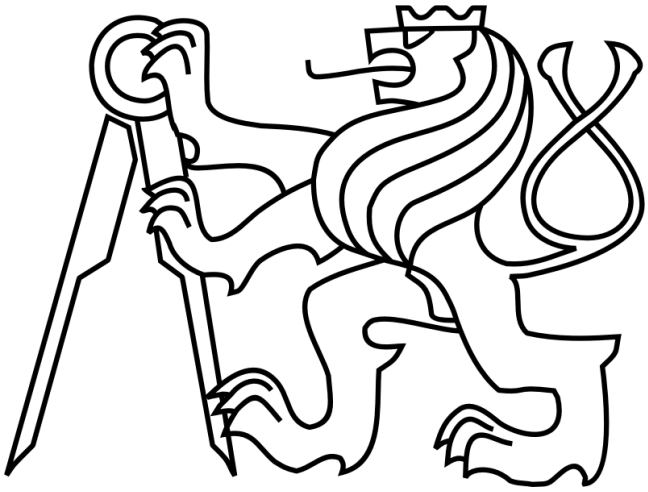
České vysoké učení technické
Fakulta elektrotechnická



Diplomová práce

Katedra řídicí techniky

Testování integrace HW a SW v elektronickém zabezpečovacím zařízení

Akademický rok
2012/2013

Jméno
Miroslav Matějů

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Miroslav Matějů**

Studijní program: Otevřená informatika (magisterský)
Obor: Počítačové inženýrství

Název tématu: **Testování integrace HW a SW v elektronickém zabezpečovacím zařízení**

Pokyny pro vypracování:

1. Seznamte se s normami pro vývoj bezpečnostně-kritického softwaru a hardwaru pro drážní aplikace EN 50128, EN 50129, příp. EN 61508.
2. Proveďte analýzu možných způsobů provádění testů integrace HW a SW v modelovém projektu.
3. Na základě provedené analýzy navrhněte vhodný způsob implementace testů integrace HW a SW.
4. Implementujte a vyhodnoťte navržené testy pro konkrétní kombinaci HW a SW.
5. Vše důkladně zdokumentujte.

Seznam odborné literatury:

EN 50128
EN 50129
EN 61508

Vedoucí: Ing. Michal Sojka, Ph.D.

Platnost zadání: do konce letního semestru 2013/2014

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

L.S.

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 23. 11. 2012

Anotace

Tato diplomová práce pojednává o technice testování, jejímž účelem je ověřit správnou spolupráci mezi navrženým hardwarem a softwarem při vývoji systému s požadavkem na bezpečnost s aplikací na železnici. Popisuje související technické normy, analýzu možného řešení a implementaci hardwarových a především softwarových nástrojů, které lze pro testování integrace hardwaru a softwaru použít.

Annotation

This diploma thesis discusses the testing technique designed to verify the proper cooperation between designed hardware and software during the development of a system with safety requirements applicable on the railways. It contains a description of the related technical standards, analysis of possible solutions and a description of implementation of hardware and especially software tools that can be used to test the integration of hardware and software.

Poděkování

Děkuji **Ing. Vítu Fáberovi, Ph.D.**, z Fakulty dopravní ČVUT za zapůjčení vývojového kitu Atmel STK500 pro návrh prototypu pomocného hardwaru k této diplomové práci.

Děkuji všem, kteří vytvořili a poskytli zdarma software a další duševní díla, které umožnily vznik této diplomové práci. Především tvůrcům programů Qt, Git, Doxygen, OpenOffice.org a rovněž Střešovické písmolijně za uvolnění fontu Lido STF, jímž je vysázen tento dokument.

Děkuji rovněž všem, kteří dávají své znalosti k dispozici ostatním lidem na webových serverech StackOverflow, Wikipedie a dalších.

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 10. května 2013 ..... Miroslav Malajčí .....

podpis

Obsah

1. Úvod

Tato diplomová práce se zabývá testováním prováděným při vývoji železničních zabezpečovacích zařízení v jejich nejmodernější, tedy elektronické implementaci.

Historie železničních zabezpečovacích systémů sahá až k polovině 19. století. Od té doby se vyvinuly ze systémů čistě mechanických přes využití kombinovaných elektromechanických zařízení, jako jsou hradlové vložky[1] a později elektromagnetická relé, až k převážně elektronickým systémům, které se používají dnes. S měnícím se provedením se vyvíjely i způsoby ověřování jejich správné činnosti, včetně testování. U mechanického zařízení, v němž byly logické podmínky vyhodnocovány pomocí vzájemného zapadání ocelových lišt do sebe, bylo na první pohled zřejmé, k jakým situacím může dojít, a zajištění bezpečnosti[1] proto sestávalo z dodržení jasně daných pravidel, jejichž ověření bylo často možné pouhým pohledem. Později byla snaha uplatňovat podobná pravidla i u zařízení s reléovou logikou. Zde se ovšem v průběhu času začaly kumulovat empirické důkazy o nedostatečnosti tohoto přístupu. Například se ukázalo, že reléová paměťová buňka může změnit svůj stav vlivem atmosférických elektrických výbojů. To přitom může mít fatální důsledky: Traťový oddíl, který není vybaven technickým zařízením pro kontrolu volnosti, se může začít jevit jako volný, přestože je ve skutečnosti obsazen vlakem. Zařízení v takovém případě dovolí vjezd dalšího vlaku do téhož oddílu.

Po příchodu integrovaných elektronických obvodů do oboru železniční zabezpečovací techniky[2] se při jejím vývoji začaly používat stejné techniky jako v jiných oborech, kde se vyskytují požadavky na bezpečnost. Díky výzkumu prováděnému v oblasti železničních zabezpečovacích systémů[1] a rovněž díky využití zkušeností získaných při vývoji bezpečných systémů v jiných aplikačních oborech již bylo zcela zřejmé, že je nutné používat techniky pro zajištění bezpečnosti, jako jsou redundance elektronických logických členů, opakované vyhodnocování podmínek a pravděpodobnostní výpočty. Mezi techniky zajištění bezpečnosti patří i testování hardwaru a softwaru elektronických zabezpečovacích zařízení. Jedním z druhů testů, které je potřeba provádět, je i testování integrace hardwaru a softwaru, jímž se zabývá tento dokument.

Testování integrace hardwaru a softwaru má za cíl ověřit, že hardware a software, jejichž vývoj do určité fáze probíhá zpravidla odděleně, jsou schopny po sloučení do cílového systému plnit v předpokládaném rozsahu bezpečnostní funkce. Například komunikace pomocí

1 Zde i v dalších místech tohoto textu je myšlen především význam odpovídající anglickému slovu *safety*, tedy zajištění systému tak, aby svou (ne)činností nezpůsobil škody na životech, majetku a životním prostředí.

2 V České republice k tomu došlo v průběhu 90. let 20. století.

průmyslových sběrnic se při vývoji samostatného softwaru nahrazuje emulací prostřednictvím protokolu IP a programy jednotlivých procesorů se spouští jako procesy na jednom osobním počítači. Úkolem je v takovém případě otestovat, jak se systém chová, pokud je vyvinutý software přenesen na samostatné procesory, které spolu komunikují po průmyslových sběrnicích, a zároveň otestovat kód, který tuto komunikaci zajišťuje.

Cílem této práce je vytvořit softwarový nástroj, které by umožňoval opakované a automatické provádění výše popsaných integračních testů, a rovněž podílet se na vývoji souvisejícího podpůrného hardwaru.

V kapitole 2 je stručně představen modelový projekt, pro nějž je nástroj vytvářen. V kapitole 3, která se zabývá technickými normami, jsou předvedeny požadavky na testovací nástroje tohoto typu. Kapitola 4 obsahuje analýzu, jak vytyčené požadavky při vývoji konkrétního nástroje naplnit. V kapitole 5 je pak zdokumentována provedená implementace softwarového nástroje a kapitola 6 demonstruje, jakému testování byl vyvinutý software podroben. Výstup této diplomové práce je shrnut v závěru – kapitole 7.

2. Modelový projekt

Softwarový nástroj, který má v rámci této práce vzniknout, je určen pro použití v konkrétním existujícím projektu, který je vyvíjen v komerční firmě, a je tedy požadována jistá úroveň ochrany know-how. Proto bude pro účely této práce zaměněn za obecnější modelový projekt popsaný níže, který ovšem respektuje omezení daná skutečným projektem. Pokud se dále v tomto dokumentu mluví o projektu a je z kontextu zřejmé, že se jedná o konkrétní instanci projektu (což se samozřejmě netýká například citací norem), je tímto označením myšlen právě modelový projekt.

Projekt, pro nějž má být v rámci této práce připraveno integrační testování softwaru/hardwaru, je generický produkt z oboru železničních zabezpečovacích systémů ve fázi vývoje, sestávající z generického softwaru a k němu příslušejícího hardwaru. To znamená, že software, který testované zařízení obsahuje, nedokáže samostatně vykonávat žádnou řídicí činnost, ale poskytuje zabezpečené prostředí pro celou řadu aplikací s požadavkem na bezpečnost. Doplněním aplikačních algoritmů k tomuto generickému softwaru dojde k vytvoření generické aplikace, která již zajišťuje fungování třídy zařízení, jakými mohou být například stavědlo, traťové zabezpečovací zařízení, železniční přejezd atd. Pro konkrétní instalaci se systém dále upřesňuje (specifikují se například počty a umístění použitých komponent, jako jsou výhybky

a návěstidla) a vzniká specifická aplikace. Vývoj generických i specifických aplikací je ovšem nad rámec modelového projektu, který se zabývá pouze generickým produktem.

Projekt je zpracováván pro úroveň integrity bezpečnosti SIL 4 (viz kapitolu 3.1.5), a vztahují se na něj tedy nejpřísnější varianty z požadavků uvedených v normách.

3. Technické normy

Technické normy, které se týkají bezpečnosti elektronických zabezpečovacích systémů používaných na železnici, jsou EN 50128, EN 50129. Nad nimi stojí ještě norma EN 50126, zabývající se železničním systémem jako celkem, v němž je zabezpečovací systém pouze subsystémem. Ta například popisuje analýzu rizik, která je součástí vývoje zabezpečovacího systému [2]. Norma EN 50129 se zaobírá schvalováním jednotlivých zabezpečovacích systémů, zatímco EN 50128 se specializuje na software.

Normy pro železniční zabezpečovací systémy připravuje Evropská komise pro normalizaci v elektrotechnice – CENELEC.

3.1. EN 50129

Nejprve se zmíním o obecnější normě EN 50129. Aktuální verze normy vyšla v roce 2003 a nahradila předchozí z roku 1998. Tuto normu mám k dispozici pouze v angličtině, proto u některých pojmů, jejichž překlad nemusí být zcela zřejmý, uvádím v závorce původní anglický výraz.

Cílem normy je sjednocení požadavků na bezpečnostně relevantní elektronické systémy v oboru železniční zabezpečovací techniky. Zabývá se funkční bezpečností nově vyvíjených, modifikovaných a rozšiřovaných železničních zabezpečovacích systémů[3]. Samotná norma bez příloh není moc obsáhlá a definuje několik základních dokumentů a postupů v souvislosti se zabezpečovacími systémy.

Jedním z nejdůležitějších je pojem **bezpečnostní případ** (*safety case*), což je dokument popisující konkrétní systém, subsystém nebo zařízení podílející se na bezpečnosti. Obsahuje definici systému, zprávu o řízení kvality (*quality management*), zprávu o řízení bezpečnosti (*safety management*), zprávu o technické bezpečnosti, související bezpečnostní případy a závěr.

3 Norma EN 50129 obvykle uvádí „systém/subsystém/zařízení“, kladené požadavky jsou ovšem stejné. Místo tohoto rozlišování proto uvádím jednoduše „systém“, protože i na „subsystém“ nebo „zařízení“ (*equipment*) lze pohlížet na jiné rozlišovací úrovni jako na „systém“.

3.1.1. Řízení kvality

Řízení kvality je podle normy důležité proto, aby se minimalizoval vliv lidských chyb v každé fázi životního cyklu produktu. Řízení kvality tedy má být součástí celého životního cyklu. Je zde také uveden příklad, jak může životní cyklus vypadat; je zobrazen sekvenčně jako postupný sled činností.

3.1.2. Řízení bezpečnosti

Řízení bezpečnosti je součástí procesu řízení RAMS (*reliability*, *availability*, *maintainability*, *safety* – spolehlivost, dostupnost, udržovatelnost, bezpečnost), definovaného v EN 50126.

Je zde definován **bezpečnostní životní cyklus** (safety life-cycle), který výše uvedený sekvenční životní cyklus doplňuje o úrovně a stanovuje vazby mezi první polovinou životního cyklu, která se zabývá návrhem na stále podrobnější úrovni, a druhou polovinou, v níž se vznikající systém sestavuje dohromady a při tom probíhají kontroly na stále vyšší úrovni. Je jasně vidět, jak se návrh vzniklý na jisté rozlišovací úrovni později na téže úrovni kontroluje – čím dříve návrh vznikl a čím je obecnější, tím později bude jeho dodržení kontrolováno. Bezpečnostní životní cyklus pak jako celek dostává tvar písmena V.

Řízení bezpečnosti zahrnuje pod pojmem **bezpečnostní organizace** (*safety organisation*) také rozdělení kompetencí ve vývojovém týmu. Toto rozdělení se liší podle úrovně integrity bezpečnosti (*safety integrity level*, SIL) a obecně lze říct, že s rostoucí úrovní je vyžadována větší nezávislost – různé kompetence nemůže vykonávat stejná osoba, nebo dokonce osoby s různými kompetencemi nemohou mít ani téhož nadřízeného.

Další součástí řízení bezpečnosti je **bezpečnostní plán**, který vzniká na začátku životního cyklu a popisuje bezpečnostně relevantní aktivity a milníky schvalování v průběhu životního cyklu. Ten má, mimo jiné, popisovat plány pro kontrolu systému, rozdělenou na verifikaci a validaci. Pojem **verifikace** označuje ověření, že kontrolovaná fáze životního cyklu naplňuje bezpečnostní požadavky identifikované v předchozí fázi. **Validace** ověřuje dokončený systém vůči jeho původní specifikaci bezpečnostních požadavků. Verifikace a validace musí obsahovat odpovídající testování a bezpečnostní analýzy.

Kromě vývoje popisuje bezpečnostní plán i stádium předání železniční autoritě, provoz a údržbu, přičemž se při požadavku na změny může konat opakování části životního cyklu, a také vyřazení z provozu a likvidaci zařízení.

3.1.3. Důkaz funkční a technické bezpečnosti

Zpráva o technické bezpečnosti, součást bezpečnostního případu, musí vysvětlit technické principy zjišťující bezpečnost návrhu. Sama o sobě má několik částí, například Zajištění správného funkčního provozu (*Assurance of correct functional operation*), dokazující správnou funkci za normálních podmínek bez chyb, Účinky chyb (*Effects of faults*), Provoz s vnějšími vlivy (*Operation with external influences*) nebo Bezpečnostně relevantní podmínky použití (*Safety-related application conditions*). Ani zde ovšem nejsou pojmy funkční a technické bezpečnosti přesně definovány.

3.1.4. Generický produkt a generická aplikace

Podle normy se může uvažovat o třech kategoriích bezpečnostního případu, lišících se svou obecností.

Nejobecnějším případem je **generický produkt**, který je nezávislý na aplikaci a může být opakovaně použit pro různé, vzájemně nezávislé aplikace. Doplněním generického produktu o sdílené aplikační funkce vzniká **generická aplikace**, která může být opakovaně použita pro třídu aplikací se společnými funkcemi. Nejkonkrétnějším případem je potom **specifická aplikace**, která může být použita pouze pro jednu konkrétní instalaci.

3.1.5. Úrovně integrity bezpečnosti

Norma ve své příloze A definuje integritu bezpečnosti a její úrovně SIL. Bezpečnostní požadavky jsou rozděleny na funkční a integritní. Funkční požadavky jsou vlastní bezpečnostně relevantní funkce, které má systém vykonávat. Integritní požadavky stanovují požadavky na úroveň integrity. **Integrita bezpečnosti** je popisována jako schopnost plnit požadované funkce bez selhání způsobených systematickými nebo náhodnými chybami. Obrana proti systematickým, tedy nekvantifikovatelným chybám je založena na dodržování určených metod, nástrojů a technik. Integrita proti náhodným chybám je zaručena pravděpodobnostními výpočty.

Stejně jako další dokumenty, norma rozlišuje 5 diskrétních úrovní integrity (*safety integrity level*, SIL) – SIL 0 až SIL 4. SIL 0 označuje neexistenci bezpečnostních požadavků, SIL 4 je úroveň s nejvyššími požadavky. Jsou stanovena rozmezí odpovídající jednotlivým SIL pro dovolený výskyt náhodných poruch, vyjádřený veličinou Přípustná míra rizika (*tolerable hazard rate*, THR), která je vztažena na hodinu provozu a jednotlivou bezpečnostní funkci. Pokud jde o prevenci systematických chyb, jsou k úrovním integrity přiřazeny techniky, které se mají použít.

Příloha E obsahuje seznam takových technik. Doporučená technika je přitom pro úrovně SIL 1 a 2 velmi často totožná, pro SIL 3 a 4 je totožná ve všech případech.

3.2. EN 50128

Normu EN 50128 zabývající se softwarem pro drážní řídicí a ochranné systémy jsem měl k dispozici jednak ve formě návrhu z roku 2009, jednak jako český překlad finální verze EN 50128:2011 označený jako ČSN EN 50128 ed. 2. Pojmy obsažené v tomto dokumentu používám primárně v češtině podle oficiálního překladu ČSN.

Norma EN 50128:2011 nahrazuje předchozí verzi EN 50128:2001, která se smí používat maximálně do dubna roku 2014. Oproti ní doplňuje požadavky na management softwaru a organizaci, definice rolí a kompetencí, nasazení a údržbu a také kapitolu o nástrojích založenou na EN 61508-2:2010 [3].

Jak už jsem zmínil dříve, oblastí zájmu této normy je software v drážních zabezpečovacích[4] systémech. Stejně jako v nadřazených normách jsou pro stanovení integrity software použity úrovně SIL. Norma EN 50128 se v souvislosti s integritou zaměřuje na prevenci systematických chyb, a stanovuje tedy techniky, které se mají pro danou úroveň integrity bezpečnosti používat. A pokud jsem u EN 50129 uvedl, že se požadované techniky často shodují jednak pro úrovně SIL 1 a 2, jednak pro SIL 3 a 4, norma EN 50128 přímo uvádí, že požadované techniky pro SIL 1 a 2 jsou shodné, stejně tak pro SIL 3 a 4. V podstatě se tedy v této oblasti rozlišují jen tři úrovně integrity bezpečnosti místo pěti.

3.2.1. Role a odpovědnosti

Podobně jako EN 50129, i EN 50128 stanovuje rozdělení rolí a odpovědností mezi pracovníky účastnící se vývoje systému. Zde jsou ovšem podrobnější, konkretizované vzhledem k procesu vývoje softwaru. Norma rozlišuje následující role:

- manažer projektu,
- manažer pro požadavky,
- návrhář,
- implementátor,
- hodnotitel,
- integrátor,
- tester,
- verifikátor,
- validátor.

4 Norma uvádí označení „řídicí a ochranné systémy“, v češtině je ovšem v praxi zažito používat pro tyto systémy, které zpravidla plní obě funkce zároveň, název „zabezpečovací systémy“.

Dále jsou uvedeny zákazy krytí více rolí u jednotlivých osob. Pro SIL 0 platí rozdělení přibližně odpovídající sloupcům výše. Pro SIL 1 a 2 se nemají krýt role integrátora a testera s dalšími, stejně tak role verifikátora a validátora. Pro SIL 3 a 4 se dále nesmí krýt role verifikátora a validátora, validátor navíc nesmí být podřízený manažera projektu. Hodnotitel musí být pro všechny úrovně zcela mimo vyvíjející organizaci (pokud nedovolí výjimku orgán pro otázky bezpečnosti, ale i v takovém případě musí být hodnotitel nezávislý na daném projektu).

Pokud jde o role ve smyslu normy, tato diplomová práce je zpracována z pozice testera. Ten podle normy provádí „spouštění softwaru za kontrolovaných podmínek s cílem zjistit jeho chování a výkonnost v porovnání s odpovídajícími specifikovanými požadavky“ [3].

3.2.2. Životní cyklus softwaru

Životní cyklus softwaru je, podobně jako v EN 50129 životní cyklus celého systému, uveden jednak v sekvenční formě, jednak v modelu tvaru písmena V, jehož sestupnou hranu tvoří fáze analýzy a návrhu, zlom ve spodní části modelu patří implementaci a vzestupná hrana odpovídá testování a validaci. Nyní jsou již ovšem specifikovány konkrétní vstupní a výstupní dokumenty pro jednotlivé úrovně, které prokazují, že příslušná etapa proběhla správně.

3.2.3. Testování softwaru

Z celého životního cyklu se podrobněji zaměřím pouze na testování softwaru, které je hlavní náplní této diplomové práce. Testování provádí tester nebo integrátor s cílem „ověřit chování nebo výkon softwaru proti odpovídající specifikaci testu v rozsahu dosažitelném zvoleným pokrytím testu“ [3]. Vstupní dokumentací pro testování je veškerá dokumentace systému, hardwaru i softwaru specifikovaná v plánu verifikace softwaru. Výstupní dokumenty jsou tvořeny dvojicemi „specifikace testování“ a „zpráva z testování“ týkajícími se celkového testování, testů integrace softwaru, integrace softwaru/hardwaru a testů softwarových komponent.

Jsou uvedeny požadavky na měřicí zařízení a použité nástroje a také na specifikace testů a zprávy z testů. Tyto dokumenty by měly být zaznamenány ve strojově čitelné formě, testy musí být opakovatelné a v rámci možností automaticky proveditelné.

3.2.4. Verifikace a validace

Výstupy z testování jsou, stejně jako všechny další výstupy projektu, verifikovány a také se uplatňují při validaci, proto ještě zmíním definice těchto činností v této normě:

*Cílem **verifikace** softwaru je přezkoušet a dospět k názoru založeném na důkazu, že položky výstupu (proces, dokumentace, software nebo aplikace) určité etapy vývoje splňují požadavky a plány s ohledem na úplnost, správnost a konzistenci. Tyto činnosti jsou řízeny verifikátorem* [3].

*Cílem **validace** je ukázat, že procesy a jejich výstupy jsou takové, že software má definovanou úroveň integrity bezpečnosti softwaru, splňuje požadavky na software a odpovídá zamýšlenému použití. Tato činnost je prováděna validátorem.*

Hlavní činnosti validace mají ukázat analýzou a/nebo testováním, že jsou všechny požadavky na software specifikovány, implementovány, testovány a splněny tak, jak je vyžadováno příslušnou úrovní SIL a vyhodnotit bezpečnostní kritičnost všech odchylek a neshod založených na výsledcích přezkoumání, analýz a testů ve vztahu k bezpečnosti [3].

Výsledky validace poté ještě podléhají hodnocení nezávislým hodnotitelem. Z tak důkladného ověřování vyplývají vysoké požadavky na úplnost, správnost a shodu s normami pro všechny činnosti, včetně testování.

3.2.5. Podpůrné nástroje a jazyky

Norma klade určité požadavky i na použité nástroje a (programovací) jazyky. Podle míry, v jaké se tyto nástroje uplatní ve finálním výrobku, jsou rozděleny do třech tříd:

- Třída T1 sdružuje nástroje, jejichž výstupy neovlivní spustitelný kód (včetně dat) výrobku.
- Třída T2 slouží k podpoře testování nebo verifikace, přičemž chyby nástroje mohou způsobit neodhalení nedostatků, ale nemohou přímo ovlivnit spustitelný software.
- Třídu T3 tvoří nástroje, které přispívají ke spustitelnému kódu (včetně dat) výrobku. Typicky jsou to překladače, generátory dat a podobně.

