VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

INTEGRACE POMOCNÝCH MODULŮ DO SYSTÉMU PRO AUTOMATIZOVANÉ FUNKČNÍ TESTOVÁNÍ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

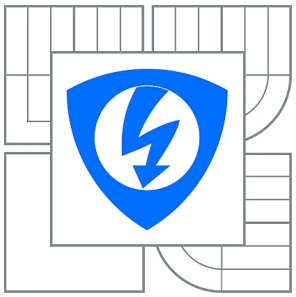
AUTOR PRÁCE
AUTHOR

ZDENĚK HŘEBÍČEK

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF TELECOMMUNICATIONS

INTEGRACE POMOCNÝCH MODULŮ DO SYSTÉMU PRO AUTOMATIZOVANÉ FUNKČNÍ TESTOVÁNÍ

INTEGRATION OF AUXILIARY MODULES INTO AUTOMATED FUNCTIONAL TEST SYSTEM

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

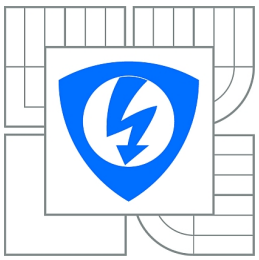
AUTOR PRÁCE — ZDENĚK HŘEBÍČEK

AUTHOR

VEDOUCÍ PRÁCE — Ing. RADOMÍR SVOBODA, Ph.D.

SUPERVISOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Bakalářská práce

bakalářský studijní obor
Teleinformatika

Student: Zdeněk Hřebíček ***ID:*** 147683
Ročník: 3 ***Akademický rok:*** 2013/2014

NÁZEV TÉMATU:

Integrace pomocných modulů do systému pro automatizované funkční testování

POKYNY PRO VYPRACOVÁNÍ:

Za účelem začlenění již existujících jednoúčelových testovacích modulů vytvořte ovladač v jazyce C# v prostředí .NET pro vybraný modul. Úkolem těchto ovladačů je komunikace s testovacími moduly přes sériovou linku, USB nebo TCP/IP, implementace metod definovaného API, které překládají univerzální testovací příkazy do protokolu příslušného testovacího modulu.

DOPORUČENÁ LITERATURA:

[1] KANER, C. – BACH, J. – PETTICHORD, B.: Lessons Learned in Software Testing. John Wiley & Sons, Inc., New York, 2011, ISBN 9780471081128
[2] Honeywell Enviracom Protocol, Data Link Layer Specification. Version 1.10, 2012. Honeywell Confidential.
[3] Honeywell Enviracom Protocol, Application Layer Specification. Version 1.10, 2012. Honeywell Confidential.

Termín zadání: 10.2.2014 ***Termín odevzdání:*** 4.6.2014

Vedoucí práce: Ing. Radomír Svoboda, Ph.D.
Konzultanti bakalářské práce:

doc. Ing. Jiří Mišurec, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Záměrem této bakalářské práce, bylo získat teoretické základy v oblasti automatizace testování embedded zařízení, zejména při použití testovacích modulů, pochopit strukturu protokolu Modbus a Enviracom, architekturu systému pro podporu automatizovaného testování ATAB a tyto znalosti následně využít k naprogramování ovladačů pro testovací moduly NTC Simulator a EnviraCOM Serial Adapter v jazyce C#.

KLÍČOVÁ SLOVA

embedded zařízení, automatizace testování, Modbus, C#, .NET, testovací moduly, ovladače, NTC Simulator, EnviraCOM Serial Adapter

ABSTRACT

The aim of this bachelor thesis was gain of theoretical background in the field of embedded device testing especially when using test modules, to understand the structure of Modbus and Enviracom protocols, architecture of system for automated testing ATAB and use that knowledge for programing drivers for test modules NTC Simulator and EnviraCOM Serial Adapter in C# language.

KEYWORDS

embedded devices, test automation, Modbus, C#, .NET, testing modules, drivers, NTC Simulator, EnviraCOM Serial Adapter

HŘEBÍČEK, Zdeněk *Integrace pomocných modulů do systému pro automatizované funkční testování*: bakalářská práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2014. 52 s. Vedoucí práce byl Ing. Radomír Svoboda, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Integrace pomocných modulů do systému pro automatizované funkční testování“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

(podpis autora)

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce panu Ing. Radomíru Svobodovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. A dále všem kteří mi při práci pomáhali a byli mi v době její tvorby oporou.

Brno ....

(podpis autora)

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

Cílem této bakalářské práce bylo získat dostatečné teoretické znalosti v problematice automatizovaného testování embedded zařízení, seznámit se s již existujícími systémy pro podporu automatizovaného testování, především však se systémem ATAB používaným ve vývojovém centru Honeywell ACS v Brně, a srovnat tyto systémy z hlediska požadavků ACS Brno. Dále bylo zapotřebí detailně nastudovat architekturu systému ATAB, jeho jednotlivé softwarové a hardwarové části a rozhraní mezi nimi. S tím souvisí také zvládnutí vývojového prostředí Visual Studio, programovacího jazyka C#, .NET frameworku, pochopení protokolu Modbus a Enviracom a jejich využití při komunikaci přes komunikační sběrnici RS-232, případně USB s testovacími moduly NTC simulator a EnviraCOM Serial Adapter.

Hlavním cílem této práce, byla ukázka zvládnutí výše zmíněných teoretických znalostí na příkladu ovladačů pro testovací modul NTC Simulator a EnviraCOM Serial Adapter.

1 EMBEDDED ZAŘÍZENÍ A JEJICH TESTOVÁNÍ

V této kapitole je popsáno co jsou to embedded zařízení, z čeho se skládají a pro co se používají. Dále též různé možnosti nebo důvody jejich testování a k čemu tato testování mohou sloužit.

1.1 Embedded zařízení

Embedded zařízení se většinou skládá z kombinace výpočetního hardwaru, softwaru a dalších součástí, které mohou být jak mechanické, tak elektronické. Tato zařízení jsou navržena tak, aby vykonávala jednu určitou funkci. Koncový uživatel může ovládat funkcionalitu zařízení a vybírat z jemu přístupných možností této funkcionality. Nemůže ale funkcionalitu měnit úpravou softwaru.

Pro přesnější pochopení definice embedded zařízení je dobré jej srovnat například s počítačem. U počítače není výrobcům předem zřejmé, k jakým účelům bude jednotlivými uživateli využíván a dodávají ho takzvaně čistý[1]. Je pouze na uživatelích daného počítače, jaký software si nainstalují a k čemu budou počítač používat. Naopak u embedded zařízení je vždy jedna, nebo více předurčených funkcí. Tuto funkci či funkce koncový uživatel nemůže změnit. Příkladem embedded zařízení je i počítačová klávesnice a myš. Každé takové zařízení obsahuje software i hardware a je navrženo pro jednu specifickou úlohu.

Častým příkladem využití embedded je spojení více zařízení do většího systému. Takovým systémem je například moderní automobil. Jedno zařízení se stará o brzdy, druhé sleduje a kontroluje emise, třetí zobrazuje informace na displeji umístěném na palubní desce.

Embedded zařízení vždy nemusí obsahovat dvojici procesor a software. Tato kombinace může být nahrazena vlastním integrovaným obvodem, který zajišťuje stejnou funkci. Využití procesoru a softwaru se ale ve většině situací jeví jako flexibilnější, levnější a méně náročné na spotřebu energie [1, 2].

1.2 Testování embedded zařízení

V nynější době se vyrábí mnoho embedded zařízení. Tato zařízení je třeba při vývoji a i při pozdější výrobě testovat. Důvodů, proč zařízení testovat, je mnoho. Zařízení se testují při vývoji například z důvodu možnosti zrychlení/zefektivnění

[1] Výrazem „čistý počítač“ označujeme skutečnost, kdy není předurčeno k čemu bude počítač sloužit.

vývoje a detekce skrytých chyb ještě před nasazením na trh. Díky průběžnému testování i po nasazení na trh je garantována správná funkčnost každého zařízení. A ve výsledku se netratí na výměně vadných kusů nebo celých sérií.

Prvním testem, který může být prováděn je test zahořovací. Při zahořovacím testu se zařízení jednoduše připojí k napájení a čekáme, jestli z některých součástek nezačne vycházet kouř. Zničené součástky je nutno podle situace buďto vyměnit nebo předimenzovat a vyměnit za jiné. Dalším testem používaným v začátcích vývoje je takzvaný smoke test (často se plete s výše zmíněným zahořovacím testem). Tak se označuje test, při kterém je vyzkoušena pouze hlavní funkce zařízení a neberou se v potaz jeho vedlejší funkce. Výhodou takového testu je jeho rychlost, velkou nevýhodou naopak neúplnost. Z tohoto důvodu se později přechází ke komplexnímu testování celého zařízení. Během vývoje můžeme zařízení testovat manuálně, pokud by však bylo později nutné testovat každý vyrobený kus, je pro tuto činnost efektivnější přejít k testování automatizovanému. Embedded zařízení se většinou testují metodou takzvaný *Black box testing* (popsanou v kapitole 1.2.3.).

Testování embedded zařízení je náročná činnost, která si žádá překonání problémů, jež se v testování běžných zařízeních nevyskytují. Mezi nedostatky které se mohou testováním embedded zařízení odhalit, patří výskyt chybné implementace logiky, matematiky a algoritmů. Dále také můžeme objevit chyby v běhu programu (větvení, smyčky), při inicializaci a přepínání do různých režimů [3, 4].

1.2.1 Manuální testování

První forma testování embedded zařízení která vznikla, je testování manuální. Proces spočívá, jak vyplývá již z názvu, v manuálním testování zařízení. Testováním se ověřuje funkčnost všech vlastností daného zařízení. Při testech je vyvíjena snaha simulovat roli koncového uživatele. Je nutné se co nejvíce přiblížit k tomu, jak se zařízení bude chovat v reálném provozu. Pro zajištění kompletního testu všech potřebných vlastností, je nutné dopředu vytvořit testovací plán. Dle takového plánu se následovně bude postupovat, proto musí zahrnovat všechny možné stavy, do jakých se zařízení může dostat.

Nevýhodou manuálního testování je časová náročnost a možnost chyby lidského faktoru. Naopak jeho výhodou je dostupnost, flexibilnost (není potřeba vše do detailu předem definovat, tester je schopen improvizace a rychlé reakce na nepředvídané situace) a finanční nenáročnost. Problém nastává ve chvíli, kdy je potřeba testovat velký počet zařízení. Manuální testování se pak stává velice neefektivním kvůli výše zmíněné časové náročnost a možným chybám [5, 6].

1.2.2 Automatizované testování

Nástupcem, nebo spíše doplňkem manuálního testování je testování automatizované. Důvod jeho vývoje a příchodu byla právě časová náročnosti manuálního testování, možnost chyby lidského faktoru a potřeba zefektivnění procesu vývoje zařízení.

Automatizované testování používá speciální software (separovaný od zařízení, které testujeme). Tento software je většinou spouštěn na takzvaném řídícím PC, na kterém testy můžeme jak spouštět, tak konfigurovat. Používá se k automatizování opakovaných, ale nutných testů, které se mohly již provádět manuálně. A může také přidávat testy další, komplexnější, které by byly manuálně velmi časově náročné, obtížně proveditelné či přímo neproveditelné.

Při zavádění automatizovaného testování by měla být brána v potaz nákladnost jeho zavedení. Je potřeba zohlednit nutnost znalosti kódu a technického vzdělání u osoby která testování zavádí a v počátcích i provádí. Proto je nutné zvážit, jestli se časem vyplatí více než testování manuální. To většinou nastává například při rozšíření výroby, když je naše zařízení natolik komplexní, že je časová náročnost testování manuálního nepřekonatelnou překážkou, nebo je nutné zefektivnit vývoj například kvůli konkurenčnímu boji. V případech, kdy se zavádí automatizované testování, zpravidla jej zpočátku používáme ve stejné míře jako testování manuální; až postupem času testování téměř plně automatizujeme. Postupně lze testy upravit tak, že jejich provedení je natolik srozumitelné a jednoduché, aby jej mohl provádět i laik, a tím se nám automatizace testování vyplatí ještě více.

Při nasazení automatizovaného testování již při vývoji zařízení můžeme vývoj urychlit, a to zejména díky rychlejšímu vyhledání chyb. Další výhodou je eliminace chyb lidského faktoru při samotném testování, vyvstává ale možnost stejné chyby při sestavování či programování testů. Chybu, která nastane při sestavování či programování testů lze ale jednou pro vždy opravit. Nevýhodou může být finanční náročnost automatizace testování, zvláště v případě, kdy nasadíme automatizované testování tam, kde se stále ještě více vyplatí testování manuální. Proto je vždy nutné provést důkladnou rozvahu před jeho zaváděním.

Automatizované testování bychom neměli brát jako náhradu testování manuálního, v praxi je zapotřebí kombinace obou druhů testování. Protože testování není pouze sekvence opakovatelných testů, ale je zapotřebí provádět i testy manuální, kterými lze objevit takové chyby, které automatizovanými testy odhalit nemusí. Manuální testy totiž většinou provádí jedinci s unikátní znalostí testovaného zařízení a díky tomu mohou objevit chybu, kterou by i obšírněji napsané automatické testy nemusely objevit. Ve výsledku bude vždy žádané, aby testování automatizované bylo všude, kde je lze nasadit a tím pádem vysoce převyšovalo testování manuální [3, 5, 6, 7].

1.2.3 Black box testing

Často používaná metoda pro testování embedded zařízení je takzvaný black box testing[2]. Jedná se o metodu při které se zkoumá funkčnost určitého zařízení bez nutné znalosti jeho vnitřní struktury. Ověřuje se pouze jeho chování navenek, neboli co se stane, je-li přiveden například určitý signál na jeho vstup. Očekává se určité chování výstupu. Chování výstupů vůči vstupům daného zařízení je známo, ale vnitřní struktura zařízení a implementace určitých funkcí ne. Z tohoto známého chování, lze potom vyvodit testovací plán a ten následně aplikovat. Na obrázku 1.1 je zobrazeno blokové schéma této testovací metody.

