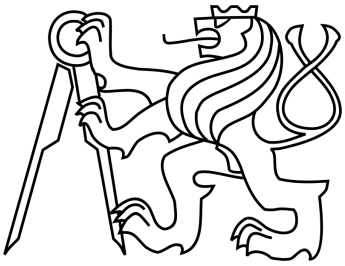
České vysoké učení technické v Praze
Fakulta elektrotechnická



Katedra elektrotechnologie
Obor: Elektrotechnika, energetika a management
Zaměření: Technologické systémy

Aplikace RTM pro přípravek Discovery
RTM application for Discovery Kit

DIPLOMOVÁ PRÁCE

Vypracoval: Arnošt Šenkýř
Vedoucí práce: Ing. Karel Künzel, CSc.
Rok: 2019

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne .....

.....
Arnošt Šenkýř

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Šenkýř** Jméno: **Arnošt** Osobní číslo: **406279**

Fakulta/ústav: **Fakulta elektrotechnická**

Katedra/ústav: **Katedra elektrotechnologie**

Studijní program: **Elektrotechnika, energetika a management**

Studijní obor: **Technologické systémy**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Aplikace RTM pro přípravek Discovery

Název diplomové práce anglicky:

RTM application for Discovery Kit

Pokyny pro vypracování:

1. Seznamte se s možnostmi přípípravku STM32F428 Discovery Kit
2. Seznamte se s koncepcí programu RTM
3. Navrhněte programové moduly (HAL) pro aplikaci RTM na daném přípravku
4. Oživte a ověřte: komunikační protokol, registraci proměnných, tabulkové a grafické sledování promměnných a další vybrané funkce
5. Ověřte funkčnost na vhodných příkladech

Seznam doporučené literatury:

- firemní dokumentace STM32F429 Discovery Kit , ST Microelectronics
- firemní dokumentace Keil microvision
- Mikulics S. Diplomová práce, CVUT v Praze, FEL, Praha 2018

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Karel Künzel, CSc., katedra elektrotechnologie FEL

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

Datum zadání diplomové práce: **03.09.2018** Termín odevzdání diplomové práce: **15.09.2018**

Platnost zadání diplomové práce: **30.09.2019**

Ing. Karel Künzel, CSc. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

.	
Datum převzetí zadání	Podpis studenta

Poděkování

Rád bych touto cestou poděkoval svému vedoucímu Ing. Karlu Künzelovi, CSc. za jeho cenné rady a odborné vedení při zpracovávání této diplomové práce.

Abstrakt

Real-Time-Monitor je diagnostický a vývojový nástroj, který pomáhá uživatelům získávat data z digitálních řídicích systémů. Skládá se z knihovny pro mikropočítače a aplikace pro PC.

V práci je popsána analýza zdrojového kódu knihovny a implementace potřebných změn pro její spuštění na mikropočítači *STM32F4*.

Abstract

Real-Time-Monitor is a diagnostic and development tool, that helps users with gathering data from control systems. It consists of microcomputer library and PC application.

This thesis describes analysis of the source code of the library and implementation of the necessary changes so it can be to run on the *STM32F4* microcomputer.

Klíčová slova

RTM protokol, řídicí systémy, získání dat, analýza dat, ARM, víceplatformní vývoj, C++, návrhové vzory, objektově orientované programování.

Keywords

RTM protocol, control systems, data gathering, data analysis, ARM, multi-platform development, C++, design patterns, object oriented programming.

Obsah

Seznam obrázků **XV**

1 Úvod **1**
1.1 Nástroj Real-Time-Monitor . . . 1
1.1.1 Knihovna RTM . . . 1
1.1.2 Aplikace RTM . . . 2
1.1.3 Komunikační protokol RTM . . . 3
1.2 Cíle práce . . . 4
1.3 Použitý hardware . . . 5
1.3.1 Původní hardware . . . 5
1.3.2 Vývojová deska STM32F429I-DISC1 Discovery . . . 5
1.4 Architektura víceplatformních aplikací . . . 6
1.4.1 HAL . . . 7
1.4.2 Ovladače . . . 7

2 Současný stav **9**
2.1 Analýza stávajícího zdrojového kódu . . . 9
2.1.1 Rozdělení zdrojového kódu do adresářů . . . 9
2.1.2 Zhodnocení přenositelnosti kódu knihovny RTM . . . 10
2.1.3 Součásti potřebné pro minimální funkčnost knihovny . . . 10
2.1.4 Možné přístupy k úpravě kódu pro přípravek STM32F429I-DISC1 . . . 11

3 Volba řešení **12**
3.1 Volba programovacího jazyka . . . 12
3.1.1 C11 . . . 12
3.1.2 C++14 . . . 12
3.1.3 Rust . . . 12
3.1.4 Zvolený programovací jazyk . . . 12
3.2 Volba vývojových nástrojů . . . 13
3.2.1 GNU Arm Embedded Toolchain . . . 13
3.2.2 Keil Microcontroller Development Kit . . . 14
3.2.3 Atollic TrueSTUDIO for STM32 . . . 14
3.2.4 Zvolené řešení . . . 14
3.3 Objektově orientované programování v C++ . . . 15
3.3.1 Koncepty jazyka C++ a odlišnosti od C . . . 15
3.3.2 Pilíře objektově orientovaného programování . . . 20
3.3.3 Návrhové vzory . . . 21
3.3.4 Dependency injection . . . 23

4 Realizace **26**
4.1 Stanovení rozsahu práce . . . 26
4.2 Úpravy pro úspěšnou kompilaci a nahrání do mikropočítače . . . 27
4.3 Vytvoření potřebných HAL a úpravy ovladačů . . . 28
4.3.1 HAL pro časovač . . . 28
4.3.2 HAL pro GPIO . . . 31
4.3.3 HAL pro sériovou komunikaci . . . 32
4.3.4 Ovladač sériové komunikace . . . 34
4.4 Minimální aplikace . . . 39
4.4.1 Inicializace . . . 39
4.4.2 Registrace vzorových proměnných . . . 40
4.4.3 Hlavní programová smyčka . . . 40
4.5 Použití knihovny RTM ve vlastním projektu . . . 41
4.5.1 Adresářová struktura . . . 41

5 Ověření funkčnosti **42**
5.1 Testování úspěšného navázání spojení . . . 42
5.2 Ověření získání dat do tabulky . . . 44
5.3 Ověření vykreslení dat do grafu . . . 45
5.4 Ověření vykreslení dat do grafu v off-line režimu . . . 47

6 Závěr **48**
6.1 Seznámení s RTM a s přípravkem STM32F429I-DISC1 . . . 48
6.2 Návrh potřebné abstrakční vrstvy a úpravy zdrojového kódu . . . 48
6.3 Ověření funkčnosti . . . 48
6.4 Celkové zhodnocení . . . 48

Odkazy **49**

Přílohy **51**
Příloha A: Obsah přiloženého CD . . . 51

Seznam obrázků

1.1 Úvodní obrazovka aplikace RTM. 3
1.2 Tabulka proměnných v aplikaci RTM. 4
1.3 Kartézský graf v aplikaci RTM. 5
1.4 Vývojová deska STM32F429I-DISC1 firmy STMicroelectronics. 6
1.5 Aplikace pro dvě platformy bez abstrakční vrstvy. 8
1.6 Aplikace pro dvě platformy s abstrakční vrstvou. 8
1.7 Aplikace pro dvě platformy s HAL a ovladačem. 8

2.1 Duplikace zdrojového kódu ovladačů. 10
2.2 Možná architektura sériové komunikace bez duplikace kódu. 11

3.1 Dědičnost tříd. 21
3.2 Návrhový vzor singleton. 22
3.3 Návrhový vzor observer. 23
3.4 Vztah třídy *Logger* s třídami *ConsoleWriter* a *FileWriter*. 24
3.5 Vytvoření instance třídy *Logger*. 25
3.6 Zápis události do konzole prostřednictvím *ConsoleWriter*. 25

4.1 UML diagram třídy *Timer*. 29
4.2 Vývojový diagram metody *tick()*. 30
4.3 UML diagram tříd pro *GPIO*. 36
4.4 UML diagram tříd pro *uart*. 37
4.5 UML diagram ovladače a HAL pro sériovou komunikaci. 38

5.1 Volba COM portu, ke kterému je připojené zařízení s knihovnou RTM. . . . 42
5.2 Seznam zařízení připojených ke zvolenému portu. 43
5.3 Hlavní obrazovka aplikace RTM po připojení k zařízení RTM. 43
5.4 Tabulkové zobrazení před přidáním proměnných. 44
5.5 Dialogové okno pro přidání proměnných do tabulky. 44
5.6 Tabulkové zobrazení se třemi proměnnými a aktivní aktualizací dat. 45
5.7 Záložka pro zobrazení grafu bez přidaných proměnných. 46
5.8 Zobrazení časového průběhu sledované proměnné. 46
5.9 Zobrazení časového průběhu sledované proměnné v offline režimu. 47

Kapitola 1

Úvod

V této diplomové práci se zabývám úpravou knihovny „Real-Time-Monitor“. Tato knihovna byla napsaná v jazyce C pro mikropočítač *TMS320F28335* firmy *Texas Instruments*; v souladu se zadáním této práce jsem jí upravil pro běh na mikropočítači *STM32F4* od firmy *STMicroelectronics*. Popisuji zde nejprve vhodnou architekturu a doporučeními pro psaní softwaru určeného k běhu na více platformách, a poté i konkrétní implementaci a provedené změny při úpravách knihovny *Real-Time-Monitor* pro běh na mikropočítači *STM32F4*. Kromě změn, které byly nutné pro fungování programu, jsem provedl i takové změny, aby budoucí úpravy pro jiné mikropočítače byly jednodušší.

1.1 Nástroj Real-Time-Monitor

Nástroj *Real-Time-Monitor* (dále označovaný jako *RTM*) je diagnostický a servisní nástroj pro digitální řídicí systémy. Umožňuje nejen získávání hodnot proměnných z těchto systémů a jejich následnou analýzu, ale i zápis dat do řídicího systému. Dále umožňuje přenos firmware do řídicího systému, a pokud cílový systém obsahuje hradlové pole, tak i jeho konfiguraci. Použití těchto funkcí lze omezit pomocí hesla. [1]

Svou koncepcí RTM nahrazuje, případně doplňuje, více různých nástrojů. Během vývoje může nahradit, případně doplnit ladicí nástroje dodávané ke konkrétnímu mikropočítači. Dále může nahradit nástroje jako osciloskopy a data loggery. Vzhledem k možnosti omezení přístupu heslem, lze RTM použít nejen při samotném vývoji, ale i při následném servisu instalovaných zařízení.

RTM se skládá ze dvou částí: Z diagnostického softwaru (dále označovaný jako *aplikace RTM*), který je spuštěný na diagnostickém PC a z knihovny napsané v jazyce C (dále označované jako *knihovna RTM*), určené pro běh na sledovaném řídicím systému.Tyto dvě komponenty budou podrobněji popsány v následujících sekcích.[1]

1.1.1 Knihovna RTM

Knihovna RTM je psaná v jazyce C. Obsahuje vše potřebné pro zprostředkování přenosu hodnot numerických proměnných mezi mikropočítačem a sledovacím PC. Knihovna umožňuje sledování proměnných těchto datových typů jazyka C (definovaných ve standardních hlavičkových souborech *stdint.h* a *float.h*)[1, 2]:

- uint16_t reprezentující příznaky
- uint32_t reprezentující příznaky
- int16_t,
- uint16_t

- int32_t
- uint32_t
- float32_t

Proměnné je možné sledovat až po jejich předchozí registraci v RTM. Nelze tedy automaticky sledovat veškeré proměnné programu. Registrace probíhá voláním jedné ze specializovaných funkcí. Jednotlivé registrační funkce se mezi sebou liší jen datovým typem registrované proměnné. Mimo ukazatele na proměnnou k registraci mají všechny registrační funkce následující parametry:

- název proměnné,
- popis proměnné,
- název jednotky,
- škálování,
- minimální povolená hodnota,
- maximální povolená hodnota,
- nutná oprávnění pro úpravu hodnoty proměnné.

Žádný z těchto parametrů nemá vliv na použití proměnné v rámci programu řídicího systému. Jedná se pouze o informace pro aplikaci běžící ve sledovacím PC.

Data z řídicího systému jsou odesílána a přijímána pomocí sériové linky, nebo pomocí Ethernetu. [3]

1.1.2 Aplikace RTM

Aplikace RTM je grafická aplikace vytvořená v jazyce Java; v době psaní této práce je aplikace ve verzi *02.00.05beta*. Aplikace RTM tvoří rozhraní mezi uživatelem a řídicím systémem. Umožňuje jednak zobrazení okamžitých hodnot a časových průběhů registrovaných proměnných, tak i jejich nastavení. Hlavní okno aplikace je na obrázku 1.1 na straně 3.

Po spuštění aplikace má uživatel na výběr mezi čtyřmi režimy:

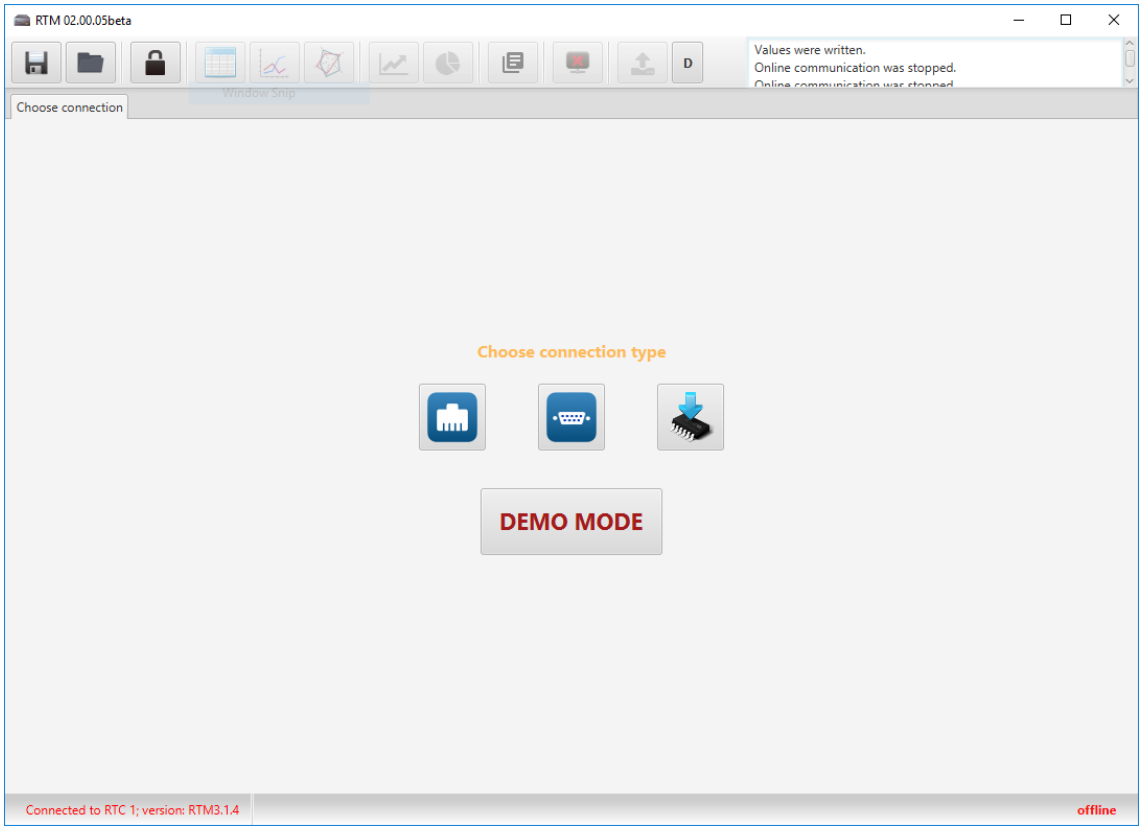
- USB/RS232,
- Ethernet,
- Service Bootloader,
- DEMO.

Režim *USB/RS232* lze použít, pokud je zařízení připojeno k PC pomocí sériové linky, nebo pomocí převodníku RS232 na USB. V současné době aplikace RTM umožňuje pouze použití převodníků firmy FTDI. Režim *Ethernet* lze použít u desek, které ho podporují, pro komunikaci prostřednictvím počítačové sítě. Režim *Service Bootloader* umožňuje nahrání nového firmware do řídicího systému prostřednictvím sériové linky. Poslední z nabízených režimů, režim *DEMO*, slouží k vyzkoušení aplikace RTM bez připojeného řídicího systému. K tomu používá vygenerovaná „vzorová“ data.

Po připojení k řídicímu systému aplikace dokáže zobrazovat hodnoty proměnných těmito způsoby:

- v tabulce,
- v kartézských souřadnicích,
- v polárních souřadnicích.

V tabulce se zobrazuje pouze aktuální hodnota (viz obrázek 1.2 na straně 4). V grafech se zobrazují hodnoty v závislosti na čase (viz obrázek 1.3 na straně 5). U zobrazení v kartézských i polárních souřadnicích se dále rozlišuje mezi *offline* a *online* režimy získávání dat. V režimu online jsou data přenášena nepřetržitě. V režimu offline se data ukládají do paměti v řídicím systému a z ní jsou periodicky načítána do aplikace RTM.



Obrázek 1.1: Úvodní obrazovka aplikace RTM umožňuje výběr mezi čtyřmi funkčními režimy.

1.1.3 Komunikační protokol RTM

Komunikační protokol je množina pravidel, která specifikuje, jak se posílají data mezi elektronickými zařízeními. Aby si mohly počítače mezi sebou vyměňovat data, musí existovat předchozí shoda o tom, jak budou informace strukturovány a jak je bude každá strana přijímat a odesílat. Protokoly se nemusejí týkat jen struktury dat, ale například i hardwarové implementace, napěťových úrovní a časování.

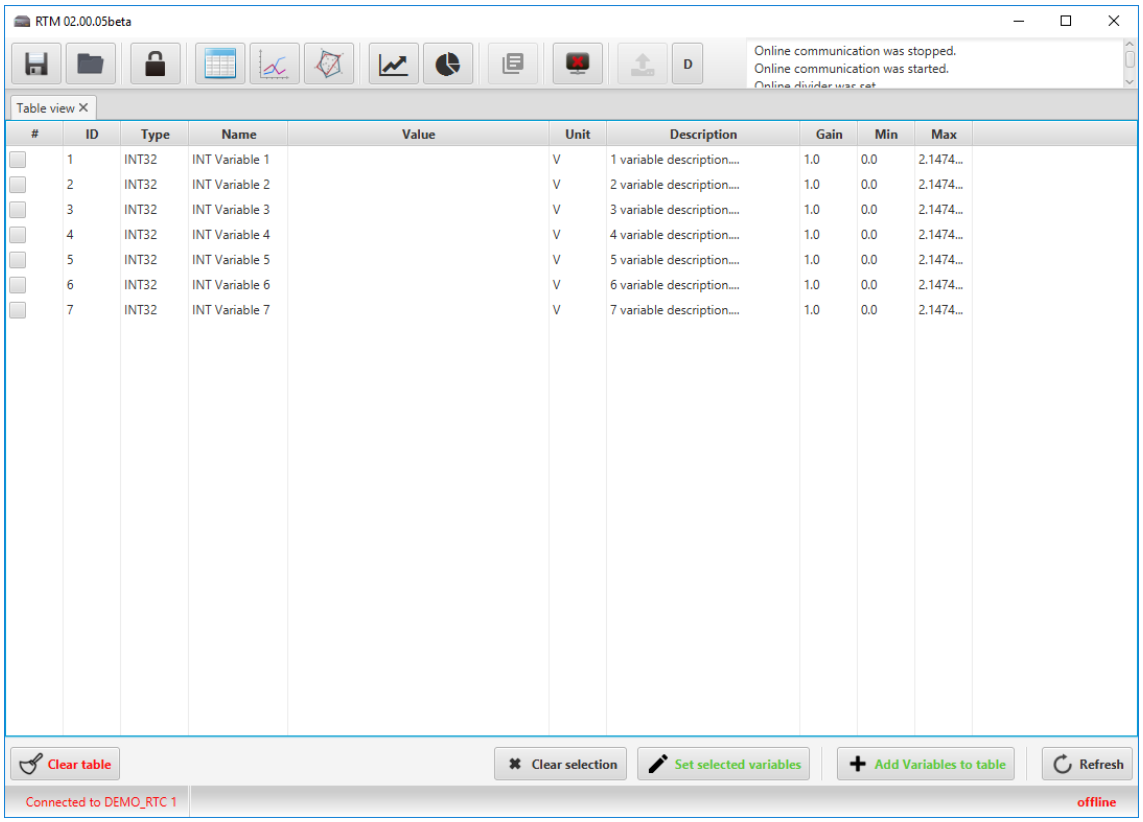
Tvar zprávy v protokolu RTM se podobá tvaru zprávy protokolu *TCP*. Protokol RTM je, stejně jako protokol TCP, stavový. Nejprve musí být vytvořeno spojení mezi oběma stranami. Teprve poté mohou být posílána data.

