ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA ELEKTROTECHNICKÁ
KATEDRA ŘÍDICÍ TECHNIKY

DIPLOMOVÁ PRÁCE

Profibus DP Master na PC

2004 Pavel Trnka

Anotace

Tato práce předkládá řešení problému připojení průmyslové sběrnice Profibus k běžnému PC bez použití speciálního hardwaru. Práce implementuje řídící jednotku sběrnice (master) pro Profibus DP (Distributed Peripherals), což je nejrozšířenější varianta tohoto standardu určená pro komunikace mezi řídícími jednotkami a vzdálenými periferiemi (snímače, akční členy).

Běžně používaná řešení jsou nákladná, protože jsou postavena na speciálních rozšiřujících kartách používajících vlastní procesory, zákaznické obvody a jiné obvody s dostatečným výkonem na splnění vysokých požadavků Profibusu.

Tato práce oproti tomu umožňuje připojit Profibus pomocí běžného sériového portu RS–232 nebo pomocí jednoduchých zásuvných PCI karet s obvody UART. Profibus DP Master je vytvořen softwarově a speciální vlastnosti hardwarových řešení jsou nahrazeny maximálním využitím běžně dostupných prostředků v PC, což bylo hlavně umožněno napsáním programu jako ovladač pro operační systémy Windows NT/2000/XP. Navíc aplikační rozhraní je kompatibilní s řešeními firmy Siemens, což umožňuje jednoduchou záměnu.

Annotation

Solution for connecting fieldbus Profibus to common PC without use of special hardware will be proposed in this work. The work implements bus control unit (master) for Profibus DP (Distributed Peripherals), which is most widely used variation of this standard. It's designed for communication between control units and distributed peripherals (sensors, actuators).

Commonly used solutions are expensive, because they are built on special expansion cards, which are using their own processors, customer's circuits or another circuits with enough power to meet high requirements of Profibus.

On the other side, this work allows for connecting Profibus to standard serial port RS–232 or to simple PCI expansion card based on UART circuit. Profibus DP Master is created as software implementation and special features of hardware implementations are substituted by maximal use of common means in PC. This was mainly possible by creating Profibus DP Master as system driver for Windows NT/2000/XP. Moreover application interface is compatible with solutions from Siemens company, which allows for simple replacement.

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 ....
podpis

Poděkování

Nejprve bych rád poděkoval svému vedoucímu diplomové práce Ing. Petru Smolíkovi za jeho vstřícnost, ochotu při konzultacích a čas, který mi věnoval. Dále Ing. Pavlu Píšovi za mnoho podnětných rad z jeho bohatých praktických zkušeností a Ing. Pavlu Burgetovi za jeho pomoc při tvorbě této diplomové práce.

Poděkování také patří mé rodině za jejich neustálou podporu a domácí zázemí po celou dobu studia. V neposlední řadě také mnoha přátelům, jejichž společnost mi vždy dodávala energii k další práci.

Obsah

1 Úvod **1**
1.1 Proč ovladač? . . . 2
1.2 Struktura této práce . . . 2

2 Profibus DP **4**
2.1 Fyzická vrstva . . . 5
2.2 Linková vrstva . . . 5
2.2.1 Provoz na sběrnici . . . 5
2.2.2 Formáty rámců . . . 7
2.2.3 Stavový automat Profibusu pro FDL vrstvu . . . 9
2.2.4 Řešení událostí na sběrnici . . . 14

3 Realizace fyzické vrstvy **16**
3.1 Použití standardního sériového portu . . . 16
3.2 Použití rozšiřujících zásuvných karet . . . 19
3.2.1 Obvod OX16PCI954 . . . 20

4 Tvorba ovladačů pro operační systém Windows **21**
4.1 Nástroje potřebné pro vývoj ovladače . . . 23
4.2 Požadavky systému na ovladač . . . 23
4.3 Rozdíl mezi Legacy a PnP ovladači . . . 24
4.3.1 Plug and Play . . . 25
4.4 Způsob práce ovladače . . . 25
4.5 Prioritní úrovně (IRQL) . . . 27
4.6 Základní struktura a rutiny Legacy ovladače . . . 27
4.6.1 DriverEntry . . . 28
4.6.2 Unload . . . 29
4.6.3 Interrupt Service Routine (ISR) . . . 30

4.6.4 Deferred Procedure Call (DPC) . . . 30
4.6.5 IoTimer . . . 31
4.6.6 Dispatch Routine . . . 31
4.7 Struktura ovladače s podporou PnP . . . 31
4.7.1 DriverEntry . . . 32
4.7.2 AddDevice . . . 33
4.7.3 DispatchPnP . . . 33
4.7.4 Interrupt Service Routine (ISR) . . . 33
4.8 Používání paměti ovladačem . . . 33
4.9 Umístění konfigurace v registrech . . . 34
4.10 Ladící prostředky . . . 34
4.11 INF File . . . 35
4.12 Property Page . . . 36

5 Realizace linkové vrstvy - ProfiM 38
5.1 Programová implementace . . . 38
5.2 Nároky Profibusu . . . 39
5.3 Ovládání převodníku a jeho využití k časování . . . 40
5.4 Časování s obvodem 16C950 . . . 42
5.5 Princip činnosti ProfiMu . . . 42
5.6 Watchdog . . . 45
5.7 Zatížení procesoru . . . 45

6 Aplikační rozhraní FDL vrstvy 47
6.1 Služby linkové vrstvy . . . 49
6.2 Komunikace s FDL vrstvou . . . 50

7 Závěr 52
7.1 Co by šlo vylepšit nebo přidat . . . 53

A Příklad použití ProfiMu 54

B Ukázková aplikace ProfiMu 57

C Seznam zkratek 59
C.1 Zkratky pro Profibus . . . 59
C.2 Zkratky pro DDK . . . 60

D Obsah přiloženého CD 61

Kapitola 1

Úvod

Průmyslová sběrnice Profibus [5] patří mezi nejrozšířenější sběrnice nasazované v průmyslu. Její nejčastěji používanou variantou je Profibus DP (Distributed Peripherals). Ta je určena pro komunikaci mezi řídícími jednotkami (PLC, průmyslové počítače) a decentralizovanými periferiemi, kterými jsou vzdálené snímače a akční členy. Místo mnohavodičového spojení, které by každou periferii připojovalo zvlášť jedním vedením, je použito jednoho propojení Profibusem s topologií lineární sběrnice, které vše spojuje.

Nasazením Profibusu je zaručena rychlá a spolehlivá výměna dat a navíc je možné jednu sběrnici sdílet více řídícími jednotkami. Na druhou stranu pokud chceme používat běžné PC jako aktivní řídící stanici master a připojit k ní Profibus DP, narazíme na vysokou cenu zásuvných karet, které Profibus mastera s využitím zákaznických obvodů a jiných hardwarových řešení implementují.

Motivací této práce proto bylo, navrhnout co nejlevnější a co nejsnáze použitelnou implementaci aktivní stanice *Profibus DP Master* pro PC, která by byla z aplikačního pohledu kompatibilní s některým v praxi rozšířeným řešením a umožnila tak jeho záměnu.

Hardwarová vrstva Profibusu je postavena na standardu RS–485 a tak pro samotné připojení Profibusu realizovaného v této práci stačí běžný sériový port RS–232 s jednoduchým převodníkem RS 232/485, který nepotřebuje externí napájení. To je řešení použitelné pro nízké přenosové rychlosti. Potřebujeme-li dosáhnout vysokých přenosových rychlostí (v současnosti je maximem 12Mbps), potom tato práce také umožňuje použít PCI karet s obvody UART (Universal Asynchronous Receiver/Transmitter), kde maximální rychlost je dána pouze výkonem procesoru a možnostmi použitého UART obvodu.

Jednoduchost připojujícího hardwaru však přesunula veškerou práci k tomu účelu obvykle používaných speciálních obvodů na software. Ten tak musí s maximálním využitím všech dostupných prostředků běžného PC nahradit hardwarová řešení.

Postupná práce na softwaru ukázala, že jedinou schůdnou cestou implementace pro moderní operační systémy je vytvořit *Profibus DP Master* jako ovladač. A tak byl vytvořen

ovladač pro operační systém Windows NT 4.0 a přidáním *Plug and Play* podpory také pro operační systémy Windows 2000 a Windows XP. Vytvořený ovladač implementuje *Profibus DP Mastera* pro PC až po FDL vrstvu (linkovou vrstvu).

Nepodceněnou součástí této práce byla také volba názvu projektu, kterým bylo vybráno slovo *ProfiM* pro jistou podobnost s názvem řešené problematiky a hlavně kvůli malému počtu relevantních odkazů nalezených internetovými vyhledávači.

1.1 Proč ovladač?

Důvodů k nutnosti vytvořit DP mastera jako ovladač bylo několik. Především to byla potřeba v operačních systémech Windows časovat s periodami v řádech desítek mikrosekund, což pro běžné aplikace není dosažitelné. Časováním je zde myšleno vyvolávání obslužných rutin po uplynutí velmi krátkých časových intervalů. Jedinou možností jak toho dosáhnout bylo využít přerušovacích možností a hardwarových vlastností standardního sériového portu nebo UART obvodů na rozšiřujících PCI kartách. To by ovšem z běžné aplikace nebylo možné, neboť těm není operačním systémem dovolen přímý přístup na hardware.

Dalším důvodem byla nutnost řídit převodník RS 232/485, což je sice teoreticky jednoduché, ale při softwarové implementaci pod Windows to představovalo náročný problém, který se také podařilo vyřešit jen díky vytvoření ovladače. Nakonec implementace ve formě ovladače začlenila rozhraní Profibusu do operačního systému, což z aplikační vrstvy přináší výhody pro další používání.

1.2 Struktura této práce

Ke čtení a dobrému pochopení této práce je vhodné mít alespoň částečnou znalost problematiky Profibusu - ta je sice zhruba popsána v kapitle 2, avšak v některých kapitolách se mohou najít principy a pojmy, které nejsou úplně vysvětleny.

Kapitola 2 zhruba popisuje principy Profibusu, především jeho linkové vrstvy, jejíž realizace je hlavní částí této práce. Podrobnější popis nalezneme například v [1] nebo přímo v normě [5].

Kapitola 3 ukazuje jak připojit Profibus k PC z hlediska fyzické vrstvy, neboli jak připojit sběrnici používající průmyslový standard RS–485.

Kapitola 4 popisuje jak se v operačních systémech Windows tvoří ovladače. Popisuje jejich základní strukturu, jaké nástroje jsou pro tvorbu potřeba a jaké jsou odlišnosti od psaní běžných programů. Tato kapitola je užitečná tím, že takto shrnuté informace o psaní ovladačů pro fyzická zařízení jsou málo dostupné. Setkáme se buď s povrchním popisem, anebo s detailní dokumentací.

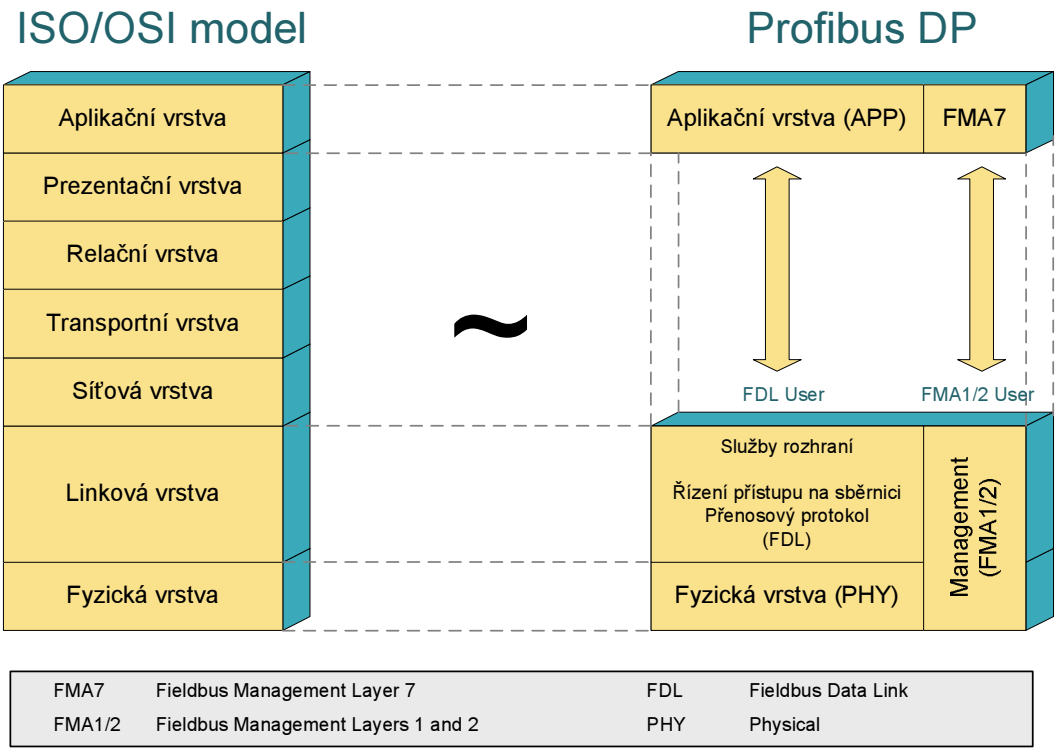
Kapitola 5 ukazuje, jak je vytvořena linková vrstva Profibus DP Mastera - jak probíhal její vývoj a jaké problémy bylo při tom potřeba vyřešit.

Kapitola 6 popisuje rozhraní, které aplikace (vyšší vrstva DP master nebo přímo FDL aplikace) používá pro komunikaci s FDL vrstvou (ovladačem).

Kapitola 2

Profibus DP

Profibus DP (Distributed Peripherals) je nejpoužívanější varianta průmyslové sběrnice Profibus, která je určena zejména pro komunikaci mezi řídícími jednotkami a decentralizovanými periferiemi. Obvykle tak jedním komunikačním kanálem nahrazuje mnohavodičové spojení řídící jednotky se snímači a akčními členy, kde použitím Profibusu je současně zajištěna spolehlivá a rychlá výměna dat.