Obr. 1.1: Blokové schéma metody testování black box testing.

Dalšími metodami testování jsou white box testing[3] a gray box testing[4]. White a gray box testing se používá také k testování embedded systémů. Pomocí emulátorů je možné krokovat program přímo v procesoru v embedded zařízení (takzvaný in circuit testing), zastavovat ho, kontrolovat jeho běh, hodnoty v paměti, měnit paměť za účelem urychlení testů (například vyvolat přechod do stavu, na který by bylo jinak nutné čekat hodiny nebo dny) [5, 8].

[2]Black box testing – v českém překladu testování černé skříňky.

[3]White box testing – v českém překladu testování bílé skříňky, občas se používá také alternativní označení glass - skleněné, nebo transparent - průhledné.

[4]Gray box testing – v českém překladu testování šedé skříňky

2 SYSTÉMY PRO AUTOMATIZOVANÉ TESTOVÁNÍ

V dnešní době je automatizované testování žádáno u široké škály embedded zařízení. Testovat je potřeba jak při vývoji, tak při samotné výrobě, jak už bylo řečeno v kapitole 1.2. Proto vznikají celé systémy pro automatizované testování.

Pro příklad jsou zde uvedeny některé vybrané produkty, které spolu nebo samostatně tvoří systémy pro automatizované testování:

- TestStand, LabView a PXI od National Instruments,
- zařízení pro testování flexLAB® od společnosti SOMA,
- software pro automatizované testy flexTAC® od společnosti SOMA,
- kompletní testovací stanice od společnosti Chroma,
- TestExec SL software pro automatizované testy spolu s moduly a celé testovací stanice od společnosti Agilent.

Testovací systémy jsou nejčastěji řešeny jednou ze dvou forem, které jsou zde popsány. Hlavní částí první z nich je testovací stanice, která obstarává vše potřebné pro testování. Na vstupech testovaného zařízení simuluje určitý signál, odpor, nebo logický stav obvodu a zároveň sbírá data o stavech výstupů zařízení. Přijímaná data z výstupů potom určitou formou zpracovává nebo pouze zaznamenává. Druhou formou testovacích systémů je řídící PC s testovacím softwarem. Tento software slouží pro tvorbu a následný běh testů. K tomuto PC jsou připojovány hardwarové moduly pro testování a sběr dat, pomocí sériové linky, USB, nebo Ethernetu. O zpracování dat a řízení testů se potom stará software na daném PC. Moduly pro testování nejsou pouze záležitostí systémů s řídícím PC, ale existují i rozšiřující moduly pro testovací stanice.

Mezi hlavní výhody těch to systémů patří:

- jejich komplexnost,
- možnost rozšíření systému o potřebné moduly,
- jednoduchá tvorba testů,
- jejich snadné nasazení.

Při nasazování systémů pro automatizované testování by se měl ten, kdo tyto testy nasazuje, vyvarovat určitým věcem. Neměl by vkládat přehnanou důvěru v komerční testovací nástroje. Tyto nástroje poskytují většinou schopnost nahrát a spouštět zaznamenaný manuální test. To může později vést k tomu, že tester spouští test bez toho, aby věděl, co přesně test kontroluje. Tímto se výsledky testů mohou stát neurčitými a nesnadno zpracovatelnými. Další věc, které je dobré se vyvarovat, pokud firma nedisponuje dostatkem prostředků, je licencovaný software. Zde může nastat problém s přístupem k testům a jejich výsledkům. Tato situace nastává kvůli

nedostatku licencí a může vést ke snížení kvality spolupráce v týmu. Této situaci lze předejít například uchováváním testovacích plánů a výsledků mimo testovací nástroj. Zde ale nastává problém s exportováním a importováním těchto dat, a přináší to do zpracování výsledků nadbytečnou složitost. Jako alternativu lze použít otevřené systémy pro testování. Pomocí těchto systémů můžeme uskutečnit většinu testů, které bychom prováděli přes placený testovací nástroj. Existuje ještě jedna poslední možnost a tou je vývoj vlastního testovacího systému.

Zavedení automatizovaného testování, by nemělo být považováno pouze za prostředek ke snížení nákladů. Prodejci testovacích nástrojů uvádí výpočty návratu z investice do jejich nástroje a tento výpočet nebývá pravdivý. A to kvůli tomu, že nezahrnuje potřebu převzetí praktik spojených s používáním jejich nástroje a průběžné náklady na jeho údržbu. Automatizace testování by měla být spíše prostředkem pro zefektivnění a urychlení práce.

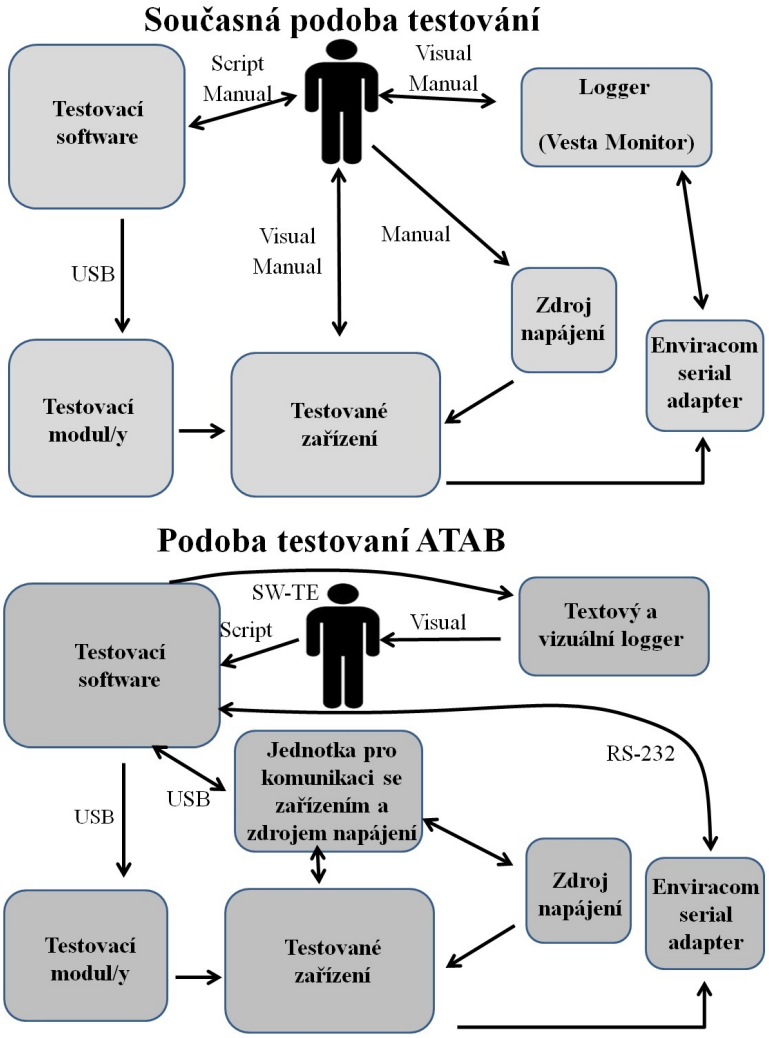
Ve vývojovém centru Honeywell ACS v Brně byla vybrána možnost vývoje vlastního testovacího systému; důvodem byla vysoká pořizovací a udržovací cena již vyvinutých systémů pro automatizované testování. Pak také to, že nemají potřebu tak komplexních testů, jaké poskytují již vyvinuté systémy a to, že vývojáři embedded zařízení mohou přímo testy vyvíjet nebo na nich spolupracovat. Tato práce v týmu může zvyšovat efektivitu a rychlost vývoje produktů. Posledním důvodem bylo, že již existovala řada přípravků (modulů) pro testování, které byly vyvinuty přímo v Brně za účelem podpory testování a jejich integrace do uzavřených komerčních systému by nebyla snadná. [9, 10, 11, 12, 13].

2.1 Popis systému pro automatizované testování ATAB

Systém ATAB[1], je aktuálně vyvíjený systém pro automatizované testování ve vývojovém centru Honeywell ACS v Brně. Tento systém se vyvíjí z důvodu možnosti zlepšení opakovatelonsti testů, jejich zpřesnění a hlavně pro redukci času potřebného na vývoj jednotlivých produktů. Dosavadní systém testování je téměř zcela manuální a je zapotřebí, aby u něj byl po celou dobu tester. Doposud musel tester manuálně nastavovat napájení, ovládat program pro řízení hardwarových modulů, a zpracovávat data z logeru. Za použití systému ATAB by měl tester napsat skript pro testovací sekvenci, samotné řízení testování a zpracování dat zajistí testovací software. Tento software bude komunikovat s testovacími moduly, výstupy zařízení a zdrojem napájení pomocí předdefinovaných rozhraní. Na obrázku 2.1 je zobrazeno

[1] ATAB – Automated Testing in ACS Brno, v překladu: automatizované testování v ACS Brno.

srovnání schémat původní podoby testování s podobou systému pro automatizované funkční testování ATAB.

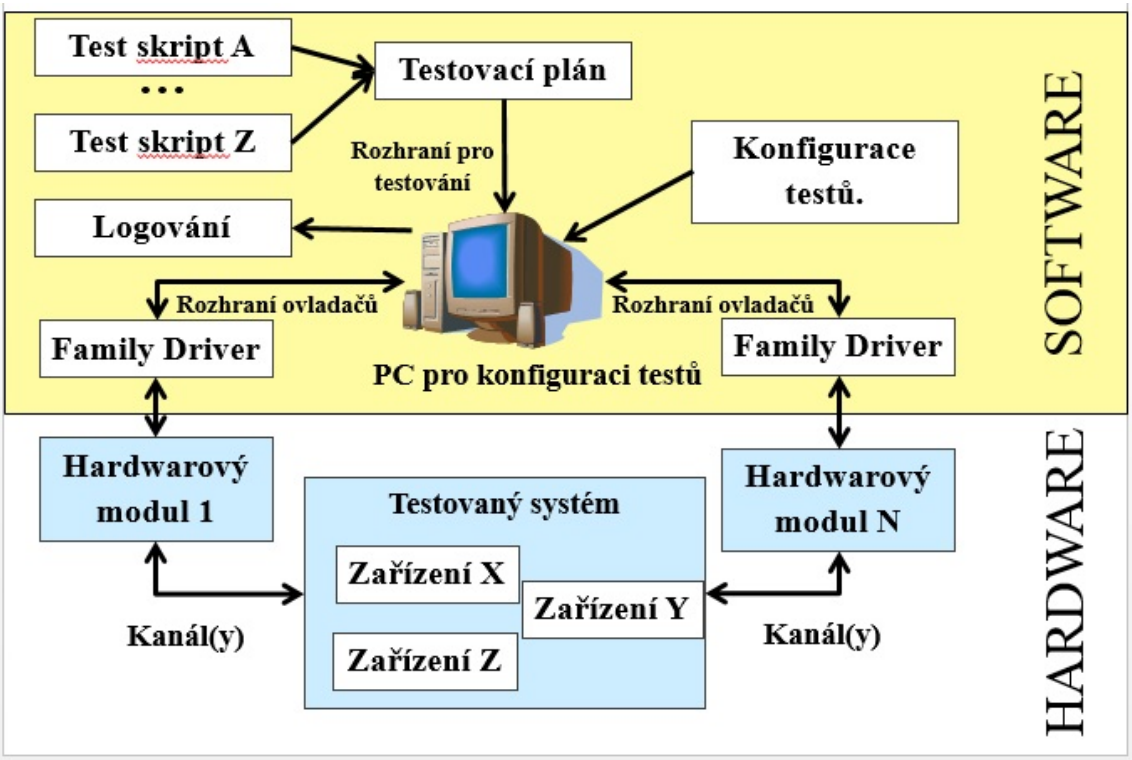


Obr. 2.1: Srovnání původní podoby testování se systémem ATAB [14].

Ústředním bodem architektury systému ATAB je PC se softwarem pro běh a konfiguraci testů. Tento software komunikuje pomocí rozhraní s ovladači testovacích modulů. Prostřednictvím ovladačů využívá dané moduly pro testování. Přes tyto rozhraní nejen ovládá jednotlivé moduly, ale také získává a zpracovává informace o stavu jejich výstupů. Na obrázku 2.2 je zobrazeno blokové schéma architektury celého systému, jak po stránce hardwaru, tak i softwaru. Tato bakalářská práce se zaměřuje právě na programování ovladačů (spadajících do jedné rodiny: Family Driver[2]). Tyto ovladače budou sloužit pro komunikaci s testovacímy moduly, specificky

[2]Family Driver – ovladače spadající do jedné „rodiny“ určené pro komunikaci s testovacímy moduly.

s těmi, které budou simulovat určité hodnoty[3] na vstupech testovaných zařízení [14].



Obr. 2.2: Architektura systému ATAB [14].

2.1.1 Rozhraní pro komunikaci řídícího software s ovladači

Pro zajištění správného zpracování a následného použití dat v předem definovaných metodách se v systému ATAB využívá rozhraní. Tato rozhraní jsou implementována v kódu jako interface[4]. Jedná se o prostředek pro definici struktury metod, které budou implementovány v jednotlivých ovladačích. Jedno rozhraní bude implementováno ve všech ovladačích, ostatní jsou specifické pro jednotlivé ovladače. Stěžejním rozhraním pro všechny ovladače je rozhraní `IHwDriver`. Metody definované tímto rozhraním budou implementovány v každém ovladači. Naopak například rozhraní `IOutputResistance`, bude implementováno pouze pro ovladač testovacího modulu NTC Simulator. Potřeba využití rozhraní je dána tím, že každá třída může dědit pouze z jedné jiné třídy, ale může implementovat libovolný počet rozhraní. Rozhraní, která jsou definována pro použití v systému ATAB je mnoho, jejich výčet a popis ale nespadá do rozsahu této práce [15].

[3]Hodnotami je myšlen odpor, určitý signál, stav logického obvodu.
[4]Interface je speciální rozhraní používané v programovacích jazycích a také v jazyce C#.

