České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky

DIPLOMOVÁ PRÁCE

Server analyzátoru protokolu Profibus DP

Praha, 2008 **Martin Samek**

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 zákona č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 20. května 2008

podpis

Poděkování

Mé poděkování patří především vedoucímu diplomové práce Ing. Pavlu Burgetovi za jeho ochotu, cenné nápady a podnětné připomínky. Dále bych mu velice rád poděkoval za jeho úsilí, které věnuje organizaci projektu CEPOT na Katedře řídicí techniky. Děkuji také Ing. Petru Smolíkovi, Ing. Pavlu Píšovi a Ing. Janu Faiglovi za jejich konzultace, bez kterých by tato práce mohla jen těžko vzniknout.

Poděkování patří také mým rodičům za to, že mi poskytli rodinné zázemí ke studiu a všemožně mě podporovali. Děkuji mojí přítelkyni Marcele za trpělivost, morální podporu a motivaci při vytváření této diplomové práce.

Abstrakt

Tato práce si klade za cíl vytvořit softwarový nástroj, který usnadní záznam, zpracování a analýzu dat přenášených po průmyslové komunikační síti Profibus DP. Práce se zabývá implementací TCP/IP serveru v operačním systému Linux. Server bude sloužit jako zprostředkovatel a zároveň předzpracovatel zachycených dat pro grafickou uživatelskou aplikaci, která je vyvíjena odděleně od této práce.

Toto řešení je založeno na využití běžně dostupného hardwaru a klade si za cíl být alternativou ke komerčním produktům, které jsou založeny na proprietárním hardwarovém vybavení a jejichž pořizovací náklady jsou velmi vysoké.

Práce využívá již existující ovladač rozšiřující komunikační karty, která umožňuje na počítači zachytávat komunikaci šířenou po sběrnici Profibus DP. Server tvoří univerzální rozhraní pro přístup ke sběrnici prostřednictvím IP sítě. Předpokládané nasazení zahrnuje použití serveru pro dlouhodobé zaznamenávání komunikace a následné offline vyhodnocení dat v klientské aplikaci. Ve školním prostředí lze využít server pro zpřístupnění jedné sběrnice více studentům současně.

Abstract

The subject of this master thesis is aimed at creation of a software tool, which will be helpful to capture, postprocess and analyse data from the Profibus DP fieldbus. This thesis discusses an implementation of the TCP/IP server under the Linux operating system. The server is intended to act as an intermediary between the Profibus DP fieldbus and a GUI application, which is being developed separately of this thesis.

This solution assumes taking advantage of commonly used hardware and it is an alternative to the commercial products. Such products are usually based on the proprietary hardware equipment, purchase costs of which are very high.

This solution is based on an existing driver for communication PCI card, which makes possible to capture data transmitted over the Profibus DP fieldbus. The server makes up an universal interface for accessing to the fieldbus through an IP network. The main usage of this server is a long-time period data capturing for later analysis in the client application. This application also finds its usage in the school environment, where more students can use one fieldbus simultaneously.

České vysoké učení technické v Praze - Fakulta elektrotechnická

Katedra řídicí techniky **Školní rok:** 2006/2007

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Martin Samek

Obor: Technická kybernetika

Název tématu: Server analyzátoru protokolu Profibus DP

Zásady pro vypracování:

Server realizujte pod operačním systémem GNU Linux, kde je k dispozici driver pro RS485 kartu. Využijte tento driver pro sběr dat komunikačního protokolu Profibus DP.

1. Navrhněte a realizujte mechanismus přijímání zpráv od komunikačního driveru.
2. Navrhněte a realizujte komunikační protokol nad TCP/IP, který bude sloužit k výměně dat a stavových i řídicích informací mezi klientem a serverem.
3. Navrhněte a realizujte filtrování zpráv, jejich ukládání na lokálním záznamovém médiu. Implementujte ukládání zpráv na žádost, tj. při splnění určité spouštěcí podmínky.
4. Zprávy přijímejte všechny, nebo aplikujte předvolený filtr.

Při implementaci serveru uvažujte, že server může v budoucnu sloužit také jako stanice typu master (např. Master Class 2) současně s funkcí analyzátoru.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí diplomové práce: Ing. Pavel Burget

Termín zadání diplomové práce: zimní semestr 2006/2007 (změna zadání: 1.10.2007)

Termín odevzdání diplomové práce: leden 2008

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

L.S.

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 17.10.2007

Obsah

1 Úvod **1**

1.1 Proč potřebujeme analyzovat komunikaci? 1

1.2 Přehled vývoje analyzátoru 2

1.3 Proč server a klient? 3

1.4 Stručný obsah kapitol 3

2 Principy sběrnice Profibus DP **5**

2.1 Fyzická vrstva 6

2.1.1 Propojení fyzické vrstvy s PC 8

2.2 Linková vrstva (FDL) 11

2.2.1 Výměna dat na sběrnici 11

2.2.2 Formát dat 14

2.2.3 Rozhraní FDL vrstvy 16

2.3 Aplikační vrstva a management sběrnice 17

3 Metody analýzy sběrnice Profibus **18**

3.1 Úrovně přístupu a hloubka analýzy 18

3.1.1 Subbitová úroveň 19

3.1.2 Bajtová úroveň 20

3.1.3 Úroveň zpráv 22

4 Implementace serveru v prostředí Linuxu **23**

4.1 Vývojové nástroje 24

4.2 Struktura aplikace 25

4.3 Program pracující jako démon 26

4.4 Zpracování konfigurace serveru 31

4.5 Činnost serveru a jeho stavový automat 34

4.6 Driver PBMaster 35
4.6.1 Formát dat 37
4.6.2 Nastavení driveru 37

5 Server/klient komunikace 40
5.1 Komunikační kanál 40
5.2 Realizace komunikačního protokolu 45
5.2.1 Metajazyk XML 46
5.2.2 Knihovna Expat 47
5.2.3 Komunikační protokol 49
5.2.4 Sémantika komunikačního protokolu 50
5.2.5 Vizualizace činnosti komunikačního protokolu 53

6 Předzpracování dat na straně serveru 55
6.1 Formát dat vhodný pro přenos soketem 55
6.2 Kódování Base64 56
6.3 Lokální ukládání dat 60
6.4 Filtrace dat na straně serveru 61
6.5 Detekce uživatelsky zadaných událostí 64

7 Závěr 66
7.1 Co by se dalo zdokonalit? 67

A Překlad a instalace ovladače PBMaster I

B Konfigurovatelné parametry serveru IV

C Obsah přiloženého CD V

D Experimentální sběrnice VI

Seznam obrázků VIII

Seznam tabulek IX

Seznam použitých zkratek X

Literatura XI

Kapitola 1

Úvod

Se stále se zvětšující složitostí průmyslových procesů rostou také výkonnostní a kvalitativní požadavky na techniku zajišťující řízení procesu a jeho automatizaci. V dnešní době *informačních technologií* si jen velice těžko dovedeme představit průmyslový proces, bez ohledu na jeho fyzikální podstatu, který by se obešel bez sofistikovaného řídicího systému. V případě, že velikost procesu a jeho prostorové uspořádání překročí určitou mez, je výhodnější s ohledem na kvalitativní a ekonomické požadavky upustit od řídicího systému klasické koncepce a využít možností distribuovaného řídicího systému.

Vzdálenosti mezi uzly distribuovaného řídicího systému se v praxi pohybují od jednotek metrů až po několik kilometrů. Je zřejmé, že takový systém se neobejde bez rychlé a vysoce spolehlivé digitální komunikační sítě, která zajistí přenos dat mezi jeho uzly v reálném čase. Za tímto účelem bylo vyvinuto velké množství sítí, které souhrnně označujeme *Fieldbus*. Často jsou tyto sítě také nazývány *sběrnice*. Mezi nejrozšířenější sítě v této oblasti bezpochyby patří sběrnice Profibus, jejíž varianta Profibus DP (*Distributed Peripherals*) je zejména v evropském průmyslu velmi rozšířená.

1.1 Proč potřebujeme analyzovat komunikaci?

V průmyslu se vyskytuje mnoho prostředí, ve kterých musíme věnovat pozornost působení nežádoucích vlivů (elektromagnetické rušení, klimatické podmínky, vibrace, prašnost, mechanické poškození, atd.) na řídicí systém a zároveň předejít poškození procesní technologie činností řídicího systému. Část nežádoucích vlivů dokážeme eliminovat kvalitním návrhem a volbou technického vybavení přizpůsobeného k činnosti v daném pro-

středí. Bohužel nedokážeme postihnout degradaci technického vybavení v průběhu času a zvláštní pozornost musíme také věnovat kompatibilitě výrobků od různých výrobců.

Pokud se dostaneme do situace, že distribuovaný řídicí systém nefunguje, tak musíme lokalizovat vadný uzel sítě nebo vadné komunikační médium. Tyto poruchy se mohou vyskytnout jako důsledek jmenovaných nežádoucích vlivů. Abychom byli schopni zajistit opatření vedoucí k nápravě, potřebujeme pomocí analýzy záznamů datové výměny na sběrnici identifikovat zdroj poruchy. Cílem této práce proto bylo navrhnout a realizovat softwarový nástroj, který umožní zaznamenávat a vyhodnocovat datovou výměnu na sběrnici v dlouhodobém měřítku a přitom si vystačí s běžně dostupným hardwarem. Zaznamenávání komunikace produkuje ohromné množství dat, a proto bylo nutné implementovat pokročilé nástroje, které usnadní vyhodnocení záznamu. Jde především o filtry zpráv dle rozličných atributů a reakci na výskyt sledované události doplněnou o funkci *pretriggering*, která umožní zaznamenání dostatečného časového intervalu komunikace před výskytem sledované události.

1.2 Přehled vývoje analyzátoru

Profibus DP využívá jako fyzickou vrstvu metalické vedení dle specifikace RS-485, pro které existují relativně levné rozšiřující PCI karty do počítače. S velkými omezeními lze použít i běžné rozhraní RS-232 doplněné o jednoduchý převodník napěťových úrovní. Princip připojení PC k Profibusu DP je podrobně vysvětlen v oddíle 2.1.1 a 4.6.

Pro zmiňované PCI karty byly v minulosti vyvinuty ovladače ProfiM a PBMaster, které umožňují uživatelským aplikacím přistupovat ke sběrnici Profibus DP na úrovni jednotlivých znaků UART. Ovladač ProfiM pro platformu Win32 implementoval Pavel Trnka ve svojí diplomové práci [10]. Myšlenku Trnkovy práce převzal Tran Duy Khanh a jako svoji bakalářskou práci [9] naprogramoval ovladač PBMaster, což je v podstatě implementace ovladače ProfiM pro operační systémy Linux rozšířená o režim analyzátoru sběrnice. Původní ProfiM umožňoval pouze činnost v režimu stanice Master. Zdrojové kódy[1] jsou volně dostupné pod GPL licencí.

Ve své diplomové práci navazuji na Tranův ovladač PBMaster. Úkolem je vyvinout analyzátor Profibusu DP, realizovaný jako aplikace pro operační systém Linux. Tuto aplikaci budu dále nazývat PBServer. Jde o TCP/IP server, který dle požadavků připojené

[1] http://www.pbmaster.org

klientské aplikace čte data ze sběrnice Profibus DP prostřednictvím ovladače PBMaster, transformuje je do formátu vhodného pro přenos a odesílá klientovi skrze TCP spojení. Klientská aplikace PBView je vyvíjena Alešem Kautským v rámci jeho diplomové práce nazvané „Klient analyzátoru protokolu Profibus DP“. Tato práce si klade za cíl nalézt vhodné principy realizace, prověřit jejich vlastnosti včetně případných omezení a na základě těchto zkušeností vytvořit vhodnou koncepci pro budoucí vývoj aplikace. Program, který je výstupem této práce, slouží hlavně pro seznámení se s vlastnostmi použitých technologií; nejedná se o verzi určenou pro reálné nasazení.

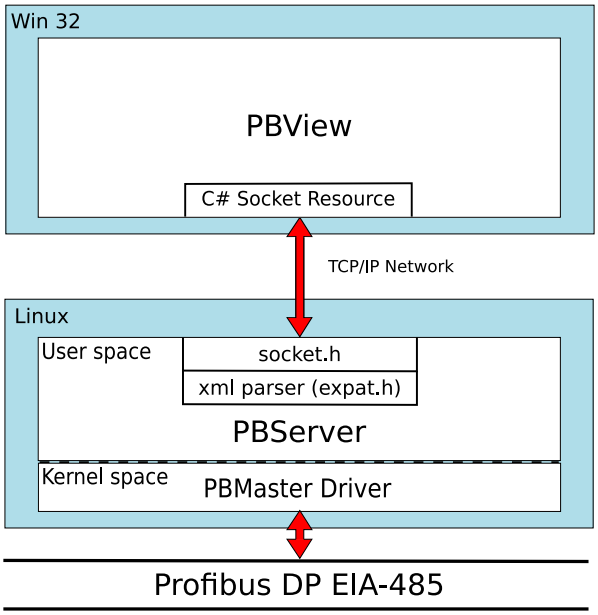
1.3 Proč server a klient?

Abychom mohli využít analyzátor ve skutečném průmyslovém prostředí, musí být jeho provedení z hlediska odolnosti a robustnosti srovnatelné se samotným Profibusem. Z toho plyne, že nemůžeme běžný kancelářský PC vybavit RS-485 rozhraním, nainstalovat software a umístit ve výrobní hale vedle svářecího robota včetně monitoru a klávesnice. Proto jsem se rozhodl použít architekturu server/klient, která umožní online nebo offline přenos dat do pohodlného prostředí kanceláře. Samotný server potom může být realizován jako relativně malý jednodeskový počítač vybavený rozhraním RS-485 a Ethernetem. Veškerý uživatelský komfort spočívá v klientské aplikaci realizované na platformě .NET, která získává zachycená data prostřednictvím serveru.

Koncepce ilustrovaná na obrázku 1.1 přináší oproti monolitickému řešení mnoho výhod. Server může provádět samostatně dlouhodobé měření a získaná data ukládat do lokální paměti. Na požadavek uživatele lze uložená data přenést do klientské aplikace za účelem dalšího zpracování a server využít k dalšímu úkolu. Offline kopie dat může sloužit k výuce nebo prezentaci. Samozřejmostí je online režim interaktivní vizualizace dat v klientské aplikaci.

1.4 Stručný obsah kapitol

Tato práce předpokládá určitou čtenářovu orientaci v problematice průmyslových sítí a programovaní v operačním systému Linux. Jsem si vědom, že v některých kapitolách pracuji s pojmy a principy, které jsem dostatečně nevysvětlil. Zde čtenáře odkáži na pří-



Obrázek 1.1: Blokové schéma celého analyzátoru

slušnou literaturu, kde jsou uvedena patřičná vysvětlení, jejichž výklad je nad rámec této práce. Základní vlastnosti a parametry Profibusu jsou popsány v kapitole 2. Programování serveru s využitím jazyka C a prostředí systému Linux je popsáno v kapitole 4.

Kapitola 2 stručně popisuje možnosti fyzického propojení sběrnice s PC za účelem sběru dat. Dále se zabývá linkovou vrstvou (FDL) a jejím významem. Krátce je také zmíněna aplikační vrstva Profibusu.

Kapitola 3 pojednává o metodách a přístupech, které můžeme využít k analýze komunikace na sběrnici, a porovnává je z hlediska náročnosti na technické vybavení a jeho cenu.

Kapitola 4 se zabývá samotným programováním v operačním systému Linux, využitím jeho služeb, přístupem k hardwaru, implementací serveru a použitím TCP/IP v naší aplikaci.

Kapitola 5 popisuje problematiku komunikace přes síť a implementaci komunikačního protokolu mezi serverem a klientem na bázi jazyka XML.

Kapitola 6 se zabývá pokročilými funkcemi, které jsou implementovány na severu a mají přispět hlavně k uživatelskému pohodlí. Jedná se zejména o uživatelské filtry, konfiguraci z GUI aplikace nebo pretriggering.

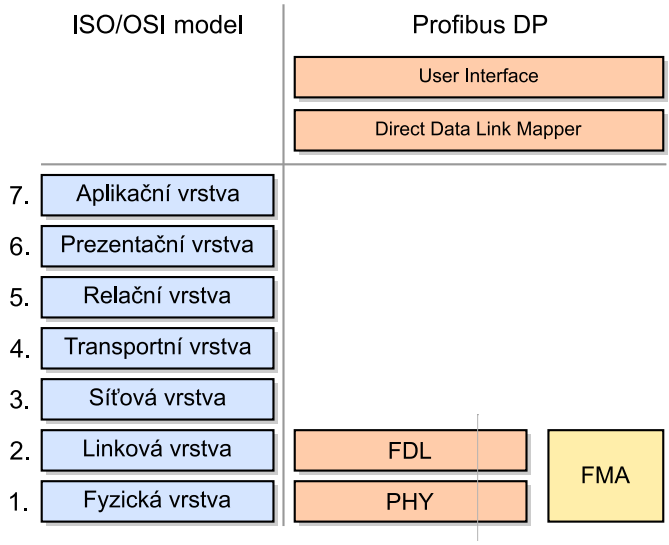
Kapitola 2

Principy sběrnice Profibus DP

Profibus DP je jedna z variant průmyslové sběrnice Profibus, která slouží převážně k propojení prvků automatizačních systémů (PLC, vstupy a výstupy, senzory, akční členy, operátorské panely, atd.) prostřednictvím rychlého sériového spojení. Nejčastěji Profibus DP slouží k rychlé cyklické výměně dat mezi relativně komplikovanou stanicí master a několika jednoduchými stanicemi slave. Architektura vrstev Profibusu DP je odvozena z referenčního ISO/OSI modelu. Celá rodina sítí Profibus používá jako linkovou vrstvu protokol FDL. Proto není vyloučeno použití PBServeru s jinou realizací fyzického media Profibusu. Podmínkou je existence síťového hardwaru pro PC a ovladače kompatibilního s operačním systémem Linux. Tato práce se zabývá pouze zpracováním dat z Profibusu DP provozovaného na fyzické vrstvě dle standardu RS-485. Vlastnosti fyzické vrstvy a služby, které poskytuje, jsou rozebrány v oddíle 2.1.

Profibus DP ve srovnání s ISO/OSI modelem implementuje pouze fyzickou a linkovou vrstvu a překvapivě nemá aplikační vrstvu (7. vrstva ISO/OSI). Důvodem pro tento přístup je zejména zachování celkové jednoduchosti protokolu. Z pohledu uživatele je absence aplikační vrstvy kompenzována tenkou vrstvou DDLM, která implementuje uživatelsky přívětivější rozhraní ke komunikačním primitivům vrstvy FDL. V této části pod pojmem uživatel rozumíme stanice v režimu DP-Master (PLC) nebo DP-Slave (vzdálené I/O), které používají komunikační služby Profibusu DP k vykonávání algoritmu řízení. Vrstva PHY poskytuje sdílené komunikační médium. Řízení přístupu a předávání oprávnění k vysílání po sdíleném médiu je plně v režii vrstvy FDL. Každá stanice musí mít před zařazením do sítě přiřazenu unikátní adresu. Dvouúrovňové řízení přístupu zajišťuje, že přístup ke sběrnici je deterministický a garantuje délku cyklu sběrnice. Vlastnosti vrstvy FDL jsou stručně popsány v oddíle 2.2. Podrobnější popis vrstvy lze najít v diplomové práci [10]. Úplná dokumentace Profibusu je součástí normy EN 50 170 Volume 2 [7].

Souvislost mezi vrstvami Profibusu DP a ISO/OSI referenčním modelem je zřejmá z následujícího obrázku 2.1.



Obrázek 2.1: Profibus DP a ISO/OSI model

2.1 Fyzická vrstva

Fyzická vrstva definuje požadavky na vlastnosti, které musí splňovat komunikační kanál. V průmyslové automatizaci Profibus DP standardně využívá jako komunikační medium vedení, které splňuje průmyslový standard RS-485 - high speed. Pro použití ve specifických prostředích doporučuje norma [7] použití optického vlákna, které má mnohem lepší parametry z hlediska elektromagnetické kompatibility. V procesní automatizaci se používá Profibus PA, jehož fyzická vrstva pracuje s mediem dle standardu IEC 1158-2.

Propojení mezi stanicemi (dle RS-485) je vedeno kroucenou stíněnou dvoulinkou, jejíž vodiče tvoří diferenční vedení a označují se RxD/TxD-P a RxD/TxD-N. Barevné značení vodičů je uvedeno v tabulce 2.2. Vedení musí být na obou koncích přizpůsobeno terminátorem. Standardně se používá přizpůsobení s impedancí 120 Ω. Pro rychlosti menší než 1500 kbps připouští norma i kabeláž bez terminátorů.

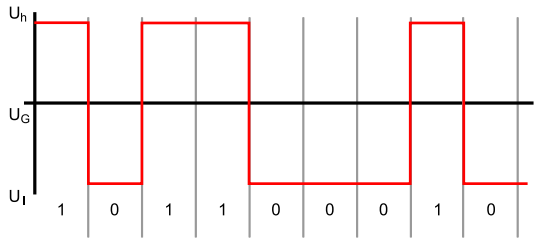
Základní organizací sítě je sběrnicová topologie, kterou lze s využitím opakovačů a odbočených větví transformovat na topologii stromovou. Délku sběrnice lze prodloužit za pomoci opakovačů. Jeden segment sítě může obsahovat až 32 jednotek s tím, že opakovač

je také považován za stanici. Počet jednotek v síti je omezen počtem adres, kterých je k dispozici 127 ($0 \div 126$, 127 je adresa oběžníku). Profibus DP umožňuje přenosové rychlosti v rozsahu $9,6$ kbps až 12 Mbps. Pro přenosové rychlosti menší než $93,75$ kbps může délka sběrnice bez opakovačů dosáhnout maximálně 1200 metrů (viz tabulka 2.1). Přesné hodnoty rychlostí a vzdáleností, stejně jako zapojení konektorů, terminátorů a mechanické vlastnosti jsou podrobně uvedeny v normě [7].

Rychlost v kbps	9,6	19,2	93,75	187,5	500	1500	12000
Délka v m	1200	1200	1200	1000	400	200	100

Tabulka 2.1: Maximální délka sběrnice (bez opakovačů) v závislosti na rychlosti

Fyzická vrstva na bázi RS-485 používá kanálové kódování NRZ (Non Return to Zero). Logická „1“ je reprezentována jako kladné konstantní diferenciální napětí mezi vodiči RxD/TxD-P a RxD/TxD-N. Naopak logickou „0“ představuje záporná konstantní úroveň diferenciálního napětí.