Obrázek 2.1: ISO/OSI model a Profibus DP

Profibus DP má vrstvenou architekturu postavenou podle modelu ISO/OSI (obrázek 2.1). Z jednotlivých vrstev využívá fyzickou, linkovou a aplikační. Nevyužité čtyři vrstvy jsou částečně zahrnuty v ostatních. Tato práce implementuje fyzickou a linkovou vrstvu a nabízí rozhraní pro aplikační vrstvu.

2.1 Fyzická vrstva

Fyzická vrstva definuje požadavky na vlastnosti přenosového kanálu. Norma Profibusu [5] nabízí jako jednu z možností pro realizaci fyzické vrstvy použití v průmyslu rozšířeného standardu RS–485. Tím jsou určeny fyzické vlastnosti kanálu a kódování dat na sběrnici.

Důležitou vlastností, která je normou pro fyzickou vrstvu definována, je rychlost komunikace. Ta je pro Profibus DP specifikována v rozsahu 9,6kbps až 12Mbps (tabulka 3.1). Norma pro ni definuje maximální přípustnou odchylku 0,3 %, což může v praxi znemožňovat dosažení některých komunikačních rychlostí. Například pokud má UART nevhodnou základní hodinovou frekvenci a pouze hrubé možnosti dělení této frekvence. Praxe však ukazuje, že přijatelná odchylka rychlosti je až 1 %.

2.2 Linková vrstva

Linková vrstva má za úkol řídit přístup na sběrnici (Medium Access Control), sestavovat vysílané rámce, dekódovat příchozí rámce a poskytovat vyšší vrstvě služby datové výměny a služby umožňující řízení a správu linkové vrstvy.

Linková vrstva je tvořena částí FDL (Fieldbus Data Link), která zajišťuje její hlavní funkce a částí FMA (Fieldbus Management), zajišťující její řízení. Ačkoliv to není úplně přesné, je linková vrstva často označována jako FDL vrstva.

2.2.1 Provoz na sběrnici

Profibus umožňuje na jednu sběrnici připojit až 127 stanic. Komunikační kanál tvořený sběrnicí se tak stává sdíleným fyzickým prostředkem, na kterém v jeden časový okamžik může vysílat pouze jedna stanice. K efektivnímu a deterministickému využití sběrnice jsou proto normou určena přesná pravidla.

Stanice jsou rozděleny na dva druhy: master a slave. Stanice typu master jsou určeny k řízení provozu na sběrnici a k iniciování komunikace. Naproti tomu, stanice typu slave může začít vysílat pouze, pokud k tomu byla stanicí typu master přímo vyzvána.

Aby mohla fungovat na sběrnici komunikace, je potřeba, aby na ní byla alespoň jedna stanice master. Řídících stanic typu master však může být i více. K tomu, aby mohly být současně na jedné sběrnici, je potřeba, aby se v řízení sběrnice střídaly. To je zajištěno tím, že stanice master si předávají pověření k vysílání tzv. token. Předávání probíhá v pořadí

rostoucích adres a master s nejvyšší adresou předává token zpět masteru s nejnižší adresou. Tímto postupným předáváním mezi sebou stanice typu master vytvářejí strukturu logického kruhu.

Struktura logického kruhu je určena seznamem LAS (List of Active Stations), který si vytváří a udržuje každý master na sběrnici. Je to seznam všech adres, v němž je pro každou stanici uvedeno, jestli se jedná o aktivní stanici master. Tento seznam si každá stanice vytváří sledováním provozu na sběrnici. Z něj také zjistí, jaké další stanici má předat token (NS - Next Station) a od jaké stanice přijde token nazpátek (PS - Previous Station). Přijme-li master token, získává pověření vysílat na sběrnici. Může tak posílat požadavky stanicím typu slave, případně komunikovat s jiným masterem.

Aby byla zajištěna horní časová hranice, za kterou se ke každému masteru opět vrátí pověření k vysílání, je doba držení tokenu omezená. Je omezená časovým intervalem T_{TR} (Time To Reach), který je definován v každé konfiguraci sběrnice Profibus a udává požadovanou maximální periodu oběhu tokenu mezi všemi stanicemi master.

Drží-li master token, je povinen před každým vysíláním zjišťovat, jestli mu pro jeho držení ještě zbývá čas. K tomuto účelu měří časový interval od okamžiku, kdy naposledy předal token další stanici master v logickém token ringu (NS). Tento interval porovnává s hodnotou parametru T_{TR}. Překročí-li časový interval tento parametr, předává token další stanici master. Zvláštním případem je, pokud čas nezbývá už při přijetí tokenu. V takovém případě může master zpracovat jeden požadavek vysoké priority.

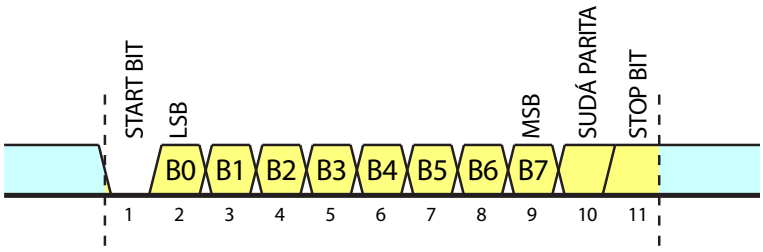
Podstatnou částí normy je popis chování Profibusu při tzv. přechodných stavech, což jsou dočasné události na sběrnici pozastavující její normální činnost, mezi které patří například: první spuštění komunikace na sběrnici, připojení nebo odpojení stanice master, výpadek tokenu, chybný rámec atd. (některé budou popsány v části 2.2.4).

Není-li sběrnice v přechodném stavu, potom obvykle probíhá cyklická výměna dat mezi stanicemi master a jejími příslušnými stanicemi slave (distribuovanými periferiemi). Při této datové výměně master cyklicky posílá každé své stanici slave výstupní data a zároveň čte data vstupní. Nejčastěji nastavuje a čte digitální nebo analogové výstupy a vstupy. Použitím speciálního příkazu sběrnice je navíc zajišťěna synchronizace vstupně/výstupních hodnot na všech stanicích slave, neboli současné nastavení všech výstupů a současné přečtení všech vstupů v jeden časový okamžik.

Celá datová výměna tak připomíná část scan cyklu v PLC a to ne náhodou, neboť právě PLC bývají nejčastějšími stanicemi typu master.

Aby bylo možné stanice za chodu na sběrnici přidávat nebo je odebírat, udržuje každý master seznam obsazenosti jednotlivých adres v jeho adresovém prostoru (GAP), což je rozsah adres od jeho adresy (TS - This Station) až po adresu následujícího mastera (NS). Tento seznam se nazývá „GAP list“ a jeho adresy jsou v určitých časových intervalech cyklicky masterem testovány pro zjištění, jestli se na nich nachází stanice typu master, stanice typu slave nebo je adresa neobsazená.

2.2.2 Formáty rámců

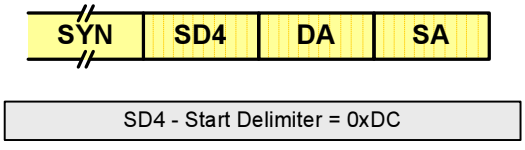


Obrázek 2.2: Formát znaku Profibusu

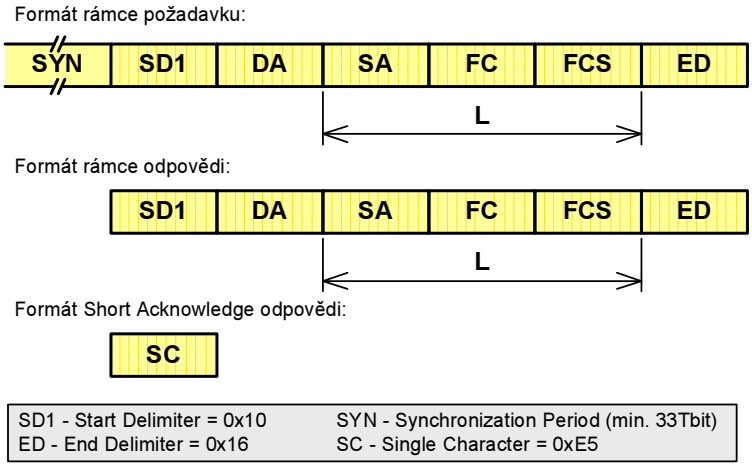
Základní datovou jednotkou na sběrnici Profibus je jedenácti bitový znak (obrázek 2.2), který se skládá z jednoho start bitu, následovaného osmi datovými bity, sudým paritním bitem a jedním stop bitem. Z těchto znaků jsou sestavovány rámce (pakety), pro něž na Profibusu platí důležitý požadavek, aby mezi následujícími znaky rámce nebyly žádné časové mezery. Tento požadavek se může zdát banálním, ale také se může stát problémem v případě, pokud použitý vysílač sériového kanálu nemá FIFO paměť a nemáme možnost pokaždé dostatečně rychle obsluhovat příchozí přerušení.
Pro veškerou komunikaci jsou na Profibusu používány celkem čtyři typy rámců:

1. Rámec bez dat (obrázek 2.4)
2. Rámec s daty fixní délky (obrázek 2.5)
3. Rámec s daty proměnné délky (obrázek 2.6)
4. Rámec pověření k vysílání - token (obrázek 2.3)

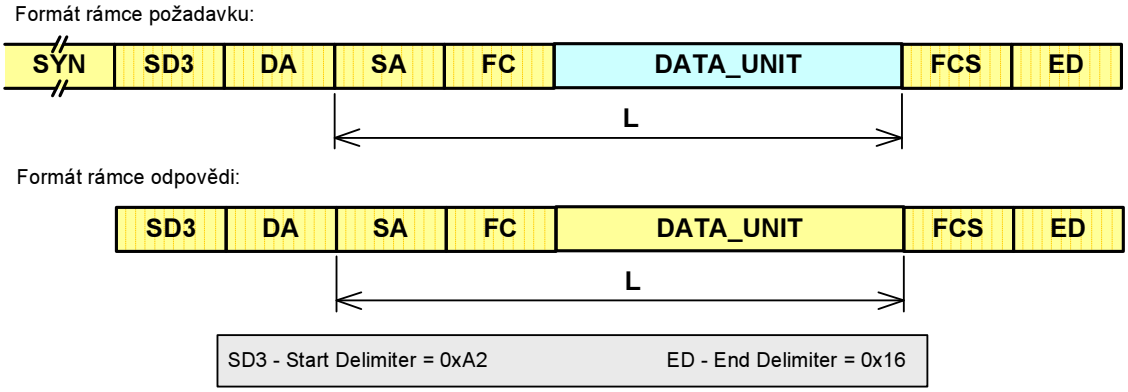
Významy zkratek názvů jednotlivých bytů v rámcích viz. příloha C.1. Každý typ rámce má přesně danou strukturu rámce požadavku a strukturu rámce odpovědi, která by na požadavek měla přijít. Zvláštní odpovědí je rámec Short Acknowledge (obrázek 2.4), který může přijít na libovolný požadavek a obvykle potvrzuje provedení požadavku.



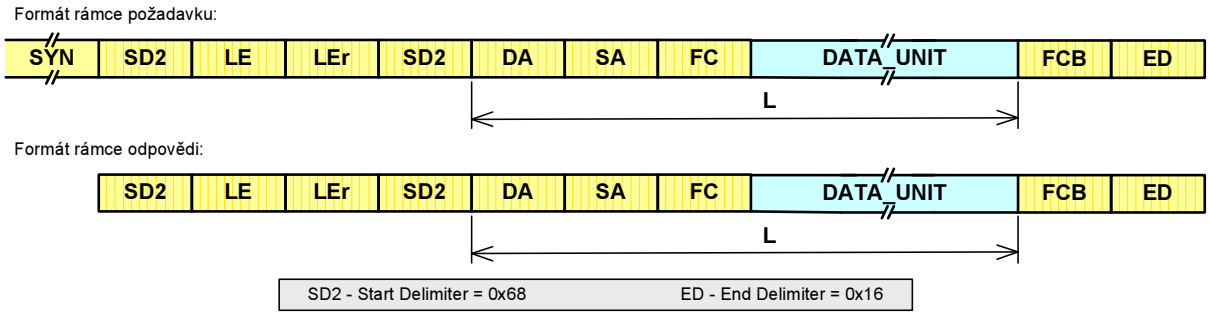
Obrázek 2.3: Formát rámce pověření (Token Frame)