Rozhraní IHwDriver

Rozhraní `IHwDriver`, musí být implementováno každým ovladačem používaným v systému ATAB.

Toto rozhraní definuje tyto metody:

- `Init()` – pro inicializaci jednotlivých modulů,
- `InitChannel()` – pro inicializaci bufferů pro jednotlivé moduly,
- `GetInterfaces()` – pro zjištění dostupných rozhraní,
- `Close()` – pro vyčištění bufferů a uzavření spojení s moduly.

Definice těchto metod musí být vždy totožná, jak je předurčeno použití rozhraní, ale implementace mohou být odlišné. Kód tohoto rozhraní je uveden v příloze A.1. Z kódu vidíme, že pro volání těchto metod jsou nutné určité parametry.

Vstupní parametry rozhraní a jejich popis:

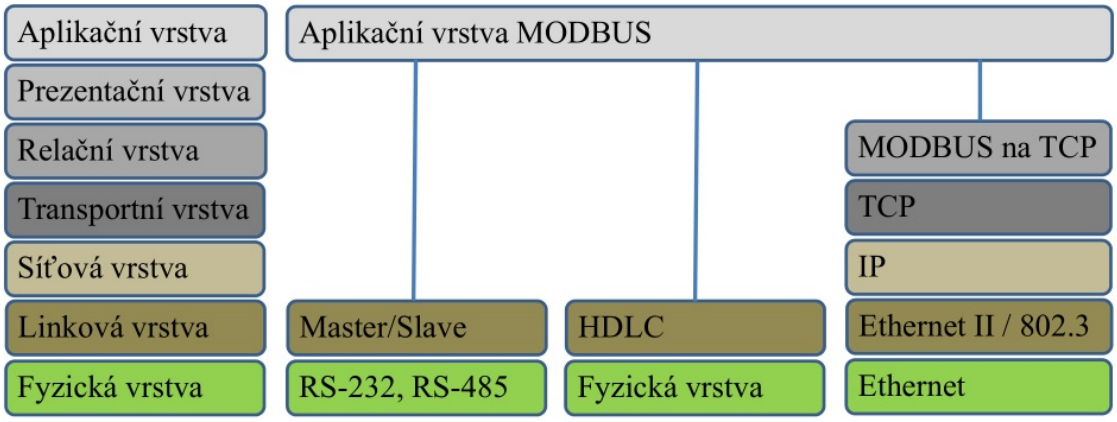
- `Int32` module – adresa modulů, ať už námi či předem daná,
- `Int32` channel – jednotlivé komunikační kanály modulů (moduly mohou mít jeden a více kanálů),
- `String` config – textové označení portu, do kterého je modul připojen (config má většinou tvar "COMN", kde N je číslo komunikačního portu).

3 PROTOKOL MODBUS

V této kapitole je popsán komunikační protokol Modbus, který se rozhodli ve vývojovém centru Honeywell ACS v Brně preferovat pro komunikaci s testovacími moduly. Tento protokol v roce 1979 vyvinula společnost Modicon (nyní Schneider Electric). Je to otevřený protokol používaný pro komunikaci přes sériovou linku, USB nebo TCP/IP. Používá se pro navázání spojení klient-server (master-slave) mezi inteligentními zařízeními. Díky jeho robustnosti a jednoduchosti se z něj stal standardní komunikační protokol a v současnosti nejvíce používaný síťový protokol užívaný pro komunikaci zařízení v průmyslové výrobě. Pomocí protokolu Modbus lze jednoduše propojit několik zařízení, které mohou být od odlišných výrobců. Modbus je v současné době možno použít i na TCP/IP sítích. Otevřená specifikace pro Modbus TCP/IP byla vyvinuta v roce 1999. Tento protokol také nebyl vždy volně přístupný, stalo se tomu tak až v dubnu roku 2004, kdy byl Modbus převeden z Schneider Electric na Modbus Organization. Kód protokolu byl volně zveřejněn zdarma pro stažení a neexistují žádné zpoplatněné licence nutné pro jeho použití [16].

3.1 Specifikace aplikačního protokolu Modbus

Aplikační protokol Modbus slouží pro komunikaci klient/server mezi zařízeními propojenými různý typy sběrnicí nebo sítí. Jedná se o protokol na bázi žádost/odpověď a je umístěný na sedmé vrstvě síťového modelu ISO/OSI.



Obr. 3.1: Model různých implementací protokolu Modbus [17].

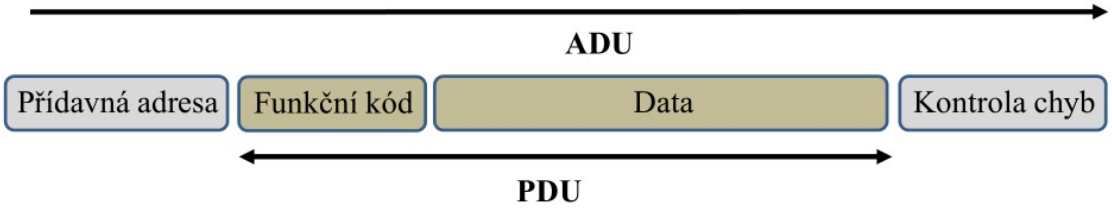
Aplikačního protokol Modbus je implementován pro tyto technologie přenosu:

- TCP/IP přes Ethernet (na portu 502),

- Asynchronní sériový přenos přes různá média (RS-232, RS-422, RS-485, optické vlákno, radiový přenos, atd.),
- Modbus plus - vysokorychlostní síť s předáváním tokenu.

Protokol Modbus poskytuje možnost jednoduché komunikace na různých typech sítí. Komunikace bude vypadat stejně jak na sériové lince, tak na sítí TCP/IP přes ethernet.

Modbus definuje PDU (jednoduchá datová jednotka – Protocol Data Unit) nezávisle na komunikaci mezi nižšími vrstvami. Při implementaci protokolu Modbus na specifickou sběrnici, nebo síť, je nutné vkládat do ADU (datová jednotka aplikační vrstvy – Aplication Data Unit) doplňující data.



Obr. 3.2: Základní rámec protokolu Modbus [17].

Jednotku aplikační vrstvy vytváří klient, který zahajuje komunikaci. Část rámce nazvaná `funkční kód` definuje operaci, kterou požadujeme po adresátovi daného rámce. Funkční kód má délku jednoho bajtu. Platný funkční kód musí být z rozsahu 1-127 (rozsah 128-255 je rezervován pro výjimky). K základním funkcím mohou být přidány podfunkční kódy, užívané pro definování dalších operací. Tyto kódy jsou umístěny v datové oblasti rámce. `Datová část` rámce obsahuje informace s upřesněním požadované operace. To může být například adresa, číslo registru, množství položek se kterými se operace bude provádět.

Jestliže proběhne přenos dat v pořádku, tak jsou v odpovědi zahrnuta požadovaná data. V poli funkčního kódu, bude kód operace, která byla provedena. Z toho vyplývá, že hodnota funkčního kódu v odpovědi je totožná s hodnotou při požadavku. Když při přenosu nastane chyba, odpověď bude obsahovat kód funkce s nejvyšším bitem, nastaveným na jedničku. Nejvyšší bit je identifikátor chyby. V datové části se potom nachází chybový kód upřesňující neúspěch.

Velikost PDU byla zděděna z první implementace protokolu Modbus po sériové lince RS-485, kde byla maximální velikost ADU 256 bajtů.
Maximální velikost PDU pro komunikaci po sériové lince = 256 - Adresa serveru (1 bajt) - CRC (2 bajty) = **253 bajtů**. Ve výsledku pak máme tyto velikosti aplikačních datových jednotek:
• RS-232 **ADU** = 253 + Adresa serveru (1 bajt) + CRC (2 bajty) = **256 bajtů**,
• TCP Modbus **ADU** = 253 bajtů + MBAP (7 bajtů) = **260 bajtů**.

Modbus definuje tři typy PDU. Jsou to:

- Požadavek (request)
  - Funkční kód = [1 bajt] funkční kód Modbusu.
  - Požadovaná data = [n bajtů] Pole, které závisí na funkčním kódu, většinou obsahuje informace, jako například proměnné a adresy, počty proměnných, přednastavená data, podfunkce.
- Odpověď (response)
  - Funkční kód = [1 bajt] funkční kód Modbusu.
  - Požadovaná data = [n bajtů] Pole, které závisí na funkčním kódu, většinou obsahuje informace, jako například proměnné a adresy, počty proměnných, přednastavená data, podfunkce.
- Vyjímka (exception)
  - Funkční kód = [1 bajt] Funkční kód Modbusu + 0x80.
  - Chybový kód = [1 bajt].

Pro kódování dat se využívá reprezentace Big-Endian. To znamená, že pokud je numerické vyjádření větší než jeden bajt, je nejdůležitější bajt poslán první. Pořadí odesílání všech bajtů je tedy inverzní. Například u čísla 0x1234 je první odeslaný bajt 0x12 potom 0x34.

Datový model protokolu, je založen na sérii tabulek, které mají odlišné charakteristiky. Čtyři hlavní tabulky jsou vypsány v tabulce 3.1:

Tab. 3.1: Datový model protokolu Modbus [17].

Primární tabulky	Typ objektu	Typ	Komentář
Diskrétní vstupy (Discrete inputs)	Single bit	Read-Only	Data poskytovaná I/O systémem.
Cívky (Coils)	Single bit	Read-Write	Data, která mohou být modifikovaná aplikačním programem.
Vstupní registry (Input registers)	16-bit word	Read-Only	Data poskytovaná I/O systémem.
Uchovávací registry (Holding registers)	16-bit word	Read-Write	Data, která mohou být modifikovaná aplikačním programem.

Adresování tabulek je závislé na konkrétním zařízení, každá z tabulek může mít maximálně 65536 položek. Kvůli zpětné kompatibilitě ale bývá rozdělena na bloky o velikosti 10000 položek.

Existují tři kategorie funkčních kódů. Jsou to:

- Veřejné funkční kódy
  - Obecně známé funkční kódy z rozsahu 1-64, 73- 99 a 111-127

  - mají garantovanou jedinečnost,
  - jsou schválené Modbus komunitou,
  - jsou veřejně zdokumentované,
  - mají dostupný test shody,
  - obsahují definované veřejně přiřazené funkce i nepřiřazené funkce rezervované pro pozdější využití.
- Uživatelem definované funkční kódy
  - existují dva rozsahy funkcí, které si může definovat sám uživatel, 65-72 a 100-110,
  - uživatel si muže vybrat a implementovat kód funkce který není podporován specifikací protokolu,
  - neexistuje žádná garance, že tento funkční kód bude unikátní,
  - když chce uživatel, aby se jeho funkce stala veřejnou, musí iniciovat RFC[1], aby představil svoji funkci veřejnosti, ta musí být schválena a zařazena mezi veřejné funkce,
  - Organizace Modbus, Inc. si výslovně rezervuje právo upravovat dané RFC.
- Rezervované funkční kódy
  - Funkční kódy které jsou používány některými společnostmi pro starší produkty a pro ty produkty, které nejsou pro veřejné použití.[17].

Přehled funkčních kóů je zobrazen na obrázku 3.4.

3.2 Specifikace protokolu Modbus na sériové lince

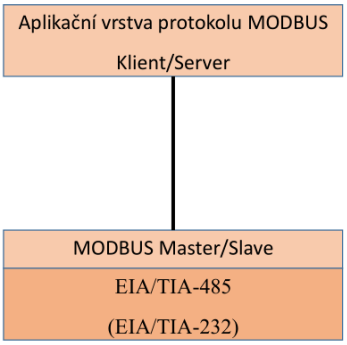
Modbus na sériové lince je master/slave protokol, tento protokol je umístěn na druhé vrstvě modelu ISO/OSI. Systém master/slave může obsahovat pouze jedno zařízení typu master. Toto zařízení posílá příkazy v jednom časovém úseku pouze na jedno zařízení typu slave a zpracovává odpovědi. Zařízení typu slave typicky nemůže vysílat žádná data bez požadavku od zařízení typu master a nemůže komunikovat s ostatními zařízeními typu slave. Na fyzické vrstvě může tento protokol používat různá rozhraní (RS-485, RS-232).

Aplikační protokol Modbus umístěný na 7. vrstvě modelu ISO/OSI poskytuje komunikaci klient/server mezi zařízeními propojenými sběrnicí nebo sítí. V protokolu Modbus na sériové lince je role klienta představována zařízením typu master a zařízení typu slave představuje server. Na obrázku 3.3 je zobrazeno srovnání protokolu Modbus s modelu ISO/OSI.

Klient může vysílat Modbus zprávy k serverům ve dvou módech.

[1]RFC – Request for Comments, V překladu žádost o komentáře.

Vrstva	Model ISO/OSI	
7	Aplikační	Aplikační protokol MODBUS
6	Prezentační	Prázdná vrstva
5	Relační	Prázdná vrstva
4	Transportní	Prázdná vrstva
3	Síťová	Prázdná vrstva
2	Datová	MODBUS na sériové lince
1	Fyzická	EIA/TIA-485 (EIA/TIA-232)



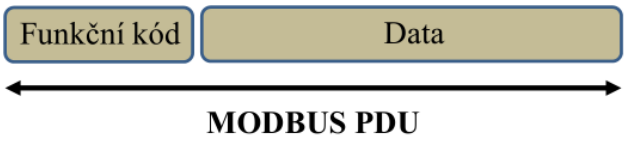
Obr. 3.3: Modbus na sériové lince a ISO/OSI [18].

- **Unicast** mód – klient adresuje pouze jeden server. Po obdržení a zpracování požadavku server pošle odpověď zpět ke klientovi. Používají se adresy 1 až 247.
- **Broadcast** mód – klient vyšle zprávu všem serverům. Server neposílá žádnou odpověď. Tento mód je nutný pro příkazy, které zapisují data do zařízení. Adresa pro broadcast je 0. Všechna zařízení musí akceptovat broadcast zprávu určenou pro zapisování.

