na který je dioda připo-

¹Periferiím jsou věnovány následující kapitoly.

jena. Ze zápisu je patrné, že k vytvoření objektu bývají (ne vždy tomu tak je) potřeba vstupní parametry. Vstupní parametry většinou určují, jaký pin se bude používat, případně nastavují/specifikují další parametry.

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

Při programování někdy bývá potřeba vložit prodlevu (čekat určitou dobu). Pro tento účel lze využít funkci `thread_sleep_for()` s argumentem funkce v milisekundách. Pro vložení kratších prodlev než je milisekunda lze použít funkce `wait_us()` a `wait_us()`, avšak tyto funkce negativně ovlivňují spotřebu a také mají vliv na výkon vícevláknové aplikace.

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

Vyzkoušejte si

Modifikujte program a zkuste rozblikat i červenou LED. Dále vyzkoušejte blikat oběma barvami zároveň, případně blikajte v polovině intervalu jednou barvou a v druhé polovině druhou barvou. Také si zkoušejte měnit intervaly čekání. Představivosti se meze nekladou.

3.1.1 Užitečné funkce VS Code

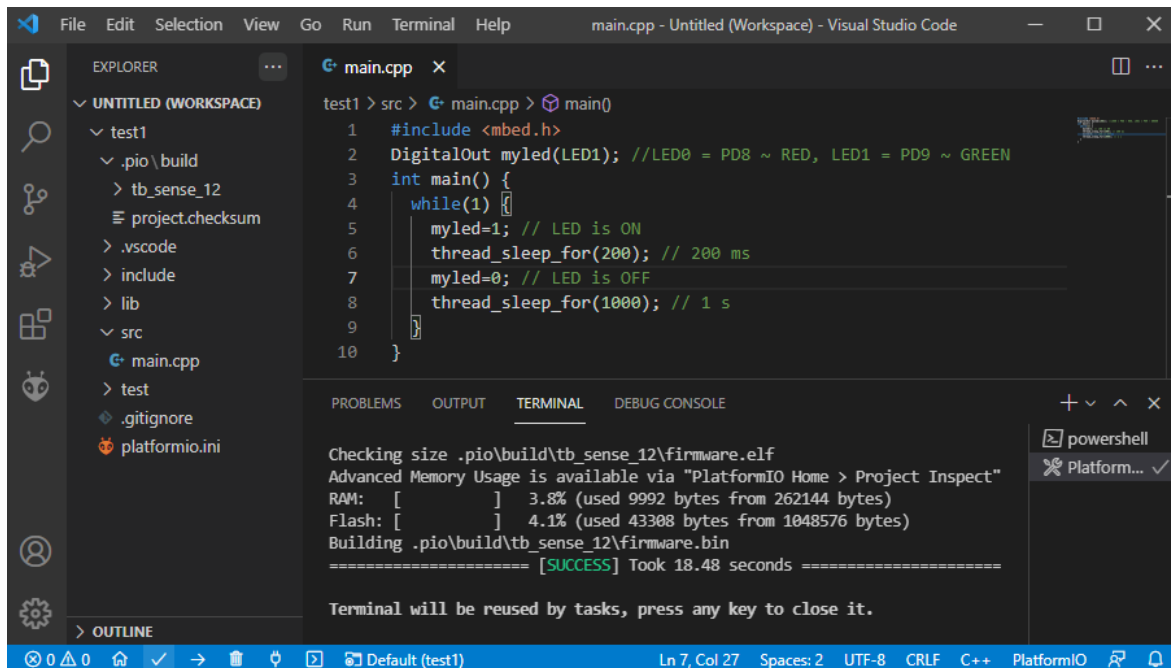
Při příležitosti vytvoření prvního programu by bylo dobré zmínit další užitečné funkce VS Code, které představují rychlou pomoc programátorovi. Pokud klikneme pravým tlačítkem na jakoukoli část kódu (např. na výše probíranou funkci `thread_sleep_for()`), pak máme možnost vybrat **Go to Declaration**. Po zvolení této možnosti se otevře hlavičkový soubor s deklarací dané funkce a ihned kupříkladu vidíme, že argument funkce se udává v milisekundách.

Teď, když už víme, jak program funguje, nezbyvá než ho zkompileovat tlačítkem PlatformIO: Build (viz kap. 2.7²). Mohlo by se zdát, že existuje i druhý způsob a to Run -> Run Without Debugging. Tento způsob však provádí stejnou akci jako bychom vybrali Run > Start Debugging, tedy spouští debuggování programu, což je náplní až následující kapitoly.

Po té, co se program zkompileje (první kompilace projektu trvá opravdu dlouho, běžně i v řádu minut), by se mělo v terminálu objevit velkými písmeny slovo SUCCESS, které samozřejmě signalizuje úspěšnou kompilaci (viz obr. 3.1). V opačném případě se objeví FAILED. Vypsání problémů se objeví části PROBLEMS.

Následuje nahrání programu do samotného mikrokontroléru. K tomu se využije tlačítko Platformio: Upload (viz kap. 2.7). Po nahrání programu se port mikrokontroléru

²FIXME

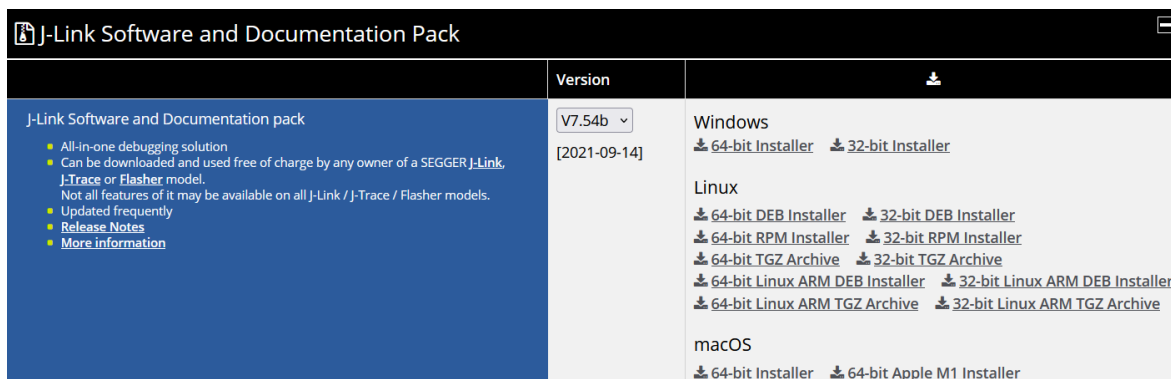


Obrázek 3.1: První program – blikání LED

odpojí a zase připojí.

3.1.2 Debugování programu ve VS Code

V minulé kapitole jsme vytvořili první program, který jsme následně zkompilevali a nahráli do desky. Jedná se o přímočarý přístup, který ovšem má jednu nevýhodu. Tou je nedokonalá orientace v právě běžícím programu. Ano, v takto jednoduchém programu to pravděpodobně není problém, ale u delších programů již jistě problém může nastat. Řešením je krokování programu neboli debugging, což je náplní této kapitoly.



Obrázek 3.2: Ovladače pro J-Link.

Ještě před samotným debugováním programu se musíme ujistit, zda máme nainstalované ovladače pro J-Link. J-Link je ta část mikrokontroléru, která obstarává debugování. Jak poznat jestli mám nainstalované právě tyto ovladače? Ve Správci zařízení by se, konkrétně v sekci Další zařízení, objevila (po připojení desky) nová ikona s

názvem Neznáme zařízení. V takovém případě je potřeba ovladače stáhnout ze zdroje³ (viz obr. 3.2) a doinstalovat.

Nyní již k samotnému debuggování. Místo použití postupu Platformio: Build a následně Platformio: Debug se nyní zvolí Run -> Start Debugging (nebo jen zkráceně F5). Nastavení desky do debug módu trvá také relativně dlouho (samozřejmě v závislosti na vašem zařízení), ale opět jenom napoprvé. Výsledné okno vidíme na obr. 3.3.

Obecně lze v debug módu zjistit mnoho informací o aktuálním stavu programu, ale jelikož účelem této práce se není naučit programovat mikrokontrolér na úrovni registrů, zaměříme se pouze na některé části. Mezi tyto části patří pohybování se v programu, k čemuž slouží nově zobrazená lišta s šesti ikonami (viz červená šipka na 3.3). Význam jednotlivých ikon zleva doprava:

Continue / Pause Slouží k pokračování/zastavení programu. Vybráním Continue program pokračuje k dalšímu breakpointu. Breakpoint můžeme přidat na řádek, na kterém chceme program zastavit. Breakpoint je vyznačen červeným kolečkem vlevo od daného řádku (viz část zvýrazněná modrou šipkou na obr. 3.3) Žlutý symbol na zmíněném obrázku reprezentuje řádek, na kterém je aktuálně program zastaven. Pokud se na žádném řádku breakpoint nenachází, program se chová jako bychom ho zkompilevali a nahráli do desky. Tradičně (v jiných vývojových prostředích) se program zastavuje na řádku, na kterém je breakpoint. Ve VS Code to však (z nejasných důvodů) neplatí vždy a závisí na konkrétní pozici breakpointu. Konkrétně v příkladu na zmíněném obrázku se program zastavuje za danými breakpointy.

Step Over Akce, která vyvolá vykonání aktuálního řádku. Pokud se na tomto řádku nachází funkce, pak bude vykonána bez pozastavení se v jejím průběhu.

Step Into Jedná se o podobnou akci jako Step Over s tím rozdílem, že pokud se na řádku nachází funkce, program se přemístí do dané funkce na první řádek její definice. Tímto způsobem se tedy pohybujeme po každém řádku programu, který procesor skutečně vykoná.

Step Out Způsobí vrácení se na řádek, kde byla aktuální funkce zavolána. Tato ikona není pro naše účely příliš důležitá.

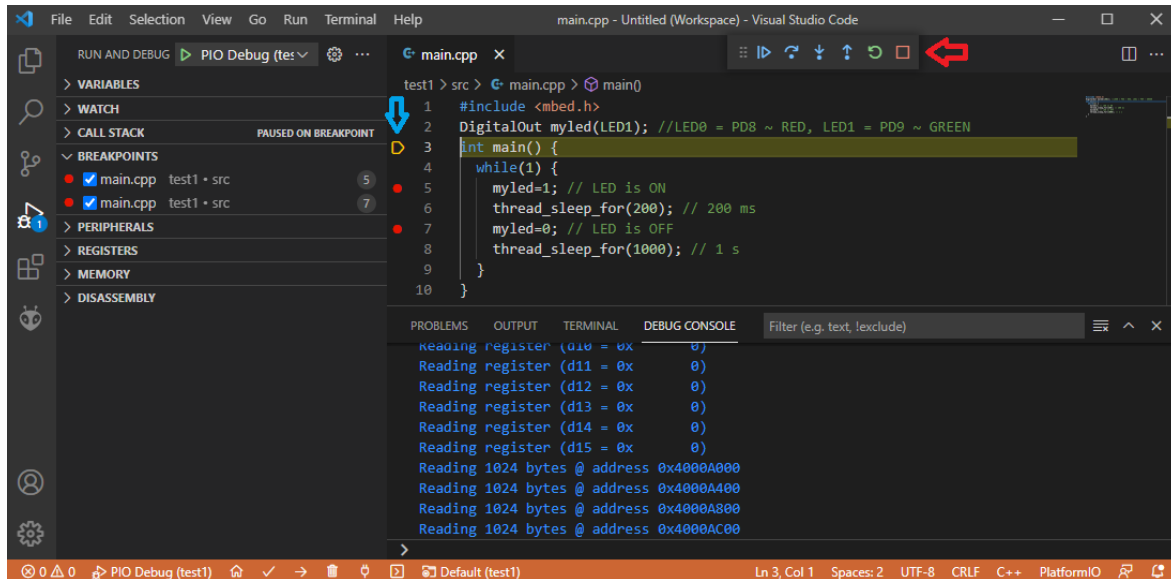
Restart Vyvolá restartování programu.

Stop Ukončí debuggování.

Vpravo od lišty aktivit se nachází sloupec s mnoha záložkami jako je např. watch, breakpoints nebo peripherals (viz obr. 3.3). Pro naše účely se zmíníme pouze dvou z nich. Záložka breakpoints označuje místa vytvořených breakpointů a v záložce watch lze vyhodnocovat a sledovat proměnné a výrazy. Pokud bychom v programu měli nějakou proměnnou, můžeme s využitím této funkce sledovat její aktuální hodnotu.

Debuggování je dobrý nástroj jak získat reálný vhled do fungování mikrokontroléru a také jej lze využít jako efektivní pomůcku při hledání chyby. Pokud se s debuggováním

³<https://www.segger.com/downloads/jlink/>



Obrázek 3.3: Debuggování prvního projektu

setkáváte poprvé, důkladně se s ním seznámete, protože nikdy nevíte, kdy ho budete potřebovat. U takto jednoduchých programů se to může zdát zbytečné, ale na seznámení je to ideální.