Obrázek 2.4: Formát rámce bez dat



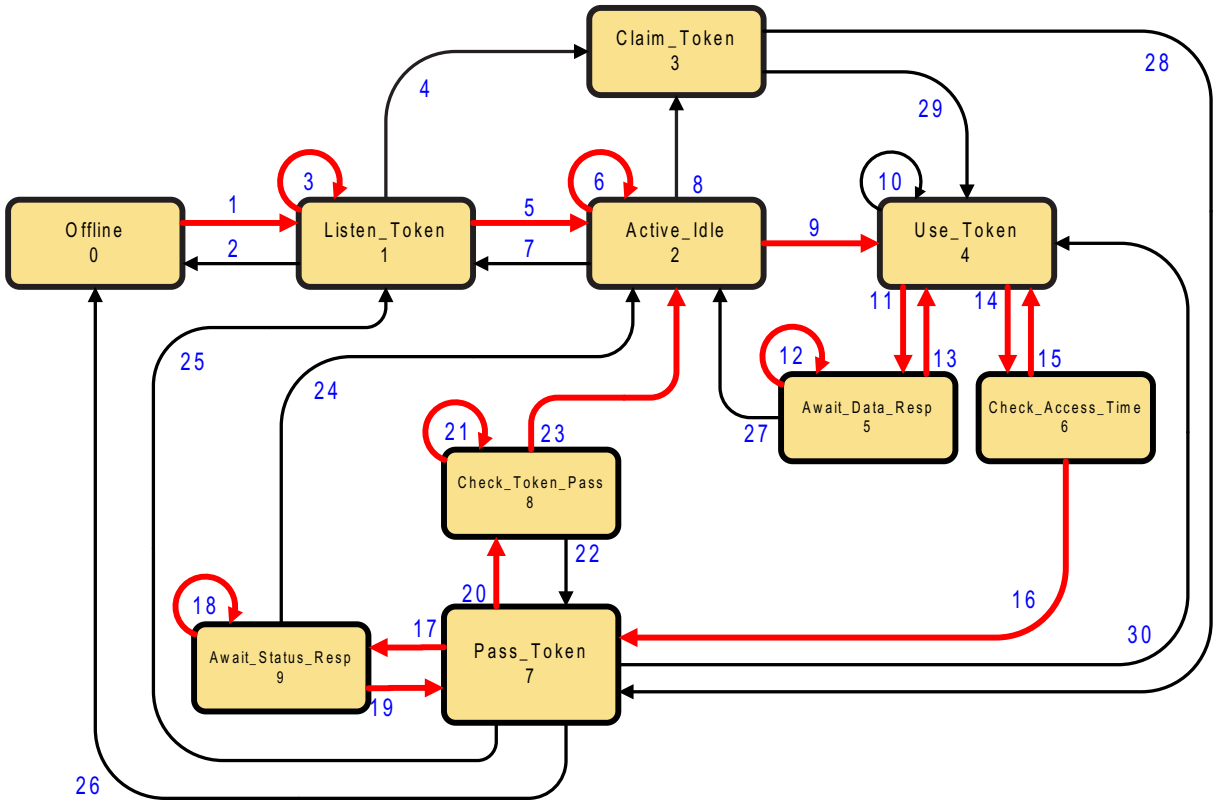
Obrázek 2.5: Formát rámce s daty fixní délky



Obrázek 2.6: Formát rámce s daty proměnné délky

2.2.3 Stavový automat Profibusu pro FDL vrstvu

Funkci FDL, neboli linkové vrstvy, lze dobře popsat a také realizovat jako stavový automat s deseti stavy a přechody mezi nimi (obrázek 2.7). Po zapnutí napájení nebo resetu začíná automat ve výchozím stavu „Offline“. Reakcemi na vnější události, požadavky nižší a vyšší vrstvy přechází automat mezi jednotlivými stavy. Červenou barvou jsou vyznačeny přechody, kterými automat prochází při činnosti stanice v multi-master systému bez výjimečných situací. Přechody vyznačené černou barvou odpovídají událostem souvisejících s kolizemi na sběrnici, ztrátou tokenu, poruchami sběrnice a změnami v logickém okruhu.



Obrázek 2.7: Stavový automat FDL vrstvy

V následujících odstavcích bude popsána činnost stavového automatu. Čísla v závorkách odpovídají očíslovaným šipkám v obrázku 2.7, které reprezentují přechody mezi jednotlivými stavy.

0	Offline

Do stavu „Offline“ vstupuje master ihned po zapnutí napájení. V tomto stavu jsou inicializovány pracovní parametry a je proveden self-test. Při inicializaci parametrů je na-

čtena konfigurace a jsou inicializovány datové struktury FDL vrstvy. Provedení self-testu je závislé na konkrétní implementaci mastera a obvykle zahrnuje test obvodů vysílače a přijímače. Ve stavu odpojení od sběrnice je uzavřen vnitřní loop-back a vyzkoušen testovací přenos dat. Po úspěšné inicializaci a self-testu přechází (1) master do stavu „Listen_Token“.

Další důvod pro přechod do stavu „Offline“ může nastat, pokud je na sběrnici detekována stanice se stejnou adresou (2) nebo pokud je zjištěna hardwarová porucha v přístupu na sběrnici (26) - porucha vysílacích nebo přijímacích obvodů.

1	**Listen_Token**

Je-li master připraven ke komunikaci, přechází do stavu „Listen_Token“. V tomto stavu je zatím stále pasivní a pouze sleduje (3) provoz na sběrnici, aby zjistil jaké stanice jsou aktivní. To zjišťuje tak, že přijímá a analyzuje všechny rámce typu „Token Frame“ (obrázek 2.3), pomocí kterých je mezi aktivními stanicemi v logickém okruhu předáváno pověření. Získáváním adres z rámců „Token Frame“ si stanice vytváří seznam aktivních stanic - LAS (List of Active Stations).

Pokud se masteru podaří při vytváření seznamu LAS odposlouchat dvakrát identický oběh tokenu mezi aktivními stanicemi, nastaví svůj stav na „Ready to enter logical token ring“ a dále pokračuje ve sledování provozu na sběrnici a aktualizaci seznamu LAS. Z vytvořeného seznamu LAS master určí svého předchůdce (PS) a následovníka (NS) v logickém okruhu. Ve stavu „Listen_Token“ zůstává až do příchodu požadavku „Request FDL Status“, který je adresován jemu. Jako odpověď master odesílá svůj stav „Ready to enter logical token ring“ a s bezprostředně následovaným příchodem pověření přechází do stavu „Active_Idle“ (5). Takto je průběh popsán v normě, ale po příchodu rámce pověření se jedná spíše o současný přechod (5) a (9) do stavu „Use_Token“.

Se vstupem do tohoto stavu spouští FDL časovač time-outu s periodou T_{TO}. Jestliže na sběrnici není do vypršení tohoto časovače detekován žádný provoz, potom FDL předpokládá, že je potřeba inicializovat nebo obnovit logický okruh. Za tímto účelem přechází master do stavu „Claim_Token“ (4).

Při odposlouchávání rámců pověření se může stát, že přijde rámec, jehož adresa odesílatele (SA) je shodná s naší adresou (TS). Jsou-li takové rámce přijaty dva, potom zřejmě stanice s naší adresou na sběrnici již existuje a je zařazena do logického kruhu. Master přechází (2) do stavu „Offline“ a o této chybě uvědomuje vrstvu FMA1/2.

2	**Active_Idle**

„Active_Idle“ je stavem, v němž je master plně zařazen do logického kruhu, avšak právě nemá pověření k vysílání (token). Sleduje provoz na sběrnici a odpovídá na požadavky, které mu jsou adresovány, případně pouze přijímá data ze sběrnice, které nevyžadují odpověď.

Přijde-li rámec pověření adresovaný pro nás (DA=TS), přechází (9) master do stavu „Use_Token“. Detekuje-li master, že byl při předávání pověření v logickém kruhu přeskočen (např. stanice PS předala token stanici NS), přechází (7) do stavu „Listen_Token“. Do tohoto stavu také přechází (7), pokud zjistí, že na sběrnici vysílá stanice s duplicitní adresou (rámec přijatý z vnějšku má SA=TS).

Se vstupem do tohoto stavu spouští FDL časovač time-outu s periodou T_{TO}. Jestliže na sběrnici není do vypršení tohoto časovače detekován žádný provoz, potom FDL předpokládá, že došlo k rozpadu logického kruhu a je potřeba jej obnovit. Za tímto účelem přechází master do stavu „Claim_Token“ (8).

3	**Claim_Token**

Do tohoto stavu přechází master v důsledku vypršení časovače pro sledování aktivity na sběrnici a to buď ze stavu „Listen_Token“ nebo „Active_Idle“. Znamená to, že došlo k rozpadu logického kruhu předávání pověření anebo naše stanice je jediný master na sběrnici. Došlo-li k rozpadu logického kruhu, tak máme již k dispozici seznamy GAPL a LAS a není nutno je znovu vytvářet, proto rovnou přecházíme (29) do stavu „Use_Token“. Aby nedošlo k souběhu současným přivlastněním si tokenu několika mastery, je v hodnotě time-outu T_{TO} zohledněna adresa TS podle vzorce:

$$T_{TO} = 6.T_{SL} + 2.TS.T_{SL},$$

kde T_{SL} je interval časového slotu - Time Slot. Díky tomu se jako první pokouší o znovuobnovení logického kruhu stanice master s nejnižší adresou.

Je-li potřeba logický kruh inicializovat, je nejprve na sběrnici dvakrát vyslán rámec pověření s adresami SA=TS a DA=TS. Potom master přechází (28) do stavu „Pass_Token“ k vytvoření seznamu GAPL a určení stanice NS pro předání pověření.

4	**Use_Token**

Stav, ve kterém je master vlastníkem oprávnění, a tudíž může inicializovat vysílání na sběrnici. Při vstupu do tohoto stavu nejprve stanice zjišťuje, kolik má času pro využití oprávnění. Z časovače „Token Rotation Timer“ zjistí, jak dlouho skutečně trval tokenu oběh logického kruhu T_{RR} (Real Rotation time). Porovnáním s požadovanou dobou oběhu T_{TR} (Target Rotation time) je určena maximální doba pro držení tokenu T_{TH} (Token Holding time) jako:

$$T_{TH} = T_{TR} - T_{RR}.$$

Zbývá-li čas ($T_{TH} > 0$), vybere master z fronty požadavků rámec k vyslání. Přednostně vybírá první požadavek, který je na řadě z fronty s vysokou prioritou anebo je-li prázdná, potom vybere požadavek z nízkoprioritní fronty. Po vyslání rámce, na nějž je očekávána

odpověď přechází (11) do stavu čekání „Await_Data_Resp“ a zároveň je spuštěn časovač „Slot Timer“, který zajišťuje vyvolání time-outu pokud dotazovaná stanice neodpovídá.

Jeli-li při vstupu do stavu „Use_Token“ čas (T_{TH}) vyčerpán, tj. ($T_{TH} <= 0$), může master ještě zpracovat jeden požadavek s vysokou prioritou.

Je-li požadavek z fronty takového typu, který nevyžaduje vysílání na sběrnici (např. požadavek na aktivaci SAP), je zpracován, odpověď poslána vyšší vrstvě a master začne zpracovávat další požadavek.

Jsou-li obě fronty prázdné, přechází (14) master do stavu „Check_Access_Time“.

5	**Await_Data_Resp**

Stav, ve kterém čekáme na odpověď po vyslání rámce požadavku. Jestliže vyslaný rámec byl typu SDN (Send Data with No Acknowledge), nečekáme na odpověď a ihned se navracíme (13) do stavu „Use_Token“.

Při čekání můžeme přijmout tyto tři typy rámců:

1. Platné potvrzení nebo odpověď od stanice, kterou jsme adresovali . Odpověď zpracujeme a přecházíme (13) zpět do stavu „Use_Token“.

2. Platný rámec, avšak nečekaného typu (např. rámec pověření) nebo od jiné stanice než očekáváme. Zřejmě někde nastala chyba a tak přecházíme (27) do stavu „Active_Idle“.

3. Neplatný rámec, tj. například rámec s chybným Start Delimiterem, End Delimiterem, chybným kontrolním součtem nebo chybnou paritou některého znaku. Přijde-li tato odpověď nebo vyprší časovač „Slot Timer“, je požadavek opakovaně vyslán. Neodpovídá-li stanice ani na opakované požadavky je o tom uvědoměna vyšší vrstva a master přechází (13) do stavu „Use_Token“.

6	**Check_Access_Time**

V tomto stavu master kontroluje, jestli mu ještě zbývá čas k držení oprávnění. Zbývá-li času dostatek ($T_{TH} > 0$), vrací (15) se do stavu „Use_Token“. Je-li všechen čas vyčerpán, přechází (16) do stavu „Pass_Token“.

7	**Pass_Token**

V tomto stavu se FDL pokouší předat pověření následující stanici v logickém okruhu (NS). Zároveň s vysíláním rámce pověření by měl master současným příposlechem ověřit,

jestli se rámec skutečně daří vyslat. Nepřichází-li z příposlechu nazpět stejný rámec, je zřejmě chyba ve vysílacích obvodech a FDL přechází (26) do stavu „Offline“.

Jestliže ještě před vysláním rámce pověření vyprší „GAP Update Time“ je obnoven nejstarší záznam v seznamu GAP a to tak, že master vyšle testované stanici z jemu příslušného GAPu rámec typu „Request FDL Status“ a přejde (17) do stavu „Await_Status_Response“. Na požadavek mohou přijít dvě odpovědi:

1. Je přijata odpověď od stanice typu slave nebo na požadavek nepřišla žádná odpověď (stanice s danou adresou na sběrnici neexistuje nebo nekomunikuje). Záznam v seznamu GAP je aktualizován a po vyslání pověření pro NS přechází (20) master do stavu „Check_Token_Pass“.

2. Je přijata odpověď od stanice typu master, která je připravena vstoupit do logického kruhu. Odpovídající master je označen jako NS (Next Station) a seznam GAP je příslušně zkrácen. Současně je tomuto masteru odeslán rámec pověření a FDL přechází (20) do stavu „Check_Token_Pass“.

9	**Await_Status_Response**

Do tohoto stavu přechází master po vyslání požadavku „Request FDL Status“ k zjištění stavu stanice na dané adrese. Při vstupu do tohoto stavu je spuštěn časovač s intervalem T_{SL}. Přijde-li odpověď nebo vyprší-li časovač (stanice na dané adrese není nebo neodpovídá), přechází FDL do stavu „Pass_Token“, kde je výsledek zpracován.

Další možností je, že při čekání na odpověď přijde rámec jiného typu, než jaký je očekáván, FDL potom předpokládá, že jiná stanice je aktivní a přechází (24) do stavu „Active_Idle“.

8	**Check_Token_Pass**

Master přechází do tohoto stavu, aby počkal na ověření úspěšného předání pověření. Předání je považováno za úspěšné pokud je ze sběrnice přijat rámec, který vyslala NS (SA=NS). Po ověření přechází (23) master do stavu „Active_Idle“. Nedaří-li se stanici NS pověření předat, vrací (22) se master do stavu „Pass_Token“, aby se pro předání pověření zjistila následující stanice po NS v logickém kruhu.

Je-li master jediným na sběrnici (mono-master system), potom rovnou bez ověřování, které nemá v takovém případě smysl, přechází přes stav „Active_Idle“ do stavu „Use_Token“.

2.2.4 Řešení událostí na sběrnici

Silnou stránkou Profibusu jsou dobře propracované postupy k řešení vyjímečných událostí na sběrnici. Většina z nich vyplývá z popsaného stavového automatu, ale některé ukážeme konkrétně.