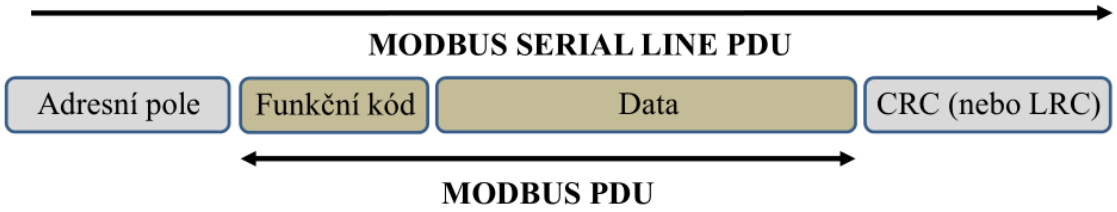
				Kódy funkcí			
				Kód	Podfunkce	hex	
Přístup k datům	**Bitový přístup**	Fyzické diskrétní vstupy	Čti diskrétní vstupy	**02**		**02**	
		Interní bity nebo fyzické cívky	Čti cívky	**01**		**01**	
			Zapiš jednu cívku	**05**		**05**	
			Zapiš více cívek	**15**		**0F**	
	16-bitový přístup	Fyzické vstupní registry	Čti vstupní registr	**04**		**04**	
		Interní registry nebo fyzické výstupní registry	Čti uchovávací registry	**03**		**03**	
			Zapiš jeden registr	**06**		**06**	
			Zapiš více registrů	**16**		**10**	
			Čti/zapiš více registrů	**23**		**17**	
			Zapiš registr s maskováním	**22**		**16**	
			Čti FIFO frontu	**24**		**18**	
	Přístup k záznamům v souborech		Čti záznam ze souboru	**20**	**6**	**14**	
			Zapiš záznam do souboru	**21**	**6**	**15**	
Diagnostika			Čti stav	**07**		**07**	
			Diagnostika	**08**	**00-18, 20**	**08**	
			Čti čítač kom. událostí	**11**		**0B**	
			Čti záznam kom. událostí	**12**		**0C**	
			Sděl identifikaci	**17**		**11**	
			Čti identifikaci zařízení	**43**	**14**	**2B**	
Ostatní			Zapouzdřený přenos CANOpen základní odkaz	**43** **43**	**13, 14** **13**	**2B** **2B**	

Obr. 3.4: Přehled veřejných funkčních kódů [19].

Klient nemá žádnou specifickou adresu, musí ji mít pouze servery. Při implementaci protokolu Modbus na specifickou sběrnici nebo síť musí být do PDU přidána některá další pole. Klient iniciuje komunikaci, vytvoří základní PDU a do něj přidá pole pro vytvoření potřebného PDU pro komunikaci.[18]



Obr. 3.5: PDU protokolu Modbus [18].



Obr. 3.6: Rámec protokolu Modbus na sériové lince [18].

3.2.1 Popis dvou módů pro komunikaci na sériové lince

Existují dva módy, pomocí kterých může komunikace na sériové lince probíhat. Jedná se o módy RTU a ASCII [18].

RTU mód

Při využití módu RTU, obsahuje každý bajt zprávy dva 4bitové hexadecimální znaky. Hlavní výhodou tohoto módu je větší hustota znaků, která umožňuje lepší propustnost než ASCII mód se stejnou přenosovou rychlostí. Každá zpráva přenášená v módu RTU musí být odeslána jako nepřetržitý proud bitů.

Formát (11 bitů) pro každý bajt v RTU módu je:

- Systém kódování.
  - 8 bitů binárně.
- Jednotlivé bity pro každý bajt.
  - 1 start bit,
  - 8 data bitů, nejméně významný bit se posílá jako první,
  - 1 bit pro kompletnost parity,
  - 1 stop bit.

Základní používaná parita je parita sudá; může být ale použita také lichá nebo žádná. Pokud se nepoužívá žádná parita; je bit pro kompletnost parity nahrazen druhým stop bitem.

Zařízení, které vysílá zprávu, ji umístí do rámce se známým začátkem a koncem. Díky tomu může zařízení, které přijímá novou zprávu rozlišit, kdy rámec zprávy začne a kdy je již zpráva kompletní. V RTU módu, jsou rámce zpráv oddělovány takzvaným tichým intervalem, který musí být dlouhý alespoň 3,5 znaku.

Adrase slave	Kód funkce	Data	CRC
1 bajt	1 bajt	0 až 252 bajtů	2 bajty CRC_{Low}, CRC_{High}

Obr. 3.7: Rámec při RTU módu [18].

Zpráva při RTU módu obsahuje pole pro kontrolu chyb, které je založeno na kontrole cyklické přebytečnosti (CRC –Cyclic Redudancy Checking). Pole CRC kontroluje obsah celé zprávy, je aplikováno bez ohledu na jakoukoliv metodu kontroly parity. Obsahuje 16bitovou hodnotu implementovanou jako dva bajty [18].

ASCII mód

Při nastavení zařízení pro vysílací mód ASCII (American Standard Code for Information Interchange – americký standartní kód pro výměnu informací) , každý bajt ve zprávě je posílán jako dva ASCII znaky. Tento mód se používá v případě, že fyzická komunikační linka, nebo samotné zařízení neumožňují komunikaci pomocí RTU módu. Tento mód je méně efektivní vzhledem k tomu, že každý bajt vyjadřujeme dvěma znaky. Například bajt 0x5B je kódován jako dva znaky: 0x35 a 0x42 (0x35=„5“, a 0x42=„B“ v ASCII)

Formát (10 bitů) pro každý bajt v ASCII módu je:

- Systém kódování
  - Hexadecimálné, znaky ASCII 0-9, A-F
  - Jeden hexadecimální znak obsahuje 4 bity dat obsažených v jednom znaku ASCII.
- Jednotlivé bity každého bajtu
  - 1 start bit
  - 7 data bitů, nejméně významný bit se posílá jako první
  - 1 bit pro kompletnost parity
  - 1 stop bit

Jako v případě RTU módu je zpráva protokolu Modbus umístěna zařízením které ji vysílá do rámce se známým začátkem a koncem. V ASCII módu je zpráva ohraničena specifickými znaky pro začátek a konec rámce. Zpráva musí začínat znakem *:*[2] a musí končit znaky *CR*, *LF*[3].

Začátek	Adresa	Funkce	Data	LRC	Konec
1 znak :	2 znaky	2 znaky	0 až 2x252 znaků	2 znaky	2 znaky CR, LF

Obr. 3.8: Rámec při ASCII módu [18].

Při módu ASCII potřebuje každý bajt dat dva znaky. Kvůli zajištění kompatibility mezi módy ASCII a RTU byla zvolena maximální velikost datové části zprávy pro mód ASCII 2x252, což odpovídá dvojnásobku velikosti datové části v módu RTU.

Při použití módu ASCII zpráva obsahuje pole pro kontrolu chyb, které je založeno na kontrole podélné přebytečností (LRC –Longitudinal Redundancy Checking), kalkulace je prováděna z obsahu celé zprávy kromě dvojtečky na jejím začátku a pole CRLF na jejím konci. LRC je aplikováno nehledě na jakoukoliv metodu kontroly parity a skládá se z jednoho bajtu, obsahujícího 8bitovou binární hodnotu. Výsledné LRC v ASCII je kódováno do dvou bajtů umístěných nakonec rámce, před CRLF [18].

3.3 Důvody využití protokolu Modbus

Protokol umožňuje komunikaci s mnoha zařízeními připojenými do jedné sítě. V našem případě případě to bude několik samostatných modulů pro testování připojených k jednomu řídícímu PC na kterém poběží testovací software.

Hlavní důvody použití protokolu Modbus:

- je to otevřený protokol, je zcela zdarma dostupný na internetu v několika různých implementacích. Díky své otevřenosti je neustále vyvíjen. Je implementován v jazyce C# a prostředí .NET, což je potřebné pro pozdější programování ovladačů,
- přesunuje jednoduché bity a slova bez mnoha požadavků, přímo do zařízení. I z tohoto důvodu je použitelný pro nespočetně zařízení od mnoha výrobců,
- jeho nasazení a údržba jsou vcelku jednoduch,
- pro pozdější potřeby komunikace přes Ethernet, má již přidělený port 502 [16].

[2]: – dvojtečka, v ASCII 3A hex.
[3]CR, LF – Carriage Return, Line Feed: znaky pro odřádkování, v ASCII 0D a 0A hex.

4 NTC SIMULATOR

Tato kapitola se věnuje popisu tetovacího modulu, takzvaného NTC Simulator a popisu vytvoření ovladače k tomuto modulu.

4.1 Popis modulu NTC Simulator

NTC Simulator je testovací modul využívaný pro testování v Honeywell ACS v Brno. Jedná se o desku plošných spojů s osmi ovladatelnými digitálními[1] potenciometry a čtyřmi plovoucími[2] potenciometry AD5235. Uspořádání jejich výstupů je zobrazeno na obrázku 4.1.

AD5235 je dvoukanálový, digitálně ovládaný potenciometr s rozlišením 1024 kroků a energeticky nezávislou pamětí. Na modulu jsou umístěny tři potenciometry s odporem 25 kΩ a jeden s odporem 250 kΩ. Pro potenciometr 25 kΩ je nejmenší krok přibližně 25 Ω.

Specifika modulu:

- Osm kanálů pro nastavování potenciometrů (0-7),
  - rozsah 25 kΩ: kanály 0,1,4,5,6,7,
  - rozsah 250 kΩ: kanály 2,3.
- Adresa modulu = 129 (pro všechny moduly).

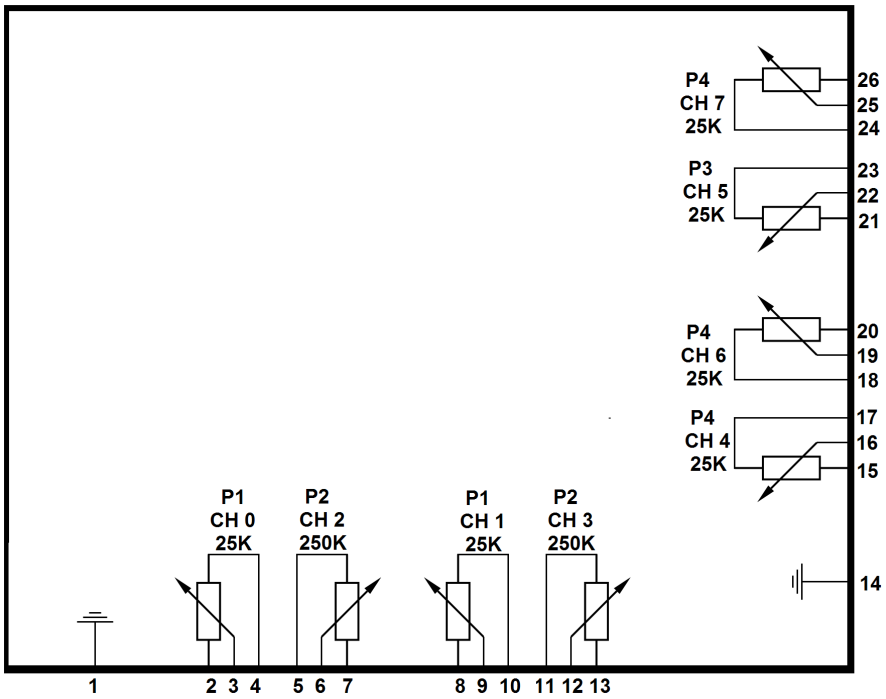
Tento modul může sloužit například k simulování odporového teplotního čidla. Při simulaci odporového čidla se výstupy modulu připojují na vstup (pro teplotní čidlo) námi měřeného zařízení namísto samotného čidla. Dle katalogového listu k danému čidlu potom víme, jaké odpory odpovídají určité teplotě. Hodnota nejmenšího kroku nastavení odporu na potenciometru 25 kΩ odpovídá 1 °F při 180 °F (83 °C) [20].

Propojení tohoto modulu s počítačem je realizováno pomocí USB kabelu. O převod signálu z USB na RS-232 a naopak se stará FTDI čip[3]. Mozkem celého modulu je potom mikrokontrolér ATMEL-ATMEGA8L-8AI. V paměti tohoto mikrokontroléru je prozatím software, který podporuje použití softwaru vycházejícího z protokolu Modbus, ale odlišného v detailech implementace. V budoucnu by měl být nahrazen implementací, která podporuje přímo protokol Modbus. Současně s výměnou tohoto softwaru bude nutné u jednotlivých modulů změnit adresy, aby bylo možné používat více těchto modulů zároveň. Použití více modulů se využívá například k simulaci hodnot na více vstupech, nebo pro jemnější nastavení odporu [20].

[1] Na obrázku 4.1 CH0-CH7 označuje piny kanálů jednotlivých digitálních potenciometrů.

[2] Na obrázku 4.1 P1-P4 označuje výstupy jednotlivých plovoucích potenciometrů.

[3] Čip firmy: Future Technology Devices International. Jde o čip pro převod signálu z USB na RS-232 a naopak.



Obr. 4.1: Schéma uspořádání potenciometrů na modulu NTC Simulator [20].

4.2 Ovladač pro NTC Simulator

Aby mohl být testovací modul NTC Simulator používaný v systému ATAB, bylo nutné pro něj naprogramovat ovladač. Tento ovladač byl naprogramován prostřednictvím vývojového prostředí Visual Studio v programovacím jazyce C#. Původně byl NTC Simulator ovládaný pomocí jednoduchého programu manuálně. Ovladač bude sloužit k tomu, aby mohl být NTC Simulator ovládaný systémem pro automatizované testování ATAB s využitím testovacího skriptu. Skript bude spouštěn na PC určeném pro běh a konfiguraci testů. Tento ovladač bude ovládat všechny moduly ze stejné rodiny. Pokud by tedy k jednomu PC bylo připojeno více modulů typu NTC Simulator, všechny je bude ovládat tento ovladač. V současné době to není možné, neboť moduly prozatím neumožňují nastavení různých adres, avšak tento ovladač je na to připraven[20].