3.2 Uživatelská tlačítka

Další externí periférii, kterou najdeme na Thunderboardu, jsou dvě tlačítka. Jedno z nich je resetovací (tj. lze s nimi resetovat běžící program) a druhé lze využít k uživatelskému vstupu. Při správném zapojení tlačítka je na GPIO pinu napěťová úroveň odpovídající log. 0 nebo log. 1. Digitální hodnotu lze získat pomocí instance třídy `DigitalIn`, jejímž argumentem je označení GPIO pinu

```
1 | DigitalIn mybutton(PIN);
```

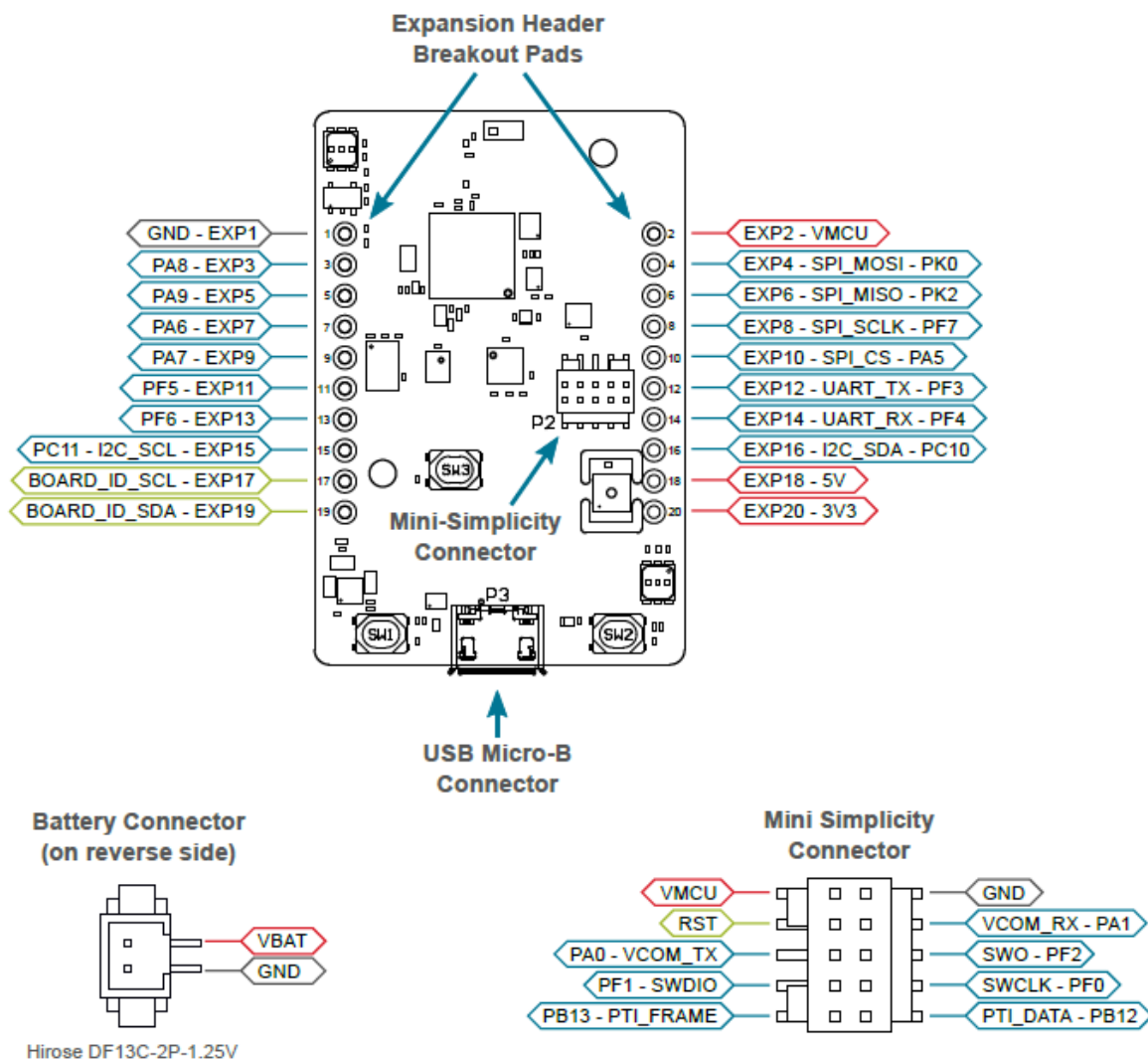
Místo symbolického názvu `PIN` doplňte `BUTTON1` nebo `BUTTON2`. Za těmito názvy se opět skrývá označení pinu, na kterém je tlačítko připojeno. U diody se do objektu zapisovalo, u tlačítka se naopak z objektu čte.

Vyzkoušejte si

Vytvořte nový projekt a napište program, který bude rozsvěcet a zhasínat LED podle toho, zda je nebo není stisknuto tlačítko. Pokuste se vysvětlit chování programu. Před kompilací se ujistěte, že máte vybraný správný projekt pro kompilaci (viz ikona Switch PlatformIO Project Environment probíraná v kap. 2.7).

3.3 Vstupně výstupní piny

Pin obecně označuje vývod elektronické součástky. Na obr. 3.4 vidíme pojmenování jednotlivých pinů na Thunderboardu, přičemž vlevo dole je ještě označení pinů na držáku baterie a vpravo dole označení tzv. mini simplicity konektoru.



Obrázek 3.4: Pinout diagram

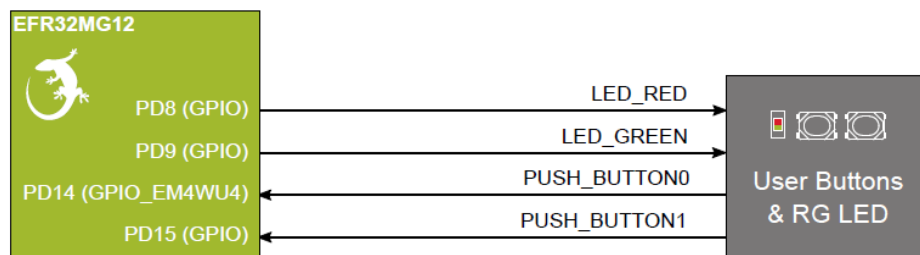
Doposud jsme se setkali s využitím pinů jako digitálního vstupu a digitálního výstupu. Na desce se také nachází piny, které nelze využít pro práci s perifériemi. Jedná se o zem (značená jako GND), zdroj napětí 3,3V a 5V (na desce značeny 3V3 a 5V) a VMCU (main board power rail). Piny `BOARD_ID_SCL` a `BOARD_ID_SDA` slouží pro identifikace přídatných desek, proto je také nemůžeme použít. Obecně je směřování pinů na desce velice flexibilní, tedy většina periférií může být připojena téměř k jakémukoli pinu.

Uživatelská tlačítka ani RG LED na obr. 3.4 není; jak tedy poznat, ke kterému pinu jsou připojeny? Tato informace lze najít v uživatelské příručce [7], případně v [9]

a [10]. Červená (R) LED je připojena na PD8, zelená (G) LED na PD9 a uživatelská tlačítka jsou připojena na piny PD14 a PD15, viz obr. 3.5.

Zapamatujte si

Písmeno P je zkratka slova pin a písmeno za ním specifikuje GPIO bránu. Místo správného označení pinu je možno použít i symbolické konstanty LED0, LED1, BUTTON1 a BUTTON2. Zkuste si v již vytvořených programech tyto konstanty nahradit.



Obrázek 3.5: Pinout RG LED a uživatelských tlačítek

Doplnění k třídě DigitalOut

Tato kapitola přinese jen malé doplnění k již zmíněné třídě `DigitalOut`, která umožňuje nastavit stav výstupu druhým argumentem konstruktoru.

```
1 | DigitalOut myled(LED0, 1);
```

Napište program, ve kterém stlačením jednoho vybraného uživatelského tlačítka změníte stav R LED, přičemž v rámci procvičení výše zmíněné problematiky využijte pro nastavení počátečního ON stavu diody druhý argument třídy. Program otestujte a ověřte, že skutečně dělá to, co od něj očekáváte.

```
1 | #include <mbed.h>
2 |
3 | DigitalOut led_r(LED0, 1); // LED0 = PD8 ~ RED
4 | DigitalIn btn_1(BUTTON1); // BUTTON1 = PD14
5 |
6 | int main() {
7 |     while (1) {
8 |         if (btn_1 == 0){
9 |             led_r = !led_r;
10 |             // je tlacitko stale stisknute?
11 |             while (!btn_1) {}
12 |         }
13 |     }
```

14 | }

Nekonečná smyčka v programu řeší situaci, kdy je tlačítko stisknuté, tedy ještě není uvolněno. Stav LED se tak skutečně změní pouze na náběžnou hranu, tzn. při stisku tlačítka. Nevýhoda tohoto přístupu je v setrvání programu v nekonečné smyčce, dokud se tlačítko neuvolní. Tato skutečnost v našem případě nevádí, ale pokud bychom program o něco málo rozšířili, pak už by mohl vzniknout problém a museli bychom využít jiný způsob řešení.

3.4 Sériová komunikace

Naše zatím velmi jednoduché programu umožňují signalizovat uživateli pomocí RG LED. Tato informace je v mnoha případech nedostačující a rádi bychom dostávali oznámení v textové podobě. Přesně k tomuto účelu slouží třídy `UnbufferedSerial` a `BufferedSerial`, pomocí kterých lze posílat znaky mezi deskou a PC prostřednictvím sériové komunikace v asynchronním režimu (UART). S třídou `BufferedSerial` se seznámíme v této kapitole, s třídou `UnbufferedSerial` o něco později, až si povíme něco o přerušení.

3.4.1 Třída `BufferedSerial`

Třída `BufferedSerial` slouží k sériovému přenosu dat s využitím periferie `UART`, která pro posílání a příjem dat používá dva samostatné piny označované jako `TX` (transmit) a `RX` (receive). Argumenty třídy jsou tři: `TX`, `RX` a rychlost přenosu – baudrate⁴. Jaké piny zvolit na funkce `TX` a `RX`? Mbed umožňuje využít jen dvojici `USBTX` a `USBRX`, za kterými se skrývají piny `PA0` a `PA1`. Co se týče baudrate, tak ta v našem případě musí být nastavena na 115200, stávající firmware desky ji neumožňuje měnit. Více o sériové komunikaci se dozvíme v kapitole 7.1.

```
1 | BufferedSerial serial_port(USBTX, USBRX, 115200);
```

Třída `BufferedSerial` nabízí řadu funkcí⁵, avšak pro naše účely prozatím postačí funkce `read()` a `write()`. Pro používání těchto funkcí (hlavně funkce `read()`) je nutné mít vytvořený nějaký buffer, do kterého se budou příchozí data ukládat. K tomu může sloužit v nejjednodušším případě pole znaků (např. o osmi prvcích). Vhodná velikost tohoto pole závisí na konkrétní aplikaci.

⁴baud je starší jednotka rychlosti přenosu a odpovídá přibližně bit/s

⁵<https://os.mbed.com/docs/mbed-os/v6.14/apis/serial-uart-apis.html>

Co je potřeba pro komunikaci s deskou?

Pro komunikaci po sériovém portu lze využít řadu různých programů, například Putty nebo Terra Term. Plugin PlatformIO, který používáme při programování, disponuje funkcí PlatformIO: Serial Monitor – ikonu najdete na stavové liště.

3.4.2 Příklad – ovládání LED příkazy přes UART

Napište program, ve kterém posláni znaku z klávesnice změní stav červené LED, přijatý textový řetězec zároveň pošlete zpátky do PC. To způsobí zobrazení napsaného znaku na klávesnici v sériovém monitoru (jinak tam tento znak nevidíte).