1.2 Cíle práce

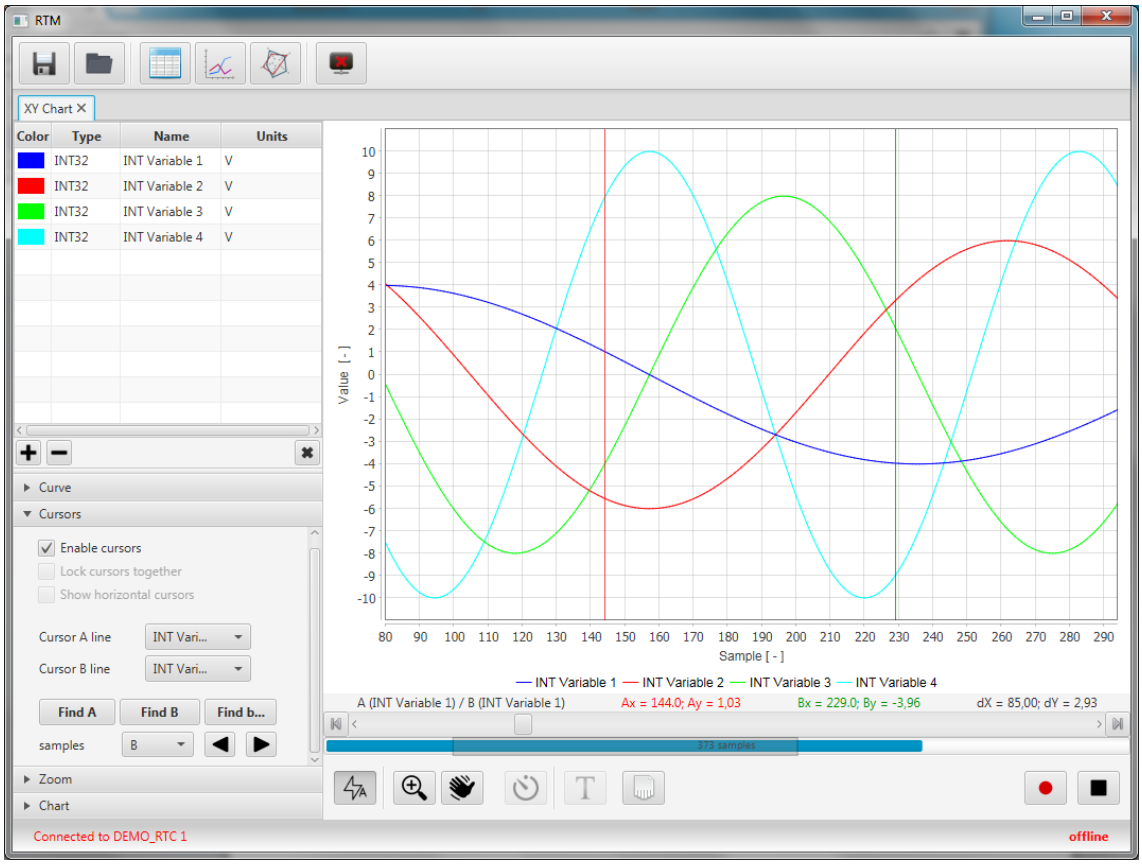
Cílem této práce bylo provést takové úpravy knihovny RTM, aby bylo možné ji použít na přípravku s procesorem *STM32F4*. Protože by nebylo možné z časových ani technických důvodů převést na nový procesor veškerou funkčnost knihovny, byly vybrány některé nejdůležitější funkce:

- registrace proměnných,
- komunikační protokol,
- přenos hodnot proměnných do aplikace RTM v PC a jejich zobrazení v grafech,
- tvorba vzorového projektu pro STM32F4 za účelem snadného nasazení knihovny a jejího dalšího použití.

Při úpravách knihovny jsem chtěl provést co nejméně zásahů do jejího vnitřního fungování. Nezbytné zásahy do knihovny by neměly být závislé na konkrétním mikropočítači a měly by být začlenitelné do původního zdrojového kódu. Toto opatření zamezí velkému odchýlení upraveného zdrojového kódu od původního, což by mohlo způsobit obtížné začleňování nových funkcí z původní knihovny.



Obrázek 1.2: Tabulka proměnných v aplikaci RTM zobrazuje pouze aktuální hodnoty.



Obrázek 1.3: Kartézský graf v aplikaci RTM umožňuje zobrazení závislosti sledovaných proměnných na čase.

1.3 Použitý hardware

1.3.1 Původní hardware

Knihovna RTM vznikala původně pro mikropočítače *Delfino* firmy *Texas Instruments*; konkrétně pro mikropočítač *TMS320F28335*. Tento procesor používá architekturu *C28x*, kterou vytvořila firma *Texas Instruments*, a která je uzpůsobená pro potřeby digitálního zpracování signálu. Tyto mikropočítače byly použité při vývoji vlastního hardware.

1.3.2 Vývojová deska STM32F429I-DISC1 Discovery

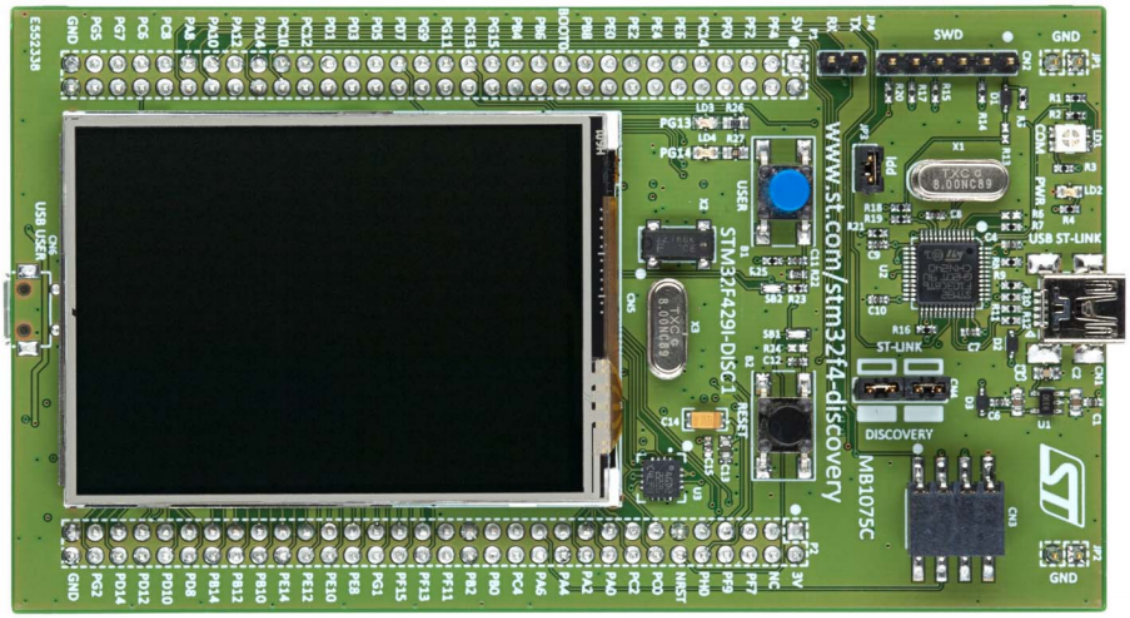
Vývojové desky se obvykle používají k vyhodnocení použitelnosti daného řešení (například konkrétního procesoru) pro zamýšlenou aplikaci; umožní tak ušetřit čas i peníze při vytváření vlastních prototypů, pokud by se řešení ukázalo jako nevhodné. Kromě procesoru na vývojových deskách bývají osazené i další periferie, které mohou být předmětem vyhodnocení.

Přípravek *STM32F429I-DISC1* je programovatelná vývojová deska od firmy STMicroelectronics. Obsahuje mikropočítač *STM32F429ZI* s jádrem *Arm Cortex-M4*. Kromě samotného mikropočítače deska obsahuje tyto periferie:

- ST-LINK/V2 ladicí nástroj,
- 2.4 palcový QVGA TFT LCD displej,
- externí 64 Mbit SDRAM paměť,

- ST MEMS gyroskop,
- USB OTG Micro-AB konektor,
- dvě uživatelské LED,
- uživatelské tlačítko.

Díky přítomnosti ladicích a programovacích nástrojů přímo na desce již není nutný žádný další specializovaný programátor. [4]



Obrázek 1.4: Vývojová deska STM32F429I-DISC1 firmy STMicroelectronics. [5]

1.4 Architektura víceplatformních aplikací

Při psaní softwaru se často setkáváme s požadavkem, aby ho bylo možné spustit na několika různých platformách. Může se jednat o běh na různých operačních systémech, různých procesorových architekturách a o práci s různými periferiemi. S požadavkem na běh na více platformách je dobré počítat již od prvotního návrhu architektury programu. Sice se dá pro běh na jiné platformě upravit i již existující program, ale v závislosti na jeho architektuře, to může být velice komplikované.

Naprostou nutností při psaní aplikací pro více platforem je svědomité oddělení kódu obsahujícího aplikační logiku od kódu, který je specifický pro konkrétní platformu. Díky tomu je při pozdějších úpravách programu pro běh na jiné platformě možné upravit pouze části kódu, které jsou pro danou platformu specifické, bez zásahu do samotné logiky programu. Při nedostatečném oddělení kódu pro danou platformu se dostaneme do situace zobrazené na obrázku 1.5 na straně 8; musíme udržovat verzi aplikace pro každou platformu, což způsobí duplikaci kódu, která může působit nemalé potíže. (Duplikace kódu se nazývá situace, kdy existuje kód řešící stejný problém ve více kopiích. To působí problémy při následných změnách logiky, protože je nutné upravit všechny kopie kódu. Tím se zvyšuje jednak časová náročnost, tak prostor pro chybu.) Duplikaci kódu se můžeme vyhnout tak, že vytvoříme abstrakční vrstvu mezi aplikační logikou a platformu, na které má aplikace běžet. Tato situace je zobrazena na obrázku 1.6 na straně 8. Tím poskytneme logické vrstvě jednotné rozhraní, které nezávisí na konkrétní platformě. Díky tomu můžeme použít stejný

kód aplikační logiky na všech platformách. Musíme však vytvořit odpovídající abstrakční vrstvy.

Abstrakční vrstva se u programování mikropočítačů skládá obvykle z *ovladače* a z *abstrakční vrstvy hardwaru* (dále nazývané „HAL“). Rozvrstvení software s využitím HAL a ovladače je zobrazené na obrázku 1.7 na straně 8.

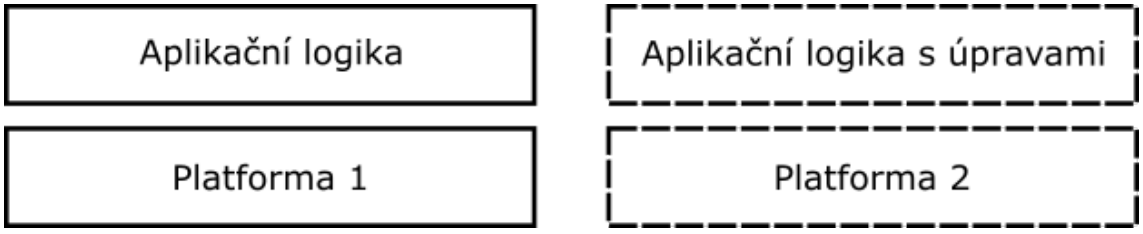
1.4.1 HAL

HAL je nejnižší abstrakční vrstva, která se používá při programování mikropočítačů. Mikropočítače obsahují různé periferie, které se mezi jednotlivými architekturami mohou výrazně odlišovat. Obvykle se nastavují pomocí zapisování do řídicích registrů. Každý mikropočítač může nabízet odlišné možnosti i způsoby nastavení jednotlivých periferií.

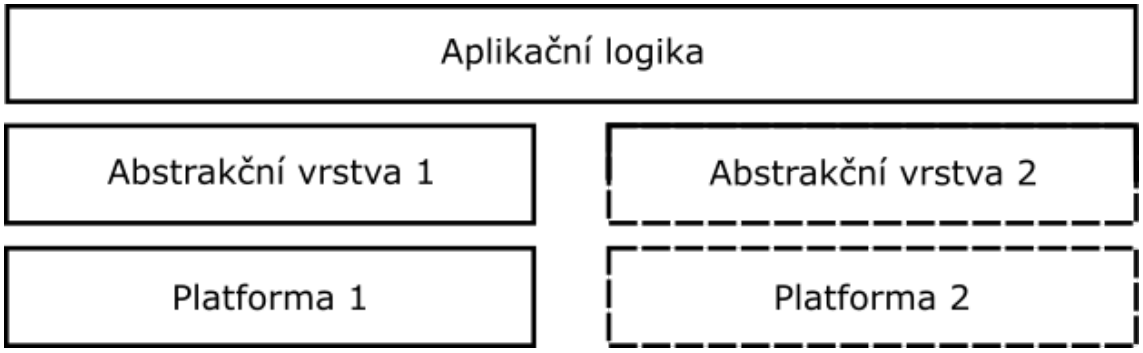
HAL slouží jako rozhraní mezi jednotlivými periferiemi a zbytkem programu. HAL ovládají jednotlivé periferie přímým zápisem do řídicích registrů. Jedná se o relativně jednoduchou abstrakční vrstvu, která by sama o sobě měla být „bezstavová“. Tím je myšleno to, že neuchovává žádné informace o stavu. HAL mohou být používané u jednoduchých periferií přímo uživatelským programem. Obvykle poskytují pouze základní funkce jako například inicializaci periferií, jejich nastavení a přístup k funkcím, které už periferie samotné nabízejí. U periferie sériové linky to může být například nastavení rychlosti přenosu, parity, polarity, počtu bitů v bajtu a odeslání nebo příjem jednoho bajtu. U složitějších periferií bývá mezi HAL a uživatelskou aplikací navíc *ovladač*.

1.4.2 Ovladače

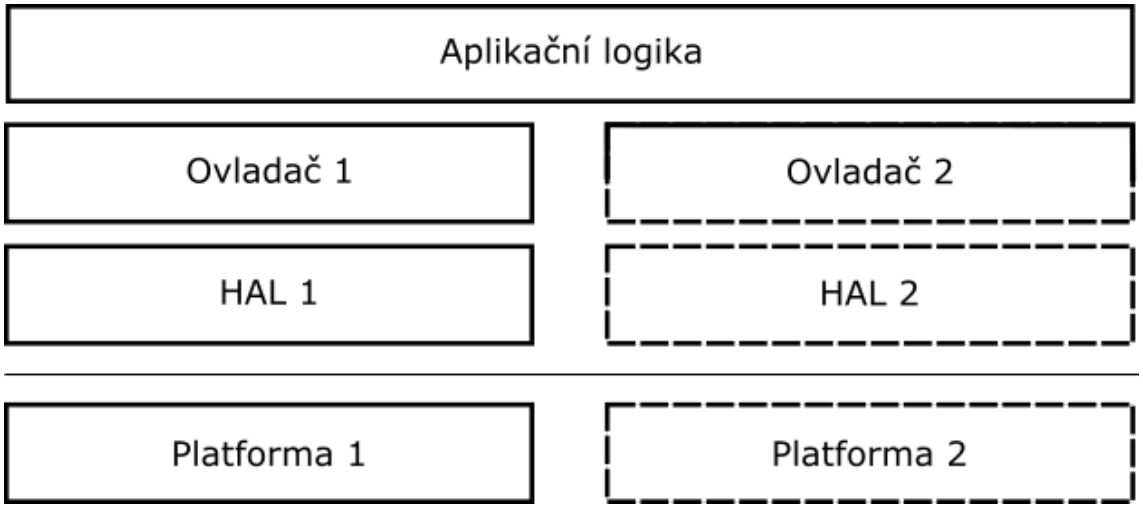
Ovladače tvoří druhou část abstrakční vrstvy mezi aplikací a hardwarem. Již nezapisují do registrů zařízení, místo toho využívají *HAL*. Obvykle implementují složitější funkcionalitu. Na rozdíl od HAL, ovladače většinou udržují i informace o stavu. U periferie sériové linky to bývá například odeslání nebo příjem celého řetězce bajtů.



Obrázek 1.5: Software pro dvě platformy bez použití abstrakce vyžaduje úpravu samotné aplikační logiky. To způsobuje duplikaci kódu.



Obrázek 1.6: Aplikační logika programu s abstrakční vrstvou může zůstat stejná bez ohledu na platformu.



Obrázek 1.7: Aplikační logika programu s abstrakční vrstvou může zůstat stejná bez ohledu na platformu. HAL odstraňuje nutnost zápisu a čtení registrů pro základní funkcionalitu periferie. Ovladač doplňuje složitější funkcionalitu.

Kapitola 2

Současný stav

2.1 Analýza stávajícího zdrojového kódu

Před začátkem úprav zdrojového kódu pro *STM32F429I-DISC1* bylo nutné provést jejich analýzu a rozhodnout, která místa budou vyžadovat úpravy. V ideálním případě by měl vyžadovat úpravy pouze zdrojový kód HAL. K menším úpravám by mohlo dojít ve zdrojovém kódu ovladačů. Samotná logika programu by se však měla měnit jen v tom nejnutnějším případě.

Autoři knihovny RTM se snažili o takovou strukturu kódu, která umožní snadné úpravy pro jiný mikropočítač. Použili k tomu dvě abstrakční vrstvy, *HAL* a *ovladač*, jak je uvedeno na obrázku 1.7 na straně 8.

2.1.1 Rozdělení zdrojového kódu do adresářů

communication

Adresář *communication* obsahuje většinu aplikační logiky samotného RTM. Jeden z nejdůležitějších modulů je *vobs_ observer_ of_ variables*. Ten zodpovídá za registraci jednotlivých proměnných a za sledování a nastavování jejich hodnot. Probíhá zde i kontrola, jestli uživatel má oprávnění pro práci s danou proměnnou.

examples

Adresář *examples* obsahuje pouze soubor *exp_ var.c*. Ten obsahuje vzorový program pro knihovnu RTM. Deklaruje několik různých proměnných, které registruje v knihovně RTM pro přenos do PC. Mimo jiné je možné měnit frekvenci blikání LED.

Tento vzorový program jsem využíval při začátku analýzy a při rozhodování, které části kódu jsou nezbytně nutné pro zprovoznění odesílání a přijímání dat.

hal_335

Tento adresář obsahuje první část abstrakční vrstvy - *HAL*, neboli abstrakční vrstvu hardwaru. Ta tvoří rozhraní pro zbytek programu k jednotlivým periferiím mikroprocesoru. Zbytek programu tak při používání periferií nemusí přistupovat přímo ke konfiguračním registrům.

Většina HALů z tohoto adresáře není nutná pro základní funkcionalitu knihovny. Vyskytují se zde HAL pro periferie, které STM32F429I-DISC1 nemá a HAL pro periferie, které nejsou nutné pro fungování knihovny RTM. Nejdůležitější HAL v tomto adresáři jsou *hal_ gpio*, *hal_ tmr* a *hal_ scia* (případně *hal_ scib*).

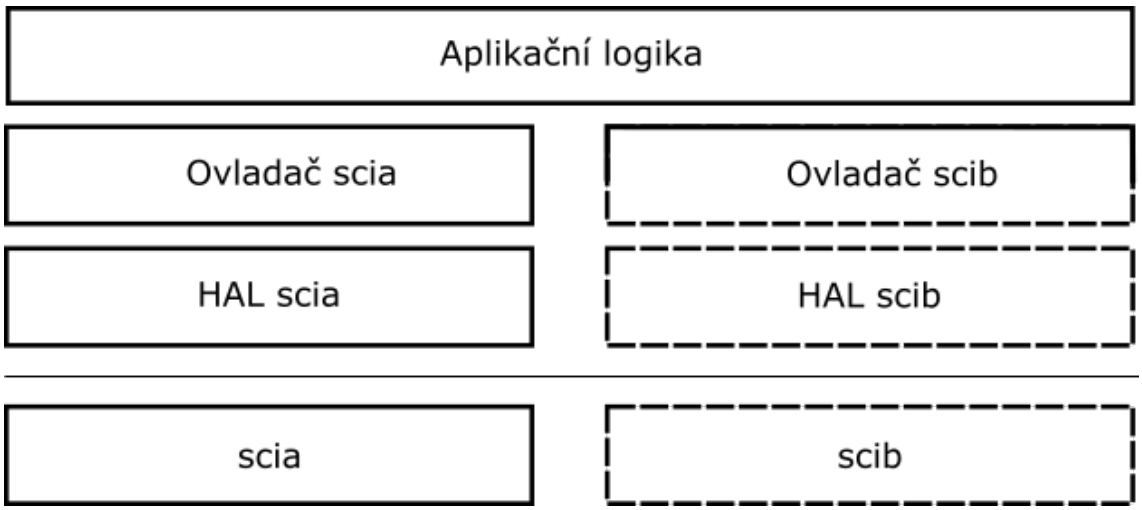
drv

Tento adresář obsahuje druhou část abstrakční vrstvy - *ovladače*. U složitějších periferií, jako je například UART, obvykle program nepoužívá periferie přímo prostřednictvím *HAL*, ale prostřednictvím ovladače. Ty k základní funkcionalitě periferie obvykle přidávají další možnosti, jako například implementaci konkrétních protokolů a dalších sofistikovaných funkcí. Nejdůležitějším ovladačem pro funkci RTM je ovladač *drv_scia* (případně *drv_scib*).

2.1.2 Zhodnocení přenositelnosti kódu knihovny RTM

Při psaní přenositelného kódu je nutné striktně oddělovat aplikační logiku od částí kódu, které jsou specifické pro konkrétní platformu. Při psaní knihovny RTM byla použita řada technik pro psaní přenositelného kódu, v některých případech však bohužel do zbytku kódu „prosakují“ implementační detaily dané platformy. Nestačí tak upravit pouze HAL, ale bude nutná i úprava ovladačů. Implementační detaily platformy navíc nejsou patrné jen uvnitř zdrojového kódu, ale i v organizaci souborů projektu.