Z tohoto rozdělení je zřejmé, že programové vybavení použité k testování lze zařadit do třídy T2. Proto se na něj vztahují následující požadavky [3]:

- *Softwarové nástroje musí být vybrány tak, aby tvořily koherentní součást činností vývoje softwaru.*
- *Výběr nástrojů ze tříd T2 a T3 musí být odůvodněný. Toto zdůvodnění musí obsahovat identifikaci potencionálních poruch, které mohou být vneseny do výstupů těchto nástrojů a opatření k vyvarování se nebo k zvládnutí těchto poruch.*

- *Všechny nástroje tříd T2 a T3 musí mít specifikaci nebo manuál, které jasně definují chování tohoto nástroje a jakékoliv instrukce nebo omezení související s jejich použitím.*
- *Při řízení konfigurace musí být zaručeno, že pro nástroje ze tříd T2 a T3 jsou použity pouze oprávněné verze.*
- *Každá nová verze nástroje, která je použita, musí být odůvodněna. Toto odůvodnění se může spolehnout na důkazy podané pro dřívější verze, jestliže*
  - *funkční rozdíly (pokud nějaké jsou) neovlivní kompatibilitu nástroje se zbytkem sady nástrojů;*
  - *je nepravděpodobné, že nová verze nástroje obsahuje významné nové neznámé chyby.*

Pro testování je tedy možné použít pouze nástroje, které výše uvedené podmínky splňují. Přednost tedy mají nástroje, které se již v projektu používají, pokud to bude možné – jednak aby byla splněna první uvedená podmínka, jednak aby se nemusely výše uvedené činnosti provádět znovu pro jednu partikulární část vývoje. Z požadavků navíc vyplývá, že záleží i na verzi použitého nástroje.

3.2.6. Specifikace integračních testů

Za specifikaci testů integrace softwaru i softwaru/hardwaru zodpovídá integrátor. Podle specifikace testů integrace hardwaru/softwaru musí být prokázáno, že software běží na hardwaru správně při využívání hardwaru přes předepsaná hardwarová rozhraní, že software je schopen zvládnout hardwarové vady tak, jak je požadováno, a musí být prokázáno předepsané časování a výkon [3]. Specifikace testů integrace softwaru/hardwaru musí dokumentovat testovací případy a data, typy testů, prostředí testů (včetně nástrojů podpůrného software a popisu konfigurace) a kritéria testu.

3.2.7. Integrace

Nyní se již dostáváme k samotné integraci, u níž norma sleduje především testování integrace jak samotného softwaru, tak softwaru/hardwaru. Vstupními dokumenty této fáze jsou specifikace testů integrace softwaru a integrace softwaru/hardwaru. K výstupům pak kromě zpráv z těchto testů patří také zpráva z verifikace integrace softwaru.

Integrace je popsána jako proces postupného slučování jednotlivých dříve testovaných komponent, aby mohla být rozhraní komponent a sestaveného softwaru dostatečně ověřena před

provedením systémové integrace a testů [3]. Zprávy z testů integrace jsou, stejně jako jejich specifikace, v zodpovědnosti integrátora. Musí prokázat korektní použití zvolených technik a opatření. V případě výskytu poruchy musí být zaznamenány okolnosti. Také se uplatní obecné požadavky na zprávy z testů [3].

- *Zpráva z testu musí uvádět jména testerů, výsledky testu a zda byly splněny cíle a kritéria testu, které jsou uvedeny ve specifikaci testu. Poruchy musí být shrnuty a dokumentovány.*
- *Testovací případy a jejich výsledky musí být zaznamenány, nejlépe ve strojově čitelné formě pro následnou analýzu.*
- *Testy musí být opakovatelné, a pokud je to možné, prováděné automaticky.*
- *Skripty testu pro automatické spuštění testu musí být verifikovány.*
- *Identita a konfigurace všech položek zahrnutých v testu (použitý hardware, software, zařízení, kalibrace zařízení, informace o verzi specifikace testu) musí být dokumentovány.*
- *Musí být poskytnuto vyhodnocení pokrytí kódu testem a úplnosti testu a musí být zaznamenány všechny odchylky.*[5]

Integrační testování se podle normy skládá ze dvou kategorií: funkčního testování/testování černé skříňky (to je uvedeno jako velmi doporučené (*highly recommended*, HR) pro všechny úrovně integrity) a testování výkonnosti, které je nepovinné pro SIL 0, doporučené pro SIL 1 a 2 a velmi doporučené pro SIL 3 a 4.

Z funkčních testů a testů černé skříňky norma uvádí následující techniky [3]:

- *provedení testovacích případů z diagramů příčina–následek (doporučeno pro SIL 3 a 4);*
- *vytváření prototypů / animací (doporučeno pro SIL 3 a 4);*
- *analýza mezní hodnoty (doporučeno pro SIL 0, pro vyšší velmi doporučeno);*
- *třídy ekvivalence a testování rozkladem vstupů (doporučeno pro SIL 0, pro vyšší velmi doporučeno);*
- *simulace procesu (doporučeno pro všechny úrovně integrity).*

Pro testování výkonnosti jsou uvedeny následující techniky [3]:

5 Překlad tohoto bodu, který není v [3] úplně jasný, byl upraven s přihlédnutím k [4].

- *lavinové/zátěžové testování (doporučeno pro SIL 1 a 2, velmi doporučeno pro SIL 3 a 4);*
- *řízení doby odezvy a omezení paměti (velmi doporučeno pro SIL 1 a vyšší);*
- *požadavky na výkonnost (velmi doporučeno pro SIL 1 a vyšší).*

4. Analýza způsobů provádění testů

Tato část má za cíl shrnout a zhodnotit dříve definované požadavky a stanovit nejvhodnější způsob jejich naplnění. Také se zde objeví nové požadavky, které z vyhodnocování základních požadavků vyplynou. Analýza je ve shodě s normami zpracovávána metodou shora dolů.

4.1. Architektura hardwaru testovacího systému

Základem architektury testovacího systému je testovaný systém (viz obrázek 1), který je doplněn o zařízení nutná k provedení testu. Testovaný systém je tvořen *kartami* – deskami plošných spojů (DPS), na nichž jsou umístěny výkonné prvky (mikroprocesory s příslušnými periferiemi). Napájení a komunikační propojení karet je zajištěno pomocí DPS zvané *backplane*.

Obrázek 1: Architektura hardwaru testovaného systému

Kvůli testování je nutné zajistit každé kartě spojení se zařízením řídícím test. Toto spojení má za úkol nahrávat na kartu testovací program a během jeho běhu vyčítat proměnné, které informují o průběhu testu, jak je stanoveno v testovacích případech. Takové spojení je možné zajistit pomocí existujícího způsobu komunikace přes rozhraní Card Access Point (CAP).

Dále je ovšem nutné zajistit přerušování komunikačních a napájecích spojů mezi kartou a backplane. Přerušování komunikačních rozhraní je požadováno testovacími případy; přerušení napájení je nutné k tomu, aby se v mikroprocesoru karty spustil zavaděč (bootloader), který zajistí přijetí nového programu posílaného přes CAP. Po nahrání programu do karty je opět nutné

přerušit napájení, čímž se zajistí nový start zavaděče, který nyní zajistí rozběhnutí nahraného programu. Tyto zásahy tedy vyžadují vložení speciálního pomocného hardwaru mezi kartu a backplane.

4.1.1. Rozpojovací zařízení

Rozpojovací zařízení, umístěné mezi kartou a backplane, musí poskytovat konektory odpovídajícími běžnému spojení karta–backplane. Jinými slovy pro připojení k backplane musí být použit stejný konektor, jako má karta, a pro připojení ke kartě stejný konektor, jako má backplane.

Mezi těmito dvěma konektory musí rozpojovací zařízení umožnit buď přímé spojení odpovídajících si kontaktů, nebo přerušení tohoto spojení na základě povelu ze zařízení řídícího test. Je tedy nutné zajistit rozpojitelnost těchto propojení pomocí elektromagnetického relé nebo elektronických zařízení s podobnou funkcí (tranzistor, přenosové hradlo, relé v pevné fázi, ...). Za tímto účelem bylo vyzkoušeno relé v pevné fázi, avšak zjistilo se, že pokud se na jeho vstup přivede periodický obdélníkový signál odpovídající svou frekvencí datové komunikaci mezi kartami, objeví se na výstupu relé zákmity, které by mohly být přijímačem interpretovány jako platný signál. Proto padla volba na elektromagnetické relé, které zajistí spolehlivé rozpojení pro všechny frekvence signálu.

Dále je nutné stanovit umístění těchto relé. V úvahu připadá umístění mimo rack (tj. soustavu backplane a k ní připojených karet) na jednom místě, kde by byla soustředěna relé pro všechna rozhraní všech použitých karet. V tomto případě by bylo možné použít některou z komerčně dostupných sad relé ovládatelných přímo z PC (viz např. [5]). Toto řešení by ovšem vyžadovalo vyvedení komunikačních rozhraní mimo rack, s čímž by bylo spojeno riziko zvýšeného elektromagnetického rušení. Proto bylo upřednostněno vytvoření DPS s relé, která se vloží do racku přímo mezi backplane a kartu.

Mimo rack je v tomto případě vyvedeno pouze napájení relé a datové spojení, které zajišťuje řízení relé. Tento datový spoj může pracovat na mnohem nižší frekvenci než samotná komunikace karet, čímž je riziko rušení sníženo. Jako nejjednodušší způsob realizace tohoto spoje byla zvolena linka SPI (*Serial Peripheral Interface*), která je přivedena do kaskádně řazených posuvných registrů se sériovým vstupem (SIN) a paralelními výstupy (POUT) 74HC595. Kromě rozhraní SPI je u těchto posuvných registrů nutné obsloužit hodinový signál RCLK, jímž se předává povel k přenosu dat ve vlastním posuvném registru do záchytného registru a na paralelní

výstupy. Signál nG je trvale zapojen na nulové napětí, takže data v záchytném registru se vždy přenáší na paralelní výstupy (POUT).

Elektromagnetické relé, které bylo zvoleno, je ovšem rozměrnější než relé v pevné fázi, takže bylo pro zachování rozměru DPS vhodného pro vložení do racku nutné zvolit pouze některá rozhraní, která půjdou rozpojit, případně jen některé signály těchto komunikačních rozhraní. Testovací případy sice pro testování integrace vyžadují rozpojování jen jednoho druhu komunikačního rozhraní, přesto se návrhář hardwaru pro toto testování rozhodl umožnit rozpojování všech rozhraní. Většina používaných rozhraní je synchronních – s datovým a hodinovým signálem, a tak bylo na základě analýzy požadavků na zařízení rozhodnuto, že pro účely testování integrace je ke zneplatnění datového signálu dostatečným opatřením rozpojení signálu hodin.

4.1.2. Zařízení řídící test

V testovacím systému by mělo existovat jedno zařízení, které bude shromažďovat informace od ostatních komponent (jak testovaných, tak pomocných) a udávat jim pokyny. Tímto způsobem je možné zajistit vykonávání testu podle předpisu testu a rovněž centrální záznam jeho výsledků.

V daném prostředí jsou na zařízení řídící test kladeny různorodé nároky. Jak bylo uvedeno výše, toto zařízení by mělo prostřednictvím rozhraní SPI řídit relé ovládající stav komunikačních

rozhraní. Toto rozhraní je typicky zahrnuto v mikrořadičích. Protože má být možné integrační testy spouštět opakovaně automaticky i pro nové verze testovaného softwaru, je vhodné, aby mělo zařízení řídící test přístup k repositáři zdrojových kódů na firemním intranetu a dokázalo je odtud stáhnout, zkompilovat a nahrát do testovaných karet.

Proto se ukazuje, že jako zařízení řídící test musí být použit osobní počítač (*Personal Computer*, PC), který jako jediný splňuje nejvíce omezující podmínku, kterou je schopnost kompilovat zdrojové kódy. Kompilátor pro příslušnou cílovou platformu je ve firmě dostupný pouze ve verzi pro PC s operačním systémem Microsoft (dále jen MS) Windows. Pokud je použit osobní počítač, jsou již s minimální dodatečnou prací splněny podmínky přístupu k repositáři (zajištěno existujícím klientem systému pro správu zdrojových kódů (*Source Code Management*, SCM) a přístupem k intranetu), správy předpisů testů a zaznamenávání výsledků (zajištěno pevným diskem a souborovým systémem) a přístupu k testovaným kartám přes CAP (existující převodník mezi USB a CAP).

V případě PC ovšem není k dispozici rozhraní SPI. Proto je nutné použít převodník ve formě mikrořadiče, který přenese povely PC na rozhraní SPI. Jako nejvhodnější komunikační rozhraní mezi PC a mikrořadičem se jeví sériová linka RS-232, kterou běžně podporují osobní počítače používané v prostředí vývoje mikroprocesorových zařízení a kterou podporuje i většina mikrořadičů prostřednictvím periferie U(S)ART doplněné o vnější převodník napěťových úrovní. Převodníku mezi komunikačními rozhraními s vlastním programovatelným mikrořadičem je možné přiřadit další funkce, které jsou pro to vhodné.

4.1.3. Mikrořadič

Mikrořadič musí být vybaven komunikačními rozhraními UART a SPI, jinak na něj nejsou kladeny vysoké nároky. Proto se jeví vhodné použít mikrořadič platformy AVR od firmy Atmel, a to z několika důvodů: Tyto mikrořadiče mají jednoduchou obsluhu potřebných periferií, takže jejich programování neubíralo čas, který se tak dal využít pro vývoj dalších součástí testovacího systému. K dispozici byl vývojový kit Atmel STK500, který umožňuje vytvořit prototyp bez nutnosti navrhovat a nechat vyrábět vlastní DPS. Zároveň jde o řadu mikrořadičů, s níž již autor této práce v minulosti pracoval a dobře ji zná.

Subsystém sestávající z mikrořadiče, jeho pomocných obvodů (programovací obvody, převodník úrovní na RS-232 apod.) a programu mikrořadiče dostal pracovní název *Integration Tests Electronics Driver* – ITED (česky „ovladač elektroniky pro integrační testy“).

4.1.4. Výsledná architektura testovacího systému

Hotová architektura hardwaru testovacího systému vyplývající z výše uvedené analýzy je uvedena na obrázku 3.

4.2. Softwarové testovací nástroje

Vzhledem k tomu, že jak hardwarová architektura testovacího systému, tak i použitá komunikační rozhraní jsou projektově specifická, není možné vybírat z mnoha veřejně dostupných nástrojů (ať už komerčních nebo volně dostupných), které by byly pro daný účel vhodné. Nástroje pro tento typ testování si firmy obvykle vyvíjejí samy pro interní používání.

Proto se jako nejschůdnější řešení ukazuje následovat tuto cestu a naprogramovat testovací nástroj svépomocí. Toto řešení bylo už v projektu použito například pro testování komunikačních protokolů, kdy vznikla ve spolupráci FEL ČVUT, Gerstner Laboratory a ProTyS, a.s., aplikace SAVS (*Semi-Autonomous Verification System*). Při přípravě testovacího nástroje je nutné zohlednit požadavky normy, tedy především nástroj důkladně otestovat (*identifikace potencionálních poruch a opatření k vyvarování se nebo zvládnutí těchto poruch* [3]) a opatřit manuálem.

Aplikační program pro PC vzniklý v rámci této práce nese pracovní název *Integration Tests Executor* – ITEx (česky „vykonavatel integračních testů“).

4.3. Operační systém pro běh testovacího nástroje

Jak už bylo uvedeno výše, nejvíce omezující podmínkou pro výběr platformy testovacího nástroje je kompilátor. Ten je dostupný pouze ve verzi pro operační systém MS Windows. I další nástroje, s nimiž může testovací aplikace v budoucnu pravděpodobně spolupracovat, jsou

k dispozici především pro tento systém. Proto je vývoj testovacího nástroje zaměřen především na operační systém MS Windows, a to ve verzi Windows XP, která je ve vývojovém týmu dosud nejčastěji používanou.

Volba operačního systému se projeví jednak v požadavcích na verzi spolupracujících nástrojů, ve volbě překladače pro testovací aplikaci samotnou a volbě platformně specifických knihoven, v našem případě především při obsluze sériového portu RS-232.

4.4. Programovací jazyk pro tvorbu testovacího nástroje

Vzhledem k odlišnostem mezi hardwarem mikrořadiče a PC je vhodné vybrat programovací jazyk pro tyto součásti odděleně.

4.4.1. Aplikace pro řízení testu

Testovací aplikace může být vytvořena v mnoha programovacích jazycích, mezi nimiž je preferován jazyk C++. V tomto jazyku je z převážné většiny vytvořen zdrojový kód vlastního programu projektu, takže ho musí každý vývojář v projektu bezpečně ovládat. Zároveň to znamená, že je možné převzít z ostatních součástí projektu již naprogramované funkční celky, které by měla testovací aplikace obsahovat. V neposlední řadě je to jazyk, s nímž má autor této práce největší zkušenosti.

4.4.2. Program pomocného mikrořadiče

Pro program pomocného mikrořadiče je výběr programovacích jazyků mnohem omezenější. Pro platformu AVR připadá pro jednoduché použití v úvahu prakticky pouze Assembler a C. Z nich je upřednostněn jazyk C, který poskytuje lepší způsob zajištění správné činnosti programu (např. typová kontrola parametrů a návratových hodnot funkcí, seskupování příkazů do bloků – funkcí, cyklů apod.). Navíc je tento jazyk alespoň částečně kompatibilní s C++, takže je možné některé části programu sdílet mezi testovací aplikací a programem mikrořadiče, nebo je alespoň postavit na společném základu.

Program mikrořadiče má být spouštěn na osmibitové variantě platformy AVR. Z toho vyplývá, že se bude preferovat používaní osmibitových proměnných. Je nutné se vyvarovat proměnných s plovoucí desetinnou čárkou, se kterými tyto procesory neumí pracovat [6]. Rovněž není v aritmeticko-logické jednotce k dispozici dělička, proto by se ve zdrojovém kódu nemělo vyskytovat dělení s výjimkou případů vyhodnotitelných při kompilaci, která se provádí na PC.

4.5. Uživatelské rozhraní testovací aplikace

Aplikace provozovaná na osobním počítači může být v zásadě dvou druhů: buď konzolová aplikace, jejíž činnost se nejčastěji určuje pomocí parametrů při jejím voláním a má k dispozici standardní vstup a výstup; nebo aplikace s grafickým uživatelským rozhraním, s níž se zpravidla interaguje v průběhu činnosti prostřednictvím klávesnice, myši a monitoru.

Pro testovací aplikaci se očekává opakované automatické spouštění bez nutné přítomnosti obsluhy, proto je vhodnější řešit ji jako konzolovou, aby mohla být plnohodnotně volána systémovým plánovačem a začleněna do skriptů (resp. v prostředí Windows dávkových souborů), které budou mít na starosti kompletní provedení testu, sestávající z aktualizace zdrojových kódů, jejich kompilace, spuštění testovací aplikace, zpracování a zveřejnění výstupu. Tento postup, kdy je výkonná, jednoúčelová testovací aplikace obalena podpůrnými skripty, se v projektu již používá například v případě pravidelné automatické kontroly dodržování kódovacích standardů.

4.6. Vývojové prostředí a knihovny pro tvorbu testovacího nástroje

Kvůli odlišným možnostem hardwaru PC a mikrořadiče a zvolených programovacích jazyků je vhodné tuto část analýzy provádět pro obě platformy samostatně.

4.6.1. Aplikace pro řízení testu

K již zvolenému programovacímu jazyku C++ je možné si zvolit mezi mnoha vývojovými prostředími, které vývoj v něm umožňují. Jsou to například C++ Builder firmy Embarcadero (dříve Borland), Eclipse CDT, MS Visual Studio, Netbeans, Xcode od firmy Apple a mnohé další [7]. Pro vývojové prostředí platí požadavky normy na použité nástroje, proto by se přednostně mělo přihlédnout k nástrojům, které se již v projektu používají a musely absolvovat hodnocení podle normy. To splňují prostředí MS Visual Studio C++ 2003, 2005, 2008, 2010 a Qt. (Pro jiné programovací jazyky je to dále Eclipse (Java) a MS Visual Studio C# 2010.)

Pokud jde o potřebné knihovny, jedná se, mimo standardních knihoven C++ a knihoven API operačního systému, především o knihovnu pro zpracování souborů XML (viz kapitolu 4.9). Z uvedených vývojových prostředí poskytuje Qt SDK (*Software Development Kit*) knihovnu pro práci s XML, kterou už autor této práce navíc použil ve své bakalářské práci. Integrované vývojové prostředí (IDE) Qt Creator poskytuje kromě podpory Qt SDK také kvalitní podporu psaní zdrojového kódu (automatické odsazování, doplňování identifikátorů apod.), podporuje různé systémy správy zdrojového kódu atd. Proto je použito právě IDE Qt Creator.

4.6.2. Program pomocného mikrořadiče

Pro mikrořadič je opět možnost volby omezenější, jako nejjednodušší varianta se nabízí použití vývojového prostředí AVR Studio dodávaného výrobcem čipu – firmou Atmel, a to ve verzi 4, která plně podporuje použitý vývojový kit STK500.[6] Výrobce označuje tuto verzi jako vyzrálou (*mature*), protože je již dostatečnou dobu běžně používána a měla by proto být prosta významnějších chyb. Případným chybám, které by se zde mohly objevit, se předejde důkladným testováním vytvořeného softwaru.

K prostředí AVR Studio nabízí výrobce i kompilátor jazyka C nazvaný `avrgcc`, který se do AVR Studia snadno integruje, čímž vznikne poměrně komfortní IDE, ačkoli zdaleka nedosahuje kvalit moderních IDE, jako je Qt Creator nebo Eclipse.

Knihovny nejsou pro práci s mikrořadičem AVR prakticky potřeba, protože obsluha periferií je jednoduchá a kvalitně zdokumentovaná. Proto stačí použít a přizpůsobit programové moduly pro obsluhu periferií, které vznikly při předchozí práci autora tohoto textu s mikrořadiči AVR.

4.7. Systém pro správu zdrojového kódu

V projektu je pro správu zdrojového kódu neboli správu verzí použit systém Subversion, který je řešen jako centralizovaný. To je vhodné řešení pro firemní prostředí, kde má každý přístup k serveru po celou svou pracovní dobu. Nevyhovuje ovšem pro tvorbu diplomové práce, kde je při odevzdání vhodné dát k dispozici historii verzí v offline formě, a pro práci z domova, tedy mimo dostupnost firemního serverového repositáře.

Z těchto důvodů a také kvůli pokročilým funkcím byla dána přednost systému správy verzí Git. Ten se již několik let úspěšně používá pro správu velkých projektů včetně operačního systému Linux a obecně je řešen tak, aby bylo obtížné v něm dosáhnout chyby jako například smazání části historie. Git jakožto decentralizovaný systém dále umožňuje mít několik kopií repositáře a tak zároveň předejít ztrátě dat při poruše nebo ztrátě záznamového média. K dispozici je i manuál popisující všechny možnosti nástroje a spousta příkladů použití na internetu. Proto Git splňuje požadavky bezpečnostní normy.

4.8. Nástroj pro dokumentaci zdrojového kódu

Bezpečnostní norma vyžaduje, aby měl nástroj specifikaci nebo manuál. Pro vnitřní specifikaci součástí programu, tedy především rozhraní funkcí a tříd, slouží dokumentace

6 V novější verzi je již podpora tohoto kitu, připojovaného pomocí portu RS-232, omezena a upřednostňuje se připojení přes USB.

zdrojového kódu. K dispozici jsou nástroje pro automatické generování této dokumentace. Autor této práce má zkušenosti s programem Doxygen, který byl zvolen jako dokumentační nástroj i pro projekt samotný, a proto je vhodné použít jej i pro související testovací nástroje, neboť jak komentáře ve zdrojovém kódu, tak vygenerovaná dokumentace budou čitelnější i pro ostatní členy týmu, zběhlé v tomto nástroji a jeho syntaxi.

4.9. Formát zaznamenávání předpisů a výsledků testů

U předpisů a výsledků testů je požadováno, aby byly strojově čitelné pro případnou pozdější analýzu. Zároveň norma mluví o tom, aby se zabránilo zanesení chyb tím, že by byla data ručně převáděna z jednoho formátu do jiného. Také je třeba mít na paměti, že ačkoli norma předpokládá strojové čtení a analýzu těchto dat, nejsou zatím v projektu dostupné příslušné nástroje. Proto by měla být současně zachována i čitelnost pro uživatele. Je vhodné použít takový formát, pro jehož zobrazení existují dostupné nástroje nebo je snadné je vytvořit. Výhodná by byla i příbuznost s formáty používanými v jiných fázích testování, takže by mohly být v budoucnu spravovány společnou třídou nástrojů.