Připojení stanice typu master na sběrnici

Po připojení stanice master na sběrnici se stanice chová pasivně a pouze sleduje provoz na sběrnici. Z rámců posílaných po sběrnici si vytváří seznam LAS a tak zjišťuje adresy dalších stanic typu master. Po úspěšném vytvoření seznamu, čeká master na přijetí požadavku „Request FDL Status“ od stanice master v jejímž adresním GAP prostoru leží. Po příchodu tohoto požadavku odesílá odpověď, ve které oznamuje, že je připraven. To způsobí, že je mu vzápětí poslán token, čímž vstupuje do logického ringu a může začít aktivně komunikovat.

Ztráta tokenu

Dojde-li z nějaké příčiny ke ztrátě tokenu, ustane provoz na sběrnici. Tento stav je stanicemi master detekován a po vypršení time-outu si stanice master s nejnižší adresou přivlastní token a reinicalizuje logický okruh. Délka time-outu je určena podle adresy stanice, tudíž master s nejnižší adresou se jako první pokusí o reinicializaci okruhu.

Odpojení stanice typu master ze sběrnice

Je-li stanice typu master náhle odpojena ze sběrnice, mohou nastat tyto dva případy:

1. V okamžiku odpojení vlastnila token jiná stanice master. Až přijde řada na odpojenou stanici, bude se jí předchozí stanice (PS) snažit předat token. Po opakovaném neúspěšném pokusu o předání tokenu odpojené stanici bude její adresa v seznamu LAS označena jako pasivní a zároveň bude z tohoto seznamu určena následující stanice z logického okruhu, které bude předán token.

2. Držela-li odpojená stanice master v okamžiku odpojení token, potom odpojením dojde k jeho ztrátě. Tento stav je řešen podle popisu z předchozího odstavce.

Dotazovaná stanice neodpovídá

Vyšle-li master požadavek, potom s posledním vyslaným bytem rámce spouští časovač time-out s periodou T_{SL}. Nezačne-li do jeho vypršení přicházet odpověď, potom master požadavek opakuje. Počet opakování je určen parametrem sběrnice `retry_ctr`. Neodpovídá-li stanice ani na opakované pokusy, je označena jako neaktivní.

Chyba při příjmu odpovědi na dotaz

Povinností každé stanice na sběrnici je udržovat v paměti rámec odpovědi na poslední požadavek ze sběrnice. Dojde-li totiž ke ztrátě nebo poškození této odpovědi, potom je stanicí master požadavek vyslán opakovaně. To, že se jedná o opakovaný požadavek se pozná podle nezměněného bitu FCB, který je součástí řídícího znaku rámce (FC - Frame Control). Tento bit je s každým novým požadavkem invertován. Dojde-li tedy požadavek se stejným FCB jako měl požadavek předchozí, opakuje stanice poslední odpověď.

Duplicita adres na sběrnici

Detekuje-li stanice odposlechem provozu na sběrnici rámce vyslané od stanice se stejnou adresou, potom po přijetí dvou takových rámců přechází do stavu „Offline“ a signalizuje chybu na sběrnici.

Kapitola 3

Realizace fyzické vrstvy

Při psaní ProfiMu bylo dbáno na důsledné oddělení hardwarově závislých částí programu od těch, které mohou být hardwarově nezávislé. To umožňuje ovladači pracovat nad různými realizacemi fyzické vrstvy Profibusu, neboli různými způsoby jak k PC připojit Profibus. Tyto realizace se mohou lišit podle složitosti, nákladnosti a maximální dosažitelné přenosové rychlosti.

Nejjednodušší možností připojení s minimálními náklady je použití standardního sériového portu PC s jednoduchým převodníkem RS 232/485 bez externího napájení. To nám umožní plnohodnotné připojení na Profibus pro přenosové rychlosti 9600bps a 19200bps. Vyšších rychlostí na standardním sériovém portu nelze dosáhnout z důvodů uvedených v části 3.2.

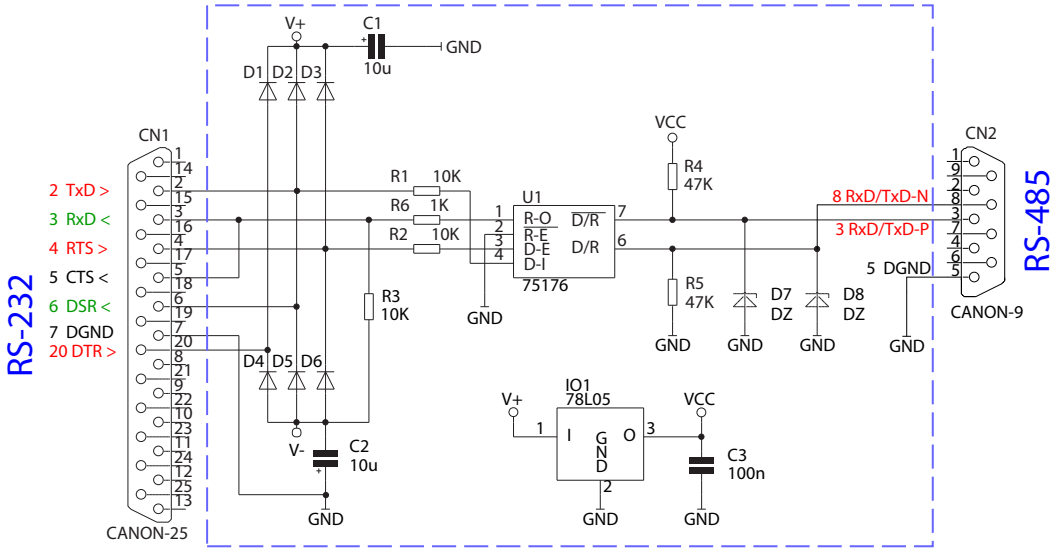
Jinou možností je použít rozšiřující zásuvné PCI karty s obvodem UART (Universal Asynchronous Receiver/Transmitter), což umožňuje posunout rychlostní limit mnohem dále a s vhodným typem obvodu a dostatečně výkonným procesorem lze dosáhnout i momentálně maximální přenosové rychlosti Profibusu 12Mbps.

3.1 Použití standardního sériového portu

K připojení Profibusu na sériový port PC je potřeba použít převodník RS 232/485. Profibus totiž pracuje na standardu RS–485 a sériový port PC na RS–232. Rozdíl mezi těmito standardy je především v napěťových úrovních, které používají k reprezentaci logických hodnot. Druhým důležitým rozdílem je také to, že standard RS–485 umožňuje propojení více než dvou zařízení po jednom vedení. Více zařízení je připojeno tak, že v jeden okamžik vysílá jenom jeden a ostatní pouze přijímají, proto zařízení, která chtějí po RS–485 komunikovat mívají přepínání směru toku dat.

Možné schéma převodníku je na obrázku 3.1. Jedná se o jednoduchý převodník postavený na obvodu 75176, který ke své činnosti nepotřebuje externí napájení. Napájen je přímo

ze sériového portu nastavením výstupu DTR na logickou jedničku. Tím se zjednodušuje použití převodníku, avšak je tím také omezeno celkové maximální zatížení budiče sběrnice. To nám omezuje maximální počet připojitelných zařízení, délku vedení a neumožňuje použití ukončovacích odporů vedení sběrnice nutných pro vyšší přenosové rychlosti.



Obrázek 3.1: Jednoduchý interface RS–232/RS–485

Potřebujeme-li převodník s lepšími vlastnostmi nezbyde, než použít převodník s externím napájením. Ten kromě výkonového budiče linky může také nabídnout galvanické oddělení, které je vzhledem k obvyklému nasazení standardu RS–485 v průmyslu často používané.

Nezbytnou součásti převodníku je přepínání směru toku dat. K tomu slouží signál RTS. Pro RTS=0 jsou výstupy budiče linky ve stavu vysoké impedance a převodník je tak přepnut na příjem, pro RTS=1 jsou budiče sběrnice aktivovány a převodník může vysílat.

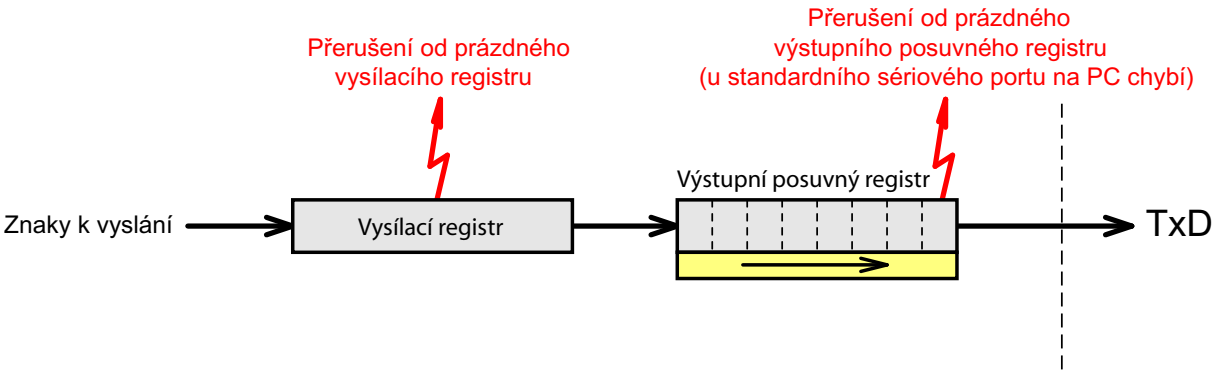
Na tomto místě se hodí poznamenat, že existují převodníky s automatickým přepínáním směru. Princip jejich činnosti spočívá v použití monostabilního klopného obvodu pro řízení směru toku dat. Tento klopný obvod přepíná na vysílání s příchodem sestupné hrany start bitu na začátku každého vysílaného znaku. Perioda klopného obvodu je nastavena na délku jednoho znaku, což znamená, že pro každou přenosovou rychlost je potřeba tuto délku periody změnit. K využití pro komunikaci na Profibusu jej však v našem případě použít nemůžeme, protože potřebujeme pomocí převodníku zajišťovat navíc ještě časování, k čemuž je potřeba softwarové ovládání směru toku dat (bude popsáno v části 5.3).

Další vlastností tohoto zapojení převodníku je hardwarový loopback. Je-li převodník přepnut na vysílání (RTS=1), potom vše co je vysláno do vstupu převodníku signálem TxD se vnitřní smyčkou vrací do RxD. Jinými slovy, vše co ze sériového portu PC přes

převodník vyšleme se také ihned na vstup sériového portu vrátí. Tuto vlastnost lze využít pro přepínání směru toku dat.

Důležitou částí převodníku, která je ProfiMem také využívána, je propojení mezi výstupem TxD a vstupem DSR. Použijeme-li s ProfiMem jiný převodník než takový jako je na obrázku 3.1, je potřeba, aby měl tuto propojku a hardwarový loop-back pro vysílaná data.

Velkou potíží při připojování sběrnice používající standard RS–485 přes převodník RS 232/485 na většinu obvodů UART s RS–232 je nemožnost generovat přerušení od prázdného **výstupního posuvného registru** (obrázek 3.2). Chceme-li totiž použít převodník RS 232/485, potřebujeme softwarově řídit směr přenosu dat. To znamená před každým vysíláním přepnout směr převodníku na vysílání a po vyslání posledního bitu posledního znaku přepnout zpět na příjem. Přepnutí směru na vysílání není problém, to lze provést před vysláním prvního bytu vysílaných dat, avšak přepnutí nazpátek již tak jednoduché není a to právě díky tomu, že většina standardních UART obvodů (tabulka 3.2) nenabízí přerušení od prázdného **výstupního posuvného registru**.



Obrázek 3.2: Přerušení vysílače UARTu

Jediné přerušení, které u takovýchto obvodů můžeme při vysílání dostat pouze říká, že je prázdný **vysílací registr**, který tak můžeme znovu naplnit, avšak **výstupní posuvný registr** ještě prázdný není a teprve dokončuje vysouvání znaku, což znamená, že znak ještě nebyl z UARTu úplně vyslán. Přepneme-li tedy směr po příchodu takového přerušení, dojde k chybě na sběrnici, neboť směr vysílání je změněn uprostřed vysílaného znaku a tak je na sběrnici vyslána pouze jeho část.

Jediný způsob jak u těchto obvodů zjistit úplné vyslání znaků je přečtení Link Status Registru (LSR), který obsahuje bit indikující prázdný **výstupní posuvný registr**. Avšak jelikož od tohoto bitu nemůžeme generovat přerušení, je potom nutné k čekání použít nekonečné smyčky, což není dobrý způsob pro programy v systémech řízených událostmi jako jsou Windows a úplně nepřijatelný způsob pokud píšeme ovladač a potřebujeme toto

PC RS–232	...	9600	19200	38400		57600		111520		
Profibus		9600	19200		45450		93750		187500	...

Tabulka 3.1: Porovnání přenosových rychlostí RS–232 v PC a Profibusu (hodnoty jsou v bps)

čekání provádět v rutině obsluhy přerušení (ISR) nebo DPC rutině (viz. kapitola 4). Tento problém se však podařilo vyřešit a jeho řešení bude popsáno v kapitole 5.