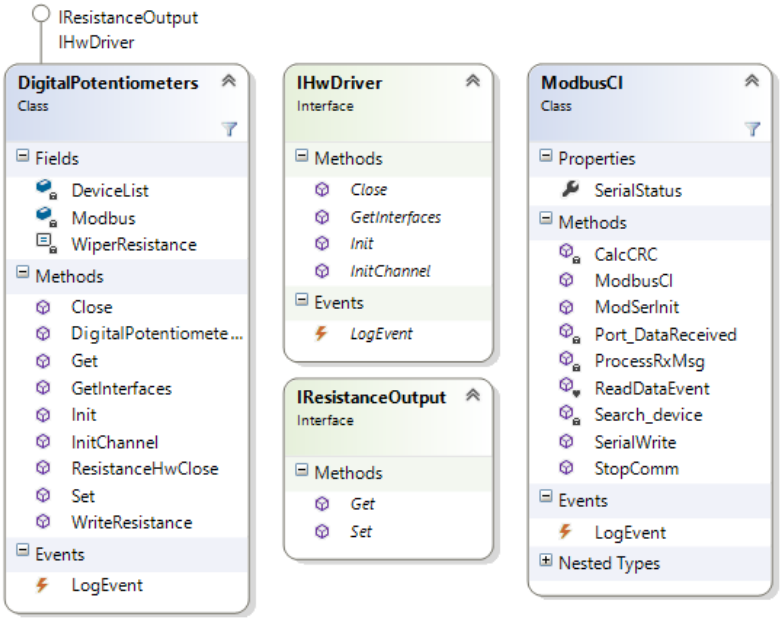
4.2.1 Popis kódu ovladače

Ovladač pro NTC Simulator se setává z jedné hlavní třídy, která volá metody pomocné třídy a implementuje dvě rozhraní. Všechny části nutné pro fungování ovladače v systému ATAB jsou zobrazeny v diagramu tříd na obrázku 4.2. Pro dodržení původního názvosloví a na požadavek zadavatele se hlavní třída nazývá `DigitalPotentiometer`. Třída `DigitalPotentiometer` implementuje rozhraní

`IHwDriver` a `IResistanceOutput`. Pro komunikaci s moduly i jejich inicializaci třída využívá metod knihovny `ModbusCl`. Tato knihovna má podobnou, ačkoliv ne tak rozsáhlou funkčnost jako implementace protokolu Modbus, která existuje v jazyce C# jako knihona třetí strany[16]. Nejedná se ale o přímou implementaci tohoto protokolu.

Ovladač obsahuje řadu metod, mezi nejdůležitější patří:

- Metody definované rozhraním `IHwDriver`,
- Metody definované rozhraním `IResistanceOutput`,
  - `Set()` – zajišťuje nastavení odporu na určitém kanálu modulu,
  - `Get()` – zajišťuje zjištění nastaveného odporu na určitém kanálu modulu.



Obr. 4.2: Diagram tříd pro ovladač modulu NTC Simulator.

4.2.2 Popis metod rozhraní

Prvním rozhraním, které ovladač implementuje, je rozhraní `IHwDriver`. Metody `Close()` a `Flush()` pracují přesně takovým způsobem, jak byly dříve popsány a jejich implementace je jednoduchá. Metoda `Init()` nejprve kontroluje, jestli modul již nebyl inicializován. Jestli nebyl, uloží hodnoty do potřebných proměnných a tuto událost zaloguje. Dále metoda zašle modulu na příslušném COM portu informaci na jakou rychlost se má nastavit. Nastavovaná přenosová rychlost je vždy 9600 baudů, což je rychlost podporovaná modulem NTC Simulator. V případě špatně zadaných údajů pro inicializaci modulu metody se díky try a catch bloku zachytí vyjímka a událost se zaloguje.

Dalším rozhraním je `IResistanceOutput`. Obsahuje pouze dvě metody. Zde jsou popsány jejich vstupní parametry:

- `Int32` module – pevná adresa modulu,
- `Int32` channel – jednotlivé potenciometry,
- `Double` value – hodnota odporu pro nastavení.

Metoda `Set()` volá metodu `WriteResistance()`, s údaji na kterém kanálu určeného modulu má být nastaven určitý odpor. Modul, na kterém se nastavuje odpor, musí být již inicializovaný. V případě špatně zadaných údajů pro nastavení odporu, se díky try a catch bloku zachytí vyjímka a událost se zaloguje [20].

Knihovna Modbus

Všechny výše uvedené metody volají metody z knihovny Modbus. Jedná se o knihovnu již používanou ve vývojovém středisku Honeywell ACS v Brně. Tato knihovna není nijak licencovaná a slouží pouze pro interní účely vývojového střediska.

Metody z knihovny Modbus které jsou v práci používány:

- `SerialInit()` – pro inicializaci modulů,
- `SerialWrite()` – pro zaslání dat do modulu,
- `calcCRC()` – pro výpočet *CRC*,
- `stopCom()` – pro ukončení komunikace na určitém COM portu,
- `ProcessRxMsg()` – pro zpracování příchozích dat z modulu.

4.2.3 Testování ovladače

Vzhledem k jednoduchosti ovladače stačilo otestovat inicializování modulu. Potom nastavit odpor na jeho potenciometrech pomocí metody `Set()` a hodnoty změřit měřícím přístrojem. Tímto byla funkčnost metody `Set()` otestována. Po nastavení hodnot metodou `Set()` stačilo zavolat na tyto kanály metodu `Get()`. Metoda `Get()` vrátila dle předpokladu hodnoty nastavené na určitých potenciometrech. Jako poslední bylo otestováno takzvané vyhození vyjímek při špatně zadaných údajích, nefunkčním spojení a pokusu o inicializaci nepřipojeného modulu.

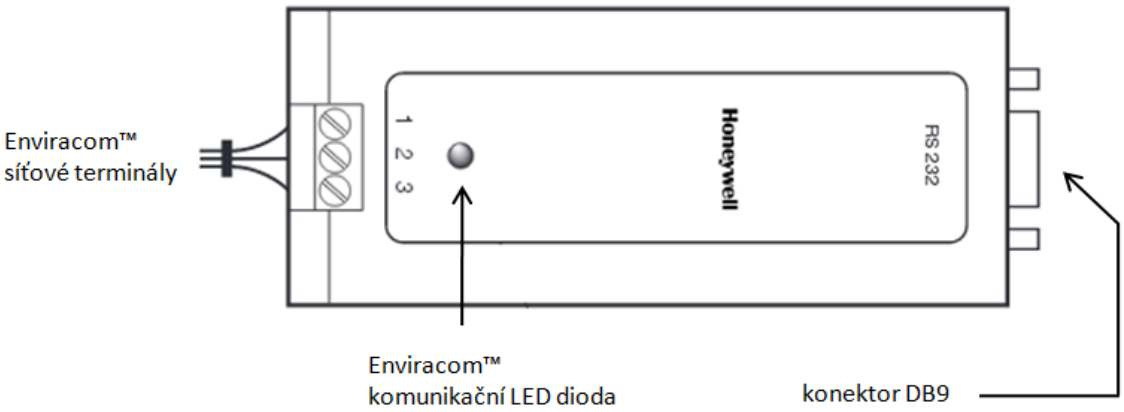
5 ENVIRACOM SERIAL ADAPTER

Tato kapitola se věnuje popisu testovacího modulu W8735A EnviraCOM Serial Adapter, dále už jen modul EnviraCOM Adapter. A popisu vytvoření ovladače k tomuto modulu, pro jeho začlenění do systému ATAB, včetně jeho testování.

5.1 Popis modulu EnviraCOM Adapter

Testovací modul EnviraCOM Adapter je produkt společnosti Honeywell, který lze (alespoň v USA) koupit a použít pro monitorování sběrnice EnviraCom. Připojoval se k hostitelskému počítači či jinému zařízení pomocí konektoru DB9[1] a sloužil jako brána či kontrolér pro přístup do Enviracom HVAC[2] systému. Modul EnviraCOM Adapter je napájen pomocí Enviracom připojení a je opticky izolovaný pro univerzální kompatibilitu. Bežně používal nízkonákladový komunikační protokol na aplikační vrstvě Enviracom. Pro potřeby testování se však užívá komunikační protokol Enviracom na datové vrstvě.

Enviracom™ je proprietální protokol firmy Honeywell používající tři vodiče. První vodič slouží jako datový, ostatní dva jsou pro napájení dvaceti čtyřmi volty (střídavými). Tento protokol umožňuje ovládacím a monitorovacím prvkům šířit zprávy týkající se statusu, alarmu a dalších informacích v síti. Informace obsahuje signalizaci požáru, sílu signálu, velikost odporu, počet cyklů, čas, poplachy a další. V našem případě přijímání a zásílání zpráv pro testované zařízení.



Obr. 5.1: Modul EnviraCOM Adapter [21].

[1]DB9 je standartní devítipinový konektor pro připojení k portu RS-232 na hostitelském zařízení pomocí sériového kabelu.

[2]HVAC-heating, ventilation, and air conditioning. V překladu topení, ventilace a klimatizace.

Pro připojeí modulu EnviraCOM Adapter k hostitelskému zařízení je třeba dodržet tato nastavení:

- Přenosová rychlost: 19200 baudů za sekundu,
- Parita: žádná,
- Datové bity: 8,
- Stop bity: 1 [22].

5.2 Ovladač pro modul EnviraCOM Adapter

Pro vytvoření ovladače k modulu EnviraCOM Adapter, bylo nejprve nutné seznámit se s datovou a aplikační vrstvou tohoto protokolu z těchto materiálů:

- Honeywell Enviracom Protocol, Data Link Layer Specification [23].
- Honeywell Enviracom Protocol, Aplication Layer Specification [24].

Přesná specifikace protokolu Enviracom a poznatky z těchto materiálů bohužel nemohou být v této práci zveřejněny kvůli tomu, že se na ně vztahuje obchodního tajemství firmy Honeywell. Tento ovladač bude sloužit ke stálému přijímání zpráv protokolu Enviracom na pozadí, k jejich případnému odesílání a dalším funkcím definovaným v rozhraní `IEnviracom`. Stejně jako v případě ovladačů pro ostatní testovací moduly, tento ovladač implementuje rozhraní `IHwDriver` pro inicializaci jednotlivých modulů a ostatní funkce. Dále ovladač implementuje výše zmíněné rozhraní `IEnviracom`. Tento ovladač je schopen obstarávat kontinuální přijímání zpráv a další funkce pro dostatečné množství modulů EnviraCOM Adapter (pro testování v systému ATAB) za předpokladu, že každý modul bude připojen do samostatného komunikačního portu.

5.2.1 Popis kódu ovladače

Samotný kód ovladače se skládá z několika tříd, rozhraní a výčtových typů, které budou popsány níže. Všechny tyto části nutné pro fungování ovladače v sytému ATAB jsou zobrazeny v diagramu tříd na obrázku 5.2. Hlavní třídou ovladače je `EnviraComSerialAdapter`. Tato třída implementuje rozhraní `IHwDriver` a `IEnviracom`. Třída `EnviraComSerialAdapter` využívá pro svůj běh metod a vlastností tříd `EnviraComVariable` `EnviraMessage`. Deklarace výčtových typů `enum`[3]: `Error`, `Priority` a `Service` je součástí souboru třídy `EnviraMessage`.

Hlavní třída `EnviraComSerialAdapter` obsahuje řadu metod a vlastností:

- Metody rozhraní `IHwDriver`.
- Metody rozhraní `IEnviracom`:

[3]enum je zkratka pro deklaraci výčtového typy v .NET frameworku.

`Send()` – odesílání zpráv na jednotlivé moduly (mohou být metodě předány jako `EnviraMessage` nebo `string`),

`Receive()` – vyčtení hodnot zpráv a jejich časových razítek z bufferu pro přijaté zprávy jako `string`,

`ReceiveEnviraMessage()` – vyčtení hodnot zpráv a jejich časových razítek z bufferu pro přijaté zprávy jako `EnviraMessage`,

`Flush()` – vymazání obsahu bufferů určitého modulu,

`WaitForMessage()` – přijetí jedné zprávy, která se bude shodovat s zadaným vzorem, navrací danou zprávu jako `string`,

`WaitForEnviraMessage()` – přijetí jedné zprávy, která se bude shodovat s zadaným vzorem, navrací danou zprávu jako `EnviraMessage`.

- Další metody:

  `ReceiveData()` – přijetí jednoho bitu z příslušného modulu a jeho případné uložení do příslušného bufferu,

  `Receiving()` – volání metody, `ReceiveData()` a v případě přijetí celé zprávy její uložení do příslušného bufferu,

  `ReceivingTask()` – zajištění kontinuálního přijímání zpráv, pro každý inicializovaný modul na pozadí,

  `SendBitArray()` – zaslání zprávy na příslušný modul bit po bitu,

  `SendingTask()` – zajištění odeslání kompletní zprávy na příslušný modul.

Většina metod implementuje takzvané vyhazování a zachytávání výjimek. Výjimky mouhou nastat při: selhání spojení, neexistující položce ve slovníku, neexistujícímu COM portu, nesouhlasnému CRC přijaté zprávy s vypočteným, špatně zadanými parametry, či vyhozením všeobecné výjimky.