Například ovladače pro sériovou komunikaci mají svůj název odvozený od periferie v mikropočítači *TMS320F28335* i přesto, že neobsahují žádný kód, který by pro tento počítač byl specifický (viz obrázek 2.1 na straně 10). Zdrojový kód ovladačů pro *scia* i



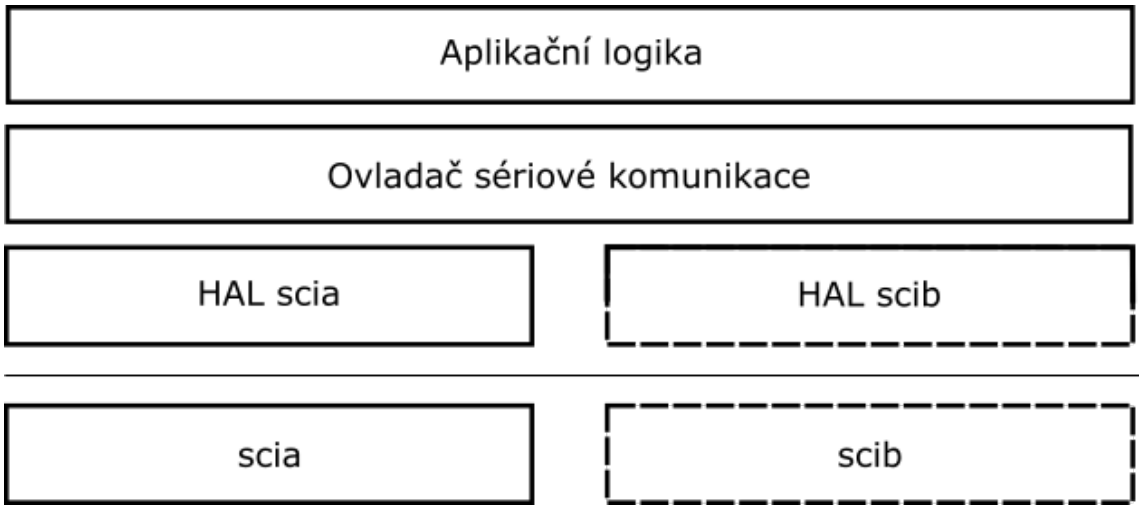
Obrázek 2.1: Duplikace zdrojového kódu ovladačů je způsobená „prosakováním“ detailů o konkrétní platformě do zdrojového kódu, který by měl být obecný.

pro *scib* se liší pouze použitým HAL. Dochází tak ke zbytečné duplikaci kódu a jakákoliv změna logiky provedená v jednom z těchto souborů musí být provedena ručně i v souboru druhém. Udržovatelnost kódu klesá s rostoucím množstvím duplicitního kódu. Razantně se zvyšuje možnost chyby, kdy programátor neprovede změny na všech instancích kódu. Lepší řešení by bylo mít pouze jeden univerzální ovladač pro sériovou komunikaci, u kterého lze nastavit který HAL má použít pro komunikaci (viz obrázek 2.2 na straně 10)

2.1.3 Součásti potřebné pro minimální funkčnost knihovny

Základní funkcí knihovny je schopnost registrace proměnných a odesílání a příjem jejich hodnot prostřednictvím USB nebo Ethernetu. Po analýze vzorové ho nasazení knihovny v souboru *exp_var.c* jsem stanovil následující minimální HAL a ovladače pro správnou funkci knihovny:

- *hal_tmr* - umožňuje použití časovačů,



Obrázek 2.2: Možná architektura sériové komunikace bez duplikace zdrojového kódu ovladače. Jednotlivé HAL poskytují přístup k periferiím pro sériovou komunikaci prostřednictvím jednotného rozhraní.

- *hal_scia*, nebo *hal_scib*,
- *drv_scia*, nebo *drv_scib* - společně s *hal_scia(b)* umožňuje odesílání a příjem dat přes sériovou linku.

2.1.4 Možné přístupy k úpravě kódu pro přípravek STM32F429I-DISC1

2.1.4.1 Zachování původní struktury zdrojového kódu

Jednou možností by bylo zachování co možná největšího množství zdrojového kódu i struktury projektu. Veškeré názvy souborů i názvy funkcí by zůstaly stejné. Lišily by se pouze jejich implementace.

HAL pro časovače by používal jeden z časovačů přípravku *STM32F429I-DISC1*. HAL pro *scia* a *scib* ve verzi pro mikropočítač *TMS320F28335* používají periferie *scia* a *scib*. Ve verzi pro přípravek *STM32F429I-DISC1* by používaly některé dvě z nabízených periferií pro sériovou komunikaci.

Použití HAL i ovladačů ze zbytku programu by zůstalo beze změny.

2.1.4.2 Vytvoření nové struktury zdrojového kódu

Druhou možností je zásadnější přepsání zdrojového kódu tak, aby byl univerzálnější a aby bylo snazší ho upravit pro další platformy. HAL by měly být psány co nejuniverzálněji, bez duplikace kódu. Kód by měl být co nejméně závislý na konkrétní platformě. Bylo by tak možné dosáhnout stavu na obrázku 2.2.

2.1.4.3 Zvolené řešení

Obě řešení mají svoje pozitiva i negativa. Hlavní výhodou prvního řešení je kompatibilita s existujícím kódem. Bylo by možné přenést existující programy využívající knihovnu RTM na jiné mikropočítače bez nutnosti jejich úprav. Nevýhodou však je, že se ještě více upevní neoptimální praktiky v kódu a v budoucnu se již s velmi vysokou pravděpodobností nezmění.

Výhodou druhé varianty je uvedení programu do stavu, že jeho další úprava pro jiný mikropočítač bude o mnoho jednodušší. Nevýhodou však je nekompatibilita s existujícími programy využívajícími knihovnu RTM, které by bylo nutné také upravit.

Kapitola 3

Volba řešení

3.1 Volba programovacího jazyka

Pro mikropočítače s architekturou *ARM* připadají v úvahu jazyky *C* ,*C++* nebo *Rust*. Každý z těchto jazyků má své výhody.

3.1.1 C11

C je osvědčený programovací jazyk pro mikropočítače. Byl použitý i při vývoji knihovny RTM. Jeho hlavní výhodou je, že má poměrně jednoduchou specifikaci, a proto pro něj existuje kompilátor pro prakticky všechny mikropočítače. Mezi nevýhody jazyka C patří slabý typový systém a časté využívání maker preprocesoru. To může mít za následek chyby, které se projeví až při běhu programu.

3.1.2 C++14

Jazyk C++ původně vznikal jako rozšíření jazyka C. Postupně se od něj odchýlil a i přes to, že oba jazyky mají velice podobnou syntaxi se jedná o dva oddělené jazyky. Oproti C nabízí C++ silný typový systém a objektově orientované programování.

Ulehčuje psaní přenositelných programů tím, že umožňuje snadné vytváření robustních abstrakcí. Ty zlepšují organizaci kódu a usnadňují oddělení jednotlivých logických celků. Tyto abstrakce přitom často nemají žádný negativní vliv na rychlost programu, protože jsou odstraněny při kompilaci.

Nevýhodou je složitější specifikace jazyka, což má za následek menší rozšíření kompilátorů jazyka C++. U moderních mikropočítačů to však zpravidla nebývá problém.

3.1.3 Rust

Jazyk *Rust* je relativně nový jazyk z roku 2010. Stejně jako C++ Rust podporuje objektově orientované programování. Zaměřuje se především na bezpečnost a to hlavně při vícevláknovém programování. Většině běžných chyb, které vznikají ve vícevláknových programech, se snaží zabránit již ve fázi kompilace.

Nevýhodou je jeho malé rozšíření způsobené tím, o jak mladý jazyk se jedná. Pro mikropočítače ARM však je k dispozici kompilátor.

3.1.4 Zvolený programovací jazyk

Po zvážení všech možností popsaných v předchozí sekci jsem se rozhodl pro jazyk C++14. Jedná se o verzi jazyka C++ z roku 2014. Oproti C má nesporné výhody, které usnadňují

psaní čitelného a přenositelného kódu. Je přitom už dostatečně vyspělý a existují kompilátory C++ pro moderní procesory. Rust jsem nevybral, protože zatím není při programování mikropočítačů příliš používaný. Navíc s ním mám nejméně zkušeností ze všech tří jazyků.

3.2 Volba vývojových nástrojů

Při vývoji jakéhokoliv programu je potřeba mít k dispozici minimálně *kompilátor* a *linker*. *Kompilátor* dokáže převést zdrojový soubor ve vstupním programovacím jazyce do jazyka výstupního - obvykle do objektového kódu. (Objektový kód je v podstatě již strojový kód cílové platformy, obsahuje však dodatečné informace umožňující přemístění návěstí a proměnných a informace o vazbách na externí proměnné a externí kód.) Výstupem kompilace jsou objektové soubory odpovídající souborům zdrojového kódu. Tyto soubory potom *Linker* složí do výsledného spustitelného souboru. Další program, který sice není nutný, ale je výhodné ho mít k dispozici, je *debugger*. Ten slouží k ladění programu. Mezi obvyklé funkce debuggeru patří krokování programu a nahlížení do proměnných. Je neocenitelnou pomůckou při hledání chyby v programu.

Pokud bychom měli k dispozici pouze *kompilátor* a *linker*, museli bychom při jakýchkoliv úpravách zdrojového kódu buď kompilovat všechny soubory, nebo ručně kompilovat ty, které byly změněny. To může být při větší velikosti projektu velice časově náročné. Práci nám může ulehčit program *make*. Ten dokáže, mimo jiné, sledovat čas poslední úpravy zdrojového kódu a objektových souborů. Pokud je objektový soubor starší, než příslušný zdrojový kód, program *make* sám spustí kompilaci tohoto souboru. Po zkompilování všech souborů, ve kterých došlo ke změnám automaticky spustí linkování.

Při vývoji programů pro mikropočítače je navíc potřeba software a hardware, který dokáže vytvořený program do mikropočítače nahrát. V případě přípravku *STM32F429I-DISC1* se jedná o software i hardware s názvem *ST-LINK*. Ten kromě možnosti nahrání programu umožňuje i ladění (debugování) programu přímo v cílovém mikropočítači. Hardwarová část programátoru *ST-LINK* je přítomna přímo na vývojové desce. [4]

Každý z těchto programů má při tvorbě software své místo. Jeden bez druhého nejsou příliš užitečné. Dohromady se obvykle označují jako „toolchain“. Stále se však jedná a skupinu diskrétních nástrojů.

3.2.1 GNU Arm Embedded Toolchain

GNU Arm Embeded Toolchain je sada vývojových nástrojů pro procesory *Arm Cortex-M* a *Arm Cortex-R* založená na *GNU open source tools* od organizace *Free Software Foundation*. Sada obsahuje tyto nástroje:

1. *GNU C/C++* kompilátor a linker,
2. sadu nástrojů pro práci s binárními soubory *Binutils*,
3. debugger *GDB*,
4. standardní knihovnu jazyka C pro vestavěné systémy *newlib*.

Tato sada, doplněná programy *make* pro správu kompilace a *ST-LINK* pro nahrání programu do mikropočítače, obsahuje vše potřebné pro vývoj pro mikroprocesory *ARM*. [6]

Výhodou použití *GNU Arm Embedded Toolchain* je minimální závislost na proprietárním software. Open-source vývojové nástroje *GNU* od *Free Software Foundation*, na kterých je tato sada založená, jsou již časem prověřené a stabilní řešení. Protože se jedná o open-source řešení, jsou tyto nástroje zdarma bez jakýchkoliv omezení. Díky tomu je možné začínat nové projekty s jejich použitím bez jakýchkoliv finančních nákladů a s dobrou jistotou toho, že v blízké době neskončí jejich podpora.

Za tyto výhody zaplatíme nižší uživatelskou přívětivostí. Všechny programy ze sady GNU Arm Embedded Toolchain mají konzolové uživatelské rozhraní. Před tím než je možné začít psát program, je nutné nakonfigurovat vývojové prostředí pro svůj projekt. Minimálně se jedná o vytvoření takzvaného *Makefile*; to je konfigurační soubor pro program *make*. V něm je potřeba nastavit různé parametry jako použitý kompilátor, linker, jejich příznaky, kde se nacházejí zdrojové soubory a co s nimi udělat. Dále je odsud možné vyvolávat jiné aplikace. Lze tedy připravit takový *Makefile*, že program *make* při spuštění automaticky zkompiluje projekt a nahraje ho do cílového mikropočítače. Kvalitně napsaná konfigurace pro *make* je rozhodující pro rychlý a pohodlný vývoj.

3.2.2 Keil Microcontroller Development Kit

Keil Microcontroller Development Kit (dále jen Keil MDK) je sada vývojových nástrojů od firmy *ARM*. Součástí této sady je i *integrované vývojové prostředí Keil µVision*. Integrované vývojové prostředí (dále jen IDE) je program, který poskytuje v jedné aplikaci textový editor a grafické rozhraní pro různé vývojové nástroje jako kompilátor, debugger a program make. Úkolem IDE je odstínit uživatele od vnitřního fungování jednotlivých nástrojů a vazeb mezi nimi.

Kromě IDE a standardní vývojové sady Keil MDK nabízí softwarové balíčky a knihovny, které usnadní psaní vlastních aplikací. Jsou mezi nimi například knihovny pro grafická rozhraní, USB Device, USB Host a pro komunikaci pomocí internetového protokolu. [7]

Vytvoření nového projektu s pomocí IDE bývá výrazně jednodušší a rychlejší, než při použití sady vývojových nástrojů přímo. Z části je to dané tím, že vývojová prostředí obvykle obsahují vzorové konfigurace pro různé scénáře. Dále je to proto, že je velice jednoduché konfiguraci dále upravovat. I další úkoly, jako například ladění programu, bývají uživatelsky přívětivější prostřednictvím IDE. [7]

Keil MDK je komerční aplikace od firmy ARM. Existuje ve čtyřech verzích, které se liší množstvím funkcí a v souladu s tím i cenou. Ve verzi *MDK-Lite* je možné vývojové prostředí používat zdarma. Velikost zdrojového kódu je však omezena na 32 kB a obsahuje pouze podmnožinu dostupných knihoven. [7]

3.2.3 Atollic TrueSTUDIO for STM32

Atollic TrueSTUDIO je sada vývojových nástrojů pro procesory řady STM32. Stejně jako u Keil MDK je jeho součástí i IDE, které je v případě programu TrueSTUDIO založené na vývojovém prostředí Eclipse. Svými funkcemi se vývojovému prostředí *Keil MDK* velice podobá. [8]

Výhodou prostředí TrueSTUDIO oproti prostředí Keil MDK je, že ho firma STMicroelectronics nabízí zdarma bez jakéhokoliv omezení. Podobného výsledku by se dalo dosáhnout použitím vývojového prostředí *Eclipse* a ruční konfigurací jeho napojení na *GNU Arm Embedded Toolchain* a *ST-LINK*. Výhodou takového řešení by byla naprostá nezávislost na proprietárním software. Nevýhodou naopak o něco menší stupeň integrace všech nástrojů.

3.2.4 Zvolené řešení

Vzhledem k časové náročnosti použití samostatných nástrojů *GNU Arm Embedded Toolchain*, nebo jejich případné integraci do vývojového prostředí *Eclipse* a vzhledem k omezením neplacené verze prostředí *Keil MDK*, jsem se nakonec rozhodl pro *Atollic TrueSTUDIO*. To mi umožnilo rychle a jednoduše začít vyvíjet kód a testovat ho přímo na přípravku STM32F429I-DISC1.

3.3 Objektově orientované programování v C++

Objektově orientované programování je programovací paradigma založené na konceptu objektů. *Objekty* v jazyce C++, podobně jako *struktury* v jazyce C, mohou obsahovat data. Oproti strukturám jazyku C však objekty mohou obsahovat procedury, v terminologii objektově orientovaného programování zvané *metody*. Metody objektu mohou přistupovat k datům objektu. Objektově orientované programy se skládají ze vzájemně interagujících objektů.

V jazyce C++ jsou objekty založeny na takzvaných *třídách*. Předtím než může být objekt vytvořen, musí existovat třída na které bude založený. Třída slouží jako předloha pro objekt a určuje, jaká má objekt datové pole a jaké má metody. Třída, které je objekt instancí, určuje jeho *datový typ*. [9]

3.3.1 Koncepty jazyka C++ a odlišnosti od C

3.3.1.1 Klíčové slovo *static*

Toto klíčové slovo se v C++ používá k označení metod a členských proměnných tříd, které jsou společné pro všechny instance třídy a existují po celou dobu běhu programu. Jedna z častých aplikací statických proměnných a metod je implementace návrhového vzoru *singleton*. Ten popisuji dále v této práci. [9]

3.3.1.2 Klíčové slovo *constexpr*

Toto klíčové slovo může být použito k označení objektů i funkcí. Vyjadřuje, že definovaný objekt, nebo funkce, může být použit v takzvaném *konstantním výrazu*. Konstantní výraz je takový výraz, jehož hodnota je známa již při kompilaci. Jejich použití je stejné, jako obyčejných proměnných. V kódu lze deklarovat například takto:

```
constexpr int cislo = 5;
```

Oproti tradičnímu způsobu definování konstant pomocí *#define* známého z C a starších verzí C++ nabízí použití *constexpr* mnoho výhod. Nevýhodou použití maker preprocesoru je to, že se jedná o pouhé textové dosazení a nerespektují typy, ani obory platnosti. Oproti tomu proměnná deklarovaná jako *constexpr* se chová stejně, jako kterákoliv jiná proměnná jazyka C++. Jediným rozdílem od běžné proměnné je, že její výskyty může kompilátor během kompilace nahradit její hodnotou.

Další výhodou je, že *constexpr* nemusí být pouze konstanta, ale může to být výraz v jazyce C++, který je během kompilace vyhodnocen. Tento výraz může obsahovat i volání funkcí, které jsou označeny jako *constexpr*. [9, 10]

3.3.1.3 Přetěžování funkcí

Přetěžování funkcí umožňuje definování více funkcí se stejným názvem. O tom, která z variant funkce se použije, je rozhodnuto během kompilace na základě typu a počtu argumentů. [9]

3.3.1.4 Přetěžování operátorů

V *C++* se operátory chovají jako běžné funkce. Lze je tedy rovněž přetěžovat. Funkce definující operátor pro daný typ lze deklarovat například takto:

```
T operator+(T levy, T pravy);
```

V tomto příkladu je deklarována funkce, která sčítá dva operandy typu T. Při použití operátoru + s proměnnými typu T bude zavolána uvedená funkce. [9]

3.3.1.5 Šablonové metaprogramování

Šablonové metaprogramování je technika, při které kompilátor při kompilaci generuje z šablon dočasný zdrojový kód, který je sloučen s kompilovaným zdrojovým kódem. Hojně se této možnosti využívá při vytváření datových struktur. Lze definovat šablony buď pro třídy, nebo pro funkce. Funkce a třídy, které jsou definované šablonou, nazýváme generické. Generickou třídu můžeme definovat například takto:

```
template<typename T>
class GenericClass{
   T value;
public:
   GenericClass() = delete;
   GenericClass(T init) :
      value(init){}
   T getValue(){
      return value;
   }
}
```

Tato šablona má jeden parametr - *T*. Tento parametr je typu *typename*; uchovává název datového typu. Třída definovaná touto šablonou má jedno datové pole *value*, a funkci *getValue()*, která slouží k jejímu získání. Hodnota *value* je nastavena při vytváření instance jako parametr konstruktoru.

Takto vytvořená šablona umožňuje vytvořit instanci třídy *GenericClass* pro uchování hodnoty jakéhokoliv typu. Konkrétní typ je dosazen za *T* při vytváření instance šablony. Takto vytvořenou šablonu lze využít následovně:

```
GenericClass<int> celeCislo(1);
GenericClass<std::string> text("ulozeny text");
```

V tomto příkladu jsou vytvořeny dvě instance šablony třídy *GenericClass*. Pro každou z nich je při kompilaci vytvořen samostatný zdrojový kód. Obdobným způsobem jako se vytváří šablony tříd, je možné vytvářet šablony funkcí. [9]

3.3.1.6 Generické datové struktury pomocí metaprogramování

Bez použití šablonového metaprogramování bychom měli dvě možnosti, jak vytvořit datovou strukturu:

1. Vytvoření zdrojového kódu v tolika podobných variantách, kolik chceme podporovat datových typů. Pro každý typ musí existovat varianta datové struktury.

2. Vytvoření univerzálního kódu, který používá ukazatele typu *void*. Tímto způsobem lze obejít typový systém.

Z hlediska bezpečnosti kódu je lepší první varianta. Její veliká nevýhoda je však to, že je potřeba vytvořit datovou strukturu pro každý typ znovu. Při zájmu o vkládání nového typu do struktury, by bylo nutné jí znovu vytvořit pro daný typ. Výhodou druhé varianty je, že vytvořený kód může být použit pro ukládání dat jakéhokoliv typu. Ukazatele na uložená data jsou však převedeny na typ *void*. Tím jsou ztraceny informace o typu; přijdeme tím o typovou bezpečnost.

Šablonové metaprogramování nám umožňuje vytvořit šablonu datové struktury. Následně je možné vytvořit instanci této šablony pro jakýkoliv datový typ. Třída vytvořená v minulé kapitole je příkladem jednoduché generické datové struktury, schopné uchovávat jeden záznam. Podobně, jak byla definovaná šablona této třídy, by bylo možné vytvořit reprezentaci jakékoliv složitější datové struktury.

Výhodou generických datových struktur je jejich silné typování. Instance šablon mají vždy pevný datový typ - stejně jako kdyby byly jednotlivé varianty vytvářeny ručně. Díky jakékoliv chyby v typech odhalíme už při kompilaci kódu. [9]

3.3.1.7 Specializace šablon v metaprogramování

V některých případech se můžeme setkat s tím, že pro některý konkrétní typ je potřeba použít odlišnou implementaci funkce nebo třídy. Tu můžeme vytvořit pomocí specializace šablon. Jako příklad uvádím specializaci třídy *GenericClass* z předchozích kapitol:

```
template<>
class GenericClass<int>{
   int value = 150;
public:
   GenericClass() = delete;
   GenericClass(int init) :
      value(init){}
   int getValue(){
      return value;
   }
   int getNegativeValue(){
      return -value;
   }
}
```

Výše uvedený kód definuje specializaci šablony pro typ *int*. Téměř se neliší od původní třídy. Na rozdíl od ní však proměnná *value* má výchozí hodnotu 150 a třída má navíc jednu metodu, která vrací hodnotu *-1 * value*.