Tyto požadavky naplňuje formát Extensible Markup Language (XML), který poskytuje způsob zápisu stromově strukturovaných dat tak, že jsou jednoznačně interpretovatelná počítačem a zároveň jsou tato data čitelná pro osobu, která se s tímto formátem seznámila. Pro XML existuje velká množina různých kategorií nástrojů umožňujících podporu zápisu a orientace v kódu, kontrolu správnosti, transformace do jiných formátů, frameworky pro práci s kódem XML ve vlastním softwaru atd. Formát XML byl již v projektu úspěšně používán pro zápis testovacích případů a výsledků testů při testování vyšších vrstev komunikačních protokolů (v již dříve zmíněném programu SAVS). Obecnější část návrhu formátu (ta, která se netýká přímo komunikačních protokolů) může být navíc z tohoto programu do jisté míry převzata i pro účel integračního testování.

4.10. Získávání informací o hardwaru

Hardware, s nímž má aplikace ITEx v průběhu testu pracovat, může mít v praxi víc druhů a verzí. Proto je nutné zajistit, aby měla aplikace přístup k datům o hardwaru, s nímž právě pracuje, a aby byla schopná otestovat, zda tento hardware plní požadované předpoklady. Jedná se o ověření identity testovaných karet, identitu ITEDu a konfiguraci rozpojitelných komunikačních rozhraní, která ITED obsluhuje.

Identitu testovaných karet by bylo možné zjistit dotazem přes rozhraní CAP, podobně jako při vyčítání proměnných běžícího programu na testované kartě. Možnost číst identifikátor karty ovšem není v dosavadním protokolu pro čtení proměnných z karet implementována. Jak vyplývá z kapitoly 3.2.1, nemůže autor této práce jakožto tester do vlastního programu karet zasahovat, proto je nutné předat požadavek na doplnění kolegům, kteří se zabývají implementací.

Identita zařízení ITED může být řešena číslem verze, z nějž vyplyne, jak lze s ITEDem komunikovat, a dále výpisem karet, jejichž rozhraní dokáže ITED rozpojovat. ITED ovšem není schopný komunikace přímo s kartami, a tedy není schopen reálně zajistit, že připojené karty jsou skutečně ty zamýšlené. I přesto může být v programu ITEDu zahrnuta identifikace karet, správná alespoň co do počtu. To, že tato identifikace odpovídá skutečnosti, musí zajistit osoba, která ITED připravuje k použití – zapojuje ho a nahrává do něj příslušný software.

Konfigurace rozhraní, která je možno rozpojovat, je určena hardwarem (plošnými spoji a relé) rozpojovací desky. Ta ovšem neobsahuje žádný způsob, jak dát tuto konfiguraci najevo ITEDu, nebo dokonce aplikaci ITEx. Proto je nutné, aby i tuto úlohu přebral ITED jakožto hardware disponující potřebnými schopnostmi, který je zároveň ve struktuře systému nejbližším sousedem rozpojovací desky. Druhy rozpojovaných rozhraní a jejich pořadí při zápisu nastavení do posuvného registru rozpojovací desky jsou tedy zapsány v ITEDu. Ten pak tuto konfiguraci poskytuje aplikaci ITEx.

5. Implementace

Po provedení analýzy můžeme přistoupit k implementaci testovacího systému. Ta se bude skládat z několika částí, které jsou řešeny samostatně, především z důvodu odlišného programovacího nebo značkovacího jazyka.

Implementace je místo, kde ve V-modelu (viz kapitolu 3.2.2) dochází ke zlomu mezi přístupem odshora dolů a odzdola nahoru. Autorovi této práce se osvědčilo před zahájením vlastní implementace – tedy psaní výkonného kódu do souboru `.c` nebo `.cpp` – vytvořit metodou shora dolů hlavičkový soubor dané jednotky, což můžeme chápat též jako poslední fázi analýzy, v níž se nadefinují rozhraní mezi jednotlivými jednotkami, třídami a funkcemi. Poté je již možné přistoupit k vytvoření výkonného kódu metodou zdola nahoru.

5.1. Formát předpisů testu

Předpisy testu se zapisují ve formátu XML, jak vyplývá z kapitoly 4.9. Je nutné do nich zapsat požadavky testu, které budou zkontrolovány při spuštění, a vlastní testovací scénář[7]. Ten je rozčleněn na kroky, a ty se skládají jednak z definice lokálních proměnných a jednak z činností zapsaných pomocí základních programovacích struktur. Syntaxe těchto struktur bude v co největší míře převzata z formátu pro předpisy testů programu SAVS, aby nebyl pracovník, který by měl pracovat s oběma programy a jejich předpisy testů, zbytečně maten odlišnostmi obou formátů.

Formát předpisů testu bude specifikován XML schématem v souboru XSD (*XML Schema Definition*). Tento formát především umožňuje přesnější určení pravidel pro obsah dokumentu než starší formát DTD (*Document Type Definition*). V aplikaci Oxygen XML Editor[8], která se v projektu používá, je možné generovat XSD z existujícího souboru XML a při tvorbě dalších podobných souborů využít XSD jednak k automatickému doplňování povinných uzlů a atributů a rovněž vytvořený dokument proti souboru XSD validovat.

5.2. Protokol ITEDComm

ITEDComm je komunikační protokol pro spojení mezi aplikací ITEx a zařízením ITED. Fyzická vrstva je řešena linkou RS-232. Parametry linky jsou stanoveny na 8 datových bitů, 1 stop-bit, sudou paritu a rychlost 115,2 kbit/s.

Linkovou a aplikační vrstvu už bylo nutné navrhnout vlastní, protože jsou specifické pro tento druh komunikace. Komunikace zpravidla probíhá tak, že PC vyšle příkaz, ITED ho zpracuje a pošle odpověď. Ke každému příkazu tedy existuje příslušná odpověď, takže tvoří přirozené páry. Komunikace není potvrzovaná jinou technikou než přenosem smysluplné odpovědi. U některých příkazů je ovšem tato odpověď tvořena prakticky pouze zopakováním příkazu, takže by ji mohlo být možné považovat za potvrzení. Stále je ovšem třeba mít na paměti, že nejde o potvrzení komunikace, ale potvrzení provedení příkazu.

7 V tomto dokumentu jde o pojem zpravidla zaměnitelný s předpisem testu. Pokud je třeba mezi těmito pojmy rozlišovat, pak chápejme testovací scénář jako ležící spíše v ideové rovině (co se má provést) a předpis testu jako jeho zápis ve formalizované podobě.

8 Viz http://www.oxygenxml.com/.

Obecný význam bajtu:		STX	CMD	ID	IDi	LEN										BCC	ETX
Příkaz	Směr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
GET VERSION	D	STX	1	BCC	ETX												
VERSION	U	STX	2	MAJ	MIN	BCC	ETX										
GET CARDS	D	STX	3	BCC	ETX												
CARDS	U	STX	4	0	CNT	1ST	LST	BCC	ETX								
GET CARD INFO	D	STX	5	ID	BCC	ETX											
CARD INFO	U	STX	6	ID	0	LEN	N0	N1	N2	N3	N4	N5	N6	N7	N8	BCC	ETX
GET INTERFACES	D	STX	7	ID	BCC	ETX											
INTERFACES	U	STX	8	ID	CNT	1ST	LST	BCC	ETX								
GET IFACE INFO	D	STX	9	ID	IDi	BCC	ETX										
IFACE INFO	U	STX	A	ID	IDi	LEN	N0	N1	N2	N3	N4	N5	N6	N7	N8	BCC	ETX
SET STATE	D	STX	B	ID	IDi	STT	BCC	ETX									
STATE ACK	U	STX	C	ID	IDi	STT	BCC	ETX									
SET DEFAULT	D	STX	D	ID	0	STT	BCC	ETX									
DEFAULT ACK	U	STX	E	ID	0	STT	BCC	ETX									

Legenda:

D	downstream – z PC do mikrořadiče
U	upstream – z mikrořadiče do PC
STX	start Tx – začátek vysílání (0x5A)
ETX	end Tx – konec vysílání (0xA5)
CMD	příkaz – jeho číslo v šestnáctkové soustavě
BCC	block check character – kontrola správnosti: XOR bajtů č. 1 až (BCC–1)
ID	identifikátor karty
IDi	identifikátor rozhraní (interface)
MAJ	major číslo verze (označení verze: major.minor)
MIN	minor číslo verze (označení verze: major.minor)
CNT	count – počet
1ST	první položka
LST	last – poslední položka
LEN	length – délka dat
Nx	datový bajt č. x
STT	state – stav rozhraní
0	nenese význam – přijímač obsah tohoto bajtu ignoruje

Tabulka 1: Protokol ITEDComm

Komunikace je tvořena rámci o délce 4–16 bajtů. Rámce vždy obsahují znak začátku rámce, pole „příkaz“ – identifikátor druhu rámce, nepovinnou datovou část (podle druhu rámce), kontrolní znak a znak konce rámce. Další obsah rámců se liší podle jednotlivých příkazů s tím, že mezi určitými skupinami příkazů lze nalézt další společné části. Kompletní přehled možných rámců a jejich obsahu je uveden v tabulce 1.

Kontrola správnosti rámců je prováděna kontrolním výpočtem BCC (*Block Check Character*), který je zvolen tak, aby ho bylo snadné provést i na osmibitovém mikrořadiči bez pokročilé aritmeticko-logické jednotky. Provádí se booleovským exkluzivním součtem na bajtech od příkazu až po bajt předcházející poli BCC. Tato kontrola je na fyzické vrstvě doplněna kontrolou parity jednotlivých přenášených bajtů.

Jednotka ITEDComm, tvořící součást programů ITED a ITEx, je založena na stejném implementačním základě, který tvoří následující funkce: `FinishFrameAndSend()` a `ReceiveFrame()`, jež jsou v implementaci v jazyce C (použité v programu pro zařízení ITED) součástí veřejného rozhraní jednotky[9], a implementační (rozuměj neveřejné) funkce `SendFrame()` a `FinishReceivingFrame()`. Uvedeným názvům je v případě programu ITEDu předřazena předpona `IC_`, v případě aplikace ITEx jsou automaticky součástí jmenného prostoru třídy, jejímiž jsou metodami.

5.3. Společná pravidla pro programování

Před popisem implementace programu pro ITED v jazyce C a aplikace ITEx v jazyce C++ si představíme společná pravidla, která byla při implementaci dodržována.

5.3.1. Definice datových typů

Po vzoru modelového projektu se zavádějí jednoduché číselné datové typy, které svým názvem stručně a jasně charakterizují přítomnost znaménkového bitu a délku proměnné v bitech, což jsou kritéria, která nemusí být pro vestavěné datové typy v C a C++ úplně jednoznačné. Zavedené datové typy se značí UI_8, UI_16 a podobně (UI v obou případech znamená *unsigned integer*, číslo počet bitů). Čísla se znaménkem o pevné bitové délce nejsou v programech této diplomové práce použita. Dále jsou k dispozici datové typy o nativní délce slova na daném stroji[10] nazvané UI_P a SI_P (*(un)signed integer – platform*).

5.3.2. Kódovací standard

Pro programy v této diplomové práci bylo stanoveno několik základních pravidel:

Identifikátory se odvozují z anglických slov, používá se tzv. *camel-case*, tedy počáteční písmeno každého slova je velké a slova se neoddělují podtržítky. To neplatí pro identifikátory určené pro preprocesor, které se zapisují pouze velkými písmeny, a proto zde slova musí být oddělena podtržítky.

Lokálně dostupné identifikátory začínají malým písmenem, globálně dostupné začínají velkým písmenem. Globálně dostupnými identifikátory jsou myšleny identifikátory dostupné mimo danou funkci, třídu nebo jednotku. Jsou to tedy i veřejné metody tříd. Naopak soukromé metody a atributy mají začínat malým písmenem.

9 To znamená, že v programu ITEDu v jazyce C jsou uvedeny v hlavičkovém souboru, aby byly přístupné uživateli jednotky. V programu C++ aplikace ITEx jsou všechny zde uvedené funkce soukromými metodami třídy ITEDComm. Rozhraní třídy ITEDComm v aplikaci ITEx tvoří metody `Get...` a `Set...`, např `GetCards()`.

10 Pro osmibitový mikrořadič jde o 8 bitů, pro 32bitový osobní počítač 32 bitů.

Identifikátory parametrů funkcí začínají podtržítkem. (Toto pravidlo bylo zavedeno v průběhu vypracovávání a jednotky implementované jako první ho nedodržují.)

Kód se zalamuje na maximální délku 80 znaků na řádek. Toto pravidlo je sice v programech této diplomové práce až na výjimky dodrženo, přesto při zpětném hodnocení jeho zavedení zhoršilo čitelnost kódu. Omezení šířky kódu je sice rozumný požadavek, ale vhodnější by byla větší šířka.

Pro odsazení kódu se používají čtyři mezery.

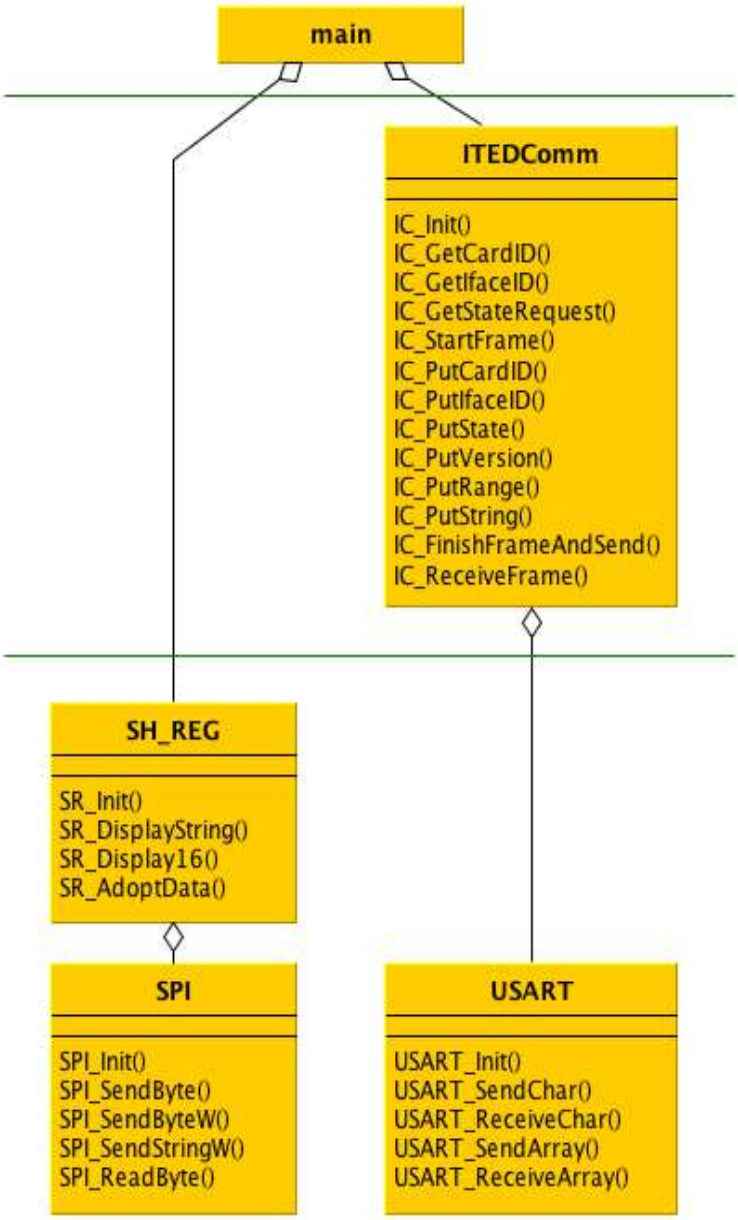
Funkce v aplikaci ITEx, které pracují s vnějším prostředím (komunikace, soubor XML), vrací zpravidla datový typ `bool`, který říká, zda operace proběhla úspěšně. Pokud mají mít tyto funkce výstup, použije se výstupní parametr – ukazatel nebo reference, do nichž se uloží výstupní hodnota. Výjimku tvoří funkce, jejichž výstup by mělo být možné přímo použít ve výrazu. V takovém případě platí: Pokud má být výstupem číslo, bude chybový výstup binární doplněk k číslu 0 (hodnota interpretovaná jako maximální dosažitelné číslo v případě bezznaménkového typu nebo −1 u znaménkového typu). Pokud má být výstupem ukazatel, je chybová hodnota `NULL`.

V komentářích k programovému kódu jsou z důvodu automatického generování české dokumentace zdrojového kódu povoleny české znaky s diakritikou v takovém kódování, aby s nimi byly použité nástroje schopné bez obtíží pracovat. (Pro aplikaci ITEx splňuje tyto podmínky kódování UTF-8, pro ITED muselo být použito kódování *Windows Latin 2* neboli *CP1250*, především proto, že AVR Studio volbu kódování neumožňuje a používá systémové.)

5.4. Program pomocného mikrořadiče (ITED)

Program mikrořadiče má za úkol především převést povely k rozpojení nebo spojení rozhraní přijaté po rozhraní USART na SPI a kromě toho odpovídat na dotazy ohledně konfigurace, které mohou z PC přicházet.

Program obsahuje výkonnou hlavní jednotku (*main*), jež zajišťuje reakci na přijímané příkazy, a moduly pro obsluhu periferií SPI a USART. Na rozhraní USART (lince RS-232) je mezi PC a zařízením ITED zaveden komunikační protokol ITEDComm. Sestavení a dekódování jeho rámců bude vyčleněno do stejnojmenné jednotky. Nad rámec rozhraní SPI je potom nutné zajistit vyslání signálu RCLK pro převzetí dat záchytným registrem. Proto je obsluha SPI zapouzdřena do jednotky SH_REG, která zajišťuje celou obsluhu posuvných registrů (*shift register*).



Obrázek 4: Architektura programu mikrořadiče

Architektura programu mikrořadiče je uvedena na obrázku 4. Z něj je zřejmé, že identifikátory spadající do jednotlivých jednotek mají předponu `SPI_`, `USART_`, `IC_` (pro ITEDComm) a podobně, aby bylo zajištěno jejich rozlišení v programu v jazyce C, který neobsahuje jmenné prostory.

Architekturu programu lze rozdělit do tří vrstev, naznačených na obrázku vodorovnými čárami. Nejnižší je přímo závislá na použitém hardwaru: Pokud se bude program přenášet na jiný typ mikrořadiče, bude tuto část nutné upravit. Implementace této vrstvy byla odvozena ze starších

kódů autora této práce pro mikrořadiče Atmel AVR, což se projevuje jednak tím, že dokumentace zdrojového kódu je v angličtině (zatímco zbytek kódu pro tuto diplomovou práci je dokumentován česky), a jednak tím, že se nepoužívají projektové, ale výchozí jednoduché datové typy. Prostřední vrstva má na starost obsluhu komunikačního rozhraní ITEDComm. Tato část je již nezávislá na hardwaru – její implementace má naopak dokonce společný základ s obsluhou rozhraní ITEDComm v aplikaci ITEx. Nejvyšší vrstva potom obsahuje kód zajišťující požadované chování zařízení.

5.4.1. Obsluha rozhraní SPI

Obsluha komunikačního rozhraní SPI je tvořena funkcemi pro zápis a čtení na rozhraní, přičemž jediná existující funkce pro čtení (`SPI_ReadByte()`) se v programu ITED nepoužívá. Je tu přítomna také funkce `SPI_Init()`, která zajistí inicializaci příslušné periferie mikrořadiče (v objektově orientovaném programování by tato činnost byla součástí konstruktoru). Funkce pro zápis jsou k dispozici v blokující a neblokující variantě, blokující varianta je vyznačena písmenem `W` (podle anglického slova *wait*) na konci názvu funkce. V programu dojde uplatnění pouze funkce `SPI_SendStringW()`, která dále volá funkci `SPI_SendByteW()`.

5.4.2. Obsluha posuvného registru

Jednotka SH_REG (*shift register*, předpona pro členy je `SR_`) pro úplnou obsluhu posuvného registru ukrývá komunikaci pomocí SPI, doplňuje k ní obsluhu signálu RCLK pro převzetí dat z posuvného do záchytného registru (funkce `SR_AdoptData()`) a nalezneme zde i funkci pro odeslání 16bitového čísla pomocí 8bitového rozhraní SPI `SR_Display16()`. Taková čísla je totiž nutné odesílat na rozpojovací desku, vybavenou dvěma kaskádně řazenými posuvnými registry 74HC595. Funkce pro inicializaci rozhraní `SR_Init()` zajistí inicializaci periferie SPI a nastavení vstupně-výstupního pinu integrovaného obvodu pro vysílání signálu RCLK.

5.4.3. Obsluha rozhraní USART

Obsluha periferie USART, která se využívá pro komunikaci po lince RS-232, je tvořena funkcemi pro odesílání a přijímání jednoho bajtu (`USART_SendChar()` a `USART_ReceiveChar()`) i více bajtů (`USART_SendArray()` a `USART_ReceiveArray()`), které odešlou nebo přijmou pole bajtů o předem známé délce. Dále zde samozřejmě nalezneme inicializační funkci `USART_Init()`, která periferii mikrořadiče nastaví na parametry komunikace určené protokolem ITEDComm.

Funkce pro odesílání i příjem pomocí rozhraní USART si v našem případě vystačí bez použití přerušení. Je to dáno především tím, že komunikace probíhá formou požadavků a odpovědí. Po inicializaci mikrořadiče nebo když zařízení odešle odpověď, dostává se do stavu, kdy čeká na přijetí dalšího příkazu a neexistuje žádná činnost, která by mezitím měla probíhat. Po přijetí příkazu zahajuje ITED činnost, po jejímž dokončení odešle odpověď a vrací se do výchozího stavu. Do té doby není nutné reagovat na jakékoli požadavky, protože ITEx čeká, až mikrořadič odpoví. Tato implementace je nejjednodušší, ovšem pro naše potřeby dostatečná.

5.4.4. Komunikace protokolem ITEDComm

Jednotka ITEDComm se skládá z funkcí pro čtení jednotlivých datových polí z přijatého datového rámce a obdobně i pro zápis polí do odesílaného datového rámce. Dále zde nalezneme skupinu funkcí, které řeší obecné postupy spojené se započetím a ukončením přijímání i odesílání rámce.

Funkce `IC_StartFrame()` zajistí vyplnění začátku rámce, přičemž druh rámce (pole označené CMD) je určen parametrem funkce. Poté se provede vyplnění datových polí funkcemi `IC_Put...()`, kde za ... dosadíme, jaké datové pole se má vyplnit. Nakonec zajistíme dokončení a odeslání rámce zavoláním funkce `IC_FinishAndSendFrame()`. Tato funkce zajistí výpočet kontrolního znaku, doplnění znaku konce rámce a poté zavolá funkci `IC_SendFrame()`, která zajistí volání platformně specifické implementace odeslání rámce.

Pro přijetí celého rámce stačí zavolat funkci `IC_ReceiveFrame()`. Tato funkce, respektive jí volaná obsluha rozhraní USART, čeká nejprve po jednom na příchozí bajty. Přijaté bajty se ukládají do bufferu přijímaného rámce. V okamžiku, kdy je zřejmé, jak dlouhý bude konkrétní rámec, předá se řízení funkci `IC_FinishReceivingFrame()`, která zajistí přijetí známého počtu zbývajících znaků a ověření platnosti přijatého rámce. Výsledek kontroly předá zpět funkci `IC_ReceiveFrame()` a ta ho předá dál jako svoji návratovou hodnotu.

Jak už bylo zmíněno dříve, implementace jednotky ITEDComm má stejný základ jak pro aplikaci ITEx, tak pro zařízení ITED. Obě implementace se od sebe ovšem v několika ohledech liší:

Použitý programovací jazyk C neobsahuje třídy ani jmenné prostory, takže funkce dané jednotky se zařadí mezi globální a vymezení jednotky je možné pouze názvem. V tomto případě je to tedy předpona `IC_`.

Další rozdíl spočívá ve straně komunikace, kterou má program zajišťovat. ITED se chová jako podřízená strana, která má vykonávat příkazy přijaté po komunikační lince. ITEDComm se tak výraznou měrou podílí na rozhodování o činnosti programu.