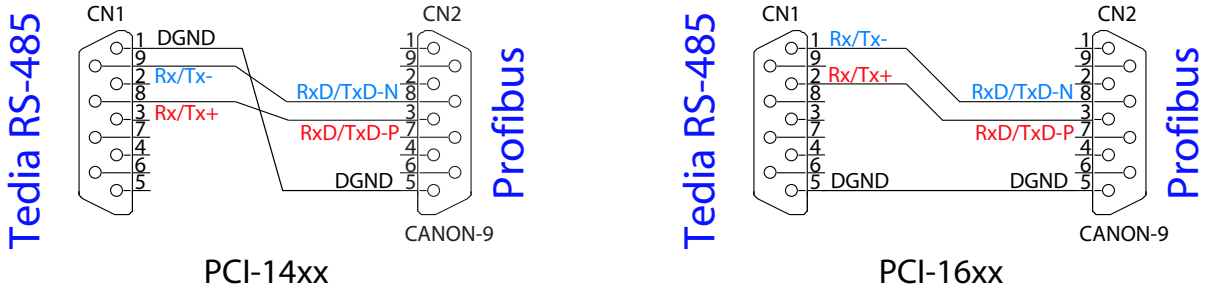
3.2 Použití rozšiřujících zásuvných karet

Pokud srovnáme přípustné přenosové rychlosti, které nabízí standardní RS–232 na PC s možnými přenosovými rychlostmi Profibusu, zjistíme, že do tolerance požadované Profibusem 0,3% se vejdeme pouze pro rychlosti 9600bps a 19200bps (tabulka 3.1). Pro dosažení vyšších rychlostí nám tak nezbývá než použít zásuvných karet s obvody UART, které mají vyšší frekvence základní hodinové frekvence s dobrými možnostmi jejího dělení. Ty nám umožní nejen dosahovat potřebných rychlostí přenosu (při použití vhodného krystalu základní hodinové frekvence), ale také přinášejí nové možnosti oproti obvodům UART standardně používaných v PC (tabulka 3.2), které jsou kompatibilní s obvodem 16C450.

Některé novější obvody UART nabízejí možnost generovat přerušení i od vyprázdnění výstupního posuvného registru. Pokud bychom tuto možnost měli, zjednodušila by se nám tím programová obsluha přístupu na sběrnici a také by se zjednodušil způsob jakým je sériový port využíván k časování. V tabulce 3.2 jsou uvedeny některé novější obvody UART spolu se základními vlstnostmi. Je vidět, že obvodem, který takové přerušení nabízí, je 16C950. Navíc nové obvody nabízejí velkou vyrovnávací FIFO paměť, což umožňuje snížit zátěž procesoru při vysokých přenosových rychlostech.

Jako další možnost hardwarového řešení proto byla vybrána karta pro PCI sběrnici PCI-1482 od firmy Tedia [10], ta je postavena na obvodu OX16PCI954, který je s 16C950 kompatibilní. Stejně tak je možné použít od této firmy libovolnou kartu z řady PCI-14xx nebo PCI-16xx. Porty na kartě je potřeba pomocí přepínačů přepnout do režimu „RS–485“ a pomocí propojky nastavit režim „High Speed“. Dále je potřeba použít jednoduchou redukci (obrázek 3.3), neboť výstupní konektor karty má jiné rozložení signálů než konektor Profibusu.

Použijeme-li jinou kartu s obvodem UART, potom nejlepší je taková, která je postavena také na obvodu kompatibilním s 16C950. Je-li použita karta s obvodem nižšího typu 450 až 750 je potřeba, aby měla alespoň možnost datového loop-backu a propojení mezi výstupem TxD a vstupem DSR (důvody viz. část 5.3). Programová obsluha takové karty by pak byla téměř stejná, jako při použití převodníku RS 232/485.



Obrázek 3.3: Redukce pro připojení karet Tedia na Profibus

Typ UART obvodu	Velikost FIFO paměti	Přerušení od prázdného posuvného vysílacího registru
450	1B	Ne
550	16B	Ne
650	128B	Ne
750	128B	Ne
950	128B	Ano

Tabulka 3.2: Vlastnosti UART obvodů

3.2.1 Obvod OX16PCI954

Obvod vyráběný firmou Oxford Semiconductor [9] určený pro použití na rozšiřujících zásuvných kartách pro PCI sběrnici. Jedná se o složitý obvod, který kromě sériových kanálů nabízí i jiné funkce, jako paralelní port, podporu IrDA atd. Pro nás je důležité to, že tento obvod integruje čtyři vysokorychlostní UART kanály, kde každý je kompatibilní s obvodem 16C950 a má 128B FIFO paměti pro vysílač i přijímač.

Podle rychlosti použitého krystalu základní hodinové frekvence může být maximální dosažitelná přenosová rychlost až 15Mbps pro asynchronní režim. Dobrou vlastností tohoto obvodu je možnost neceločíselného dělení hodinové frekvence, což lépe umožňuje současně dosáhnout několika netypických přenosových rychlostí pro jeden hodinový krystal.

Dále je tento obvod připraven pro připojení na sběrnici PCI a plně integruje podporu Plug and Play s možností načíst konfiguraci z paměti EEPROM. V neposlední řadě umožňuje přerušovat od prázdného vysílacího posuvného registru.

Kapitola 4

Tvorba ovladačů pro operační systém Windows

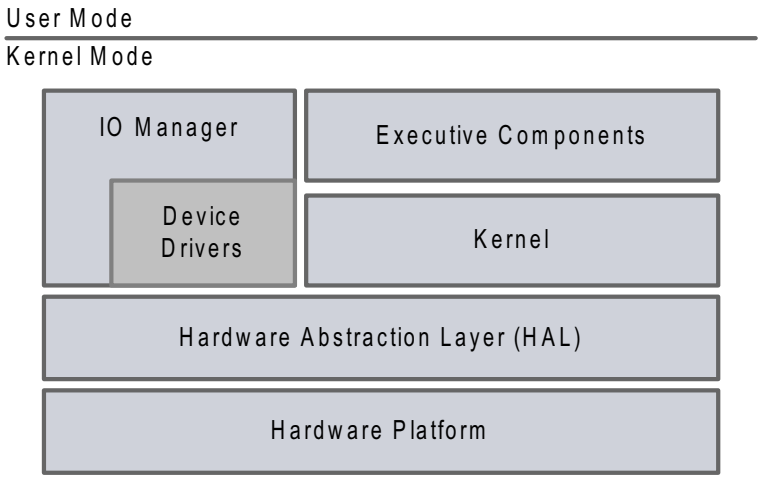
Jelikož psaní ovladače představovalo podstatnou část této diplomové práce, ukážeme v této kapitole jak se v operačních systémech Windows vytvářejí ovladače pro fyzická zařízení. Existují i jiné druhy ovladačů, avšak v této kapitole se budeme výhradně zabývat ovladači pro fyzická zařízení. Popíšeme jejich základní strukturu, ukážeme jaké nástroje jsou pro jejich tvorbu potřeba a jak se liší psaní ovladačů od psaní běžných programů.

Potřebujeme-li vytvořit ovladač pro operační systém Windows, narazíme z počátku na nedostatek informací, které by řešení tohoto problému zhruba popsaly, ukázaly jednoduché příklady a nastínily postup. Součástí DDK (Driver Development Kit) od Microsoftu je sice velmi dobrá dokumentace, ale ta je pro první přiblížení příliš detailní a nenabízí celkový pohled na problematiku. Na druhou stranu s využitím Internetu najdeme mnoho návodů a popisů, které jsou však v naprosté většině příliš povrchní. Problém je totiž v tom, že pro úvod do tvorby ovladačů neexistuje nic jako krátký obligátní program „Hello World“. Při tvorbě ovladačů se sice často vychází z ukázkových zdrojových kódů, které jsou součástí DDK, avšak i pro nejjednodušší použitelné případy mají počty řádek v řádu tisíců. Je to dáno tím, že i nejjednodušší ovladač musí splňovat některé základní požadavky a musí mít rutiny, které systému umožňují s ovladačem komunikovat. Tím také vzniká potřeba dobré znalosti mechanismů činnosti operačního systému, jehož součást ovladač tvoří.

Chceme-li přímo přistupovat na hardware pod operačními systémy Windows, což je náš případ, nebo využívat nízkoúrovňových služeb systému, potom potřebujeme být v takzvaném privilegovaném režimu. Programy ve Windows totiž pracují na dvou úrovních. První z nich je neprivilegovaná úroveň neboli User mode (Ring 3). Ta je určena pro běžné aplikace, které nejsou hardwarově závislé a vystačí si se službami poskytovanými přes API

(Application Interface) operačního systému. V této úrovni běží většina aplikací operačního systému a nejsou v ní povoleny privilegované instrukce procesoru, ke kterým patří například instrukce vstupu a výstupu *in, out* a instrukce *halt*. Omezení umožňují systému kontrolu nad činností uživatelských programů, jejich vzájemné oddělení a zvyšují tak stabilitu a ochranu operačního systému.

Druhá z úrovní je privilegovaná neboli Kernel mode (Ring 0), v té pracuje jádro operačního systému a jeho součásti (obrázek 4.1), ke kterým patří i ovladače. Kód spouštěný na této úrovni již není omezen limity user módu, což mu umožňuje přímý přístup na hardware a kontrolu nad svým během. Zároveň však je tím zvýšeno nebezpečí chyb. Chyby v programu na úrovni kernel módu totiž obvykle vedou přímo k havárii systému. Tvorba ovladače proto vyžaduje velice důsledné a pečlivé psaní zdrojového kódu. Zacyklení v programu nebo deadlock v kritických částech kódu, jako například v obsluze přerušení, způsobují zastavení systému a například přístup do paměti, která nepatří ovladači způsobí téměř jisté zhroucení celého systému.



Obrázek 4.1: Kernel Mode

Ke zvýšení těchto rizik ještě přispívá používání jazyka C, který je k tvorbě ovladačů převážně používán. Jeho silnou stránkou je práce s ukazateli, avšak díky velké benevolenci v této oblasti se může například stát, že díky chybě v programu dojde k použití neinicializovaného ukazatele odkazujícího na paměť nepatřící programu. Takovou chybu lze ve velkém projektu lehce udělat a navíc se nemusí projevovat pokaždé a tak je potřeba při psaní kódu vysoké obezřetnosti.

4.1 Nástroje potřebné pro vývoj ovladače

Ovladače je možné vyvíjet v libovolném jazyce, který k volání funkcí používá konvenci jazyka C, avšak plnou podporu vývoje nalezneme pouze pro jazyk C (hlavičkové soubory, ukázky zdrojových kódů). Není dokonce ani zvykem pro psaní ovladačů používat jazyka C++, ale převážně jazyk C. Bylo by to sice možné, ale doporučením Microsoftu je používat jazyk C.

Pro psaní ovladačů se obvykle používá následující skupina nástrojů, která je kromě Visual Studia součástí „MSDN Professional Subscription“:

- Windows NT free build
- Windows NT checked build (není nutností)
- Software Development Kit (SDK)
- Windows NT Device Driver Kit (DDK)
- Visual Studio (VS nebo VC++)

Co znamená *checked build* a *free build*? S těmito pojmy se v oblasti ovladačů setkáváme často. Jsou to dvě různé verze, do nichž lze přeložit ovladač nebo jakýkoliv jiný program. *Checked build* znamená, že program byl přeložen s přidáním ladících informací a rozšířené vnitřní kontroly chyb, což nám umožňuje snadnější vývoj, ale také snižuje rychlost a zvyšuje paměťové nároky. Oproti tomu *free build* je určena pro finální překlad s plnou optimalizací, kdy jsou ladící informace odděleny od kódu.

Balík DDK je určen pro tvorbu ovladačů, obsahuje hlavičkové soubory a statické knihovny, které jsou nezbytné pro jejich přeložení. Pro svou práci potřebuje SDK, ze kterého využívá překladač a další nástroje.

4.2 Požadavky systému na ovladač

Při psaní ovladače je třeba dbát na to, aby splňoval určitá kritéria daná operačním systémem. Jejich splnění umožňuje systému dobrou flexibilitu a rychlou reakci na události. Těmito kritérii jsou:

Přepnutelnost a přerušitelnost kódu ovladače. Jelikož kód ovladače poběží pod preemptivním multitaskingovým operačním systémem, musí být odolný pro případ přepnutí nebo přerušení. To znamená, že dočasné přerušení běhu kódu nesmí způsobit jeho chybu. Je třeba počítat s tím, že přepnutí nebo přerušení může přijít kdykoliv a tak je nutné zajistit, aby v takových případech nedošlo k deadlockům nebo problémům se sdílenými daty. Systém k tomu poskytuje širokou paletu nástrojů jako jsou spinlocky, kritické sekce atd.

Je třeba uvážit, že ačkoliv přerušení maskuje příchod jiných přerušení stejné úrovně priority, jeho obsluha může být kdykoliv pozastavena příchodem přerušení s vyšší prioritou.

Konfigurovatelnost hardware a software. Ovladač by neměl být závislý na konkrétním nastavení hardware. Například napíšeme-li ovladač sériového portu, měl by být schopen pracovat se všemi sériovými porty v počítači (COM1,COM2 atd.).

Paketování I/O operací. Všechny požadavky na ovladač přicházejí ve formě paketů, které se nazývají IRP (I/O Request Packet). Každý ovladač musí mít rozhraní, které umožňuje tyto pakety přijímat, zpracovávat a odpovědi opět odesílat ve formě IRP. Ovladač by měl být schopen pracovat s několika rozpracovanými požadavky najednou.

4.3 Rozdíl mezi Legacy a PnP ovladači

Legacy ovladač je pojmenování zavedené od Windows 2000 pro starý typ ovladačů, který nemá podporu PnP (Plug and Play). Chceme-li vytvořit plnohodnotný ovladač pro operační systémy Windows 2000 a vyšší je potřeba, aby splňovaly požadavky WDM (Windows Driver Model), ke kterým hlavně patří:

- **Podpora Plug and Play** - podpora PnP je nutná pro získání hardwarových prostředků a neomezenou funkci ovladače v PnP operačním systému.
- **Podpora pro Power Management** - zařízení, pro které je ovladač psán Power Management přímo hardwarově podporovat nemusí, ale ovladač by měl alespoň implementovat jeho rozhraní pro operační systém. Jinak pokud například je v systému jeden ovladač bez Power Managementu, pak systém nemůže přejít do úsporného režimu.

Nesplňuje-li ovladač některou z těchto podmínek, potom z pohledu Windows 2000 se jedná o tzv. Legacy driver, tedy ovladač starého typu, který v rámci kompatibilty s Windows NT může být používán. Problém však může nastat při konfliktu mezi PnP a Legacy ovladači. Například pokud chceme použít Legacy ovladač pracující se sériovým portem, potřebujeme ze systému odstranit standardní PnP sériový ovladač, což je téměř nemožné. Navíc mohou nastat problémy s tím, že PnP manager nepřipojí rozsah portů potřebný pro legacy ovladač a tak je nakonec jednodušší podporu PnP do ovladače přidat.