Při vytvoření instance šablony třídy *GenericClass* s parametrem *int* bude použita automaticky tato specializace. [9]

3.3.1.8 Podmíněné povolení konkrétních šablon

Při psaní šablon funkcí se můžeme setkat s tím, že potřebujeme mít možnost vytvoření instance dané šablony něčím podmínit. K tomu slouží šablona struktury ze standardní knihovny:

```
template<bool B, typename T = void> struct enable_if
```

Tato šablona má dva argumenty; první je logická hodnota, druhá je datový typ. Pokud je logická hodnota rovna *true*, struktura enable_if obsahuje členskou proměnnou *type*, ve které je uložena hodnota druhého parametru šablony. Pokud je logická hodnota rovna *false*, struktura proměnnou *type* neobsahuje.

Pro podmíněné povolení konkrétních šablon se využívá struktura *enable_if* v kombinaci s faktem, že chyby kompilace, které jsou způsobené nevhodným vytvořením instance šablony, jsou ignorovány. Tato vlastnost se nazývá *SFINAE - Substitution Failure Is Not An Error*.

Příkladem může být následující šablona funkce:

```
template<typename T>
std::enable_if<true,T>::type add(T a, T b){
   return a+b;
}
```

Zde je deklarována funkce s návratovým typem *std::enable_if<true,T>::type*. Protože je logická hodnota *true*, bude návratový typ *T*. Kdyby byla logická hodnota *false*, nebyl by návratový typ definovaný. To by v běžném kódu způsobilo chybu kompilace, ale protože zde se jedná o vytvoření instance šablony, nedojde v souladu s *SFINAE* k chybě. Instance

šablony však nebude vytvořena.

Často je podmíněné povolení šablon funkcí využíváno k omezení datových typů, se kterými může šablona pracovat. Výše uvedená šablona funkce by šla upravit, aby povolovala pouze sčítání numerických typů:

```
template<typename T>
std::enable_if<std::is_integral<T> || std::is_floating_point<T>,T>::type add(T
    a, T b){
   return a+b;
}
```

Při volání této funkce s číselnými argumenty dojde k vytvoření příslušné instance šablony. Při jejím volání s jinými typy dojde k chybě, protože instance šablony nebude vytvořena. [9]

3.3.1.9 Výčtové typy *enum* a *enum class*

Výčtový typ *enum* slouží k přiřazení jmen k celočíselným konstantám. Využívá se především ke zlepšení čitelnosti zdrojového kódu, protože umožňuje použití smysluplných názvů místo číselných konstant. Klasický enum se v jazyce C++ chová stejně jako v jazyce C. Deklarovat *enum* a vytvořit proměnnou o jeho typu můžeme například takto:

```
enum Result {
   failed = 0,
   success = 1
};
enum Result vysledek = failed;
```

Velkou nevýhodou klasického *enum* je, že po takovéto deklaraci jsou v dané oblasti platnosti dvě nová jména: *failed* a *success*. To je problém, pokud potřebujeme deklarovat další *enum*, který by obsahuje stejná jména číselných konstant. To může nastat například při použití cizí knihovny, která tyto jména používá.

Odpovědí na tento problém je nový *enum class*, který byl do specifikace C++ přidán v roce 2011. Od klasického *enum* ho odlišují dva hlavní znaky:

1. Je silně typový bez implicitních konverzí na číselné typy,
2. Vytváří svojí vlastní oblast platnosti, do které patří jednotlivé jeho konstanty.

První bod přispívá ke snížení množství chyb v kódu. U obyčejného *enum* lze mezi sebou srovnávat dvě nesouvisející konstanty deklarované v různých enum. To u *class enum* není možné, bez explicitní konverze. Druhý bod řeší problém se shodami jmen, protože konstanty jsou vázané na daný *class enum*. Používají se následovně:

```
enum class Result{
   failed = 0,
   success = 1
};
Result vysledek = Result::failed;
```

Jak je vidět z ukázkového kódu, je vždy nutné specifikovat výčtový typ, kterého se konstanty týkají. [9, 10]

3.3.1.10 Reprezentace příznaků pomocí *enum class*

Silné typování, veliká výhoda *enum class*, může být v některých případech naopak překážkou. Protože se *enum class* na rozdíl od běžného *enum* implicitně nepřevádí na číselné typy, nelze proměnné typu *enum class* použít jako operandy číselných operací jako například *bitový součin* nebo *bitový součet*. Následující kód je tedy neplatný:

```
enum class FileAccess{
   read = 0,
   write = 1
};

FileAccess access = read | write;
```

U obyčejného *enum* by došlo automaticky ke konverzi operandů na číselnou hodnotu, bitový součet těchto číselných hodnot a k následné konverzi výsledku zpět na původní typ.

Při použití *enum class* je nutné explicitně převést operandy na číselný typ a výsledek převést zpět na původní typ. Validní kód by mohl vypadat následovně:

```
FileAccess access = static_cast<FileAccess>(
   static_cast<int>(read) | static_cast<int>(write)
);
```

Dalším problémem by mohlo být, že *enum class* nemusí nutně reprezentovat pojmenovanou hodnotu typu *int*, ale například by mohlo na pozadí *enum class* být *short*. Naštěstí v *C++* existuje způsob, jak zjistit, který typ je na pozadí konkrétního *enum class*. Kód by šlo napsat lépe takto:

```
using number_type = typename std::underlying_type<FileAccess>::type;
FileAccess access = static_cast<FileAccess>(
   static_cast<number_type>(read) | static_cast<number_type>(write)
);
```

Tento zápis bitové operace však je však nepřiměřeně dlouhý na to, abychom ho používali pokaždé. Místo toho můžeme přetížit operátor bitového součtu tak, aby potřebné operace probíhaly automaticky.

3.3.1.11 Využití šablon funkcí pro lepší implementaci bitových operací

Bitové operátory můžeme přetížit a definovat je tak, aby automaticky prováděly potřebné převody. Abychom mohli definovat operátory pro jakékoliv typy *enum class*, je nutné využít šablon. Každý nový typ *enum class* by jinak potřeboval svou vlastí implementaci.

Generickou funkci pro operátor logického součtu můžeme definovat následovně:

```
template<typename T>
std::enable_if<std::is_enum<T>,T>::type operator| (T levy, T pravy){
   using number_type = typename std::underlying_type<FileAccess>::type;
   return static_cast<FileAccess>(
      sstatic_cast<number_type>(levy) | static_cast<number_type>(pravy)
   );
}
```

Vytvoření instance této šablony je podmíněné tím, že argumenty jsou výčtového typu. Nevýhodou však je, že vytvořením uvedeného přetížení operátoru bitového součtu, umožníme jeho použití pro všechny výčtové typy. Tím se znovu připravíme o část typové bezpečnosti.

Ideální stav by byl takový, aby bylo možné rozlišovat běžné *enum class*, pro které by bitové operace neměly být funkční a enum class reprezentující příznaky, kde tyto operace potřebujeme. Můžeme toho dosáhnout s pomocí specializace šablon a podmíněného vytváření jejich instancí.

Nejprve vytvoříme šablonu struktury, která obsahuje logickou hodnotu:

```
template<typename T>
struct enable_enum_class_flags {
   static constexpr bool value = false;
}
```

Tuto šablonu můžeme nyní využívat pro podmíněné vytvoření instance šablony operátoru.

Toho dosáhneme doplněním kontroly logické hodnoty proměnné *value* do předchozího příkladu:

```
template<typename T>
std::enable_if<std::is_enum<T> && enable_enum_class_flags<T>::value,T>::type
    operator| (T levy, T pravy){
   using number_type = typename std::underlying_type<FileAccess>::type;
   return static_cast<FileAccess>(
      static_cast<number_type>(levy) | static_cast<number_type>(pravy)
   );
}
```

Nyní potřebujeme nějaký mechanismus, jak změnit logickou hodnotu ve struktuře *enable_ enum_ class_ flags* pro konkrétní datové typy. K tomu využijeme specializaci šablon. Můžeme definovat například následující specializaci struktury *enable_ enum_ class_ flags*:

```
template<>
struct enable_enum_class_flags<FileAccess> {
   static constexpr bool value = true;
}
```

Definicí této šablony umožníme využití operátoru bitového součtu pro operandy typu *enum class FileAccess*. Stejným způsobem by bylo možné povolit operátor bitového součtu pro jakýkoliv jiný *enum class*. Protože povolování operátorů tímto způsobem by bylo zbytečně zdlouhavé, můžeme si usnadnit práci pomocí makra preprocesoru:

```
#define ENABLE_ENUM_CLASS_FLAGS(T) template<> \
struct enable_enum_class_flags<T>{ \
    static constexpr bool value = true; \
}
```

Toto makro nám umožní snadné povolení bitových operátorů pro konkrétní typ. Praktické použití by vypadalo následovně:

```
enum class FileAccess{
   read = 0,
   write = 1
};
ENABLE_ENUM_CLASS_FLAGS(FileAccess);
```

3.3.2 Pilíře objektově orientovaného programování

3.3.2.1 Zapouzdření

Zapouzdření je jeden z klíčových konceptů objektově orientovaného programování. Spočívá v omezení přístupu k některým datovým polím a metodám třídy. Pomocí zapouzdření je možné dosáhnout integrity dat, protože kód využívající třídu, nemůže volně měnit její vnitřní stav; k tomu musí použít metody třídy.

K omezení přístupu se používají klíčová slova *public, private* a *protected*. Veškeré proměnné a metody tříd jsou ve výchozím stavu *private* (na rozdíl od struktur, kde je výchozí *public*).

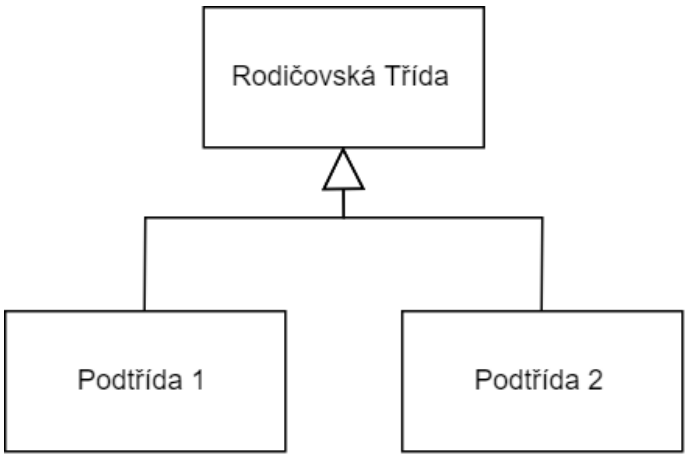
Proměnné a metody třídy označené klíčovým slovem **private** jsou přístupné pouze z vnitřku této třídy. Proměnné a metody označené klíčovým slovem **protected** jsou přístupné zevnitř třídy a ze všech tříd, které jí rozšiřují. Proměnné a metody označené klíčovým slovem **public** jsou přístupné všem.

Někdy však může nastat situace, kdy potřebujeme udělit přístup k *private* proměnným a metodám i funkci, která není členem třídy. K tomu můžeme použít klíčové slovo **friend**. Při deklaraci třídy deklarujeme členskou funkci jako „přátelskou“. Ta poté může

přistupovat i k *private* členům třídy. [9, 11, 10]

3.3.2.2 Dědičnost

je v objektově orientovaném programování způsob, jak stanovit vztah mezi třídami „třída je“ a případně definovat specializovanou verzi existující třídy. Třída, od které je děděno, se nazývá *rodičovská třída*. Třída, která dědí od rodičovské třídy (rozšiřuje rodičovskou třídu), se nazývá *podtřída*. Jednu třídu může rozšiřovat více podtříd. UML diagram zobrazující tento vztah je na obrázku 3.1; zobrazuje dvě třídy, které dědí od třídy „Rodičovská Třída“.



Obrázek 3.1: Dědičnost tříd. Na obrázku jsou zobrazeny dvě třídy, které dědí od rodičovské třídy.

Podtřída od své rodičovské třídy přejímá veškerá datová pole i všechny metody. Podtřída může definovat nová datová pole i metody a tím se specializovat. Zděděné metody může podtřída předefinovat. Musí však zachovat stejné vstupní parametry metod i datový typ návratové hodnoty. [9]

3.3.2.3 Polymorfismus

se označuje možnost volání metod a přístupu k datovým polím *podtřídy* prostřednictvím ukazatele, který je ukazatelem na typ *rodičovské třídy*. Můžeme tedy vytvořit objekt typu *podtřída* a ukazatel na něj uložit do proměnné typu *rodičovská třída*. To je zvláště výhodné při psaní modulárního software a různých abstrakčních vrstev; rodičovská třída definuje rozhraní a tím je garantováno, že ho respektují všechny podtřídy. Kteroukoliv podtřídu potom můžeme používat prostřednictvím tohoto rozhraní. Je tedy možné v programu vyměnit jednu podtřídu za jinou, bez nutnosti změn v kódu, který ji využívá. [9]

3.3.3 Návrhové vzory

V dobře strukturované objektově orientované architektuře se, při řešení podobných problému, vždy setkáme s opakujícími se vzory. Často opakované vzory byly popsány a pojmenovány. Je dobrým zvykem snažit se pro řešení běžných úkolů používat existující návrhové vzory. Jejich použití usnadní orientaci v kódu nejen jeho autorovi, ale i dalším lidem, kteří s kódem budou v budoucnu pracovat. [11]

V dalších odstavcích popisuji návrhové vzory, které jsem využil při vytváření abstrakční vrstvy pro knihovnu RTM.

3.3.3.1 Singleton

Singleton se využívá v případech, kdy potřebujeme zajistit, že existuje pouze jedna instance třídy v celé aplikaci. Konkrétní implementace se může lišit v závislosti na použitém programovacím jazyce. UML diagram tohoto návrhového vzoru je k vidění na obrázku 3.2. V C++ je nutné podniknout následující kroky: [11]

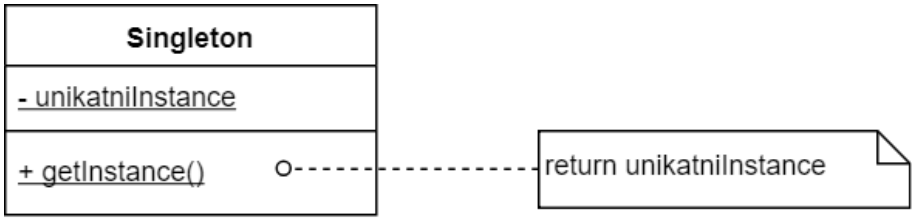
1. Označit konstruktor třídy klíčovým slovem *private*. Tím se znemožní volání konstruktoru kódem vně třídy a tedy i vytvoření instance.

2. Označit *kopírovací konstruktor* a *přesouvací konstruktor* pomocí „= delete;“. Tím se zabrání kompilátoru v jejich implicitnímu generování. Jejich existence by mohla umožnit kopírování objektu a tím vytvoření nežádaných instancí. Alternativou by mohlo být ruční implementace obou těchto konstruktorů tak, aby nevytvářely novou instanci.

3. Vytvořit *statické* datové pole, které obsahuje ukazatel na instanci třídy.

4. Vytvořit *statickou* metodu, která vrací ukazatel na instanci. Pokud instance zatím neexistuje, tak ji nejprve vytvoří.

UML diagram návrhového vzoru *singleton* je na obrázku 3.2.



Obrázek 3.2: Návrhový vzor singleton se používá všude tam, kde je nutné zajistit možnost vytvoření pouze jedné instance třídy.

3.3.3.2 Multiton

Multiton je návrhový vzor téměř totožný se vzorem *singleton*. Je u něj, stejně jako u singletonu, nutné zabránit v přístupu ke konstruktorům; místo toho uživatel dostane ukazatel na instanci třídy od specializované metody této třídy. Jediný rozdíl oproti singletonu je ten, že může existovat více než jedna instance.

Multitonu se využívá například při reprezentaci systémových zdrojů, kterých sice může být větší množství než jeden, ale i tak jich je omezené množství. [11]

3.3.3.3 Observer

Observer je další často používaný návrhový vzor. Používá se, když má nějaký objekt „sledovat“ jiný objekt. Příkladem může být kreslicí program, který umožňuje vkládat body a spojovací úsečky. Poloha bodu je definovaná pomocí souřadnic. Poloha čáry, je definovaná pomocí dvou bodů. Představme si následující situaci:

1. Vytvoříme bod A definovaný souřadnicemi.

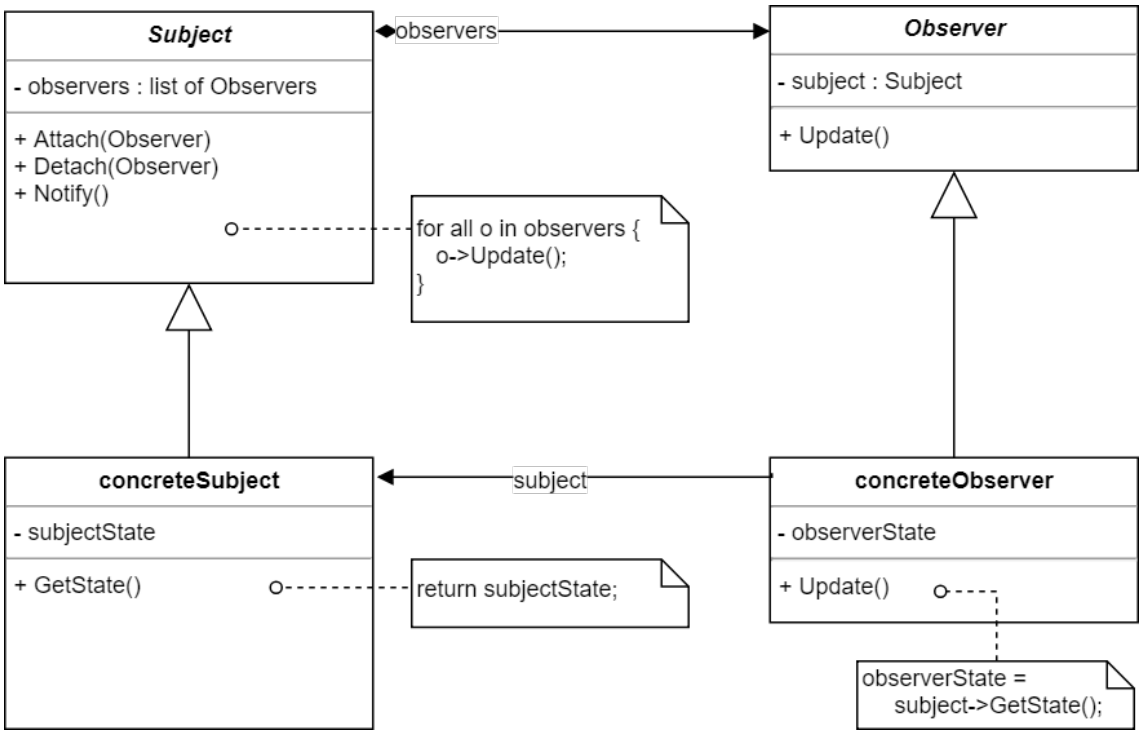
2. Vytvoříme bod B definovaný souřadnicemi.

3. Vytvoříme úsečku definovanou body A a B.

4. Změníme souřadnice bodu A.

Tímto postupem dosáhneme stavu, který by neměl nikdy nastat: úsečka je definovaná body A a B, po změně souřadnic boudu A se ale nezmění souřadnice počátku čáry. Musíme zajistit nějaký mechanismus aktualizace souřadnic čáry při změně souřadnic bodů, které ji definují. Právě k tomu slouží návrhový vzor *Observer*. Název návrhového vzoru *Observer* sice evokuje, že sledující třída aktivně sleduje sledovanou třídu. Opak je však pravdou. Sledovaná třída v případě potřeby upozorní sledující třídu o význačné události. Návrhový vzor *Observer* funguje následovně:

Existují dvě základní rozhraní: *Subject* a *Observer*. Subject reprezentuje sledovanou třídu a *Observer* reprezentuje sledující třídu. Subject si uchovává seznam *Observerů*, kteří se registrovali (nebo byli registrováni) pomocí metody *Attach(Observer)*. Je na třídě *Subject* aby rozhodla, které události jsou natolik význačné, že upozorní třídu *Observer* na změnu. Rozhraní *Subject* a *Observer* musí mít pro použití konkrétní implementaci. V situaci s body a úsečkami popsané výše by třída reprezentující úsečku implementovala rozhraní *Observer*. Třída implementující bod by implementovala rozhraní *Subject*. Při změně souřadnic by třída reprezentující bod volala metodu *Notify()*, která upozorní všechny registrované *Observery*, že došlo ke změně. Třída reprezentující úsečku by si následně aktualizovala souřadnice příslušného bodu. Grafické znázornění návrhového vzoru *Observer* je na obrázku 3.3 [11]



Obrázek 3.3: Návrhový vzor observer se používá ve chvíli, kdy potřebujeme aby jeden objekt sledoval stav a změny jiného objektu.