Třída `EnviraComVariables` se sestává pouze z vlastností, které jsou potřebné pro správné přijímání a odesílání zpráv. Při inicializaci jednotlivých modulů se vytvoří vždy nová instance této třídy. Instance této třídy jsou ukládány do slovníku:

```
PortVars = new ConcurrentDictionary<int, EnviraComVariables>();
```

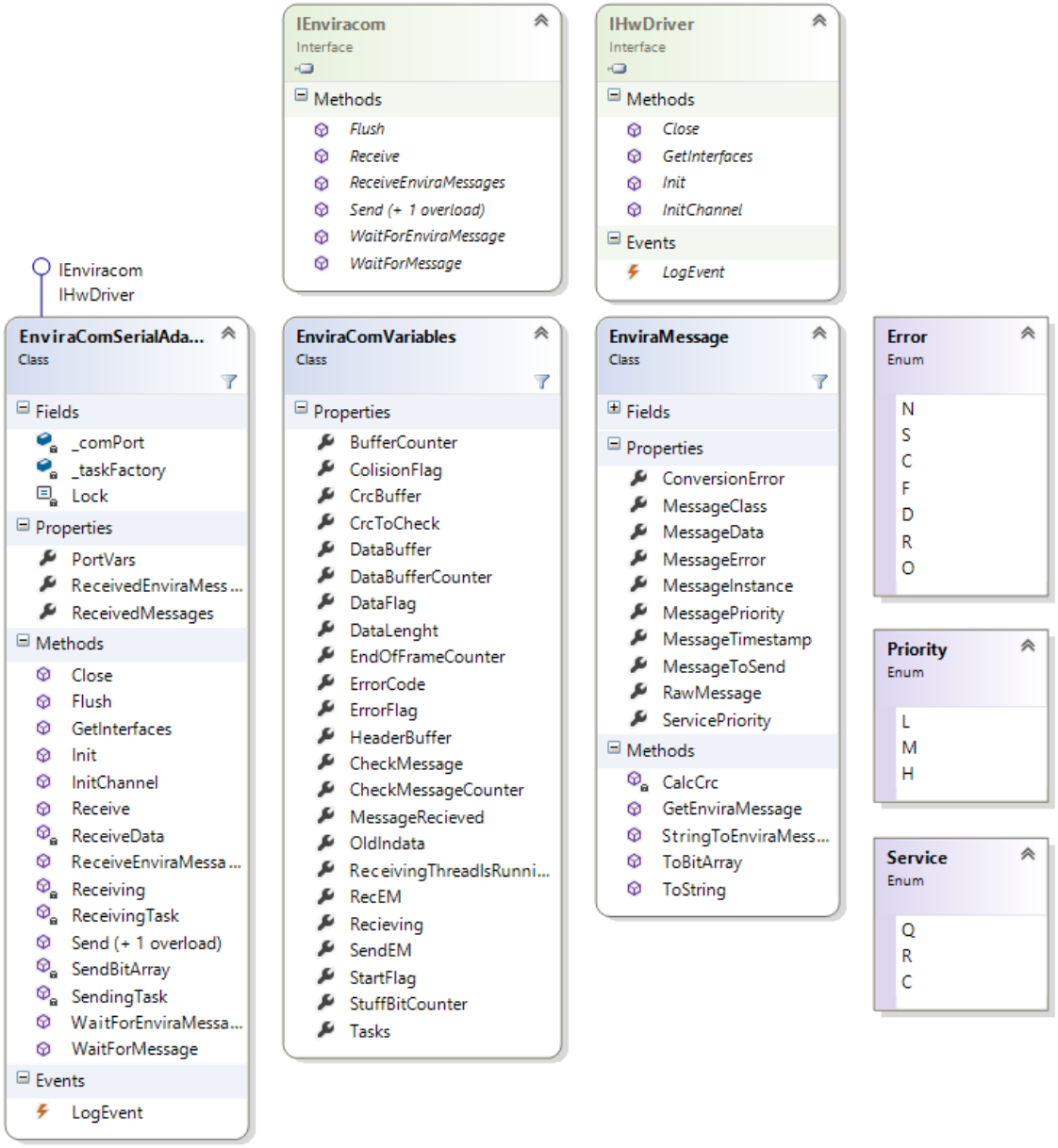
Ve slovníku jsou ukládány a v případě potřeby volány pod určitým klíčem. Klíč pro jednotlivé instance se shoduje s číslem modulu. Číslo modulu se přiřazuje libovolně při konfiguraci testovacího skriptu. Typ slovníku je `ConcurrentDictionary`, což je souběžný slovník. Tento slovník se od běžného liší tím, že do něj mohou soubežně přistupovat úlohy jednotlivých modulů bez rizika kolize.

Poslední třídou, kterou je třeba popsat, je `EnviraMessage`. Jedná se o datový typ pro ukládání a práci se zprávami, které posílá a přijímá modul EnviraCOM Adapter. Obsahuje vlastnosti jak pro jednotlivé části zprávy, tak pro zprávu jako `string`, nebo `BitArray` a časovou známku. Tři z vlastností jsou výše zmíněné výčt;, jde o výčet priorit zprávy, servisního pole zprávy a chyb. Použitím výčtových typů získáme

pevně daný výčet námi deklarovaných hodnot. Kdybychom namísto výčtového typu `enum` použili typ `string`, nebyl by jeho obsah pevně dán a mohl by obsahovat jakkoliv dlouhý text.

Hlavními metodami této třídy jsou:

- `GetEnviraMessage()` – převod přijatých bitů do `EnviraMessage`,
- `StringToEnviraMessage()` – převod z typu `string` do `EnviraMessage`,
- `ToBitArray()` – převedení `EnviraMessage` do pole bitů a to kvůli odesílání,
- `ToString()` – přetížení metody ToString pro potřeby této třídy.



Obr. 5.2: Diagram tříd pro ovladač modulu EnviraCOM Adapter.

5.2.2 Popis metod rozhraní

Prvním rozhraním, které ovladač implementuje, je rozhraní `IHwDriver`. Za zmínku zde stojí specifická implementace metody `InitChannel()`. Po inicializaci spojení metodou `Init()` nastane instanciování potřebných bufferů třídy `EnviraComVariables` a hlavně přiřazení jedinečné úlohy metodě ReceivingTask, pomocí metod třídy `TaskFactory`[4]. Díky těmto úlohám, může ovladač ihned po inicializaci přijímat v reálném čase zprávy daného modulu. Dalším rozhraním je rozhraní `IEnviracom`; toto rozhraní definuje metody které budou volány testovacím skriptem. Tato skutečnost z nich dělá sťežejní metody celého ovladače a proto zde budou podrobněji popsány. Přesná podoba rozhraní `IEnviracom` je uvedena v příloze A.2 Rozhraní `IEnviracom`.

Metody rozhraní `IEnviracom` vyžadují řadu vstupních parametrů. Vstupní parametry vyžadované pro volání jednotlivých metod můžeme vidět v definici rozhraní v příloze A.2.

Vstupní parametry metod rozhraní `IEnviracom` a jejich typy:

- `Int32` module – námi přidělená adresa modulu,
- `Int32` channel – jednotlivé komunikační kanály modulů, moduly EnviraCOM Adapter mají pouze jeden,
- `String[]` messages – pole zpráv ve formátu `String`,
- `Int32` timeoutMs – časový limit ať už pro odeslání nebo čekání na zprávu,
- `EnviraMessage[]` messages – pole zpráv ve formátu `EnviraMessage`,
- `String` messagePattern – vzor hledané zprávy používaný metodou `WaitForMessage()`.

Metoda `Send()`: Základní funkce této metody je odesílat zprávy jak již vyplývá z jejího názvu. Nejprve metoda pozastaví úlohu pro kontinuální přijímání zpráv, která běží na pozadí a až potom vždy vytvoří novou úlohu pro každou odesílanou zprávu. Před skončením metody se ještě znovu vytvoří úloha na pozadí pro kontinuální přijimání zpráv. Při vypršení času, během kterého má metoda stihnout odeslat všechny zprávy, se doposílá poslední rozpracovaná zpráva a metoda vrátí hodnotu `timeoutMs`. V případě že metoda stihne odeslat všechny zprávy včas, vrátí čas v milisekundách, ve kterém byly zprávy odeslány. V metodě je ošetřeno zadání špatného formátu zprávy, proměnné `timeoutMs`, neexistující/neicnicializovaný modul, či jiná chyba spojení. Při selhání odesílání jakékoliv zprávy (ne z důvodu vypršení timeoutu), metoda zaloguje jeho důvod zprávu a modul, kde došlo k chybě. Tato metoda má jedno přetížení, u kterého požaduje jako parametr `messages` typ `EnviraMessage[]`, namísto `String[]`.

Metoda `Receive()`: Metoda pouze vrací obsah bufferu na přijaté zprávy, který se plní pomocí úlohy na pozadí; po vrácení zpráv je obsah bufferu smazán. V metodě

[4]TaskFactory je třída zajišťující vytváření a správu úloh, které lze spustit na pozadí programu.

je ošetřeno volání metody na neexistující buffer.

Metoda `ReceiveEnviraMessage()`: Tato metoda je implementována shodně s metodou `Receive()`, líší se pouze v návatovém typu. Vrací `EnviraMessage[]`, namísto `KeyValuePair<long, String>[]`.

Metoda `Flush()`: Metoda pro vyčištění všech bufferů námi zvoleného modulu. V metodě je ošetřeno volání metody na neexistující buffer.

Metoda `WaitForMessage()`: Metoda pro čekání na zprávu určenou dle zadaného vzoru, vzor může být buďto přesný nebo může obsahovat patametry x/X, které zastávají jakýkoliv znak. Vzor se porovnává se zprávou od začátku; cokoliv je ve zprávě delší než vzor je při kontrole shody ignorováno. V metodě je ošetřeno volání metody na neinicializovaný modul, chybu spojení a špatný formát parametru `timeoutMs`. Parametr `timeoutMs` může nabývat hodnoty -1, a všech kladných hodnot z rozsahu proměnné typu `Int32`, kde -1 zastává nekonečno. Při shodě vzoru se zprávou je navrácen celý obsah bufferu přijatých zpráv, kde je zpráva, která se shodovala se vzorem poslední. Po návratu těchto zpráv je obsah bufferu smazán. Pokud vyprší `timeoutMs` a shoda nenastane, metoda vrátí `null`[5]. Tato metoda pracuje se zprávou ve formátu `String`.

Metoda `WaitForEnviraMessage()`: Tato metoda je implementována shodně s metodou `WaitForMessage()`, líší se pouze v návatovém typu. Vrací `EnviraMessage[]`, namísto `KeyValuePair<long, String>[]`.

5.2.3 Testování ovladače

Jednou z důležitých částí tvorby ovladače bylo jeho testování. Testování bylo prováděno jednoduchými testy pomocí softwaru testovacího systému ATAB. Jeho účelem bylo odladit veškeré nedostatky ovladače a ověřit funkčnost v určitých situacích. Testy byly prováděny pomocí testovacího skriptu. Vzhled kódu testovacího skriptu najdeme v příloze A.3 Kód pro testování ovladače. Skript pro testování se skládá se z několika takzvaných `TestCase`, což jsou jednotlivé modelové případy pro testování.

V průběhu vývoje ovladače bylo zapotřebí tyto a další testy provádět opakovaně. Jako u kteréhokoliv jiného testování byly počáteční výsledky testování nevyhovující a po těchto testech následovala vždy úprava kódu. Tento proces se opakoval tak dlouho, dokud výsledky testování nedopadly dle předpokladů. Testy uvedené v této kapitole již prokázaly takové výsledky jaké po nich byli požadovány od začátku testování.

Pro testování byly použity dva moduly EnviraCOM Adapter, propojené mezi

[5]null je speciální klíčové slovo v C# reprezentující referenci, která neukazuje na žádný objekt.

sebou a jeden modul DHW[6]. Při zapnutí tohoto modulu je generováno několik Envira zpráv, což bylo ideální pro testování našeho ovladače.

Jak lze vidět z kódu v příloze A.3, v metodě `Main()` se nejprve vytvoří instance bufferu na zprávy `messages`, poté se inicializuje modul vytvořením instance třídy `TestConfigurator` s vstupní proměnnou `"testConfig.xml"`. Obsah této vstupní proměnné je ukázán v příloze A.5. Z tohoto souboru lze vidět, jaké informace o modulu je potřeba skriptu dodat pro použití ovladače modulu EnviraCOM Adapter.

Prvními testy byly: základní test metod rozhraní IEnviracom a základní test metod rozhraní IEnviracom, spolu s příjmem zpráv z DHW. V testu šlo v první řadě o odeslání zpráv bez vyhození vyjímky.

V prvním testu, metoda `WaitForMessage()` vrátila pouze `null`, protože neprobíhal žádný příjem zpráv, a tak nemohla žádná zpráva odpovídat vzoru. Když byl test opakován spolu se zapnutím DHW a vzor metody se shodoval s některou z vysílaných zpráv. Metoda vrátila všechny přijaté zprávy a jako poslední byla zpráva, která odpovídala danému vzoru. Tímto testem bylo ověřeno i to, že jsou zprávy kontinuálně přijímány na pozadí. V konečné fázi testu bylo posláno ještě několik zpráv, otestováno jestli funguje metoda `flush()`. Výstup testu je zobrazen v příloze A.1 a A.2. V kódu jde potom o metodu `TestCase1()`.

Druhým testem byl test metody Send(), spolu s příjmem zpráv z DHW a test metody Send() po příjmu zpráv z DHW. V tomto testu nás zajímalo hlavně, jestli nedojde při souvislém přijímání a pokusu k odesílání ke kolizi. V dalším testu, jestli po přijímutí zpráv zvládne ovladač navázat bezchybným vysíláním. Výstup testu je zobrazen v příloze A.3 a A.4.V kódu jde potom o metody `TestCase2()` a `TestCase3()`.

V poslední řadě bylo otestovní zalogování výjimek, testovalo se zadání špatného formátu obecných proměnných pro metody, špatného formátu proměnné `timeoutMs`, inicializace neexistujícího modulu, odpojení modulu. Výstup testu je zobrazen v příloze A.5. Testování těchto výjimek bylo prováděno pomocí `TestCase1()`, s úpravou některých proměnných, a pomocí úpravy souboru `testConfig.xml`.

[6]DHW je modul používaný v HVAC, jedná se o domestic hot water module, v překladu domácí modul teplé vody.

ZÁVĚR

Úkolem této práce bylo získat patřičné teoretické základy v oblasti automatizovaného testování při vývoji embedded zařízení a tyto teoretické základy využít pro vytvoření ovladače pro vybraný testovací modul.

První část práce je zamřena obecně na automatizaci testování embedded zařízení, a to zejména při použití hardwarových testovacích modulů. To bylo popsáno v kapitolách 1 Embedded zařízení a jejich testování a 2 Systémy pro automatizované testování.