Obrázek 2.2: Princip kódování Non Return to Zero

Po dobu, kdy stanice nevysílá, musí být vysílač této stanice neaktivní a ve stavu vysoké impedance. Nejsou-li na sběrnici vysílána data a vysílací obvody všech stanic se nalézají ve stavu vysoké impedance, je vlivem impedančního přízpůsobení na sběrnici napěťová úroveň odpovídající log.1. Oproti tomu přijímací obvody zůstávají aktivní neustále a v klidovém intervalu přijímají log.1. Tato vlastnost je používána k synchronizaci. Parametr T_{SYN} udává minimální časový interval (od ukončení posledního přenosu), po který musí být sběrnice v klidové úrovni, tzn. po tuto dobu všechny stanice přijímají úroveň log.1. Žádná stanice nesmí začít vysílat dříve než po uplynutí tohoto času. Standardně platí, že $T_{SYN} = 33t_{BIT}$.

Rozhraní mezi vrstvou PHY a vrstvou FDL je tvořeno službami, které poskytuje vrstva PHY vrstvě FDL. Pro popis činnosti je úmyslně použit abstraktní popis, protože

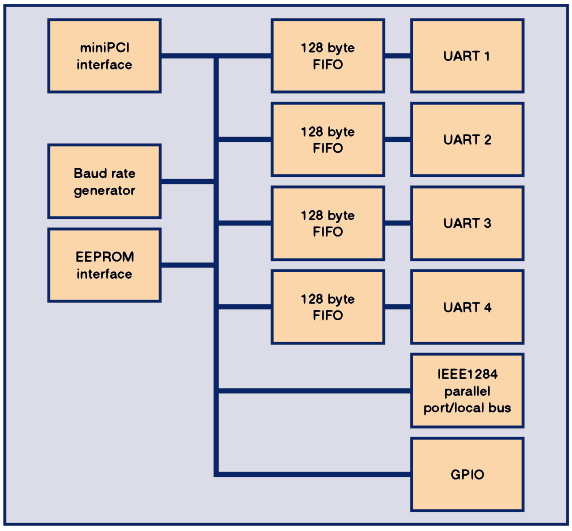
konkrétní implementace může být různá. PHY zajišťuje přijímání a odesílání bitů po médiu. Z pohledu FDL vrstvy je každý bit abstraktně považován za FDL symbol. Jednotlivé FDL symboly jsou základním kamenem UART znaků. Rozhraní vrstvy PHY obsahuje dvě funkční primitiva sloužící k přijímání a odesílání dat. Ovladač FDL pomocí primitiva `PHY_DATA.request` požaduje vykonání přenosu dat od vrstvy PHY. Naopak pomocí `PHY_DATA.indication` informuje vrstva PHY ovladač FDL o přijetí FDL symbolu. Obě primitiva musí při použití obsahovat hodnotu. `PHY_DATA.request` může nabývat hodnot `ZERO` (odpovídá log.0), `ONE` (odpovídá log.1) a `SILENCE` (odpojí vysílač, když není potřeba odeslat další FDL symbol). Vykonání `PHY_DATA.request` musí být synchronizováno s periodou t_{BIT}. Hodnota primitiva `PHY_DATA.indication` indikuje napěťovou úroveň, která je v daném okamžiku na sběrnici. Význam hodnot `ZERO` a `ONE` je shodný jako u `PHY_DATA.request`. Vykonání `PHY_DATA.indication` musí být rovněž synchronizováno s periodou t_{BIT}.

2.1.1 Propojení fyzické vrstvy s PC

Rozhraní RS-485/PCI je na straně PC realizováno modifikovanou variantou komunikační karty TEDIA PCI-1482U osazené řadičem Oxford Semiconductor OXmPCI954 [1]. Výrobcem karty je plzeňská firma TEDIA spol. s r.o. Řadič OXmPCI954 je komplexní obvod, který mimo jiné obsahuje čtyři nezávislé sériové porty vyhovující standardu UART. Každý port je vybaven FIFO vyrovnávací pamětí o velikosti 128 B a dovoluje dosáhnout špičkové přenosové rychlosti až 15 Mbps v asynchronním režimu. Blokové schéma řadiče je uvedeno na obrázku 2.3. Z pohledu jádra operačního systému tvoří komunikační rozhraní obvod UART 16C950, který generuje přerušení v okamžiku, když jeho vyrovnávací paměť obsahuje nová data.

Karta TEDIA PCI-1482U je v základní konfiguraci vybavena komunikačním portem SIO1 a SIO2. Pracovní režim komunikačního portu lze nezávisle nastavit pomocí DIP přepínače SW1 (resp. SW2). K dispozici je režim RS-232, RS-422 a RS-485. Je vhodné po každé manipulaci s kartou ověřit, že příslušný port je nastaven do režimu v souladu s požadovanou aplikací. Volbě režimu RS-485 na komunikačním portu SIO1 odpovídá nastavení DIP přepínače SW1(1:OFF, 2:OFF). Význam DIP přepínače SW2 pro port SIO2 je stejný. Umístění přepínače SW1 a SW2 je patrné z obrázku 2.4. Nastavení DIP přepínačů pro další režimy lze dohledat v dokumentaci karty [2].

Speciálně pro potřeby aplikací s Profibusem DP byly pro Katedru řídicí techniky zakázkově vyrobeny dva kusy karet TEDIA PCI-1482U/48MHz, které nesou výrobní



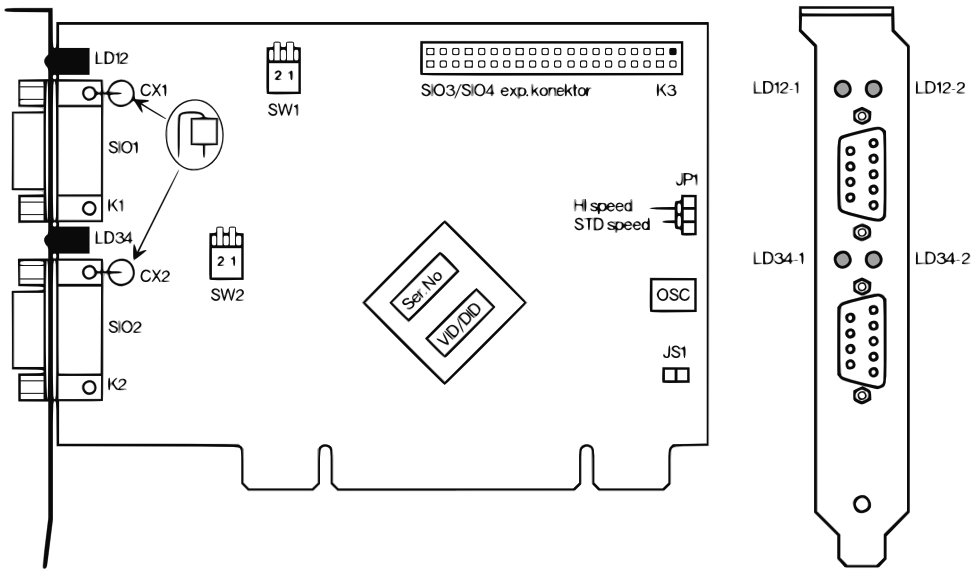
Obrázek 2.3: Blokové schéma řadiče OXmPCI954,
převzato z dokumentace výrobce [1].

čísla **114823053** a **114823054**. Oproti sériově vyráběné kartě TEDIA PCI-1482U na nich byly provedeny následující úpravy:

1. Původní oscilátor s frekvencemi $1,8432$ MHz a $14,7456$ MHz byl nahrazen typem s frekvencí 48 MHz.
2. Optické oddělovací členy byly vyměněny za typ s propustností 100 Mbps.
3. Budiče RS-485 byly vyměněny za typ s propustností 30 Mbps.
4. Ochranné transily RS-485 byly odstraněny.

Přínosem uvedených úprav je zvýšení spolehlivosti a propustnosti dat rozhraní RS-485. Použití komponent navržených pro vyšší nominální přenosové rychlosti umožnilo spolehlivý provoz i na rychlosti 12 Mbps. Protože se jedná o zásadní změny parametrů karty, nelze v systémech Windows použít ovladače dodávané výrobcem. Aby bylo možné bezpečně odlišit modifikované karty od běžné prodejní verze, byla identifikační data řadiče OXmPCI954 nastavena na odlišné hodnoty (více viz [2]).

Připojení komunikačního média ke stanicím je provedeno standardním konektorem D-sub 9 pinů. Na straně stanice je osazen konektor typu zástrčka (female), na kabelu vidlice (male). Vzhledem k tomu, že karta TEDIA PCI-1482U/48MHz má porty SIO1 a SIO2 osazeny v provedení vidlice, je nutné vložit mezi kabel a kartu redukci zástrčka-zástrčka



Obrázek 2.4: Schéma karty TEDIA PCI-1482U/48MHz,
převzato z dokumentace výrobce [2].

s propojením kontaktů, tak jak je uvedeno v tabulce 2.2, nebo přímo na konec kabelu osadit zástrčku se zapojením dle druhého sloupce tabulky 2.2. Karta již obsahuje aktivní přizpůsobení sběrnice RS-485, proto není nutné instalovat konektor s terminátorem.

Číslo kontaktu		Označení vodiče	
Profibus DP	PCI karta	Název	Barva
3	8	RxD/TxD-P	červená
8	9	RxD/TxD-N	zelená

Tabulka 2.2: Propojení vodičů Profibus DP s PCI kartou

Návod na instalaci ovladače pro kartu TEDIA PCI-1482U/48MHz je pro úplnost uveden v příloze A. Způsob práce s ovladačem a formát poskytovaných dat jsou vysvětleny v oddíle 4.6.

Alternativně lze propojit PC s Profibusem DP prostřednictvím běžného sériového portu RS-232 doplněného o pasivní převodník napěťových úrovní. Úskalí této metody spočívá v rozdílných časováních sériového portu a Profibusu DP. Norma [7] povoluje odchylku maximálně $0,3$ % od nominální rychlosti. Toto kriterium splňují pouze rychlosti $9,6$ kbps a $19,2$ kbps. Z tohoto důvodu nelze použít běžný sériový port pro analyzátor komunikace. Podrobně je tato metoda rozebrána v diplomové práci [10], kapitola 3.2.

2.2 Linková vrstva (FDL)

Vrstva FDL tvoří linkovou vrstvu Profibusu DP a v podstatě implementuje celý komunikační protokol. Její úlohou je řídit přístup stanic ke sdílenému komunikačnímu médiu (MAC) a zajistit předávání zpráv. Vzhledem k deterministickému chování Profibusu DP je řízení přístupu kritickou úlohou. Mimo to řeší vrstva FDL tzv. **přechodné stavy**, které se během provozu mohou vyskytnout.

Řízení přístupu ke sdílenému mediu je zajištěno kombinovanou metodou. Stanice jsou z pohledu práva vysílat rozděleny na aktivní stanice (*Master*) a pasivní stanice (*Slave*). Pasivní stanice smí vysílat na sběrnici pouze tehdy, když je přímo dotázána aktivní stanicí. Aktivní stanice může vysílat na sběrnici (dotazovat se pasivní stanice) pouze tehdy, pokud drží oprávnění (*Token*). Každá pasivní stanice je svázána (hardwarovou konfigurací) s konkrétní aktivní stanicí; více aktivních stanic tedy nemůže sdílet jednu pasivní stanici.

Vyskytuje-li se na sběrnici více aktivní stanic (*multimaster*), je nutné zajistit, aby se postupně vystřídaly v přístupu k médiu. Proto musí aktivní stanice, která právě drží token, jej předat (*Token Passing*) po uplynutí určitého času další aktivní stanici s vyšší adresou. Tímto chováním aktivní stanice vytvářejí logický kruh (*Token Ring*). Aktivní stanice, která je v logickém kruhu poslední (má nejvyšší adresu), předává token aktivní stanici s nejnižší adresou. Tím je logický kruh uzavřen. Struktura logického kruhu je popsána seznamem aktivních stanic LAS, který si každá aktivní stanice vytváří a udržuje sledováním komunikace na sběrnici. Z tohoto seznamu také stanice zjistí, jaké následující stanici (NS - Next Station) má token předat a od které stanice (PS - Previous Station) má token opět očekávat. Pasivní stanice se nepodílejí na předávání tokenu, a proto také nejsou členy logického kruhu. Podrobnosti týkající se časových intervalů, které musejí být stanicemi při komunikaci dodrženy, jsou uvedeny v normě [7] nebo v diplomové práci [10].

2.2.1 Výměna dat na sběrnici

Aktivní stanice po dobu, kdy je oprávněna držet token, může uskutečnit výměnu dat s ostatními aktivními a pasivními stanicemi. Rozlišujeme čtyři typy komunikace, které jsou uvedeny sestupně podle priority vykonání:

1. přijetí a předání tokenu,
2. acyklický požadavek nebo požadavek/odpověď,

3. cyklická datová výměna požadavek/odpověď,
4. vyhledání stanice nově vstupující do logického kruhu.

Události probíhající na sběrnici lze rozdělit na cyklickou datovou výměnu mezi aktivními a pasivními stanicemi a tzv. přechodné stavy, což jsou dočasné stavy pozastavující běžnou činnost sběrnice, většinou iniciované nějakou vnější nebo vnitřní událostí. Jmenovitě se jedná hlavně o:

- ztrátu tokenu,
- výskyt více tokenů,
- chybu v předání tokenu,
- kolizi adres stanic,
- vadu přijímače a/nebo vysílače některé stanice,
- připojení a odpojení stanice za provozu.

Při návrhu komunikačního protokolu Profibusu DP bylo věnováno maximální úsilí tomu, aby důsledky vyjmenovaných událostí nevedly k nekontrolovanému rozpadu komunikace a neohrozily tak chod výrobního procesu. Profibus DP dokáže uvedené události deterministicky vyřešit a eliminovat původce události vhodnou rekonfigurací logického kruhu.

Cyklická datová výměna a algoritmus řízení

Cyklická výměna dat je normální provozní stav sběrnice a je svázána se „*scan cyklem*“ jednotlivých PLC (aktivních stanic). Na začátku scan cyklu provádí PLC ovzorkovaní svých fyzických vstupů a přečte hodnoty distribuovaných vstupů, které jsou připojeny Profibusem DP a namapovány do paměťového prostoru PLC. Ovzorkované hodnoty vstupů jsou uloženy do „obrazu vstupů“. Následně je vykonán algoritmus řízení. Hodnoty „obrazu výstupů“ jsou vypočteny ze znalosti stavů vstupů a paměťových proměnných (Mealyho automat). Na konci scan cyklu jsou hodnoty výstupů zapsány do fyzických a distribuovaných výstupů.

Právě pomocí cyklické výměny dat realizuje aktivní stanice čtení a zápis hodnot vstupů a výstupů, které jsou umístěny na některé pasivní stanici. Pro korektní činnost řídicího algoritmu je potřeba zajistit, aby cyklus Profibusu DP byl dostatečně krátký i pro rychlé procesy.

Přidání stanice do logického kruhu

Každá aktivní stanice nejprve provede v přiděleném čase požadovanou komunikaci a ve zbývajícím čase se věnuje prohledávání adresového prostoru od svojí adresy (TS - This Station) k následující aktivní stanici NS. Tento adresový prostor se nazývá *GAP* a je reprezentován seznamem *GAP List*. Na jedno držení tokenu připadá dotázání jedné adresy z GAP Listu. Každá aktivní stanice je odpovědná za vyhledání nově příchozí stanice ve svém GAPu a její uvedení do logického kruhu. To provede tak, že cyklicky dotazuje jednotlivé adresy ve svém GAP Listu žádostí `Request FDL Status`. Pokud stanice obdrží z některé adresy kladou odpověď „ready“, je zřejmé, že se na ní nachází nová stanice, která je připravená vstoupit do logického kruhu. Stanice je připravena vstoupit do logického kruhu právě tehdy, když má vybudovaný svůj vlastní LAS a její FDL řadič přejde ze stavu `Listen Token` do stavu `Active Idle`. Odpověď „not ready“ znamená, že nová stanice ještě nestačila vybudovat svůj LAS a setrvává ve stavu `Listen Token` (viz [7], str. 110). Pasivní stanice odpovídá zprávou „slave station“. Pasivní i nepřipravenou stanici zaznamená prohledávající stanice do svého GAP Listu. Na odpověď „ready“ reaguje aktivní stanice tak, že nově příchozí stanici předá token a uloží ji do svého LAS jako NS. Nová stanice zahájí svoji komunikaci včetně prohledávání svého GAP a konečně předá token svojí NS, která byla původně NS stanicí stanice zajišťující přidání nové stanice. Následující stanice zjistí, že obdržela token od jiné stanice a její adresu uloží do svého LAS jako PS. Tím je proces přidání stanice do logického kruhu ukončen.

Během výše popsané procedury jsou zároveň kontrolovány již známé stanice, zda neopustily logický kruh. Známá stanice, která neodpovídá na výzvu `Request FDL Status` je automaticky vyřazena z logického kruhu. Nastane-li situace, že stanice, která náhle opustila logický kruh, držela token, dojde k jeho ztrátě. Řešení této přechodné události je rozebráno níže. Je-li v GAPu nalezena stanice, která odpoví „master in logical token ring“, jde o chybu, protože z definice je GAP adresní prostor mezi dvěma aktivními stanicemi. Nalezená stanice je přeskočena a token předán stanici NS, která je uvedena v LAS stanice držící token. Přeskočená stanice musí přejít do stavu `Listen Token`, vybudovat nový LAS a počkat na opětovné zařazení do logického kruhu.

Inicializace logického kruhu

Inicializace systému Profibus DP je speciální případ aktualizace LAS a GAP Listu. Aktivní stanice po přípojení napájení (PON - power on) a provedení vlastní inicializace přejde do stavu `Listen Token`. Protože logický kruh ani token zatím neexistují, vyprší

časovač aktivity sběrnice T_{TO} a aktivní stanice s nejnižší adresou vygeneruje token, který dvakrát po sobě předá sama sobě (DA = SA = TS). Tím dá najevo ostatním aktivním stanicím na sběrnici, že je jedinou stanicí v logickém kruhu. V tomto okamžiku obsahuje její GAP List všechny adresy. Stanice začne prohledávat GAP odesíláním zprávy `Request FDL Status` a první nalezenou aktivní stanici, která odpoví „ready“, si zaznamená do LAS jako následující stanici (NS). Vstup dalších stanic do logického kruhu je shodný s postupem uvedeným v předchozím oddíle.

Pokud dojde ke **ztrátě tokenu**, např. jestliže stanice držící token je odpojena od napájení, je nutné provést částečnou inicializaci logického kruhu. Komunikace na sběrnici ustane, protože žádná stanice nedrží oprávnění vysílat. Po vypršení časovače nečinnosti sběrnice T_{TO} vygeneruje stanice s nejnižší adresou token a zahájí standardní cyklickou komunikaci. Délka časovače T_{TO} je odvozena z adresy stanice. Vyhledávání stanic není potřeba provádět, protože LAS a GAP List v jednotlivých stanicích stále existují.

Doba oběhu tokenu

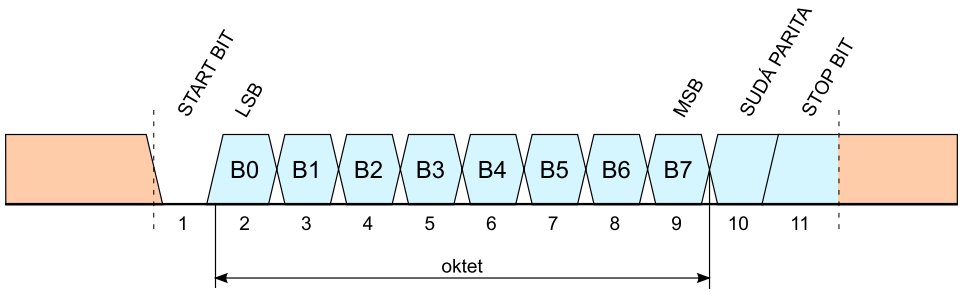
V okamžiku, kdy aktivní stanice obdrží token, spustí se časovač, který měří dobu oběhu tokenu. Běh časovače je zastaven okamžikem opětovného přijetí tokenu od PS. Výsledkem je čas T_{RR} (*real token rotation time*), který říká, za jak dlouho stanice znovu získala oprávnění vysílat. Pro zajištění správné činnosti celého systému je nutné stanovit předpokládanou dobu oběhu tokenu T_{TR} (*target token rotation time*), jejíž hodnota je závislá na počtu aktivních stanic v síti a době potřebné k předání tokenu T_{TC}.

Kolize adres na sběrnici

Aktivní stanice průběžně odposlouchávají komunikaci na sběrnici. V případě, že aktivní stanice dvakrát zachytí rámec, který má v hlavičce uvedenou shodnou zdrojovou adresu jako je její vlastní adresa, stanice automaticky přechází do stavu `Offline` a signalizuje chybu sběrnice (*Bus Failure*).

2.2.2 Formát dat

Každý datový rámec je tvořen posloupností UART znaků, které jsou základní datovou entitou linkové vrstvy. Znak UART se skládá z jedenácti bitů resp. FDL symbolů. Jeho struktura je znázorněna na obrázku 2.5. Znak začíná start bitem, který má vždy hodnotu log.0. Následuje osm datových bitů, kde nejméně významný bit je první v pořadí.



Obrázek 2.5: Formát UART znaku

Za posledním datovým bitem je umístěn paritní bit. Profibus DP používá sudou paritu, tzn. paritní bit je nastaven na log.1 právě tehdy, když počet bitů majících hodnotu log.1 je lichý. Znak končí jedním stop bitem, který musí mít vždy hodnotu log.1.

Synchronizace přijímacího obvodu je provedena na sestupné hraně start bitu. Všechny následující bity jsou vzorkovány uprostřed periody t_{BIT}. Vzorkováním stop bitu je ověřeno, zda nedošlo k rozpadu synchronizace. Jeho hodnota musí být vždy log.1. Norma připouští odchylku od nominální modulační rychlosti $\pm 0,3$ %.

Jednotlivé rámce musí být sestaveny a odvysílány na sběrnici tak, aby mezi dílčími znaky rámce nebyly prodlevy. Za stop bitem předcházejícího znaku musí ihned navazovat start bit následujícího znaku a jejich odstup nesmí být větší něž t_{BIT}. Naopak mezi jednotlivými rámci vysílanými na sběrnici musí být dodržena synchronizační časová prodleva o velikosti alespoň $33t_{BIT}$.

Hodnoty řídících znaků jsou úmyslně stanoveny tak, že vzájemně mají Hammingovu vzdálenost rovnu čtyřem. K dispozici tak je pět řídících znaků, které uvozují příslušný rámec. Konkrétně jde o SC (`0xE5`), SD1 (`0x10`), SD2 (`0x68`), SD3 (`0xA2`) a SD4 (`0xDC`). Komunikační protokol neumožňuje fragmentaci rámců. Všechna požadovaná data se musejí vejít do jednoho rámce nebo může být poslán další rámec. Norma [7] uvažuje čtyři základní typy rámců:

- rámec s pevnou délkou bez datového pole (SC a SD1),
- rámec s pevnou délkou a datovým polem (SD3),
- rámce s proměnou délkou datového pole (SD2),
- token (SD4, rámec o délce tři bajty).