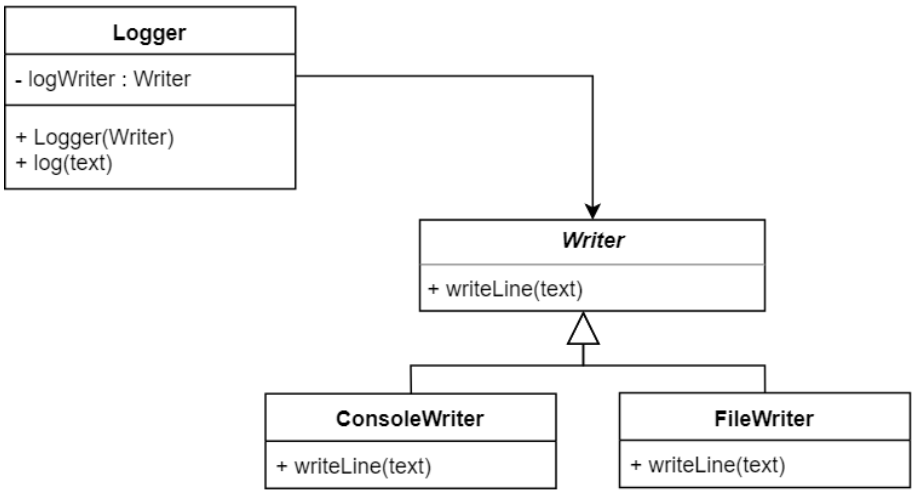
3.3.4 Dependency injection

Při psaní programu se velice často setkáme s tím, že na sobě jednotlivé části různým způsobem vzájemně závisí. U objektově orientovaného programování se to projevuje tím, že jeden objekt (obvykle zvaný „klient“) pro nějakou svou činnost využívá jiný objekt (obvykle zvaný „služba“). U procedurálního programování to obvykle znamená, že jedna funkce (klient) k něčemu využívá jinou funkci (službu). S pojmem *dependency injection*

se setkáme častěji u objektově orientovaného programování. Proto ho dále budu popisovat pouze pro něj.

Dependency injection spočívá v tom, že nedovolíme, aby si objekt *klient* sám vytvářel instanci třídy *služba*. Místo program vytvářející instanci třídy *klient* vytvoří i instanci třídy *služba*. Objektu *klient* poté instanci třídy služba předá (všechny třídy *služba* musejí mít stejné rozhraní). Tím dosáhneme lepšího oddělení zodpovědnosti a umožní nám to vytvářet opakovaně využitelný kód. Lépe si to lze představit na příkladu:

Mějme třídu *Logger*, která zajišťuje *logování* (zaznamenávání událostí) v programu. Ta umožňuje vytvoření zprávy s definovaným textem, s časovou známkou a její uložení. K tomu třída *Logger* využívá rozhraní *Writer*, které umožňuje zápis řádku textu na cílové místo pomocí metody *writeLine(text)*. Třídy *ConsoleWriter* a *FileWriter* implementují toto rozhraní. *ConsoleWriter* umožňuje vypsat řádek textu do konzole. *FileWriter* umožňuje přidat řádek textu na konec souboru. Třída *Logger* může využít k uložení zprávy kteroukoliv z těchto dvou tříd. Vztah mezi jednotlivými třídami je zobrazen na obrázku 3.4.

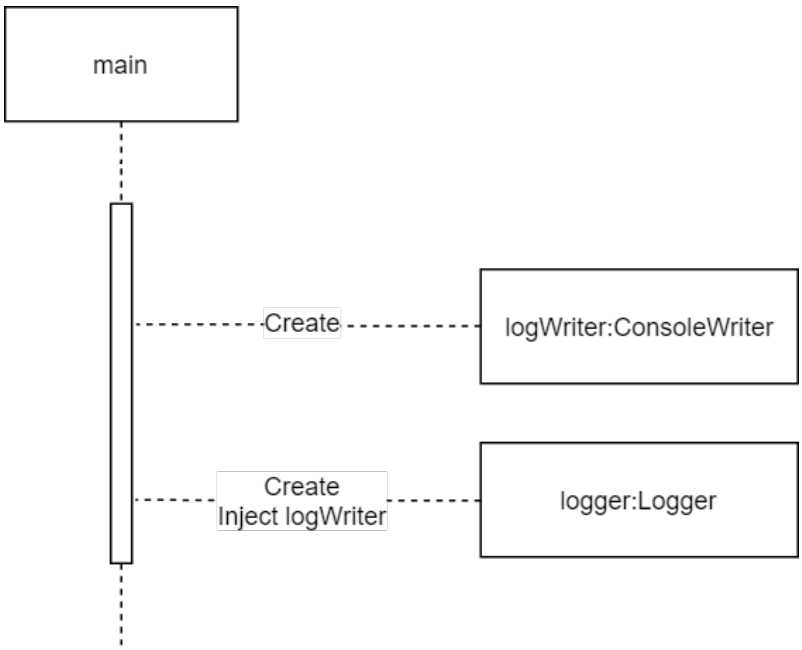


Obrázek 3.4: Vztah třídy *Logger* s třídami *ConsoleWriter* a *FileWriter*.

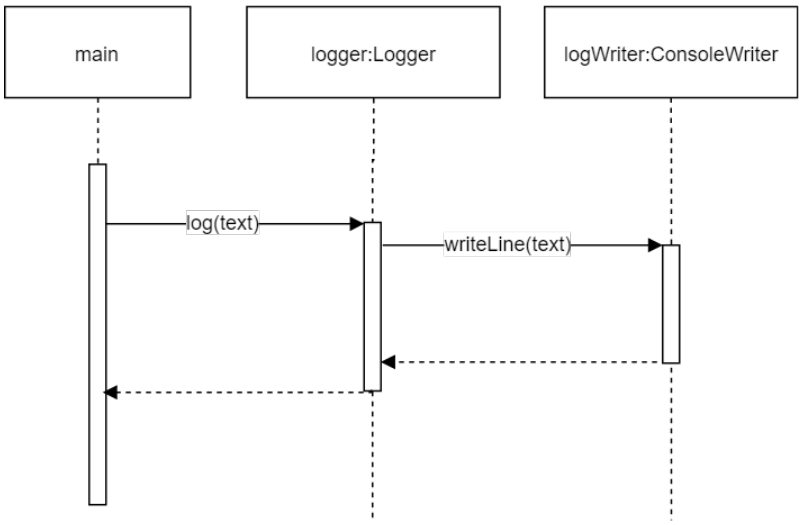
Bez použití *dependency injection* by třída *Logger* sama vytvářela instanci třídy *FileWriter*, nebo *ConsoleWriter*. V takovém případě by bylo uvnitř třídy *Logger* rozhodnuto o tom, jestli se bude zapisovat do souboru, nebo do konzole a která třída se k tomu účelu použije. Nevýhodou je to, že třída *Logger* může vybírat jen z těch těch možností, které existovaly v době jejího psaní.

Při použití dependency injection o vytvoření *ConsoleWriter*, nebo *FileWriter* rozhoduje uživatel třídy *Logger*. Třída *Logger* při svém vytváření očekává pouze ukazatel na třídu implementující rozhraní *Writer*. Ten potom využívá ke svému fungování. Nemusí přitom vědět nic o implementaci konkrétní třídy. Hlavní výhodou je to, že je kdykoliv možné přidat nové třídy implementující rozhraní *Writer* a použít je k zápisu událostí pomocí třídy *Logger* bez jejích jakýchkoliv úprav. Mohli bychom například přidat třídu *SerialWriter*, která bude odesílat řádky po sériové lince.

Tvorba instance třídy Logger s dependency injection je znázorněna na obrázku 3.5 na straně 25. Ve funkci *main()* je nejprve vytvořena instance třídy *ConsoleWriter*. Následně je vytvořena instance třídy *Logger*, která jako parametr konstruktoru dostane již dříve vytvořenou instanci třídy *ConsoleWriter*. Zápis události se provádí pomocí funkce *log(string)*. *Logger* využije třídu *Writer* pro zápis události na určené místo. V případě tohoto příkladu bude vypsaná do konzole. Postup volání funkcí je zobrazen na obrázku 3.6 na straně 25. [11]



Obrázek 3.5: UML Call diagram vytvoření instance třídy *ConsoleWriter* a její použití při vytváření třídy *Logger*.



Obrázek 3.6: UML Call diagram zápisu události do konzole prostřednictvím třídy *ConsoleWriter*

Kapitola 4

Realizace

4.1 Stanovení rozsahu práce

Protože program vznikal pro použití se specifickým hardwarem, nebude nutné upravovat všechen zdrojový kód. Některé části programu nemohou být použity vůbec. To je například kód, který zajišťuje konfiguraci a používání hradlového pole; to se totiž na cílové vývojové desce vůbec nenachází. Jiné části, jako například kód pro použití externí RAM, by sice použít šly, nicméně nejsou nezbytné pro fungování knihovny RTM a proto jsem se rozhodl je z důvodu nedostatku času neimplementovat.

Práci na knihovně lze rozdělit na tři hlavní části:

1. Kompilace samotné knihovny pro cílový mikroprocesor,
2. Vytvoření nezbytných HAL,
3. Vytvoření minimální aplikace pro otestování funkčnosti.

Jediným úkolem *první* části je pouze úspěšná kompilace a úspěšné nahrání kódu knihovny do cílového přípravku. Zatím se zde netestuje funkčnost knihovny ani nevolají se žádné její funkce. Tento krok je nezbytný, protože cílový přípravek obsahuje mikroprocesor jiné architektury s jinou velikostí paměti. Oba kompilátory mohou mít nestandardní rozšíření, která nejsou k dispozici na druhé platformě. Proto nelze čekat, že kód bude možné bez jakýchkoliv úprav zkompilovat pro druhou platformu.

Úkolem *druhé* části bylo vytvoření stanovení nezbytných HAL, jejich implementace a provedení nutných úprav ovladačů. Po analýze vzorového projektu jsem vybral následující komponenty jako nezbytné pro funkci minimální aplikace:

- HAL pro sériovou komunikaci,
- Ovladač pro sériovou komunikaci,
- HAL pro GPIO,
- HAL pro časovač.

Třetím, krokem bylo vytvoření minimální aplikace, která umožní otestování vytvořeného řešení a poslouží jako vzor pro použití upravené knihovny jejím dalším případným uživatelům. V testovacím programu je vytvořené několik proměnných, které jsou zaregistrované do knihovny RTM. Dále zde probíhá časovaná komunikace přes sériovou linku.

4.2 Úpravy pro úspěšnou kompilaci a nahrání do mikropočítače

První krok se ukázal být relativně jednoduchý. Při pokusu o kompilaci a spuštění programu jsem narazil jen na tři problémy:

Klíčové slovo *interrupt* Toto klíčové slovo je nestandardní rozšíření jazyka C/C++ firmou *Texas Instrumets*. Jejich kompilátor pro architekturu C28x ho využívá pro označení funkce pro obsluhu přerušení. Označením funkce klíčovým slovem *interrupt* dojde při jejím vyvolání k uložení stavu všech strojových registrů a k vygenerování speciální návratové sekvence z funkce. U procesorů ARM dochází k ukládání registrů při volání obslužných funkcí automaticky a proto u nich toto klíčové slovo neexistuje.

První problém jsem tedy vyřešil pouhým smazáním všech výskytů klíčového slova *interrupt*.

Velikost paměti RAM použitého přípravku je menší, než u původní desky. Program tedy už bylo možné zkompilovat, při spuštění však došlo k systémové chybě při pokusu o zápis do části paměti, která je určená pouze ke čtení. Velikost bufferů knihovny RTM jsem proto snížil na polovinu. Po této úpravě již lze program nejen zkompilovat, ale i spustit. Cenou za to je kratší doba, po kterou je přípravek schopen zaznamenávat data v offline režimu.

Integrace se zdrojovým psaném v C++ Autoři knihovny RTM nepočítali s tím, že by se knihovna využívala z jiného programovacího jazyka než z C. Kompilátor C++ provádí takzvaný „name mangling“. Během něj mění názvy proměnných, funkcí a někdy i datových typů. Slouží k tomu, aby linker byl schopný rozlišit mezi různými proměnnými nebo funkcemi se stejným názvem. Díky němu je možné například přetěžování funkcí, nebo omezení oboru platnosti proměnných.

Při kompilací kódu, který je psaný čistě v C++ name mangling nepředstavuje problém. Názvy jsou upravovány deterministicky; díky tomu název funkce deklarovaný v hlavičkovém souboru knihovny odpovídá jejímu názvu v objektovém souboru při linkování. Problém může nastat při použití knihovny napsané v jazyce *C* v programu psaném v jazyce C++; protože kompilátor pro jazyk C name mangling neprovádí, tak názvy funkcí v objektovém souboru zůstávají stejné. Při vložení hlavičkového souboru knihovny do kódu psaném v jazyce C++, při kompilaci proběhne *name mangling* všech názvů v kódu, a to včetně těch z hlavičkového souboru použité knihovny. Ve fázi linkování nebude možné ztotožnit deklarované názvy funkcí, na kterých proběhl name mangling, s názvy funkcí v knihovně.

Tomu se dá v kódu psaném v C++ zabránit označením deklarací, které se týkají programu napsaného v C. Jejich názvy se použijí přímo tak, jak jsou deklarovány. Slouží k tomu klíčové slovo *extern "C"*. Tím dáváme kompilátoru najevo, že se deklarace týkají názvů definovaných v jazyce C. Klíčové slovo *extern "C"* můžeme použít dvojím způsobem. Je možné buď označit každou deklaraci zvlášť:

```
extern "C" void funkce1();
extern "C" void funkce2();
```

Nebo můžeme označit celý blok najednou:

```
extern "C" {
   void funkce1();
   void funkce2();
}
```

Při používání knihoven psaných v C se využívá druhá varianta, protože umožňuje obsažení všech deklarací najednou. Při použití kódu jak byl uveden výše bychom však narazili na další problém, tentokrát při pokusu o kompilaci kódu v jazyce C, který používá tuto knihovnu; jazyk C nezná klíčové slovo *extern "C"*. Tento problém můžeme snadno vyřešit pomocí preprocesoru. Při kompilaci kompilátorem C++ je definované makro __ *cplusplus*. Můžeme ho využít pro podmíněné vložení kódu v hlavičkovém souboru tímto způsobem:

```
#ifdef __cplusplus
extern "C"{
#endif

   void funkce1();
   void funkce2();

#ifdef __cplusplus
}
#endif
```

Alternativní řešení by bylo hlavičku knihovny neupravovat, ale místo toho jí označit pomocí klíčového slova *extern "C"* v místě vložení do zdrojového kódu. To by vypadalo takto:

```
extern "C" {#include "knihovna.h"}
```

Výhodou třetí varianty je, že není nutné žádným způsobem měnit použitou knihovnu. Na druhou stranu bývá dobrým zvykem při psaní knihoven v jazyce C napsat hlavičkový soubor knihovny dle varianty 2. To umožní její používání z C i C++ bez toho, aby její uživatel potřeboval vědět v jakém jazyce byla napsaná. Proto jsem se rozhodl upravit hlavičkové soubory knihovny RTM dle varianty 2 a myslím si, že by bylo vhodné, aby to do ní bylo oficiálně přidáno i jejími autory.

4.3 Vytvoření potřebných HAL a úpravy ovladačů

Tento krok vyžadoval více úprav, než krok první. To je způsobeno značnou odlišností obou mikropočítačů. Díky tomu, že při vývoji knihovny bylo s možností přenesení na jinou platformu počítáno, nebylo potřeba provádět žádné změny v samotné logice protokolu RTM. Veškeré změny se omezily na HAL a ovladače.

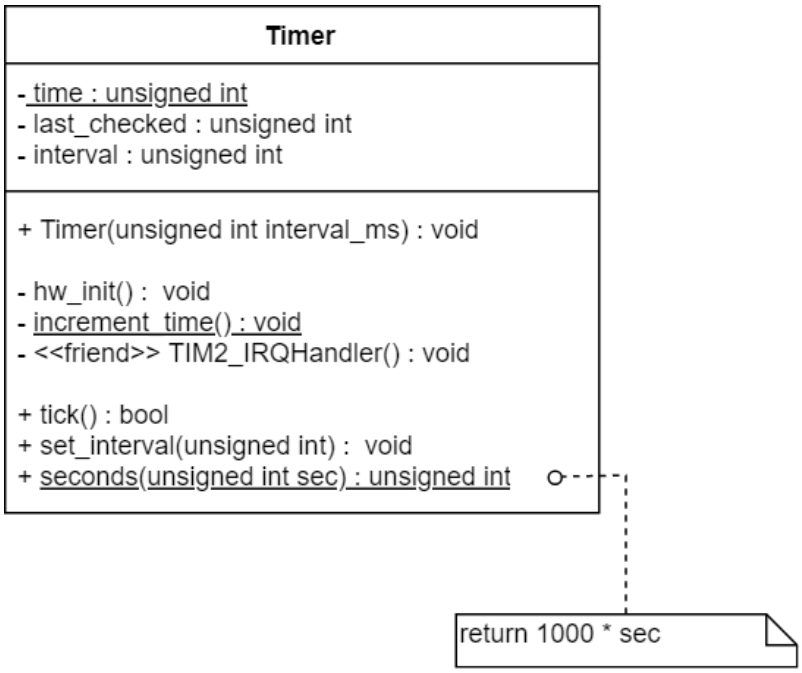
Při provádění změn jsem se rozhodl pro větší zásah do kódu a pro jeho přepis v jazyce C++ za použití objektově orientovaného programování. To mi umožnilo snadné využití abstrakcí a dosažení značné modularity programu.

4.3.1 HAL pro časovač

Při použití knihovny RTM je nutné některé operace spouštět periodicky, ale ne při každém průběhu hlavní programové smyčky. K tomu byl použitý časovač mikropočítače, který s nastavenou periodou vyvolává přerušení. Ke spouštění opakujících se funkcí nebyla použita přímo funkce obsluhující přerušení. HAL časovače používá funkci obsluhující přerušení k inkrementaci vnitřní proměnné, která zaznamenává uplynutou dobu. HAL se pro časování používá nepřímo; HAL sám nevolá žádnou funkci po uplynutí definovaného času. Místo toho poskytuje funkci *uint16_t hal_tmrDelay(uint32_t delay, uint16_t id)*, která vrací logickou hodnotu v závislosti na tom, jestli od jejího posledního volání uplynul čas definovaný parametrem *delay* (delay udává dobu v desetinách milisekundy). Parametr *id* udává číslo časovače. HAL pro časovač umožňuje časování ve více nezávislých intervalech. Funkce *tmrDelay* může být použitá v hlavní smyčce pro spouštění časovaných událostí následovně:

```
int main(){
   while(1){
      if(hal_tmrDelay(100,0)){
         // kod spusteny kazdych 10ms
      }
      // kod spusteny pri kazdem prubehu cyklu
   }
}
```

Výhodou takového přístupu je možnost zachování krátkých obslužných funkcí přerušení (v porovnání s tím, obslužné funkce přímo vykonávaly nějakou práci). To je důležité především u mikropočítačů, které neumožňují vnořování přerušení (volání dalšího přerušení ve chvíli, kdy jedno je právě obsluhováno). Cenou za to je, že časování nemusí být úplně přesné a je závislé na době trvání průběhu hlavní smyčkou. Pokud například doba hlavní smyčky je 1 s, není možné časovat s periodou 100 ms.



Obrázek 4.1: UML diagram třídy *Timer*.

HAL časovače pro mikropočítač STM32F429ZI jsem napsal tak, aby jeho použití bylo podobné původnímu HAL psaném v jazyce C. Základní princip funkce je tedy stejný - vyvolené přerušení pouze inkrementuje proměnnou uchovávající informaci o čase a samotný časovaný kód se volá z hlavní programové smyčky.

Na rozdíl od původní implementace je však časovač reprezentovaný třídou v jazyce C

4.3.1.1 Třída Timer

Třída *Timer* pro svou funkci využívá tři proměnné: *time*, *interval* a *last_ checked*. Je zde využíváno chování vlastností proměnných, které reprezentují celá nezáporná čísla; rozdíl dvou takovýchto proměnných vyjde správně i v případě, že u jedno z operandů došlo k přetečení, nebo podtečení hodnoty. Díky tomu není nutné proměnnou *time* nulovat, ale je možné jí nechat neustále inkrementovat. Hodnota rozdílu proměnných *time* - *last_ checked* stále bude platná (Toto platí, pokud nedojde k vícenásobnému přetečení. Vzhledem k tomu, že k přetečení časovače dojde přibližně jednou za 5 dnů, lze předpokládat, že k vícenásobnému přetečení během jednoho průchodu programovou smyčkou nedojde.). UML diagram třídy *Timer* je na obrázku 4.1.

Proměnné třídy Timer

static unsigned int time tato proměnná je inkrementovaná pokaždé, když mikropočítač vyvolá přerušení. Uchovává tak celkový uplynulý čas.

unsigned int interval tato proměnná uchovává požadovanou periodu časovače. Je nastavena při konstrukci objektu.

unsigned int last_checked tato proměnná slouží pro kontrolu uplynuté doby od počátku periody časovače. Po uplynutí periody je její hodnota nastavena na aktuální hodnotu proměnné *time*.

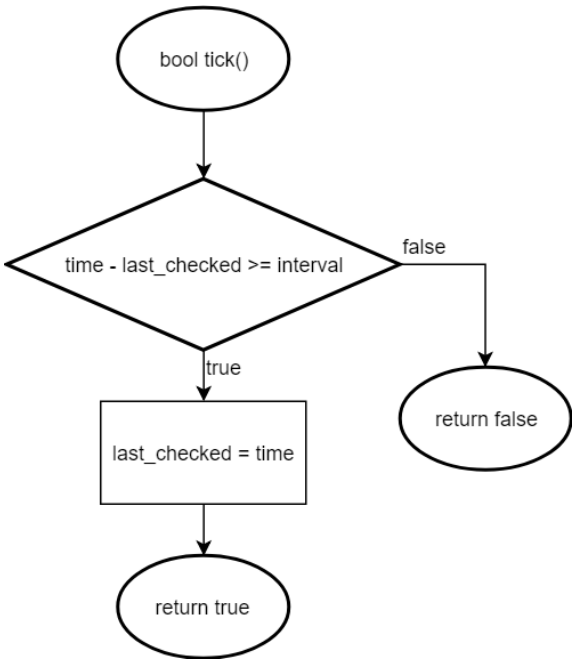
Metody třídy Timer

Třída *Timer* má smazaný výchozí konstruktor. Je nežádoucí, aby bylo možné vytvořit instanci této třídy bez nastavení intervalu. Díky tomu je objekt typu *Timer* za celou dobu svého života validní a připravený k časování.

void set_interval() Metoda slouží k přenastavení intervalu časovače po jeho vytvoření. Neslouží k časování více událostí s různými intervaly, ale k občasnému přenastavení.

bool tick() Metoda tick() se volá v hlavní programové smyčce. Metoda vrací logickou hodnotu *true*, pokud od posledního volání uběhl interval nastavený při konstrukci třídy (vypočte se jako rozdíl *time - last_checked*). V takovém případě se proměnná *last_checked* nastaví na aktuální hodnotu časovače. Pokud stanovený interval neuplynul, funkce vrací logickou hodnotu *false* a hodnoty proměnných nemění. Vývojový diagram metody *tick()* je na obrázku 4.2.

static unsigned int seconds(unsigned int) Tato metoda převádí čas zadaný v sekundách na milisekundy. Slouží především ke zlepšení čitelnosti programu. Jedná se o statickou metodu, takže je možné jí využít i k nastavení intervalu při konstrukci objektu.