Poslední odlišnost lze vidět v tom, že jednotka ITEDComm pro mikrořadič dovoluje nadřízené části programu[11] vstupovat hlouběji do implementace, tedy především přímo k odesílanému nebo přijímanému komunikačnímu rámci. Ten je vyčítán (v případě přijímaného rámce) nebo vyplňován (v případě rámce odesílaného) po jednotlivých polích nadřízeným programem. Odpovědnost za vyplnění všech datových polí tak nese nadřazený program, zatímco v aplikaci ITEx je vyplnění celého rámce zaručeno tím, že je obsah všech polí zpravidla vyžadován veřejnou metodou třídy ITEDComm, která obsahy polí rámce přebírá jako své parametry. Tento způsob implementace byl zvolen z několika důvodů:

U programu v jazyce C nelze v průběhu jedné funkce definovat nové lokální proměnné a ukončovat platnost starých, a proto byl upřednostněn postup, kdy se jednotlivá pole dat do odesílaného rámce předávají postupně voláním jednotlivých funkcí jednotky ITEDComm, místo aby musely být nejprve uloženy do pomocných proměnných a poté se předaly najednou jako parametry jedné funkce.

Ještě výraznější je tento rozdíl v případě čtení přijatého rámce. V tomto případě by funkce musely buď vracet víc proměnných najednou (bylo by tedy nutné vytvořit z nich strukturu a do ní data kopírovat), nebo by se musely uplatnit výstupní parametry, kdy se funkci předává ukazatel na paměťové místo, kam má uložit výsledek své operace. Každopádně by bylo nutné obsadit paměť i pro proměnné, které třeba vůbec nejsou využity, pokud by byly spolu s jinými, potřebnými proměnnými součástí struktury nebo parametry téže funkce. Přitom pracujeme s jednoduchým mikrořadičem s omezenými systémovými zdroji, takže je vhodné těmito zdroji příliš neplýtvat.

Použitý přístup je omezen požadavkem na sekvenčnost programu – pokud delší dobu samostatnými funkcemi pracujeme nad globálním polem, jakými jsou i buffery pro přijímaný a odesílaný rámec, nesmí do těchto míst zasahovat žádná jiná část programu. To je v případě ITEDu zaručeno jednovláknovým během programu bez přerušení.

11 V programu pro ITED jde o hlavní část programu (*main*), ovšem obecně by mohl být kód jednotky ITEDComm volán jakoukoli jinou součástí programu.

5.4.5. Hlavní část programu ITED

Hlavní část programu (*main*) zajistí inicializaci periferií a proměnných, které bude ITED vracet jako odpovědi dotazů na konfiguraci. Poté se již vstupuje do nekonečné hlavní smyčky. Ta začíná voláním funkce `IC_ReceiveFrame()`, která čeká na přijetí platného rámce. Pokud přijatý rámec není platný, vracíme se na začátek smyčky; v opačném případě dojde k vyhodnocení obsahu rámce a reakci na něj. Nejprve je tedy nutné načíst druh rámce a podle toho se zachovat. To zajistí volání funkce `IC_GetCommand()`, která vrátí některou hodnotu z výčtu (`enum`) příkazů protokolu ITEDComm. Získaná hodnota se využije pro volbu větve ve struktuře `switch`, jež tvoří stěžejní část hlavního programu. Jednotlivé větve struktury `switch` se potom zpravidla skládají z načtení dat z přijatého rámce, vykonání požadované činnosti, sestavení odpovědi a jejího odeslání.

Hlavní program může signalizovat svou činnost pomocí osmi LED dostupných na kitu STK500. Ty jsou zapojeny v negativní logice (svítí při nulovém napětí na příslušném výstupním pinu mikrořadiče), a proto jsou v programu zpravidla nastavovány číselnou konstantou s operátorem logické negace (~). Pomocí pohledu na stav LED a zároveň do zdrojového kódu hlavního programu lze snadno určit, jakou činnost program naposledy prováděl.

5.5. Aplikační program pro řízení testu (ITEx)

Aplikace ITEx běžící na osobním počítači má přehled o stavu celého testovacího systému a řídí vykonávání testovacího scénáře. Znamená to tedy, že musí umět komunikovat se všemi zúčastněnými stranami (testované karty, ITED), být schopna číst předpis testu ve formátu XML a vlastními schopnostmi nebo s pomocí ostatních komponent zajišťovat provedení v něm uvedených operací. Výsledky činnosti testovacího systému má potom ukládat do protokolu z testu.

Návrh aplikace ITEx je pro snazší správu proveden objektově. To umožňuje v souladu s ideou objektově orientovaného programování[8] popsat skutečný systém sestávající například z objektů testovaných karet, ITEDu apod., zajistí se skrytí implementace před ostatními částmi programu a automatické provedení všech inicializací a deinicializací (pomocí konstruktorů a destruktorů). Současně se tím definují moduly, po nichž je možné aplikaci postupně navrhovat a testovat.

Jak vyplývá z výše uvedeného popisu funkcí, bude aplikace rozčleněna do tří částí. Těmi jsou komunikace, čtení a zpracovávání předpisu testu a tvorba protokolu z testu (logování). Logování přitom nebude v rámci této práce řešeno, protože vyhrazený časový rámec nedovoluje kvalitní

návrh a implementaci tohoto subsystému. Jeho návrh se přitom typově podobá návrhu ostatních částí softwaru a implementace bude pravděpodobně jednodušší[12] než v případě interpreta předpisu testů. Proto bude pozornost zaměřena na zajímavější záležitosti ohledně komunikace mezi zařízeními a interpretace předpisu testu ve formátu XML.

5.5.1. Architektura aplikace ITEx

Architektura celého navrženého aplikačního programu ve formě diagramu tříd (*class diagram*) UML je uvedena na obrázku 5. Toto schéma zobrazuje všechny třídy, které byly v rámci vytváření této diplomové práce implementovány, a jejich nejdůležitější veřejné metody a atributy (neveřejné[13]). Parametry a návratové typy jsou uvedeny pouze tehdy, pokud je to u nich zajímavé. Návratové typy metod a typy proměnných jsou uvedeny v syntaxi odpovídající jazyku Pascal: za dvojtečkou za názvem proměnné nebo funkce. Uvedené názvy nebo typy parametrů jsou brány pouze kvalitativně[14]. Na následujících stranách jsou jednotlivé komponenty a třídy probrány jednotlivě.

5.5.2. Použití kontejneru `std::map` v aplikaci ITEx

V architektuře aplikace ITEx se několikrát setkáme s datovým typem *mapa*. Na obrázku 5 je pro zjednodušení reprezentován hranatými závorkami jako pole. Mapa je datový kontejner ze standardní knihovny C++, který umožňuje ukládat páry (`std::pair`) dvou hodnot libovolných datových typů a podle prvního prvku páru (klíče) vyhledávat druhý prvek. V aplikaci ITEx je mapa použita pro správu a vyhledávání objektů pomocí textového názvu. Týká se to rozhraní jedné karty (kontejner `ifaceRecords` třídy `Card`), celých karet (kontejner `cardRecords` třídy `CardManager`) a testových proměnných (kontejner `varRecords` třídy `VariableManager`).

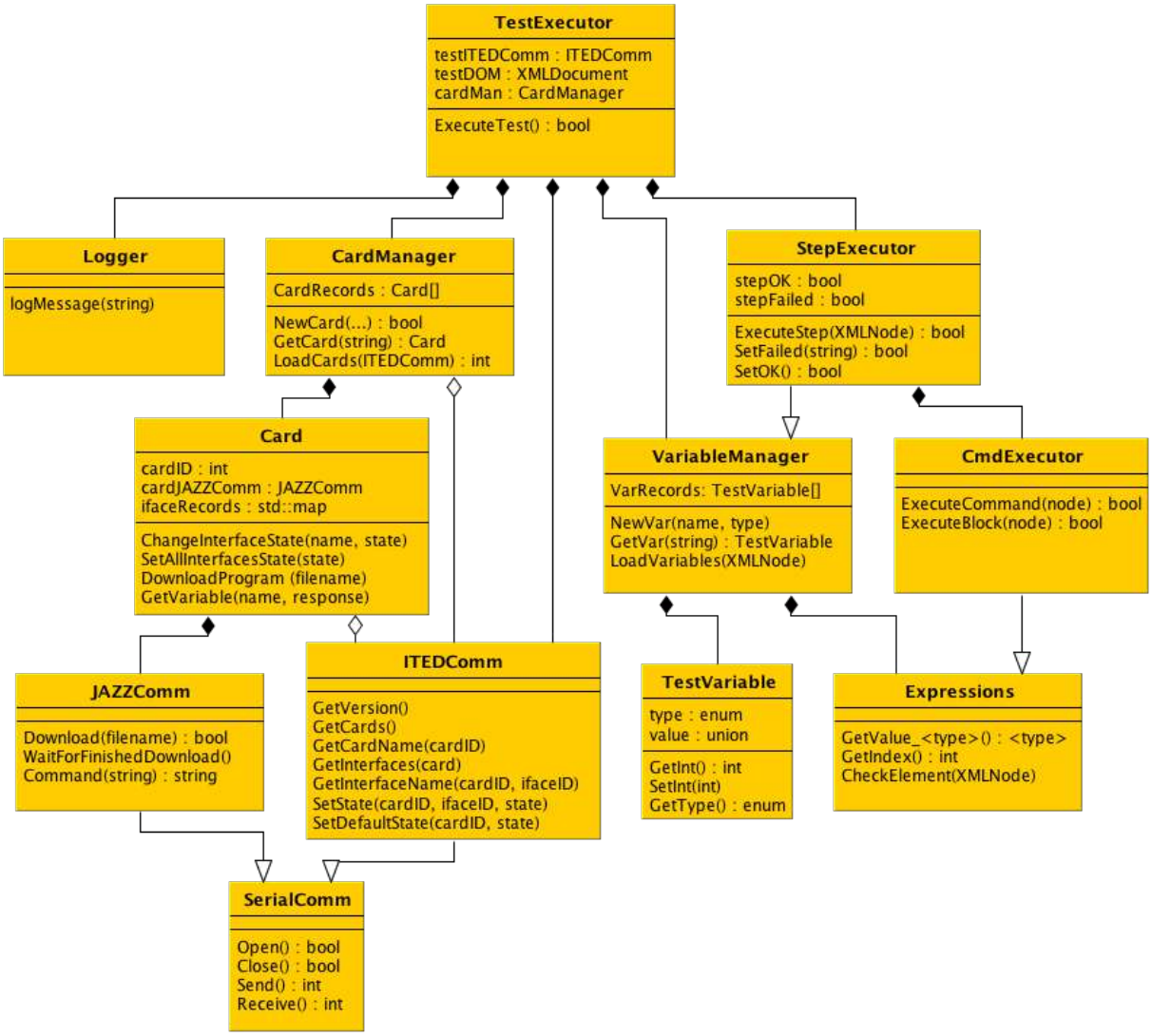
S použitím mapy ze standardní knihovny ovšem souvisí jedna nepříjemnost, a to vytváření, kopírování a rušení dočasných objektů. Objekt, který do mapy vkládáme, je vytvořen jako dočasný lokální objekt v metodě, která ho do mapy ukládá, ten je posléze dvakrát kopírován a nakonec se dočasné objekty zruší. Kopírování objektů navržených pro aplikaci ITEx nezvládl kompilátor ošetřit sám, a bylo proto nutné u tříd, jejichž objekty se ukládají do mapy

12 I pro zamýšlený zápis protokolu ve formátu XML by bylo možné použít pouze standardní funkce typu `sprintf`. Stačí zajistit, aby byly při ukončení části zápisu uzavírány příslušné uzly ukončovacím tagem.

13 Při implementaci byla dodržena zásada, že k atributům třídy by se mělo přistupovat přes její metody. Proto mohou být všechny atributy neveřejné.

14 Rozlišuje se pouze pravdivostní hodnota, číslo, textový řetězec apod., nikoli už počet bitů u čísla, způsob uložení řetězce apod.

(`TestVariable` a `Card`, kvůli atributu třídy `Card` navíc i třída `JAZZComm`) napsat kopírovací konstruktory.



Obrázek 5: Architektura aplikace ITEx ve formě diagramu tříd UML

5.5.3. Komunikační část aplikace

ITEx musí pro svou správnou činnost komunikovat nejméně s jedním zařízením ITED a s jednou nebo více testovanými kartami. Oba tyto druhy komunikace probíhají prostřednictvím sériového portu počítače: U rozhraní ITEDComm již bylo zmíněno, že jako fyzická vrstva je použita linka RS-232. Při komunikaci s kartami přes rozhraní CAP se sice dříve zmíněný převodník připojuje do PC pomocí sběrnice USB, ovšem do operačního systému se nainstaluje jako virtuální sériový port. Proto je možné pro komunikaci s ITEDem i testovanými kartami použít společnou třídu `SerialComm` pro obsluhu sériového portu, z níž je odvozena třída

`ITEDComm`, zajišťující komunikaci se zařízením ITED, a `JAZZComm`, zajišťující komunikaci s testovanými kartami.

U třídy `JAZZComm` je zřejmé, že zprostředkovává komunikaci s jednotlivými kartami. Taktéž se ale dá říct, že i `ITEDComm` zajišťuje (mimo jiné) obsluhu jednotlivých karet, respektive jejich komunikačních rozhraní a napájení. Proto může být třída `JAZZComm` spolu s příslušnými metodami třídy `ITEDComm` zastřešena společným objektem reprezentujícím testovanou kartu. Ten bude instancí třídy `Card`, která sdružuje přístup k jedné kartě přes oba druhy komunikace.

Poslední třídou, kterou lze zařadit ke komunikačním funkcím, je `CardManager`. Jak napovídá název, jedná se o správce objektů karet. Jednotlivým objektům karet tato třída zajišťuje místo v paměti (pomocí dynamického pole objektů zahrnujícího zajištění alokace a dealokace paměti pro zahrnuté objekty) a vyhledávání karet podle jejich názvu, který je používán v předpisu testu. Objektu typu `CardManager` je také svěřeno načítání konfigurace karet z ITEDu při inicializaci testovací aplikace, které má za následek vytvoření objektů karet v běžící aplikaci, a tím tedy naplnění objektu `CardManager`.

5.5.3.1. Třída `SerialComm`

`SerialComm` je třída, která zapouzdřuje volání funkcí operačního systému pro obsluhu sériového portu RS-232. Vzhledem k tomu, že se aplikace vyvíjí především pro operační systém MS Windows (viz kapitolu 4.3), počítá se především s voláním odpovídajících funkcí v tomto systému. To se projeví především na typech a počtu parametrů metod třídy `SerialComm`. Týká se to především konfigurace sériového portu, kdy je zodpovědnost za práci se systémovou strukturou `COMMCONFIG` přenesena o úroveň výš na třídu, která `SerialComm` používá. Na znamení toho jsou také dotyčné metody pojmenovány `GetConfigWin()`, `SetConfigWin()` a `SetTimeoutWin()`. Pokud by bylo nutné doplnit `SerialComm` o podporu dalších operačních systémů, byly by vytvořeny přenositelné metody `GetConfig()` a `SetConfig()` s jednotným rozhraním pro uživatele třídy.

Kromě výše zmíněných jsou k dispozici metody, jejichž účel je zřejmý z názvu: `Open()`, `Close()`, `Send()` a `Receive()`. Metody `Send()` a `Receive()` jsou dostupné pro vstupní pole znaků se znaménkem i bez něj (`signed` i `unsigned char`), protože se v různých částech programu vyžadují obě možnosti.

Otevření portu je kromě volání metody `Open()` možné nechat provést přímo v konstruktoru (nastavením druhého, nepovinného parametru na hodnotu `true`). Destruktor zajistí zavření portu, pokud je v okamžiku rušení objektu otevřený.

5.5.3.2. Třída `ITEDComm`

`ITEDComm` je, jak už bylo několikrát zmíněno, třída postavená na stejném základě jako komunikační jednotka `ITEDComm` v programu ITEDu. Tato implementace ve formě třídy C++ využívá jako fyzickou vrstvu třídu `SerialComm`, od které dědí. Tato dědičnost ovšem není fundamentální a do budoucna se dá uvažovat o přepracování tak, aby byl objekt `SerialComm` pouze atributem třídy `ITEDComm`.

Třída `ITEDComm` zajišťuje veškerou komunikaci protokolem ITEDComm a názvy jejích metod se podobají názvům příkazů protokolu ITEDComm. Nalezneme zde tedy veřejné metody `GetVersion()`, `GetCards()`, `GetCardName()`, `GetInterfaces()`, `GetInterfaceName()`, `SetState()` a `SetDefaultState()`. Tyto metody odstiňují uživatele třídy od celé komunikace, takže po zavolání metody proběhne celá komunikace a výsledkem je přímo vyčtený údaj, který se uloží do výstupních parametrů. Samotné sestavení a odeslání příkazu, přijetí a dekódování odpovědi mají na starost soukromé metody třídy `ITEDComm`, které již známe z implementace komunikace ITEDComm v jazyce C: `sendFrame()`, `receiveFrame()`, `finishFrameAndSend()` a `finishReceivingFrame()`.

Konstruktor třídy `ITEDComm` vyžaduje zadání názvu sériového portu, který ihned otevře a drží otevřený po celou dobu existence objektu. Implementace třídy ITEDComm předpokládá pro každé[15] zařízení ITED jeden objekt této třídy.

5.5.3.3. Třída `JAZZComm`

`JAZZComm` je, stejně jako `ITEDComm`, potomkem třídy `SerialComm`, ale i v tomto případě platí, že tato dědičnost není fundamentální. Práce se sériovým portem se od třídy `ITEDComm` liší v tom, že je nutné přidělovat port dvěma druhům komunikace: Třída `JAZZComm` zajišťuje jednak přímou komunikaci s běžícím programem na testované kartě, kdy se komunikace pomocí rámců obsahujících dotazy a odpovědi do jisté míry podobná protokolu ITEDComm; a jednak je zde režim downloadu programu do karty, kdy musí třída `JAZZComm` zajistit uvolnění (včetně

15 Pokud by jich mělo být v budoucnu více. Současná implementace aplikace ITEx počítá s jedním globálním objektem třídy `ITEDComm`.

uzavření) portu aplikací ITEx a zavolat externí aplikaci, která download zajistí. Nutno ovšem podotknout, že v době dokončení diplomové práce nebyla tato aplikace v konzolové variantě, která by umožnila volání jiným programem, automatické proběhnutí a předání řízení zpět, ještě připravena, a její činnost byla simulována aplikací MockupApp, která po předem stanovenou dobu v řádu jednotek až desítek sekund provádí pouze aktivní čekání a poté skončí s návratovým kódem, který byl určen argumentem zadaným při volání této aplikace.

Download programu je kromě volání externí aplikace zajímavý také tím, že jde o jedinou situaci v programech této diplomové práce, kdy je potřebné provádět dvě operace současně. Zde musí být spuštěna aplikace pro download, která začne testované kartě vysílat požadavek na přijímání programu. Ovšem tento požadavek umí na kartě přijmout pouze zavaděč, který je k dispozici nedlouhou dobu po připojení napájení karty a záhy předá řízení programu, který byl na kartu již dříve nahrán. Proto je nutné nejprve spustit downloadovací aplikaci jako paralelně běžící proces a ponechat si řízení. Poté se musí odeslat ITEDu příkaz k zapnutí napájení pro příslušnou kartu. Nyní je již možné ITEx uspat a počkat na doběhnutí externí aplikace, která dá svým návratovým kódem najevo, zda se download zdařil.

Toto chování je z hlediska třídy `JAZZComm` zajištěno metodou `Download()`, která spustí externí aplikaci a zahájí download, a metodou `WaitForFinishedDownload()`, která zastaví běh programu, dokud není download programu do karty dokončen. Celé řízení popsané asynchronní operace má potom na starost funkce `Card::DownloadProgram()`.

Pro komunikaci s běžícím programem prostřednictvím dotazů a odpovědí se použije veřejná metoda `Command()`, která podobně jako metody `ITEDComm::Get…()` zapouzdřuje volání soukromých metod. Zde jsou to `formFrameAndSend()` a `receiveAndParseFrame()`. V případě třídy `JAZZComm` dochází pouze k jednomu způsobu sestavování a odesílání rámce a stejně tak k jednomu způsobu přijetí a dekódování, proto jsou tyto činnosti seskupeny do jedné funkce pro každý směr komunikace. Autorovi této práce ovšem připadalo vhodné zdůraznit i názvem těchto funkcí, že zajišťují jak komunikaci, tak i přípravné, resp. dokončovací operace.

Z předchozího odstavce je zřejmé, že komunikace po rozhraní CAP nebyla převzata z existující implementace v projektu, ale byla nově implementována. Byl sice učiněn pokus o přenesení implementace z PC aplikace Browser, která se běžně používá pro komunikaci s kartami přes CAP, ale vyskytlo se několik komplikací. Zdrojový kód této aplikace je napsán částečně v C++ a částečně v jazyce Pascal. Navíc poskytuje příliš široké spektrum ne příliš dobře

dokumentovaných možností na to, aby mohl být kód snadno přenesen. Proto byla z aplikace Browser převzata (a to navíc s úpravami) jen malá část zdrojového kódu, který byl v projektu ITEx umístěn do složky `browser`. Jedná se o výpočet CRC, který se používá ke kontrole rámců procházejících přes rozhraní CAP. V současné implementaci je tento výpočet použit pouze pro odesílané rámce, aby je druhá strana přijala; u přijatých rámců se správnost CRC zatím nekontroluje.

5.5.3.4. Třída `Card`

Třída `Card` zapouzdřuje komunikaci s kartami probíhající po rozhraních CAP a ITEDComm tak, aby se karta jevila pro ostatní součásti programu jako jednolitý předmět v souladu s principy objektově orientovaného programování [8].

To se projeví již v konstruktoru. Konstruktor karty potřebuje znát její označení (`cardID`) v ITEDu a spojení na ni (konstantní ukazatel na objekt ITEDComm), rovněž potřebuje znát název sériového portu, přes nějž se má vést komunikace prostřednictvím třídy `JAZZComm`. Na základě těchto informací konstruktor zavolá soukromou metodu `loadInterfaces()`, která načte (pomocí několika komunikačních cyklů protokolu ITEDComm) z ITEDu rozhraní, jež je možné u dané karty obsluhovat, a uloží je do mapy `ifaceRecords`.

Tato rozhraní je následně možné obsluhovat metodou `SetAllInterfacesState()`, která přímo odešle příslušný příkaz do ITEDu, a metodou `ChangeInterfaceState()`, která vyhledá ID rozhraní podle zadaného názvu a to pak použije pro komunikaci s ITEDem. U obou těchto metod je možné zadat, v souladu s protokolem ITEDComm, osmibitové číslo stavu. Reálně ovšem ITED podporuje pouze stav 0 (rozepnuto) a 1 (sepnuto).

Prostřednictvím metody `DownloadProgram()` můžeme do karty nahrát program. Tato metoda již zajistí celý proces downloadu, skládající se ze dvou paralelně prováděných úloh, jež jsou popsány v kapitole 5.5.3.3. Metoda `GetVariable()` zajistí načtení hodnoty zveřejněné proměnné z programu běžícího na kartě. Znamená to sestavit a vyslat odpovídající příkaz po rozhraní CAP a vyčíst z karty vytvořenou odpověď. O velkou část z toho se postará funkce `JAZZComm::Command()`, kterou metoda `GetVariable()` volá.

5.5.3.5. Třída `CardManager`

`CardManager` slouží ke správě objektů testovaných karet. V aplikaci ITEx existuje jeden objekt této třídy, který má na starost všechny karty zúčastněné v testu. Jak již bylo zmíněno v kapitole 5.5.2, provádí se vlastní správa objektů pomocí datového typu *mapa*.

Do této mapy je možné přidat jednotlivou kartu prostřednictvím metody `NewCard()`, která ovšem vyžaduje zadání velkého množství parametrů: názvu, pod nímž má být karta dohledatelná, ID karty v ITEDu, ukazatele na objekt `ITEDComm`, který ovládá rozhraní karty, a název portu, přes nějž lze s kartou komunikovat přes `JAZZComm`.

Upřednostňuje se tedy načtení všech karet z ITEDu pomocí metody `LoadCards()`, která vyžaduje jediný parametr – ukazatel na objekt `ITEDComm`. Port pro `JAZZComm` se potom ke kartám určuje na základě jejich názvu vyčteného z ITEDu podle klíče určeného konfiguračním souborem.