Jejich struktura a význam jsou uvedeny v normě [7] v oddíle 4.6 a podrobně vysvětleny v diplomové práci [10] na stranách 7 a 8. Činnost FDL vrstvy každé aktivní stanice je určena stavovým automatem, jehož přechody reprezentují jednotlivé události na sběrnici. Diagram a popis jednotlivých stavů lze najít v normě [7] v oddíle 4.1.5 nebo v diplomové práci [10] na stranách 9 - 13.

2.2.3 Rozhraní FDL vrstvy

Vrstva FDL poskytuje pouze základní komunikační primitiva uživatelské aplikaci. O rozšíření těchto primitiv se stará tenká vrstva DDLM. K dispozici jsou čtyři komunikační služby sloužící k přenosu dat:

Odeslání dat s potvrzením (*Send Data with Acknowledge, SDA*) umožňuje aktivní stanici odeslat uživatelská data určité vzdálené stanici, která po jejich zpracování odešle potvrzení o úspěšném přijetí. Pokud dojde k chybě, musí aktivní stanice přenos opakovat.

Odeslání dat bez potvrzení (*Send Data with No Acknowledge, SDN*). Tato služba dovoluje aktivní stanici odeslat data jedné (*Unicast*), více (*Multicast*) nebo všem (*Broadcast*) stanicím na sběrnici. Aktivní stanice je informována, že data byla úspěšně odeslána, ale nedostane potvrzení o jejich bezchybném přijetí, což by v případě více cílových stanic bylo i značně problematické.

Odeslání dat s požadavkem odpovědi (*Send and Request Data with Reply, SRD*). Díky této službě může aktivní stanice odeslat data určité vzdálené stanici a v ten samý okamžik požádat o data, která má vzdálená stanice k dispozici. Služba také umožňuje aktivní stanici pouze požádat vzdálenou stanici o poskytnutí dat, aniž by jí byly nějaká data poslána (pole uživatelských dat ve zprávě SD2 má nulovou délku). Aktivní stanice od vzdálené stanice obdrží požadovaná data nebo informaci, že nová data nejsou k dispozici. Obě reakce slouží jako potvrzení přijetí dat vzdálenou stanicí. Pokud došlo k chybě v přenosu dat, vzdálená stanice odešle chybové potvrzení a aktivní stanice musí opakovat přenos.

Cyklické odesílaní a požadování dat (*Cyclic Send Request Data with Reply, CSRD*) je periodickou variantou služby SRD a umožňuje aktivní stanici cyklicky posílat data vzdálené stanici a požadovat zaslání aktuálních dat. Služba také umožňuje posílat cyklicky požadavek neobsahující data. Vzdálená stanice cyklicky odpovídá

zasláním požadovaných dat nebo informací, že data nebyla k dispozici. Vzdálená stanice, která je takto periodicky dotazována, musí být uvedena v seznamu, tzv. Poll List.

2.3 Aplikační vrstva a management sběrnice

Protokol Profibus DP na rozdíl od protokolu Profibus FMS neimplementuje aplikační vrstvu ve smyslu ISO/OSI modelu. Je totiž navržen pro konkrétní relativně omezenou množinu aplikací. Proto mohou být funkce aplikační vrstvy, na rozdíl od obecnějšího protokolu Profibus FMS, realizovány přímo v uživatelském rozhraní. Z komunikačních služeb FDL vrstvy uvedených v předchozím oddíle využívá protokol Profibus DP prostřednictvím DDLM vrstvy pouze službu SRD a SDN. Uživatelské funkce jsou mapovány vrstvou DDLM přímo na komunikační primitiva vrstvy FDL.

Norma [7] v části 8, oddíle 9 definuje funkce a jejich parametry, které musí být implementovány na každé aktivní a pasivní stanici kompatibilní s Profibusem DP.

Kapitola 3

Metody analýzy sběrnice Profibus

Průmyslová sběrnice Profibus DP představuje komplexní systém, který v sobě zahrnuje větší množství subsystémů. Tento celek může být zkoumán z mnoha různých kvalitativních a kvantitativních hledisek. Je na uživateli, aby zvolil vhodnou rozlišovací úroveň analýzy.

V případě komunikační sítě je hloubka zkoumání úzce svázána s vrstvami ISO/OSI modelu. Podle konkrétní vrstvy, na které si přejeme provádět analýzu, volíme vhodné hardwarové a softwarové vybavení. Důležité je si uvědomit, že vlivem zapouzdření jednotlivých vrstev (nižší vrstva je z vyšší vrstvy dostupná pouze přes svoje rozhraní) klesá rozlišovací schopnost, které jsme schopni dosáhnout. Profibus DP implementuje dvě komunikační vrstvy a uživatelské rozhraní, proto lze říci, že existují tři úrovně analýzy.

3.1 Úrovně přístupu a hloubka analýzy

Napěťové úrovně jsou definovány standardem RS-485 a jak bylo uvedeno v kapitole 2.1, používá se kanálové kódování NRZ. Diagnostiku fyzické vrstvy můžeme provést například připojením logického analyzátoru, který umožní odlišit jednotlivé bity. Z posloupnosti bitů následně provedeme rekonstrukci znaků, rámců a zpráv. Kvalitativní vlastnosti signálu šířeného po vedení lze analyzovat použitím digitálního osciloskopu, který umožní pozorovat průběh signálu. Většina komerčních produktů obsahuje vedle klasického rozhraní RS-485 i digitální osciloskop vybavený funkcí interpretace dat ze signálu. Tento přístup se nazývá metoda „**subbitová**“. Její výhodou je, že pracuje přímo s fyzickou vrstvou (vodiči), a tak žádná informace nemůže zůstat skryta v nižších vrstvách.

Mnohem menší požadavky na technické vybavení má metoda spočívající v zachytávání jednotlivých znaků na sběrnici. K tomuto účelu může dobře posloužit například PCI karta nesoucí rozhraní RS-485 a řadič UART. Ze zachycené posloupnosti znaků lze následně rekonstruovat jednotlivé rámce a interpretovat je protokolem Profibus DP. Tato metoda pracuje se znaky na úrovni FDL vrstvy, a proto se označuje jako metoda „**bajtová**“. Tato práce se zabývá pouze získáváním a zpracováním dat bajtovou metodou. Tomu odpovídá i použitý hardware.

Oproti tomu **„metoda zpráv“** většinou používá proprietární síťový hardware a knihovnu implementující příslušné uživatelské rozhraní. Toto API zpřístupňuje data obsažená v rámcích obvykle bez dalších doprovodných údajů. Z tohoto důvodu je tato metoda spíše vhodná pro kontrolu činnosti uživatelské aplikace než pro analýzu datové výměny na sběrnici Profibus DP.

3.1.1 Subbitová úroveň

Zásadní vliv na spolehlivost komunikační sítě má kvalita signálu, který je šířen metalickým vedením. Chyby způsobené nekvalitním signálem patří k těm, které se projevují zcela náhodně a jejich detekce a lokalizace je velmi obtížná. Kvalitativní vlastnosti signálu lze posoudit zobrazením jeho průběhu pomocí digitálního osciloskopu. Na trhu jsou k dispozici produkty (např. ProfiTrace 2 od firmy Procentec nebo PB-T3 Cable Tester od firmy Softing), které integrují digitální osciloskop s šířkou pásma 100 MHz a RS-485 rozhraní v jednom zařízení relativně malých rozměrů. Tyto testery se připojují k běžnému PC (notebooku) prostřednictvím standardní sběrnice USB.

Dlouhé vedení je elektrický obvod s rozprostřenými parametry. Vlastnosti vedení lze charakterizovat čtyřmi parametry, které se udávají na jednotku délky a ovlivňují šíření signálu. Odpor a indukčnost vedení jsou jeho podélné parametry (v modelu jsou řazeny sériově), zatímco svod a parazitní kapacita mezi vodiči jsou příčné vlastnosti (v modulu řazeny paralelně). Všechny čtyři veličiny jsou vnitřní vlastnosti vedení a mají vliv na tvar a napěťovou úroveň obdélníkového signálu. S vlastnostmi vedení úzce souvisí jeho impedanční přizpůsobení. Poškození nebo absence terminátorů se projevuje šířením odrazů po vedení.

Další nezanedbatelný vliv má prostředí, kterým kabel Profibusu DP prochází. Svářecí a rotační stroje podobně jako transformátory a jiná zařízení, která obsahují indukční prvky, mohou způsobovat zarušení vodičů, a tak znehodnocovat signál. V dnešní době nelze ani vyloučit vliv elektromagnetického pole pocházejícího od různých bezdrátových

technologií, ať už se jedná o komunikátory, vysílačky, mobilní telefony nebo radiové datové spoje.

Kapacita vedení způsobuje zaoblování strmosti náběžných a sestupných hran a indukčnost vedení se projevuje překmity na náběžné a sestupné hraně. Odpor a svod vedení způsobuje útlum šířeného signálu. Špatné impedanční přizpůsobení má za následek vznik odrazů od volného konce vedení a následnou interferenci signálu s odrazy. Vnější prostředí může znehodnotit užitečný signál střídavým silovým napětím, které má za následek změny magnetického pole v okolí a indukci napětí na vedení sběrnice. Pokud je vliv těchto jevů výrazný, může dojít k útlumu a deformaci signálu v takovém rozsahu, že přijímací obvody stanic nemusí dokázat rozlišit úroveň odpovídající log.1 od úrovně log.0. Zarušení vedení, vznik výrazných překmitů a napěťových špiček způsobuje saturaci přijímacích obvodů, která může vést až k jejich zničení.

Existuje možnost z detekce napěťových úrovní získat posloupnost bitů a postupně se dobrat až k jednotlivým rámcům. Tento přístup se ale nepoužívá příliš často. V praxi jsou nejprve zvlášť otestovány vlastnosti přenosového media a posouzena kvalita signálu v jednotlivých uzlech sítě. Teprve pokud jsou na úrovni fyzické vrstvy eliminovány nebo alespoň potlačeny zdroje poruch, přistupuje se k analýze komunikace na úrovni linkové vrstvy. Zde jako zdroj informace již slouží příslušné síťové rozhraní (RS-485), nikoliv digitální osciloskop. Na druhou stranu se data linkové vrstvy používají jako zdroj doplňkových informací k průběhům zaznamenaným osciloskopem. Velmi užitečnou funkcí je možnost zobrazit konkrétní průběh napětí na sběrnici doplněný o informaci, která stanice v tomto okamžiku vysílala.

3.1.2 Bajtová úroveň

Nejmenším rozlišitelným objektem v bajtové úrovni je právě jeden bajt. Další podrobnosti jsou skryty za rozhraním fyzické vrstvy, proto nedokážeme jednoznačně identifikovat chyby způsobené nedostatečnou kvalitou signálu a přenosového media, jak bylo uvedeno v předchozím oddíle. Tato práce využívá bajtový (znakový) přístup, protože ovladač PB-Master (viz oddíl 4.6) vrací data zachycená na sběrnici právě jako posloupnost bajtů. S využitím znalostí o formátech jednotlivých rámců FDL vrstvy lze v proudu dat jednotlivé rámce odlišit a provést jejich interpretaci.

Protože protokol FDL neumožňuje fragmentaci rámců, může každý rámec teoreticky obsahovat maximálně 255 znaků, jelikož pro udání délky rámce s proměnnou délkou dat je vyhrazen pouze jeden bajt (LE). Toto omezení při analýze komunikace znamená, že každý zachycený rámec musí obsahovat kompletní informaci, která je předepsaná normou [7]. Při zachytávání komunikace se rozdílem časových známek průběžně kontroluje, zda mezi vysílanými rámci byla dodržena doba klidu na sběrnici T_{SYN} a zároveň lze zjistit, zda se mezi UART znaky (viz obr. 2.5) rámce nevyskytují prodlevy. Nedodržení intervalu T_{SYN} a výskyt prodlev mezi znaky může poukazovat na špatnou synchronizaci vysílající stanice, potažmo poškození stanice nebo špatný signál v místě připojení stanice.

Na úrovni rámce lze kontrolovat obsah jednotlivých bajtů vzhledem k jejich významu. Každý rámec má na prvním místě umístěn řídící znak (viz 2.2.2), který určuje typ rámce potažmo jeho formát. Všechny řídící znaky mají Hammingovu vzdálenost a jsou tak odolné vůči bitovému posunu. Dále rámec obsahuje zdrojovou adresu (SA) a cílovou adresu (DA). Z vývoje předávání zpráv na sběrnici je možno ověřit, že se na komunikaci podílely pouze ty stanice, které se na ní podílet měly. Korektní ukončení rámce lze rozpoznat podle přítomnosti znaku ED (`0x16`) - *end delimiter* na poslední pozici v rámci. Ukončovacímu znaku předchází znak FCS (*Frame Check Sequence*), což je kontrolní součet sestavený z dat v rámci. Hodnota kontrolního bajtu je vypočtena jako aritmetický součet hodnot bajtů DA, SA a FC (*Frame Control*) s tím, že přenos mimo 8-bitové hodnoty je ignorován.

V případě rámců s pevnou nebo proměnnou délkou datového pole jsou při výpočtu FCS uvažovány hodnoty všech datových bajtů (DATA_UNIT). Rámce s pevnou délkou obsahující data mají datový blok vždy dlouhý přesně 8 bajtů. Kontrolním výpočtem vlastní hodnoty a porovnáním s FCS je ověřena integrita dat v přijatém rámci. Pokud rámce s pevnou délkou po přijetí obsahují jiný než normou předepsaný počet znaků, jedná se pravděpodobně o chybu ve firmwaru vysílající stanice. Nesouhlasící kontrolní hodnota FCS poukazuje na poškození dat při přenosu médiem. Rámec SC (`0xE5`) - (*single character*) je tvořen jediným bajtem a slouží jako odpověď ve službě SDA. Rámec tokenu tvoří tři bajty. Za řídícím znakem SD4 (`0xDC`) následuje bajt s adresou příjemce DA a bajt s adresou odesílatele SA. Token a SC nejsou ukončeny znakem ED.

V rámcích s proměnnou délkou dat může být obsaženo $4 \div 249$ bajtů dat, které v sobě zahrnují DA, SA, FC a DATA_UNIT (max. 246 B). Rámec je uveden řídícím znakem SD2, za nímž následují znaky LE a LEr. Hodnota bajtu LE znamená počet datových znaků v rámci. Stejná hodnota je z důvodu potlačení výskytu chyby posunutí zopakována v bajtu LEr. Kontrolou obou hodnot a porovnáním se skutečným počtem znaků lze zjistit,

zda při přenosu nedošlo ke ztrátě dat. Za znakem LEr je zopakován řídicí znak rámce SD2. Teprve pak následují data DA, SA, FC a DATA_UNIT uzavřená znaky FCS a ED.

Souslednost rámců

Vedle vnitřního obsahu rámců lze zkoumat jejich návaznost (dotaz/odpověď). Norma [7] definuje, v jakém pořadí se smí rámce vyskytovat. Nestandardní vývoj výměny rámců indikuje výpadek stanice nebo poškození komunikačního media. Např. pokud stanice opakovaně posílá zprávu FDL_Status jiné stanici a nedostává odpověď, je zřejmé, že došlo k výpadku stanice. Možné výměny rámců jsou uvedeny v normě v oddíle 4.8 na straně 136.

3.1.3 Úroveň zpráv

Vzhledem k tomu, že metoda zpráv pracuje až na výjimky s rozhraním, které je poskytováno ovladačem vyšší vrstvy (FDL, DP), jsou její možnosti analýzy omezené. Nejde ani tak o metodu analýzy, jako spíš o získávání uživatelských dat. Rozhraní ovladačů je již velice abstraktní a komunikace je realizována pomocí objektů (struktur). Výměna dat má charakter komunikace „otázka/odpověď“, takže nelze postupně získávat všechna data jako kontinuální tok. Dalším omezením je, že použitý ovladač vyžaduje zadat během inicializace síťovou adresu, pod kterou bude na sběrnici vystupovat. Tímto okamžikem vstupuje do dění na sběrnici a může jej nežádoucím způsobem ovlivnit. Subbitová a bajtová metoda přistupuje ke sběrnici pasivně, a tak není výměna dat ovlivněna činností aplikace.

Existuje možnost využít část funkcí rozhraní vrstvy FDL a DP, které slouží ke konfiguraci a diagnostice stanic na sběrnice. Tyto funkce umožňují dotazovat se pasivních stanic a aktivních stanic třídy 1 a vyčítat jejich stavové informace. Analyzátor pak pracuje jako aktivní stanice Master třídy 2. Úroveň zpráv dokáže sice provést diagnostiku jednotlivých stanic, ale jde o informace získané na uživatelské úrovni a komunikační sběrnice je transparentní.

Kapitola 4

Implementace serveru v prostředí Linuxu

Operační systém GNU Linux sám o sobě poskytuje bohatou škálu nástrojů pro vývoj a ladění aplikací. Za tímto účelem obsahuje většina binárních distribucí Linuxu vývojové balíčky knihoven obsahující příslušné hlavičkové soubory. Výjimku tvoří distribuce Gentoo[1], která je od základu koncipována tak, že celá její instalace probíhá překladem zdrojových kódů daného softwaru. Odpadá tedy potřeba instalovat vývojové balíčky. Překladový řetězec (*toolchain*) je základní součástí distribuce.

Vyvstane-li potřeba přeložit program pro jinou cílovou platformu (*target*), než na které bude proveden překlad (*build*), použije se metoda nazývaná křížový překlad (*cross compile*). V distribuci Gentoo lze překladač pro křížový překlad elegantně vytvořit pomocí nástroje `crossdev`[2]. Na následujícím příkladu je demonstrován postup sestavení překladače pro platformu PowerPC, na které běží OS GNU Linux se standardní knihovnou GNU C Library. Explicitně je specifikována verze jádra 2.6.16:

```
# crossdev -t powerpc-linux-gnu --kernel 2.6.16 --stage4
```

GNU toolchain je množina programovacích nástrojů sloužících k vývoji aplikací. Zahrnuje v sobě překladač `gcc`, balík nástrojů Binutils, debugger `gdb`, nástroj pro řízení překladu `make`, assembler a autotools (automake, autoconf, autoheader).

GNU Binutils je kolekce programů sloužících k manipulaci s objektovým kódem. Nejdůležitějším je linkovací program `ld`. Dále Binutils obsahují assembler `as` a profiller `gprof`, který umožňuje provádět dynamickou analýzu kódu aplikace.

[1] http://www.gentoo.org

[2] Dokumentaci lze zobrazit příkazem: crossdev –help nebo crossdev -t help.

4.1 Vývojové nástroje

PBServer je jako projekt vyvíjen v prostředí NetBeans 6.0.1[3] s nainstalovaným rozšířením, které přidává podporu pro psaní programů v jazyce C/C++. Hlavní výhodou prostředí NetBeans je, že je naprogramováno v jazyce Java a není tak svázáno s konkrétním operačním systémem nebo grafickým prostředím. Prakticky lze NetBeans provozovat na jakékoliv platformně, na které je k dispozici JRE (Java Run-Time) prostředí. Další výhodou je možnost do NetBeans integrovat vývojové nástroje jako je `gcc`, `gdb` a `make`, stejně tak lze ukládat celý projekt pomocí systémů na správu verzí jako je CVS nebo SVN. Pro použití SVN je nutné doinstalovat rozšíření, které přidá podporu pro tento systém správy verzí. Projekt PBServeru se nachází na přiloženém CD v adresáři `/src/profibus-nb`. Zdrojové kódy PBServeru jsou uloženy odděleně v adresáři `/src/trunk`. Konfigurační soubor překladu je přítomen v každém podadresáři stromu zdrojových kódů a jmenuje se `Makefile.omk`.

Pro řízení překladu zdrojových kódů serveru a přidružených utilit je použit nástroj OMK Make-System[4], který byl vyvinut na Katedře řídicí techniky Ing. Pavlem Píšou a Ing. Michalem Sojkou. OMK funguje jako nadstavba programu `make` a přináší vyšší uživatelský komfort spočívající v přehlednější a srozumitelnější konfiguraci než běžný `make`. Užitečnou vlastností OMK je, že při překladu ukládá soubory s objektovým kódem a výsledné slinkované binární soubory do zvláštních adresářů odděleně od zdrojových kódů. Strom zdrojových souborů tak zůstává během překladu netknutý.

Program `valgrind` je nástroj pro ladění aplikací se zaměřením na kontrolu práce s dynamicky přidělovanou pamětí. Program `valgrind` pracuje na principu spuštění programu ve virtualizovaném prostředí metodou JIT. Používá se zejména k vyhledávání neoprávněné manipulaci s pamětí, když se aplikace začne chovat nestandardně nebo náhodně padá s chybou segmentace (SIGSEGV). Program `valgrind` schopen odhalit:

- použití neinicializované paměti,
- čtení nebo zápis do paměti, která byla uvolněna pomocí `free()`,
- čtení nebo zápis do paměti mimo meze bloku alokovaného funkcí `malloc()`,
- „memory leak“,
- místa ve vícevláknových aplikacích, kde dochází k výskytu „podmínek souběhu“.

[3] http://www.netbeans.org
[4] http://rtime.felk.cvut.cz/omk/

4.2 Struktura aplikace

Kompletní aplikace se skládá ze dvou spustitelných programů. Stěžejním prvkem je program `pbserver`, který realizuje kompletní server a následující text se zabývá tím, jak byl vytvořen a jak funguje. Doplňkový význam má program `pbspy`, což je testovací utilita, která byla vytvořena hlavně za účelem ověření funkčnosti komunikace mezi serverem a klientskou aplikací.

PBServer je program, který pracuje jako TCP server a jeho výkonné části jsou organizovány do vláken. Server podle zvolené konfigurace předává data zachycená na sběrnici Profibus DP klientské aplikaci. Principy komunikace mezi serverem a klientskou aplikací jsou podrobně popsány v kapitole 5. Způsob čtení dat ze sběrnice je uveden v oddíle 4.6. O předzpracování dat serverem pojednává kapitola 6.

Výkonný kód PBServeru je rozdělen do několika zdrojových a hlavičkových souborů. Hlavním důvodem tohoto rozdělení do více souborů je udržet odděleně hlavní důležité funkce, které řídí chod serveru, od funkcí pomocných (zpracování konfigurace, apod.). Projekt `pbserver` obsahuje tyto soubory zdrojových kódů:

`pbserver.h` - Základní hlavičkový soubor PBServeru. Obsahuje odkazy na potřebné hlavičkové soubory, které jsou součástí systému. Dále jsou v něm definovány výchozí hodnoty některých konfigurovatelných parametrů, konfigurační struktury a struktury obsahující parametry vláken. Slovník komunikačního protokolu a hodnoty stavových proměnných jsou definovány pomocí konstant `ST_*` a `PB_*`.

`pbserver.c` - Hlavní zdrojový soubor serveru. Obsahuje funkci `main()` a další důležité funkce, které odpovídají jednotlivým stavům serveru. Dále jsou v něm definovány funkce potřebné pro činnost XML parseru (viz oddíl 5.2.2) a vláknové funkce pro čtení a další zpracování dat.