4.3.1 Plug and Play

Požadavky na ovladač podporující PnP:

PnP ovladač nesmí zabírat hardwarové prostředky přímo. Místo toho ovladač sestaví seznam prostředků, které potřebuje a pošle jej PnP Manageru. Mezi prosředky, o které ovladač žádá, patří: čísla požadavků přerušení IRQ, rozsahy I/O portů, DMA kanály a rozsahy paměťově mapovaných přístupů na zařízení. Po shromáždění všech požadavků na prostředky rozhodne PnP manager, jak je nejlépe rozdělit, aby všem bylo pokud možno vyhověno, a které prostředky tak budou kterému ovladači přiděleny. Tyto prostředky jsou pro ovladač PnP Managerem automaticky zabrány, o čemž je ovladač informován pomocí IRP zprávy IRP_MN_START_DEVICE. Teprve až příchodem této zprávy může ovladač začít přistupovat na hardware a to výhradně pouze za použití přidělených prostředků.

PnP ovladač musí pracovat podle PnP Manageru. PnP Manager spravuje zařízení v systému jako celek a vyzývá ovladač k obsluze jeho jednotlivých zařízení. PnP ovladač, například, nesmí při svém nahrání (DriverEntry) vyhledávat svoje zařízení, ale místo toho musí být navržen tak, aby při nalezení zařízení pro ovladač byla systémem volána funkce AddDevice. PnP Manager volá funkci AddDevice pro každé nalezené zařízení, které je ovladač schopen obsluhovat. Zařízení je ovladačem obsluhováno, dokud PnP Manager nezastaví jeho činnost.

PnP ovladač by měl být maximálně flexibilní. Může-li zařízení pracovat s různou konfigurací hardwarových prostředků (například může používat různá přerušení IRQ nebo různé rozsahy I/O portů) měl by ovladač poskytnout všechny možně kombinace prostředků PnP Manageru, aby ten měl co nejlepší možnosti jak uspokojit všechny požadavky na prostředky.

4.4 Způsob práce ovladače

Pokusme se nyní zhruba popsat činnost PnP ovladače ve Windows.

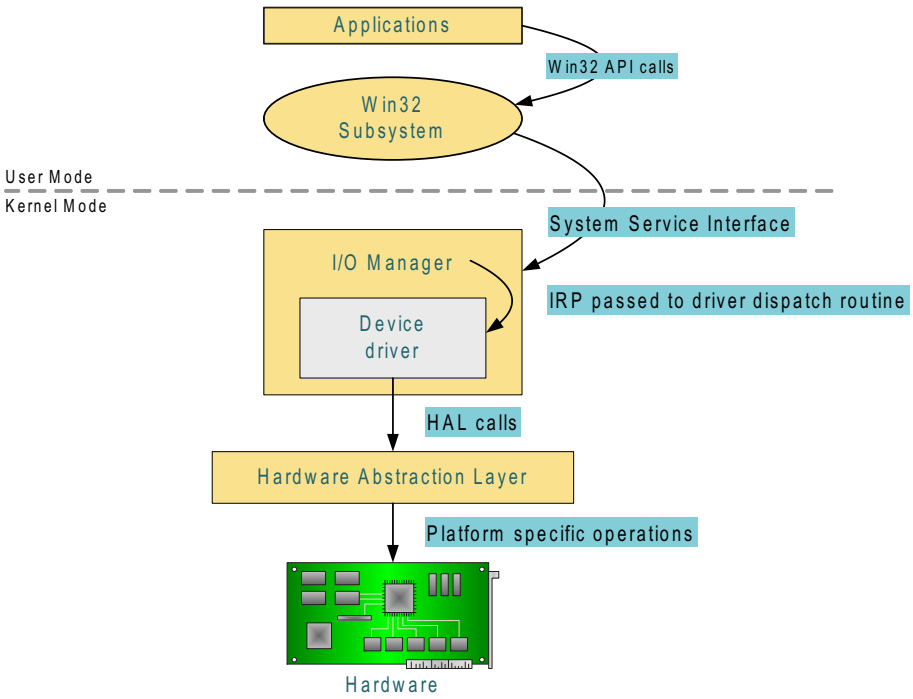
Ovladač je spuštěn automaticky při startu systému nebo ručně z podnětu uživatele. Při svém spuštění zaregistruje adresy několika důležitých rutin, které tak vytvoří interface mezi operačním systémem a ovladačem. Prostřednictvím volání těchto rutin operační systém komunikuje s ovladačem. Dále přechází do stavu čekání.

PnP Manager podle informací uložených do registrů při instalaci ovladače ví, jaká zařízení je schopen obsluhovat a jaké hardwarové prostředky potřebuje. Pokud je takové zařízení v počítači nalezeno, je o tom ovladač informován voláním rutiny AddDevice.

Pro každé obsluhované zařízení je v rutině AddDevice vytvořen objekt Device Object, který toto zařízení reprezentuje a umožňuje tak jednomu kódu ovladače obsluhovat více

zařízení. K zařízení ovšem ovladač ještě nemá přiděleny hardwarové prostředky a tak opět přechází do stavu čekání.

Přístup na hardware je umožněn až poté, co ovladači přijde od systému zpráva typu IRP_MN_START_DEVICE. Poté si ovladač zjistí přidělené prostředky a inicializuje hardware. Od tohoto okamžiku může začít plnit požadavky na I/O operace.



Obrázek 4.2: Zpracování IRP

Pokud chce aplikace poslat ovladači nějaký požadavek, musí nejprve k němu otevřít souborový handle, pomocí příkazu `CreateFile` s registrovaným jménem, tvořící identifikátor ovladače, jako například `"\\.\ProfiM"`. Přes tento handle pak posílá ovladači požadavky. Ty se přes API rozhraní dostanou k I/O Manageru (obrázek 4.2). Ten z nich vytváří pakety IRP, které posílá ovladači. Ten požadavky začne zpracovávat a po tuto dobu IRP drží. Po vyřízení požadavku je IRP označen jako hotový a je do něj uložen výsledek. I/O manager potom výsledek posílá zpět aplikaci. Používání IRP paketů zjednodušuje ovladači práci s požadavky, neboť jejich správa je zajištěna I/O managerem.

I/O operace prováděné s fyzickým zařízením jsou vykonávány přes HAL (Hardware Abstraction Layer), který tvoří abstraktní vrstvu skrývající implementační detaily každé konkrétní hardwarové platformy.

4.5 Prioritní úrovně (IRQL)

Velmi důležitou vlastností pro všechny rutiny ovladače je prioritní úroveň, na které jsou spouštěny. Na rozdíl od běžných programů v user módu jsou jednotlivé rutiny ovladače spouštěny na různých prioritních úrovních (IRQL - Interrupt request level). To na jaké úrovní rutina běží je podstatné, protože *určuje jaké funkce jádra může volat*. Pro vyšší úrovně priority tak například nelze volat některé funkce pro alokaci paměti, synchronizační funkce (`KeWaitForSingleObject`,...) nebo třeba funkce pro přístup do registrů. Omezení jsou dána přibližně tak, aby rutiny spouštěné s vyšší prioritní úrovní nepoužívaly funkce potřebující stránkovanou pamět (důvod v části 4.8) nebo takové, které jsou časově náročné. To odpovídá požadavku systému na co nejrychlejší proběhnutí rutin se zvýšenou prioritou.

Úroveň, na které může být funkce volána se tak stává nedílnou vlastností všech funkcí jádra, které ovladač může volat a je potřeba jí vzít v úvahu při každém použití. Tuto úroveň lze najít v dokumentaci [12].

Úroveň, na které rutina poběží, je pro každou standardní rutinu ovladače určena systémem nebo hardwarovou úrovní (DIRQL), na které fyzické zařízení přerušuje (tabulka 4.1).

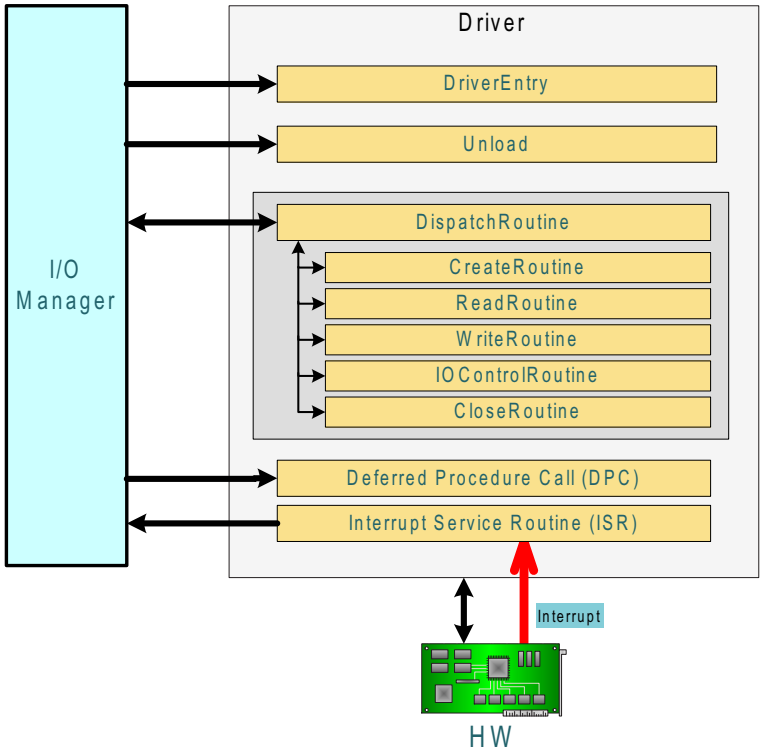
IRQL	Masková přerušení	Standardní rutiny běžící na této úrovni
PASSIVE_LEVEL	žádná	**DriverEntry**, Unload, **AddDevice**, **Dispatch** rutiny a ovladačem vytvořená vlákna.
DISPATCH_LEVEL	Maskuje přerušení s úrovní DISPATCH_LEVEL. Přerušení od fyzických zařízení však mohou nastat.	**DpcForIsr**, CustomTimerDpc, CustomDPC, IoTimer, Cancel, StartIo routiny.
DIRQL	Všechna přerušení s IRQL≤DIRQL. Může nastat přerušení s vyšší prioritou.	**ISR** a SyncCritSection

Tabulka 4.1: Prioritní úrovně

4.6 Základní struktura a rutiny Legacy ovladače

Nejprve popíšeme strukturu Legacy ovladače a v další části ukážeme její rozšíření pro podporu PnP. Legacy ovladač je v nejjednodušší podobě tvořen několika základními rutinami (obrázek 4.3). Ty na jedné straně spolupracují s I/O Managerem, přes který probíhá veš-

kerá komunikace mezi ovladačem a aplikacemi, a na druhé straně přistupuje ovladač na obsluhovaný hardware.



Obrázek 4.3: Struktura Legacy ovladače

4.6.1 DriverEntry

Základní rutina každého ovladače. I/O Manager volá DriverEntry ihned po náhrání ovladače a jeho spuštění. Jako parametr předává ukazatel na objekt Driver Object, kterým I/O Manager reprezentuje instanci ovladače. DriverEntry nastavuje ukazatele na standardní a Dispatch rutiny, reprezentující vstupní body obslužných funkcí pro ovladač, které umožňují spolupráci I/O Manageru s ovladačem:

```
DriverObject->MajorFunction[IRP_MJ_CREATE] = CreateRoutine;
DriverObject->MajorFunction[IRP_MJ_CLOSE]  = CloseRoutine;
DriverObject->MajorFunction[IRP_MJ_READ]   = ReadRoutine;
DriverObject->MajorFunction[IRP_MJ_WRITE]  = WriteRoutine;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IOControlRoutine;
DriverObject->DriverUnload = UnloadDriver;
```

Každá obsluha může mít svoji samostatnou funkci, ale bývá zvykem vytvořit jednu Dispatch rutinu, která obsluhuje všechny požadavky (typ požadavku je předáván jako jeden z parametrů I/O Managerem).

Dále v DriverEntry ovladač alokuje a inicializuje potřebné paměťové struktury pro svoji činnost (viz. 4.8). Zvláštností je obvyklá potřeba ovladače získat prostor v tzv. nonpaged paměti, neboli takové paměti, která není systémem stránkována, tudíž není nikdy uložena na disk. To je nutné především pro rychlý přístup do paměti ve stavech se zvýšenou prioritou (IRQL), kdy přístup k datům uloženým v části paměti, která byla vystránkovaná, by zdržoval anebo nebyl vůbec možný (např. v obslužné rutině přerušení - ISR).

V DriverEntry obvykle také ovladač zjišťuje svoji konfiguraci ukládanou do registrů (viz. 4.9). Cestu k nim může získat dvěma způsoby. Buď mohou být uložené na místě určeném jedním z parametrů, se kterým I/O Manager volá DriverEntry nebo na místě určeném z Driver Objectu podle položky *DriverObject→HardwareDatabase*.

Narozdíl od PnP ovladače provádí Legacy ovladač inicializaci hardwaru zařízení již v DriverEntry. Před prvním přímým přístupem na zařízení však také musí hardwarové prostředky od systému získat. To znamená získat I/O porty, rozsahy paměti s namapovanými registry zařízení nebo přerušení, přes které bude na zařízení přímo přistupovat. Jejich správa je ve Windows NT 4.0 prováděna přes záznamy v registrech, které jsou umístěny v části *\Registry\Machine\Hardware\ ResourceMap*. Pro jejich získání volá ovladač funkce *IoAssignResources*, *IoReportResourceUsage* a *HalAssignSlotResources*. Jestliže při žádání o prostředky nenastane žádný konflikt, jsou požadavky ovladače přidány do registrů. Tento mechanismus zabraňuje ovladačům, aby převzaly již zabrané prostředky jiných ovladačů a způsobily tak konflikt na zařízení, které již jiný ovladač inicializoval.