Pro práci s kartou, která je zavedena v objektu `CardManager`, slouží metoda `GetCard()`. Ta na základě zadaného názvu vrátí ukazatel na objekt příslušné karty, pokud je tedy karta zadaného názvu známa.

5.5.4. Proměnné testu

V předpisech testu je možné využívat proměnné testu. Ty mají svůj typ, jméno a hodnotu, stejně jako běžné proměnné v programovacím jazyce C++. Na rozdíl od nich se ovšem parametry (typ a jméno) testových proměnných dozvíme až za běhu programu při čtení předpisu testu. Jejich druh i počet se navíc mohou mezi jednotlivými scénáři lišit. Proto je nutné zavést proměnné testu jako samostatné dynamicky vytvářené objekty. Třída k tomu určená se nazývá `TestVariable` a podobně jako karty jsou její objekty spravovány objektem třídy `VariableManager`.

Objekt `TestVariable` dále může obsahovat buď jednoduchou proměnnou, nebo pole o předem známém počtu prvků, které se alokuje dynamicky. Pro objekty `TestVariable` je dále nutné naprogramovat dynamickou typovou kontrolu (tj. kontrolu typu proměnné prováděnou za běhu programu), kterou jazyk C++ neobsahuje.

Třída `VariableManager`, podobně jako `CardManager`, má na starost i hromadné načtení spravovaných objektů – v tomto případě načtení proměnných z předpisu testu. Pokud má

proměnná při své definici v předpisu testu přímo zadánu hodnotu, musí být zároveň vyhodnocen výraz reprezentující její hodnotu a výsledek uložen do proměnné.

5.5.4.1. Třída `TestVariable`

Objekty třídy `TestVariable` reprezentují jednotlivé proměnné testu. Tyto proměnné musí mít přiřazený typ. Těmi může být v současné době celé číslo se znaménkem (`VT_INTEGER`) nebo pole takových čísel o předem známé délce (`VT_INTEGER_ARRAY`); implementace je ovšem připravena na zavedení dalších datových typů. Typ proměnné je nutné vyplnit už při volání konstruktoru. Pokud má jít o pole, musí se zadat také počet prvků. První prvek pole má index 0 a poslední má index o jedničku nižší, než je počet prvků, jak je zvykem v jazycích založených na C. Typ existující proměnné je možné zjistit metodou `GetType()`.

Pro načtení celočíselné hodnoty proměnné slouží metoda `GetInt()`, jejímž jediným, a navíc nepovinným parametrem je index prvku v poli. Ten je nepovinný, protože u proměnné, která není polem, nemá tento parametr význam. Pokud nebude zadán u pole, vrátí metoda hodnotu prvního prvku (s indexem 0). Implementace metody kontroluje, zda je zadaný index menší než délka pole a nedovolí adresování paměti mimo alokované pole.

Metoda `SetInt()` slouží k nastavení nové hodnoty proměnné. Pokud jde o pole, je opět nutné vyplnit nepovinný druhý parametr – index, jinak bude přepsán první prvek (s indexem 0).

5.5.4.2. Třída `VariableManager`

Ke správě testových proměnných slouží třída `VariableManager`. Protože se podobá účelem, i její metody se velice podobají třídě `CardManager`, kterou jsme již probírali v kapitole 5.5.3.5.

K vytvoření nové proměnné slouží metoda `NewVar()`. Vyžaduje zadání názvu proměnné, jejího typu (z výčtu `TestVariable::VarType`) a v případě pole i počtu prvků.

Ke hromadnému načtení proměnných je určena metoda `LoadVariables()`. Ta načte proměnné z určeného uzlu `<variables>` v předpisu testu. Pokud má proměnná zadanou počáteční hodnotu, je k jejímu vyhodnocení využito schopností třídy `Expressions`, kterou si probereme v kapitole 5.5.5.1.

Metoda `GetVar()` slouží k získání ukazatele na proměnnou (přesněji objekt `TestVariable`) na základě jejího názvu. Tato metoda je jako jediná v celé aplikaci deklarována

jako `virtual`, aby bylo umožněno její rozšíření ve třídě `StepExecutor`, která je potomkem třídy `VariableManager`.

5.5.5. Interpret předpisu testu

Další důležitou součástí aplikace ITEx je čtení a interpretace předpisu testu, zapsaného ve formátu XML. Předpis testu začíná fází kontroly požadavků, kdy se ověří, zda prostředí testu (především dostupné karty a jejich rozhraní) odpovídá požadavkům testu. Následuje příprava testu, v níž se do karet nahraje testovací program. Potom již následuje samotný test. Ten je zpravidla rozdělen do několika kroků. U každého kroku se dá jednotlivě rozhodnout, zda proběhl úspěšně, a v případě chyby stanovit příčinu neúspěchu. Krok testu se dá dále rozdělit na několik příkazů, které odpovídají základním strukturám počítačového programu: Nalezneme zde čtení a nastavování proměnných, logické výrazy, podmínky a cykly. Tyto struktury budou během činnosti aplikace čteny ze souboru a postupně prováděny – interpretovány.

Výše uvedenému členění odpovídá i navržená objektová struktura interpreta předpisu testu: V rámci aplikace bude existovat jeden objekt třídy `TestExecutor`, který zajistí kontrolu požadavků a přípravu testu, založí globální proměnné testu (ty jsou platné po celou dobu testu) a poté postupně pro každý krok vytvoří samostatný objekt `StepExecutor` a předá mu řízení. `StepExecutor` založí lokální proměnné (platné po dobu vykonávání kroku) a poté předá řízení objektu `CommandExecutor`. Ten již zajistí interpretování jednotlivých příkazů, které nalezne při průchodu určené části souboru XML. Pokud je při tom potřeba vyhodnocovat aritmetické nebo logické výrazy, využije se k tomu třída `Expressions`.

5.5.5.1. Třída `Expressions`

Třída `Expressions` sdružuje sadu funkcí, které slouží k vyhodnocení aritmetických a logických výrazů zapsaných v předpisu testu pomocí jazyka XML. Vzhledem k tomu, že ke své činnosti využívá testové proměnné, vyžaduje konstruktor této třídy ukazatel na objekt `VariableManager`, v němž by měly být potřebné proměnné uloženy.

Vyhodnocení různých výrazů je seskupeno podle toho, co má být výsledkem. V případě metody `GetValue_int()`, jak napovídá i název, je výsledkem celé číslo. To může nastat v případě přímo zadané celočíselné hodnoty v tagu `<val>`, požadavku na vyčtení hodnoty proměnné určené tagem `<var>` nebo v případě aritmetického výrazu v tagu `<expression>`. Podle toho také metoda po rozlišení obsahu volá příslušnou soukromou metodu `cmdVal_int()`, `cmdVar_int()`, nebo `cmdArithmExpression()`.

Analogická situace nastává v případě, kdy má být výsledkem pravdivostní hodnota (datový typ `bool`). K tomu slouží metoda `GetValue_bool()`, která podobně jako v případě popsaném o odstavec výše volá soukromé metody `cmdVal_bool()`, `cmdVar_bool()`, nebo `cmdBinaryExpression()`.

Při práci s poli se využije metoda `GetIndex()`, která vyčte hodnotu indexu proměnné, na nějž se předpis testu odkazuje. Index může být určen buď přímo hodnotou, nebo odkazem na jinou proměnnou. V případě, že se určení indexu nepodaří najít, vrací metoda hodnotu 0, která stejně jako úplné vynechání indexu při volání funkce `TestVariable::GetInt()` způsobí, že se bude pracovat s prvním prvkem pole.

Poslední veřejnou metodou je `CheckElement()`. Tu volají soukromé metody třídy `Expressions` a `CommandExecutor` (jež je potomkem `Expressions`), aby se ujistily, že uzel, který jim byl předán v parametru, odpovídá typu, s nímž umí daná metoda pracovat. Zároveň provede metoda `CheckElement()` převod objektové reprezentace uzlu na typ, který umožňuje (mimo jiné) práci s atributy tagů v jazyce XML.

5.5.5.2. Třída `CommandExecutor`

Třída `CommandExecutor` zajišťuje vykonávání příkazů zapsaných v předpisu testu ve formátu XML. Její konstruktor vyžaduje zadání ukazatele na objekt `StepExecutor` a `CardManager`, se kterými se při vykonávání příkazů testu pracuje. Provedení příkazu se zajistí zavoláním metody `ExecuteCommand()` a předáním objektové reprezentace příslušného uzlu souboru XML jako parametru. Metoda rozpozná, jaký příkaz jí byl předán, a poté zavolá jednu z mnoha soukromých metod, které se specializují na jednotlivé příkazy. Metoda `ExecuteBlock()` slouží k vykonání všech příkazů, které se v souboru XML nacházejí za sebou v jednom bloku, neboli jako poduzly jednoho uzlu. Tento nadřízený uzel se předává metodě `ExecuteBlock()` jako parametr. Podstatou této metody je poté opakované volání `ExecuteCommand()` pro všechny jeho poduzly.

Mezi soukromými metodami, starajícími se o vykonávání jednotlivých druhů příkazů, můžeme nalézt několik společných znaků. Je to například skutečnost, že žádný příkaz implementovaný soukromou metodou třídy `CommandExecutor` nemá výstup (mimo obvyklý `bool`, viz kapitolu 5.3.2). Příkazy mající výstup byly totiž sdruženy do třídy `Expressions`, aby byly dostupné i z částí programu, v nichž se nejedná o provádění bloku příkazů – tedy především při vytváření proměnných. Většina metod třídy `CommandExecutor` obsahuje volání metod

třídy `Expressions`, aby vyhodnotily výraz či proměnnou, s nimiž pracují. Metody zajišťující podmíněné nebo cyklické vykonání bloku příkazů (tedy `cmdIf()` a `cmdFor()`) obsahují volání metody `ExecuteBlock()`, takže je na blok podřízených příkazů pohlíženo zcela stejně, jako kdyby byly zapsány přímo v hlavním bloku kroku testu.

5.5.5.3. Třída `StepExecutor`

Objekty třídy `StepExecutor` dostávají za úkol provést celý krok testu. Ten se může skládat z definice lokálních proměnných daného kroku a nejméně jednoho příkazu. Aby mohl mít objekt `StepExecutor` vlastní proměnné, dědí od třídy `VariableManager`. To zajišťuje, aby byl `StepExecutor` v některých případech, kdy je to výhodné, zaměnitelný za `VariableManager`.[16]

`StepExecutor` obsahuje pouze jednu metodu určenou pro volání objektem, který je ve struktuře programu (viz obrázek 5) nad ním, totiž `ExecuteStep()`. Ta zajistí vytvoření lokálních proměnných a zavolá funkci `CommandExecutor::ExecuteBlock()`, jež vykoná všechny příkazy kroku. V průběhu vykonávání příkazů má `CommandExecutor` k dispozici metodu `GetVar()`, která rozšiřuje funkci `VariableManager::GetVar()` o vyhledávání zadané proměnné mezi globálními, pokud nebude nalezena mezi lokálními. Dále může `CommandExecutor` využít metody `SetFailed()` nebo `SetOK()`, kterými dává najevo výsledek kroku, když narazí na odpovídající příkaz v předpisu testu.

5.5.5.4. Třída `TestExecutor`

Všechny probrané třídy zastřešuje třída `TestExecutor`. Její konstruktor vyžaduje zadání ukazatele na objekt `ITEDComm`, který bude celý test používat. Dále třída obsahuje jedinou veřejnou metodu `ExecuteTest()`, jejímž zavoláním se spouští vykonávání celého testového scénáře ze souboru zadaného parametrem metody.

Soukromé metody vyčleňují přípravné fáze do samostatných operací, jejichž výsledek lze kontrolovat a v případě chyby dojde k ukončení provádění testu. Nejprve tak dojde k načtení předpisu testu a vytvoření jeho objektové reprezentace ve formě DOM (*Document Object Model*). To zajistí metoda `loadFile()`. Následuje kontrola verze zařízení ITED (přesněji jeho softwaru) zajištěná metodou `checkITEDVersion()`, načtení karet z ITEDu, kontrola

16 Především pokud je třídě `Expressions` předáván ukazatel na objekt `VariableManager`, aby pracovala s proměnnými v něm obsaženými. To může nastat jak v době, kdy existuje pouze globální `VariableManager` s globálními proměnnými, tak v době, kdy existuje i `StepExecutor` s příslušnými lokálními proměnnými a přístupem ke globálním proměnným. Takto mezi nimi třída `Expressions` nemusí rozlišovat.

požadavků předpisu testu metodou `checkRequirements()`, příprava testu implementovaná v metodě `prepareTest()` a pak již načtení globálních proměnných a vykonávání jednotlivých kroků testu.

6. Testování

Každý vytvořený software je potřeba testovat, zvláště pokud má plnit bezpečnostní funkce. Proto i program mikrořadiče ITED a aplikace ITEx procházely během svého vývoje testováním. Testy byly v případě obou programů vytvářeny ručně jako jednotkové testy (*unit tests*) na jednotlivé jednotky či třídy, které při vývoji postupně vznikaly. Cílem byl samozřejmě úspěšný průchod všech testů. Pokud k tomu na první pokus nedošlo, bylo nutné zjistit příčiny neúspěchu a program podle toho náležitě upravit.

Vykonání testu se zajistí nahrazením hlavního programu (`main.c` nebo `main.cpp`) jedním ze souborů s předpisem testu, které jsou umístěny ve složce se zdrojovým kódem programu v podsložce `tests`, zkompilováním a spuštěním. Jednotlivé testy jsou pojmenovány podle jednotky, ke které se vztahují, a s případným upřesněním za podtržítkem, pokud k jedné jednotce vzniklo více testů.

6.1. Program zařízení ITED

K programu mikrořadiče v zařízení ITED byly vytvořeny testy pro obsluhu periferie USART a komunikaci protokolem ITEDComm. Pro periferii SPI a obsluhu posuvného registru samostatné testy nevznikly, protože byl jednak k jejich ovládání použit již vyzkoušený kód z dřívějších projektů a jednak byla rozpojovací deska, se kterou se po těchto rozhraních komunikuje, dodána až v době, kdy už byla otestována komunikace přes ITEDComm pro mikrořadič i aplikaci ITEx, a tak mohla být obsluha posuvného registru testována přímo zasíláním příslušných rámců protokolu ITEDComm z PC, bez nutnosti připravovat speciální testovací kód na mikrořadiči.

6.1.1. Periferie USART

Testy pro samostatný USART používají jako protistranu PC se spuštěným terminálovým programem, jako je například HyperTerminal, který je součástí MS Windows XP. Sériový port musí být v tomto programu nastaven stejně jako pro ITEDComm, tedy na 8 datových bitů, 1 stop-bit, sudou paritu a rychlost 115,2 kbit/s.

Jednodušší test `usart_char.c` poslouchá na portu RS-232, a pokud přijde znak (8 bitů), zobrazí ho v binárním kódu na 8 LED, které jsou součástí vývojového kitu, a poté znak odešle zpět do PC.

O něco málo komplexnější test `usart_array.c` slouží k otestování přijímání dat do pole a odeslání pole. Program sbírá přicházející znaky v počtu určeném konstantou `ARRAY_LENGTH` a po přijetí daného počtu znaků je všechny odešle zpět do PC, kde je terminálový program zobrazí.

6.1.2. Komunikace protokolem ITEDComm

Testy pro komunikační protokol ITEDComm jsou rovněž dva. Jednodušší z nich se jmenuje `ic_getVersion.c`. Program provede inicializaci zařízení a rozsvítí LED0. Po úspěšném přijetí rámce protokolu ITEDComm se rozsvítí LED1. Pokud je zaslaný příkaz rozpoznán jako GET VERSION, rozsvítí se LED2 a program odešle odpověď obsahující číslo verze 0.1. V případě chyby se rozsvítí LED3.

Druhý test pro ITEDComm se jmenuje přímo `itedcomm.c` a testuje všechny příkazy definované v době jeho vzniku – jsou v něm tedy zahrnuty všechny příkazy s výjimkou SET DEFAULT a DEFAULT ACK. Podmínky pro rozsvícení LED0 až LED2 jsou shodné jako v `ic_getVersion.c`. Dále program rozsvítí LED3 při přijetí příkazu GET CARDS. Poté zašle přednastavenou odpověď: *existují dvě karty s čísly 1 a 2*. Při přijetí příkazu GET CARD INFO se rozsvítí LED4 a odešle se přednastavený název karty, který je *ukA* pro kartu 1 a *ukB* pro kartu 2. Po přijetí příkazu GET INTERFACES se rozsvítí LED5 a program odpoví, že pro kartu 1 *existují dvě rozhraní s čísly 1 a 2*, pro kartu 2 *existují dvě rozhraní s čísly 3 a 4*. Na příkaz GET IFACE INFO program odpovídá vždy pro 1. rozhraní dané karty názvem *power*, pro 2. rozhraní názvem *xbus*. Při přijetí tohoto příkazu se rozsvítí LED6. LED7 slouží pouze k indikaci chyby – její rozsvícení je naprogramováno ve větvi `default` struktury `switch`, takže k jejímu rozsvícení dojde, pokud přijatý příkaz není rozpoznán. Všechny ostatní LED se po proběhnutí všech výše zmíněných příkazů rozsvítí a zůstanou v rozsvíceném stavu. Posledním příkazem, který se projevuje jinak než ostatní, je SET STATE. Při jeho vyvolání se nastaví příslušný stav na LED1 až LED4, která byla v příkazu určena číslem rozhraní (1 až 4). To se ovšem děje v pozitivní logice, takže při povelu k zapnutí rozhraní příslušná LED zhasne. Současně ovšem LED reagují na výše uvedené příkazy, takže např. LED1 se opět rozsvítí při úspěšném přijetí každého dalšího rámce přišlého po SET STATE.

6.1.3. Výstup testů

Uvedenými testy bylo ověřeno správné fungování komponent programu pro zařízení ITED. Funkce celého programu byla ověřena jeho používáním při navazujících testech celého systému.

6.2. Aplikace ITEx

V aplikaci ITEx vznikaly podobným způsobem během postupné implementace tříd i jejich jednotkové testy. Některé testy, zvláště ty, které se týkají interpreta předpisu testu, využívají vstupní soubory XML. Tyto soubory jsou rovněž umístěny ve složce `tests` a zpravidla jsou pojmenovány stejně jako samotný program testu.

6.2.1. Komunikace protokolem ITEDComm

V aplikaci ITEx nebyla testována samotná komunikace přes sériový port, protože její implementace byla převzata ze starších projektů autora této práce, a mohl být rovnou testován protokol ITEDComm.

Ve shodě s odpovídajícím programem pro ITED byla nejprve otestována pouze funkce zjištění verze ITEDu. K tomu slouží test `itedcomm_getVersion.cpp`. Program se po spuštění zeptá ITEDu na jeho verzi a výsledek vypíše na konzoli.

Test na celou jednotku ITEDComm se jmenuje jednoduše `itedcomm.cpp`. Postupně prochází jednotlivé příkazy protokolu a průběh vypisuje na konzoli. Pokud je ITED, s nímž se tento test provádí, vybaven testovacím programem pro ITEDComm[17], bude průběh testu zřejmý i z postupného rozsvěcování LED na kitu.

6.2.2. Komunikace pomocí třídy `JAZZComm`

Komunikaci pomocí třídy `JAZZComm` chyběla do poslední fáze externí aplikace pro download, proto vznikly dva testy. První z nich testuje pouze přímou komunikaci s běžícím programem, druhý používá k simulaci downloadu aplikaci MockupApp (viz kapitolu 5.5.3.3).

Test `jazzcomm_console.cpp` se pokusí vyčíst z běžící karty jednu proměnnou. Výsledek zobrazí na konzoli. Test `jazzcomm.cpp` provede totéž, ale ještě předtím spustí externí proces downloadu a počká na jeho dokončení.

6.2.3. Třída `VariableManager`

Test pro třídu `VariableManager` je první, který kromě souboru s programem testu využívá vstupní soubor XML. Test založí několik proměnných manuálně a několik načte ze

17 Jako *main* je použit soubor `tests/itedcomm.c`.

souboru XML. To se týká jak jednoduchých proměnných, tak i polí. Poté nastavuje a čte uložené hodnoty proměnných. Na závěr je objekt `VariableManager` zrušen a je otestován přístup k již dealokované paměti.

6.2.4. Třída `Card`

Pro třídu `Card` existují dva testy. Prvním z nich je soubor `card_itedcomm.cpp`. Jak napovídá název, specializuje se na funkce, které souvisí s komunikací přes ITEDComm. V podstatě se jedná o podobný test jako dříve uvedený `itedcomm.cpp` s tím rozdílem, že je do něj nyní zahrnuto vytvoření objektů pro nalezené karty.

Druhý, komplexní test `card.cpp` již zahrnuje nejen komunikaci přes ITEDComm, ale i funkce poskytované třídou JAZZComm, tedy download programu a vyčtení proměnné z běžící karty. Program se tedy s využitím metod poskytovaných třídou `Card` pokusí nahrát do karty software, vyčíst z ní proměnnou a poté vypíše karty a rozhraní obsluhované ITEDem.

6.2.5. Třída `CardManager`

Testy pro třídu `CardManager` se mohou zdát nekompletní, protože oba ve svém názvu obsahují upřesnění `itedcomm`. Nicméně `CardManager` slouží ke správě objektů karet a poskytování ukazatelů na ně ostatním součástem programu, a zde tedy stačí na vráceném ukazateli otestovat pouze část funkcí, protože všechny metody třídy `Card` již byly otestovány jejími vlastními testy.

Test `cardmanager_itedcomm.cpp` na rozdíl od předchozích testů nenačítá karty z ITEDu vlastní implementací, ale již prostřednictvím příslušné metody třídy `CardManager`. Karty, s nimiž je poté požadována komunikace, jsou pokaždé specifikovány názvem a vybírány z objektu `CardManager` místo toho, aby na něj byl v hlavním programu udržován vlastní ukazatel.

Totéž provádí i test `cardmanager_itedcomm_set-default.cpp`. Liší se ovšem tím, že do testované komunikace zahrnuje i příkaz SET DEFAULT, který byl do protokolu ITEDComm přidán v průběhu vypracovávání.

6.2.6. Třída `CommandExecutor`

Test `commandexecutor_basic.cpp` pro stejnojmennou třídu ověřuje schopnost provést skupinu příkazů uvedených v předpisu testu, jehož příslušná část je reprezentována souborem `commandexecutor.xml`. Vzhledem k tomu, že příkazy samy o sobě nemohou

signalizovat úspěch nebo neúspěch provedení (lze určit, zda byly příkazy vykonány, ale nikoli zda daly správné výsledky), je možné tento test plně vyhodnotit pouze krokováním programu.

6.2.7. Třída StepExecutor

Test `stepexecutor.cpp` provádí, podobně jako výše uvedený případ, vyhodnocení skupiny příkazů uvedených ve výtahu z předpisu testu, který je reprezentován souborem `stepexecutor.xml`. Tentokrát je již ovšem zahrnuta definice lokálních proměnných kroku a práce s nimi. Vyhodnocení tohoto testu je rovněž možné pouze krokováním programu.

Test `stepexecutor_getVar.cpp` již předpokládá existenci třídy `TestExecutor` a je zaměřen na kontrolu práce s globálními a lokálními proměnnými a překrytí globálních proměnných lokálními. Test již podporuje automatické vyhodnocení výsledků pomocí podmínek zapsaných v předpisu testu, a proto je zahrnuto vypsání výsledku testu na konzoli.

6.2.8. Třída `TestExecutor` a celý program ITEx

Třída `TestExecutor` a s ní i celý program je testován buď testem `testexecutor.cpp`, který provede spuštění aplikace na celém předpisu testu, který ovšem není připraven speciálně pro testování aplikace ITEx a jeho výsledky lze ověřit pouze krokováním programu. Pro otestování celé aplikace včetně třídy `TestExecutor` je určen `basic-test.cpp` popsaný v kapitole 6.3.

6.3. Celkové testování vyvinutého systému

Pro otestování celého systému zahrnujícího aplikaci ITEx se všemi komponentami, zařízení ITED s konečným programem a testovaný systém s vloženou rozpojovací deskou, je určen test `basic-test.cpp`. Tento soubor je zároveň velmi podobný konečnému souboru `main.cpp`, od nějž se liší především tím, že potřebná nastavení (předpis testu, porty) jsou určena již při kompilaci. Zároveň je, možná v rozporu se svým názvem, jedním z nejkomplexnějších mezi uvedenými testy a vyhodnocuje schopnost systému provádět všechny zamýšlené postupy.