`pblib.c` - Obsahuje doplňkové funkce sloužící zejména ke zpracování voleb příkazové řádky, konfiguračního souboru, provedení výchozího nastavení serveru, vytvoření logovacího souboru a souboru zámku a funkce pro logování činnosti serveru. Také jsou zde definovány struktury definující volby příkazové řádky (viz oddíl 4.4).

`pbfilter.c` - Definuje funkce pro analýzu, pokročilou filtraci, ukládání dat a zpracování událostí. Dílčí funkce jsou zařazovány do „*pipe-line*“ zpracování v souladu se zadanou konfigurací.

`base64.h` - Hlavičkový soubor knihovny Base64 obsahuje potřebná makra a prototypy kódovacích/dekódovacích funkcí.

`base64.c` - Obsahuje implementace algoritmů na kódování a dekódování Base64. Zde je také definována kódová abeceda.

`pbm_ioctl.h` - Hlavičkový soubor ovladače PBMaster obsahující definici konstant parametrů pro volání funkce `ioctl()`.

Knihovna Base64 je přeložena jako statická knihovna a pomocí OMK je s ní PBServer svázán. Program `pbspy` se celý nachází v souboru `pbspy.c`. Výsledná utilita se po spuštění připojí k PBServeru a simuluje komunikaci mezi serverem a klientskou aplikací. Samotná zakódovaná data nejsou utilitou interpretována. Pro ověřování funkčnosti komunikace jsem používal nástroj Wireshark[5] (dříve Ethereal), který umožňuje nahlédnout do TCP rámců, které si vyměňuje server s klientem.

4.3 Program pracující jako démon

V prostředí operačních systémů Unix/Linux může každý proces pracovat v popředí nebo pozadí. Proces běžící v popředí komunikuje s uživatelem prostřednictvím terminálu, který zajišťuje standardní vstup a výstup. Obvykle je výstup procesu spojen s obrazovkou terminálu a vstup s klávesnicí terminálu. Oproti tomu proces běžící na pozadí pracuje osamoceně bez vazby na terminál, potažmo obrazovku a klávesnici. Uživatel může zjistit, jaký je stav procesu (běží/neběží), ale už jen velice těžko zjistí, co proces dělá. Ve světě systémů Unix/Linux používáme pro takové programy označení *démon*. Toto označení snese srovnání s označením *služba* používaným pro Windows Services v prostředí Win32.

Servery jsou převážně programy pracující v pozadí, jenž jednou spuštěny, běží stále, dokud nejsou ukončeny vnitřní chybou nebo nějakou jinou vnější událostí. Během svého běhu očekávají příchozí požadavky, zpracují je a očekávají další. Stejným způsobem je implementován i PBServer, který běží na počítači vybaveném PCI kartou TEDIA PCI-1482U/48MHz, a očekává požadavek klientské aplikace na dodání dat zachycených z Profibusu DP.

Démon se v Linuxu implementuje tak, že serverový program těsně po svém spuštění provede proces odpoutání se (*detach*) od terminálu a přejde do režimu činnosti na pozadí.

[5] http://www.wireshark.org

Důležitou charakteristikou démon procesů je, že jejich rodičovským procesem je vždy proces init (PID: 1), zatímco u běžných procesů je rodičem terminál nebo *shell*. V kódu PB-Serveru jsem implementoval funkci `void run_daemon()`, která vykoná všechny potřebné kroky a vytvoří proces běžící na pozadí, pokud byl v příkazové řádce nebo konfiguračním souboru zvolen režim činnosti démon.

V prvním kroku proces vytvoří kopii sebe sama pomocí systémového volání `fork()`. Oba procesy nyní vykonávají totožný kód, ale liší se identifikátorem procesu (PID). Rodičovský proces je ukončen systémovým voláním `exit(0)` a osiřelý dceřiný proces je adoptován procesem init. Tím je splněna základní podmínka pro „démony“:

```
pid_t fork_return = fork();
if (fork_return < 0)
   exit(1);            // fork error
if (fork_return > 0)
   exit(0);            // fork success - parent process exits
```

Dceřiný proces po svém rodiči zdědí atributy jako jsou identifikátor skupiny procesů (session) a otevřené file deskriptory. Voláním funkce `setsid()` je proces přesunut do samostatné skupiny a odpoután od terminálu, který jej spustil. Dále jsou uzavřeny všechny zděděné otevřené deskriptory souborů:

```
setsid();
for (i = getdtablesize(); i >= 0; --i) {
    close(i);
}
```

Funkce `int getdtablesize(void)` vrací maximální počet otevřených file deskriptorů, které proces může vlastnit. Z bezpečnostních důvodů je nutné opět vytvořit trojici základních file deskriptorů: standardní vstup - *stdin*, standardní výstup - *stdout* a standardní chybový výstup - *stderr* a svázat je se speciálním zařízením `/dev/null`. Tímto krokem se vyvarujeme nepříjemnostem, které by mohl způsobit fakt, že funkce standardní knihovny předpokládají existenci `stdin`, `stdout` a `stderr`. Zde bych rád podotkl, že `stdin` má identifikátor 1, `stdout` 2 a `stderr` 3:

```
int i = open("/dev/null", O_RDWR);  // open stdin
dup(i);                             // open stdout
dup(i);                             // open stderr
```

Funkce `int dup(int fd)` vytvoří kopii file deskriptoru pod novým identifikátorem, jehož hodnota je o jedna vyšší. Po spuštění procesu můžeme předpokládat, že proces nemá před voláním `fork()` otevřeny jiné file deskriptory kromě těch standardních. Pokud je tato podmínka splněna, lze přesměrování `stdin`, `stdout` a `stderr` na zařízení `/dev/null` provést efektivněji pomocí funkce `freopen()`. Tato funkce slouží ke změně asociace file deskriptorů. Její první parametr specifikuje soubor, který bude otevřen, druhý parametr určuje režim činnosti a poslední parametr je identifikátor file deskriptoru:

```
freopen("/dev/null", "r", stdin);
freopen("/dev/null", "w", stdout);
freopen("/dev/null", "w", stderr);
```

Nyní neexistuje žádná cesta, pomocí které by se dalo se serverem komunikovat nebo získat stavové informace. Např. výstup funkce `printf()` je nyní přesměrován do zařízení `/dev/null`. Řešením je vytvořit funkci, která bude stavové zprávy ukládat do speciálního souboru (*log*). Server běží s právy uživatele root, proto je dobré pomocí funkce `chdir()` nastavit pracovní adresář a chránit vytvořené soubory nastavením oprávnění `int umask(027)`.

```
void log_message(int type, char *message) {
  if (logfile == NULL) {
    fprintf(stderr, "[%d] %d: %s\n", getpid(), type, message);
    fflush(stderr);
  }
  else {
    fprintf(logfile, "[%d] %d: %s\n", getpid(), type, message);
    fflush(logfile);
  }
}
```

Z mnoha důvodů je užitečné zajistit, aby v jeden okamžik mohla běžet právě jedna instance PBServeru. Toto omezení je logické z důvodu, že k dispozici je pouze jedna PCI karta pro čtení dat a že se tak vyvarujeme neodhadnutelných důsledků, které by mohly vzniknout vlivem sdílení zdrojů (*race condition*).

Účinnou metodou jak zajistit, že běží jen jedna instance serveru, je uzamknutí souboru, který slouží pro tento účel, tzv. zámek (souborová obdoba mutexu z knihovny

pthreads). Soubor uzamkneme funkcí `int lockf()`. Jejím prvním parametrem je deskriptor souboru, se kterým si přejeme pracovat, druhý parametr je konstanta specifikující operaci (F_TLOCK - zamknout) a poslední parametr určuje, kolik bajtů od současné pozice v souboru má být takto uzamknuto (0 - celý soubor). Pro přehlednost zapíšeme do souboru identifikátor procesu běžící instance PBServeru:

```
int lock = open(_servercfg->lockfilename, O_RDWR|O_CREAT, 0640);
if (lock_fd < 0) {
  log_message(1, "Can't create/open lock file.");
  exit(1);
}
if (lockf(lock_fd, F_TLOCK, 0) < 0) {
  log_message(1, "Can't lock file. Another instance running.");
  exit(1);
}
write(lock_fd, pidstr, strlen(pidstr));
```

Posledním krokem v cestě za funkčním programem pracujícím na pozadí je zaregistrování příjmu signálů, které jsou jedním ze základních prostředků meziprocesní komunikace, viz kapitola 5. Protože standardní výstup je přesměrován, neexistuje žádná jiná možnost, jak dát procesu najevo, aby provedl nějakou činnost. Signály se obvykle používají např. pro ukončení procesu, restart procesu nebo znovunačtení konfigurace. Příslušné funkce a konstanty jsou nadefinovány v hlavičkové souboru `<signal.h>`. Zaregistrování obsluhy signálu se provede pomocí funkce `signal(int signum, void *handler)`. Funkce přijímá jako první parametr číslo signálu. Druhý parametr je ukazatel na funkci typu void, která je zavolána jako obsluha signálu. Použití konstanty SIG_IGN znamená, že signál bude ignorován. Naopak konstanta SIG_DFL nastaví standardní obsluhu signálu:

```
void signalHandler(int signal) {
  switch(signal) {
  case SIGHUP:
      reloadCfg();
      break;
  case SIGTERM:
      log_message(1, "Exiting...");
      svr_godown = 1;
```

```
      break;
  }
}
signal(SIGHUP, signalHandler);    // handle SIGHUP
signal(SIGTERM, signalHandler);   // handle SIGTERM
```

Každý proces posílá svému rodiči signál SIGCHLD, kterým ho informuje o ukončení svojí činnosti. Návratovou hodnotu dceřiného procesu získá rodič pomocí systémového volání `wait()` a zároveň provede výmaz dceřiného procesu z tabulky aktivních procesů. Pokud volání `wait()` selže nebo rodič z důvodu vnitřní chyby vůbec toto volání neprovádí, zůstává proces stále uveden v tabulce procesů (označen příznakem Z), přestože ukončil svoji činnost a jeho paměť a další zdroje byly uvolněny. Takový proces se nazývá *zombie*. Pokud rodičovský proces dlouhodobě nezpracovává návratové hodnoty svých dceřiných procesů a neruší jejich identifikátory, může dojít k vyčerpání volných identifikátorů procesů a potažmo ke ztrátě schopnosti operačního systému spouštět další programy. Řešením, jak se vyhnout výskytu zombie procesů za cenu ztráty návratové hodnoty, je explicitní ignorování signálu SIGCHLD rodičem, protože pak je záznam v tabulce procesů smazán automaticky včetně návratové hodnoty dceřiného procesu.

```
signal(SIGCHLD, SIG_IGN);
signal(SIGTTOU, SIG_IGN);
signal(SIGTTIN, SIG_IGN);
signal(SIGTSTP, SIG_IGN);
```

Pokud se v systému přesto objeví zombie proces, lze ho ručně odstranit ukončením jeho rodiče. Proces, který ztratil svého rodiče, je adoptován procesem `init`. `Init` je první proces, který je spuštěn jádrem při startu systému. Periodicky provádí volání `wait()` a stará se tak o likvidaci zombie procesů. Většinou je též užitečné ignorovat signály SIGTSTP, SIGTTOU a SIGTTIN, které jsou posílány terminálem. Ošetřením obsluhy signálů končí proces odpoutání se od terminálu a program je připraven pracovat na pozadí jako démon a vykonávat požadovanou činnost.

4.4 Zpracování konfigurace serveru

PBServer je od základu navržen tak, aby většina vnitřních parametrů byla uživatelsky konfigurovatelná. Podrobný seznam všech parametrů, které lze uživatelsky měnit, a formát konfiguračního souboru jsou uvedeny v příloze B. Konfiguraci serveru lze provádět ve třech úrovních:

1. Parametry zadané při spuštění pomocí příkazové řádky mají nejvyšší prioritu.
2. Konfigurační soubor umožňuje mít připravené často používané konfigurace.
3. Konstanty, které jsou deklarované direktivami preprocesoru.

Výchozí hodnoty

Na začátku hlavičkového souboru `pbserver.h` je blok obsahující definice důležitých parametrů. Výchozí hodnoty těchto parametrů lze změnit před překladem zdrojových kódů. Např.:

```
#define PROTOCOL_VER  1.0
#define DEVICE        "/dev/pbmaster0"
```

Volby příkazové řádky

Zpracování argumentů příkazového řádku je velmi často prováděná činnost a jejich syntaktická analýza představuje velmi zdlouhavou a namáhavou práci. Z těchto důvodů obsahuje knihovna GNU C funkci `getopt_long()`, která tuto činnost při psaní programů v C/C++ značně usnadňuje. V prostředí systému Linux se argumenty dělí na *volby* a ostatní argumenty (*hodnoty*). Volby určují, jakým způsobem se má program chovat, a hodnoty jsou vyhrazeny pro jména souborů apod. Funkce `getopt_long()` je velmi robustní a umožňuje zpracovat jak krátké, tak dlouhé zápisy voleb včetně jejich hodnot.

Krátká volba se skládá z pomlčky a jednoho znaku případně následovaného hodnotou. Znak může být malé i velké písmeno anglické abecedy. Př. `-d`,`-d 5`.

Dlouhá volba se zapisuje jako dvě pomlčky a jméno volby, také může být následována hodnotou. Názvy mohou obsahovat pouze malá písmena. Př. `--debug`, `--debug 5`.

Oba dva druhy zápisu jsou ekvivalentní a jejich používání závisí na osobních preferencích uživatele. Dlouhé volby jsou snazší na zapamatování, zatímco krátké volby umožňují rychlý zápis. Popsané formátování voleb je součástí standardu POSIX.

Operační systém při spuštění programu zajistí předání argumentů zadaných z příkazové řádky do programu v podobě textového řetězce. V jazyce C/C++ jsou argumenty přístupné v proměnné `*argv[]`. Počet argumentů obsažených v poli `argv` je uložen v proměnné `argc`. Běžnou praxí je tyto proměnné použít jako parametry funkce `main()`:

```
int main(int argc, char *argv[]) { }
```

Funkce `getopt_long()` je definována v hlavičkovém souboru `getopt.h`. Před jejím použitím je nutné vytvořit speciální datové struktury popisující uspořádání voleb. Typy těchto struktur jsou rovněž definovány v hlavičkovém souboru `getopt.h`:

```
int getopt_long(int argc, char * const argv[],
                const char *short_options,
                const struct option *long_options, int *longindex);
```

Funkce očekává celkem pět argumentů. První argument je počet řetězců v poli `argc`. Druhý argument je ukazatel na pole `argv`. Ukazatel `short_options` ukazuje na znakový řetězec obsahující seznam krátkých voleb. Každá volba je definována jedním znakem. Volby, za kterými se předpokládá hodnota, jsou vyznačeny dvojtečkou. Čtvrtý argument je ukazatel na pole struktur popisujících vazby mezi krátkými a dlouhými volbami. Poslední argument není důležitý a běžně je nastaven na NULL.

```
static const char* short_options = "c:Dhl:v";
```

Dlouhé volby jsou specifikovány polem `long_options`, jehož prvky jsou struktury `struct option`. Každý prvek odpovídá jedné volbě a obsahuje čtyři položky. První položka obsahuje jméno dlouhé volby. Druhá položka obsahuje 1, pokud je za volbou očekávána hodnota. Třetí položka je NULL a poslední položka obsahuje znak odpovídající příslušné krátké volbě. Poslední prvek pole by měl mít všechny položky nulové. Pole struktur odpovídající řetězci `short_options` má následující tvar:

```
static const struct option long_options[] = {
  {"config",  1, NULL, 'c'},
  {"daemon",  0, NULL, 'D'},
  {"help",    0, NULL, 'h'},
```

```
  {"logfile", 1, NULL, 'l'},
  {"verbose", 0, NULL, 'v'},
  {NULL,      0, NULL,  0 }
};
```

Obvykle se funkce `getopt_long()` volá v cyklu, dokud v poli `argv` existuje volba, která ještě nebyla zpracována. Při každém průchodu cyklem funkce provede syntaktickou analýzu jedné volby a vrátí znak odpovídající této volbě nebo hodnotu -1, pokud pole `argv` neobsahuje další volby. Návratová hodnota je vyhodnocena příkazem switch, v jehož těle dojde ke zpracování dané volby. Následuje-li za volbou hodnota, je uložena jako řetězec a přístupná v podobě globální proměnné `optarg`. Vyskytne-li se v poli `argv` neplatná volba, vrátí funkce `getopt_long()` znak „?“. Standardně chovající se program vypíše na terminál zprávu o zadání neplatné volby, vytiskne přípustné volby a ukončí se. Vyčerpávající popis funkce `getopt_long()` je uveden v [8]. Další funkce pro zpracování voleb příkazové řádky jsou popsány v manuálových stránkách[6]. V kódu PBServeru je celá procedura zpracování příkazové řádky řešena v těle funkce `parse_cmdln()`, která je volána během inicializace serveru (viz 4.5):

```
static void parse_cmdln(int argc, char *argv[]) { ... }
```

Konfigurační soubor

Jméno konfiguračního souboru musí být určeno pomocí volby příkazové řádky `-c file`, resp. `--config file`. Po zpracování voleb příkazové řádky (viz předchozí oddíl) je jméno konfiguračního souboru uloženo do konfigurační struktury serveru `t_servercfg`, která je přístupná přes globální ukazatel `*srvcfg`. Není-li jméno souboru zadáno nebo soubor neexistuje, pokusí se server v pracovním adresáři otevřít výchozí konfigurační soubor definovaný konstantou `CFG_FILE` v hlavičkovém souboru `pbserver.h`. Pokud ani tento soubor neexistuje nebo neobsahuje platnou konfiguraci, server ohlásí chybu a ukončí se.

```
srvcfg->cfgfilename = optarg;
```

Jméno konfiguračního souboru je předáno funkci `parse_cfgfile()`, která provede otevření souboru pro čtení, zpracování voleb a uzavření souboru. Formát konfiguračních direktiv má velmi jednoduchý formát. Přípustné jsou dva druhy zápisu:

[6] V Linuxu příkazem: man 3 getopt.

- Název direktivy tvořený textovým řetězcem a následovaný číselnou hodnotou, přičemž název a hodnota jsou odděleny znakem „=“. Př. `baudrate = 187500`
- Název direktivy tvořený textovým řetězcem, který je následovaný textovým řetězcem. Řetězce jsou odděleny znakem „=“. Př. `logfile = pbserver.log`

Na jednom řádku konfiguračního souboru smí být uvedeno pouze jedno přiřazení hodnoty parametru. Při výskytu dalších textových řetězců je zobrazeno chybové hlášení. Názvy direktiv rozlišují velká a malá písmena.

4.5 Činnost serveru a jeho stavový automat

PBServer se může od okamžiku spuštění do okamžiku ukončení nacházet v jednom z celkem devíti stavů. Přechody mezi stavy jsou iniciovány komunikačním protokolem, konfiguračními parametry nebo výskytem chyby. V základní konfiguraci PBServer očekává připojení klienta, v jehož režii je další vývoj chování serveru. Na příkaz od klienta PBServer přejde do samostatného režimu činnosti „*standalone*“ a provádí zadanou činnost. Režim standalone může být také nastaven jako výchozí v konfiguraci PBServeru. Souvislosti mezi stavy jsou znázorněny na obrázku 4.1. Jednotlivým stavům odpovídá následující chování serveru:

Inicializace - Jsou inicializovány základní konfigurační struktury `srvcfg` a `readercfg`. Ve statické funkci `config_init()` jsou výchozí hodnoty konfiguračních parametrů uloženy do struktur. Také jsou vytvořeny proměnné identifikátorů vláken.

Zpracování příkazové řádky - Zde je provedena analýza voleb příkazové řádky (viz 4.4) a získaná nastavení jsou uložena do konfiguračních struktur. Pokud není zadáno jméno konfiguračního souboru, server se ukončí přechodem přes stav „Chyba“.

Zpracování konfiguračního souboru - Funkce `parse_cfgfile()` otevře konfigurační soubor a zpracuje po jednotlivých řádcích uvedené volby včetně nastavení příslušných proměnných.

Spuštění démona - V případě, že byla v konfiguraci zadána volba pro provoz v režimu démon, je zavolána funkce `runDaemon()`, která vykoná požadované kroky (viz 4.3).

Čekání na klienta - Server vytvoří soket a provede jeho konfiguraci. Pak čeká ve funkci `listen()` na příchozí spojení od klientské aplikace.

Klient připojen - Spojení s klientem je navázáno pomocí funkce `accept()`. Následně je spuštěno vlákno, které vykonává funkci `client_thread()` a asynchronně zpracovává požadavky přicházející od klientské aplikace. Dojde k vytvoření instance XML parseru a registrace jeho obslužných funkcí. XML parser vyhodnocuje jednotlivé performativy komunikačního protokolu (viz kapitola 5) a zajišťuje předání požadavku ke zpracování do příslušné funkce. Mezitím hlavní vlákno očekává připojení dalšího klienta.

Čtení dat - Čtení dat ze zařízení ovladače PBMaster je řešeno v samostatném vlákně, které vykonává funkce `busread_thread()`. Mechanismus zpracování dat připomíná „*pipeline*“ zpracování a je podrobně popsán v kapitole 6. Výsledná data jsou odeslána klientovi pomocí funkce `write()`.

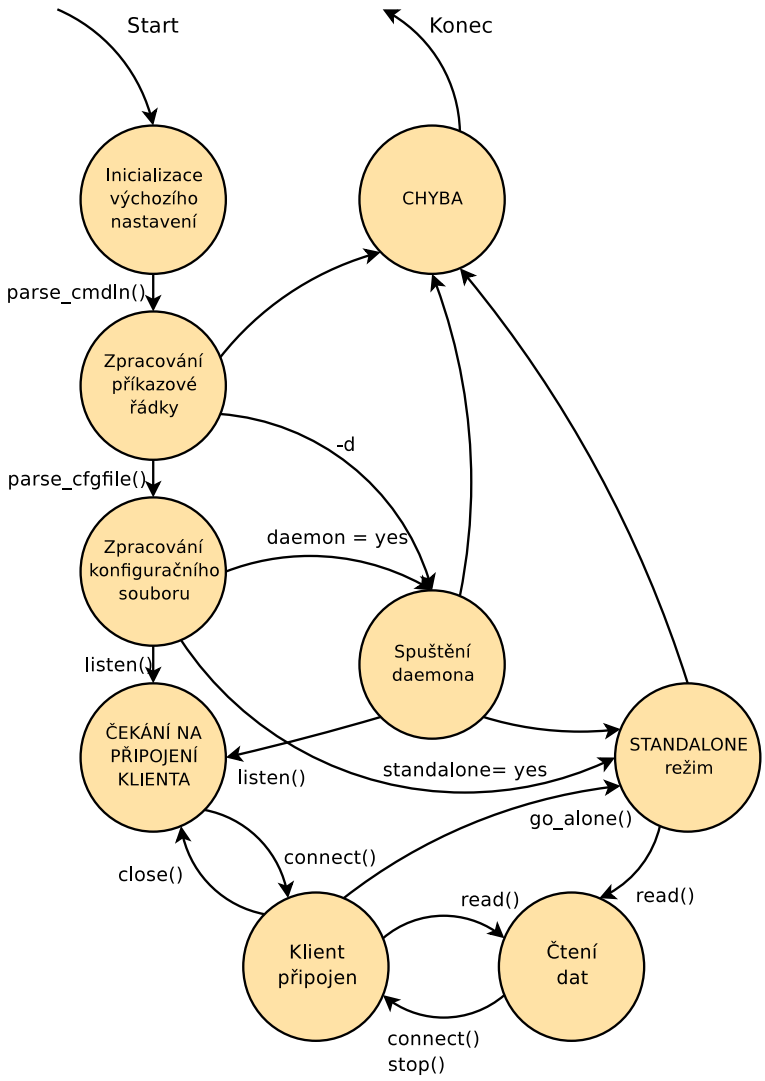
Samostatný režim - Server čte, zpracovává a ukládá data do lokálního souboru dle předem nastavené konfigurace bez interakce s klientskou aplikací. Zároveň očekává připojení klienta a na jeho příkaz poskytne data z offline úložiště.