Druhá část práce se zabývá architekturou systému pro podporu automatizovaného testování ATAB; jejímu popisu se práce věnuje v podkapitole 2.1 Popis systému pro automatizované testování ATAB. V této a předešlé kapitole, se práce věnuje i přehledu a srovnání ostatních systémů testování.

V poslední řadě před samotnou tvorbou ovladače, bylo nutno nastudovat strukturu protokolu Modbus a jeho použití na komunikační sběrnici RS-232, případně USB: tato struktura a její použití je popsána v kapitole 3 Protokol Modbus. Dále bylo nutné seznámit se s vývojovým prostředím Visual Studio, platformou .NET a programovacím jazykem C#. Nutnost studia struktury protokolu Enviracom vyvstala až později a detaily jeho struktury nemohou být v této práci zveřejněny, z důvodu že jde o proprietální protokol společnosti Honeywell. Pouze v kapitole 5 EnviraCOM Serial Adapter, je stručný popis k čemu tento protokol slouží.

Modulem, pro který měl být vytvořen ovladač, byl NTC Simulator. Popis zmíněného modulu a ovladačů k němu je popsán v kapitole 4 NTC Simulator. Zmíněný modul nepodporoval přímo protokol Modbus, nýbrž jen jemu podobný komunikační protokol. To se mělo změnit a daný ovladač měl být upraven pracovníkem firmy Honywell pro použití s protokolem Modbus a rozšířen o implementaci rozhraní `IResistanceOutputExtend` což se však dosud nestalo.

Dalším vybraným modulem byl EnviraCOM Serial Adapter a úkolem bylo vytvořit ovladač pro tento modul. Popis dalšího modulu, ovladačě k němu a jeho testování je popsán v kapitole 5 EnviraCOM Serial Adapter.

Hlavním cílem této práce bylo vytvořit ovladač pro vybraný testovací modul, tento cíl byl splněn. Navíc byl vytvořen ovladač pro druhý modul EnviraCOM Serial Adapter.

LITERATURA

[1] BARR, Michael a Anthony J MASSA. *Programming embedded systems: with C and GNU development tools.* 2 ed. Sebastopol: O'Reilly, 2006, xxi, 301 s. ISBN 978-0-596-00983-0.

[2] HEATH, S. *Embedded systems design.* Vyd. 1. New York: Prentice-Hall, 2003, 430 s. ISBN 07-506-5546-1.

[3] PALM, David. Overcoming the Unique Challenges of Test Automation on Embedded Systems. *Automated Software Testing Magazine* [online]. roč. 2013, Duben [cit. 2013-11-12]. Dostupné z URL: <http://www.automatedtestinginstitute.com/home/ASTMagazine/2013/AutomatedSoftwareTestingMagazine_April2013.pdf>.

[4] HLAVA, Tomáš. Smoke testy. *Testování softwaru* [online]. 20.8.2011 [cit. 2013-11-10]. Dostupné z URL: <http://testovanisoftwaru.cz/tag/smoke-testy/>.

[5] KANER, C. – BACH, J. – PETTICHORD, B. *Lessons Learned in Software Testing.* John Wiley & Sons, Inc., New York, 2011, ISBN 9780471081128

[6] RAMAN, Ganesh. Beginner Guide For Software Testing. *Software testing interview faqs* [online]. WordPress.com, 30.05.2009 [cit. 2013-11-21]. Dostupné z URL: <http://softwaretestinginterviewfaqs.wordpress.com/2009/05/30/beginner-guide-for-software-testing/>.

[7] ZALLAR, Kerry. Practical Experience in Automated Testing. *Methods & tools: Software Development Magazine.* [online]. [cit. 2013-11-12]. Dostupné z URL: <http://www.methodsandtools.com/archive/archive.php?id=33>.

[8] CHANGEDE, Anuja. What are white-box, black-box and gray-box testing? *Career Ride* [online]. [cit. 2013-11-10]. Dostupné z URL: <http://www.careerride.com/Testing-white-box-black-box-gray-box.aspx>.

[9] CHROMA SYSTEMS SOLUTIONS, Inc. *Automated Test Systems.* [online]. Chroma Systems Solutions [cit. 2013-11-11]. Dostupné z URL: <http://www.chromausa.com/automatedtestsystems.php>.

[10] NATIONAL INSTRUMENTS CORPORATION. *Web National Instruments.* [online]. [cit. 2013-11-11]. Dostupné z URL: <http://czech.ni.com/>.

[11] AGILENT. *Web Agilent.* [online]. [cit. 2013-11-11]. Dostupné z URL: <http://www.home.agilent.com/agilent/home.jspx?lc=eng&cc=CZ>.

[12] SOMA GMBH. *Web SOMA GmbH.* [online]. [cit. 2013-11-11]. Dostupné z URL: <http://www.soma.de/en/>.

[13] 7 Deadly Sins of Automated Software Testing. SMITH, Adrian. *Engineering design and Agile software development* [online]. 03.01.2012 [cit. 2013-11-20]. Dostupné z URL: <http://www.agileengineeringdesign.com/2012/01/7-deadly-sins-of-automated-software-testing/>.

[14] SVOBODA, Radomír. HONEYWELL. *Software Test Improvement.* Brno, 2013

[15] Interfaces *C# Programming Guide.* [online]. Microsoft [cit. 2013-12-06]. Dostupné z URL: <http://msdn.microsoft.com/en-US/Library/ms173156%28v=vs.110%29.aspx>.

[16] Modbus FAQ: About the Protocol. [online]. http://www.modbus.org, Modbus Organization, Inc. [cit. 2013-11-18]. Dostupné z URL: <http://www.modbus.org/faq.php>.

[17] MODBUS APPLICATION PROTOCOL SPECIFICATION [online]. V1.1b3: http://www.modbus.org, Modbus Organization, Inc., 2012. [cit. 2013-11-18]. Dostupné z URL: <http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf>.

[18] MODBUS over Serial Line: Specification and Implementation Guide [online]. V1.02: http://www.modbus.org, Modbus Organization, Inc., 2006. [cit. 2013-11-18]. Dostupné z URL: <http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf>.

[19] RONEŠOVÁ, Andrea. Přehled protokolu MODBUS. [online]. 2005. [cit. 2013-11-18]. Dostupné z URL: <http://home.zcu.cz/ronesova-/bastl/files/modbus.pdf>.

[20] SVOBODA, Radomír. HONEYWELL. *NTC Simulator.* Brno, 2013

[21] Honeywell *W8735A1005 EnviraCOM™ Serial Adapter NSTALLATION INSTRUCTIONS* [online]. 2002. [cit. 2014-04-25]. Dostupné z URL: <https://customer.honeywell.com/resources/Techlit/TechLitDocuments/69-0000s/69-1351.pdf>.

[22] Honeywell *W8735A1005 EnviraCOM™ Serial Adapter SPECIFICATION DATA* [online]. 2004. [cit. 2014-04-25]. Dostupné z URL: <https://customer.honeywell.com/resources/Techlit/TechLitDocuments/68-0000s/68-3063.pdf>.

[23] *Honeywell Enviracom Protocol, Data Link Layer Specification.* Version 1.10, 2012. Honeywell Confidential.

[24] *Honeywell Enviracom Protocol, Application Layer Specification.* Version 1.10, 2012. Honeywell Confidential.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

ADU datová jednotka aplikační vrstvy – Aplication Data Unit

ASCII americký standartní kód pro výměnu informací – American Standard Code for Information Interchange

ATAB automatizované testování v ACS Brno – Automated Testing in ACS Brno

CRC kontrola cyklické přebytečnosti – Cyclic Redudancy Checking

DHW domáci modul teplé vody – Domestic How Watter module

HVAC topení, ventilace a klimatizace – Heating, Ventilation, and Air Conditioning

LRC kontrola podélné přebytečností – Longitudinal Redundancy Checking

PDU jednoduchá datová jednotka – Protocol Data Unit

RFC žádost o komentáře – Request for Comments

RTU vzdálená ovládací jednotka – Remote Terminal Unit

SEZNAM PŘÍLOH

A PŘÍLOHY K TEXTU PRÁCE

A.1 Rozhraní IHwDriver

```
using System;

namespace AutomatedTesting
{
    public delegate void HwMessageDelegate(MessageContainer Message);

    public interface IHwDriver
    {

        Boolean Init(Int32 module, String config);

        Boolean InitChannel(Int32 module, Int32 channel, String config);

        Type[] GetInterfaces();

        void Close();

        event HwMessageDelegate LogEvent;
    }
}
```

A.2 Rozhraní IEnviracom

```
using System;
using System.Collections.Generic;

namespace AutomatedTesting
{

    public interface IEnviracom
    {

        Int32 Send(Int32 module, Int32 channel, String[] messages, Int32
            timeoutMs);

        Int32 Send(Int32 module, Int32 channel, EnviraMessage[] messages, Int32
            timeoutMs);

        KeyValuePair<long, String>[] Receive(Int32 module, Int32 channel);

        EnviraMessage[] ReceiveEnviraMessages(int module, int channel);

        void Flush(Int32 module, Int32 channel);

        KeyValuePair<long, String>[] WaitForMessage(Int32 module, Int32 channel,
            String messagePattern, Int32 timeoutMs);

        EnviraMessage[] WaitForEnviraMessage(Int32 module, Int32 channel, String
            messagePattern, Int32 timeoutMs);
    }
}
```

A.3 Kód pro testování ovladače

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;

namespace AutomatedTesting
{
    class Program
    {

        private static void Main(string[] args)
        {
            KeyValuePair<long, string>[] messages  = new KeyValuePair<long,
                string>[0];
            var tc = new TestConfigurator("testConfig.xml");
            var testProgram = new Program();
            Console.WriteLine("=====================================");
            Console.WriteLine("[{0}] Inicializováno",
                DateTime.Now.ToString("hh:m:ss:ffff"));
            Console.WriteLine("=====================================\r");

            testProgram.TestCase1(tc, messages);

            //čekání na print screen
            Thread.Sleep(180000);
            tc.Finish();
        }

        private  void OutputWriteLine(KeyValuePair<long, string>[] messages)
        {
            Console.WriteLine("[{0}] Výpis návratových hodnot přechozí metody:",
                DateTime.Now.ToString("hh:mm:ss:ffff"));
            if (messages != null && messages.Count() != 0)
            {
                foreach (var msg in messages)
                {
                    Console.WriteLine(msg);
                }
            }
            else
            {
                Console.WriteLine("[{0}]Metoda nevrátila žádné zprávy",
                    DateTime.Now.ToString("hh:mm:ss:ffff"));
            }
        }

        private  void TestCase1(TestConfigurator tc, KeyValuePair<long, string>[]
            messages )
        {
            Console.WriteLine("=====================================");
            Console.WriteLine("[{0}] Testuji metodu WaitForMessage()",
                DateTime.Now.ToString("hh:m:ss:ffff"));
            Console.WriteLine("=====================================\r");
            Console.WriteLine("[{0}] Volání metody WaitForMessage() na ECOM_1:",
                DateTime.Now.ToString("hh:mm:ss:ffff"));
            messages = tc.WaitForMessage("ECOM_1", "ECOM", "M 12X0 XX X 03",
```

```
            10000);
        OutputWriteLine(messages);
        Console.WriteLine("=====================================");
        Console.WriteLine("[{0}]Testuji metody Send(), Receive() a přijímání
            zpráv na pozadí", DateTime.Now.ToString("hh:mm:ss:ffff"));
        Console.WriteLine("=====================================\r");
        Console.WriteLine("[{0}]Volání metody Send() pro odeslání tří zpráv
            ECOM_2", DateTime.Now.ToString("hh:mm:ss:ffff"));
        tc.Send("ECOM_2", "ECOM", new string[] { "L 1111 FF R 00", "M 2222 00
            Q 00", "H 3333 00 C 00" }, -1);
        Console.WriteLine("[{0}]Uspání hlavního vlákna na půl sekundy",
            DateTime.Now.ToString("hh:mm:ss:ffff"));
        Thread.Sleep(500);
        Console.WriteLine("[{0}]Volání metody Receive() na ECOM_1",
            DateTime.Now.ToString("hh:mm:ss:ffff"));
        messages = tc.Receive("ECOM_1", "ECOM");
        OutputWriteLine(messages);
        Console.WriteLine("=====================================");
        Console.WriteLine("[{0}]Testuji metodu Flush()",
            DateTime.Now.ToString("hh:mm:ss:ffff"));
        Console.WriteLine("=====================================\r");
        Console.WriteLine("[{0}]Volání metody Send() pro odeslání tří zpráv
            ECOM_2", DateTime.Now.ToString("hh:mm:ss:ffff"));
        tc.Send("ECOM_2", "ECOM", new string[]
            {"L 3FFF FF R 3F 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16
                01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16" +
                " 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 01 02 03 04
                    05 06 07 08 09 10 11 12 13 14 15"
                , "M 3E70 00 Q 00", "H 0000 00 C 00"}, -1);
        Console.WriteLine("[{0}]Uspání hlavního vlákna na půl sekundy",
            DateTime.Now.ToString("hh:mm:ss:ffff"));
        Thread.Sleep(500);
        Console.WriteLine("[{0}]Volání metody Flush() na ECOM_1",
            DateTime.Now.ToString("hh:mm:ss:ffff"));
        tc.Flush("ECOM_1", "ECOM");
        Console.WriteLine("[{0}]Volání metody Receive() na ECOM_1",
            DateTime.Now.ToString("hh:mm:ss:ffff"));
        messages = tc.Receive("ECOM_1", "ECOM");
        OutputWriteLine(messages);
    }