Možné řešení:

```

1  #include <mbed.h>
2
3  DigitalOut    led_r(LED0, 1); // LED0 = PD8 ~ RED
4  BufferedSerial serial(USBTX, USBRX, 115200);
5
6  int main() {
7      char buffer[8] = {0}; // buffer pro přijata data
8      uint32_t num;        // pocet prijatych bytu
9
10     while (1) {
11         if (num = serial.read(buffer, sizeof(buffer))) {
12             // inverze stavu LED
13             led_r = !led_r;
14             // tisk prijatych dat zpet do terminalu - echo
15             serial.write(buffer, num);
16         }
17     }
18 }
```

Mbed OS implementuje funkce standardní knihovny (`getchar`, `putchar()`, `printf()`, `scaf()`), které automaticky vytváří sériové spojení s využitím virtuálního COM portu. Nabízí se tedy i jednodušší řešení, ovšem je třeba si uvědomit, že program běží v nekonečné smyčce a složitější komunikace bude vyžadovat náročnější ošetření.

```

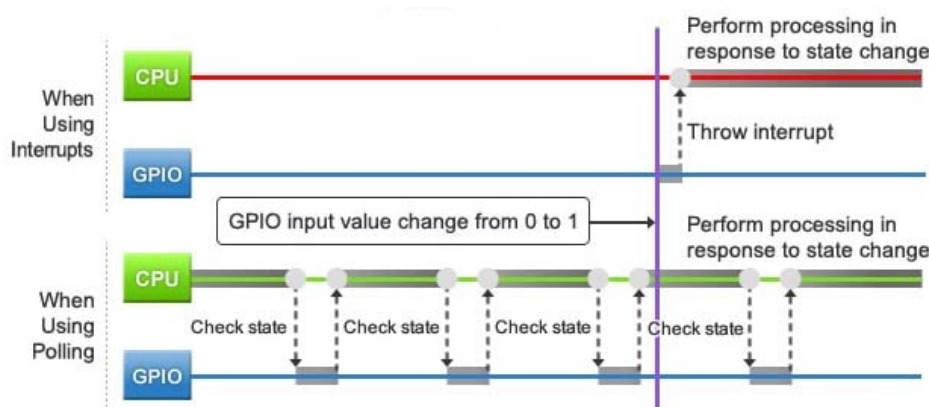
1  #include <mbed.h>
2
3  DigitalOut led_r(LED0, 1); // LED0 = PD8 ~ RED
4
5  int main() {
6      char buffer;
7
8      while (1) {
```

```
9     buffer = getchar();
10    putchar(buffer);
11    led_r = !led_r;
12    }
13 }
```

4 | Přerušeni

V této kapitole se seznámíme s technikou zvanou přerušeni. Přerušeni (angl. interrupt) představuje velmi mocný nástroj jak reagovat na události, které potřebují obsloužit v nepředvídatelných časech. Pokud se přerušeni aktivuje, řadič přerušeni vyvolá signál, na jehož základě procesor zastaví aktuálně prováděný program, zkontroluje, která periferie přerušeni vyvolala a na základě toho vykoná požadované akce, tzv. obsluhu přerušeni. Obsluhou přerušeni je funkce, nazývaná **ISR** (Interrupt Service Routine).

Přerušeni je zpravidla detekováno na hardwarové úrovni¹, procesor je využit pouze pro obsluhu. V příkladu vytvořeném v kap. 3.3 jsme v každém cyklu nekonečné smyčky zjišťovali stav uživatelských tlačítek. Tento způsob obsluhy se nazývá pooling a jeho grafická podoba a porovnání s přerušeni je znázorněna na obr. ??.



Zatímco systematická kontrola stavu periferií je časově náročná, obsluha přerušeni proběhne pouze v případě, že k přerušeni skutečně dojde. Procesor se tak o vstupy nemusí starat, takže neztrácí strojový čas potřebný pro jiné výpočty. V kombinaci s některým z úsporných režimů lze také významně snížit spotřebu procesoru. Obecně je možné přerušeni parametrizovat, např. zvolit jejich prioritu, přerušeni seskupovat, atd. Při použití Mbed je možné pouze zvolit, zda se obsluha přerušeni pomocí instance třídy `InterruptIn` vyvolá náběžnou nebo sestupnou hranou přerušovacího signálu.

¹V tomto textu uvažujeme asynchronní vnější přerušeni, vyvolaná periferiemi. Kromě toho je možné obsluhovat vnitřní přerušeni, signalizující problémy při zpracování strojových instrukcí (dělení nulou, porušení ochrany paměti), nebo synchronní softwarové přerušeni, vyvolané speciální instrukcí.

4.1 Funkce třídy `InterruptIn`

Veškeré informace o použití třídy `InterruptIn` jsou uvedeny v dokumentaci². Výpis a krátké shrnutí jednotlivých funkcí:

`mode()` Nastavení módu pinu.

`read()` Čtení aktuální log. úrovně na daném pinu.

`rise()` Obsluha přerušení vyvolána náběžnou hranou přerušovacího signálu.

`fall()` Obsluha přerušení vyvolána sestupnou hranou přerušovacího signálu.

`disable_irq()` Zákaz obsluhy přerušení

`enable_irq()` Povolení obsluhy přerušení.

4.1.1 Příklad – obsluha tlačítek s využitím přerušení

Pro obsluhu vnějšího přerušení vyvolaného tlačítkem připojeným k pinu GPIO brány je třeba nejprve vytvořit instanci třídy `InterruptIn`. Argumentem konstruktoru je označení pinu, v tomto případě symbolický název `BUTTON1`.

```
1 | InterruptIn btn_1(BUTTON1); // BUTTON1 = PD14
```

Dále je třeba definovat, jakým způsobem bude přerušení obslouženo. Na název obslužné rutiny nejsou kladeny žádné požadavky, je ovšem vhodné, aby bylo z názvu patrné, k jakému účelu funkce slouží. Funkce je vždy bez návratové hodnoty a nepracovává žádné argumenty.

```
1 | void btn1_isr() {  
2 |     // kód pro obsluhu preruseni  
3 | }
```

Posledním z kroků, nutných k použití přerušení³, je registrace obslužné rutiny, tj. propojení funkce pro obsluhu a signálu, který přerušení vyvolal. K tomu slouží funkce `rise()` nebo `fall()`, jejímž argumentem je ukazatel na ISR.

```
1 | btn_1.rise(btn1_isr);
```

Nyní již máme všechny indicie k vytvoření programu, který využívá přerušení. Cílem je vytvořit jednoduchý program pro rozsvícení/zhasnutí R a G LED po stisku některého z uživatelských tlačítek. Každá z diod je řízena jiným tlačítkem. Pozn.: tlačítka reagují naopak, než bychom možná očekávali, tedy při použití funkce `rise()` se

²<https://os.mbed.com/docs/mbed-os/v6.15/apis/interruptin.html>

³Bez použití Mbed OS bude pravděpodobně nutné ještě přerušení povolit.

vyvolá přerušení na sestupnou hranu a při použití funkce `fall()` na hranu náběžnou, vyzkoušejte si. Důvodem je zapojení tlačítka⁴, kvůli kterému, jak víme již z kap. 3.3, hodnota 1 reprezentuje nestlačené tlačítko, nikoli stlačené.

```

1  #include <mbed.h>
2
3  DigitalOut led_r(LED0);    // LED0 = PD8 ~ RED
4  DigitalOut led_g(LED1);    // LED1 = PD9 ~ GREEN
5  InterruptIn btn_1(BUTTON1); // BUTTON1 = PD14
6  InterruptIn btn_2(BUTTON2); // BUTTON2 = PD15
7
8  void btn1_isr() {
9      led_r = !led_r;
10 }
11
12 void btn2_isr() {
13     led_g = !led_g;
14 }
15
16 int main() {
17     btn_1.fall(btn1_isr);
18     btn_2.fall(btn2_isr);
19     while (1) {}
20 }

```

Je možné volat různou obsluhu přerušení na náběžnou a spádovou hranu na stejném pinu, případně lze volat jedno ISR jako reakci na obě hrany. S využitím druhého způsobu můžeme získat stejné chování programu jako v kap. 3.3 (ale s využitím přerušení), vyzkoušejte si.

Zapamatujte si

Obecně obsluha přerušení nesmí obsahovat nekonečné cykly, časově náročné operace či dynamickou alokaci paměti. Jedná se totiž o časově náročné operace a to jde proti tomu, že očekávaná doba strávená v obsluze přerušení je v řádu mikrosekund. Při použití globálních proměnných je vhodné jejich inicializace s klíčovým slovem `volatile`.

4.2 Třída `UnbufferedSerial`

V kap. 3.7.1 jsme se seznámili s třídou `BufferedSerial`. Mbed disponuje také třídou `UnbufferedSerial`, která je z velké části stejná jako předchozí třída, ale umožňuje větší

⁴FIXME

kontrolu nad tokem dat⁵. Navíc třída `UnbufferedSerial` umožňuje používat přerušování, což si právě v této části vyzkoušíme.

Jak název třídy napovídá, při používání sériové komunikace není potřeba vytvářet pomocný buffer. Je to kvůli tomu, že přijetí dat si uživatel musí obsloužit sám programově a to po jednotlivých znacích. Objekt třídy se vytvoří obdobně jako u třídy `BufferedSerial`:

```
1 | UnbufferedSerial serial_port(USBTX, USBRX, 115200);
```

Obsluha přerušování (v tomto případě `serial_isr()`) se k sériovému přerušování přiřadí za pomoci metody `attach()`:

```
1 | serial_port.attach(serial_isr, SerialBase::RxIrq);
```

Vytvořte program, který bude vykonávat identickou funkci jako příklad v kap. 3.4.2. Pro zopakování: přijme znak z PC, pošle ho zpátky a změní stav červené LED. V programu samozřejmě použijte třídu `UnbufferedSerial` a využijte přerušování.

Možné řešení (část kódu je převzata z řešeného příkladu v [16]):

```
1 | #include <mbed.h>
2 |
3 | UnbufferedSerial serial_port(USBTX, USBRX, 115200);
4 | DigitalOut      led_R(LED0); // LED0 ~ PD8 (RED)
5 |
6 | void serial_isr() {
7 |     char c;
8 |     // Read the data to clear the receive interrupt
9 |     if (serial_port.read (&c, 1)) {
10 |         // Echo the input back to the terminal
11 |         serial_port.write (&c, 1);
12 |         R_LED = !R_LED;
13 |     }
14 | }
15 |
16 | int main() {
17 |     serial_port.attach(serial_isr, SerialBase::RxIrq);
18 |     while (1) {}
19 | }
```

Třída `UnbufferedSerial` tedy umožňuje využít výhod přerušování u sériové komunikace. Tímto způsobem jsme tak schopni ušetřit čas v hlavní smyčce a tím tak celý program zefektivnit.

⁵viz <https://os.mbed.com/docs/mbed-os/v6.15/apis/unbufferedserial.html>

5 | Čítače

Čítač (angl. označovaný jako Timer) je vnitřní periferie procesoru, která plní funkci počítáče (angl. counter) hodinových impulsů. Vzhledem k tomu, že frekvence hodinového signálu je známa, může čítač fungovat i jako časovač (angl. timer). Čítače mají široké možnosti, avšak pro účely tohoto textu se spokojíme s představou zařízení, které po uplynutí určitého předem definovaného času vyvolá přerušení. Toto přerušení pak může být vyvoláno jednorázově, nebo se může pravidelně opakovat až do okamžiku, kdy je čítač zastaven.

Mikrokontroléry ARM typicky obsahují řadu čítačů, určených k různým účelům. Mbed OS pro ovládání čítačů vytváří abstraktní popis, který sice neumožňuje vybrat konkrétní čítač, ale zato nabízí unifikované rozhraní orientované na účel měření času.

5.1 Třída Timeout

Nejjednodušší použití čítače umožňuje třída `Timeout`, která vykonává jednorázové zpoždění, jenž je ovšem velmi přesné a je zajištěno hardwarově. Přehled a krátké vysvětlení jednotlivých funkcí, více informací najdete v dokumentaci¹:

`attach()` Určení adresy obsluhy přerušení (první argument) a doby v sekundách (druhý argument), za kterou se vyvolá.

`attach_us()` To stejné s tím rozdílem, že se doba zadává v mikrosekundách.

`disable()` Zakázání přerušení.

V následujícím příkladu bude zelená LED blikat s periodou 500 ms a po 2 s od startu programu se rozsvítí červená LED.

```
1 | #include <mbed.h>
2 |
3 | DigitalOut led_r(LED0); // LED0 = PD8 ~ RED
4 | DigitalOut led_g(LED1); // LED1 = PD9 ~ GREEN
5 | Timeout flipper;
6 |
```

¹<https://os.mbed.com/docs/mbed-os/v6.15/apis/timeout.html>

```
7 void flip_r() {
8     led_r = !led_r;
9 }
10
11 int main() {
12     flipper.attach(flip_R, 2);
13     while (1) {
14         // zelena LED blika s periodou 500 ms
15         led_g = !led_g;
16         thread_sleep_for(500);
17     }
18 }
```

Co přesně program dělá? Po zavolání funkce `attach()` se spustí čítač, který má na vstupu hodinový signál o nějaké frekvenci (není třeba ji znát, třída popisující čítač zajistí potřebné výpočty). Čítač čítá náběžné hrany hodinového signálu, doba 2 s je přepočítána na určitý počet těchto hran. Jakmile se hodnota čítače (počet náběžných hran) dostane na nastavenou hodnotu, dojde k vyvolání přerušení a přerušení vyvolá obsluhu přerušení.