Chyba - Všechna nestandardní ukončení procházejí přes chybový stav, který je reprezentován funkcí `server_shutdown()`. Tato funkce se postará o uvolnění alokované paměti, uzavření otevřených file deskriptorů, korektní ukončení všech běžících vláken a zaznamenání chyby do logovacího souboru.

4.6 Driver PBMaster

PBServer získává data z Profibusu DP skrze ovladač PBMaster, který byl vytvořen v rámci bakalářské práce [9] a upraven autorem pro lepší spolupráci s rozšiřující PCI kartou TEDIA PCI-1482U/48MHz. Podrobný postup překladu a instalace ovladače je uveden v příloze A. Pro úspěšný překlad je nutné mít nainstalovaný překladový řetězec pro požadovanou platformu tak, jak bylo popsáno v oddíle 4.1.

Server s ovladačem komunikuje pomocí souboru zařízení `/dev/pbmaster0`. Toto zařízení tvoří rozhraní mezi jádrem a uživatelským prostorem. Formát dat tak, jak se vyskytují na Profibusu DP, není vhodný pro další zpracování. Rámce postrádají jakoukoliv in-

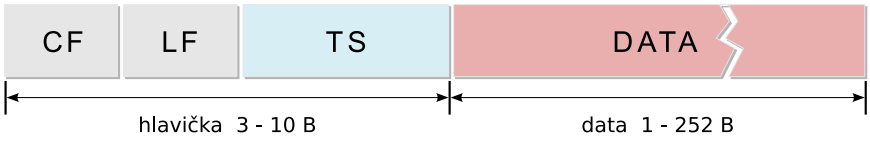


Obrázek 4.1: Stavový diagram chování serveru

formaci o čase a svojí velikosti. Z tohoto důvodu ovladač doplňuje každý zachycený rámec o časovou známku vztaženou k času nahrání ovladače do jádra a velikost rámce v bajtech. Časová známka je uváděna v jednotkách t_{bit}, které odpovídají době potřebné pro přenos jednoho bitu po sběrnici při dané modulační rychlosti. Přiřazení časové známky je nutné provádět již na úrovni ovladače, protože v uživatelském prostoru nelze běžně dosáhnout dostatečné rozlišovací schopnosti.

4.6.1 Formát dat

Každý rámec se skládá z hlavičky a těla, ve kterém jsou uložena vlastní zachycená data. Graficky je tato struktura znázorněna na obrázku 4.2. Význam jednotlivých polí je pro lepší přehlednost shrnut v tabulkách 4.1 a 4.2. Hlavička může mít délku $3 \div 10$ B v závislosti na délce časové známky. Hlavička začíná řídicím polem CF (*control field*), které má vždy délku 1 B. Bity 0 a 1 udávají velikost časové známky TS (*time stamp*). Přípustné hodnoty a jejich význam jsou uvedeny v tabulce 4.2. Bity 2 až 7 jsou vyhrazeny pro budoucí využití a nyní jsou nulové. Další pole je označeno LF (*length field*). Jeho délka je 1 B a udává délku datového pole DATA. Časová známka TS má proměnnou délku, která je určena polem CF. Přípustná je délka 1, 2, 4 nebo 8 B. Časovou známkou končí hlavička a začíná datová oblast DATA, která může obsahovat $1 \div 252$ B dat. Rámec s nulovou délkou dat neexistuje.



Obrázek 4.2: Formát rámce

4.6.2 Nastavení driveru

Při konfigurování činnosti serveru je také zapotřebí mít možnost změnit aktuální parametry ovladače. Požadovaná rychlost sběrnice Profibus DP je serveru předána pomocí

Označení	Délka	Význam
CF	1 B	bit 2-7 - nevyužito bit 0-1 - délka časové známky
LF	1 B	délka zachycených dat v B unsigned char, hodnota $0 \div 255$
TS	1/2/4/8 B	časová známka
DATA	$1 \div 252$ B	zachycená data

Tabulka 4.1: Význam datových polí v rámci

Bit		Hodnota	Délka	Datový typ
1.	0.			
0	0	0	1 B	unsigned char
0	1	1	2 B	unsigned short
1	0	2	4 B	unsigned int
1	1	3	8 B	unsigned long long

Tabulka 4.2: Přípustné hodnoty 1. a 0. bitu v poli CF

příkazů komunikačního protokolu (viz 5.2.3). Rychlost sběrnice lze nastavit parametrem „baud“ modulu pbm_950pci při jeho nahrání do jádra příkazem `insmod` (viz příloha A). Tento způsob ale není vhodný pro uživatelskou změnu rychlosti prostřednictvím serveru.

Systém Linux poskytuje nástroj umožňující změnit parametry zařízení prostřednictvím jejich speciálních souborů. Ovladač PBMaster v systému vytvoří znakové zařízení reprezentované souborem `/dev/pbmaster0`. Pomocí standardní funkce `ioctl()` může uživatelská aplikace komunikovat s hardwarovými zařízeními nebo součástmi jádra a překročit tak bariéru mezi uživatelským prostorem a prostorem jádra. Rozhraní ioctl umožňuje prostřednictvím jediného systémového volání dopravit do jádra různé požadavky pro rozdílná zařízení. Tyto požadavky musí být předány jako vektor. Jádro následně podle handleru zařízení předá požadavek zařízení. Funkce `ioctl()` je definována v hlavičkovém souboru `sys/ioctl.h`:

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...);
```

Prvním parametrem funkce je file deskriptor získaný pomocí funkce `open()`, která je volána na speciální soubor daného zařízení. Druhým parametrem je kód požadavku, který je specifický pro každé zařízení. Třetí argument je void ukazatel na paměť obsahující hodnotu, která bude nastavena. Kódy požadavků jsou definovány jako konstanty v hlavičkovém souboru `pbm_ioctl.h`:

```
#define PBM_IOCTL_MAGIC        'p'
#define PBM_IOCSETD_BAUD       _IOW(PBM_IOCTL_MAGIC, 1, unsigned int)
#define PBM_IOCSETD_RESET_TS   _IOW(PBM_IOCTL_MAGIC, 2, unsigned int)
```

Samotné nastavení parametrů ovladače se provede voláním funkce `ioctl()` nad otevřeným file deskriptorem zařízení. Je nutné kontrolovat návratovou hodnotu funkce, zda nastavení proběhlo korektně:

```
int fd;
int baud_rate;

baud_rate = 187500;
fd = open(PB_DEVICE, O_RDONLY);

ioctl(fd, PBM_IOCSETD_BAUD, &baud_rate);
ioctl(fd, PBM_IOCSETD_RESET_TS, 0);
```

V závislosti na konfiguraci serveru je časová známka nulována automaticky před každým zahájením zachytáváním dat. Ve výchozím nastavení je nulována pouze při prvním otevřením souboru zařízení funkcí `open()`. Po provedení výše popsaných nastavení je driver připraven dodávat data, jejichž zpracování je rozebráno v kapitole 6.

Kapitola 5

Server/klient komunikace

K přenosu informace mezi dvěma běžícími procesy používáme prostředky meziprocesní komunikace (IPC). Jedná se o množinu technik, které nám umožňují, aby proces A poskytl data procesu B a naopak. Není podmínkou, aby komunikující procesy běžely na jednom počítači. Jednotlivé techniky dělíme na předávání zpráv, synchronizaci, sdílení paměti a vzdálené volání procedur (RPC). V prostředí operačního systému Linux nejčastěji používáme signály, pojmenované a nepojmenované roury, semafory nebo sdílenou paměť.

V případě analyzátoru je komunikace mezi serverem a klientem realizována pomocí soketů (Socket). Pro lepší orientaci si soket můžeme představit jako pojmenování pro konce obousměrné roury, kterou procházejí data od serveru ke klientovi a naopak. Soket je velice univerzální nástroj, který skrze jediné API umožňuje sestavit virtuální komunikační kanál nezávisle na technologii a topologii komunikačního média. To je také důvod, proč pro nastavení soketu je zapotřebí relativně velké množství úkonů. Z pohledu ISO/OSI modelu sokety realizují virtuální spojení mezi procesy na úrovni transportní vrstvy (4. vrstva ISO/OSI i TCP/IP modelu) a nižší vrstvy jsou z pohledu uživatele zapouzdřeny.

5.1 Komunikační kanál

Reálné použití analyzátoru vychází z předpokladu, že komunikačním médiem bude síť typu Ethernet. S ohledem na objem dat, který může být špičkově přenášen, alespoň Ethernet 100BASE-T. Z toho plyne rozumné omezení, že server i klient by se měly vyskytovat v rámci jedné sítě LAN. Přístup prostředím Internetu neumožňuje online přenos

dat do klientské aplikace. Server analyzátoru využívá BSD implementaci soketů, která je výchozí pro TCP/IP a operační systém Linux. V prostředí Win32 a .NET nalezneme kompatibilní implementaci Winsock.

Každý soket je specifikován třemi parametry: jmenným prostorem, stylem komunikace a protokolem. Pro specifikaci jmenného prostoru používáme množinu konstant PF_*, které určují, jak jsou sokety adresovány. Adresa lokálního soketu odpovídá jménu obyčejného souboru. V Internetu je soket určen IP adresou a TCP nebo UDP portem. Styl komunikace definuje, jak jsou data rozdělena na pakety (rámce), kolik účastníků se účastní komunikace a jak jsou pakety adresovány (jednotlivě nebo existuje spojení). Rozlišujeme komunikační styl orientovaný na spojení (SOCK_STREAM) a styl orientovaný na data (SOCK_DGRAM).

Server analyzátoru používá styl orientovaný na spojení, protože ten vytvoří dvoubodové spojení a garantuje doručení paketů v pořadí, v jakém byly odeslány. V prostředí Internet tomuto stylu odpovídá protokol TCP. Naopak datově orientovaný styl se pro naše účely nehodí, protože nevytváří spojení, nezaručuje doručení ani zachování pořadí paketů. V Internetu odpovídá tomuto stylu protokol UDP. Zde je třeba si uvědomit, že spolehlivost protokolu TCP je zajištěna za cenu nárůstu režijních dat a opakovaných přenosů (souhrnně *overhead*).

Pro zpřístupnění potřebných funkcí, struktur a konstant ve zdrojovém kódu serveru musíme vložit odkazy na tyto hlavičkové soubory:

`<sys/socket.h>` obsahuje základní funkce pro práci se sokety a definuje potřebné datové struktury.

`<netinet/in.h>` definuje rodinu adres AF_INET a AF_INET6, které jsou používány v Internetu a obsahují IP adresu a číslo TCP a UDP portu.

`<arpa/inet.h>` definuje funkce pro manipulaci s IP adresami, zejména převod mezi různými reprezentacemi bajtů.

Server po zpracování parametrů příkazové řádky a načtení konfiguračního souboru přejde do režimu režimu daemon voláním funkce `run_daemon()` tak, jak bylo uvedeno v kapitole 4.3. V dalším kroku otevře nový soket pomocí funkce `socket()`,

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

jejíž návratovou hodnotou je file deskriptor nového soketu. Při konkrétní implementaci v serveru je vhodné otestovat návratovou hodnotu a v případě neúspěchu ukončit server. Tato situace může nastat v případě, že jádro operačního systému neobsahuje podporu pro sokety nebo proces vyčerpal maximální počet otevřených file deskriptorů.

```
if ((sock_fd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
  log_message(0, "Unable to create socket. Exiting.");
  exit(1);
}
```

Konstanta PF_INET znamená, že soket používá síťový protokol IPv4 a adresace rozhraní bude provedena pomocí IP adresy. Protože analyzátor vyžaduje spolehlivou výměnu dat, je použit komunikační styl SOCK_STREAM. V kontextu s IPv4 je tedy použit protokol TCP.

V případě, že otevření soketu a připojení klienta neproběhne korektně, nebo je zapotřebí server nestandardně restartovat (nedojde k volaní funkce `close()` na file deskriptor procesu), nelze použít stejnou IP adresu a port pro vytvoření soketu v nové instanci serveru. Soket je operačním systémem automaticky uzavřen a jeho file deskriptor uvolněn až po určité časové prodlevě od zániku procesu serveru. Opakovatelné použití IP adresy a soketu zajistíme úpravou parametrů soketu ihned po jeho prvním vytvoření pomocí funkce `setsockopt()`. Nastavení jednotlivých parametrů lze získat pomocí funkce `getsockopt()`:

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockopt(int s, int level, int optname, void *optval, int *optlen);
int setsockopt(int s, int level, int optname, const void *optval,
               int optlen);
```

Kompletní seznam všech parametrů, které lze u soketu nastavit, nalezneme v [8] nebo v manuálové stránce uvedených funkcí[1]. Pro naší potřebu nastavíme na *true* parametr `SO_REUSEADDR`:

```
int yes = 1;
setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));
```

[1] V Linuxu příkazem: man 2 setsockopt.

Funkce `setsockopt()` provede nastavení parametru soketu, který je specifikován deskriptorem `sock_fd`. Konstanta SOL_SOCKET říká, že nastavení proběhne na úrovni soketu. Dále je funkci předána, pomocí ukazatele na typ `void`, adresa místa paměti, kde je uložena hodnota, na kterou se má parametr SOL_SOCKET nastavit. Posledním parametrem funkce je velikost objektu uloženého na adrese `&yes`. Jako bylo uvedeno v případě funkce `socket()`, je i zde vhodné kontrolovat návratovou hodnotu funkce `setsockopt()`.

Nyní má server k dispozici nakonfigurovaný soket, který ale zatím není svázán s žádnou IP adresu a TCP portem. Přiřazení IP adresy a TCP portu provedeme voláním funkce `bind()`, které předáme deskriptor soketu a ukazatel na speciální strukturu typu `sockaddr`, ve které provedeme příslušná nastavení:

```
struct sockaddr_in our_addr;

our_addr.sin_family = AF_INET;
our_addr.sin_addr.s_addr = INADDR_ANY;
our_addr.sin_port = htons(_servercfg->bindport);
```

Konstanta AF_INET, která koresponduje se jmenným prostorem PF_INET, určuje, že se jedná o IPv4 adresu. Obdobně pomocí konstanty INADDR_ANY do struktury uložíme IP adresu, která je přiřazena síťovému rozhraní serveru. V případě, že server fyzicky obsahuje více síťových rozhraní, je třeba věnovat pozornost tomu, kterou adresu obsahuje konstanta INADDR_ANY. Posledním přiřazením do struktury `our_addr` je nastavení čísla portu, na kterém bude server očekávat komunikaci.

Při manipulaci s IP adresami a čísly portů v programu můžeme narazit na problematiku řazení bajtů na různých platformách. Počítače založené na architektuře Intel x86 a z ní odvozené používají řazení bajtů *little-endian*, kde nejméně významný bajt je v paměti uložen na nejnižší adrese. Počítače založené na procesorech od firmy Motorola naopak ukládají na nejnižší adresu nejvíce významný bajt. Toto řazení se nazývá *big-endian*. Fakt, že různé počítačové platformy reprezentují např. 32-bitový Integer různě, má zásadní dopad na schopnost vzájemně komunikovat přes síť, která je založena na jednom protokolu. Řešení spočívá v tom, že všechny sítě používají řazení *Network order*, které odpovídá řazení big-endian. Pro zjednodušení práce s IP adresami poskytuje BSD sockets API sadu funkcí provádějících konverze mezi jednotlivými řazeními. V případě struktury `our_addr` je číselná hodnota TCP portu zpracována funkcí `htons()`, která mění pořadí bajtů z řazení počítače do řazení sítě:

```
#include <arpa/inet.h>
uint16_t htons(uint16_t hostshort);
```

Funkce `bind()` provede svázání soketu s konkrétní IP adresou a TCP portem. Význam jednotlivých parametrů, které funkce očekává, je zřejmý[2]:

```
bind(sock_fd, (struct sockaddr*) &our_addr, sizeof(struct sockaddr));
```

Dalším krokem ve vytváření funkčního soketu je zavolání funkce `listen()`, která nastaví soket do režimu čekání na spojení. Druhým parametrem funkce je celočíselná hodnota `backlog`, která určuje kolik příchozích spojení čekajících na vyřízení může být maximálně zařazeno do fronty. Příchozí spojení nad tento limit budou aktivně odmítnuta. Server s ohledem na velký objem přenášených dat nepřipouští více spojení současně.

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Funkce `listen()` se vrací okamžitě a samotné vyčkávání na příchozí spojení provádí funkce `accept()`, která při svém zavolání uloží do struktury `their_addr` parametry prvního spojení ve frontě. Její návratovou hodnotou je file deskriptor nového soketu `new_fd`, jež slouží ke komunikaci s druhou stranou. Proto původní soket `sock_fd` může být použit pro opětovné volání funkce `accept()`. Pokud je fronta prázdná, funkce opět čeká na příchozí spojení. V případě, že serverová aplikace má obhospodařovat více spojení současně, pak se volání funkce `accept()` provádí cyklicky v nekonečném cyklu, dokud není dosažen limit zdrojů použitelných pro paralelní zpracování (vláken, procesů). Taková aplikace je obvykle řešena jako více vláknová.

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

struct sockaddr_in their_addr;
new_fd = accept(sock_fd, (struct sockaddr *)&their_addr, &sin_size);
```

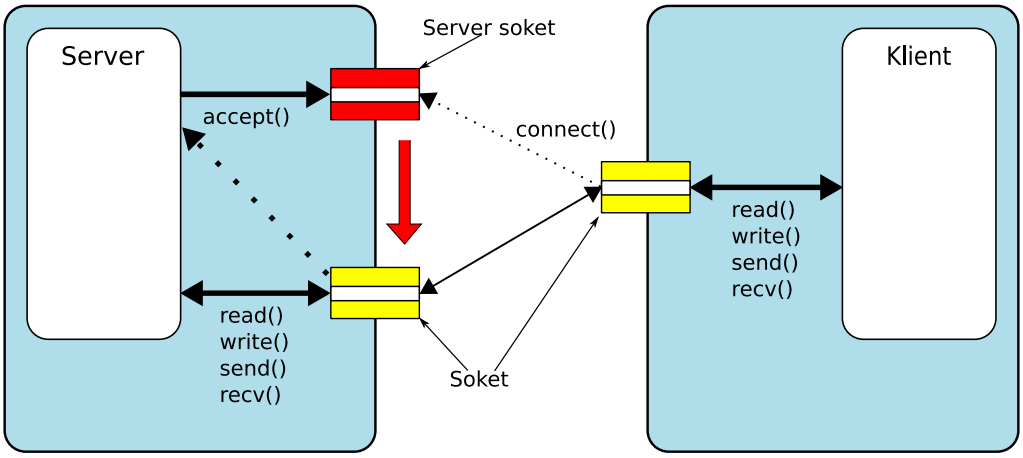
Chování funkcí `listen()` a `accept()` můžeme sledovat pomocí příkazu `netstat`[3]. V prvním výpisu je server ve stavu po úspěšném volání funkce `listen()` a funkce `accept()` čeká na příchozí spojení:

[2]Dokumentace v Linuxu: man 2 bind.
[3]netstat -tupan | grep pbserver

```
Proto In Out Local addr      Remote addr     Status    PID/Program name
tcp    0  0   0.0.0.0:5555   0.0.0.0:*       LISTEN    14043/pbserver
```

Po připojení klienta můžeme vidět, že původní serverový soket opět čeká na příchozí spojení a skrze nově vytvořený soket probíhá komunikace s klientem:

```
Proto In Out Local addr      Remote addr     Status    PID/Program name
tcp    0  0   0.0.0.0:5555   0.0.0.0:*       LISTEN    14043/pbserver
tcp    0  0   127.0.0.1:5555 127.0.0.1:2301  CONNECTED 14043/pbserver
```



Obrázek 5.1: Princip funkce `accept()`

Výsledkem výše uvedených kroků je úspěšně navázané spojení mezi serverem a klientem, které využiji pro přenos řídicích a konfiguračních příkazů a také k přenosu dat zachycených ze sběrnice Profibus. K odesílání a přijímání dat skrze soket se používají standardní funkce `send()`, `recv()`, `write()` a `read()` a způsob práce se soketem je podobný práci s otevřeným souborem na lokálním médiu.

5.2 Realizace komunikačního protokolu

Uživatelská data jsou soketem přenášena jako posloupnost bajtů v proudu. Pro účely aplikace je nutné zavést komunikační protokol na úrovni aplikační vrstvy (7. vrstva ISO/OSI), který definuje formát uživatelských zpráv, jejich význam a způsob přenosu

aplikačních dat. Aplikační data mnohdy představují složité objekty a struktury, které je zapotřebí přenášet skrze TCP spojení v serializovaném tvaru a následně rekonstruovat jejich strukturu. K dispozici je množství existujících protokolů pro přenos zpráv a serializaci strukturovaných dat nebo se můžeme pokusit implementovat protokol vlastní.

5.2.1 Metajazyk XML

V poslední době se velice rozšířilo využití metajazyku XML jako nástroje pro organizaci uživatelských dat a serializaci strukturovaných dat. Díky jeho standardizaci a široké podpoře napříč platformami odpadají problémy s kompatibilitou. Proud příchozích dat je průběžně zpracováván XML analyzátorem, který nazýváme *parser*. Zprávy aplikačního protokolu jsou tvořeny XML elementy a struktura aplikačních dat je reprezentována vzájemným vnořením těchto elementů a jejich atributy. Jazyk XML pracuje s následujícími pojmy:

Tag (značka) - vymezuje elementy v dokumentu.

Element - logická komponenta dokumentu, může obsahovat vnořené elementy.

Obsah elementu - data mezi otevíracím a zavíracím tagem.

Atribut - popisuje vlastnost elementu.

Entita - pojmenovaná posloupnost znaků.

Pro účely aplikačního protokolu využíváme strukturálních vlastností jazyku XML. Z pohledu modelu ISO/OSI lze proto XML považovat za prezentační vrstvu (6. vrstva), protože tvoří kontext mezi entitami aplikačními vrstvy a jejich reprezentací použitou v nižších vrstvách. Obecný XML dokument má následující strukturu:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<root>
  <element_1>data</element_1>
  <element_2 atribut="true">
    <element_3>data</element_3>
  </element_2>
</root>
```

Při identifikaci zprávy je tedy nutné v proudu dat nalézt otevírací a uzavírací tag elementu. V podstatě se jedná o parsování klasického XML dokumentu s tím rozdílem, že nevíme, jak je velký a kde je jeho konec. Při čtení posloupnosti bajtů z proudu mohou bajty přicházet nepravidelně v různém množství. Proto je nutné parsování realizovat postupně pomocí tzv. SAX parseru, který postupuje sekvenčně podle aktuálního množství příchozích bajtů. Tomuto stylu se říká progresivní parsování nebo také *push* parsování a jeho hlavní výhodou je, že dokáže pracovat s dokumenty, které nemůžeme celé v jednom okamžiku udržovat v paměti. Pokud je množství přijatých bajtů dostatečné a parser rozpozná některou XML strukturu, pak generuje příslušnou událost. Pro každou takovou událost musíme v parseru zaregistrovat obslužnou rutinu, tzv. *handler*, ve které jsme schopni danou událost zpracovat, extrahovat přijatá data a vracet je do dalších částí aplikace.