Před spuštěním testu je nutné připravit veškerý hardware do stavu, kdy je možné test spustit, tedy jednotlivá zařízení propojit a připojit je k napájení. Poté můžeme přikročit ke spuštění testovací aplikace. Ta nejprve ověří, že má k dispozici kartu *ukA* s rozhraními *power* a *SBus*. Poté do této karty nahraje software. Následuje vlastní testovací scénář.

První krok testu slouží především k ověření správné funkce zařízení ITED a komunikace s ním. Provede se pokus o start karty s rozpojeným komunikačním rozhraní *SBus*, které je ovšem

nutné k tomu, aby karta skutečně nastartovala. Po uplynutí času potřebného ke startu karty se program pokusí vyčíst z ní jednu proměnnou. Pokud se proměnná nenačte, označí se krok jako v pořádku proběhlý, jinak za neúspěšný.

V druhém kroku proběhne další pokus o nastartování karty, tentokrát již ovšem s propojeným rozhraním *SBus*. Po době potřebné ke startu karty se program opět pokusí načíst z karty tutéž proměnnou. Tentokrát bude ovšem přijata jakákoli kladná hodnota proměnné, zatímco jiný výsledek bude považován za chybu.

Třetí krok testuje vyčtení dalších dvou proměnných, které ale tentokrát nepocházejí z kořenového jmenného prostoru karty, takže je nutné zadat jejich úplnou cestu, která má několik úrovní a celkovou délku více než 40 znaků. Krok testu bude úspěšný, pokud budou načtené hodnoty nezáporné.

Poslední krok testu se zaměřuje na cyklické čtení proměnných, jejichž hodnota se v průběhu času zaručeně mění. Tyto proměnné jsou čteny pomocí for-cyklu do pole a jsou tak otestovány i příslušné funkce aplikace ITEx. Krok proběhne úspěšně, pokud je u první proměnné nalezena rostoucí sekvence, u druhé stačí, aby nebyla dvakrát za sebou načtena stejná hodnota[18].

Pokud všechny kroky testu proběhnou úspěšně, program vypíše na konzoli hlášení o úspěšném průběhu a skončí.

6.4. Vyhodnocení testů

Všechny uvedené testy byly vykonány a v případě problémů došlo k opravě, takže nyní všechny testy, které pro existující program připadají v úvahu, probíhají úspěšně. Zvláštní pozornost je věnována testu `basic-test.cpp`, který ověřuje testovací systém jako celek. Pomocí něj byla otestována spolupráce s testovaným systémem, u nějž je nyní možné automaticky vyhodnocovat určené stavy na základě předpisu testu zapsaného ve formátu XML.

Navržený testovací systém lze proto používat ke stanovenému účelu v rozsahu, který je zahrnut v současné implementaci. Je například potřeba doplnit komponentu pro záznam výsledků testu, aby byly splněny požadavky normy.

18 Ve skutečnosti bude i v tomto případě načítána rostoucí sekvence, ale test takto pokryje více možností interpreta.

7. Závěr

Cílem této práce bylo analyzovat, navrhnout a implementovat systém pro automatické testování integrace hardwaru a softwaru v elektronickém železničním zabezpečovacím zařízení. Po analýze možností byla navržena a posléze provedena implementace testovacího systému, skládajícího se z mikrořadiče obsluhujícího rozpojovací desku plošných spojů, která manipuluje s rozhraními testovaného systému, a z aplikace pro řízení testu na osobním počítači. Tyto součásti byly implementovány tak, aby umožňovaly provádět požadované testy na testovaném zařízení, což bylo ověřeno testováním jednotlivých komponent v průběhu vývoje a provedením celkového testu.

Vyvinutý systém bude po doplnění komponenty pro záznam průběhu testu a uživatelského manuálu splňovat požadavky bezpečnostních norem a lze ho používat ke stanovenému účelu a nadále ho rozšiřovat.

8. Zdroje

[1] LESO, M., DOBIÁŠ, R., FARAN, A., et al. *Železniční zabezpečovací technika a systémy.* Koncept vysokoškolského skripta. Praha, 2010. 245 s.

[2] *EN 50129:2003 E* [PDF]. Brusel (Belgie): CENELEC, 2003. 94 s.

[3] *ČSN EN 50128*. Ed. 2. Praha: Úřad pro technickou normalizaci, metrologii a státní zkušebnictví, 2012. 108 s.

[4] *EN 50128:2009 E* [PDF]. WGA11 version 11. Brusel (Belgie): CENELEC, 2008. 137 s.

[5] *USB Relays Control – 32 USB Relay Controller – 3131* [online]. Netanja (Izrael): Intelligent Appliance [cit. 28. února 2013]. Dostupné na: <http://www.usb-relay.org/page10.php>.

[6] *8-bit AVR Instruction Set* [online]. San Jose (Kalifornie, USA): Atmel Corporation, 2010 [cit. 22. dubna 2013]. Dostupné na: <http://www.atmel.com/images/doc0856.pdf>.

[7] Comparison of integrated development environments. In *Wikipedia, the free encyclopedia* [online]. Wikimedia Foundation Inc., 2007–, strana naposledy upravena 6. dubna 2013 [cit. 9. dubna 2013]. Dostupné na: <http://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments>.

[8] ECKEL, B. *Myslíme v jazyku C++*. 1. vyd. Praha: GRADA, 2000. 552 s. ISBN 80-247-9009-2.

9. Seznam zkratek, obrázků, tabulek

Seznam použitých zkratek

Seznam obrázků

Seznam tabulek

Rejstřík

10. Přílohy

Přílohy k této diplomové práci jsou k dispozici především v elektronické podobě na přiloženém CD. Jsou to zejména elektronické verze tohoto dokumentu ve formátu PDF, a to jednak v tiskové verzi `dp_Mateju_tisk.pdf` a jednak ve verzi s hypertextovými odkazy `dp_Mateju.pdf`, která je určena pro zobrazení na počítači.

Dále jsou na CD uloženy složky `ITED` a `ITEx`, obsahující zdrojové kódy, projektové soubory AVR Studia nebo Qt Creatoru a dokumentaci z nástroje Doxygen ve formátu HTML.

Dokumentace z nástroje Doxygen je také uvedena na následujících stranách, nejprve pro program zařízení ITED a dále pro aplikaci ITEx.

Program pomocného mikrořadiče pro testování integrace HW/SW

„Integration Tests Electronics Driver“ neboli „Ovladač elektroniky pro integrační testy“ slouží jako pomocné zařízení při automatizaci testů integrace HW/SW.

Umožňuje osobnímu počítači pomocí sériového portu RS-232 ovládat rozhraní SPI a vyčítat přednastavenou konfiguraci.

Program je určen pro osmibitovou platformu Atmel AVR. Byl používán s mikrořadičem ATmega16, pro jiné mikrořadiče řady AVR mohou být potřeba některé drobné úpravy v platformové vrstvě, tedy jednotkách SPI (**spi.h**), SH_REG (**sh_reg.h**) a USART (**usart.h**).

Dokumentace souborů

Dokumentace souboru src/defs.h

Zakladni sdilene definice.

Definice maker

1 #define **MAX_UI_8_VALUE** ((**UI_8**)-1)
maximum UI_8

Definice typů

2 typedef unsigned char **UI_8**
osmibitove cislo bez znamenka
3 typedef unsigned short **UI_16**
16bitove cislo bez znamenka
4 typedef unsigned char **UI_P**
nativni cislo bez znamenka

Detailní popis

Zakladni sdilene definice.

Verze:
1.0

Datum:
2013-02-22

Autor:
Miroslav Mateju, `mateju.miroslav@azd.cz`

Definice jednoduchych datovych typu pouzivanych v projektu.

Dokumentace souboru src/itedcomm.h

Komunikace po seriovem portu protokolem ITEDComm.

```
#include "defs.h"
```

Výčty

5 enum **IC_Command** { **IC_NO_ITED_COMMAND**, **IC_GET_VERSION**, **IC_VERSION**, **IC_GET_CARDS**, **IC_CARDS**, **IC_GET_CARD_INFO**, **IC_CARD_INFO**, **IC_GET_INTERFACES**, **IC_INTERFACES**, **IC_GET_IFACE_INFO**, **IC_IFACE_INFO**, **IC_SET_STATE**, **IC_STATE_ACK**, **IC_SET_DEFAULT**, **IC_DEFAULT_ACK** }

prikaz pro komunikaci s ITEDem Funkce

6 void **IC_Init** (void)

Inicializace komunikacniho rozhrani.

7 enum **IC_Command IC_GetCommand** (void)

Přijatý příkaz.

8 **UI_8 IC_GetCardID** (void)

Načtení ID karty z přijatého rámce.

9 **UI_8 IC_GetIfaceID** (void)

Načtení ID rozhraní z přijatého rámce.

10 **UI_8 IC_GetStateRequest** (void)

Načtení požadovaného stavu rozhraní.

11 void **IC_StartFrame** (enum **IC_Command** cmd)

Započetí odesílaného rámce.

12 void **IC_PutCardID** (**UI_8** cardID)

Uložení ID karty do odesílaného rámce.

13 void **IC_PutIfaceID** (**UI_8** ifaceID)

Uložení ID rozhraní do odesílaného rámce.

14 void **IC_PutState** (**UI_8** state)

Uložení čísla stavu do odesílaného rámce.

15 void **IC_PutVersion** (**UI_8** major, **UI_8** minor)

Uložení informace o verzi ITEDu do odesílaného rámce.

16 void **IC_PutRange** (**UI_8** first, **UI_8** last)

Uložení rozsahu (karet, rozhraní) do odesílaného rámce.

17 void **IC_PutString** (const char *string)

Uložení textového řetězce do odesílaného rámce.

18 void **IC_FinishFrameAndSend** (void)

Dokončení a odeslání rámce.

19 **UI_P IC_ReceiveFrame** (void)

Přijetí rámce.

Detailní popis

Komunikace po seriovem portu protokolem ITEDComm.

Verze:

1.0

Autor:

Miroslav Mateju, `mateju.miroslav@azd.cz`

Datum:

2013-03-06

Dokumentace výčtových typů

enum IC_Command

prikaz pro komunikaci s ITEDem

Hodnoty výčtu:

IC_NO_ITED_COMMAND oznacuje chybny prikaz

Dokumentace funkcí

void IC_FinishFrameAndSend (void)

Dokončení a odeslání rámce.

Vypočte Block Check Character (BCC) pro odesílaný rámec, uzavře rámec znakem ETX a odešle rámec.

UI_8 IC_GetCardID (void)

Načtení ID karty z přijatého rámce.

Návratová hodnota:

ID karty (1-254).

Vracené hodnoty:

MAX_UI_8_VALUE	ID karty se nepodařilo načíst.

enum IC_Command IC_GetCommand (void)

Přijatý příkaz.

Návratová hodnota:

Vrátí přijatý příkaz.

Vracené hodnoty:

IC_NO_ITED_COMMAND	Značí chybu. Rámec by neměl být dále zpracováván.

UI_8 IC_GetIfaceID (void)

Načtení ID rozhraní z přijatého rámce.

Návratová hodnota:

ID rozhraní (1-254).

Vracené hodnoty:

MAX_UI_8_VALUE	ID rozhraní se nepodařilo načíst.

UI_8 IC_GetStateRequest (void)

Načtení požadovaného stavu rozhraní.

Návratová hodnota:

Požadovaný stav (0-254).

Vracené hodnoty:

MAX_UI_8_VALUE	Číslo stavu se nepodařilo načíst.

void IC_Init (void)

Inicializace komunikacniho rozhrani.

Pozor:

Nutno zavolat pred jakoukoli jinou funkci z teto jednotky.

void IC_PutCardID (UI_8 *cardID*)

Uložení ID karty do odesílaného rámce.

Parametry:

cardID	ID karty pro vložení do rámce (1-254)

void IC_PutIfaceID (UI_8 *ifaceID*)

Uložení ID rozhraní do odesílaného rámce.

Parametry:

ifaceID	ID rozhraní pro vložení do rámce (1-254)

void IC_PutRange (UI_8 *first*, UI_8 *last*)

Uložení rozsahu (karet, rozhraní) do odesílaného rámce.

Parametry:

first	číslo první položky z rozsahu
last	číslo poslední položky z rozsahu

void IC_PutState (UI_8 *state*)

Uložení čísla stavu do odesílaného rámce.

Parametry:

state	číslo stavu pro vložení do rámce (1-254)

void IC_PutString (const char * *string*)

Uložení textového řetězce do odesílaného rámce.

Parametry:

string	řetězec znaků ve formě C-stringu (zakončený \0)

void IC_PutVersion (UI_8 *major*, UI_8 *minor*)

Uložení informace o verzi ITEDu do odesílaného rámce.

Parametry:

major	číslo hlavní verze
minor	číslo podverze

UI_P IC_ReceiveFrame (void)

Přijetí rámce.

Přijme rámec do bufferu.

Vracené hodnoty:

0	Rámec nebyl přijat.
ne0	Byl přijat rámec.

void IC_StartFrame (enum IC_Command *cmd*)

Započetí odesílaného rámce.

Zneplatní rámec v IC_SendFrame, vymaže předchozí rámec z bufferu a vyplní znaky STX a CMD.

Parametry:

cmd	příkaz, který se má vložit jako znak CMD

Dokumentace souboru src/main.c

Hlavni funkce zarizeni ITED.

```
#include "defs.h"
#include "itedcomm.h"
#include "sh_reg.h"
#include <avr/io.h>
```

Definice maker

20 #define **DEBUG_LED_MAIN**

enables debug info

21 #define **LED_PORT** PORTC
22 #define **MAX_STRING_LENGTH** 10

maximalni delka textoveho retezce v konfiguraci VCETNE ukonceni ('\0')

23 #define **CARD_COUNT** 2
24 #define **CardFirst** 1
25 #define **CardLast** (CardFirst + CARD_COUNT - 1)
26 #define **IFACE_COUNT** 13

Funkce

27 int **main** ()

Detailní popis

Hlavni funkce zarizeni ITED.

Verze:
0.3 beta

Autor:
Miroslav Mateju, `mateju.miroslav@azd.cz`

Datum:
2013-03-11

Dokumentace souboru src/sh_reg.h

Shift register driver.

```
#include "defs.h"
```

Funkce

28 void **SR_Init** (void)

Initialize shift register interface.

29 void **SR_DisplayString** (char data[], int length)

Display a string on parallel output.

30 void **SR_Display16** (**UI_16** data)

Display a 16-bit number.

31 void **SR_AdoptData** (void)
Pass data in 74HC595's shift register to its storage register.

Detailní popis

Shift register driver.

Verze:
1.0

Autor:
Miroslav Mateju, `mateju.miroslav@azd.cz`

Datum:
2013-03-28 Functions to manage 74HC595 shift register and its parallel output.

Poznámka:
SR_IOPORT in sh_reg.c must be defined. Example:
```
#define SR_IOPORT PORTB
```

Dokumentace funkcí

void SR_Display16 (UI_16 *data*)

Display a 16-bit number.

Sends a 16-bit number to cascade of 595s and shows it at parallel output.

Parametry:

data	number (bit-array) to display

void SR_DisplayString (char *data*[], int *length*)

Display a string on parallel output.

Executes SPI_SendString() and **SR_AdoptData()**.

Parametry:

data	array of bytes to send
length	count of bytes in "data"

Dokumentace souboru src/spi.h

SPI driver for ATmega.

Funkce

32 void **SPI_Init** (void)
Initialize the SPI peripheral for communication.
33 void **SPI_SendByte** (char data)
Send 8 bits via SPI, does not wait for finishing.
34 void **SPI_SendByteW** (char data)
Send 8 bits via SPI, waits for finished transfer.
35 void **SPI_SendStringW** (char data[], int length)
Send one or more bytes via SPI, waits for finished transfer.
36 char **SPI_ReadByte** (void)

Read 8 bits from SPI data register.

Detailní popis

SPI driver for ATmega.

Verze:
1.0

Autor:
Miroslav Mateju, `mateju.miroslav@azd.cz`

Datum:
2013-03-28

Poznámka:
SPI_Init() must be called as first functions of this driver.
This SPI driver does not use interrupts. Waiting is done using "while", so no other activities will be done meanwhile (except interrupts, if enabled).

Dokumentace funkcí

char SPI_ReadByte (void)

Read 8 bits from SPI data register.

Reads the data already received by using SPI_SendByte(W) or SPI_ReceiveByte(W).

Návratová hodnota:
a byte of received data

void SPI_SendByte (char *data*)

Send 8 bits via SPI, does not wait for finishing.

Parametry:

data	a byte of data to send

void SPI_SendByteW (char *data*)

Send 8 bits via SPI, waits for finished transfer.

Parametry:

data	a byte of data to send

void SPI_SendStringW (char *data*[], int *length*)

Send one or more bytes via SPI, waits for finished transfer.

Parametry:

data	array of bytes to send
length	count of bytes in "data"

Dokumentace souboru src/usart.h

USART driver for ATmega.

```
#include "defs.h"
```

Funkce

37 void **USART_Init** (void)

Initialization of USART interface for communication.

38 void **USART_SendChar** (**UI_8** input)

Sends one byte using USART.

39 **UI_8 USART_ReceiveChar** (void)

Picks up received character from USART.

40 void **USART_SendArray** (**UI_8** *sendArray, **UI_P** length)

Sends a byte array using USART.

41 void **USART_ReceiveArray** (**UI_8** *recvArray, **UI_P** length)

Receives a byte array from USART.

Detailní popis

USART driver for ATmega.

Verze:

1.0

Autor:

Miroslav Mateju, `mateju.miroslav@azd.cz`

Datum:

2013-03-05

IMPORTANT NOTES

USART_Init() must be called before using any other functions from this driver. This will enable global interrupts (SEI). Global interrupt enable (SEI) will be set when transferring data via USART, it also must be set in order to receive data via USART.

Dokumentace funkcí

void USART_Init (void)

Initialization of USART interface for communication.

Poznámka:

Enables global interrupts.

void USART_ReceiveArray (UI_8 * *recvArray*, UI_P *length*)

Receives a byte array from USART.

Blocking until receiving not finished.

Parametry:

recvArray	Array to receive into.
length	Count of bytes to receive.

UI_8 USART_ReceiveChar (void)

Picks up received character from USART.

Návratová hodnota:

Received byte.

void USART_SendArray (UI_8 * *sendArray*, UI_P *length*)

Sends a byte array using USART.

Blocking until sending not finished.

Parametry:

sendArray	Array of bytes to send.
length	Count of bytes to send.

void USART_SendChar (UI_8 *input*)

Sends one byte using USART.

Aplikace pro řízení testů integrace HW/SW

„Integration Tests Executor“ neboli „Vykonavatel integračních testů“ slouží k řízení automatických testů integrace HW/SW.

Program je určen pro operační systém Windows a framework Qt. Z Qt se využívá práce s XML. Přenositelnost mezi operačními systémy je omezena třídou **SerialComm**, která je implementována pouze pro Windows, a také spolupracujícími nástroji.

Spuštění aplikace

Ke spuštění aplikace je potřeba spustitelný soubor `itex.exe` . V pracovním adresáři se musí nacházet konfigurační soubory `cards.cfg` a `downloader.cfg` a dynamicky linkované knihovny `QtCore4.dll` a `QtXml4.dll`.

Při spuštění automatického testu je potřeba vyplnit dva povinné parametry:

42 `predpis_testu` – cesta k souboru XML s předpisem testu, odpovídajícím formátu, který je popsán souborem itex.xsd
43 `ITED_port` – název sériového portu, který se má používat pro komunikaci se zařízením ITED

Příklad

```
itex.exe .\basic-test.xml COM1
```

Konfigurační soubory

Soubor `downloader.cfg` obsahuje na prvním řádku cestu ke spustitelnému souboru, který se má použít pro nahrávání programu do karet.

Příklad

```
D:\Programs\Downloader\downloader.exe
```

Soubor `cards.cfg` obsahuje přiřazení sériových portů jednotlivým testovaným kartám. Na každém řádku je uveden název karty a označení portu.

Příklad

```
ukA COM4
ukB COM5
```

Překlad (kompilace) programu

Program je přeložitelný kompilátorem pro Windows s dostupným frameworkem Qt. Doporučuje se překládat program v prostředí Qt Creator. Vyzkoušen byl překladač z MS Visual C++ a minGW.

Seznam plánovaných úprav

Člen CardManager::LoadCards (ITEDComm *const _ITEDComm)
Porty pro **JAZZComm** by se měly brát z konfiguračního souboru XML.

Člen JAZZComm::Download (const char *filename)
Implementovat, až bude k dispozici konzolový CapFlash.

Člen JAZZComm::downloaderExecutable []
Načíst název programu z konfiguračního souboru XML.

Člen JAZZComm::receiveAndParseFrame (std::string &response)
Tady by se mělo zkontrolovat CRC, zatím bereme všechny rámce ohraničené určenými znaky jako platné.

Dokumentace tříd

Dokumentace třídy Card

Obsluha testované karty.

`#include <card.h>`

Veřejné metody

44 bool **ChangeInterfaceState** (const std::string &_name, **UI_8** _newState)
Změna stavu rozhraní karty.

45 bool **SetAllInterfacesState** (**UI_8** _newState)
Nastavení stavu všech rozhraní karty.

46 bool **DownloadProgram** (const char *_filename)
Nahraje do obsluhované karty program ze zadaného souboru.

47 bool **GetVariable** (const char *_variable, std::string &_response)
Načte z karty proměnnou.

48 **Card** (**UI_8** _cardID, **ITEDComm** *const _ITEDComm, const char *_JAZZCommPort)
Konstruktor karty se známým umístěním připojení.

49 **Card** (const **Card** &_source)
Kopírovací konstruktor.

Privátní typy

50 typedef std::map< std::string,
51 **UI_8** > **ifaceRecs**

Privátní metody

52 bool **loadInterfaces** ()
Načtení rozhraní karty do IfaceRecords.

Privátní atributy

53 const **UI_8 cardID**
*ID karty při komunikaci přes **ITEDComm**.*

54 **ITEDComm** *const **cardITEDComm**
*Ukazatel na globální objekt **ITEDComm**, který komunikuje s ITEDem příslušným k rozhraním dané karty.*

55 **JAZZComm cardJAZZComm**
*Objekt pro komunikaci přes **JAZZComm**.*

56 ifaceRecs **ifaceRecords**
Úložiště informací o rozhraních karty.

Detailní popis

Obsluha testované karty.

Zapouzdřuje veškerou komunikaci s testovanou kartou probíhající prostřednictvím rozhraní **JAZZComm** a **ITEDComm.**

Dokumentace konstruktoru a destruktoru

Card::Card (UI_8 *_cardID*, ITEDComm *const *_ITEDComm*, const char * *_JAZZCommPort*)

Konstruktor karty se známým umístěním připojení.

Parametry:

_cardID	ID karty při komunikaci přes **ITEDComm**
_ITEDComm	objekt obsluhující rozhraní karty
_JAZZCommPort	sériový port pro komunikaci přes **JAZZComm**

Card::Card (const Card & *_source*)

Kopírovací konstruktor.

Pozor:

Není ošetřena obsluha téhož portu JAZZCommu oběma objekty!

Dokumentace k metodám

bool Card::ChangeInterfaceState (const std::string & *_name*, UI_8 *_newState*)

Změna stavu rozhraní karty.

Pošle ITEDu příkaz na změnu stavu rozhraní.

Vracené hodnoty:

true	Stav byl úspěšně změněn.

Parametry:

_name	název rozhraní
_newState	stav, který má být nastaven

bool Card::DownloadProgram (const char * *_filename*)

Nahraje do obsluhované karty program ze zadaného souboru.

Vracené hodnoty:

true	Program byl úspěšně nahrán na testovanou kartu.

Parametry:

_filename	název souboru podle konvence OS

bool Card::GetVariable (const char * *_variable*, std::string & *_response*)

Načte z karty proměnnou.

Funkce odešle kartě příkaz k načtení proměnné a předá zpět výsledek.

Vracené hodnoty:

true	Příkaz se podařilo vykonat.
false	Příkaz se NEpodařilo vykonat.