5.2 Třída Ticker

Třída `Timeout` má poměrně limitované možnosti uplatnění. Daleko více by se hodila třída, která by fungovala podobně s tím rozdílem, že by vyvolávala přerušení periodicky po dané době. Přesně k tomuto účelu slouží třída `Ticker`. Opět se pro její funkci využívá čítač, který znovu čítá do předem definované hodnoty (resp. doby). Po této době se však čítač nezastaví, ale způsobí přerušení a spustí se znovu. To má za následek periodické vyvolávání přerušení. Metody jsou stejné jako u třídy `Timeout`, více informací naleznete v dokumentaci².

V následujícím příkladu budou blikat obě dvě LED s různými periodami blikání.

```
1 #include <mbed.h>
2
3 #define R_LED_TOGGLE_TIME 1.0 // in seconds
4 #define G_LED_TOGGLE_TIME 2.0 // in seconds
5
6 DigitalOut R_LED (LED0); // LED0 = PD8 ~ RED
7 DigitalOut G_LED (LED1); // LED1 = PD9 ~ GREEN
8
9 Ticker flipper1;
10 Ticker flipper2;
11
12 void flipper1_isr() {
13     R_LED = !R_LED;
```

²<https://os.mbed.com/docs/mbed-os/v6.15/apis/ticker.html>

```

14 }
15
16 void flipper2_isr() {
17     G_LED = !G_LED;
18 }
19
20 int main() {
21     flipper1.attach (flipper1_isr, R_LED_TOGGLE_TIME);
22     flipper2.attach (flipper2_isr, G_LED_TOGGLE_TIME);
23     while(1);
24 }

```

5.3 Třída Timer

V řadě aplikací je potřeba přesné měření krátkých časových úseků. K tomuto účelu Mbed OS disponuje třídou `Timer`. Rozhraní třídy tvoří metody `start()`, `stop()` a `reset()`, jejichž smysl je z názvů zřejmý. Více informací lze najít v dokumentaci³.

Čítač se spustí metodou `start()` a počítá náběžné hrany hodinové signálu dokud není zavolána metoda `stop()`. Čas uplynulý mezi voláním metod, získaný jako hodnota čítače přepočítaná na čas, lze získat funkcí `read_ms()` nebo `read_us()`.

Následující program změří dobu potřebnou k vykonání funkce `printf()`.

```

1  #include <mbed.h>
2
3  Timer tim;
4
5  int main() {
6      int start_time, end_time;
7      tim.start();
8      start_time = tim.read_us();
9      printf("Hello World!\n");
10     end_time = tim.read_us();
11     printf("Cas: %d us.\n", end_time - start_time);
12 }

```

Výsledný čas vychází cca 1,5 ms, což opravdu není málo. Funkce `printf()` představuje časově náročnou operaci, takže ji není dobré používat v obsluze přerušení.

Mimochodem, bude program fungovat, když nemá nekonečnou smyčku? Po vykonání druhé funkce `printf()` se procesor zastaví nebo co se stane? Jak může „nic nedělat“? Procesor se samozřejmě nezastaví. Ve výsledném kódu kompilátor vytvoří nekonečná smyčka, ve které se procesor zacyklí.

³<https://os.mbed.com/docs/mbed-os/v6.15/apis/timer.html>

5.3.1 Příklad – měření reakční doby

Cílem tohoto příkladu je vytvořit program, který bude měřit dobu reakce osoby po rozsvícení LED. K tomuto účelu využijeme tlačítko a červenou LED, která se bude rozsvěcet v náhodných časech. Čas mezi rozsvícením LED a stlačením tlačítka nazveme reakční dobou a budeme ho měřit v ms.

Program může provádět měření opakovaně, dokud bude reakční doba pod určitou maximální hranicí. Situace, kdy testovaná osoba stlačila tlačítko před rozsvícením, je nepříjemná a musí být ošetřena. Program bude reagovat pouze na stlačení tlačítka, nikoli na jeho uvolnění a dobu mezi stlačením a uvolněním tlačítka může testovaná osoba využít na prohlédnutí dosaženého výsledku, který je odeslán do PC. Program tedy bude pokračovat až po uvolnění tlačítka.

Pro získání náhodného čísla, reprezentujícího dobu, za kterou se LED rozsvítí, využijeme funkce standardní knihovny C, které Mbed implementuje. Generátor pseudonáhodných čísel je inicializován funkcí `srand()`. Na rozdíl od generátorů skutečně náhodných čísel (kvantové generátory, šum) je generátor pseudonáhodných založen na deterministickém algoritmu a proto je třeba jen inicializovat vhodným téměř náhodným číslem, např. počtem vteřin od 1. ledna 1970, které lze získat jako návratovou hodnotu `time(NULL)`. Samotné pseudonáhodné číslo je pak získáno jako návratová hodnota funkce `rand()`, která je typicky datového typu `unsigned int`. Náhodné číslo `num` v nějakém konkrétním intervalu `INTERVAL` posunutě o offset `OFFSET` lze získat např. takto (náhodné číslo využijeme pro reprezentaci času v intervalu 300–2300 ms):

```
1 | #define INTERVAL 2000
2 | #define OFFSET 300
3 |
4 | srand (time (NULL));
5 | uint32_t num = (rand() % INTERVAL) + OFFSET;
```

Dalším krokem je kontrola, zda nedošlo k předčasnému stlačení tlačítka. Kontrolu bude program provádět každou ms:

```
1 | //wait approximately num ms but watch the button
2 | while(num) {
3 |     num--;
4 |     if (!button) {
5 |         // the button pressed before the LED lights up
6 |         quit = true;
7 |         num = 0;
8 |     }
9 |     wait_ms (1);
10 | }
```

Pokud nebylo tlačítko stlačeno předčasně, následuje rozsvícení LED a změření reakční doby:

```

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

```

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

```

1 | //get the result in ms
2 | trial_time_in_ms = trial.read_ms();
3 | //the button was pressed in time
4 | if (trial_time_in_ms <= MAX_REACTION_TIME) {
5 |     serial.printf("\t%d ms\r\n", trial_time_in_ms);
6 |     trial.reset(); //DO NOT FORGET TO RESET
7 | } else {
8 |     serial.printf("\tTime: %d ms\r\n", trial_time_in_ms);
9 |     serial.printf("GAME OVER!!\n\r");
10 |     quit = true;
11 | }

```

Zkompletujte program v rámci samostatné práce.

5.4 Třída PwmOut

Další třída, která používá čítače, je `PwmOut`. Než se s touto třídou seznámíme blíže, začneme s motivačním příkladem. RG LED figurovala doposud téměř v každém příkladu. Rozsvítit a zhasnout, nic víc neumí. Znamená to ale, že již nemůže překvapit?

5.4.1 Motivační příklad

Program v kapitole 3.1 v nekonečném cyklu rozsvěcel na 200 ms, zhasne ji a čeká 1 s. To znamená, že poměr dob, kdy LED svítí a nesvítí, je v poměru 1:5. Tento poměr zachováme, avšak základní dobu, která je v tomto případě 1 s, změníme o řád na 100 ms.

```

1 | myled = 1; // LED is ON
2 | thread_sleep_for (20); // 20 ms
3 | myled = 0; // LED is OFF
4 | thread_sleep_for (100); // 100 ms

```

Co se stane? Dioda bude rychle blikat. Změní-li se základní doba o další řád (na 10 ms), pak LED již (zdánlivě, vlivem setrvačnosti lidského oka) blikat nebude. Proč? Lidské oko není schopné rozlišit tak rychlou změnu (rozsvíceno/zhasnuto). Dioda ovšem nesvítí s takovou intenzitou, jako když je zapnuta po celou dobu.

Aby byl tento efekt znatelnější, vytvoříme program, ve kterém se bude měnit výše zmíněný poměr (poměr mezi stavy rozsvíceno a zhasnuto) a to v rámci dostatečně krátké časové periody (např. 18 ms). Změna může kupříkladu nastat po stisknutí tlačítka. K obsluze tlačítka využijte přerušení.

Možné řešení:

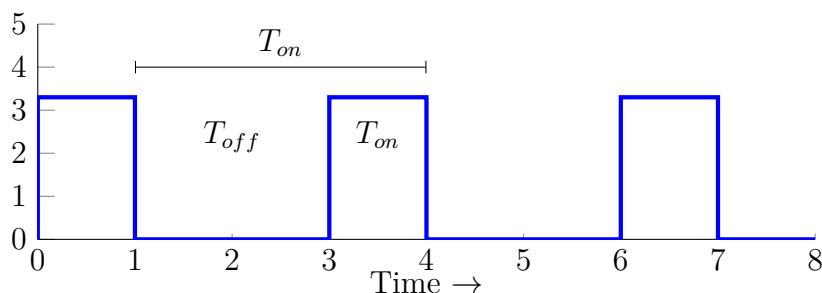
```
1  #include <mbed.h>
2
3  #define BASE_TIME 18 // in milliseconds
4  #define NUM_STEPS 9 // an integer <= BASE_TIME
5  #define STEP BASE_TIME/NUM_STEPS]
6
7  DigitalOut myled(LED0); // LED0 = PD8 ~ RED
8
9  InterruptIn btn1_i(BUTTON1); // BUTTON1 = PD14
10
11 uint8_t active_time = 0;
12
13 void btn_isr(){ // change active_time
14     active_time += STEP;
15     if (active_time > BASE_TIME){
16         active_time = 0;
17     }
18 }
19
20 int main() {
21     btn1_i.fall(btn_isr);
22     while(1) {
23         myled=1; // LED is ON
24         thread_sleep_for(active_time);
25         myled=0; // LED is OFF
26         thread_sleep_for(BASE_TIME-active_time);
27     }
28 }
```

Doba čekání po rozsvícení a zhasnutí LED se po stisku tlačítka mění. Začíná se ve stavu, kdy LED 100% času nesvítí (0% svítí). Pokud je `NUM_STEPS` rovno 18 (viz výše), pak se každým stiskem tlačítka zvětší doba, kdy LED svítí o cca 11%. Po prvním kroku LED svítí 11% času a 89% nesvítí. Tímto způsobem se svítivost diody postupně zvyšuje, až nakonec dosáhne stavu: 0% času nesvítí a 100% času svítí.

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

5.4.2 PWM – Pulsní šířková modulace

Vraťme se k prvnímu vytvořenému programu z kapitoly 3.1. Jak vypadá výstupní napětí na pinu `PA_5`? Ideální grafické znázornění je na obr. 5.1. Protože jsme takto jednoduše sestrojili pulzně-šířkově modulovaný sign. (angl. Pulse Width Modulation, zkráceně PWM).



Obrázek 5.1: Pulsní šířková modulace

V motivačním příkladu byl zmíněn poměr, kdy LED svítí a nesvítí, což by podle obr. 5.1 odpovídalo poměru T_{on} a T_{off} . Tato hodnota se standardně neuvádí, místo ní se však používá pojem střída (angl. duty cycle). Pro střidu D signálu platí:

$$D = 100 \cdot \frac{T_{on}}{T_{on} + T_{off}}$$

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

Při práci s PWM signálem je třeba mít na paměti, že nejmenším časová jednotka používaná v Mbed OS je 1 μ s. Proto v případě nastavení T_p na 7 μ s a střidy 50% Mbed OS nenastaví T_{on} na 3,5 μ s, nýbrž na 3 μ s.

5.4.3 Funkce třídy PwmOut

Využitím třídy `PwmOut` lze na zvoleném GPIO pinu vytvořit podobný signál jako na obr. 5.1. Způsob, jak mikrokontrolér PWM signál vytváří, je však odlišný – je využit čítač.

Třída `PwmOut` nabízí mnoho funkcí⁴, mezi které patří:

`period()` Nastavení periody signálu v s

`period_ms()` Nastavení periody signálu v ms

`period_us()` Nastavení periody signálu v μ s

⁴viz <https://os.mbed.com/docs/mbed-os/v6.15/apis/pwmout.html>

`pulsewidth()` Nastavení šířky pulzu T_{on} v s

`pulsewidth_ms()` Nastavení šířky pulzu T_{on} v ms

`pulsewidth_us()` Nastavení šířky pulzu T_{on} v μ s

`write()` Nastavení střídy PWM signálu

`read()` Získání hodnoty střídy PWM signálu

`suspend()` Zastavení generování PWM signálu

`resume()` Obnovení generování PWM signálu

Při generování PWM signálu je třeba nejprve nastavit perioda signálu a až potom šířka pulzu T_{on} . Maximální rozlišení v Mbed OS je v μ s. Střída PWM signálu je nepřímý parametr, při volání funkce `write()` se ve skutečnosti mění šířka pulzu T_{on} .