Jazyk XML rozlišuje malá a velká písmena (*case-sensitive*) a elementy se nesmějí překrývat. Validní XML dokument může obsahovat pouze jediný kořenový element (*root element*). Proto obvykle umísťujeme XML zprávy do nějakého kořenového elementu a základní úroveň komunikace se pohybuje v hloubce vnoření $h = 1$. Kořenový element představuje hloubku vnoření $h = 0$. V případě, že by element zprávy byl zároveň kořenovým elementem, pak by přijetí jeho ukončovacího tagu znamenalo ukončení i celého dokumentu a restartování parseru, protože přijetí další zprávy by vygenerovalo chybu způsobenou výskytem elementu za koncem dokumentu. V komunikaci server/klient znamená přijetí ukončovacího tagu kořenového elementu, že odesílající strana ukončuje spojení.

XML dokument a tedy i aplikační protokol je určen *deklarací dokumentu*, která předepisuje typ dokumentu. Dále vymezuje množinu přípustných elementů, atributů a obsahů elementů. Deklarace definuje strukturu dokumentu a slouží k jeho validaci. Pro zápis pravidel struktury dokumentu slouží jazyk DTD, který obsahuje deklarace elementů, atributů, entit a notací. Definiční soubor komunikačního protokolu je součástí přílohy.

5.2.2 Knihovna Expat

Knihovna Expat je proudově orientovaný nízkoúrovňový, a proto velice rychlý a paměťově nenáročný XML parser, který je napsán v jazyce C a je dostupný[4] jako *open source*. Protože knihovna Expat není závislá na dalších knihovnách, je její použití v programech psaných v C/C++ velice jednoduché. Knihovna sama o sobě nesplňuje specifikaci SAX

[4] http://expat.sourceforge.net/

parseru, protože neimplementuje některé pokročilé vlastnosti XML. Nicméně pro zpracování elementů komunikačního protokolu je plně dostačující a mnoho projektů ji využívá jako základ pro vytvoření parseru typu SAX nebo DOM. Expat tvoří základ pro progresivní přístup a obsluhu událostí pomocí rutin. Přístup k funkcím knihovny Expat ze zdrojového textu serveru zajistíme vložením odkazu na hlavičkový soubor `expat.h` a parametrem pro linkovací program, kterým ho informujeme, že má přilinkovat knihovnu Expat. Již nám nic nebrání v deklarování objektu parseru a vytvoření jeho instance pomocí funkce `XML_ParserCreate()`:

```
pbserver_LIBS = pthread expat

#include <expat.h>
XML_Parser parser = XML_ParserCreate(NULL);
```

Poté musíme připravit sadu funkcí, které budou obsluhovat události generované parserem během zpracovávání proudu dat. Funkce `startElement()` je volána pokaždé, když parser nalezne otevírací tag. Obdobně funkce `endElement()` je reakcí na uzavírací tag. Funkce `handleCharacterData()` zpracovává data, která se vyskytují mezi tagy. Jejich prototypy vypadají takto:

```
void startElement(void *data, const char *el, const char **attr);
void endElement(void *data, const char *el);
void handleCharacterData(void *data, const XML_Char *buffer, int len);
```

Důležitým parametrem funkcí je ukazatel na void `*data`. Pomocí něj je funkcím dostupný objekt, skrz který si mohou vyměňovat data, např. hloubku zanoření ve struktuře dokumentu. Tento objekt v parseru registrujeme pomocí funkce:

```
void XML_SetUserData(XML_Parser p, void *userData);
```

V dalším kroku registrujeme události. Voláním funkce `XML_SetElementHandler()` přiřadíme funkci `startElement()`, resp. `endElement()` jako obsluhu k výskytu otevírajícího resp. uzavírajícího tagu. Obdobně funkce `XML_SetCharacterDataHandler()` provede registraci funkce `handleCharacterData()` jako obsluhy výskytu dat mezi elementy:

```
XML_SetUserData(parser, &fcn_data);
XML_SetElementHandler(parser, startElement, endElement);
XML_SetCharacterDataHandler(parser, handleCharacterData);
```

Nyní je parser korektně nakonfigurován a připraven ke zpracování proudu dat. Po každém načtení dat z přijímacího bufferu je zavolána funkce `XML_Parser()`, jejímž parametrem je ukazatel na objekt parseru, ukazatel na buffer obsahující přijatá data, délka dat v bufferu a celočíselná hodnota informující parser, zda se jedná o poslední část dokumentu.

```
int XML_Parse(XML_Parser p, const char *s, int len, int isFinal);
```

V programu je vhodné kontrolovat, zda při zpracování bufferu nedošlo k nějaké chybě, a při ukončení komunikace uvolnit paměť zrušením instance parseru (zavoláním funkce `XML_ParserFree()`):

```
while ((len = read(sock_fd, rcvbuff, BUFFER_LENGTH -1)) > 0) {
  rcvbuff[length] = '\0';

  if (! XML_Parse(parser, rcvbuff, len, done)) {
    log_message(0, "Parse error at line %d: %s \n",
              (int) XML_GetCurrentLineNumber(parser),
              XML_ErrorString(XML_GetErrorCode(parser)));
    exit(-1);
  }
  if (args->shutdown > 0) {
    XML_ParserFree(parser);
    break;
  }
}
```

5.2.3 Komunikační protokol

Komunikace je založena na výměně zpráv, které jsou reprezentovány XML elementy. Množina jmen elementů tvoří slovník protokolu. Jednotlivá jména představují komunikační performativy. Vývoj komunikace mezi serverem a klientem postupně tvoří XML dokument, který obsahuje vyměňované zprávy jako své elementy. Protože je komunikace obousměrná, vznikají postupně dva XML dokumenty. Jeden dokument představuje komunikaci od serveru ke klientovi a druhý od klienta k serveru. V aplikaci klient/server navazuje klient TCP spojení a server jej přijímá. Mechanismus navázání TCP spojení (4. vrstva) a jeho implementace byly popsány v oddíle 5.1. Po navázání TCP spojení

může být zahájena vlastní výměna zpráv aplikačního protokolu, která musí začít na obou stranách otevřením XML dokumentu. Začátek XML dokumentu je deklarován odesláním hlavičky XML dokumentu, ve které je specifikováno kódování přenášených dat:

```
<?xml version="1.0" encoding="utf-8"?>
```

Protokol předpokládá, že uživatel může požadovat v rámci jedné relace opakované čtení dat s různým nastavením. Proto lze cyklicky přecházet mezi blokem konfigurace a blokem přenosu dat. Komunikaci mezi klientskou aplikací a serverem můžeme rozdělit na následující části:

1. navázání spojení
2. výměnu konfiguračních parametrů
3. vlastní přenos uživatelských dat
4. ukončení spojení.

Všechny komunikační performativy protokolu začínají pro zvýšení jednoznačnosti prefixem „pb“, kromě performativu `<d>`. V tabulce 5.1 jsou uvedeny performativy, jejich význam a odesílající strana. Kořenový element je `pbanalyzer`, který obsahuje vnořené elementy reprezentující příslušné zprávy a přechody mezi jednotlivými stavy komunikace.

Atributy, které přísluší k některým elementům, jsou uvedeny v tabulce 5.2. Syntaxe komunikačního protokolu je definována DTD souborem, který lze najít na přiloženém CD (viz Příloha C).

Atribut `ver` specifikuje verzi komunikačního protokolu. Přípustná je reálná hodnota v rozsahu $1.0 \div 9.9$ tvořená jedním celým a jedním desetinným místem odděleným tečkou. Atribut `id` slouží k identifikaci účastníků komunikace (pro budoucí využití). Server má vždy `id=0`. Klientské aplikace mohou mít přidělen identifikátor `id` v rozsahu celočíselných hodnot $1 \div 10$ podle priority (1 - nejvyšší priorita).

5.2.4 Sémantika komunikačního protokolu

Sémantika komunikačního protokolu předepisuje, jaké pořadí zpráv musí být během komunikace dodrženo a kdy které zprávy mohou být odeslány. Pokud dojde během komunikace k výskytu zprávy, která není performativem protokolu, protistrana na ni reaguje obecnou chybovou zprávou `pberr` a ukončí spojení.

Performativ	Párový	Atribut	Strana	Význam
pbanalyzer	ano	-	S, K	zahájení a ukončení komunikace
pbclient	ne	ver, id	K	klient se identifikuje serveru
pbserver	ne	ver, id	S	odpověď na identifikaci klienta
pbconfig	ano	-	S, K	označení bloku konfigurace
pbset	ne	více	K	požadavek nastavení serveru
pbget	ne	více	K	požadavek vrácení hodnoty nastavení
pbans	ne	více	S	odpověď serveru na pbset a pbget
pberr	ne	více	S	chybová odpověď na pbset a pbget
pbdata	ano	count, time	S, K	označení bloku uživatelských dat
d	ano	-	S	označení rámce uživatelských dat
pbdatastop	ne	-	K	požadavek na ukončení odesílání dat

Tabulka 5.1: Performativy komunikačního protokolu

Zahájení komunikace

Klientská aplikace iniciuje komunikaci odesláním XML hlavičky a zprávy `pbclient`. Verze komunikačního protokolu klienta a serveru `ver` se musejí shodovat. Naopak identifikátor `id` musí mít každá aplikace unikátní. V případě výskytu chyby odpoví server zprávou `pberr` doplněnou atributem `reason` obsahujícím název kolidujícího atributu. Zpráva `pberr` je opakována pro každý chybný atribut jednotlivě:

```
<pberr reason="id"/>
```

Server potvrzuje účast na komunikaci odesláním zprávy `pbserver` doplněné o svoje identifikátory `id` a `ver`. Následně klient otevírá konfigurační blok zasláním zprávy `pbconfig`. Server odpovídá rovněž zprávou `pbconfig`. Opuštění konfiguračního bloku může iniciovat pouze klient uzavřením elementu `pbserver`. Server reaguje uzavřením elementu na svojí straně. Klient může kdykoliv znovu otevřít konfigurační blok opětovným odesláním zprávy `pbconfig`.

Konfigurační blok

V konfiguračním bloku probíhá komunikace stylem žádost/potvrzení nebo případně stylem otázka/odpověď s využitím zpráv `pbset`, `pbget` a `pberr`. Jiné zprávy nejsou povoleny.

Atribut	Element	Význam
baudrate	pbset, pbget, pbans, pberr	přenosová rychlost Profibusu
count	pbdata	množství zpráv požadované klientem
device	pbset, pbget, pbans, pberr	volba zařízení ovladače
filteron	pbset, pbget, pbans, pberr	nastavení filtru
id	pbclient, pbserver	identifikátor serveru, klienta
reason	pberr	vrací název neplatného atributu
standalone	pbset, pbget, pbans, pberr	zapnutí samostatného režimu
time	pbdata	počáteční čas záznamu
triggeron	pbset, pbget, pbans, pberr	nastavení pretriggeringu
tsreset	pbset, pbget, pbans, pberr	nastavení resetu časové značky
ver	pbclient, pbserver	verze komunikačního protokolu, nyní 1.0

Tabulka 5.2: Význam atributů

Klient může požadovat nastavení některého parametru serveru zasláním zprávy `pbset` doplněné atributem a jeho hodnotou. Server potvrzuje provedení nastavení zasláním zprávy `pbans`, která obsahuje nastavené parametry a jejich aktuální hodnotu. Pokud klient zašle požadavek na nastavení neexistujícího parametru nebo požaduje nastavení parametru na neplatnou hodnotu, server odpoví zprávou `pberr`. Obdobně pomocí zprávy `pbget` klient požaduje vrácení hodnoty některého parametru serveru. Kromě atributu `reason` mohou ostatní atributy nabývat pouze číselných hodnot. U atributů očekávajících nastavení zapnout/vypnout, ano/ne jsou přípustné hodnoty „0“ pro vypnutí a „1“ pro zapnutí volby.

Příklad komunikace v konfiguračním bloku

Korektní chování klienta a serveru:

```
KLIENT=> <pbset baudrate="187500"/>
SERVER=> <pbans baudrate="187500"/>
KLIENT=> <pbget baudrate=""/>
SERVER=> <pbans baudrate="187500"/>
```

a v případě chybného požadavku:

```
KLIENT=> <pbset baudrate="784333"/>
SERVER=> <pberr reason="baudrate"/>
```

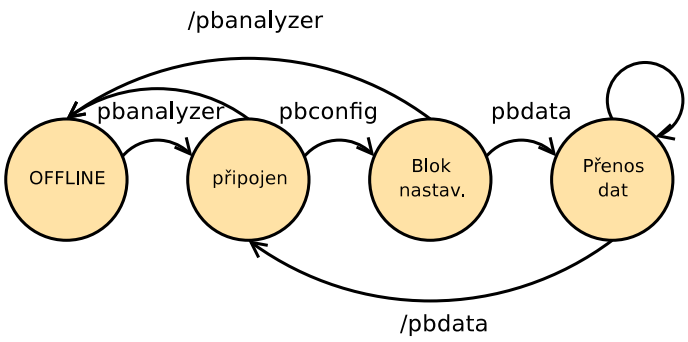
Přenos uživatelských dat

Server začne odesílat data zachycená z Profibusu na žádost klientské aplikace. Klient otevírá datový blok odesláním zprávy `pbdata`, která může být doplněna atributem `count` specifikujícím počet rámců Profibusu DP, který má být zachycen. Přípustná je celočíselná hodnota větší než nula. Pokud je atribut `count` nulový nebo vynechán, server posílá data, dokud od klienta neobdrží zprávu `pbdatastop`. Otevření datového bloku potvrdí server zasláním zprávy `pbdata` obsahující atribut `time`, který obsahuje počáteční čas záznamu dat. Po odeslání požadovaného množství rámců nebo po příchodu zprávy `pbdatastop` server uzavře datový blok uzavírající zprávou `pbdata`. Datové rámce jsou přenášeny uvnitř elementu `<d>`. V datovém bloku nejsou povoleny jiné zprávy než `pdatastop` a `d`. Po opuštění datového bloku může klient zažádat o přechod do konfiguračního bloku, znovuotevření datového bloku nebo ukončení komunikace a uzavření relace se serverem.

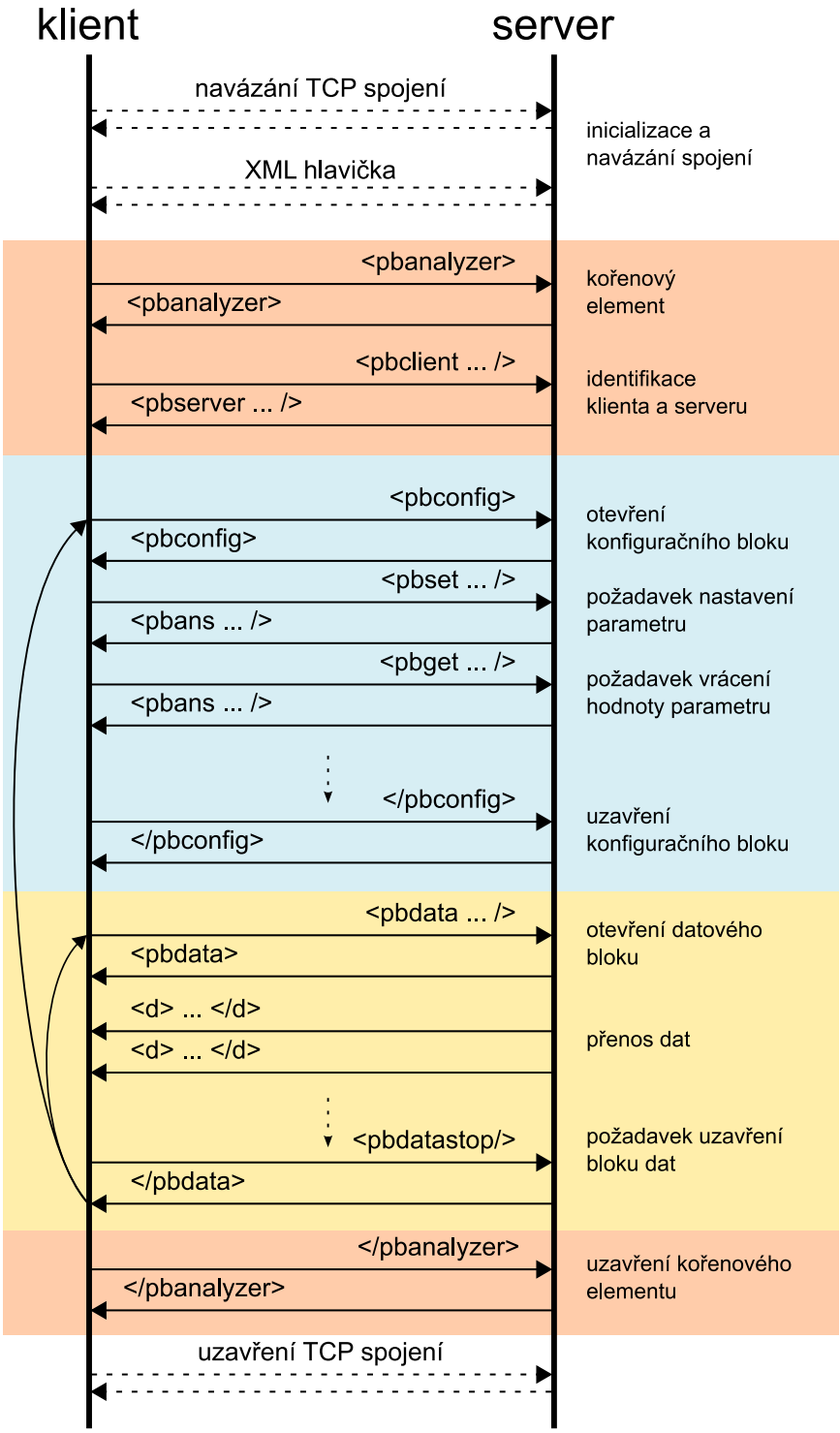
Ukončení komunikace

Klient může kdykoliv jednostranně požádat o ukončení komunikace uzavřením kořenového elementu `pbanalyzer` ve svém dokumentu. Server smí ukončit spojení pouze v případě výskytu chyby a to odesláním zprávy `pberr` a uzavřením aktuálně otevřených elementů včetně kořenového elementu. Důvodem ukončení spojení může být příjem špatně formátované zprávy, zprávy s nevyhovující syntaxí, s nepovolenými hodnotami atributů nebo z důvodů nějaké nedefinované vnější události.

5.2.5 Vizualizace činnosti komunikačního protokolu



Obrázek 5.2: Stavy a přechody komunikačního protokolu



Obrázek 5.3: Průběh ideální komunikace mezi serverem a klientem v čase

Kapitola 6

Předzpracování dat na straně serveru

6.1 Formát dat vhodný pro přenos soketem

Výchozí představa komunikačního protokolu počítala s přenosem binárních dat přímo ve struktuře XML dokumentu. Binárními daty rozumíme posloupnost oktetů, každý v rozsahu dekadických hodnot $0 \div 255$. Standard jazyka XML definuje pro vymezení nestrukturovaných dat v rámci XML dokumentu speciální element CDATA, který začíná značkou `<![CDATA[` a je ukončen značkou `]]>`. Parser ignoruje výskyt XML elementů v nestrukturovaných datech. Pokud ale při čtení nalezne některý řídicí znak (`<, >, &, %`), nahradí ho příslušnou XML entitou. Tato vlastnost je nepřijatelná pro přenášení binárních dat vnořených v elementu CDATA. Navíc XML standard připouští pouze *tisknutelné znaky* v elementu CDATA. Z těchto příčin nelze přenášet binární data prostě vložená do XML dokumentu.

Při hledání vhodného řešení problému přenosu binární dat jsem postupně rozpracoval čtyři alternativy, které byly posuzovány z hlediska objemu přenášených dat, výpočetní náročnosti, dopadu na stávající kód aplikací a složitosti implementace. Každá alternativa byla konzultována s vedoucím práce a autorem související diplomové práce.

Hexadecimální reprezentace dat spočívá v převodu binárních dat na hexadecimální reprezentaci. Každý bajt je reprezentován dvojicí číslic (0-F). Takto formátovaná data lze přenášet uvnitř elementu CDATA. Implementace je velmi snadno realizovatelná a výpočetní složitost je zanedbatelná. Nevýhodou tohoto řešení je dvojnásobný objem dat, který by bylo potřeba přenést soketem.

Kódování Base64 je datový formát zobrazující binární data jako posloupnost tisknutelných ASCII znaků. Znaky jsou 7-bitové zarovnané na 8 bitů, rozsah hodnot $0 \div 127$. Objem přenášených dat vzroste o 33%. Nasazení ve stávajícím programu je snadné. Příslušné funkce jsou k dispozici ve standardních knihovnách. Náročnost výpočtu není omezující.

Odpojení XML parseru je opatření prováděné na straně klientské aplikace. Po příchodu daného elementu je XML parser odpojen od proudu příchozích dat, která jsou pak k dispozici v nezměněné podobě. V praktickém nasazení jsme nedokázali vyřešit udržení spojení po odpojení parseru, vyprázdnění jeho vyrovnávací paměti před opětovným připojením a restart parseru. Toto řešení nezvětšuje objem přenášených dat přídavným kódováním, výpočetní složitost je nulová, ale implementace se ukázala být velmi problematická.