Obrázek 4.2: Vývojový diagram metody *tick()*.

Použití třídy Timer

Při vytváření instance třídy *Timer* se pomocí parametru konstruktoru nastaví interval v ms. Tento interval sice lze měnit, ale počítá se s tím, že bude povětšinou nechaný konstantní. Pro časování se dvěma různými intervaly by měly být použity dvě instance třídy.

Po vytvoření instance můžeme použít metodu *tick()* pro časování událostí. Použití je obdobné, jako v původním HAL. Na rozdíl od původního řešení se však časovací funkce volá bez parametrů. Oba parametry jsou reprezentovány interně v objektu časovače. Objekt časovače se dá použít následovně:

```
int main(){
   Timer sekunda(1000); // nastaveni intervalu na 1000 ms
   while(1){
      if(sekunda.tick()){
         // kod spusteny kazdou sekundu
      }
      // kod spusteny pri kazdem prubehu cyklu
   }
}
```

4.3.2 HAL pro GPIO

HAL pro GPIO slouží k nastavení parametrů univerzálních vstupů a výstupů mikropočítače. Slouží k odstínění uživatele od nutnosti znalosti funkce konkrétního hardwaru. Skládá ze se dvou tříd a několika výčtových typů. UML diagram HAL pro GPIO je na obrázku 4.3 na straně 36.

Použité návrhové vzory

Třída *Gpio* (bude popsaná dále v textu) využívá návrhový vzor multiton. Na zařízení je několik portů, které se však používají všechny stejně. Nemělo proto smysl vytvářet pro každý z nich speciální třídu. Místo toho jsou všechny porty reprezentovány jednou třídou, které se během konstrukce nastaví který z portů má využívat.

Může existovat pouze jedna instance třídy pro každý port. Proto jsem použil návrhový vzor multiton. Metoda třídy pro získání instance má jako parametr požadovaný port; pokud již existuje instance pro tento port, bude na ní vrácen ukazatel. Pokud existují jen instance pro jiné porty, případně pokud neexistuje žádná instance, bude tato instance vytvořena.

4.3.2.1 Třída *PortSettings*

Tato třída slouží k uchování konfigurace pro piny GPIO. Obsahuje veškeré parametry, které u každého pinu lze nastavovat. Pro nastavení pinu tedy stačí jeho číslo a instance této třídy.

4.3.2.2 Třída *Gpio*

Třída Gpio slouží k ovládání GPIO portů na přípravku. Umožňuje nastavení jednotlivých pinů (k tomu používá třídu *PortSettings*) a jejich základní ovládání.

Proměnné třídy *Gpio*

static std::map<Port, Gpio*> usedPorts Proměnná usedPorts je typu „mapa“. Uchovává páry *klíč →hodnota*. Pomocí klíče je možné přistupovat k jednotlivým hodnotám, podobně jako u pole lze přistupovat k hodnotám na základě číselného indexu. Proměnná typu *Port* ze slouží jako klíč pro hodnotu typu *ukazatel na Gpio*.

V této mapě jsou uloženy ukazatele na instance třídy *Gpio*. Pro každý klíč tato mapa obsahuje maximálně jeden záznam.

static std::map<Port, GPIO_TypeDef*> gpioRegisters Tato mapa uchovává ukazatele na jednotlivé GPIO registry. Ty jsou definovány v hlavičkových souborech poskytovaných výrobcem mikropočítače a slouží k ovládání hardware GPIO.

GPIO_TypeDef* portRegister V této proměnné se uchovává registr GPIO spojený s konkrétním portem. Při konstrukci objektu je získán z mapy gpioRegisters. Tím zamezuji nutnosti potřebný registr získávat z mapy při jeho každém použití.

std::bitset<16> used_pins V této proměnné se uchovává informace o tom, které piny z daného portu jsou již použity.

Port port V této proměnné se uchovává hodnota výčtového typu *Port*. Ta určuje, který port je spojený s danou instancí.

Metody třídy *Gpio*

inline uint16_t getPortIndex() Vrací číselný index portu GPIO dané instance třídy *Gpio*. Používá se například při spuštění hodin pro daný port.

inline void enableClock() Inicializuje a spustí hodiny pro daný port. Pokud hodiny již běží, nedělá nic.

static Gpio* getInstance(Port) Slouží k získání instance třídy *Gpio*. Požadovaný port je zadán jako parametr při volání. Metoda vrátí ukazatel na instanci třídy, která patří k danému portu. Pokud neexistuje, bude nejprve vytvořena.

void init(std::bitset<16>, PortSettings) Slouží k inicializaci portu. Piny definované v proměnné typu *bitset* jsou nastaveny v souladu s hodnotami nastavenými ve třídě *PortSettings*.

write(Pin, uint16_t) Slouží k zápisu logické hodnoty na pinu portu.

read(Pin, uint16_t) Slouží k přečtení logické hodnoty na pinu portu.

4.3.3 HAL pro sériovou komunikaci

Při psaní HAL pro sériovou komunikaci jsem se snažil o co největší univerzálnost výsledného řešení. Vytvořil jsem abstraktní třídu *uart*, která slouží jako rozhraní pro zbytek programu. Samotná třída *uart* neobsahuje žádnou implementaci, pouze deklaraci virtuálních metod. Je to tedy plně abstraktní třída.

Konkrétní implementaci sériové komunikace obsahují třídy, které rozšiřují třídu *uart*. Jako součást této práce jsem implementoval třídu *uart4*, které rozšiřuje třídu *uart* a umožňuje komunikaci prostřednictvím čtvrté sériové linky mikropočítače.

Zbytek programu by měl HAL k práci se sériovou linkou používat pouze prostřednictvím ukazatele typu *uart*. Díky tomu je kdykoliv možné začít používat jinou sériovou linku (případně VCP), pouhou výměnou instance třídy na kterou ukazatel ukazuje za jinou třídu rozšiřující třídu *uart*.

UML diagram HAL pro UART je na obrázku 4.4 na straně 37.

4.3.3.1 Použité návrhové vzory

Observer Třída *uart* implementuje návrhový vzor observer. To jí umožňuje informovat registrované třídy o definovaných význačných událostech. V tomto případě to slouží k informování ovladače sériové linky o přijetí nových dat. Jaký k tomu je důvod bude podrobněji popsáno v sekci týkající se ovladače.

Singleton Třída *uart4* implementuje návrhový vzor *singleton*. Ten zajišťuje existenci pouze jedné instance dané třídy. Protože tato třída reprezentuje systémový zdroj, který je fyzicky na zařízení přítomen jen jednou, vybízelo se využití tohoto návrhového vzoru.

Díky tomu, že vždy existuje maximálně jedna instance třídy *uart4*, není nutné nijak řešit sdílený přístup k fyzickému rozhraní *uart4*.

4.3.3.2 Třída *uart*

Třída *uart* je abstraktní třída, která slouží jako rozhraní pro sériovou komunikaci. Sama neobsahuje žádnou implementaci. Třída rozšiřuje rozhraní *subject* návrhového vzoru *observer*. To jí umožňuje informovat registrované třídy o přijetí nových dat po sériové lince.

Metody třídy *uart*

virtual void init() Slouží k inicializaci sériové linky. Implementaci zajišťuje třída, která rozšiřuje třídu *uart*.

virtual uint16_t putc() Slouží k odeslání jednoho znaku po sériové lince.

virtual uint16_t open() Slouží k řízení přístupu k sériové lince. Uživatel linky by měl před jejím použitím zavolat metodu open(). Pokud vrátí *true*, může linku používat.

virtual uint16_t close() Slouží k řízení přístupu k sériové lince. Uživatel linky by měl tuto metodu zavolat ve chvíli, kdy s užíváním skončil. Tím umožní přístup k sériové lince ostatním uživatelům.

Tento způsob řízení přístupu k sériové lince jsem převzal z původního řešení. Bohužel pro svoji efektivitu vyžaduje spolupráci uživatele knihovny, protože by mohl linku využívat i přes její neúspěšné otevření. Tím by mohl narušit komunikaci jiné části programu.

4.3.3.3 Třída *uart4*

Třída *uart4* rozšiřuje třídu *uart*. Implementuje abstraktní metody třídy *uart* a umožňuje komunikaci po čtvrté sériové lince na zařízení.

Proměnné třídy *uart4*

static uint16_t rxChar Uchovává poslední přijatý znak. V budoucnu možnost nahradit datovou strukturou, která je schopná uchovat více znaků.

static uart4* instance Uchovává ukazatel na instanci třídy *uart4*. Je použita pro implementaci návrhového vzoru *singleton*. Pokud obsahuje hodnotu *nullptr*, instance zatím nebyla vytvořena.

static bool isOpen Uchovává informaci o tom, jestli je sériová linka někým otevřena pro použití.

Metody třídy *uart4*

Třída obsahuje všechny metody třídy *uart*. Mimo těchto metod má následující:

static void setRxChar(uint16_t) Slouží k nastavení proměnné rxChar, která uchovává poslední přijatý znak. Proměnnou rxChar nenastavuje přímo obsluha přerušení, ale nastavuje ji prostřednictvím této metody. Díky tomu při nahrazení proměnné rxChar datovou strukturou schopnou uchovávat více záznamů, nebude nutné nijak měnit obsluhu přerušení.

static uart4* getInstance() Tato metoda implementuje návrhový vzor *singleton*. Slouží k získání ukazatele na instanci třídy uart4. Pokud tato instance neexistuje, bude nejprve vytvořena. Po vytvoření existuje po celou dobu běhu programu.

friend void UART4_IRQHandler() Tato funkce není metodou třídy *uart4*. Jedná se pouze o deklaraci funkce obsluhující přerušení jako „friend" funkce třídy *uart4*. Díky tomu má tato funkce přístup k „private" členům třídy *uart4*, i přesto že není sama jejím členem.

4.3.4 Ovladač sériové komunikace

Při psaní programu jsem se snažil o co nejmenší možné množství úprav kódu, který obsahoval logiku protokolu RTM. V původní verzi programu se vyskytovaly dvě kopie kódu ovladače sériové komunikace - *scia* a *scib*. Obě komponenty obsahovaly téměř stejný zdrojový kód. Lišily se pouze použitým HAL pro sériovou komunikaci. V zájmu zachování co nejmenší duplicity kódu jsem jeden z duplicitních zdrojových souborů smazal a druhý upravil tak, aby bylo možné jeho univerzální použití bez ohledu na to, který sériový port se aktuálně používá. K tomu používám metodu *dependency injection*, kterou jsem popsal v předchozích kapitolách. Úprava kódu se skládala z těchto částí:

1. Odstranění platformě závislých klíčových slov a volání funkcí,
2. přejmenování proměnných a funkcí odkazujících na konkrétní sériové rozhraní,
3. vytvoření „obalu" na existující zdrojový kód ovladače tak, že se jedná o třídu v jazyce C++.

Díky úpravě na třídu je možné ovladač snadno integrovat do zbytku modulární aplikace a případně ho snadno v budoucnu nahradit jinou implementací.

Kompletní UML diagram pro sériovou komunikaci je na obrázku 4.5 na straně na straně 38.

4.3.4.1 Použité návrhové vzory

Observer V původní implementaci ovladač sériové komunikace obsluhoval přerušení sériové linky. Kdybych to tak ponechal, byl by ovladač vázán na konkrétní hardware. Proto jsem se rozhodl přesunout obsluhu přerušení přímo do HAL a tím dosáhnout nezávislosti ovladače na konkrétní sériové lince.

Po odstranění obsluhy přerušení z kódu ovladače bylo nutné přidat nějaký jiný mechanismus, jak ovladač informovat o nově příchozích datech. K tomu jsem použil návrhový vzor *observer*. Třída *uart* díky tomu může třídu *uart_driver* upozornit vždy, když přijme nová data.

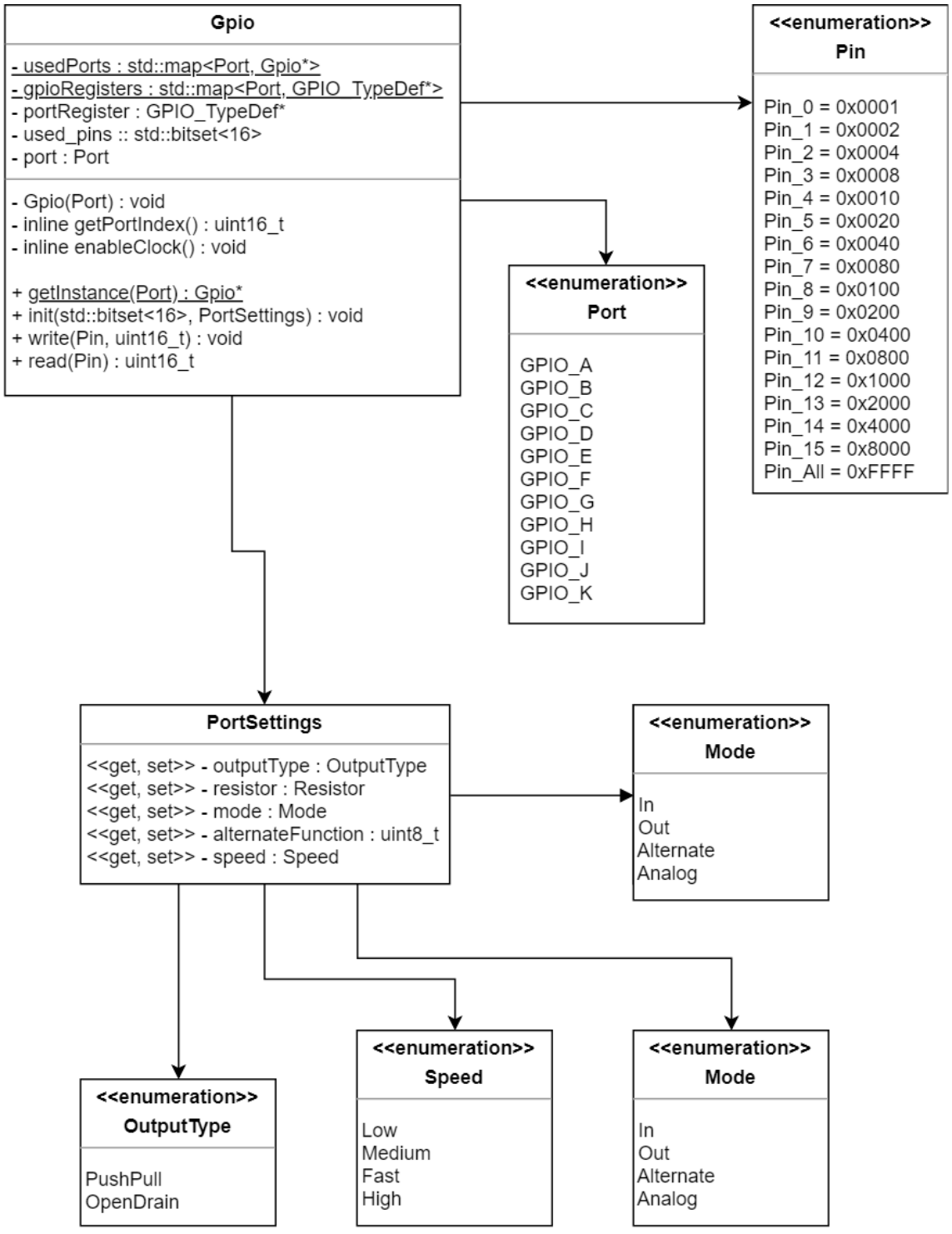
Celková architektura řešení pro sériovou komunikaci je na obrázku 4.5. To zobrazuje implementaci návrhového vzoru *observer* a vztah mezi ovladačem a HAL.

Proměnné a metody třídy *uart_ driver*

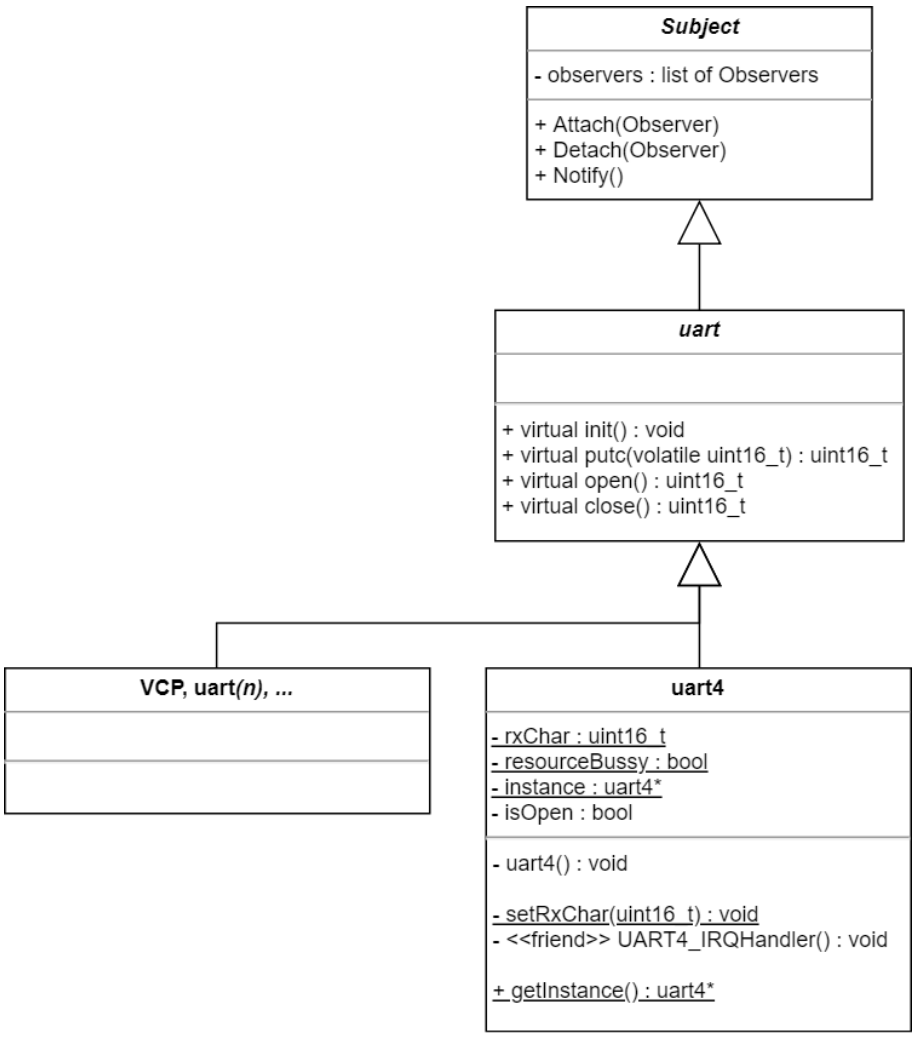
Protože kód třídy *uart_ driver* je téměř nezměněný od původní implementace, uvádím zde pouze proměnné a metody, které jsou v ní nové.

Proměnná *uart* uart_ hal* Tato proměnná uchovává ukazatel na instanci třídy *uart*. Ukazatel na požadovanou instanci je nastaven během konstrukce objektu. Díky tomu je možné vyměnit používanou konkrétní třídu pro HAL bez jakékoliv změny třídy *uart_ driver*.