Parametry:

_variable	proměnná ve formátu Property Console
_response	odpověď karty, pokud byla načtena

bool Card::loadInterfaces () [private]

Načtení rozhraní karty do IfaceRecords.

Vracené hodnoty:

true	Rozhraní byla úspěšně načtena.

bool Card::SetAllInterfacesState (UI_8 *_newState*)

Nastavení stavu všech rozhraní karty.

Pošle ITEDu příkaz na nastavení stavu všech rozhraní.

Vracené hodnoty:

true	Stav byl úspěšně změněn.

Parametry:

_newState	stav, který má být nastaven

Dokumentace k datovým členům

ITEDComm* const Card::cardITEDComm [private]

Ukazatel na globální objekt **ITEDComm**, který komunikuje s ITEDem příslušným k rozhraním dané karty.

Dokumentace pro tuto třídu byla generována z následujících souborů:

57 card.h
58 card.cpp

Dokumentace třídy CardManager

Správce testovaných karet.

`#include <cardmanager.h>`

Veřejné metody

59 bool **NewCard** (std::string _name, **UI_8** _cardID, **ITEDComm** *const _ITEDComm, const char *_JAZZCommPort)

Vytvoření nového objektu karty.

60 **Card** * **GetCard** (std::string _name)

Získání ukazatele na objekt karty.

61 **UI_P LoadCards** (**ITEDComm** *const _ITEDComm)

Načtení karet z ITEDu.

Privátní typy

62 typedef std::map< std::string,

63 **Card** > **cardRecs_t**

Privátní atributy

64 cardRecs_t **cardRecords**

Detailní popis

Správce testovaných karet.

Spravuje karty, s nimimž se pracuje v testovacím scénáři. Zajišťuje vytváření a rušení jejich objektů a poskytuje k nim přístup podle jména.

Dokumentace k metodám

Card * CardManager::GetCard (std::string *_name*)

Získání ukazatele na objekt karty.

Najde kartu podle jména a vrátí ukazatel na ni.

Návratová hodnota:

Ukazatel na nalezenou kartu.

Vracené hodnoty:

NULL	V případě, že karta zadaného jména neexistuje.

Parametry:

_name	název hledané karty

UI_P CardManager::LoadCards (ITEDComm *const *_ITEDComm*)

Načtení karet z ITEDu.

Zjistí, jaké karty obsluhuje ITED, s nímž je spojen zadaný **ITEDComm**, a zaregistruje je.

Parametry:

_ITEDComm	ukazatel na **ITEDComm**, z nějž mají být načteny karty

Návratová hodnota:

Počet načtených karet.

Vracené hodnoty:

0	Nenačetla se žádná karta - mohlo dojít k chybě.

Plánované úpravy:

Porty pro **JAZZComm** by se měly brát z konfiguračního souboru XML.

bool CardManager::NewCard (std::string *_name*, UI_8 *_cardID*, ITEDComm *const *_ITEDComm*, const char * *_JAZZCommPort*)

Vytvoření nového objektu karty.

Zaregistruje novou kartu.

Vracené hodnoty:

false	Objekt karty se nepodařilo vytvořit (např. již nějaká toho jména existuje).

Parametry:

_name	název proměnné
_cardID	ID karty při komunikaci přes **ITEDComm**
_ITEDComm	objekt obsluhující rozhraní karty
_JAZZCommPort	sériový port pro komunikaci přes **JAZZComm**

Dokumentace pro tuto třídu byla generována z následujících souborů:

65 **cardmanager.h**
66 cardmanager.cpp

Dokumentace třídy CommandExecutor

Vykonavatel příkazu testu.

`#include <commandexecutor.h>`

Diagram dědičnosti pro třídu CommandExecutor

Veřejné metody

67 bool **ExecuteCommand** (const QDomNode &_command)
Vykoná předaný příkaz.
68 bool **ExecuteBlock** (const QDomNode &_block)
Vykoná blok příkazů.
69 **CommandExecutor** (**StepExecutor** *const _step, **CardManager** *const _cardMan)
*Konstruktor pro použití objektem **StepExecutor**.*

Privátní metody

70 bool **cmdConnect** (const QDomNode &_cmd)
Všechny funkce cmd vrací bool, který říká, zda příkaz proběhl úspěšně.*
71 bool **cmdDisconnect** (const QDomNode &_cmd)
Odpojení rozhraní.
72 bool **cmdTimedelay** (const QDomNode &_cmd)
Pozastavení vykonávání na určenou dobu.
73 bool **cmdGetvar** (const QDomNode &_cmd)
Načtení proměnné z testované karty.
74 bool **cmdSet** (const QDomNode &_cmd)
Nastavení hodnoty proměnné testu.
75 bool **cmdIf** (const QDomNode &_cmd)
Podmíněné vykonání příkazu / skupiny příkazů.
76 bool **cmdFor** (const QDomNode &_cmd)
Opakované vykonání příkazu / skupiny příkazů.
77 bool **cmdOk** (const QDomNode &_cmd)
Označení kroku testu za úspěšný.
78 bool **cmdFail** (const QDomNode &_cmd)
Označení kroku testu za NEúspěšný.

Privátní atributy

79 **StepExecutor** *const **step**
*odkaz na nadřízený **StepExecutor***

80 **CardManager** *const **cardMan**
odkaz na správce karet

Detailní popis

Vykonavatel příkazu testu.

Načte příkaz testu (nebo jejich blok) a vykoná ho.

Dokumentace konstruktoru a destruktoru

CommandExecutor::CommandExecutor (StepExecutor *const *_step*, CardManager *const *_cardMan*)

Konstruktor pro použití objektem **StepExecutor**.

Objekt **CommandExecutor** musí mít přístup k objektům **CardManager** a Logger, které jsou součástí zastřešujících objektů **StepExecutor** a **TestExecutor**.

Parametry:

_step	krok testu, pod nějž příkaz patří
_cardMan	správce karet

Dokumentace k metodám

bool CommandExecutor::cmdConnect (const QDomNode & *_cmd*) `[private]`

Všechny funkce cmd* vrací bool, který říká, zda příkaz proběhl úspěšně.

Připojení rozhraní.

Parametry:

_cmd	uzel s příkazem v souboru testu

bool CommandExecutor::cmdDisconnect (const QDomNode & *_cmd*) `[private]`

Odpojení rozhraní.

Parametry:

_cmd	uzel s příkazem v souboru testu

bool CommandExecutor::cmdFail (const QDomNode & *_cmd*) `[private]`

Označení kroku testu za NEúspěšný.

Parametry:

_cmd	uzel s příkazem v souboru testu

bool CommandExecutor::cmdFor (const QDomNode & *_cmd*) `[private]`

Opakované vykonání příkazu / skupiny příkazů.

V tagu <number> je možno uvedením atributu varID vytvořit novou proměnnou testu (nebo zadat stávající), která se bude měnit spolu s indexem cyklu. Vytvořená proměnná v současné implementaci existuje po zbytek kroku testu (do dosažení </step>, přesněji </execute>).

Parametry:

_cmd	uzel s příkazem v souboru testu

bool CommandExecutor::cmdGetvar (const QDomNode & *_cmd*) `[private]`

Načtení proměnné z testované karty.

Parametry:

_cmd	uzel s příkazem v souboru testu

bool CommandExecutor::cmdIf (const QDomNode & *_cmd*) `[private]`

Podmíněné vykonání příkazu / skupiny příkazů.

Parametry:

_cmd	uzel s příkazem v souboru testu

bool CommandExecutor::cmdOk (const QDomNode & *_cmd*) `[private]`

Označení kroku testu za úspěšný.

Parametry:

_cmd	uzel s příkazem v souboru testu

bool CommandExecutor::cmdSet (const QDomNode & *_cmd*) `[private]`

Nastavení hodnoty proměnné testu.

Parametry:

_cmd	uzel s příkazem v souboru testu

bool CommandExecutor::cmdTimedelay (const QDomNode & *_cmd*) `[private]`

Pozastavení vykonávání na určenou dobu.

Parametry:

_cmd	uzel s příkazem v souboru testu

bool CommandExecutor::ExecuteBlock (const QDomNode & *_block*)

Vykoná blok příkazů.

Vykoná příkazy, které jsou potomky zadaného uzlu XML dokumentu. Pokud nějaký příkaz selže, pokračuje vykonáváním dalších příkazů.

Vracené hodnoty:

true	Příkazy byly vykonány úspěšně.
false	V některém příkazu došlo k chybě.

Parametry:

_block	rodičovský uzel bloku příkazů

bool CommandExecutor::ExecuteCommand (const QDomNode & *_command*)

Vykoná předaný příkaz.

Vracené hodnoty:

true	Příkaz byl vykonán úspěšně.

Parametry:

_command	uzel s příkazem v souboru testu

Dokumentace pro tuto třídu byla generována z následujících souborů:

81 commandexecutor.h
82 commandexecutor.cpp

Dokumentace třídy Expressions

Sada funkcí, které slouží k vyhodnocení výrazů v XML.

`#include <expressions.h>`

Diagram dědičnosti pro třídu Expressions

Veřejné metody

83 **Expressions** (**VariableManager** *const _varMan)
Konstruktor.
84 bool **GetValue_int** (const QDomNode &_cmd, **SI_P** &_result)
Zastřešení příkazů vracejících proměnnou typu int.
85 bool **GetValue_bool** (const QDomNode &_cmd, bool &_result)
Zastřešení příkazů vracejících proměnnou typu bool.
86 **UI_P GetIndex** (const QDomNode &_cmd)
Zastřešení zjištění indexu pro pole testovacích proměnných.
87 QDomElement **CheckElement** (const QDomNode &_node, QString _name)
Zkontroluje název tagu a vrátí ho jako QDomElement.

Chráněné atributy

88 **VariableManager** *const **varMan**

Privátní metody

89 bool **cmdVal_int** (const QDomNode &_cmd, **SI_P** &_result)
Načtení přímo zadané hodnoty typu int.
90 bool **cmdVal_bool** (const QDomNode &_cmd, bool &_result)
Načtení přímo zadané hodnoty typu bool.
91 bool **cmdVar_int** (const QDomNode &_cmd, **SI_P** &_result)
Načtení hodnoty proměnné typu int.
92 bool **cmdVar_bool** (const QDomNode &_cmd, bool &_result)
Načtení hodnoty proměnné typu bool.
93 bool **cmdArithmExpression** (const QDomNode &_cmd, **SI_P** &_result)
Vyhodnocení aritmetického výrazu.
94 bool **cmdBinaryExpression** (const QDomNode &_cmd, bool &_result)

Vyhodnocení logického výrazu.

Detailní popis

Sada funkcí, které slouží k vyhodnocení výrazů v XML.

Všechny funkce (s výjimkou getIndex()) vrací bool, který říká, zda příkaz proběhl úspěšně. Výsledek operace je potom uložen do posledního parametru.

Dokumentace konstruktoru a destruktoru

Expressions::Expressions (VariableManager *const *_varMan*)

Konstruktor.

Vyžaduje přístup ke správci proměnných.

Parametry:

_varMan	Ukazatel na **VariableManager**, který obsahuje proměnné, jež se používají ve vyhodnocovaných výrazech.

Dokumentace k metodám

QDomElement Expressions::CheckElement (const QDomNode & *_node*, QString *_name*) `[inline]`

Zkontroluje název tagu a vrátí ho jako QDomElement.

Parametry:

_node	uzel dokumentu XML k ověření
_name	požadovaný název uzlu

bool Expressions::cmdArithmExpression (const QDomNode & *_cmd*, SI_P & *_result*) `[private]`

Vyhodnocení aritmetického výrazu.

Parametry:

_cmd	uzel s příkazem v souboru testu
_result	výstupní proměnná

bool Expressions::cmdBinaryExpression (const QDomNode & *_cmd*, bool & *_result*) `[private]`

Vyhodnocení logického výrazu.

Parametry:

_cmd	uzel s příkazem v souboru testu
_result	výstupní proměnná

bool Expressions::cmdVal_bool (const QDomNode & *_cmd*, bool & *_result*) `[private]`

Načtení přímo zadané hodnoty typu bool.

Parametry:

_cmd	uzel s příkazem v souboru testu
_result	výstupní proměnná

bool Expressions::cmdVal_int (const QDomNode & *_cmd*, SI_P & *_result*) [private]

Načtení přímo zadané hodnoty typu int.

Parametry:

_cmd	uzel s příkazem v souboru testu
_result	výstupní proměnná

bool Expressions::cmdVar_bool (const QDomNode & *_cmd*, bool & *_result*) [private]

Načtení hodnoty proměnné typu bool.

Parametry:

_cmd	uzel s příkazem v souboru testu
_result	výstupní proměnná

bool Expressions::cmdVar_int (const QDomNode & *_cmd*, SI_P & *_result*) [private]

Načtení hodnoty proměnné typu int.

Parametry:

_cmd	uzel s příkazem v souboru testu
_result	výstupní proměnná

UI_P Expressions::GetIndex (const QDomNode & *_cmd*)

Zastřešení zjištění indexu pro pole testovacích proměnných.

Index může být uložen buď v atributu index přímo, nebo jako volání proměnné atributem indexID. Pokud je v uzlu přítomno obojí, dá se přednost přímo zadané hodnotě.

Návratová hodnota:

Určený index v poli.

Vracené hodnoty:

0	Index se nepodařilo načíst (a nebo byla skutečně zadána 0).

Parametry:

_cmd	uzel, který má obsahovat určení indexu

bool Expressions::GetValue_bool (const QDomNode & *_cmd*, bool & *_result*)

Zastřešení příkazů vracejících proměnnou typu bool.

Obdoba ExecuteCommand pro příkazy, které mají výstupní proměnnou, tj. <val>, *a <expression>. Tuto funkci volají příkazy, které chtějí převzít nějakou hodnotu.*

Parametry:

_cmd	uzel, v němž se má skrývat hodnota
_result	výstupní proměnná

bool Expressions::GetValue_int (const QDomNode & *_cmd*, SI_P & *_result*)

Zastřešení příkazů vracejících proměnnou typu int.

Obdoba ExecuteCommand pro příkazy, které mají výstupní proměnnou, tj. <val>, *a <expression>. Tuto funkci volají příkazy, které chtějí převzít nějakou hodnotu.*

Parametry:

_cmd	uzel, v němž se má skrývat hodnota
_result	výstupní proměnná

Dokumentace pro tuto třídu byla generována z následujících souborů:

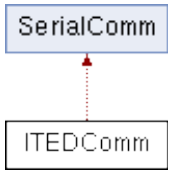
95 expressions.h
96 expressions.cpp

Dokumentace třídy ITEDComm

Komunikace se zařízením ITED protokolem **ITEDComm**.

`#include <itedcomm.h>`

Diagram dědičnosti pro třídu ITEDComm



Veřejné metody

97 bool **GetVersion** (**UI_P** &major, **UI_P** &minor)
Načtení informací o verzi ITEDu.
98 bool **GetCards** (**UI_8** &count, **UI_8** &first, **UI_8** &last)
Načtení počtu a číslování karet obsluhovaných ITEDem.
99 bool **GetCardName** (**UI_8** cardID, std::string &name)
Načtení názvu karty.
100 bool **GetInterfaces** (**UI_8** cardID, **UI_8** &count, **UI_8** &first, **UI_8** &last)
Načtení počtu a číslování rozhraní karty obsluhovaných ITEDem.
101 bool **GetInterfaceName** (**UI_8** cardID, **UI_8** ifaceID, std::string &name)
Načtení názvu rozhraní.
102 bool **SetState** (**UI_8** cardID, **UI_8** ifaceID, **UI_8** state)
Nastavení nového stavu.
103 bool **SetDefaultState** (**UI_8** cardID, **UI_8** state)
Nastavení nového výchozího stavu.
104 **ITEDComm** (const char ***port**)
Vyžaduje zadání portu, který rovnou otevře.

Privátní typy

105enum **ITEDCommand** { **NO_ITED_COMMAND**, **GET_VERSION**, **VERSION**, **GET_CARDS**, **CARDS**, **GET_CARD_INFO**, **CARD_INFO**, **GET_INTERFACES**, **INTERFACES**, **GET_IFACE_INFO**, **IFACE_INFO**, **SET_STATE**, **STATE_ACK**, **SET_DEFAULT**, **DEFAULT_ACK** }

příkaz pro komunikaci s ITEDem Privátní metody

106 bool **sendFrame** (**ITEDCommand** cmd, **UI_8** cardID=0, **UI_8** ifaceID=0, **UI_8** param=0)
Odeslání rámce.
107 bool **receiveFrame** ()
Přijetí rámce.
108 **ITEDCommand getCommand** ()
Přijatý příkaz.

109 bool **finishFrameAndSend** (**UI_P** posBCC)
Dokončení a odeslání rámce.
110 bool **finishReceivingFrame** (**UI_P** firstFreePosition, **UI_P** bytesToReceive)
Dokončení přijímání rámce.

Privátní atributy

111 bool **receivedValidFrame**
Proměnná udávající, zda je k dispozici platný přijatý rámec.
112 **UI_8 sendFrameBuffer** [**maxFrameLength**]
buffer pro ukládání obsahu odesílaného rámce
113 **UI_8 receiveFrameBuffer** [**maxFrameLength**]
buffer pro ukládání obsahu přijatého rámce

Statické privátní atributy

114 static const **UI_P maxFrameLength** = 16
Maximální délka rámce v bajtech. Použije se pro alokaci bufferu.
115 static const **UI_8 STX** = 0x5A
znak začátku rámce
116 static const **UI_8 ETX** = 0xA5
znak konce rámce
117 static const **UI_P posSTX** = 0
pozice znaku STX
118 static const **UI_P posCMD** = 1
pozice znaku CMD
119 static const **UI_P posID** = 2
pozice znaku ID
120 static const **UI_P posIDi** = 3
pozice znaku IDi
121 static const **UI_P posLEN** = 4
pozice znaku LEN

Detailní popis

Komunikace se zařízením ITED protokolem **ITEDComm**.

Zapouzdřuje veškerou komunikaci s ITEDem po sériovém portu a poskytuje snadno použitelné rozhraní ve formě funkcí get... a set... Stará se o vytvoření rámce pro odesílaný příkaz a dekódování přijatého rámce.

Dokumentace k členským výčtům

enum ITEDComm::ITEDCommand `[private]`

příkaz pro komunikaci s ITEDem

Hodnoty výčtu:

NO_ITED_COMMAND označuje chybný příkaz

Dokumentace konstruktoru a destruktoru

ITEDComm::ITEDComm (const char * *port*)

Vyžaduje zadání portu, který rovnou otevře.

Poznámka:

Po otevření portu je vhodné zkontrolovat verzi ITEDu s použitím funkce **GetVersion()**.

Parametry:

port	název portu v OS

Dokumentace k metodám

bool ITEDComm::finishFrameAndSend (UI_P *posBCC*) `[private]`

Dokončení a odeslání rámce.

Vypočte Block Check Character (BCC) pro odesílaný rámec, uzavře rámec znakem ETX a odešle rámec.

Vracené hodnoty:

true	Odeslání proběhlo úspěšně.

Parametry:

posBCC	index bajtu v rámci, kde bude umístěn BCC

bool ITEDComm::finishReceivingFrame (UI_P *firstFreePosition*, UI_P *bytesToReceive*) `[private]`

Dokončení přijímání rámce.

Zavolá se v okamžiku, kdy je podle typu rámce známa jeho délka. Po přijetí provede kontrolu rámce a aktualizuje receivedValidFrame.

Vracené hodnoty:

true	Přijatý rámec je platný.

Parametry:

firstFreePosition	pořadí bajtu, od nějž se má začít přijímat
bytesToReceive	počet bajtů, které zbývá přijmout

bool ITEDComm::GetCardName (UI_8 *cardID*, std::string & *name*)

Načtení názvu karty.

Vracené hodnoty:

true	Podařilo se načíst název karty.

Parametry:

cardID	ID karty, pro kterou má být načten název
name	načtený název karty

bool ITEDComm::GetCards (UI_8 & *count*, UI_8 & *first*, UI_8 & *last*)

Načtení počtu a číslování karet obsluhovaných ITEDem.

Vracené hodnoty:

true	Podařilo se načíst informace o kartách.

Parametry:

count	načtený počet karet
first	načtené číslo první karty
last	načtené číslo poslední karty

ITEDComm::ITEDCommand ITEDComm::getCommand () `[private]`

Přijatý příkaz.

Návratová hodnota:

Vrátí přijatý příkaz.

Vracené hodnoty:

NO_ITED_COMMAND	Došlo k chybě.

bool ITEDComm::GetInterfaceName (UI_8 *cardID*, UI_8 *ifaceID*, std::string & *name*)

Načtení názvu rozhraní.

Vracené hodnoty:

true	Podařilo se načíst název rozhraní.

Parametry:

cardID	ID karty, s jejímž rozhraním se pracuje
ifaceID	ID rozhraní, jehož název má být načten
name	načtený název rozhraní

bool ITEDComm::GetInterfaces (UI_8 *cardID*, UI_8 & *count*, UI_8 & *first*, UI_8 & *last*)

Načtení počtu a číslování rozhraní karty obsluhovaných ITEDem.

Vracené hodnoty:

true	Podařilo se načíst informace o rozhraních.

Parametry:

cardID	ID karty, pro kterou mají být načtena rozhraní
count	načtený počet rozhraní
first	načtené číslo prvního rozhraní
last	načtené číslo posledního rozhraní

bool ITEDComm::GetVersion (UI_P & *major*, UI_P & *minor*)

Načtení informací o verzi ITEDu.

Vracené hodnoty:

true	Podařilo načíst informace o verzi.

Parametry:

major	načtené číslo hlavní verze
minor	načtené číslo podverze

bool ITEDComm::receiveFrame () `[private]`

Přijetí rámce.

Pokusí se z bufferu OS převzít přijatý rámec. Převzatý rámec se potom dá číst pomocí funkcí get... této třídy.

Vracené hodnoty:

true	Podařilo se převzít platný rámec.

bool ITEDComm::sendFrame (ITEDCommand *cmd*, UI_8 *cardID* = 0, UI_8 *ifaceID* = 0, UI_8 *param* = 0) `[private]`

Odeslání rámce.

Příkaz musí být ze sady "downstream", jinak funkce vrátí chybu (false).

Vracené hodnoty:

true	Odeslání se podařilo.
false	Došlo k chybě.

Parametry:

cmd	příkaz, který má být odeslán
cardID	ID karty pro příkazy, které ho používají
ifaceID	ID rozhraní pro příkazy, které ho používají
param	další parametr (v současnosti STT u SET_STATE)

bool ITEDComm::SetDefaultState (UI_8 *cardID*, UI_8 *state*)

Nastavení nového výchozího stavu.

Nastaví shodný stav všem rozhraním karty.

Vracené hodnoty:

true	Stav byl úspěšně nastaven.

Parametry:

cardID	ID karty, s jejímž rozhraním se pracuje
state	požadované číslo stavu

bool ITEDComm::SetState (UI_8 *cardID*, UI_8 *ifaceID*, UI_8 *state*)

Nastavení nového stavu.

Vracené hodnoty:

true	Stav byl úspěšně nastaven.

Parametry:

cardID	ID karty, s jejímž rozhraním se pracuje
ifaceID	ID rozhraní, se kterým se pracuje
state	požadované číslo stavu

Dokumentace pro tuto třídu byla generována z následujících souborů:

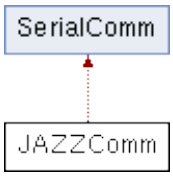
122itedcomm.h
123itedcomm.cpp

Dokumentace třídy JAZZComm

Komunikace s jednou jazzovskou kartou po rozhraní CAP.

`#include <jazzcomm.h>`

Diagram dědičnosti pro třídu JAZZComm



Veřejné metody

124 bool **Download** (const char *filename)

Spuštění downloadu programu do karty.

125 void **WaitForFinishedDownload** ()

Vyčkání na dokončení downloadu.

126 bool **IsDownloadedSuccessfully** ()

Kontrola správného dokončení downloadu.

127 bool **Command** (const char *command, std::string &response)

Odeslání příkazu a převzetí odpovědi.

128 **JAZZComm** (const char ***port**)

Vyžaduje zadání sériového portu, který má objekt obsluhovat.

129 **JAZZComm** (const **JAZZComm** &source)

Kopírovací konstruktor.