Datový soket je běžný soket sloužící pouze k přenosu binárních dat. Aplikace předpokládá vytvoření druhého soketu vedle již existujícího. Stávající soket a protokol definovaný v oddíle 5.2.3 pak slouží pouze k signalizaci a konfiguraci. Jde o optimální řešení z hlediska objemu přenášených dat. Implementace v již existujícím kódu serveru a kódu klientské aplikace je problematická, protože vyžaduje přidání dalších vláken a paralelní zpracování událostí, což představuje rozsáhlé změny v již hotových částech. V budoucím vývoji je třeba počítat s nasazením této alternativy.

Po uvážení konkrétních výhod a nevýhod možných řešení jsme zvolili implementačně jednodušší metodu spočívající v kódování dat Base64. Data v tomto formátu jsou přenášena přímo uvnitř elementu `<d>`, viz 5.2.3.

6.2 Kódování Base64

Base64 je kódovací algoritmus, který umožňuje obecná binární data zakódovat takovým způsobem, že výsledek je tvořen pouze běžnými tisknutelnými znaky z první poloviny ASCII tabulky. Kódová abeceda obsahuje 64 tisknutelných znaků. Jde tedy o velká (A-Z), malá (a-z) písmena, číslice (0-9) a znaky „+“, „/“. Speciální význam má znak „=“, který slouží k zarovnání zakódovaného řetězce tak, aby jeho délka byla vždy dělitelná čtyřmi beze zbytku. Tento způsob kódování je vhodný pro vkládání binárních dat do XML dokumentu nebo emailových zpráv posílaných protokolem SMTP. Pomocí Base64

kódovaní lze také přenášet binární data skrz systémy pracující pouze se 7-bitovými znaky. XML dokument nesmí obsahovat prvních 33 znaků ASCII tabulky (speciální a řídicí, netisknutelné znaky). Proto je nezbytně nutné zachycená binární data před odesláním síťovým soketem zakódovat. Kódování ochrání data proti nechtěné substituci vyhrazených XML znaků pomocí XML entit, a tak zabrání znehodnocení dat a chybovým stavům parseru.

Princip kódování Base64 je v podstatě velmi jednoduchý. Algoritmus postupně zpracovává vstupní proud dat po trojicích znaků. Každá trojice je převedena na čtveřici znaků kódové abecedy. Tabulka kódové abecedy je indexována od nuly:

`ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/`

1. Do vstupního bufferu o velikosti 24 bitů jsou uloženy tři bajty ze vstupu. Jsou-li k dispozici méně než tři bajty, je zbytek bufferu vyplněn nulami.
2. Buffer je rozdělen na čtyři bloky po 6-ti bitech (6 bitů poskytuje právě 64 různých hodnot).
3. Dekadická hodnota každého bloku je použita jako klíč pro nalezení znaku v kódové abecedě. Názorně viz obrázek 6.1.
4. Na výstup jsou zapsány čtyři znaky z kódové abecedy.

0xDC								0x02								0x26								Vstupní data
1	1	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1	0	24-bitový buffer
55						0						8				38								Index do kódovací tabulky
3						A						H				m								Base64 reprezentace

Obrázek 6.1: Příklad převodu tokenu do Base64 reprezentace

Algoritmus je opakován pro všechny zbývající trojice bajtů ze vstupního proudu. Pokud na vstupu zbývají právě 3 bajty, jsou zpracovány normálním způsobem. Jsou-li k dispozici pouze dva bajty, je zbytek bufferu doplněn nulami, proveden převod, a na výstup přidán jeden znak „=“. V případě, že zbývá na vstupu pouze jeden bajt, jsou na výstup doplněny znaky „=“ dva. Tyto znaky signalizují dekodéru, kolik nulových bitů

bylo přidáno v důsledku zarovnání bufferu na 24 bitů, a mají být odstraněny z rekonstruovaných dat. Princip doplňování znaku „=“ zajišťuje, že Base64 kódovaný řetězec bude mít vždy délku dělitelnou čtyřmi. Tuto vlastnost lze využít také jako velmi jednoduchou kontrolu integrity dat.

Implementace algoritmu pro kódování a dekódování Base64 je součástí balíčku GNU Coreutils, který je nezbytnou součástí každé linuxové instalace. Aktuální zdrojové kódy lze nalézt na domovské stránce projektu[1]. Vývojovou verzi balíčku lze získat také pomocí systému pro správu verzí Git:

```
git clone git://git.sv.gnu.org/coreutils
```

Nejdůležitější jsou soubory `base64.h` a `base64.c`, které jsou součástí zdrojového kódu PBServeru jako statická knihovna. Potřebné funkce jsou nadefinovány v hlavičkovém souboru:

```
#include <base64.h>
```

Před použitím je vhodné ověřit, že implementace Base64 v serverové a klientské aplikaci používají stejnou kódovací tabulku. V současnosti Coreutils i Microsoft Visual Studio 2005 používá standardní kódovací tabulku.

Z pohledu serveru jsou zajímavé pouze funkce, které provádějí kódování dat (dekódování probíhá v klientské aplikaci). Samotný proces kódování je implementován ve funkci `base64_encode()`, která očekává čtyři parametry:

```
void base64_encode(const char *in, size_t inlen,
                   char *out, size_t outlen)
```

Parametr `*in` je ukazatel na řetězec obsahující data připravená k zakódování a parametr `inlen` je délka dat. Třetí parametr je ukazatel na alokovanou paměť, která musí být dostatečně velká, aby se do ní zakódovaný řetězec vešel celý. V případě, že není alokovaná paměť dostatečná, je uložena jen příslušná část řetězce. Poslední parametr je ukazatel na proměnnou obsahující velikost alokované paměti. Délku řetězce po zakódování lze přibližně určit podle vzorce

$$l_{out} = \frac{4(l_{in} + 2)}{3} \,. \tag{6.1}$$

Programově je bezpečnější během výpočtu na prvním místě dělit. Sníží se tak riziko přetečení výsledku:

[1] http://www.gnu.org/software/coreutils/

```
size_t outlen = 4 * ((inlen + 2) / 3);
```

Knihovna base64 obsahuje také funkci s názvem `base64_encode_alloc()`, která na rozdíl od funkce `base64_encode()` provede při každém volání inicializaci dostatečného množství paměti. Její prototyp má tvar:

```
size_t base64_encode_alloc(const char *in, size_t inlen, char **out)
```

První a druhý parametr mají stejný význam jako u funkce `base64_encode()`. Třetí parametr je ukazatel na proměnou, do které bude po úspěšném vykonání funkce uložen ukazatel na nově alokovanou paměť obsahující zakódovaná data. Uvolnění paměti je nutné ošetřit samostatně. V případě úspěšného vykonání funkce obsahuje návratová hodnota délku zakódovaných dat. Pokud došlo k přetečení paměti, vrací funkce `0` a proměnná `out` je nastavena na `NULL`. Došlo-li k chybě při přidělování paměti (*malloc*), obsahuje návratová hodnota funkce velikost požadovaného bloku paměti. Funkce `base64_encode_alloc()` je v podstatě zapouzdřením funkce `base64_encode()`, která realizuje samotné zakódování dat.

Přestože funkce `base64_encode_alloc()` nabízí vyšší programátorský komfort než funkce `base64_encode()`, není vhodná pro online kódování dat. Přidělení a uvolnění paměti je časově náročná operace a její časté opakování znamená neefektivní zacházení s procesorovým časem. To může mít za následek ztrátu dat při měření na vyšších komunikačních rychlostech Profibusu DP. Tato funkce najde uplatnění zejména tam, kde dochází k volání funkce a potažmo alokaci paměti jednorázově nebo v malé míře (např. kódování přílohy emailové zprávy).

Při cyklickém zpracování dat je efektivnější používat pro jejich uložení stále stejný paměťový prostor a vyvarovat se tak časté alokaci a uvolňování paměti. V základním nastavení má PBServer k dispozici 2048 B velký buffer sloužící pro načítání dat. Abychom mohli použít kódovací funkci `base64_encode()`, potřebujeme alokovat dostatečně velkou paměť pro zakódovaná data. Protože známe maximální velikost bloku dat, která mohou být při jednom volání funkce `read()` načtena, lze velikost potřebné paměti vypočítat pomocí vzorce 6.1. Pro 2048 B velký vstupní buffer je potřeba alokovat výstupní buffer o velikosti alespoň 2734 B.

```
#define RCVBFR_LEN  2048
size_t outlen = 4 * ((RCVBFR_LEN + 2) / 3);
unsigned char *frm_buffer = (unsigned char *) malloc(RCVBFR_LEN);
unsigned char *enc_buffer = (unsigned char *) malloc(outlen);
```

Zakódovaná data, která jsou uložena v paměti odkazované ukazatelem `*enc_buffer`, mohou být přímo vložena do elementu `<d>` a zapsána do soketu pomocí funkce `write()`. Po odeslání je alokovaná paměť znovu použita pro uložení dalšího bloku kódovaných dat. Praktické měření ukázalo, že při použití funkce `base64_encode_alloc()` začalo docházet ke ztrátě dat již při rychlosti Profibusu DP 187, 5 kbps.

6.3 Lokální ukládání dat

Lokální ukládání dat je aktivováno položkou „standalone“ v konfiguračním souboru nebo příkazem komunikačního protokolu (viz oddíl 5.2.3). V případě, že je ukládání zvoleno konfiguračním souborem, začne server data ukládat okamžitě po provedení inicializace a zpracování konfigurace. Je-li samostatný režim aktivován klientskou aplikací, ukládání je zahájeno ihned po odpojení klientské aplikace. K ukončení ukládání dat dojde při opětovném připojení klientské aplikace. Ukládání je automaticky ukončeno, pokud velikost souboru na paměťovém médiu přesáhne 4 GB, nebo na paměťovém médiu zbývá méně něž 50 MB volného místa.

Data jsou do souboru ukládána v surové podobě v blocích o velikosti maximálně 100 kB. Jednotlivé bloky jsou v souboru odděleny řetězcem (tagem) `<d>`. Oproti komunikačnímu protokolu 5.2.3 je vynechán párový uzavírací řetězec `</d>`. Na začátku souboru je uložena hlavička, která obsahuje datum, čas a komunikační rychlost sběrnice Profibus DP, na které byla data zachycena. Časová informace udává začátek záznamu dat a časové známky uložené v datech jsou vztaženy k této hodnotě. Datum a čas jsou uloženy v souboru jako textový řetězec o délce 14 znaků. Formát data je: „YYYYmmddHHMMSS“. Za datumem následuje číselná hodnota o velikosti 4 B v pořadí bajtů little-endian (datový typ `uint32_t`) udávající rychlost. Na 19. bajtu začíná otevírací řetězec `<d>`, za kterým následuje první blok dat, který je ukončen před dalším výskytem `<d>`. Bloky jsou takto postupně ukládány za sebe. Poslední blok dat v souboru je ukončen řetězcem `</d>`. Za ním je uložena číselná hodnota o délce 4 B (datový typ `uint32_t`) udávající kolik rámců bylo do souboru celkem uloženo. Vnitřní struktura souboru je dobře patrná z následujícího schématu:

```
[ 14B datum ][ 4B rychlost ]<d> ... <d> ... <d> ... </d>[ 4B počet]
```

Formát zápisu dat do souboru byl úmyslně zvolen tak, aby byl pokud možno velmi jednoduchý a příliš neměnil strukturu dat. Tento předpoklad umožňuje použít pro čtení

dat ze sběrnice i ze souboru stejný mechanismus a dokonce i funkci `busread_thread()`, která je vykonávána vláknem `bus_thread`. Z pohledu uživatelské aplikace je čtení dat ze sběrnice prostřednictvím souboru zařízení `/dev/pbmaster0` a zápis/čtení dat do/ze souboru `/tmp/busdump01.bin` stejnou operací. Přesuny dat jsou vždy prováděny pomocí elementárních funkcí `read()` a `write()`, které pracují s file deskriptory daných souborů. Funkce `read()` a `write()` jsou definovány v hlavičkovém souboru `unistd.h`. File deskriptor je získán návratovou hodnotou funkce `open()`. Funkce očekává jako parametr textový řetězec obsahující jméno souboru na disku, hodnotu určující režim práce se souborem a určení výchozích přístupových práv. Jednotlivé konstanty `O_*` a `S_*` jsou vyhodnoceny jako logický součet. Význam jednotlivých konstant lze dohledat v [8] na straně 88, nebo v manuálových stránkách[2]:

```
int fd = open(data_file, O_CREAT|O_RDWR|O_TRUNC, S_IRUSR|S_IWUSR);
```

Při zpracování dat ze souboru jsou jednotlivé bloky, které jsou odděleny řetězcem `<d>`, načteny do odesílacího bufferu `encbfr` a následně zakódovány pomocí Base64 kódování, viz oddíl 6.2. Zakódovaná data jsou odeslána klientovi stejným mechanismem jako v případě online dat čtených ze sběrnice. Oddělovací řetězec je při čtení dat průběžně přeskakován. Zakódovaný blok dat je uzavřen mezi tagy `<d>`, `</d>` a odeslán jako součást XML dokumentu.

6.4 Filtrace dat na straně serveru

Základní metody filtrování zachycených dat jsou založeny na online jednoprůchodovém zpracování proudu dat dodávaného komunikačním ovladačem. Online filtrace může být s ohledem na zatížení procesoru prováděna pouze jednoprůchodově, protože při komunikačních rychlostech Profibusu DB nad 187.5 kbps by začalo docházet ke ztrátě rámců. Aktuální blok rámců je vždy načten do vyrovnávacího bufferu `frm_bfr` a ukazatel na tuto paměť je předán funkci, která provádí samotnou interpretaci dat a rozhodne o odfiltrování určitého rámce. Korektní zacházení s ukazateli a vyvarování se nadbytečného kopírování dat v paměti je nezbytné s ohledem na zatížení procesoru. Současná implementace filtru je velmi nedokonalá jak z pohledu efektivity využívání procesorového času, tak z pohledu vlastností rámců, na jejichž základě se dají rámce filtrovat.

[2] V Linuxu příkazem: man 2 open

Základní nastavení filtrace spočívá v určení základní politiky, jak filtr globálně zachází s rámci. V režimu „výběrového filtru“ jsou implicitně odfiltrovány všechny rámce vyjma těch, které vyhoví filtrovacím pravidlům, která jsou explicitně nadefinována. Oproti tomu v režimu „vylučovacího filtru“ jsou implicitně propouštěny všechny rámce a je zapotřebí pomocí pravidel explicitně definovat, které rámce mají být odfiltrovány. Filtrovací pravidla jsou definována jako „if“ podmínky, které mohou být aktivovány nebo deaktivovány.

Filtrovací funkce `pbblock_filter()` je aplikována na každý blok rámců, ve kterém zpracovává úvodní hlavičky ovladače PBMaster (viz 4.6). Z pole LF zjistí funkce délku Profibus DP rámce, který následuje za hlavičkou. Tato hodnota je důležitá pro nalezení začátku hlavičky dalšího rámce. Celá hlavička je uložena do dočasné proměnné `ts_temp`. V případě, že Profibus DP rámec je odfiltrován, odpovídá funkce `pbblock_filter()` za smazání hlavičky tak, aby se neobjevila ve výstupním bufferu.

Zpracování samotného Profibus DP rámce probíhá v těle funkce `pbframe_filter()`, které předá funkce `pbblock_filter()` ukazatel na první znak. Funkce `pbframe_filter()` provede vyhodnocení Profibus DP rámce vůči filtrovacím pravidlům. Její návratovou hodnotou je ukazatel na znak v bloku rámců. Zpracování bloku pokračuje od znaku daného ukazatelem. Pokud má být rámec odfiltrován, vrátí funkce ukazatel na poslední znak v rámci a předá zpracování zpět funkci `pbblock_filter()`, která smaže jeho hlavičku (uloženou dočasně v proměnné `ts_temp`) a pokračuje ve vyhodnocování bloku rámců od daného ukazatele. Nemá-li být rámec odfiltrován, vrátí funkce `pbframe_filter()` ukazatel na jeho začátek. Profibus DP rámec tak dostane zpět svoji hlavičku a je překopírován do výstupního bufferu. Celý proces je opakován, dokud funkce `pbblock_filter()` nedosáhne konce bloku rámců.

Funkce `pbframe_filter()` rozlišuje rámce hledáním jednoho ze znaků SD1, SD2, SD3, SD4 a SC. Rámce jsou ukončeny znakem ED. Výjimku z tohoto pravidla tvoří rámec tokenu a „*Single Character*“. Tento nedostatek je vyřešen známou fixní délkou rámce tokenu a SC. U rámců s proměnnou délkou uživatelských dat SD2 je jejich délka získána pomocí údaje o délce dat, který je uložen v bajtu LE a LEr (viz 3.1.2). Teprve načtením zopakovaného znaku SD2 začíná zpracování samotného rámce. Protože hodnoty řídicích znaků mohou kolidovat s uživatelskými daty, která jsou obsažena v rámci, lze konec rámce jednoznačně určit pouze z hodnoty LE.

Současná implementace filtrů neumožňuje filtrovat jednotlivé rámce na základě vlastností, které vyplývají z rozšířené adresy (*Address Extension*) a významu bitů v bajtu FC (*Frame Control*). Rozšířená adresa je přítomna, pokud je nejvyšší (7. bit) v bajtu SA nebo DA nastaven na log.1. Rámce lze na základě hodnoty 6. bitu v bajtu FC filtrovat

podle typu, zda jde o dotaz nebo odpověď. Nyní je možné filtrovat rámce podle těchto vlastností:

- typu rámce (SD1, SD2, SD3, SD4 a SC),
- filtrace všech tokenů,
- SA a DA (dle adres),
- přítomnosti rozšířené adresy,
- délky uživatelských dat (u rámců SD2),
- typu rámce Request/Response.

Pravidla vytvořená nad uvedenými vlastnostmi nelze kombinovat. Filtrovací pravidla jsou organizována ve vyhodnocovacím stromu. Hlavní vyhodnocovací strom se rozpadá na několik podstromů, kde každý z nich představuje filtrovací pravidla aplikovatelná na konkrétní druh Profibus DP rámců (viz 2.2). Podstrom každého druhu rámce se dále opět dělí na jemnější vlastnosti, podle kterých lze rámce tohoto druhu posuzovat (délka, adresové rozšíření, FC, atd.).

Filtrovací pravidla

Funkce `pbframe_filter()` zpracovává filtrovací pravidla, která aplikuje na příslušný významový bajt v Profibus DP rámci. Každé pravidlo může být součástí výběrového nebo vylučovacího filtru. Pokud rámec v režimu výběrového filtru nesplní alespoň jedno pravidlo, je zahozen. Naopak při zpracování vylučovacím filtrem je rámec zahozen, pokud vyhoví alespoň jednomu pravidlu.

Každé pravidlo je reprezentováno seznamem hodnot, které jsou uloženy v poli. Poslední prvek pole je příznak `type`, který určuje, zda filtrovací pravidlo patří do výběrového nebo vylučovacího filtru. Vyhodnocování pravidla vůči hodnotě z rámce probíhá pomocí operace XOR. Tímto způsobem je hodnota z rámce porovnána vůči všem hodnotám v seznamu daného filtru. V případě, že výsledek exkluzivního součtu je nula, bajt vyhověl pravidlu a s rámcem je naloženo podle toho, zda jde o výběrový nebo vylučovací filtr. Pokud rámec danému pravidlu nevyhověl, pokračuje funkce `pbframe_filter()` zpracováním dalšího významového bajtu/bitu, dokud nevyčerpá všechna pravidla nebo rámec nevyhoví pravidlu, které rozhodne o jeho osudu.

6.5 Detekce uživatelsky zadaných událostí

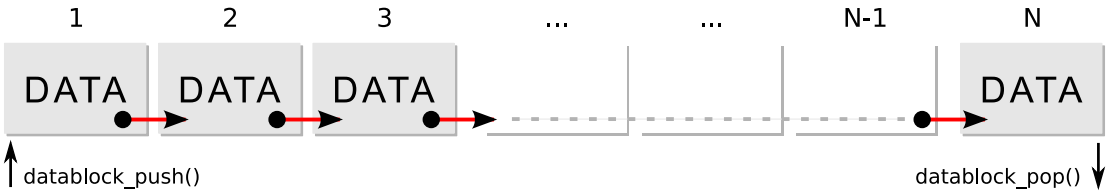
„*Pretriggering*“ je označení pro způsob zahájení záznamu dat s tím, že zároveň bude také zaznamenán dostatečně dlouhý úsek dat, který těsně předcházel události spouštějící záznam. Stejnou funkcí jsou například vybaveny osciloskopy nebo záznamová zařízení kamerových bezpečnostních systémů. Důvod, proč se pretriggering používá, je zřejmý. Pro snažší pochopení princip vysvětlím na souvislosti s kamerovým systémem.

Pokud chceme zaznamenat a vyhodnotit náhlou neočekávanou událost (výpadek stanice, vniknutí nepovolané osoby do střeženého prostoru), máme k dispozici dva možné přístupy. První možností je kontinuálně zaznamenávat všechna data (data na sběrnici, záběry z kamery) a následně je prohledat a vyhodnotit. Což může být velmi nákladné jak z hlediska skladování velkého objemu dat, tak potřeby zařízení nebo člověka provádějícího vyhodnocení. Efektivnější je druhá možnost, která je založena na definování události (výskyt zprávy, změna v obraze z kamery), která způsobí spuštění záznamu. Většinou chceme ale analyzovat děje, které těsně předcházely výskytu sledované události (výpadek napájení, vykopnutí dveří), proto je potřeba spustit záznam o něco dříve. Řešením, jak splnit tento požadavek aniž by došlo k porušení kauzality dějů, je dostatečně velká vyrovnávací paměť, která je cyklicky přepisována a v okamžiku výskytu sledované události je její obsah uložen na pevné paměťové médium. Další záznam již probíhá standarním způsobem.

Koncept implementace pretriggeringu v případě sběrnice Profibus DP je podobný filtraci, které byla popsána v předchozím oddíle. V kódu PBServeru zatím není pretriggering implementován. Následující text popisu teoreticky princip činnosti. Tyto závěry byly vytvořeny na základě konzultací a na základě zkušeností získaných při práci s komerčními produkty (viz 3.1.1). V režimu pretriggeringu server pracuje samostatně bez interakce s klientskou aplikací. Pokud dojde k výskytu sledované události, jsou příslušná data uložen na lokální paměťové médium tak, jak bylo popsáno v oddíle 6.3. Po připojení jsou tato data k dispozici ke stažení do klientské aplikace.

Nejdůležitější částí pretriggeringu je načítání dat a vyrovnávací paměť (viz obrázek 6.2), do které jsou data ukládána. PBServer průběžné čte data po blocích rámců tak, jak jsou vracena driverem. Každý blok rámců je uložen do dynamicky alokované paměti. Pro každý blok rámců je pomocí funkce `malloc()` samostatně alokována nová paměť. Pro reprezentaci uspořádání vyrovnávací paměti jsou použity struktury `data_node`, které uchovávají ukazatele na paměťové bloky. Struktury jsou za pomoci ukazatelů organizovány do jednosměrně spojového seznamu.