Po získání prostředků uvádí ovladač svoje zařízení do počátečního stavu a obvykle jej připraví pro generování přerušení. Aby přerušení mohl zpracovávat, potřebuje ještě pomocí funkce *IoConnectInterrupt* zaregistrovat obslužnou rutinu přerušení (ISR).

4.6.2 Unload

Ovladač, který může být odstraněn za běhu systému, musí mít Unload rutinu. Ta je zodpovědná za uvolnění všech systémových objektů a prostředků, které ovladač zabral pro svou činnost. Teprve potom může být ovladač odstraněn. Deaktivace ovladače může způsobit závažnou chybu pokud je použita nevhodná sekvence rušení. Například na začátku rutiny Unload je smazán nějaký objekt, potom ještě před deaktivací přerušení je vyvolána ISR a v té je smazaný objekt použit, což způsobí kritickou chybu systému. Je proto vhodné postupovat podle následujícího postupu:

1. Deaktivovat všechny zdroje přerušení na fyzických zařízeních. Následně od nich odpojit obsluhu přerušení voláním funkce *IoDisconnectInterupt*.

2. Zajistit, že žádná jiná rutina ovladače nebude nadále používat prostředky, které chceme uvolnit - například zavoláním *IoStopTimer* deaktivujeme volání IoTimer rutiny ovladače atd.

3. Uvolnit systémové objekty a prostředky, které byly zabrány při inicializaci ovladače nebo v průběhu jeho činnosti.

4.6.3 Interrupt Service Routine (ISR)

Jedna z nejdůležitějších rutin ovladače fyzického zařízení. ISR je obslužná rutina, která je vyvolána pokud zařízení, které ovladač obsluhuje generuje žádost o přerušení (IRQ). Je volána v režimu zvýšené priority (IRQL=DIRQL, tj. úroveň specifikovaná jako parametr při volání IoConnectInterrupt) a tak svým během maskuje přerušení stejné a nižší prioritní úrovně. To klade vysoké nároky na rychlost zpracování přerušení. ISR by měla proběhnout co nejrychleji, aby systém nepřicházel o vymaskovaná přerušení. Obvykle tak ISR pouze přečte stav zařízení spolu s nejdůležitějsími daty, donutí zařízení ukončit generování přerušení a požádá o DPC neboli Deferred Procedure Call, což je funkce, která je jádrem systému později vyvolána a již ve snížené úrovni priority (IRQL=DISPATCH_LEVEL) dokončí obsluhu přerušení.

Důvodem pro použití DPC rutiny je také to, že některé funkce nemohou být volány v režimu zvýšené priority, ve které běží ISR a tak mohou být provedeny až v DPC.

Jako příklad můžeme uvést příjem dat sériovým kanálem s pamětí FIFO. Naplní-li se FIFO paměť až po určitou mez, vygeneruje sériový kanál přerušení indikující příchod dat. Tím je vyvolána ISR, která data z FIFO paměti nevyčte rovnou, ale pouze si uloží identifikátor typu přerušení, požádá o DPC a rychle vrátí řízení. Vzápětí systém vyvolá DPC rutinu (běžící již s nižší prioritní úrovní IRQL=DISPATCH_LEVEL), která dokončí obsluhu přerušení tím, že vyčte z FIFO paměti přijatá data. DPC již běží v nižší úrovni priority a tak může data zpracovat a případně předat do aplikační vrstvy.

Problémem při použití DPC jako rutiny dokončující obsluhu přerušení po proběhnutí ISR, je možné dlouhé časové zpoždění mezi voláním ISR a DPC. Tudíž se v některých časově náročných případech nevyhneme nutnosti provést celou obsluhu přerušení v ISR.

4.6.4 Deferred Procedure Call (DPC)

Deferred Procedure Call neboli odložené volání procedury - význam této funkce pro použití s rutinou obsluhy přerušení (ISR) byl popsán v předchozím odstavci. DPC určená pro dokončení zpracování obsluhy přerušení bývají také nazývána DpcForIsr.

4.6.5 IoTimer

Potřebuje-li ovladač funkci, která by byla volána v pravidelných intervalech, pak mu systém nabízí rutiny IoTimeru. Nejmenší časový interval s jakým může být volána je kolem $10ms$ a obvykle se používá ke zjištění time-outu I/O operací na zařízení nebo pro pravidelnou kontrolu činnosti ovladače (Watchdog).

Při použití standardní rutiny IoTimeru je systémem zaručeno její pravidelné volání v intervalech přibližně jedné sekundy. Pro spuštění periodického volání rutiny IoTimer se použije funkce *IoInitializeTimer*. Tato funkce není nic jiného než DPC rutina vyvolávaná po přerušení od systémového časovače.

4.6.6 Dispatch Routine

Má-li I/O Manager připravený pro ovladač požadavek ve formě IRP paketu, potom volá zaregistrovanou Dispatch rutinu. Ta jej přebírá a začíná jeho zpracování. O jaký požadavek se jedná je určeno hlavním funkčním kódem IRP paketu (IRP_MJ_xxx). Pro každý hlavní funkční kód může být zaregistrována jedna dispatch rutina. Častokrát ale místo několika oddělených rutin má ovladač rutinu jen jednu, která IRP zpracovává podle hlavního funkčního kódu.

Obecně ovladače zpracovávají následující požadavky, které jsou definovány těmito hlavními funkčními kódy:

IRP_MJ_CREATE znamená požadavek aplikace z user módu na vytvoření souborového handle, které bude představovat komunikační kanál asociovaný s ovladačem.

IRP_MJ_CLOSE indikuje, že aplikace uzavírá komunikaci s ovladačem.

IRP_MJ_READ znamená I/O požadavek na přenos dat z fyzického zařízení do systému.

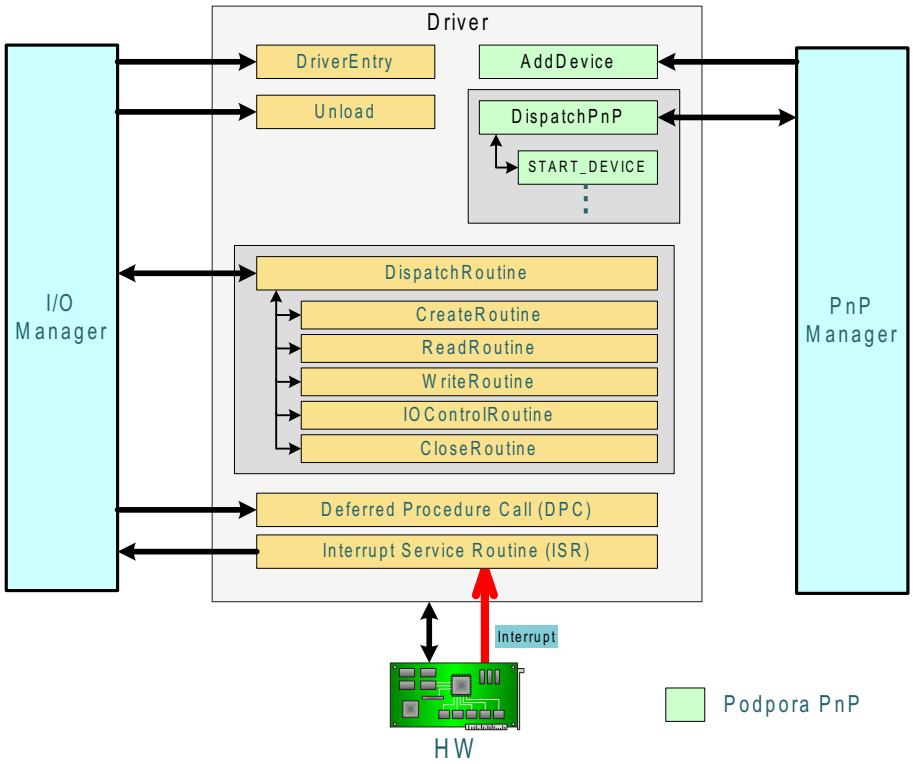
IRP_MJ_WRITE znamená I/O požadavek na přenos dat ze systému do fyzického zařízení. V IRP objektu je ukazatel na buffer, který aplikace poskytla pro požadovaná data.

IRP_MJ_DEVICE_CONTROL indikuje požadavek na ovladač, aby vykonal určitou operaci se zařízením. Ta je určena pomocným kódem IRP_MN_xxx. Jejich významy jsou buď definovány systémem nebo to mohou být vlastní kódy definované konkrétně pro daný ovladač a zařízení.

4.7 Struktura ovladače s podporou PnP

Chceme-li, aby náš ovladač měl zahrnutou PnP podporu, potom do něj musíme přidat několik dalších rutin (obrázek 4.4). Kromě toho se také změní význam několika rutin stan-

dardních a to především kvůli jinému způsobu práce s hardwarovými prostředky.



Obrázek 4.4: Struktura PnP ovladače

4.7.1 DriverEntry

Přidáme-li podporu PnP, pak je rutina DriverEntry zodpovědná pouze za inicializaci ovladače, což kromě definování vstupních bodů obslužných rutin také například znamená inicializaci datových struktur společných pro všechna zařízení podporovaná ovladačem. Samotná inicializace hardware se však přesouvá jinam. O nalezeném zařízení je ovladač informován voláním rutiny AddDevice, ale ani v té na hardware ještě nepřistupuje.

DriverEntry je volána pouze pro první výskyt zařízení podporovaného ovladačem. Jsou-li například v počítači dvě rozšiřující karty, které ovladač podporuje, je při nalezení první z nich PnP Managerem nahrán ovladač, volána DriverEntry a dále AddDevice. Při nalezení druhé karty je již ovladač inicializován a tak přidávání karty začíná až od AddDevice.
Pro rutinu AddDevice definuje DriverEntry pro I/O Manager ukazatel na vstupní bod:

```
DriverObject->DriverExtension->AddDevice = DDAddDevice;
```

a také přidává ukazatele na obslužné funkce pro PnP, Power Management a System Control:

```
DriverObj->MajorFunction[IRP_MJ_PNP]            = DispatchPnp;
DriverObj->MajorFunction[IRP_MJ_POWER]          = DispatchPower;
DriverObj->MajorFunction[IRP_MJ_SYSTEM_CONTROL] = DispatchSystemControl;
```

4.7.2 AddDevice

Rutina nezbytná pro každý ovladač s podporou PnP. Jak již bylo uvedeno, je volána při nalezení zařízení, které ovladač podporuje. Při vyvolání vytváří Device Object reprezentující toto fyzické, logické nebo virtuální zařízení. Všechny informace o ovladačem vytvořených Device Object uchovává I/O Manager v příslušném Driver Objectu, který reprezentuje instanci ovladače. AddDevice tedy pouze provádí přípravu před inicializací nalezeného zařízení.

4.7.3 DispatchPnP

Dispatch rutina přijímající IRP pakety s hlavním funkčním kódem IRP_MJ_PNP, které jsou ovladači posílány PnP Managerem. Požadavky se týkají stavu PnP zařízení. Nejdůležitějším je IRP paket s vedlejším funkčním kódem IRP_MN_START_DEVICE. Spolu s ním získává ovladač hardwarové prostředky a může začít komunikovat se zařízením.

4.7.4 Interrupt Service Routine (ISR)

S přídáním PnP podpory nabývá na důležitosti návratová hodnota ISR rutiny. Ta systému zpětně říká, jestli přerušení bylo pro nás a jestli jsme jej obsloužili. V systémech s PnP totiž může nastat takový případ, že vektor přerušení je sdílen více fyzickými zařízení. Nutným požadavkem na ISR rutinu potom je, aby ihned po svém vyvolání zjistila, jestli přerušení bylo generováno od zařízení, které obsluhuje (obvykle přečtením registru identifikace přerušení - například pro UART registr IIR). Není-li tomu tak, musí vrátit hodnotu FALSE, aby systém vyvolal další rutinu obsluhy přerušení. Nedodržení tohoto požadavku může vést k nefunkčnosti systému.

4.8 Používání paměti ovladačem

Systém Windows používá tzv. virtuální paměť. To mu umožňuje využívat více paměti RAM než je skutečně fyzicky nainstalováno. Dosahuje toho tím, že paměť je rozdělena na menší části - stránky, které mohou být v případě potřeby odloženy do sekundární paměti, tvořené obvykle pevným diskem. Toto odkládání neboli stránkování je obvyklé pro

programy a data v user módu. Pro programy v kernel módu je potřeba některé části paměti před stránkováním chránit.

Konkrétně například v rutině obsluhy přerušení nemůžeme používat žádná data, která nejsou chráněna proti stránkování. Pokud by tato data byla zrovna odložena na disk a nastalo by přerušení, bylo by potřeba je z disku načíst zpět do paměti. To je ovšem časově náročné a to si nemůžeme dovolit, jelikož obsluha přerušení běží v režimu zvýšené priority.

V těchto případech se pro data používá paměti alokované z tzv. nestránkované paměti (non-paged pool). Pro náš případ tj. psaní ovladače bývá zvykem, že většina globálních proměnných a dat je uložená v jedné struktuře, která je jako tzv. „Device Extension“ připojena k objektu, který reprezentuje konkrétní instanci ovládaného zařízení (Device Object). „Device Extension“ je nestránkované a vytváří se při volání `IoCreateDevice` v rutině AddDevice. Ta se volá při přidávání fyzického zařízení a jejím parametrem je velikost požadované nestránkované paměti pro „Device Extension“.

Device Extension je přístupný ze všech rutin ovladače a tak tvoří základní oblast s proměnnými a daty. Pokud potřebujeme alokovat nestránkovanou pamět za běhu ovladače (vytváření front, bufferů atd.) můžeme k tomu použít funkci jádra `ExAllocatePool` s parametrem `PoolType` nastaveným na hodnotu `NonPagedPool`. Ta se pokusí provést alokaci, ale je potřeba mít na paměti, že „non-paged pool“ je omezeným zdrojem systému a tak alokace může skončit neúspěchem.