5.4.4 Příklad

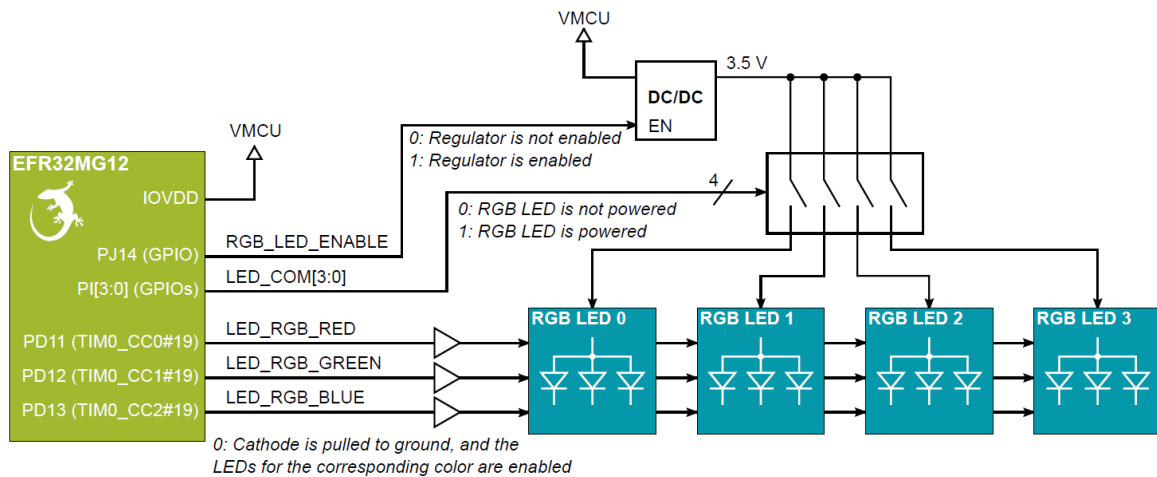
Vytvořte program s obdobnou funkcionalitou jako příklad v kap. 5.4.1, avšak s využitím třídy `PwmOut`, přičemž pro obsluhu tlačítka opět využijte přerušení. Použijte zelenou LED, červenou LED nelze třídou `PwmOut` využít.

```

1  #include <mbed.h>
2
3  #define BASE_TIME 18 // in milliseconds
4  #define NUM_STEPS 9 // an integer <= BASE_TIME
5  #define STEP BASE_TIME/NUM_STEPS
6
7  PwmOut myled_pwm(LED1); // LED1 = PD9 ~ GREEN
8  InterruptIn btn1_i(BUTTON1); // BUTTON1 = PD14
9
10 volatile uint8_t active_time = 9;
11
12 void btn_isr(){ // change pulse width
13     active_time += STEP;
14     if (active_time > BASE_TIME){
15         active_time = 0;
16     }
17     myled_pwm.pulsewidth_ms(active_time);
18 }
19
20 int main(){
21     myled_pwm.period_ms(BASE_TIME);
22     myled_pwm.pulsewidth_ms(active_time);
23     btn1_i.fall(&btn_isr);
24     while(1){}
25 }
```

5.4.5 Příklad

Thunderboard disponuje čtyřmi RGB LED[3]. Pro napájení všech RGB LED (viz obr. 5.2) je nutné aktivovat regulátor, který zajišťuje konstantní napájení diod, což se realizuje jednoduše nastavením pinu **PJ14** do log. 1. Obdobně se aktivuje i napájení jednotlivých diod pomocí pinů **PI0** až **PI3**. Pro nastavení RGB složek slouží piny **PD11**, **PD12** a **PD13**. Katody jednotlivých diod jsou spolu svázaný, proto nelze nastavovat barvy u každé diody zvlášť, ale je možná nastavovat pouze jednu společnou barvu.



Obrázek 5.2: Zapojení RGB LEDs (převzato z [7])

Vytvořte program, ve kterém budete prostřednictvím PC, konkrétně pomocí kláves R, G a B, měnit střidu jednotlivých barevných složek. Pro zpracovávání znaků z PC využijte přerušení.

Možné řešení:

```

1  #include <mbed.h>
2
3  #define BASE_TIME 10 //in milliseconds
4  #define NUM_STEPS 5 //an integer <= BASE_TIME
5  #define STEP BASE_TIME/NUM_STEPS
6
7  PwmOut red_pwm(PD11);
8  PwmOut green_pwm(PD12);
9  PwmOut blue_pwm(PD13);
10
11 DigitalOut regulator(PJ14, 1);
12 DigitalOut led1(PI0, 1);
13 DigitalOut led2(PI1, 1);
14 DigitalOut led3(PI2, 1);
15 DigitalOut led4(PI3, 1);
16
17 UnbufferedSerial serial_port(USBTX, USBRX, 115200);
18

```

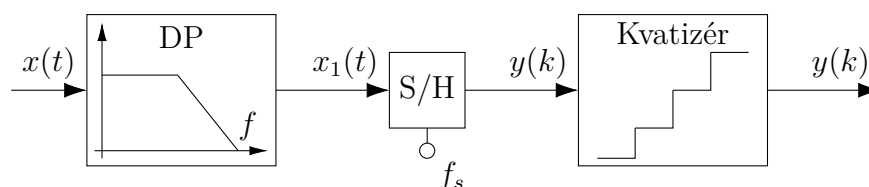
```
19  InterruptIn btn1_i(BUTTON1); // BUTTON1 = PD14
20
21  volatile uint8_t R_active_time = 0;
22  volatile uint8_t G_active_time = 0;
23  volatile uint8_t B_active_time = 0;
24
25  void serial_isr()
26  {
27    char c;
28    if (serial_port.read(&c, 1)) { // Read the data to clear
29      the receive interrupt.
30      serial_port.write(&c, 1); // Echo the input back to the
31      terminal.
32      if (c=='r' || c=='R'){
33        R_active_time += STEP;
34        if (R_active_time > BASE_TIME){
35          R_active_time = 0;
36        }
37        red_pwm.pulsewidth_ms(R_active_time);
38      } else if (c=='g' || c=='G'){
39        G_active_time += STEP;
40        if (G_active_time > BASE_TIME){
41          G_active_time = 0;
42        }
43        green_pwm.pulsewidth_ms(G_active_time);
44      } else if (c=='b' || c=='B'){
45        B_active_time += STEP;
46        if (B_active_time > BASE_TIME){
47          B_active_time = 0;
48        }
49        blue_pwm.pulsewidth_ms(B_active_time);
50      } else {}
51    }
52  }
53
54  int main(){
55    red_pwm.period_ms(BASE_TIME);
56    green_pwm.period_ms(BASE_TIME);
57    blue_pwm.period_ms(BASE_TIME);
58    red_pwm.pulsewidth_ms(R_active_time);
59    green_pwm.pulsewidth_ms(G_active_time);
60    blue_pwm.pulsewidth_ms(B_active_time);
61    serial_port.attach(&serial_isr, SerialBase::RxIrq);
62    while(1){}
```

6 | Analogový svět

Doposud jsme využívali GPIO brány prostřednictvím tříd `DigitalIn` a `DigitalOut` v režimu digitálního vstupu a výstupu. Pokud chceme pracovat v analogovém režimu, např. měřit výstupní napětí senzorů fyzikálních veličin (teplota, světlo, tlak, ...), musí být mikrokontrolér vybaven vhodným převodníkem. Procesor na desce Thunderboard disponuje jak analogově-digitálním, tak dvěma digitálně-analogovými převodníky. Ovšem zatímco třídu `AnalogIn` lze využít, digitálně-analogový převodník Thunderboardu podporován není¹ a třídu `AnalogOut` využít nelze.

6.1 Analogově-digitální převodník

K převodu analogového signálu $x(t)$ do digitální formy $y(k)$ je třeba nejprve analogový signál vzorkovat v diskrétních časových intervalech a získaným vzorkům po kvantizaci přiřadit hodnotu ve vhodném binárním kódu. Obecné schéma převodníku je vyobrazeno na Obr. 6.1. Samotné vzorkování je realizováno obvodem Sample&Hold (S/H), před který bývá zařazena dolní propust z důvodu omezení aliasingu.

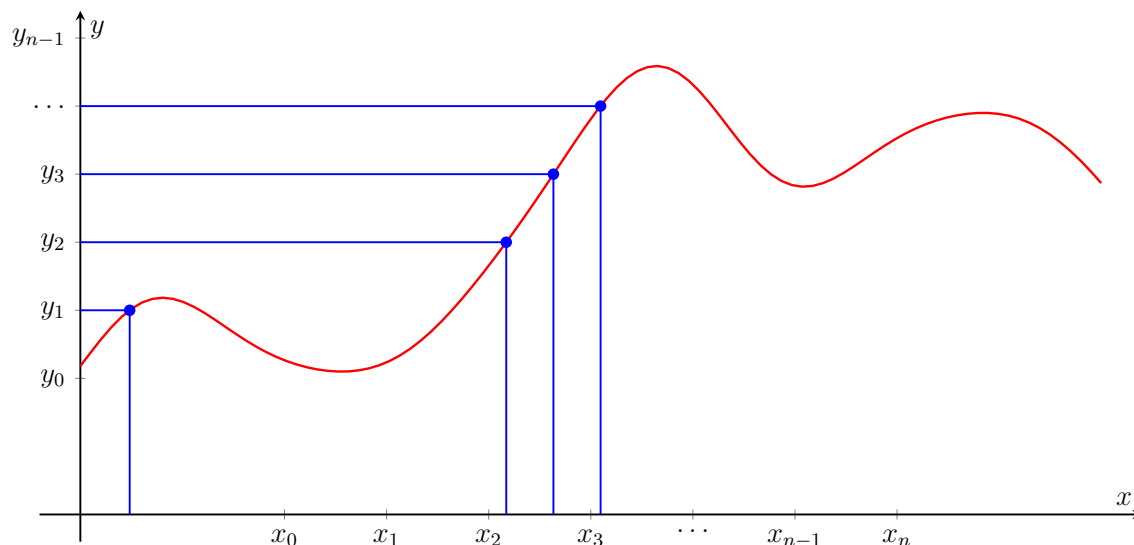


Obrázek 6.1: Generic scheme for an ADC.

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

Na rychlost vzorkování a kvalitu kvantizace lze klást řadu podmínek. Pro naše účely je důležité vědět, že Thunderboard disponuje jedním 12 bitovým 1 Msps (mega samples per second) ADC a dvěma VDAC (písmeno V pouze značí novější provedení DAC) se vzorkovací frekvencí 500 ksps.

¹viz <https://github.com/ARMmbed/mbed-os/issues/7240>



Obrázek 6.2: Příklad vzorkování analogového signálu

6.2 Třída `AnalogIn`

Jak název napovídá, třída umožňuje nakonfigurovat pin GPIO brány pro měření analogového signálu. ADC nabízí značné množství parametrů, které se dají nastavovat. V mbedu však tyto parametry nastavovat nemůžeme, jsou zadány „napevno“. Pevné nastavení parametrů má své nevýhody, např. problém s měřením napětí s vysokým vstupním odporem (viz úvod kap. 4.5 v [2]). [1]

Třída `AnalogIn` obsahuje pouze dvě metody:

`read()` Vrací číslo datového typu `float` v intervalu 0–1, 0 odpovídá spojení s GND, 1 odpovídá napájecímu napětí 3.3 V

`read_u16()` Vrací 16 bitové číslo datového typu `uint16_t`, 0 odpovídá spojení s GND, 65536 odpovídá napájecímu napětí 3.3 V

Problematika vybrání vhodného pinu pro použití třídou `AnalogIn` opět není triviální a musí se dohledávat v datasheetu (viz [10]) nebo lépe v referenčním manuálu (viz [9]). Pro vyzkoušení funkce probrané třídy bychom k desce připojili různá napětí a testovali, zda je naměřené napětí správné. My však k Thunderboardu nic připojovat nebudeme a funkčnost třídy si vyzkoušíme trochu jinak (viz další kapitola).

6.3 Příklad – generování náhodných čísel

V příkladu 5.3.1 byl použit generátor pseudonáhodných čísel. Generátor skutečných náhodných čísel lze vytvořit ADC, který není připojen k žádnému zdroji napětí (floating input).