Privátní metody

130 bool **formFrameAndSend** (const char *command)

Sestaví rámec CAP a pošle zadaný příkaz.

131 bool **receiveAndParseFrame** (std::string &response)

Přijme rámec CAP, zkontroluje ho a vytáhne text odpovědi.

132 bool **openPort** ()

Otevření a konfigurace sériového portu.

Privátní atributy

133 QProcess **downloader**

Objekt spuštěného procesu downloaderu.

Statické privátní atributy

134 static const **UI_P maxResponseLength** = 2048

Maximální délka odpovědi Property Console na poslaný příkaz.

135 static const **UI_P maxReceiveAttempts** = 2048

Maximální počet pokusů o přijetí rámce ze zařízení.

136 static const char **downloaderExecutable** []

Cesta k programu, který se má spustit při volání Download.

Detailní popis

Komunikace s jednou jazzovskou kartou po rozhraní CAP.

Komunikace na rozhraní CAP se dělí do dvou hlavních částí: download programu a komunikace s běžícím programem. Třída **JAZZComm** realizuje oba tyto druhy komunikace a stará se i o přidělování sériového portu jednomu nebo druhému způsobu. Veškerá komunikace během testu probíhající přes sériový port CAP by měla jít přes ni.

Dokumentace konstruktoru a destruktoru

JAZZComm::JAZZComm (const char * *port*)

Vyžaduje zadání sériového portu, který má objekt obsluhovat.

Tento port by měl být od vytvoření objektu obsluhován pouze jím.

Parametry:

port	název portu v OS

JAZZComm::JAZZComm (const JAZZComm & *source*)

Kopírovací konstruktor.

Pozor:

Není ošetřena obsluha téhož portu oběma objekty!

Dokumentace k metodám

bool JAZZComm::Command (const char * *command*, std::string & *response*)

Odeslání příkazu a převzetí odpovědi.

Funkce odešle kartě příkaz a předá zpět výsledek.

Vracené hodnoty:

true	Příkaz se podařilo vykonat.
false	Příkaz se NEpodařilo vykonat.

Parametry:

command	příkaz Property Console, např. "get *"
response	odpověď karty, pokud byla načtena

bool JAZZComm::Download (const char * *filename*)

Spuštění downloadu programu do karty.

Funkce provede pokus o download programu po rozhraní CAP. **Plánované úpravy:**

Implementovat, až bude k dispozici konzolový CapFlash.

Poznámka:

Je nutné zajistit, aby karta přešla po zavolání do stavu, kdy přijme posílaný program (zajistí se ITEDem – voláním **ITEDComm**).

Vracené hodnoty:

true	Nahrávání programu se podařilo zahájit.
false	Nahrávání programu se NEpodařilo zahájit.

Parametry:

filename	Určení souboru pro download podle konvence OS.

bool JAZZComm::formFrameAndSend (const char * *command*) `[private]`

Sestaví rámec CAP a pošle zadaný příkaz.

Parametry:

příkaz	Property Console

Vracené hodnoty:

true	Příkaz se podařilo odeslat.
false	Příkaz se NEpodařilo odeslat.

bool JAZZComm::IsDownloadedSuccessfully ()

Kontrola správného dokončení downloadu.

Po dokončení downloadu (výsledek IsDownloadInProgress() se změní z true na false) je možné zkontrolovat, zda download proběhl úspěšně.

Vracené hodnoty:

true	Download proběhl úspěšně.

bool JAZZComm::openPort () `[private]`

Otevření a konfigurace sériového portu.

Vracené hodnoty:

true	Port je připraven k použití.
false	Port nyní nelze používat.

bool JAZZComm::receiveAndParseFrame (std::string & *response*) [private]

Přijme rámec CAP, zkontroluje ho a vytáhne text odpovědi.

Parametry:

response	Odpověď karty, pokud byla načtena.

Vracené hodnoty:

true	Odpověď se podařilo přijmout.
false	Odpověď se NEpodařilo přijmout.

Plánované úpravy:

Tady by se mělo zkontrolovat CRC, zatím bereme všechny rámce ohraničené určenými znaky jako platné.

void JAZZComm::WaitForFinishedDownload ()

Vyčkání na dokončení downloadu.

Uspí program (neplýtvá výkonem procesoru), dokud download nedoběhne.

Dokumentace k datovým členům

const char JAZZComm::downloaderExecutable[] [static], [private]

Cesta k programu, který se má spustit při volání Download.

Plánované úpravy:

Načíst název programu z konfiguračního souboru XML.

const UI_P JAZZComm::maxReceiveAttempts = 2048 [static], [private]

Maximální počet pokusů o přijetí rámce ze zařízení.

Je lepší, aby jich bylo spíš více než méně, protože se může stát, že bude potřeba vytáhnout z bufferu celou předchozí odpověď, kterou si někdo před námi nevyzvedl. Ale nějaké omezení tu být musí, aby se program nezasekl v nekonečné smyčce.

const UI_P JAZZComm::maxResponseLength = 2048 [static], [private]

Maximální délka odpovědi Property Console na poslaný příkaz.

Použije se pro určení velikosti bufferu.

Dokumentace pro tuto třídu byla generována z následujících souborů:

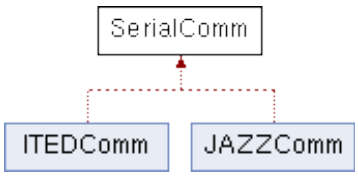
137jazzcomm.h
138jazzcomm.cpp

Dokumentace třídy SerialComm

Komunikace po sériovém rozhraní.

```
#include <serialcomm.h>
```

Diagram dědičnosti pro třídu SerialComm



Veřejné metody

139 unsigned long **Send** (const char *data, int n)
vysílání dat
140 unsigned long **Send** (const unsigned char *data, int n)
vysílání dat - bezznaménková verze
141 unsigned long **Receive** (char *data, int n)
příjem dat
142 unsigned long **Receive** (unsigned char *data, int n)
příjem dat - bezznaménková verze
143 bool **SetConfigWin** (LPCOMMCONFIG lpCC)
nastavení komunikačního rozhraní
144 bool **GetConfigWin** (LPCOMMCONFIG lpCC)
čtení konfigurace rozhraní
145 int **SetTimeoutWin** (unsigned long interval, unsigned long multipl, unsigned long konst)
nastavení timeoutu sériového portu
146 bool **Open** ()
Otevření portu.
147 bool **Close** ()
Zavření portu.
148 **SerialComm** (const char *_port, bool open=true)
Vyžaduje zadání portu, který má objekt obsluhovat.
149 **~SerialComm** ()
Pokud je port otevřený, zavře ho.

Chráněné atributy

150 std::string **port**
název portu, který objekt obsluhuje

Privátní atributy

151 HANDLE **handle**
systémový handler otevřeného portu
152 volatile bool **port_open**
je port právě otevřený?

Detailní popis

Komunikace po sériovém rozhraní.

Třída je určena pro obsluhu jednoho sériového portu. Pokud je nutné pracovat s více sériovými porty, je nutné vytvořit pro každý z nich jeden objekt třídy **SerialComm**.

Dokumentace konstruktoru a destruktoru

SerialComm::SerialComm (const char * *_port*, bool *open* = `true`)

Vyžaduje zadání portu, který má objekt obsluhovat.

Parametry:

port	název portu v OS
open	Určuje, zda se má port rovnou otevřít.

Dokumentace k metodám

bool SerialComm::Close ()

Zavření portu.

Pokud je port otevřený, zavře ho. Pokud je port již zavřený, vrátí true.

Návratová hodnota:

true, pokud je nyní port zavřený (nově nebo od dříve)

bool SerialComm::GetConfigWin (LPCOMMCONFIG *lpCC*)

čtení konfigurace rozhraní

pomocí windowsovské konfigurační třídy

Parametry:

lpCC	načtená konfigurace

Návratová hodnota:

true, pokud funkce proběhla správně

bool SerialComm::Open ()

Otevření portu.

Pokud je port zavřený, otevře ho.

Návratová hodnota:

true, pokud se port podařilo otevřít

unsigned long SerialComm::Receive (char * *data*, int *n*)

příjem dat

Uloží do pole přijatá data, která jsou již v bufferu OS.

Parametry:

data	buffer pro uložení dat
n	velikost bufferu v bajtech

Návratová hodnota:

počet přijatých bajtů

unsigned long SerialComm::Receive (unsigned char * *data*, int *n*) `[inline]`

příjem dat - bezznaménková verze

Uloží do pole přijatá data, která jsou již v bufferu OS.

Parametry:

data	buffer pro uložení dat
n	velikost bufferu v bajtech

Návratová hodnota:
počet přijatých bajtů

unsigned long SerialComm::Send (const char * *data*, int *n*)

vysílání dat

Parametry:

data	data k odeslání
n	délka dat v bajtech

Návratová hodnota:
počet odeslaných bajtů

unsigned long SerialComm::Send (const unsigned char * *data*, int *n*) `[inline]`

vysílání dat - bezznaménková verze

Parametry:

data	data k odeslání
n	délka dat v bajtech

Návratová hodnota:
počet odeslaných bajtů

bool SerialComm::SetConfigWin (LPCOMMCONFIG *lpCC*)

nastavení komunikačního rozhraní

pomocí windowsovské konfigurační třídy

Parametry:

lpCC	požadovaná konfigurace

Návratová hodnota:
true, pokud funkce proběhla správně

int SerialComm::SetTimeoutWin (unsigned long *interval*, unsigned long *multipl*, unsigned long *konst*)

nastavení timeoutu sériového portu

podle vzorce (multipl*pocet_bytu)+konst, pocet_bytu=14 Časy jsou v milisekundách.

Dokumentace pro tuto třídu byla generována z následujících souborů:

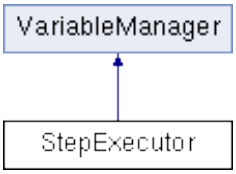
153serialcomm.h
154serialcomm.cpp

Dokumentace třídy StepExecutor

Vykonavatel kroku testu.

`#include <stepexecutor.h>`

Diagram dědičnosti pro třídu StepExecutor



Veřejné metody

155 bool **ExecuteStep** (const QDomNode &_step)
Zajistí vykonání kroku testovacího scénáře ze zadaného souboru.
156 **TestVariable** * **GetVar** (const std::string &_name)
*Získání proměnné z lokálního nebo globálního objektu **VariableManager**.*
157 void **SetFailed** (const QString &_message)
Označí, že krok selhal.
158 void **SetOK** ()
Označí, že krok skončil úspěšně.
159 **StepExecutor** (**VariableManager** *const _globVarMan, **CardManager** *const _cardMan)
*Konstruktor pro použití objektem **TestExecutor**.*

Privátní atributy

160 **VariableManager** *const **globalVarMan**
správce globálních proměnných
161 **CardManager** *const **cardMan**
správce karet
162 bool **stepOK**
Informace, zda krok proběhl úspěšně.
163 bool **stepFailed**
Informace, zda krok selhal.
164 QString **failMessage**
Zpráva o prvním selhání, ke kterému v kroku došlo.

Detailní popis

Vykonavatel kroku testu.

Zajišťuje volání jednotlivých příkazů kroku testu a poskytuje jim proměnné z globálního objektu **VariableManager** celého testu i lokální proměnné kroku, které **StepExecutor** spravuje.

Dokumentace konstruktoru a destruktoru

StepExecutor::StepExecutor (VariableManager *const *_globVarMan*, CardManager *const *_cardMan*)

Konstruktor pro použití objektem **TestExecutor**.

Objekt **StepExecutor** musí mít přístup k objektům, které jsou součástí zastřešujícího objektu **TestExecutor**.

Parametry:

_globVarMan	správce globálních proměnných
_cardMan	správce karet testu

Dokumentace k metodám

bool StepExecutor::ExecuteStep (const QDomNode & *_step*)

Zajistí vykonání kroku testovacího scénáře ze zadaného souboru.

Vracené hodnoty:

true	Krok testu proběhl úspěšně.

Parametry:

_step	uzel <step> souboru testu

TestVariable * StepExecutor::GetVar (const std::string & *_name*) `[virtual]`

Získání proměnné z lokálního nebo globálního objektu **VariableManager**.

Pokusí se načíst proměnnou z lokálního objektu **VariableManager** tohoto kroku. Pokud neexistuje, hledá v globálním objektu **VariableManager** celého testu. To znamená, že globální proměnné jsou kryty lokálními, což by mělo být očekávatelné chování.

Návratová hodnota:

Ukazatel na nalezenou proměnnou.

Vracené hodnoty:

NULL	Proměnná zadaného jména neexistuje.

Parametry:

_name	název hledané proměnné

Reimplementuje stejnojmenný prvek z **VariableManager** (*s.Chyba: zdroj odkazu nenalezen*).

void StepExecutor::SetFailed (const QString & *_message*)

Označí, že krok selhal.

Určeno pro volání objektem **CommandExecutor**.

Parametry:

_message	Zpráva s popisem selhání.

void StepExecutor::SetOK ()

Označí, že krok skončil úspěšně.

Určeno pro volání objektem **CommandExecutor**.

Dokumentace pro tuto třídu byla generována z následujících souborů:

165stepexecutor.h
166stepexecutor.cpp

Dokumentace třídy TestExecutor

Vykonavatel testu.

`#include <testexecutor.h>`

Veřejné metody

167 bool **ExecuteTest** (const char *_filename)
Zajistí vykonání celého testovacího scénáře ze zadaného souboru.
168 **TestExecutor** (**ITEDComm** *const _ITEDComm)
Konstruktor testu s ITEDCommem.

Privátní metody

169 bool **loadFile** (const char *_filename)
Načtení souboru s testem.
170 bool **checkITEDVersion** ()
Kontrola verze ITEDu.
171 bool **checkRequirements** ()
Kontrola požadavků testu.
172 bool **checkCard** (QDomElement &_cardElement)
Kontrola karty.
173 bool **checkConnection** (QDomElement &_connElement, **Card** *const _card)
Kontrola rozhraní karty.
174 bool **prepareTest** ()
Příprava testu.

Privátní atributy

175 **ITEDComm** *const **testITEDComm**
*ukazatel na **ITEDComm**, který má test používat*
176 **CardManager cardMan**
správce karet používaných v testu
177 QDomDocument **testDOM**
předpis testu ve formátu XML DOM

Detailní popis

Vykonavatel testu.

Objekt této třídy má na starosti celé vykonání testu. V hlavním programu by se tedy měla maximálně načíst konfigurace, předat objektu **TestExecutor** prostředí (ITED, karty) a zavolat **TestExecutor::ExecuteTest()**.

Dokumentace konstruktoru a destruktoru

TestExecutor::TestExecutor (ITEDComm *const *_ITEDComm*)

Konstruktor testu s ITEDCommem.

Založí objekt testu, jenž má po celou dobu k dispozici spojení s ITEDem.

Parametry:

_ITEDComm	Ukazatel na objekt **ITEDComm**, který má test používat.

Dokumentace k metodám

bool TestExecutor::checkCard (QDomElement & *_cardElement*) [private]

Kontrola karty.

Zkontroluje, zda je karta uvedená v souboru testu skutečně dostupná.

Vracené hodnoty:

true	Požadavky testu jsou splněny.

Parametry:

_cardElement	element XML s kartou (v requirements)

bool TestExecutor::checkConnection (QDomElement & *_connElement*, Card *const *_card*) [private]

Kontrola rozhraní karty.

Zkontroluje, zda je rozhraní karty uvedené v souboru testu skutečně dostupné.

Vracené hodnoty:

true	Požadavky testu jsou splněny.

Parametry:

_connElement	element XML s rozhraním
_card	karta, které rozhraní přísluší

bool TestExecutor::checkITEDVersion () [private]

Kontrola verze ITEDu.

Zkontroluje, zda verze ITEDu splňuje požadavky tohoto programu. Současná implementace přijme verzi vyšší nebo rovnou požadované.

Vracené hodnoty:

true	Verze ITEDu je pro tento program v pořádku.

bool TestExecutor::checkRequirements () [private]

Kontrola požadavků testu.

Zkontroluje, zda dostupné prostředky (ITED, karty) odpovídají požadavkům uvedeným v souboru s testem.

Vracené hodnoty:

true	Požadavky testu jsou splněny.

bool TestExecutor::ExecuteTest (const char * *_filename*)

Zajistí vykonání celého testovacího scénáře ze zadaného souboru.

Parametry:

_filename	Název souboru s testem.

Vracené hodnoty:

true	Test proběhl úspěšně.

bool TestExecutor::loadFile (const char * *_filename*) [private]

Načtení souboru s testem.

Parametry:

_filename	Název souboru s testem.

Vracené hodnoty:

true	Soubor byl úspěšně načten pro zpracování.

bool TestExecutor::prepareTest () `[private]`

Příprava testu.

Zajistí download testované verze softwaru do testovaných karet.

Vracené hodnoty:

true	Test se podařilo připravit.

Dokumentace pro tuto třídu byla generována z následujících souborů:

178testexecutor.h
179testexecutor.cpp

Dokumentace třídy TestVariable

Proměnná testovacího scénáře.

`#include <testvariable.h>`

Třídy

180 union **ValUnion**

Veřejné typy

181enum **VarType** { **VT_INTEGER**, **VT_INTEGER_ARRAY** }

datový typ uložené proměnné Veřejné metody

182 **VarType GetType** ()
Načtení typu proměnné.
183 **SI_P GetInt** (**SI_P** _index=0)
Načtení hodnoty proměnné typu VT_INTEGER(_ARRAY).
184 bool **SetInt** (**SI_P** _value, **SI_P** _index=0)
Uložení hodnoty proměnné typu VT_INTEGER(_ARRAY).
185 **TestVariable** (**VarType** _type, **UI_P** _count=0)
Konstruktor se zadáním typu vytvářené proměnné.
186 **TestVariable** (const **TestVariable** &_source)
Kopírovací, nebo spíš přesouvací konstruktor.
187 **~TestVariable** ()
Destruktor proměnné.

Privátní atributy

188 **VarType type**
189 union **TestVariable::ValUnion value**

Detailní popis

Proměnná testovacího scénáře.

Tato třída zajišťuje obsah proměnné, která může být různého typu. Vytvoření, držení a dostupnost podle jména zajistí **VariableManager**.

Proměnné typu pole (_ARRAY) jsou indexovány od 0 podle konvence jazyků založených na C, ale interně je v 0. prvku uložena velikost pole (kontrolovaná při přístupu do pole) a ke všem indexům se při přístupu přičítá 1.

Dokumentace k členským výčtům

enum TestVariable::VarType

datový typ uložené proměnné

Hodnoty výčtu:

VT_INTEGER celé číslo (se znaménkem)

VT_INTEGER_ARRAY pole celých čísel (se znaménkem)

Dokumentace konstruktoru a destruktoru

TestVariable::TestVariable (VarType *_type*, UI_P *_count* = 0)

Konstruktor se zadáním typu vytvářené proměnné.

Parametry:

_type	typ proměnné
_count	počet prvků pole (jde-li o pole)

TestVariable::TestVariable (const TestVariable & *_source*)

Kopírovací, nebo spíš přesouvací konstruktor.

Pokud je proměnná pole, alokuje nové pole a překopíruje jeho obsah. Kopírovací konstruktor je nutný pro správnou funkci vkládání do mapy.

TestVariable::~TestVariable ()

Destruktor proměnné.

Pokud je proměnná pole, je nutné ho dealokovat.

Dokumentace k metodám

SI_P TestVariable::GetInt (SI_P *_index* = 0)

Načtení hodnoty proměnné typu VT_INTEGER(_ARRAY).

Parametry:

_index	Pokud je proměnná pole, je nutné určit index.

Návratová hodnota:

Hodnota proměnné typu VT_INTEGER(_ARRAY).

Vracené hodnoty:

MAX_SI_P	V případě chyby (proměnná neexistuje nebo má jiný typ).

TestVariable::VarType TestVariable::GetType ()

Načtení typu proměnné.

Návratová hodnota:

Typ proměnné

bool TestVariable::SetInt (SI_P *_value*, SI_P *_index* = 0)

Uložení hodnoty proměnné typu VT_INTEGER(_ARRAY).

Parametry:

_value	Hodnota proměnné typu VT_INTEGER(_ARRAY) k uložení.
_index	Pokud je proměnná pole, je nutné určit index.

Vracené hodnoty:

true	Pokud se podařilo hodnotu zapsat.

Dokumentace pro tuto třídu byla generována z následujících souborů:

190testvariable.h
191testvariable.cpp

Dokumentace unie TestVariable::ValUnion

Veřejné atributy

192 **SI_P integer**
193 **SI_P * integer_array**

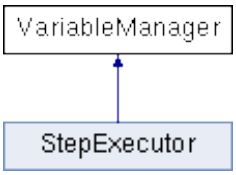
Dokumentace pro tuto unii (union) byla generována z následujícího souboru:
194 testvariable.h

Dokumentace třídy VariableManager

Správce proměnných testovacího scénáře.

`#include <variablemanager.h>`

Diagram dědičnosti pro třídu VariableManager



Veřejné metody

195 **TestVariable * NewVar** (std::string _name, **TestVariable::VarType** _type, **UI_P** _count=0)
Vytvoření nové proměnné.
196 virtual **TestVariable * GetVar** (const std::string &_name)
Získání proměnné.
197 **UI_P LoadVariables** (const QDomNode &_variables)
Načtení proměnných z XML souboru testu.

Privátní typy

198 typedef std::map< std::string,
199 **TestVariable** > **varRecs_t**

Privátní atributy

200 varRecs_t **varRecords**

Detailní popis

Správce proměnných testovacího scénáře.

Spravuje proměnné potřebné v testovacím scénáři. Zajišťuje jejich vytváření a rušení a poskytuje k nim přístup podle jména.

Dokumentace k metodám

TestVariable * VariableManager::GetVar (const std::string & *_name*) `[virtual]`

Získání proměnné.

Najde proměnnou podle jména a vrátí ji.

Návratová hodnota:

Ukazatel na nalezenou proměnnou.

Vracené hodnoty:

NULL	Proměnná zadaného jména neexistuje.

Parametry:

_name	název hledané proměnné

Reimplementováno v **StepExecutor** (*s.Chyba: zdroj odkazu nenalezen*).

UI_P VariableManager::LoadVariables (const QDomNode & *_variables*)

Načtení proměnných z XML souboru testu.

Parametry:

_variables	Uzel <variables> souboru testu.

Návratová hodnota:

Počet načtených proměnných.

Vracené hodnoty:

0	Nenačetla se žádná proměnná - mohlo dojít k chybě.

TestVariable * VariableManager::NewVar (std::string *_name*, TestVariable::VarType *_type*, UI_P *_count* = 0)

Vytvoření nové proměnné.

Zaregistruje novou proměnnou a přiřadí jí typ.

Vracené hodnoty:

NULL	Proměnnou se nepodařilo vytvořit (např. již nějaká toho jména existuje).

Parametry:

_name	název proměnné
_type	datový typ proměnné
_count	počet prvků pole (jde-li o pole)

Dokumentace pro tuto třídu byla generována z následujících souborů:

201variablemanager.h
202variablemanager.cpp

Dokumentace souborů

Dokumentace souboru delay.h

Pozastavení programu na zadaný čas.

```
#include "defs.h"
```

Funkce

203 void **Delay** (**UI_P** secs)
Pozastavení programu na zadaný čas.

Detailní popis

Pozastavení programu na zadaný čas.

Datum:
2013-03-25

Autor:
MiM (Miroslav Matějů)

Dokumentace funkcí

void Delay (UI_P *secs*)

Pozastavení programu na zadaný čas.

Zastaví program ve spinlocku na zadanou dobu.

Parametry:

secs	čas pozastavení v sekundách

Dokumentace souboru main.cpp

Hlavní část programu.

```
#include <iostream>
#include "defs.h"
#include "itedcomm.h"
#include "testexecutor.h"
```

Definice maker

204 #define **ITEDCOMM_PORT** (argv[2])
205 #define **XML_FILENAME** (argv[1])

Funkce

206 int **main** (int argc, char *argv[])

Detailní popis

Hlavní část programu.

Datum:
2013-05-09

Autor:
MiM

Zajišťuje načtení a kontrolu parametrů předaných z příkazové řádky, vytváří objekty typu **ITEDComm** a **TestExecutor**, spouští test a hlásí výsledky.

Požadované parametry programu: predpis_testu ITED_port