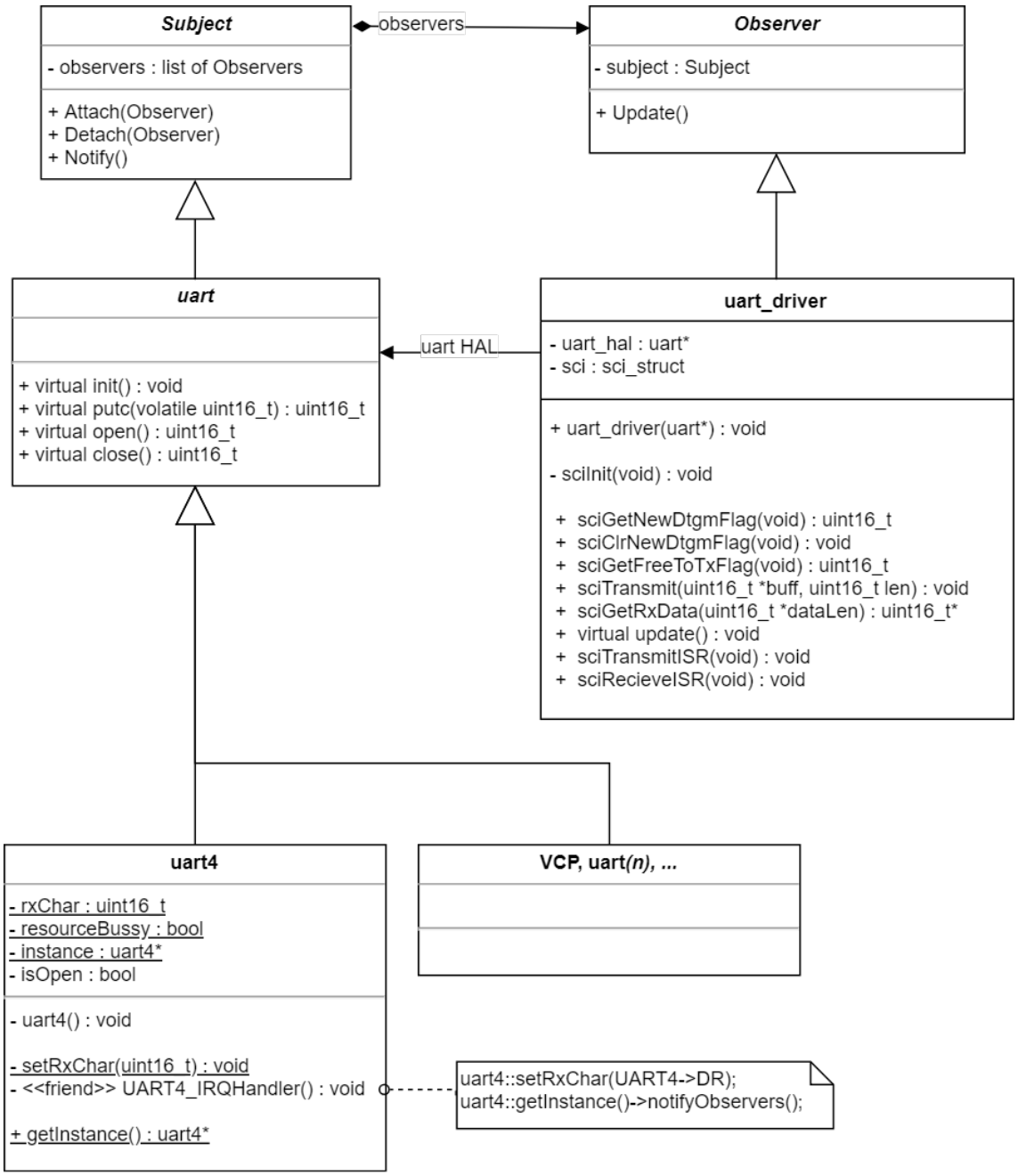
Metoda *virtual void update()* Tato metoda implementuje abstraktní metodu zděděnou z třídy *Observer*, která implementuje stejnojmenný návrhový vzor. Metodu *update()* volá třída *uart* při přijetí dat po sériové lince. V těle této metody je pouze zavolaná metoda *sciRecieveISR()*, která dříve sloužila pro obsluhu přerušení sériové linky.



Obrázek 4.3: UML diagram tříd pro *GPIO*.



Obrázek 4.4: UML diagram tříd pro *uart*.



Obrázek 4.5: UML diagram ovladače a HAL pro sériovou komunikaci.

4.4 Minimální aplikace

Minimální aplikace je nejjednodušší možná aplikace, která poslouží k otestování funkčnosti upravené knihovny. Musí umožňovat otestování všech bodů zadání. Po analýze vzorové aplikace pro mikroprocesor Delfino jsem dospěl k tomu, že aplikace musí mít následující:

1. Inicializaci hardware mikropočítače,
2. vytvoření proměnné, která slouží jako hlavní buffer pro knihovnu RTM,
3. inicializaci knihovny RTM,
4. několik vzorových proměnných pro přenos pomocí RTM,
5. programovou smyčku, která pomocí časovače periodicky spouští záznam proměnných knihovnou RTM a komunikaci po sériové lince.

4.4.1 Inicializace

Protože jsem při vytváření HAL použil objektově orientované programování, není nutné žádné HAL ani ovladače explicitně inicializovat. Veškerá potřebná inicializace proběhne automaticky v rámci konstruktorů, při vytváření instancí potřebných tříd. Protože logiku samotné knihovny jsem nijak neměnil, ani jsem pro ni nevytvářel objektově orientovaný *wrapper*, je nutné jí ručně inicializovat. Pro použití knihovny samotné stačí jeden hlavičkový soubor *rtm_monitor.h*. K inicializaci knihovny slouží funkce *rtm_Init*. Ta má následující parametry:

1. *myAddress* slouží k nastavení adresy tohoto zařízení,
2. *maxTime* definuje maximální čas mezi datagramy při odesílání a příjmu dat. Čas je měřen v počtu volání funkce *cpr_SysTickFunc()*,
3. *maxRxCnt* definuje maximální počet pokusů o odeslání,
4. *maxNoAckCnt* definuje maximální počet datagramů bez potvrzení o přijetí,
5. **rtm_mem* obsahuje ukazatel na místo v paměti určené pro *RTM*.

Inicializace knihovny a vytvoření potřebných objektů pro časovač a uart může vypadat například takto:

```
#include <hardware_abstraction/drv/uartDriver.hpp>
#include <hardware_abstraction/hal/timer/tmr.hpp>
#include <rtm_communication/rtm_rtmonitor.h>
rtm_memory_t var_rtm_mem;
{
   constexpr uint16_t address = 0x01;
   constexpr uint16_t t_between_retransmits = 1000;
   constexpr uint16_t retransmit_count = 3;
   constexpr uint16_t messages_wo_ack = 2;

   rtm_Init(address, t_between_retransmits, retransmit_count, messages_wo_ack,
       &var_rtm_mem);
}

using hal::disc1::Timer;
using hal::disc1::uart4;

Timer rtm_tick(10);
UartDriver uart4(uart4::getInstance());
rtm_dtgm_t var_dtgmTx;
```

Hodnoty parametrů funkce by mohly být uvedeny přímo při jejím volání. Místo toho jsem použil dočasné proměnné označené jako *constexpr* v bloku kódu. Tyto proměnné slouží ke zlepšení čitelnosti kódu. Díky označení constexpr nemají žádný vliv na běh programu, protože je kompilátor dosadí při kompilaci a díky použití v bloku kódu nejsou viditelné ve zbytku programu.

4.4.2 Registrace vzorových proměnných

Proměnné, které chceme sledovat, není nutné deklarovat nijak specifickým způsobem; deklarují se stejně, jako kterékoliv jiné proměnné. Proměnné můžeme kdykoliv po jejich deklaraci registrovat do knihovny RTM. Od této registrace budou jejich hodnoty odesílány prostřednictvím protokolu RTM. K registraci proměnných slouží rodina funkcí *vObs_registerVar[typ](...)*. Mezi sebou se liší jen tím, jaký datový typ proměnné jsou schopné registrovat (kdyby byl při psaní knihovny použit jazyk C++, mohly by se všechny funkce díky přetěžování jmenovat stejně). Pro registraci proměnné typu *uint16_t* se používá funkce *vObs_registerVarUint16*. Všechny funkce z této rodiny mají následující parametry:

1. **address_p* obsahuje ukazatel na registrovanou proměnnou,
2. **name_p* obsahuje název registrované proměnné, který se bude zobrazovat v programu na sledovacím počítači,
3. **descr_p* obsahuje popis registrované proměnné,
4. **units_p* obsahuje jednotku registrované proměnné,
5. *min, max* a *gain* definují minimální a maximální hodnotu proměnné a zisk.
6. *vObs_access_t* definuje úroveň oprávnění nutnou k práci s touto proměnnou.

V následující ukázce kódu je vytvořena proměnná typu *uint16_t* a je zaregistrovaná v protokolu RTM:

```
uint16_t RedLedOn = 1;
vObs_registerVarUint16(
   &RedLedOn,
   (char*) "Red LED",
   (char*) "Controls the red LED",
   (char*) "-",
   1,
   0,
   65535,
   VOBS_SERVICE
);
```

4.4.3 Hlavní programová smyčka

V hlavní programové smyčce dochází (kromě jiné práce, kterou program provádí) k periodickému získávání aktuálních hodnot proměnných a jejich odesílání prostřednictvím sériové linky. Získávat hodnoty proměnných při každém průběhy smyčky by bylo zbytečné; proto je použitá třída *Timer* popsaná dříve v textu a hodnoty se ukládají s definovanou periodou. Jednoduchá hlavní smyčka může vypadat například takto:

```
while (1) {
   if (rtm_tick.tick()) {
      rtm_SmpTick();
      rtm_CprTick();
   }

   while (uart4.sciGetNewDtgmFlag()) {
      uint16_t len;
      rtm_dtgm_t* var_dtgmRx;

      var_dtgmRx = (rtm_dtgm_t*) uart4.sciGetRxData(&len);
      if (len >= CPR_HEADER_BYTE_LEN && len <= CPR_DATAGRAM_BYTE_LEN) {
         rtm_SaveDtgmFromUSB(var_dtgmRx);
         uart4.sciClrNewDtgmFlag();
      }
   }

   if (uart4.sciGetFreeToTxFlag()) {
      rtm_dtgm_t var_dtgmTx;
      uint16_t var_dtgmLen = rtm_LoadDtgmForUSB(var_dtgmTx);
      if (var_dtgmLen > 0) {
         uart4.sciTransmit(var_dtgmTx.buff_a, var_dtgmLen);
      }
   }
}
```

Kód, který zde byl popsaný již stačí k otestování základní funkce aplikace a může být použitý jako základ pro vývoj jiného programu. V testovacím programu, který je přiložený k této práci, je vytvořeno o jednu proměnou více a jsou použité k ovládání LED na přípravku.

4.5 Použití knihovny RTM ve vlastním projektu

Při použití knihovny ve vlastním projektu lze postupovat dvojím způsobem. Při vytváření nového projektu by bylo nejjednodušší použít projekt minimální aplikace, který je přiložený k této práci. Ten je již nakonfigurovaný pro použití s přípravkem STM32F429I-DISC1. Pro začátek práce stačí pouze tento projekt otevřít v Atollic studiu a rozšířit ho pro svoje účely.

Pro začlenění do již existujícího projektu stačí zkopírování adresářů *hardware_ abstraction* a *rtm_ communication* do adresáře projektu. Musejí být k dispozici také knihovny poskytnuté firmou ST k přípravku. Poté je nutné vložit do zdrojového kódu potřebné hlavičkové soubory a inicializovat knihovnu podobně, jako je ukázané v minimální aplikaci.

4.5.1 Adresářová struktura

```
rtm_port
├── Libraries ..... knihovny k přípravku od ST
├── Utilities ..... knihovny k přípravku od ST
└── src
    ├── hardware_abstraction ..... hal, ovladače, pomocné funkce
    ├── rtm_communication ..... knihovna RTM
    └── main.cpp ..... minimální aplikace
```

Kapitola 5

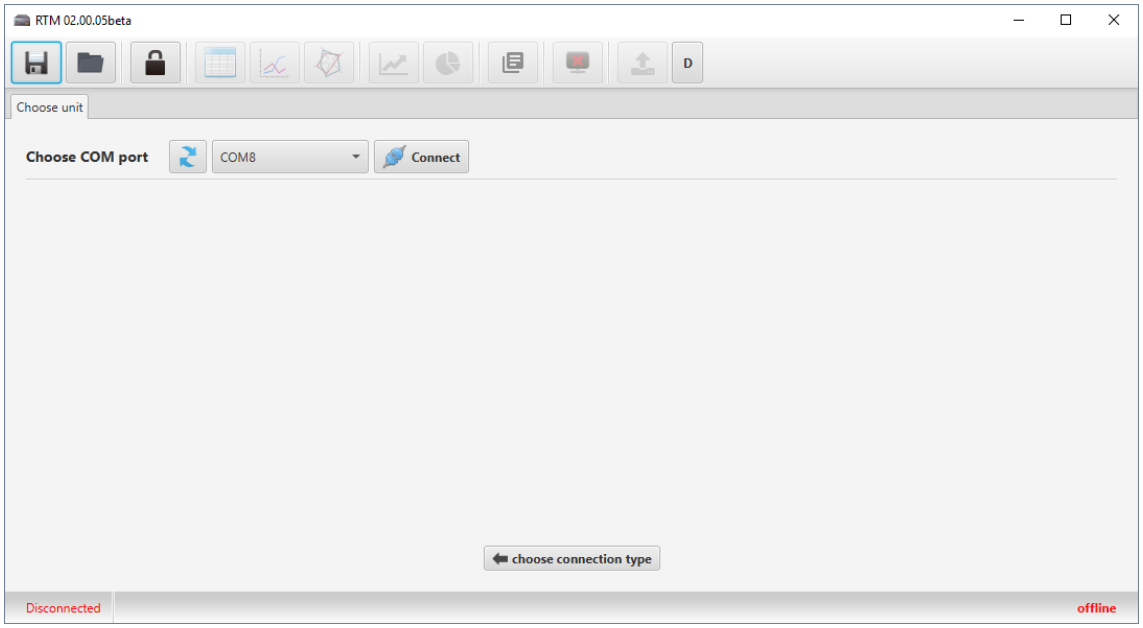
Ověření funkčnosti

K ověření funkčnosti jsem využíval minimální aplikaci, která byla popsaná v předchozí kapitole. Tu jsem pouze o něco rozšířil tak, aby bylo možné sledovat, jestli jsou načítané hodnoty proměnných správné a jestli komunikace funguje správně v obou směrech. Za tímto účelem jsou v aplikaci vytvořeny dvě číselné proměnné, které jsou zaregistrované v knihovně RTM. V rámci hlavní programové smyčky jsou na základě hodnot těchto proměnných ovládány LED.

5.1 Testování úspěšného navázání spojení

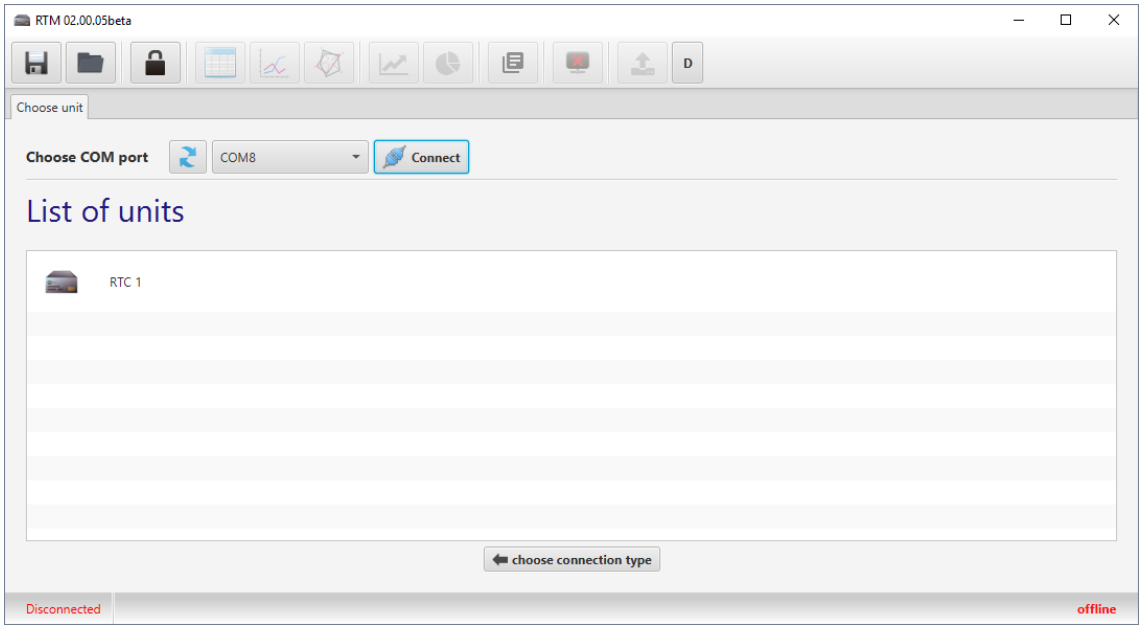
Prvním krokem bylo testování schopnosti navázání spojení s počítačem. Pro připojení k počítači jsem použil převodní kabel z *USB* na *RS232* od firmy FTDI. V rámci této práce jsem vytvořil HAL pouze pro *UART4*, který využívá piny *PC10* jako *TX* a *PC11* jako *RX*. Je nutné spojit *TX pin* převodního kabelu s *RX pinem* přípravku a *RX pin* převodního kabelu s *TX pinem* přípravku. Dále je nutné zem převodního kabelu se zemí přípravku. K tomu je možné využít jakéhokoliv pinu *GND*, je však vhodné zvolit co nejbližší.

Po připojení k počítači proběhla automatická instalace ovladačů převodního kabelu. Po jejím úspěšném ukončení jsem spustil aplikaci *RTM* a zvolil možnost spojení *RS232/USB*. Po této volbě se otevře obrazovka uvedená na obrázku 5.1. Na této obrazovce máme mož-



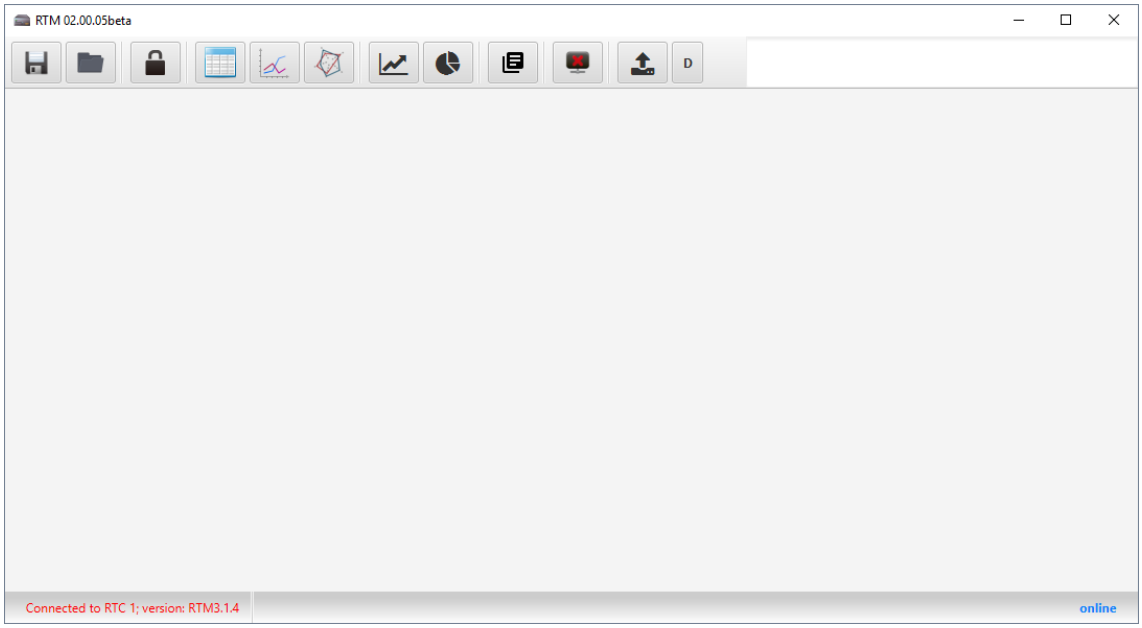
Obrázek 5.1: Volba COM portu, ke kterému je připojené zařízení s knihovnou RTM.

nost zvolit COM port, prostřednictvím kterého se chceme k zařízení připojit. Nezobrazují se zde všechny COM porty, ale pouze ty, které jsou přiřazené převodníkům firmy *FTDI*. Volbu COM portu potvrdíme tlačítkem *Connect*. Po jeho stisknutí dojde k pokusu o komunikaci s koncovým zařízením prostřednictvím vybrané sériové linky. Pokud je k ní připojeno nějaké zařízení podporující protokol *RTM*, bude zobrazeno. Na obrázku 5.2 je vidět jedno zařízení RTM. Komunikace tedy funguje. Kliknutím na zařízení se k němu připojíme.



Obrázek 5.2: Seznam zařízení připojených ke zvolenému portu.

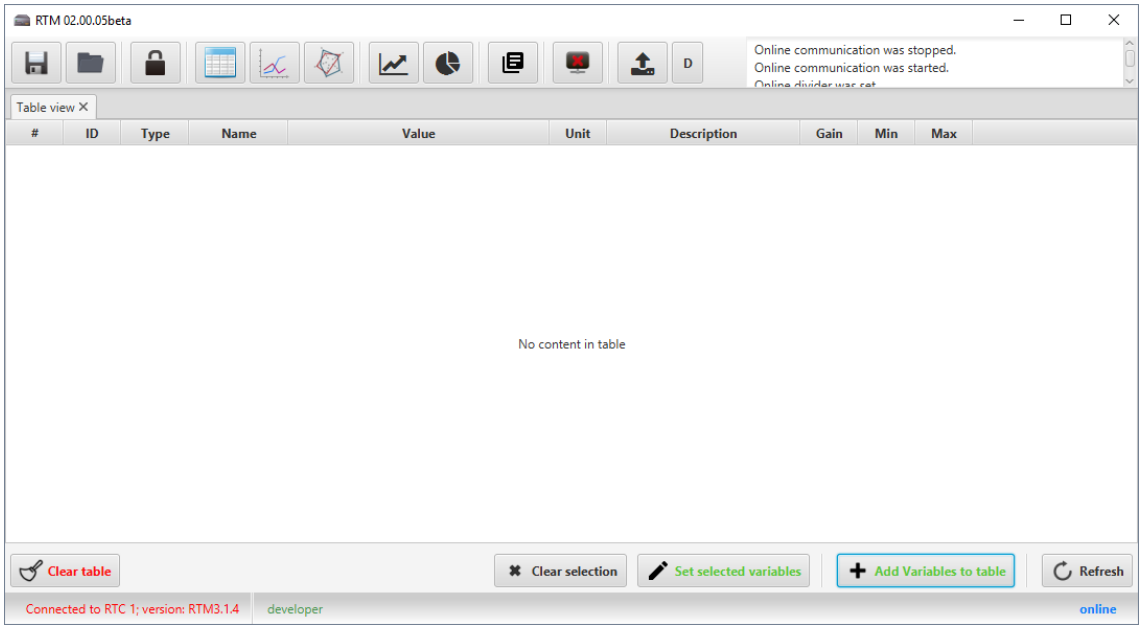
Po připojení k zařízení se otevře hlavní obrazovka aplikace RTM. Její snímek je na obrázku 5.3. Z té se můžeme dostat k dalším funkcím aplikace. Z nabízených funkcí jsou nejvýznamnější dvě; zobrazení dat v tabulce a v grafu. Zobrazení v tabulce poslouží k ověření obousměrné komunikace. Zobrazení dat v grafu poslouží k ověření schopnosti odesílat data v reálném čase a k ověření funkce off-line režimu.