Klíčová část kódu by se mohla změnit např. takto:

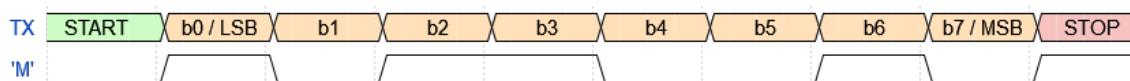
```
1  #include "mbed.h"
2
3  AnalogIn floating_input(PF5);
4
5  int main() {
6      srand (floating_input.read_u16());
7
8      for (int x = 0; x < 5; x++) {
9          printf ("num %x: %d\n\r", x, rand() % 20);
10         thread_sleep_for(300);
11     }
12 }
```


7 | Sériové sběrnice

Pro komunikaci s vnějšími periferiemi se nejčastěji používají sériové sběrnice, tvořené zpravidla dvěma až třemi vodiči. Přenos dat bývá řízen synchronizačním hodinovým signálem, zbytek vodičů je určen pro přenos dat. Typickými zástupci jsou sběrnice **SPI** (Serial Peripheral Interface) a **I²C** (Inter-Integrated Circuit), které bývají často označovány termínem přístrojové sběrnice. U těchto sběrnic navíc nejméně jeden z účastníků přenosu vystupuje v roli řadiče sběrnice (master), ostatní zařízení bývají podřízená (slave). Úkolem řadiče je generování hodinového signálu a případně řízení sběrnice. Kromě zmíněných dvou sběrnic se podíváme ještě na sériové rozhraní **UART**, se kterým jsme se seznámili již v kapitole 3.4.

7.1 UART

Sériové rozhraní UART transformuje přicházející/odcházející data z/do sériového bitového proudu. Rozhraní bylo navrženo pro přenos znaků, pro jejichž kódování je třeba 7 bitů. Kromě užitečné informace jsou zároveň přenášeny informace pro synchronizaci (povinný START bit a nepovinný STOP bit), případně paritní bit pro kontrolu správnosti informace. Na obr. 7.1 vidíme zakódovaný znak 'M', který má v ASCII tabulce přiřazené číslo 77, což je binárně **0b01001101**. První bit je START bit, za ním následují datové bity b0–b7. Jednotlivé datové bity jsou ve výsledné sekvenci zakódovány obráceně, tzn. nejvíce významný bit MSB (Most Significant Bit) je až na konci (před STOP bitem) a nejméně významný bit LSB (Least Significant Bit) je první (ihned za START bitem). Na konci je již zmíněný STOP bit, kterých může být i několik.



Obrázek 7.1: Příklad přenosu znaku sběrnici UART.

Písmeno M je tedy zakódováno do desetibitové sekvence **0101100101**, která se pošle po sériové lince zvolenou rychlostí (baudrate). Poznamenejme, že pokud je kupříkladu baudrate nastavena na 9600, pak doba trvání jednoho bitu je $1/9600$ s, což je cca 0,1042 ms. Pokud se po sériové lince neposílají data, pak je na ni napětí odpovídající log. 1, nikoli log. 0.

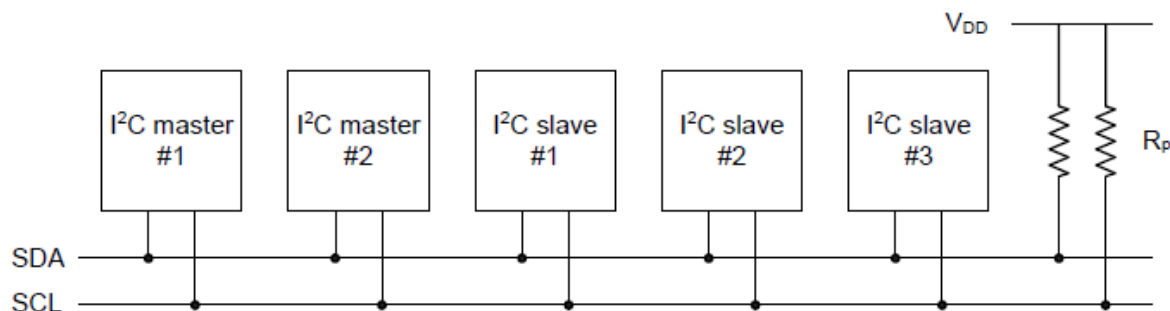
Protože UART nepoužívá synchronizační hodinový signál, je třeba, aby obě komunikující zařízení byla nastavena na stejnou komunikační rychlost. Moderní mikrokon-

troléry mívají ve výbavě pokročilejší rozhraní USART (Universal Synchronous Asynchronous Receiver Transmitter), které může pracovat v synchronním módu, ve kterém periférie posílající data generuje také hodinový signál. Synchronní mód je zpravidla poloduplexní, protože jeden vodič je použit pro přenos synchronizačního hodinového signálu, generovaného vysílající stranou.

USART je obecně oproti UART mnohem komplikovanější, umožňuje však generovat data v mnoha formátech požadovaných v různých standardizovaných protokolech (např. IrDA, LIN, Smart Card atd.). USART v asynchronním módu se chová stejně jako UART. Umožňuje tedy generovat stejná data jako na obr. ??.

7.2 I²C

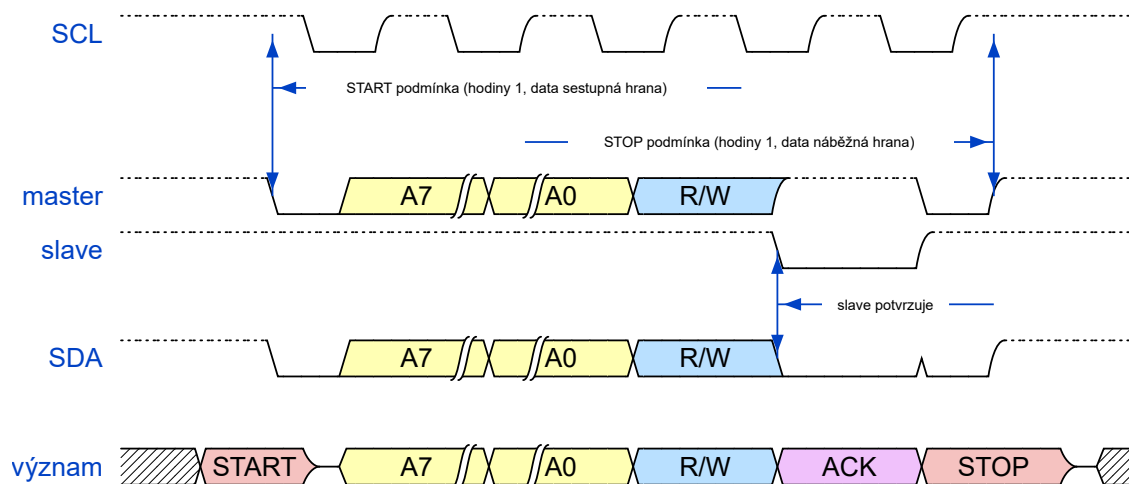
I²C je sériová sběrnice, která umožňuje komunikovat s vnějšími perifériemi prostřednictvím dvou vodičů. Vodič SDA (Serial Data Line) je určen pro obousměrný přenos dat a vodič SCL (Serial Clock Line) je určen pro přenos synchronizačního hodinového signálu. K jedné sběrnici I²C může být zpravidla připojeno až 128 zařízení. Toto omezení je dáno velikostí adresního prostoru (viz dále). Obecně se jedná o multi-master sběrnici, což znamená, že několik zařízení může vykonávat roli řadiče sběrnice. Příklad zapojení sběrnice je na obr. 7.2.



Obrázek 7.2: Příklad zapojení zařízení pro komunikaci prostřednictvím sběrnice I²C.

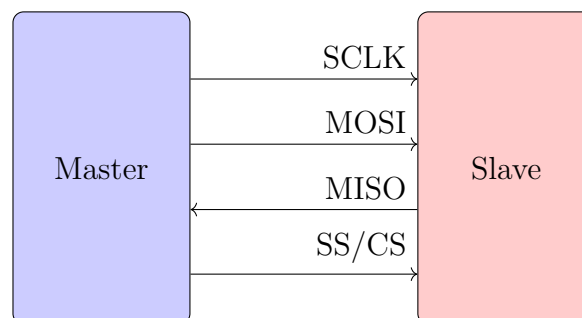
Na obr. ?? také vidíme (v pravé části) dva rezistory, pomocí kterých jsou vodiče SDA a SCL připojeny k napájení. Oba vodiče jsou v tzv. open-drain zapojení, což znamená, že zařízení přivádějí na svůj výstup log. 0 (spínají do nuly). Komunikaci vždy iniciuje master, slave začne posílat data když o ně master požádá.

Zjednodušený příklad komunikace na I²C sběrnici lze vidět na obr. ?. Když se master rozhodne začít komunikaci, začne START bitem, sestupná hrana STOP bitu musí přijít v intervalu, kdy je hodinový signál v log. 1. Následuje sedmibitová adresa slave zařízení, se kterým chce komunikovat. Jednotlivá zařízení tedy mají přidělenou adresu, pomocí které se rozeznávají. Osmý bit rozhoduje o tom, jestli chce master zapisovat nebo číst. Po přijetí osmi bitů potvrzuje slave ACK bitem a následuje STOP bit, jehož náběžná hrana opět přichází v intervalu, jsou je hodinový signál v log. 1. Po adresaci se posílají data v osmibitových rámcích. Více informací o I²C najdete např. v [23]. [9] [23]

Obrázek 7.3: Komunikační rámec sběrnice I²C.

7.3 SPI

SPI (Serial Peripheral Interface) představuje poměrně rychlou sběrnici, která zprostředkovává obousměrnou synchronní sériovou komunikaci po třech vodičích. Jeden z účastníků je vždy v roli mastera a druhý roli slave, příklad propojení je na obr. ??



Obrázek 7.4: Sériové rozhraní SPI.

Význam zkratk jednotlivých pinů:

SCLK Serial Clock, synchronizační hodinový signál

MOSI Master Out Slave In, komunikace Master → Slave

MISO Master In, Slave Out, komunikace Slave → Master

SS/CS Slave Select / Chip Select , výběr daného zařízení

Princip komunikace je následující. Master řídí celou komunikaci mezi zařízeními. Do všech zařízení typu Slave generuje hodinové sign. (SCK). Pokud chce s daným Slavem komunikovat, pak ho pomocí pinu NSS určí, Slave to pozná a začíná komunikace. Master pak data posílá po pinu MOSI, zatímco Slave po pinu MISO. [1]

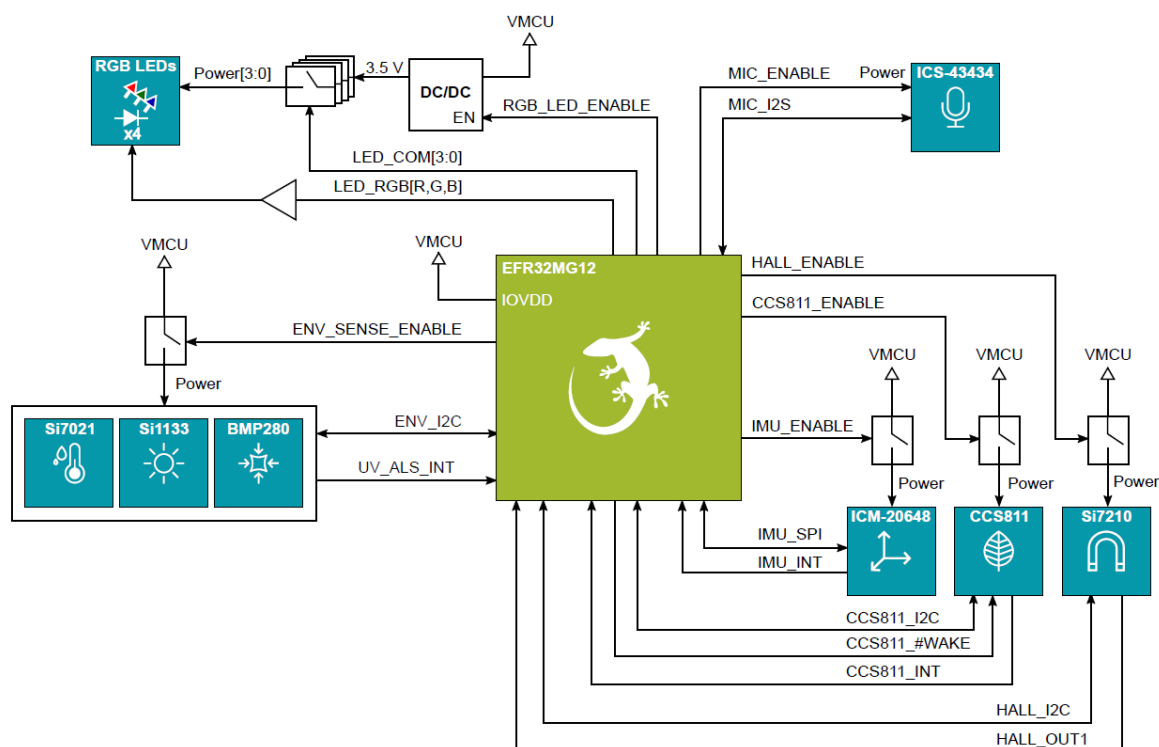
Jak již bylo řečeno, pomocí SPI se realizuje komunikace s šestiosým inerciálním senzorem. V podstatě se opět nemusíme o nic starat, jelikož celá komunikace je již naimplementována v knihovně, kterou budeme využívat.