Funkce `datablock_push(void *datablock)` zajišťuje vytvoření nové struktury, svázání s paměťovým blokem a její vložení na začátek spojového seznamu. Další bloky jsou postupně přidávány vždy na první místo ve spojovém seznamu. Ihned po přidání nového bloku do seznamu je zavolána funkce `pbblock_filter()`, která provede rozbor rámců uložených v bloku. V případě, že je nalezen rámec, který splňuje spouštěcí pravidlo, je celá vyrovnávací paměť uložena do lokálního souboru. Data zachycená po příchodu spouštěcí události jsou do souboru ukládána přímo. Spouštěcí pravidla jsou definována stejně jako filtrovací (viz 6.4) s tím rozdílem, že mají nastaven jiný příznak `type`.



Obrázek 6.2: Princip činnosti vyrovnávací paměti

Délka spojového seznamu určuje, kolik dat bude v paměti k dispozici v okamžiku příchodu sledované události. Obecně seznam může obsahovat právě N bloků. Paměťové bloky jsou průběžně přidávány, dokud seznam nedosáhne délky N. V okamžiku, kdy je spojový seznam naplněn, přejde správa vyrovnávací paměti do druhé fáze. Před přidáním nového bloku na začátek seznamu je z jeho konce pomocí funkce `datablock_pop()` odstraněn poslední blok (časově nejstarší) a jeho paměť je uvolněna voláním funkce `free()`. Tímto mechanismem je zajištěno, že ve vyrovnávací paměti bude vždy právě N paměťových bloků.

Uživatelská konfigurace pretriggeringu spočívá v sestavení pravidla pro detekci spouštěcí události a nastavení časového intervalu T_{in} a T_{out}, (obě hodnoty se udávají v sekundách). Interval T_{in} ovlivňuje, kolik dat bude uchováváno ve vyrovnávací paměti před příchodem spouštěcí události. V programu je čas přepočten na velikost paměti v bajtech. Obdobně parametr T_{out} říká, kolik dat bude ještě uloženo do souboru po příchodu spouštěcí události.

Kapitola 7

Závěr

V rámci této diplomové práce se podařilo vytvořit serverovou aplikaci, která je článkem řetězce programů sloužících k provádění vzdálené diagnostiky sběrnice Profibus DP. Mnou zvolené řešení klade minimální požadavky na hardwarové vybavení a tedy i finance. V průběhu této práce jsem získal cenné zkušenosti týkající se programování pod operačním systémem Linux a provozu PLC od firmy Siemens.

Tato diplomová práce navázala na ovladač vytvořený v rámci bakalářské práce [9] a zároveň otevřela prostor pro další navazující práce, které by mohly rozšířit spektrum jejího využití. Velmi těsně na ni navazuje paralelně realizovaná diplomová práce, která se zabývala vývojem klientské aplikace umožňující provést analýzu zachycených dat v pohodlí grafického rozhraní Windows XP. Počáteční stádia vývoje provázely určité nesnáze spojené s častou změnou specifikace produktu. Přesto se v průběhu času podařilo dobře vymezit smysl serveru a pole jeho činnosti. Výsledkem je implementace komunikačního TCP serveru, který zajišťuje distribuci dat získaných z Profibusu DP prostřednictvím běžné IP sítě do klientských aplikací.

Koncepce PBServeru se v průběhu vývoje měnila podle toho, jakým problémům jsem čelil a jak se mi dařilo tyto problémy řešit. Volba komunikačního protokolu založeného na jazyce XML přinesla spolehlivou výměnu zpráv, ale i významná omezení týkající se přenosu binárních dat (viz 6.1). Jako dočasné řešení tohoto problému bylo zvoleno kódování Base64 (viz 6.2), které umožnilo pokračovat ve vývoji, aniž by bylo nutné zásadním způsobem přepsat již existující části kódů. Praktické experimenty však ukázaly, že toto kódovaní dat není příliš vhodné. Při komunikačních rychlostech okolo 500 kbps začalo docházet k selhávání XML parseru nebo pádům klientské aplikace. Navíc neustálé kopírování surových a zakódovaných dat v paměti serveru také není optimální. V budoucnosti by měl být tento nedostatek vyřešen použitím druhého komunikačního soketu (viz 7.1).

Spolupráce s driverem PBMaster nebyla od počátku bezproblémová. S autorem ovladače jsem postupně vyřešil problém duplicity časových známek, výpadky v přidělování časových známek a rozšíření ovladače o možnost resetování časové značky z uživatelské aplikace. Ukládání zachycených dat na lokální paměťové médium funguje spolehlivě, ale následné offline kopírování dat do uživatelské aplikace je omezeno Base64 kódováním. Filtraci dat se podařilo implementovat pouze částečně. Rozšíření filtrů o další atributy bude možné po zavedení datového soketu. Funkce sledování uživatelských události (pretriggering) zůstala z důvodu časových a technických problémů pouze ve stádiu principiálního návrhu (viz 6.5).

Ověření činnosti serveru a klientské aplikace bylo provedeno na experimentální sběrnici (viz. Příloha D). Funkčnost aplikace byla průběžně ověřována při rychlosti $187,5$ kbps. Při komunikačních rychlostech do $187,5$ kbps nedochází k výpadkům dat a celá aplikace pracuje korektně. Při vyšších rychlostech se začnou projevovat výpadky ve zpracování dat způsobené neoptimálními přesuny dat v paměti, což způsobí pád klientské aplikace. Tyto nedostatky by měly odstranit níže navržené kroky. Současná verze PBServeru není určena pro skutečné nasazení, ale poslouží jako odrazový můstek pro další diplomové práce, které se budou zabývat vývojem analyzátoru.

7.1 Co by se dalo zdokonalit?

Stávající řešení PBServeru trpí osudem prvotiny, a proto by se další vývoj aplikace měl zaměřit na odstranění některých nedostatků a omezení. Jde zejména o:

- Nahrazení neoptimálního kódování Base64 dalším komunikačním soketem sloužícím pouze k přenosu binárních dat. Dalšího zefektivnění přenosu binárních dat lze dosáhnout použitím UDP protokolu. Pro obsluhu více klientských aplikací bude účelné použít UDP multicast.

  Zajímavých výsledků by bylo možné dosáhnout aplikováním principu dvou soketů, který používá služba FTP pro přenos velkých binárních souborů. V případě služby FTP se tento způsob přenosu dat nazývá „aktivní režim“. Komunikace probíhá tak, že se klient standarním způsobem připojí k serverovému soketu, jak bylo popsáno v kapitole 5. Toto spojení slouží pouze k přenosu řídicích příkazů. V okamžiku, kdy klient požádá server o zaslání binárních dat, otevře zároveň na svojí straně serverový soket. Číslo portu, na kterém očekává připojení, odešle serveru řídicím

soketem. Server se připojí na daný port a zahájí přenos binárních dat. Po ukončení přenosu je datový soket uzavřen.

- Možnost implementovat v serveru „*Boss-Worker*“ těsně souvisí s předcházejícím bodem.
- Rozšíření možnosti filtrace o další atributy (LSAP, DSAP, Status, ...) a ukládat více datových relací na lokální paměťové médium.
- Umožnění obousměrného přenosu dat tak, aby na vzdáleném počítači mohla běžet aplikace chovající se jako Master třídy 1/2. Tato funkce vyžaduje změny v ovladači PBMaster.
- Optimalizaci kódu aplikace, aby nedocházelo k výpadkům dat při rychlostech vyšších než $187,5$ kbps.

Příloha A

Překlad a instalace ovladače PBMaster

Překlad ovladače předpokládá, že přeložené zdrojové kódy právě běžícího jádra se nacházejí v adresáři `/usr/src/linux`. Ovladač lze přeložit proti verzi jádra 2.6.10 a vyšší. Verzi právě běžícího jádra zjistíme příkazem `uname`:

```
# uname -rs
Linux 2.6.23-gentoo-r9
```

Nemáme-li k dispozici zdrojové kódy právě běžícího jádra, můžeme se pokusit nainstalovat příslušný balíček. Většina distribucí poskytuje ke svým jádrům balíčky se zdrojovými kódy. Pokud takový balíček neexistuje, nezbývá nám jiná možnost než stáhnout zdrojové kódy z webu[1] nebo použít verzi, které se nachází na CD viz Příloha C.

Překlad ovladače

Zdrojové kódy ovladače najdeme na CD v adresáři `/src/pbmaster.tar.bz2`, nebo stáhneme novější verzi z webu[2]. Zdrojové kódy rozbalíme do adresáře `/usr/src/profibus`, kde vznikl podadresář `pbm_drv`. Přepneme se do adresáře `pbm_drv` a příkazem `make` provedeme překlad. Výchozí rychlost Profibusu může být specifikována ještě před překladem ovladače ve zdrojovém souboru pbm_950pci.c jako proměnná `baud`:

```
# cd pbm_drv
# make
```

[1] http://www.kernel.org
[2] http://www.pbmaster.org

Během překladu by nemělo docházet k výpisu varovných nebo chybových hlášení. Chování se může ovšem lišit v závislosti na verzi jádra a překladového řetězce. Nyní bychom měli mít k dispozici soubory `pbmcore.ko`, `pbm_fma.ko` a `pbm_950pci.ko`, což jsou hotové jaderné moduly připravené k nahrání do jádra.

Instalace ovladače

Některé verze jádra mohou rozpoznat kartu TEDIA PCI-1482U/48MHz jako standardní sériový port a načíst automaticky moduly `8250_pnp`, `8250` a `serial_core`, které by v této situaci kolidovali s naším ovladačem PBMaster. Jejich přítomnost v jádře ověříme následujícími příkazy:

```
# lsmod | grep 8250_pnp
# lsmod | grep 8250
# lsmod | grep serial_core
```

Pokud jsou moduly v jádře nahrány, odstraníme je příkazem `rmmod`:

```
# rmmod 8250_pnp
# rmmod 8250
# rmmod serial_core
```

Nyní nahrajeme do jádra moduly ovladače PBMaster v uvedeném pořadí. Modulu `pbm_950pci` předáme jako jeho parametr rychlost profibusu v bitech za sekundu:

```
# insmod pbmcore.ko
# insmod pbm_fma.ko
# insmod pbm_950pci.ko 187500
```

Úspěšnost ověříme příkazem `dmesg` nebo v souboru `/var/log/syslog`. Cesta i jméno souboru se záznamem se může lišit podle použité distribuce Linuxu. V záznamu vyhledáme, pod jakým *major* číslem se ovladač zaregistroval, např. 253. Nyní ručně vytvoříme zařízení ovladače `/dev/pbmaster0`. Funkce pro automatické vytvoření zařízení programem *udev* není v ovladači zatím implementována. Funkčnost ovladače ověříme čtením ze zařízení například pomocí příkazu `cat`:

```
# mknod /dev/pbmaster0 c 253 0
# cat /dev/pbmaster0
```

Skript pbinit

Protože celou operaci skládající se z uvedených kroků je třeba při ověřování funkčnosti a odstraňování chyb provádět opakovaně a často, vytvořil jsem skript `pbinit`, který celou tuto operaci automatizuje. Skript najdeme na CD v adresáři `/sbin`. Před použitím je ho zapotřebí překopírovat do adresáře, který je zahrnut v proměnné `PATH` a nastavit mu práva pro spouštění:

```
# chown root:root /sbin/pbinit
# chmod 755 /sbin/pbinit
```

Použití skriptu je velmi intuitivní. Překlad, nahrání modulů do jádra a vytvoření zařízení provedeme příkazem:

```
# pbinit load 9600
```

Parametr udávající rychlost Profibusu DP lze vynechat, v tomto případě je použita výchozí hodnota uvedená v souboru `pbm_950pci.c`. V původní verzi je nastavena rychlost na 187500 bps.

Pro odstranění modulů z jádra analogicky použijeme:

```
# pbinit unload
```

V případě potřeby lze v těle skriptu upravit proměnné určující cesty zdrojových kódů nebo umístění výsledných modulů.

Příloha B

Konfigurovatelné parametry serveru

Volba		Konf. soubor	Konstanta	Typ	Význam
dlouhá	krátká				
baud	b	baudrate	-	int	rychlost Profibusu DP
config	c	-	CFG_FILE	string	konfigurační soubor
daemon	D	daemon	DAEMON	int	režim činnosti daemon
device	d	device	PB_DEVICE	string	zařízení PBMaster
help	h	-	-	-	vypíše seznam voleb
-	-	lockfile	LOCK_FILE	string	lock soubor
-	-	logfile	LOG_FILE	string	soubor protokolu
port	p	port	MYPORT	int	TCP port
reset-ts	-	tsreset	TS_RESET	int	reset časové známky
-	-	standalone	-	int	„standalone“ režim
workdir	w	workdir	WORK_DIR	string	pracovní adresář serveru

Tabulka B.1: Konfigurační direktivy a jejich význam

Příloha C

Obsah přiloženého CD

K této práci je přiloženo CD, na kterém jsou uloženy nezbytné zdrojové kódy.

```
CDRoot
 |-- doc
 |   |-- xml
 |   |   |-- Klient2Server.dtd
 |   |   |-- Server2Client.dtd
 |   |   |-- c2s.xml
 |   |   `-- s2c.xml
 |   |-- DP Martin Samek.pdf
 |   `-- pbmaster.README
 |-- sbin
 |   `-- pbinit
 `-- src
     |-- gentoo-sources-2.6.23-gentoo-r9.tar.bz2
     |-- pbmaster.tar.bz2
     `-- pbserver.tar.bz2
```

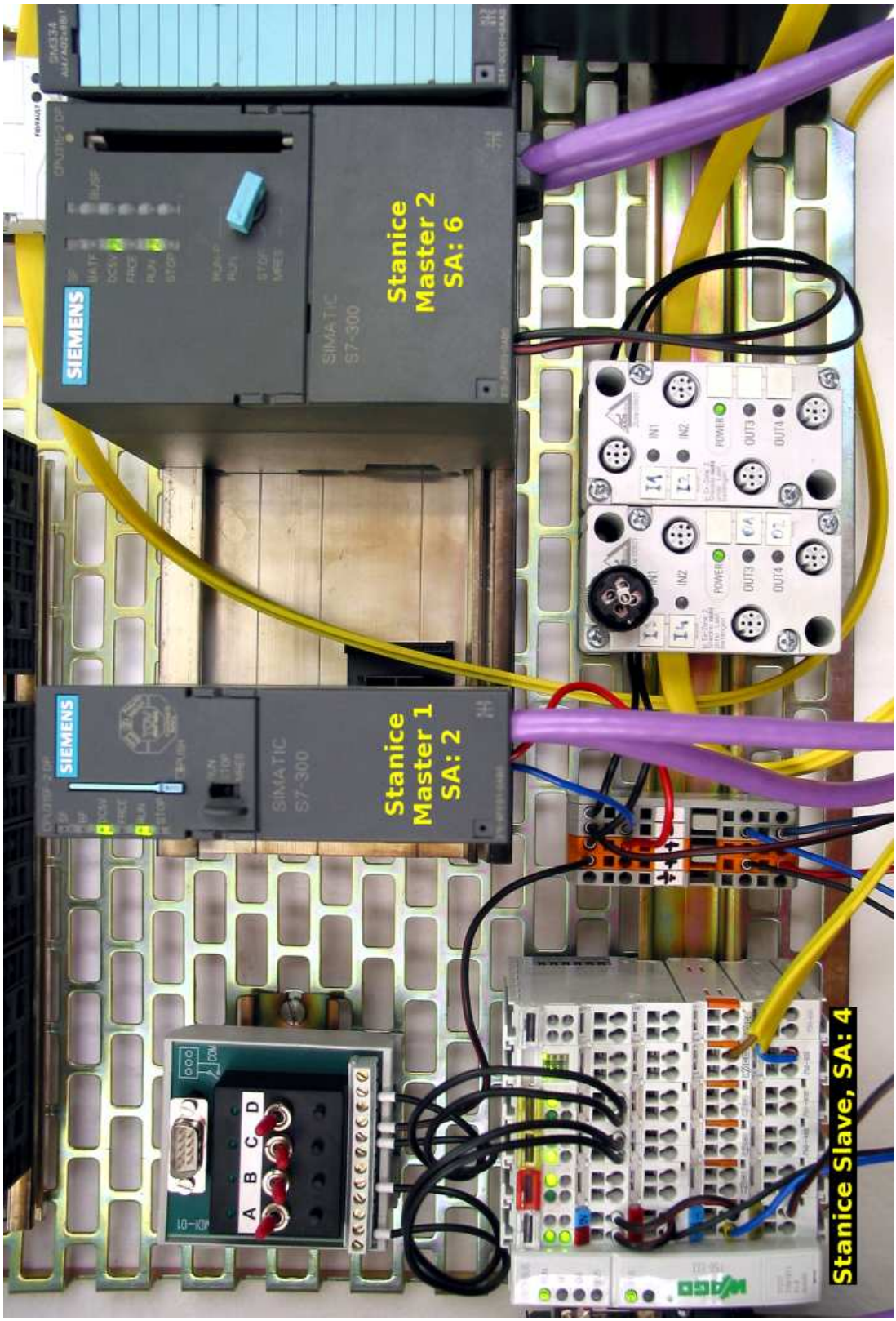
Příloha D

Experimentální sběrnice

Během vývojových pracích na aplikaci PBServer a kolegově PBView bylo vytvořeno malé pracoviště vybavené potřebným vybavením pro realizaci sběrnice Profibus DP, která sloužila jako zdroj dat. Ke sběrnici byl připojen běžný počítač vybavený PCI kartou TEDIA PCI-1482U/48MHz (viz 2.1.1). Z hardwarového vybavení byly použity následující prvky z portfolia firem Siemens a Wago:

1. SIEMENS Simatic S7-300, CPU315F-2 DP, P/N: 315-0FF01-0AB0, adresa: 2.
2. SIEMENS Simatic S7-300, CPU315-2 DP, P/N: 315-2AF03-0AB0, adresa: 6.
3. WAGO I/O SYSTEM 750, Profibus DPV1 750-332, adresa: 4, osazený moduly:
   (a) WAGO 750-512, 2x DO AC 230 V, DC 30 V, AC/DC 2 A.
   (b) WAGO 750-400, 2x DI DC 24 V, 3 ms.
   (c) WAGO 750-400, 2x DI DC 24 V, 3 ms.
   (d) WAGO 750-655, AS-Interface Master module.
   (e) WAGO 750-600, ukončovací modul interní sběrnice.

Na následující fotografii je zachyceno rozložení modulů a jejich propojení tak, jak byly používány během vývoje. V případě pokračování prací na vývoji této aplikace doporučuji vyjít z této konfigurace.



Obrázek D.1: Experimentální sběrnice Profibus DP

Seznam obrázků

1.1 Blokové schéma celého analyzátoru 4

2.1 Profibus DP a ISO/OSI model 6
2.2 Princip kódování Non Return to Zero 7
2.3 Blokové schéma řadiče OXmPCI954 9
2.4 Schéma karty TEDIA PCI-1482U/48MHz 10
2.5 Formát UART znaku 15

4.1 Stavový diagram chování serveru 36
4.2 Formát rámce 37

5.1 Princip funkce `accept()` 45
5.2 Stavy a přechody komunikačního protokolu 53
5.3 Průběh ideální komunikace mezi serverem a klientem v čase 54

6.1 Příklad převodu tokenu do Base64 reprezentace 57
6.2 Princip činnosti vyrovnávací paměti 65

D.1 Experimentální sběrnice Profibus DP VII

Seznam tabulek

2.1 Maximální délka sběrnice (bez opakovačů) v závislosti na rychlosti 7
2.2 Propojení vodičů Profibus DP s PCI kartou 10

4.1 Význam datových polí v rámci 37
4.2 Přípustné hodnoty 1. a 0. bitu v poli CF 38

5.1 Performativy komunikačního protokolu 51
5.2 Význam atributů 52

B.1 Konfigurační direktivy a jejich význam IV

Seznam použitých zkratek

t_{BIT}	Je čas, který trvá po sběrnici přenést právě jeden bit.
.NET	„dotnet“, Prostředí potřebné pro běh aplikací a nabízející jak spouštěcí rozhraní, tak potřebné knihovny.
BSD	Berkeley Software Distribution
DDLM	Direct Data Link Mapper
DOM	Document Object Model
DTD	Document Type Definition
FDL	Field Data Link
GAP	Adresový prostor mezi dvěma aktivními stanicemi.
IPC	Inter-Process Communication, meziprocesní komunikace
JIT	Just-In-Time compilation
LAS	List of Active Stations, seznam aktivním stanic.
MAC	Media Access Control
PCI	Peripheral Component Interconnect, počítačová sběrnice pro připojení periferií k základní desce
PHY	Označení pro první vrstvu protokolu Profibus DP.
POSIX	Portable Operating System Interface
RPC	Remote Procedure Calls, vzdálené volání procedur
SAX	Simple API for XML
SMTP	Simple Mail Transfer Protocol, protokol určený k přenosu zpráv elektronické pošty
TCP	Transmission Control Protocol, jeden ze základních protokolů Internetu, představuje transportní vrstvu ISO/OSI
UART	Universal Asynchronous Receiver and Transmitter, řadič převádějící data mezi paralelní a sériovou formou

Literatura

[1] Oxfordshire: Oxford Semiconductor LTD. *OXmPCI954 Data Sheet, Rev. Jun 2005.* 72 p., 2005. URL: http://www.oxsemi.com/products/serial/OXmPCI954.html.

[2] Plzeň TEDIA spol. s r.o. *Komunikační karty pro sběrnici PCI, Uživatelská příručka, rev. 09.2007.* 24 s., 2007. URL: http://www.tedia.cz.

[3] BLEBTA, Jan. *Diagnostický uzel pro Profibus.* České vysoké učení technické, Elektrotechnická fakulta, Katedra řídicí techniky, 2001. Diplomová práce.

[4] HEROUT, Pavel. *Učebnice jazyka C, III. upravené vydání.* KOPP, 2001.

[5] KERNIGHAN, Brian W.; RITCHIE, Dennis M. *Programovací jazyk C.* Computer Press, a.s., 2006.

[6] MITCHELL, Mark; OLDHAM, Jeffrey; SAMUEL, Ales. *Pokročilé programování v operačním systému Linux.* SoftPress s.r.o., 2002.

[7] PInternational. *Profibus Specification, Edition 1.0.* 1998. Normative Parts of PROFIBUS-FMS, -DP, -PA according to the European Standard EN 50 170 Volume 2.

[8] STONES, Richard; MATTHEW, Neil. *Linux, Začínáme programovat.* Computer Press, a.s., 2000.

[9] Tran Duy Khanh. *Profibus FDL vrstva pro Linux.* České vysoké učení technické, Elektrotechnická fakulta, Katedra řídicí techniky, 2006. Bakalářská práce.

[10] TRNKA, Pavel. *Profibus DP Master na PC.* České vysoké učení technické, Elektrotechnická fakulta, Katedra řídicí techniky, 2004. Diplomová práce.