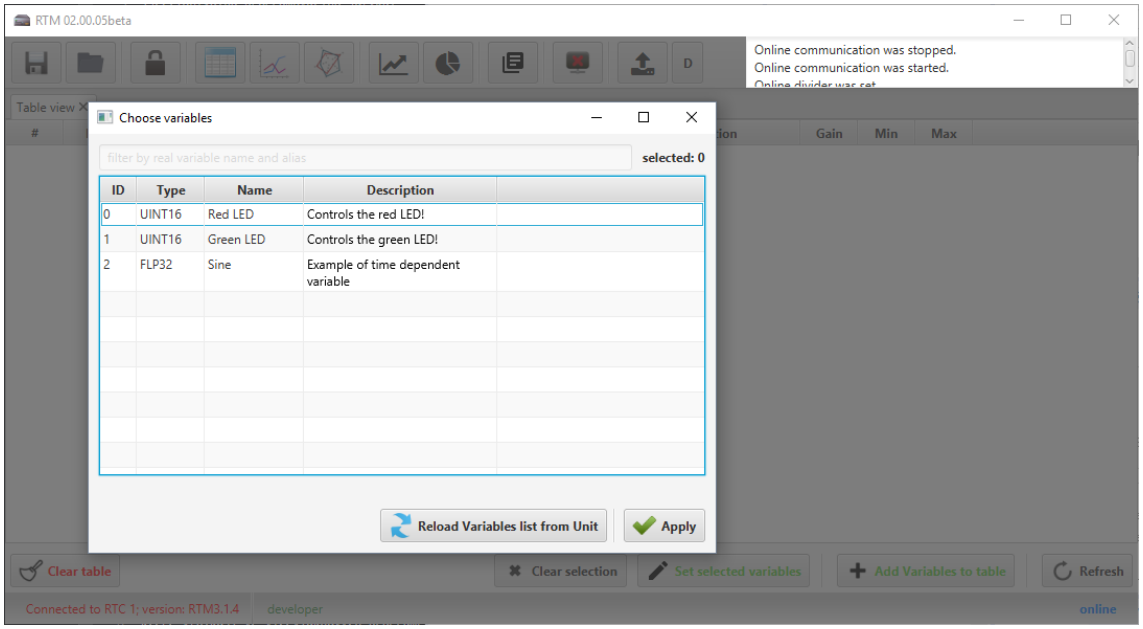
Obrázek 5.3: Hlavní obrazovka aplikace RTM po připojení k zařízení RTM.

5.2 Ověření získání dat do tabulky

Pro přepnutí aplikace do tabulkového režimu slouží tlačítko v nástrojovém panelu, které na sobě má symbol tabulky. Po jeho stisknutí se v aplikaci otevře nová záložka, která je zachycena na obrázku 5.4. Ta zatím neobsahuje žádné proměnné. Pro přidání proměnných do tabulky je nutné stisknout tlačítko „Add Variables to table“ v pravém dolním rohu aplikace. Otevře se dialogové okno, ve kterém je seznam všech proměnných, které jsou přístupné pro sledování a úpravy. Tento seznam se může lišit v závislosti na tom, jestli je uživatel přihlášen do aplikace jako vývojář. V tomto okně je možné vybrat proměnné ke sledování. Pro výběr více proměnných je nutné držet klávesu *ctrl*. Výběr se potvrdí tlačítkem „apply“ v pravém dolním rohu okna.

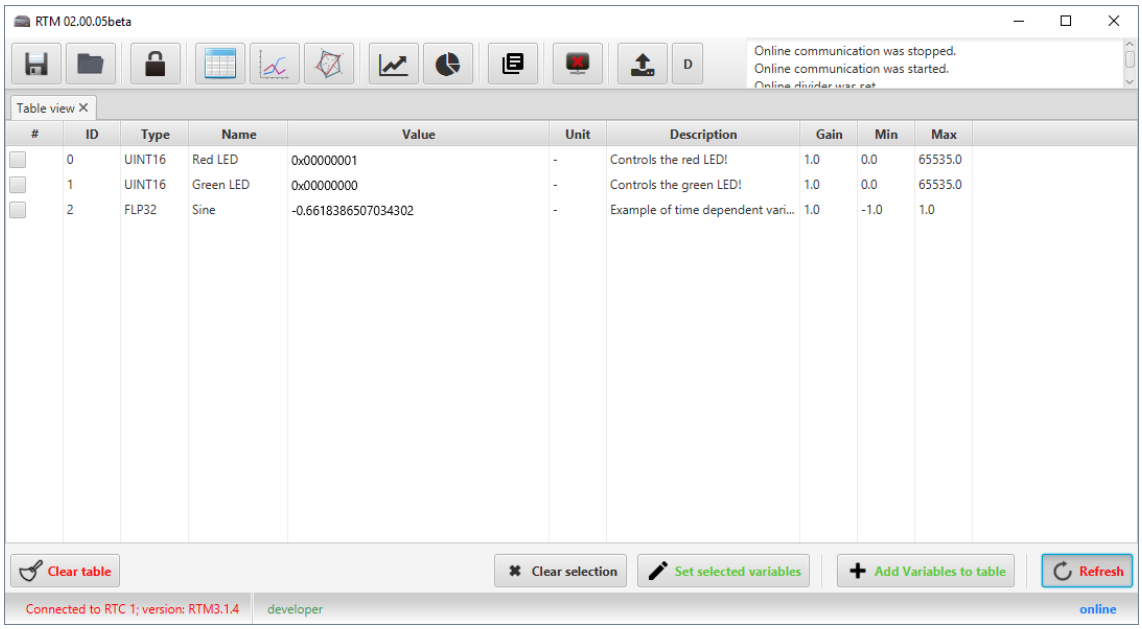


Obrázek 5.4: Tabulkové zobrazení před přidáním proměnných.



Obrázek 5.5: Dialogové okno pro přidání proměnných do tabulky.

Přidané proměnné se zobrazí v hlavním okně včetně doplňkových informací (jednotka, popis, maximum, minium a zisk). Zatím se však nezobrazuje jejich aktuální hodnota. Pro zobrazení aktuálních hodnot je potřeba přepnout tlačítko „Refresh“ v pravém dolním rohu aplikace; po jeho stisknutí se začnou periodicky hodnoty proměnných aktualizovat. Jeho opětovným stlačením je možné aktualizaci hodnot znovu zastavit. Ukázka tabulky s proměnnými je na obrázku 5.6. Jak je zřejmé z obrázku, přenos hodnot ze zařízení je funkční.



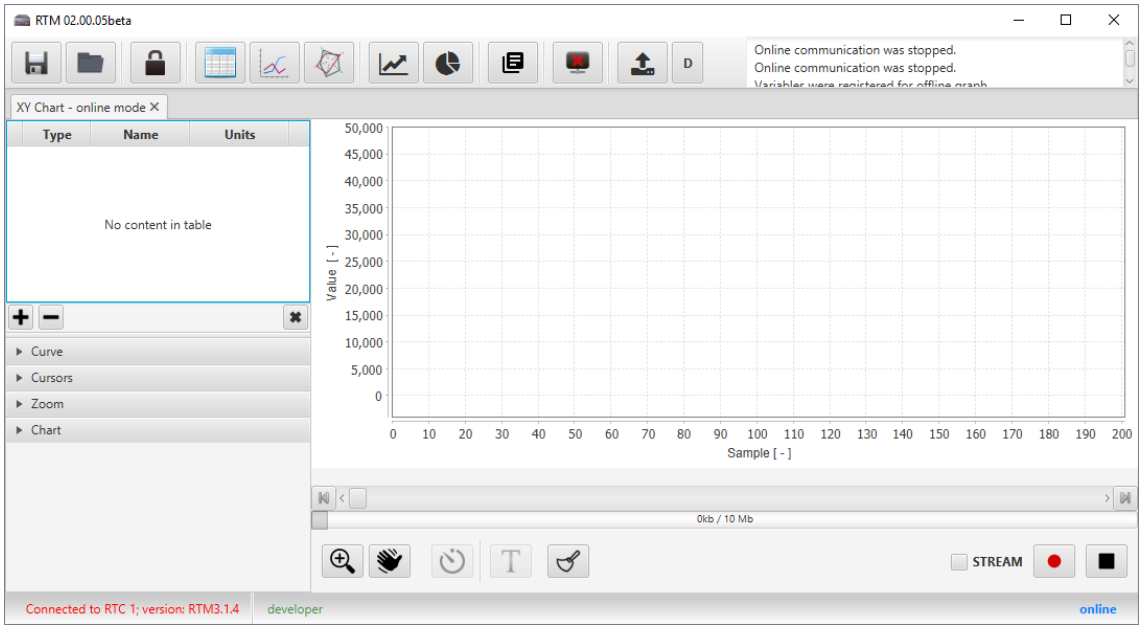
Obrázek 5.6: Tabulkové zobrazení se třemi proměnnými a aktivní aktualizací dat.

5.3 Ověření vykreslení dat do grafu

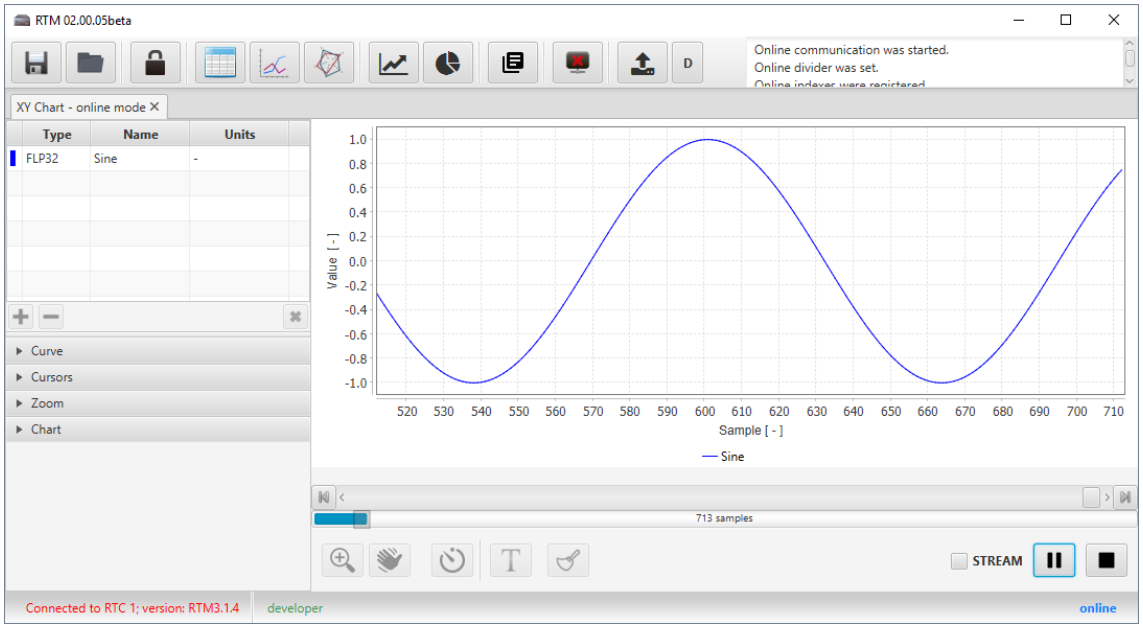
Pro přepnutí aplikace RTM do režimu grafu slouží tlačítko se symbolem grafu napravo od tlačítka tabulkového režimu. Slouží k zobrazení časových průběhů hodnot proměnných. Po jeho stisknutí se otevře nový panel s rozhraním pro práci s grafy (viz obrázek 5.7). Většinu místa v něm zabírá prostor pro graf. V levé části je možné přidávat sledované proměnné; k tomu slouží tlačítko se symbolem *plus*. Po jeho stisku se otevře stejné dialogové okno, jako když se přidávají proměnné v tabulkovém režimu.

Po přidání žádaných proměnných je možné spustit záznam. K tomu slouží tlačítko v pravém dolním rohu aplikace se symbolem nahrávání (červená tečka). Po jeho stisku se začne vykreslovat graf. Záznam grafu je možné kdykoliv zastavit stiskem tlačítka „pauza“ nebo „stop“. Délka záznamu není ničím omezena. V tomto případě je v mikropočítači generován sinusový průběh proměnné. Jak je vidět z obrázku 5.8, přenos dat funguje správně.

Obdobným způsobem by bylo možné otestovat záznam polárního grafu. Protože princip přenosu dat je stejný a rozdíl je jen na straně aplikace, bylo by testování zbytečné.



Obrázek 5.7: Záložka pro zobrazení grafu bez přidaných proměnných.



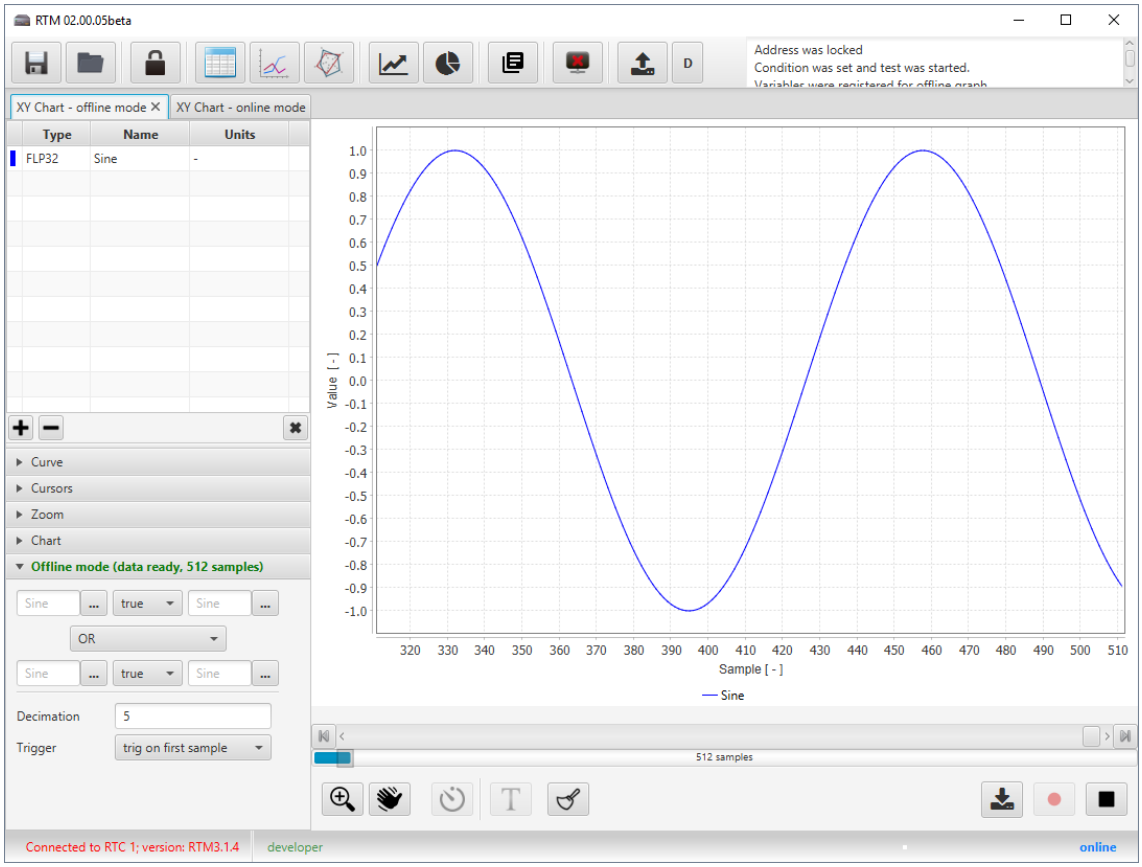
Obrázek 5.8: Zobrazení časového průběhu sledované proměnné.

5.4 Ověření vykreslení dat do grafu v off-line režimu

Do režimu off-line grafu je možné aplikaci přepnout pomocí sedmého tlačítka nástrojové lišty. Stejně jako režim grafu popsaný v předchozí sekci slouží k zobrazení časových průběhů hodnot proměnných. V tomto případě však nedochází k přenosu dat z přípravku v reálném čase; místo toho jsou ukládána do vnitřního bufferu a jsou přenesena až po jeho naplnění.

Proces přidání proměnných k zobrazení je totožný jako u online grafu a nebudu ho znovu popisovat. U offline grafu je možné odložit spuštění záznamu do doby, kdy bude splněná nastavená podmínka; ta se v současné verzi software skládá z výrazu, který obsahuje dvě srovnání. Příklad nastavení podmínky je vidět v levé dolní části obrázku 5.9. Takto nastavená podmínka bude vždy splněna a ke spuštění záznamu dojde ihned po stisknutí tlačítka se symbolem nahrávání.

Po naplnění bufferu přípravku dojde k automatickému zastavení záznamu. Pro přenesení dat do sledovacího PC a jejich zobrazení je nutné stisknout tlačítko *download*, které se nachází nalevo od tlačítka se symbolem nahrávání. Jak je vidět z obrázku 5.9, přenos dat v off-line režimu funguje správně.



Obrázek 5.9: Zobrazení časového průběhu sledované proměnné v offline režimu.

Kapitola 6

Závěr

6.1 Seznámení s RTM a s přípravkem STM32F429I-DISC1

V rámci této práce jsem se seznámil s knihovnou RTM a s principem její funkce, dále s přípravkem STM32F429I-DISC1 a s jeho možnostmi. Seznámení se s knihovnou i s přípravkem bylo nezbytné, abych mohl provést úpravy potřebné pro zprovoznění knihovny na tomto přípravku.

6.2 Návrh potřebné abstrakční vrstvy a úpravy zdrojového kódu

Po analýze zdrojového kódu jsem z existující abstrakční vrstvy *HAL* a z existujících ovladačů stanovil jejich podmnožinu, která je nutná k minimální funkci knihovny. Jako minimální funkci označuji schopnost komunikace s počítačem a oboustranné vyměňování hodnot proměnných.

Vytvořil jsem abstrakční vrstvu *HAL* pro sériovou komunikaci, pro časovač a pro GPIO. Všechny nově vytvořené *HAL* jsem navrhl jako třídy v jazyce C++. Dále jsem vytvořil objektový „wrapper“ ovladače sériové komunikace. Díky využití objektově orientovaného programování jsem docílil dobrého oddělení jednotlivých částí programu od sebe navzájem. Ovladač sériové komunikace tak existuje pouze jeden, na rozdíl od původních dvou se stejnou implementací, a o použitém sériovém rozhraní se rozhodne až při inicializaci ovladače volbou konkrétní abstrakční vrstvy pro sériovou komunikaci.

Při psaní kódu jsem se snažil hlavně o uživatelskou přívětivost. Některé části abstrakční vrstvy by bylo možné optimalizovat pro výkon. Snadnost využití by tím však utrpěla.

6.3 Ověření funkčnosti

Pro ověření funkčnosti jsem vytvořil minimální aplikaci, která umožňovala otestování stanovené minimální funkcionality. Tou bylo úspěšné spojení s programem v PC, čtení dat z proměnných, zápis dat do proměnných a jejich vynesení do grafu.

Pomocí této aplikace jsem otestoval komunikaci. Všechny stanovené funkce bylo možné používat. Hodnoty proměnných lze na počítači jak zobrazovat, tak upravovat. Také je možné zobrazit data ve formě grafu.

6.4 Celkové zhodnocení

Vytvořil jsem port knihovny tak, jak bylo definováno v zadání. Je možné se spojit s aplikací RTM v počítači a obousměrně vyměňovat data.

V rámci úprav knihovny jsem provedl některé změny, které by mohly být implementovány i do původní knihovny. Takovou změnou je například podmíněné obklopení hlavičkových souborů pomocí *extern "c"*. Dále jsem vytvořil abstrakční vrstvu *HAL*, která není vázaná jen na knihovnu RTM, ale může být použitá i samostatně.

Následujícím krokem by mohlo být zprovoznění dalších funkcí knihovny RTM jako například aktualizace firmwaru zařízení prostřednictvím aplikace RTM; k tomu by však bylo nutné vytvoření vlastního bootloaderu.

Seznam literatury

1. *The RTM – User View*. 2015. Č. JC2015-TCU-0008b.
2. *ISO/IEC 9899:201x*. 2010. Č. N1548.
3. ZDENEK, J.; HAJEK, J.; BREJCHA, M.; KUENZEL, K. *Getting Started – RTM Tool*. 2017. Č. JC2016-TCU-0019h.
4. *Discovery kit with STM32F429ZI MCU*. 2017.
5. *STM32F429 Discovery board*. Dostupné také z: `https://www.st.com`.
6. *GNU Arm Embedded Toolchain*. Dostupné také z: `https://developer.arm.com/open-source/gnu-toolchain/gnu-rm`.
7. *MDK Microcontroller Development Kit*. Dostupné také z: `http://www2.keil.com/mdk5/`.
8. *TrueSTDUIO for STM32*. Dostupné také z: `https://atollic.com/`.
9. *C++ reference*. Dostupné také z: `https://en.cppreference.com`.
10. MEYERS, S. *Effective Modern C++*. 2017.
11. GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.

Přílohy

Příloha A: Obsah přiloženého CD

Text práce Tato složka obsahuje zdrojové soubory sázecího systému LaTeX k této diplomové práci.

Vzorový projekt.zip Tato komprimovaná složka obsahuje zdrojové soubory vzorového projektu. Obsahuje projekt pro Atollic TrueSTUDIO.