8 | Komunikace s periferiemi

V této kapitole se zaměříme na zapojení jednotlivých senzorů na desce Thunderboard a komunikaci s ním. Zdrojem informací pro tuto část je převážně [7]

8.1 Zapojení senzorů na desce Thunderboard

Blokové schéma desky Thunderboard je zobrazeno na obr. 8.1. Ovládání RGB LED již známe z kap. 4.8.5, tím se tedy již zabývat nebudeme. Z obr. 8.1 je patrné, že je vždy potřeba senzoru aktivovat napájení, jinak zkratka nebude fungovat. Ze schématu je také patrné, že senzory Si7021, Si1133 a BMP280 sdílí stejnou sběrnici a napájecí vodič. Aktivací `ENV_I2C` se proto připojí k napájení všechny tři senzory.



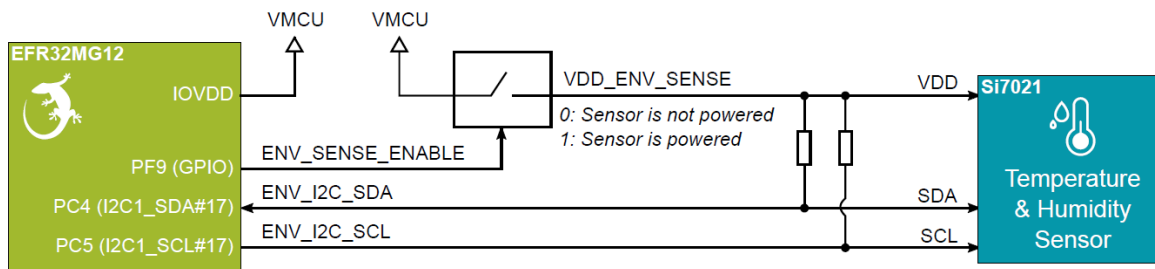
Obrázek 8.1: Blokové schéma desky SL Thunderboard.

V následujícím textu se seznámíme s jednotlivými senzory, a to jak s jejich zapojením, tak se způsobem, jak senzor správně nakonfigurovat a přečíst z něj data.

Mikrofonem se zabývat nebudeme.

8.1.1 Si7021 Relative Humidity and Temperature Sensor

Napájení senzoru Si7021 je aktivováno přivedením log. 1 PF9 (`ENV_SENSE_ENABLE` na obr. 8.2). Přivedení log. 1 na tento pin způsobí také aktivaci senzorů Si1133 a BMP280.

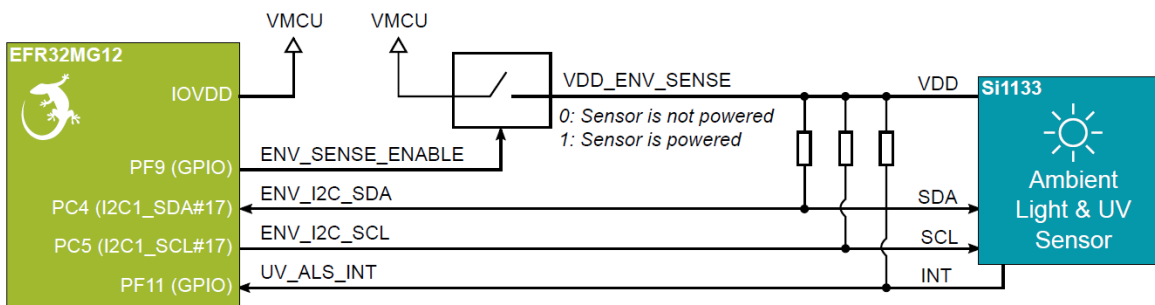


Obrázek 8.2: Si7021 Relative Humidity and Temperature Sensor.

Dále k senzoru Si7021 vedou vodiče `ENV_I2C_SDA` (pin PC4) a `ENV_I2C_SCL` (pin PC5), které připojují senzor k mikrokontroléru prostřednictvím sběrnice I²C.

8.1.2 Si1133 UV Index and Ambient Light Sensor

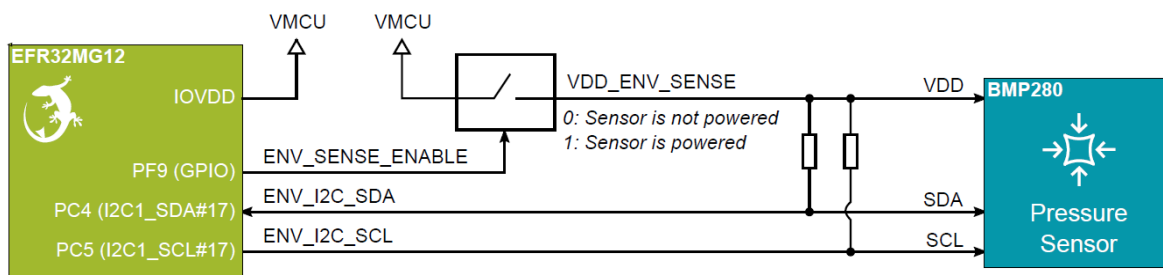
Zapojení senzoru Si1133 (viz obr. 8.3) je velmi podobné zapojení senzoru Si7021. Rozdíl je v tom, že k senzoru Si1133 vede navíc vodič `UV_ALS_INT`, který umožňuje vyvolat přerušování na konci měření okolního světla.



Obrázek 8.3: Si1133 UV Index and Ambient Light Sensor.

8.1.3 BMP280 Barometric Pressure Sensor

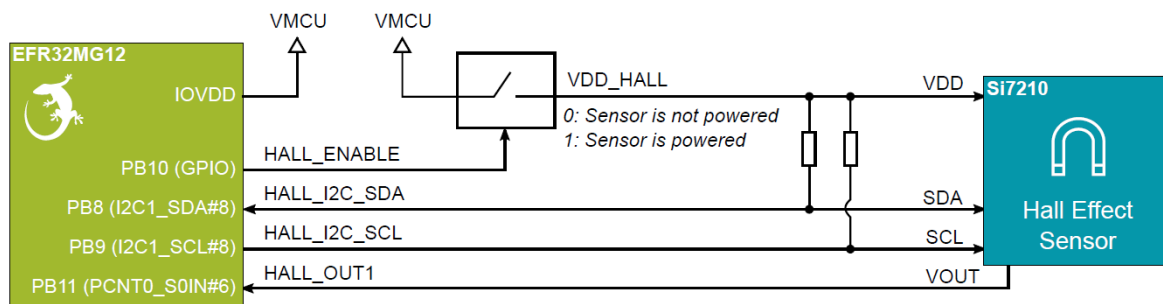
Zapojení senzoru BMP280 (viz obr. 8.4) je identické zapojení senzoru Si7021, senzor je připojen na stejnou sběrnici I²C.



Obrázek 8.4: BMP280 Barometric Pressure Sensor.

8.1.4 Si7210 Hall Effect Sensor

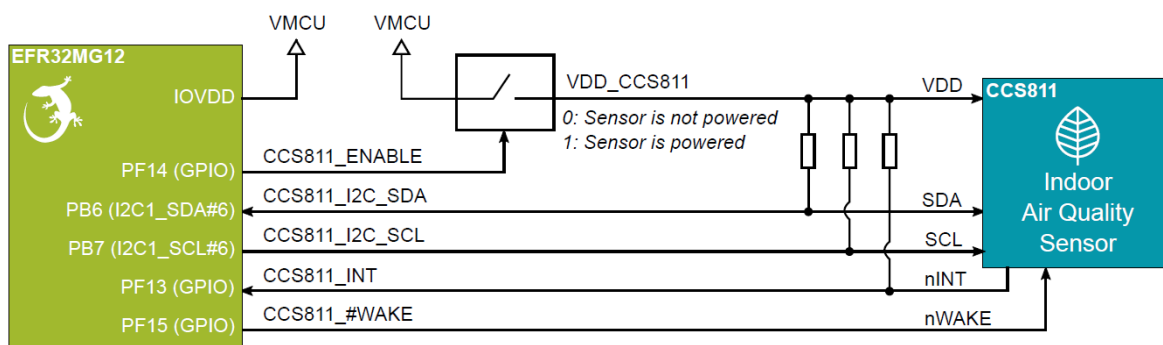
Napájení Si7210 je aktivováno log. 1 na pinu **PB10** (**HALL_ENABLE** na obr. 8.5) a senzor komunikace prostřednictvím I²C pomocí pinů **PB8** (**HALL_I2C_SDA**) a **PB9** (**HALL_I2C_SCL**). Vodič **HALL_OUT1** připojený na pin **PB11** může sloužit pro generování digitálního poplachu, pokud hodnota měřeného magnetického pole přesáhne předem definovaný práh.



Obrázek 8.5: Si7210 Hall Effect Sensor.

8.1.5 CCS811 Indoor Air Quality Gas Sensor

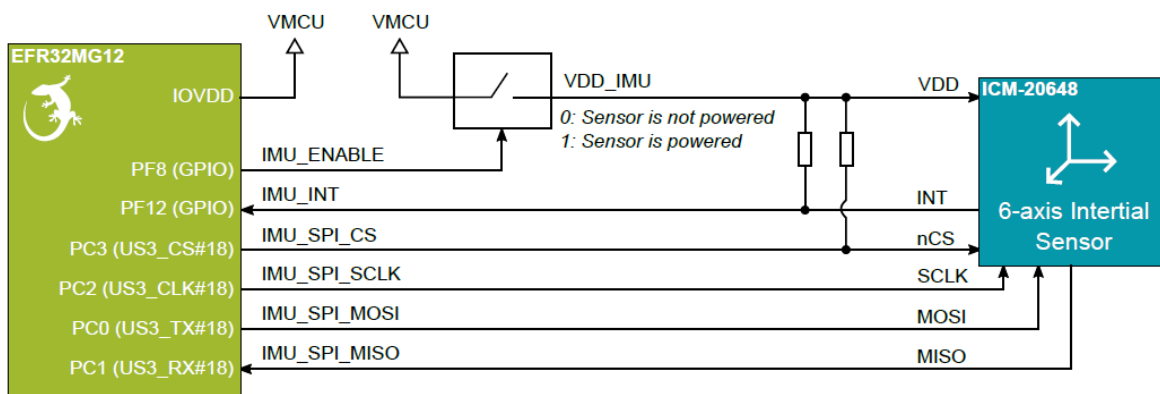
Napájení pro senzor CCS811 se aktivuje pinem **PF14** (**CCS811_ENABLE** na obr. 8.6) a k I²C sběrnici se senzor připojen pomocí pinů **PB6** a **PB7**. Pin **PF13** (**CCS811_INT**) může být použit pro generování přerušování v případě, že jsou k dispozici nové vzorky. Pinem **PF15** se senzor budí ze spánku.



Obrázek 8.6: CCS811 Indoor Air Quality Gas Sensor.

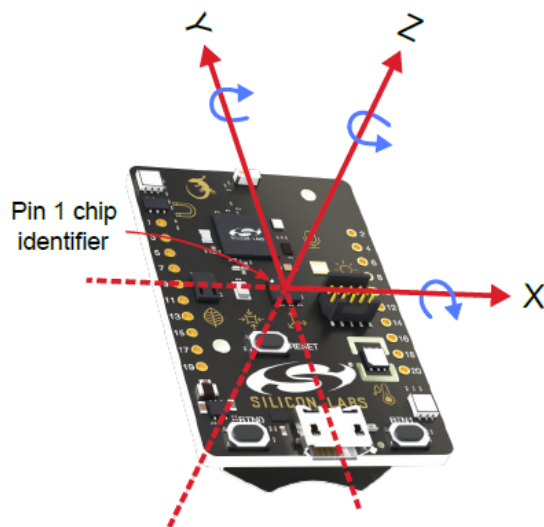
8.1.6 ICM-20648 6-Axis Inertial Sensor

Šestiosý inerciální senzor využívá pro komunikaci rozhraní SPI (viz obr. 8.7). Pro aktivaci senzoru slouží vodič `IMU_SPI_CS` (`PC3`) a pro vedení hodinového signálu vodič `IMU_SPI_SCLK` (`PC2`). Data do senzoru se posílají vodičem `IMU_SPI_MOSI` (`PC0`) a ke čtení ze senzoru slouží vodič `IMU_SPI_MISO` (`PC1`). Napájení je aktivováno log. 1 na pinu `PF8`.



Obrázek 8.7: ICM-20648 6-Axis Inertial Sensor.

Získaná data o poloze je samozřejmě potřeba vztáhnout k nějaké souřadné soustavě. Základní souřadnou soustavu vidíme na obr. 8.8.



Obrázek 8.8: Orientace os na Thunderboardu.