    private void TestCase2(TestConfigurator tc, KeyValuePair<long, string>[]
        messages )
    {
        Console.WriteLine("=====================================");
        Console.WriteLine("[{0}]Testuji metody Send(), Receive() a přijímání
            zpráv na pozadí", DateTime.Now.ToString("hh:mm:ss:ffff"));
        Console.WriteLine("=====================================\r");
        Console.WriteLine("[{0}]Volání metody Send() pro odeslání tří zpráv
            ECOM_2", DateTime.Now.ToString("hh:mm:ss:ffff"));
        tc.Send("ECOM_2", "ECOM", new string[] { "L 1111 FF R 00", "M 2222 00
            Q 00", "H 3333 00 C 00" }, -1);
        Console.WriteLine("[{0}]Uspání hlavního vlákna na půl sekundy",
            DateTime.Now.ToString("hh:mm:ss:ffff"));
        Thread.Sleep(500);
        Console.WriteLine("[{0}]Volání metody Receive() na ECOM_1",
            DateTime.Now.ToString("hh:mm:ss:ffff"));
        messages = tc.Receive("ECOM_1", "ECOM");
        OutputWriteLine(messages);
```

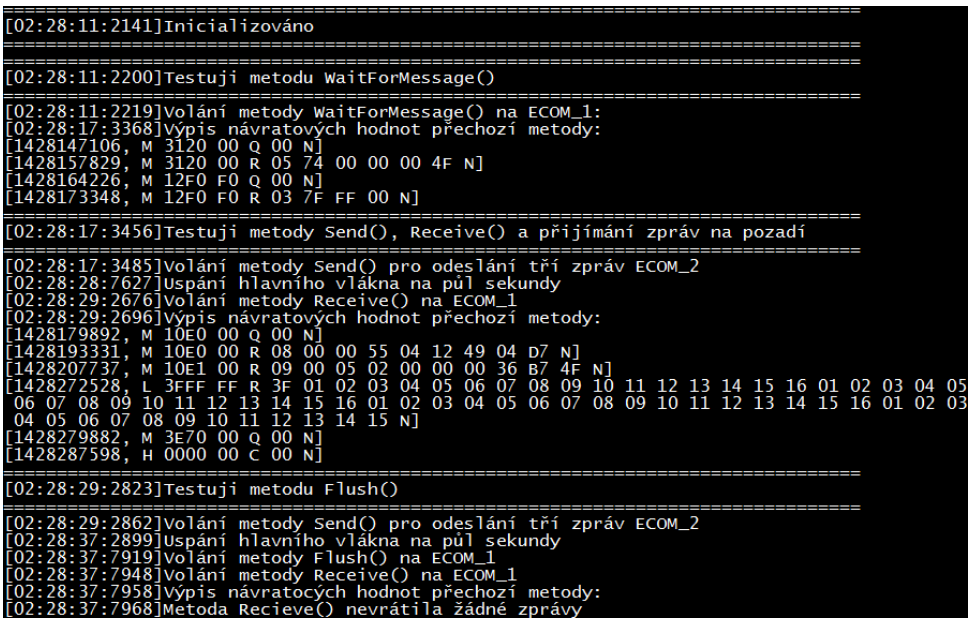
```
        }

        private void TestCase3(TestConfigurator tc, KeyValuePair<long, string>[]
            messages )
        {
            Console.WriteLine("======================================");
            Console.WriteLine("[{0}]Testuji metody Send(), Receive() a přijímání
                zpráv na pozadí", DateTime.Now.ToString("hh:mm:ss:ffff"));
            Console.WriteLine("======================================\r");
            Console.WriteLine("[{0}]Uspání hlavního vlákna na patnáct sekund pro
                přijetí věech zpráv z DHW na pozadí",
                DateTime.Now.ToString("hh:mm:ss:ffff"));
            Thread.Sleep(15000);
            Console.WriteLine("[{0}]Volání metody Send() pro odeslání tří zpráv
                ECOM_2", DateTime.Now.ToString("hh:mm:ss:ffff"));
            tc.Send("ECOM_2", "ECOM", new string[] { "L 1111 FF R 00", "M 2222 00
                Q 00", "H 3333 00 C 00" }, -1);
            Console.WriteLine("[{0}]Uspání hlavního vlákna na půl sekundy",
                DateTime.Now.ToString("hh:mm:ss:ffff"));
            Thread.Sleep(500);
            Console.WriteLine("[{0}]Volání metody Receive() na ECOM_1",
                DateTime.Now.ToString("hh:mm:ss:ffff"));
            messages = tc.Receive("ECOM_1", "ECOM");
            OutputWriteLine(messages);
        }
    }
}
```

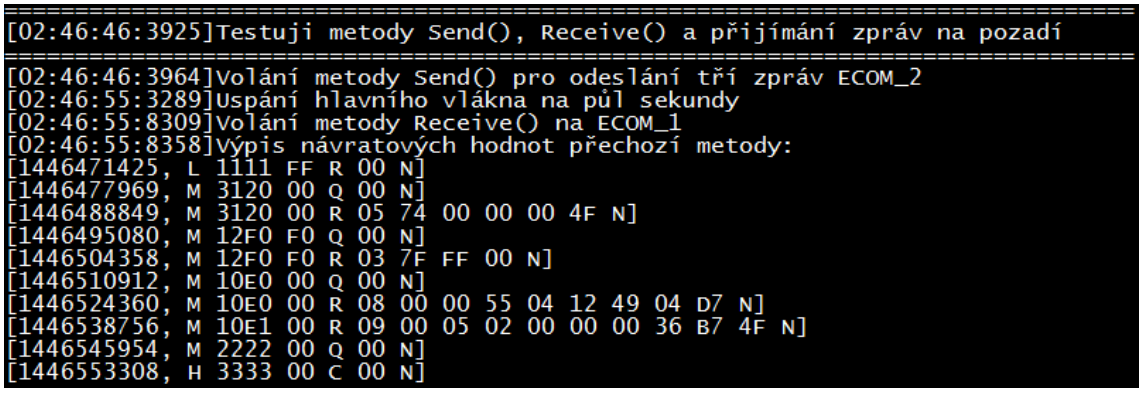
A.4 Testování ovladače pro EnviraCOM Serial Adapter

```
================================================================================
[01:21:06:6972]Inicializováno
================================================================================
================================================================================
[01:21:06:7001]Testuji metodu WaitForMessage()
================================================================================
[01:21:06:7020]Volání metody WaitForMessage() na ECOM_1:
[01:21:16:7186]Výpis návratových hodnot přechozí metody:
[01:21:16:7206]Metoda WaitForMessage() nevrátila žádné zprávy
================================================================================
[01:21:16:7235]Testuji metody Send(), Receive() a přijímání zpráv na pozadí
================================================================================
[01:21:16:7265]Volání metody Send() pro odeslání tří zpráv ECOM_2
[01:21:24:7321]Uspání hlavního vlákna na půl sekundy
[01:21:25:2341]Volání metody Receive() na ECOM_1
[01:21:25:2380]Výpis návratových hodnot přechozí metody:
[1321232154, L 3FFF FF R 3F 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 01 02 03 04 05
 06 07 08 09 10 11 12 13 14 15 16 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 01 02 03
 04 05 06 07 08 09 10 11 12 13 14 15 N]
[1321239664, M 3E70 00 Q 00 N]
[1321247341, H 0000 00 C 00 N]
================================================================================
[01:21:25:2488]Testuji metodu Flush()
================================================================================
[01:21:25:2507]Volání metody Send() pro odeslání tří zpráv ECOM_2
[01:21:33:2417]Uspání hlavního vlákna na půl sekundy
[01:21:33:7447]Volání metody Flush() na ECOM_1
[01:21:33:7564]Volání metody Receive() na ECOM_1
[01:21:33:7584]Výpis návratocých hodnot přechozí metody:
[01:21:33:7594]Metoda Recieve() nevrátila žádné zprávy
```

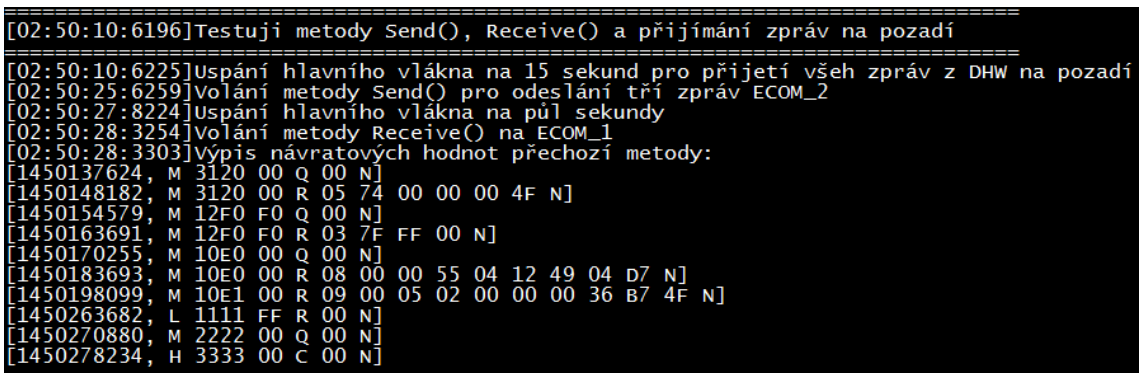
Obr. A.1: Základní test metod rozhraní IEnviracom



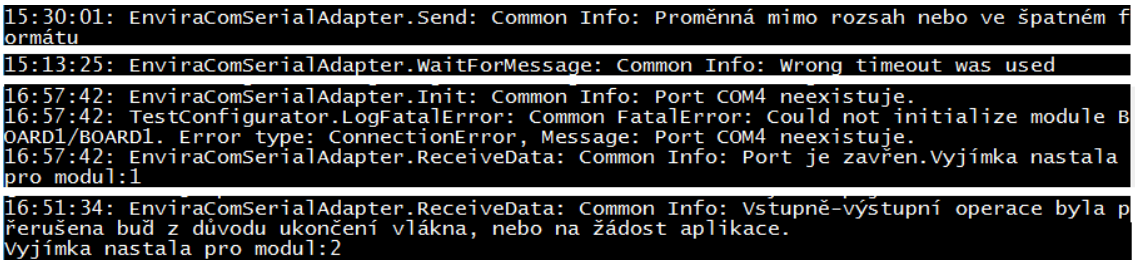
Obr. A.2: Základní test metod rozhraní IEnviracom, s příjmem zpráv z DHW



Obr. A.3: Test metody Send(), s příjmem zpráv z DHW



Obr. A.4: Test metody Send(), po příjmu zpráv z DHW



Obr. A.5: Test zalogování vyjímek

A.5 testConfig.xml

```
<AutoTestConfig>
  <device name="ECOM_1">
    <family name="EnviraComSerialAdapter">
      <module config="COM3" name="BOARD1" address="1">
        <interface name="IEnviracom">
          <channel config="" name="ECOM" address="0">
          </channel>
        </interface>
      </module>
    </family>
  </device>
  <device name="ECOM_2">
    <family name="EnviraComSerialAdapter">
      <module config="COM6" name="BOARD2" address="2">
        <interface name="IEnviracom">
          <channel config="" name="ECOM" address="0">
          </channel>
        </interface>
      </module>
    </family>
  </device>
  <logger config="" type="SimpleLogger"  />
</AutoTestConfig>
```

B OBSAH CD PŘILOŽENÉHO K BAKALÁŘSKÉ PRÁCI

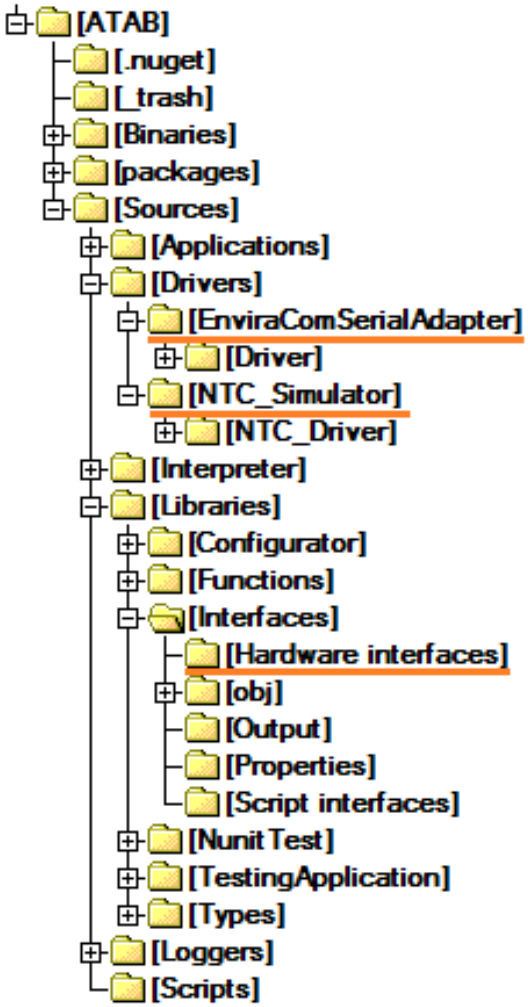
CD přiložené k bakalářské práci, obsahuje pouze text vlastní práce a návod pro přístup ke zdrojovým kódům. Zdrojové kódy ovladačů vytvořených v rámci bakalářské práce nemohou být přílohou této práce kvůli tomu, že se na ně vztahuje obchodního tajemství firmy Honeywell.

V případě oprávněné potřeby zdrojových kódů (například za účelem vypracování posudku této práce), k nim vedou dvě cesty; První je IS VUT kde jsou uloženy jako tajné soubory. Druhou je kontaktování Honeywell HTS Brno, Tuřanka 96, 62700 Brno, pan Radomír Svoboda, e-mail: radomir.svoboda@honeywell.com.

Na obrázku B.1, je zobrazena struktura složek zdrojových kódů, s vyznačenými stěžejními složkami. Zdrojový kód se skládá z jedné takzvané Solutiony ATAB, a několika projektů. Obsahuje ovladače pro NTC Simulator a EnviraCOM Serial Adapter, knihovny a rozhraní potřebné pro běh těchto ovladačů, aplikaci pro konfiguraci a spouštění testů s ukázkou testu ovladače pro modul EnviraCOM Serial Adapter a jednoduchý logger.

B.0.1 Požadavky pro spuštění kódu

Pro práci se zdrojovými kódy je nutné mít nainstalované Visual Studio Express 2012, či novější a .NET Framework 4.



Obr. B.1: Struktura složek zdojových kódů