8.2 Příklad – získání dat ze senzorů

V následujícím textu si , jak získat data ze senzorů nacházejících se na Thunderboardu. Budeme postupovat podle následujícího postupu:

1. Vytvořte si nový projekt.
2. Stáhněte si knihovny jednotlivých senzorů, názvy knihoven jsou: AMS_CSS8111, AMS_ENS210, BMP280, ICM20648, Si1133, Si7021 a Si7210, přičemž vždy se jedná o hlavičkový a zdrojový soubor.
3. Do složky include vložte hlavičkové soubory a do složky src zdrojové soubory.

Pozn.: celý příklad je převzatý z [17]. Pro správnou funkci bylo potřeba opravit resp. aktualizovat některé části kódu, jelikož projekt byl vytvořen na začátku roku 2017 a není zcela kompatibilní s aktuální verzí Mbed. V tomto textu je rozdělen na dílčí části, ze kterých by mělo být jasně vidět, jak probíhá komunikace s jednotlivými senzory. Pro konverzi dat senzoru, které jsou datového typu `float`, využijeme funkci `ftos()` vytvořenou v kapitole¹.

8.2.1 Inicializace

Prvním krokem je vložení potřebných knihoven:

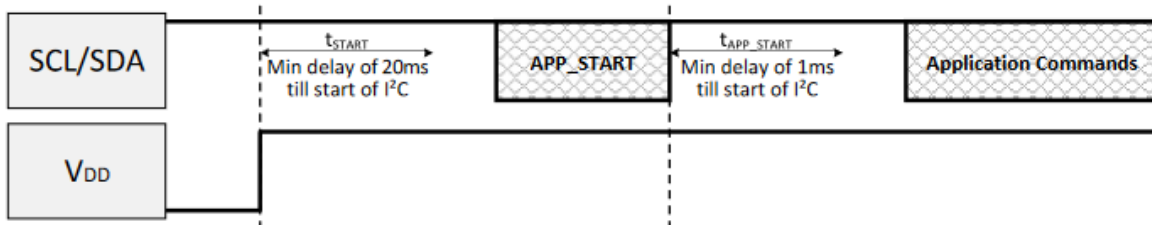
```
1 #include <string>
2 #include "mbed.h"
3 #include "BMP280.h" // barometric pressure sensor
4 #include "AMS_CCS811.h" // indoor air quality gas sensor
5 #include "Si7021.h" // relative hum. and temp. sensor
6 #include "Si7210.h" // hall effect sensor
7 #include "Si1133.h" // UV index and ambient light sensor
8 #include "ICM20648.h" // 6-axis inertial sensor
```

Dále je potřeba nakonfigurovat piny `PF9`, `PF14`, `PF15`, `PB10` a `PB8` jako DigitalOut. Pomocí nich aktivujeme hodiny v příslušných senzorech a pomocí pinu `PF15` se probouzí senzor CCS811 (viz kap. 8.1.5).

```
1 /* Turn on power supply to ENV sensor suite
2    Si7021, Si1133 and BMP280 */
3 DigitalOut env_en(PF9, 1);
4 // Turn on power to CCS811 sensor
5 DigitalOut ccs_en(PF14, 1);
6 // Set CCS_WAKE pin to output
7 DigitalOut ccs_wake(PF15, 1);
8 // Turn on power to hall effect sensor
9 DigitalOut hall_en(PB10, 1);
10 // Turn on power to hall effect sensor
11 DigitalOut hall_en(PB10, 1);
12 // Turn on power to IMU
13 DigitalOut imu_en(PF8, 1);
```

¹FIXME

Před začátkem inicializace senzorů počkáme nějaký čas, aby se stihly jednotlivé senzory uvést do aktivního módu. Jako první budeme inicializovat CCS811 a právě u tohoto senzoru je potřeba mít přivedené napájení alespoň 20 ms před začátkem komunikace (viz obr. 8.9).



Obrázek 8.9: CCS811 časování.

Abychom tuto prodlevu dodrželi, tak přidáme čekání kupříkladu 100 ms. Zkuste si sami tuto dobu zkrátit např. na 5 ms a uvidíte, že se CCS811 nenainicializuje správně.

8.2.2 Inicializace a čtení senzoru CCS811

Senzor kvality vzduchu CCS811 je mikrokontroléru připojen po sběrnici I²C, která je tvořena piny **PB6** a **PB7**. Argumentem konstruktoru třídy **AMS_CCS811** je instance třídy **I2C** a wake pin **PF15**. V tomto případě musíme nastavit i mód senzoru, který určuje frekvenci měření, nastavme např. každých 10 sekund. Najděte v **AMS_CCS811.h** proměnnou typu **enum** s názvem **OP_MODES** a podívejte se na možnosti nastavení.

```

1  I2C ccs_i2c (PB6, PB7);
2  bool gassensor_en = true;
3  uint32_t eco2, tvoc;
4
5  /* Initialize air quality sensor */
6  AMS_CCS811* gasSensor = new AMS_CCS811 (&ccs_i2c, PF15);
7  if(!gasSensor->init()) {
8      printf("Failed CCS811 init\r\n");
9      gassensor_en = false;
10 } else {
11     if(!gasSensor->mode(AMS_CCS811::TEN_SECOND)) {
12         printf("Failed to set CCS811 mode\r\n");
13         gassensor_en = false;
14     }
15 }
16
17 /* Measure air quality */
18 if(gassensor_en) {
19     gasSensor->has_new_data();
20     eco2 = gasSensor->co2_read();
21     tvoc = gasSensor->tvoc_read();
22

```

```
23     printf("  CCS811:\r\n");
24     printf("CO2: %ld ppm\r\n", eco2);
25     printf("VoC: %ld ppb\r\n", tvoc);
26 }
```

8.2.3 Inicializace a čtení senzoru BMP280

Inicializace senzoru BMP280 proběhne vytvořením instance třídy `BMP280`, argumentem konstrukturu je instance třídy `I2C`, popisující připojení. Návrátové hodnoty při čtení teploty a tlaku jsou datovém typu `float`, takže je nutné pro vypsání hodnoty použít rutinu pro převod.

```
1  I2C env_i2c(PC4, PC5);
2
3  float pressure, temperature;
4  char buff[10];
5
6  BMP280* pressureSensor = new BMP280(env_i2c);
7  temperature = pressureSensor->getTemperature();
8  pressure = pressureSensor->getPressure();
9
10 printf("  BMP280:\r\n");
11 ftos(pressure, buff);
12 printf("Pressure: %s bar\r\n", buff);
13 ftos(temperature, buff);
14 printf("Temperature: %s degC\r\n", buff);
```

8.2.4 Inicializace a čtení senzoru Si7021

Argumentem konstrukturu třídy `Si7021`, která umožňuje přístup k senzoru, není instance `I2C`, ale přímo symbolická označení pinů, které sběrnici tvoří. Návrátové hodnoty jsou celočíselné. Aby bylo možné pracovat s desetinnými čísly, reálná čísla odpovídající teplotě a relativní vlhkosti jsou vnitřně násobena 1000.

```
1  int32_t temperature;
2  uint32_t humidity;
3
4  /* Initialize RHT */
5  Si7021* rhtSensor = new Si7021(PC4, PC5);
6
7  /* Measure temperature and humidity */
8  rhtSensor->measure();
9  rhtSensor->measure();
10 temperature = rhtSensor->get_temperature();
11 humidity = rhtSensor->get_humidity();
```

```

12
13 printf(" Si7021:\r\n");
14 printf("Temperature: %ld.%03ld degC\r\n", temperature/1000,
15 abs(temperature%1000));
16 printf("Humidity: %ld.%03ld %%\r\n", humidity/1000, humidity
    %1000);

```

8.2.5 Inicializace a čtení senzoru Si7210

Instance třídy Si7210 se inicializuje podobně jednoduše, jako u předchozích senzorů připojených na sběrnici I²C. Nově si navíc vyzkoušíme přečtení libovolného registru (např. SI72XX_HREVID) z paměti senzoru. K tomu třída disponuje funkcí `readRegister()`, které se zadá název registru a ukazatel na proměnnou datového typu `uint8_t`, do které se výsledek uloží. Adresy registrů jsou definované jako makra v Si7210.h. Senzor pracuje v úsporném režimu a před čtením hodnot z registrů je třeba jej probudit voláním funkce `wakeup()`. Při měření voláním funkce `measureOnce()` je probouzecká funkce volána implicitně.

```

1 I2C hall_i2c(PB8, PB9);
2
3 /* Initialize Si7210 and check if it is alive */
4 silabs::Si7210* hallSensor = new silabs::Si7210(&hall_i2c);
5 uint8_t id;
6 hallSensor->wakeup();
7 hallSensor->readRegister(SI72XX_HREVID, &id);
8 printf("Hall ID: %d\r\n", id);

```

Jakou hodnotu bychom měli ze senzoru dostat? To zjistíme v datasheetu senzoru (viz [25]) nebo jednodušeji na obr. 8.10. Po spuštění programu nezapomeňte zkontrolovat, zda senzor vrací správnou hodnotu. Hodnota chipid je `0x01` a hodnota revid je `0x04`.

| ADDR | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|-------------|---|---|---|------------|---|---|---|
| 0xC0 | chipid (RO) | | | | revid (RO) | | | |

Obrázek 8.10: Obsah registru SI72XX_HREVID ležícím na adrese 0xC0.

```

1 /* measure HALL */
2 hallSensor->measureOnce();
3
4 printf(" Si7210:\r\n");
5 printf("Temperature: %d.%02d degC\r\n",
6 hallSensor->getTemperature()/1000,
7 abs(hallSensor->getTemperature()%1000));
8 printf("Magn. Field Strenth: %i.%03d mT\r\n",

```

```
9 |         (int)(hallSensor->getFieldStrength()/1000),
10 |         abs(hallSensor->getFieldStrength() % 1000));
```

8.2.6 Inicializace a čtení senzorů Si1133 a ICM20648

Inicializace senzorů Si1133 a ICM20648 se provede vytvořením objektu třídy a následným zavoláním funkce open():

```
1 | bool lightsensor_en = true;
2 | float light, uv;
3 | char buff1[10];
4 |
5 | /* Initialize light sensor */
6 | Si1133* lightSensor = new Si1133(PC4, PC5);
7 | if(!lightSensor->open()) {
8 |     printf("Something is wrong with Si1133, disabling..\n");
9 |     lightsensor_en = false;
10 | }
11 |
12 | /* measure light */
13 | if(lightsensor_en) {
14 |     lightSensor->get_light_and_uv(&light, &uv);
15 |
16 |     printf("  BMP280:\r\n");
17 |     ftos(light, buff1);
18 |     printf("Light: %s lux\r\n", buff1);
19 |     ftos(uv, buff1);
20 |     printf("UV: %s\r\n", buff1);
21 | }

1 | bool imu_en = true;
2 | float acc_x, acc_y, acc_z, gyr_x, gyr_y, gyr_z;
3 | char buff1[10], buff2[10], buff3[10];
4 |
5 | /* Initialize the IMU*/
6 | ICM20648* imu = new ICM20648(PC0, PC1, PC2, PC3, PF12);
7 | if(!imu->open()) {
8 |     printf("Something is wrong with ICM20648, disabling..\n");
9 |     imu_en = false;
10 | }
11 |
12 | /* measure imu */
13 | if(imu_en) {
14 |     imu->get_temperature(&temperature3);
15 |     imu->get_accelerometer(&acc_x, &acc_y, &acc_z);
16 |     imu->get_gyroscope(&gyr_x, &gyr_y, &gyr_z);
```

```
17
18     printf("   ICM20648:\r\n");
19     ftos(temperature3, buff1);
20     printf("Temperature: %s degC\r\n", buff1);
21     ftos(acc_x, buff1);
22     ftos(acc_y, buff2);
23     ftos(acc_z, buff3);
24     printf("Acc: %s %s %s\r\n", buff1, buff2, buff3);
25     ftos(gyr_x, buff1);
26     ftos(gyr_y, buff2);
27     ftos(gyr_z, buff3);
28     printf("Gyro: %s %s %s\r\n", buff1, buff2, buff3);
29 }
```

Literatura

- [1] Jan Kočí. Výuka vestavných systémů s využitím mbed. Master's thesis.
- [2] Silicon Labs. Sltb004a: Explore the capabilities of thunderboard sense. Online: <https://www.silabs.com/development-tools/thunderboard/thunderboard-sense-two-kit>. cit. 2021-9-12.
- [3] Silicon Labs. Ug309: Thunderboard sense 2 user's guide. Online: <https://www.silabs.com/documents/public/user-guides/ug309-sltb004a-user-guide.pdf>. cit: 2021-9-12.
- [4] Silicon Labs. Ug313: Thunderboard sense 2 bluetooth low energy demo user's guide. Online: <https://www.silabs.com/documents/public/user-guides/ug313-sltb004a-app-user-guide.pdf>. cit: 2021-9-12.