4.9 Umístění konfigurace v registrech

Ve Windows NT 4.0 jsou informace v registrech týkající se ovladače vztaženy ke službě (Service), která je na ovladač registrována. Konfiguraci tak můžeme uložit do větve *\Registry\Machine\System\CurrentControlSet\Services\DriverName*. To je také cesta, kterou ovladač získá od systému jako parametr rutiny `DriverEntry` volané při spouštění ovladače systémem.

Naproti tomu u PnP ovladače, konfigurace přísluší samotnému konkrétnímu zařízení a je uložena u záznamu zařízení v registrech. Cesta k nim může například pro sériový port COM2 být *\Registry\Machine\System\CurrentControlSet\Enum\ACPI\PNP0501\2\Device Parameters*. Jak je vidět, tak umístění už není tak zřejmé a proto se pro přístup ke konfiguraci ovladače používá systémová funkce `IoOpenDeviceRegistryKey`, která podle daného zařízení otevře správnou větev registrů.

4.10 Ladící prostředky

Součástí snad všech dnešních programátorských balíků (např. C++ Builder) jsou výkonné ladící nástroje. Ty dovolují sledovat běh programu, odchytávat chyby, krokovat program,

přidávat ladící body apod. Programátor, který je na tyto nástroje zvyklý, bude jejich ekvivalent při psaní ovladačů jen těžko hledat, obzvláště pokud píše ovladač pro fyzické zařízení a ne například ovladač k souborovému systému disku.

Důvodů je hned několik. Za prvé ovladače běží ve zvláštním režimu jako součást jádra systému. Chyby na této úrovni v naprosté většině způsobují havárii systému. Navíc většina ovladačů je svázána s fyzickým zařízením, jehož obsluha musí probíhat takřka v reálném čase, což téměř znemožňuje pozastavení běhu programu pro ladící účely. Další potíž způsobuje fakt, že rutiny ovladače (obsluha přerušení, dispatch rutinty, DPC rutiny atd.) mohou být prováděny současně a ještě k tomu na různých prioritních úrovních.

Součástí DDK je ladící program *WinDbg* od Microsoftu, který je však hodně těžkopádný a navíc k ladění ovladačů vyžaduje dva počítače propojené sériovou linkou. Lepším řešením je ladící nástroj *SoftICE* od firmy Compuware. Ten umožňuje ladit ovladače na jednom počítači a jeho možnosti jsou širší.

Přes různé složité ladící programy je nejpoužívanější způsob kontroly činnosti ovladače vypisování textových řetězců přímo z programu pomocí funkce `DbgPrint` (ntddk.h), například:

```
DbgPrint( "ProfiM: Pridelene prostredky Port=0x%x Irq=%d",
          DeviceExtension->Port, DeviceExtension->Irq );
```

Toto volání způsobí vypsání řetězce na standardní ladící výstup. Ten může být zobrazován pomocí volně šiřitelného programu *Debug View* od firmy Sysinternals, který je na přiloženém CD.

4.11 INF File

Nezbytnou součástí ovladače pro operační systémy Windows 2000/XP je tzv. INF file, což je textový soubor, který obsahuje všechny nezbytné informace pro instalaci ovladače do systému. Je v něm uvedeno, které soubory jsou součástí ovladače a kam se mají překopírovat, jaká zařízení ovladač podporuje, informace pro zápis nastavení do registrů a mnoho dalších informací.

Struktura souboru INF pro ovladač je dosti složitá a komplexní a tak se vyplatí pro první verzi použít nástroje `geninf.exe`, který je součástí DDK. Ten po vyplnění několika dialogových oken se základními údaji o ovladači vygeneruje hrubou kostru, avšak ruční úpravě se stejně nevyhneme.

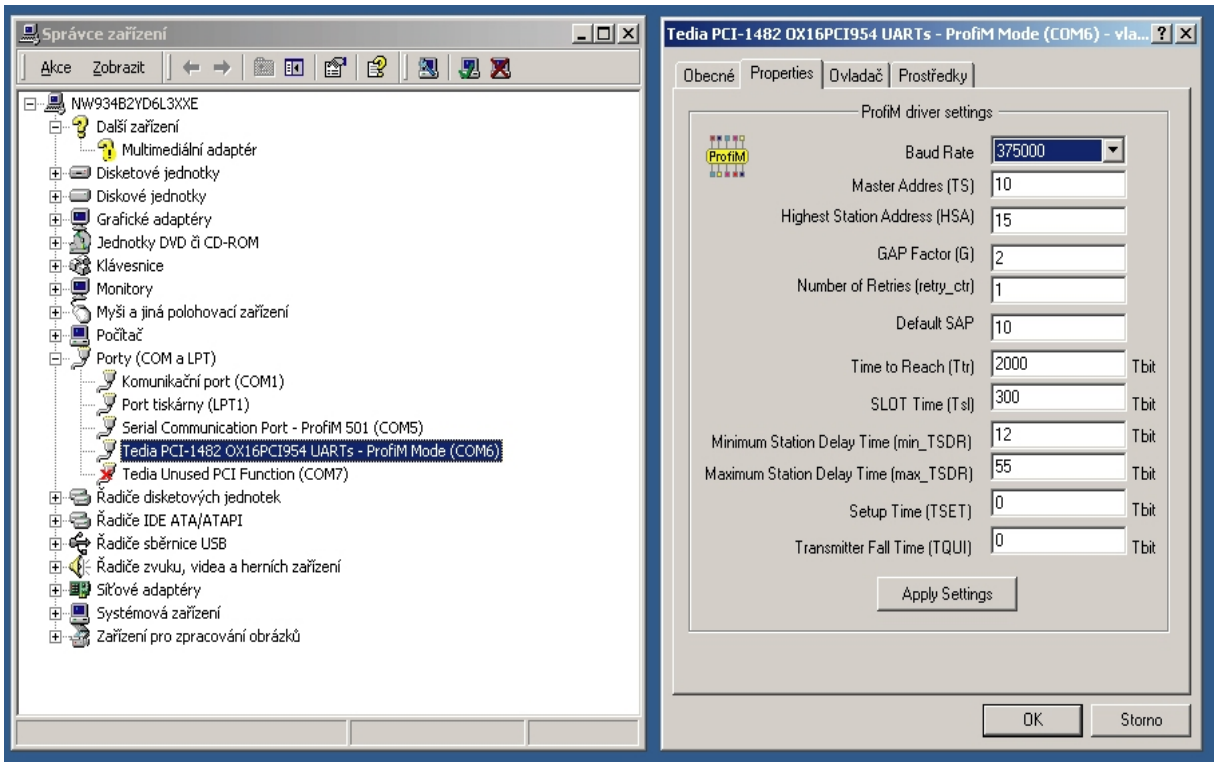
Důležitou informací pro operační systém, kterou INF file obsahuje, je seznam zařízení, které ovladač podporuje a se kterými může pracovat. Tato část může vypadat například takto:

```
[ControlFlags]
ExcludeFromSelect=PCI\VEN_1760&DEV_8004
ExcludeFromSelect=PCI\VEN_1760&DEV_8005
[Tedia]
%PCI\VEN_1760&DEV_8004.DeviceDesc% = PCI_950A,  PCI\VEN_1760&DEV_8004
%PCI\VEN_1760&DEV_8005.DeviceDesc% = NoDrv, PCI\VEN_1760&DEV_8005
```

Operačnímu systému říká, že ovladač podporuje PCI zařízení s číslem výrobce 1760 a číslem zařízení 8004 a 8005. Tato čísla jednoznačně identifikují každé zařízení pro PCI sběrnici a jsou výrobcům hardwaru přidělována mezinárodní organizací PCI–SIG.

4.12 Property Page

Máme-li vytvořený ovladač, téměř jistě také potřebujeme nějakým způsobem nastavovat jeho pracovní parametry. K tomu je nejlepší vytvořit takzvanou Property Page. Ta nám umožňuje přidat několik stránek do okna vlastností zařízení (do něj se dostaneme přes Správce zařízení otevřením okna vlastností pro konkrétní zařízení - obrázek 4.5).



Obrázek 4.5: Property Page

Jinou možností je napsat si vlastní aplikaci, která po svém spuštění umožní nastavování parametrů. Property page má však oproti tomu několik výhod:

- Jeden ovladač může obsluhovat více zařízení - Property page je automaticky používána pro každou instanci a její konkrétní parametry.
- Při otevírání Property page dostaneme od systému cestu do registrů k uloženým parametrům zařízení.
- Není nutno hledat aplikaci pro nastavování parametrů. Property page je přímo přístupná ze Správce zařízení.

Property page je dynamická knihovna DLL, která má operačním systémem definovaný přístupový bod:

```
BOOL APIENTRY PropertyPageProvider( LPVOID Info,
                                    LPFNADDPROPSHEETPAGE AddFunc,
                                    LPARAM Lparam )
```

Potřebuje-li systém zobrazit naší Property page, nahraje dynamickou knihovnu a volá tuto funkci. Ta má za úkol vytvoření okna dialogu, které se přidá jako záložka do okna vlastností zařízení, dále načte parametry uložené v registrech a začne zpracovávat zprávy od systému, které jsou pro smyčku zpráv Property page oproti normálnímu oknu rozšířeny.

Kapitola 5

Realizace linkové vrstvy - ProfiM

Chování linkové vrstvy je velmi dobře popsáno stavovým automatem, jak bylo ukázáno v kapitole 2. Z toho také vychází i způsob jakým je možné linkovou vrstvu realizovat. Ta je vytvořena jako softwarový stavový automat s deseti základními stavy (viz. část 2.2.3), který je po svém spuštění řízen následujícími událostmi:

- příjem požadavku z rozhraní FLD user (rozhraní pro vyšší vrstvy nebo přímý přístup na FDL)
- příjem znaku ze sběrnice (příjem celého rámce)
- událost od časovače (vypršení time-outu, například při čekání na odpověď nebo sledování aktivity na sběrnici)

Tyto události způsobují příslušnou odezvu a přechody mezi stavy. Ačkoliv je princip stavového automatu velice jednoduchý, lze s ním realizovat i velmi složité a komplexni chování.

5.1 Programová implementace

V první fázi byl DP master vytvářen jako běžný program pro user mód. K přístupu na sériový port a jeho ovládání bylo použito standardních služeb operačního systému. Prvním zavažnějším problémem, který se objevil ihned v počátku, bylo ovládání převodníku RS 232/485. Problémem bylo především přepínání směru toku dat, které způsobovalo neustálé potíže.

K řízení převodníku, jak bylo popsáno v kapitole 3.1, nabízejí Windows zdánlivě na první pohled řešení. Pomocí služby `SetCommState`, nastavující parametry komunikačního zařízení, lze aktivovat automatické přepínání směru signálem RTS (nastavením parametru `fRtsControl` na hodnotu `RTS_CONTROL_TOGGLE`). To by mělo zajistit přepnutí směru na vysílání po zápisu dat na port a okamžité přepnutí na příjem po dokončení vysílání. Málo

dokumentovanou skutečností ovšem je, že toto přepínání má při zpětném přepínání na příjem velká zpoždění.

Není totiž implementováno hardwarově (obvody UART až po 16C950 tuto možnost nenabízejí), ale místo toho je k řízení směru použito funkce napojené na systémový časovač s dlouhou periodou. Ta je vyvolávána v intervalech přibližně 20ms a pokud zjistí, že vysílání skončilo, přepne na příjem. Časové zpoždění, které tak může vzniknout mezi vysláním posledního znaku rámce a přepnutí zpět na příjem je ovšem pro použití na Profibus příliš dlouhé. Dotazovaná stanice na sběrnici totiž pak může začít vysílat odpověď na náš požadavek ještě v době, kdy výstupy převodníku stále drží sběrnici, čímž vznikne kolize.

Ruční přepínání směru vysílání převodníku se také ukázalo jako téměř nemožné z důvodů popsaných v 3.1, navíc pro dosažení spolehlivé odezvy mastera a použití převodníku k dosažení časování s velmi krátkými periodami se ukázal user mód jako nevhodný, díky jeho malým možnostem pro aplikace reálného času. Proto byl DP master přepsán na ovladač pro běh v kernel módu. To také umožnilo později přidat podporu PCI zásuvných karet s obvody UART jako alternativu připojení Profibusu na vysokých přenosových rychlostech.

S přechodem na ovladač však bylo potřeba přepsat celý zdrojový program z jazyka C++ na C. První verze byla totiž celá postavena s využitím objektově orientovaného programování, to ovšem jazyk C neumožňuje a tak bylo nutné objekty odstranit a vrátit se k neobjektovému přístupu.

Ovladač byl nejprve vytvořen pouze pro operační systém Windows NT 4.0, který u ovladačů nevyžaduje podporu PnP. Ta byla později přidána a tak je možné jej používat i pro operační systémy Windows 2000 a XP.

5.2 Nároky Profibusu

Pro umožnění práce s Profibusem potřebujeme obecně zajistit plnohodnotnou schopnost komunikovat po sběrnici RS–485 a především mít možnost časování s krátkými periodami, což představuje velkou potíž, pokud takových period potřebujeme dosáhnout v systému Windows. Pro představu délek těchto period, které potřebujeme dosáhnout si vezměme příklad Profibusu běžícího na rychlosti 1,5Mbps. Všechny časové intervaly pro Profibus, jako délky time-outů, maximální zpoždění a nezbytné časové prodlevy, jsou udávány s jednotkou t_{bit}. Ta je definována jako interval daný délkou přenosu jednoho bitu po sběrnici, neboli převrácenou hodnotou přenosové rychlosti $t_{bit} = 1/$(přenosová rychlost v bps). Pro rychlost 1,5Mbps vychází $t_{bit} \doteq 0,67\mu s$. Z praktického hlediska vzhledem k délkám časovaných period stačí přesnost rozlišení asi desetkrát menší, takže pro rychlost 1,5Mbps potřebujeme dosáhnout minimálních period asi $6\mu s$, což je z pohledu user módu Windows nedosažitelné. Tam je minimální dosažitelná perioda v řádech milisekund.

Jedinou užitečnou funkcí, kterou Windows nabízejí pro práci s krátkými časovými inter-

valy je Perfomance Counter. Ten dovoluje zjišťovat časové okamžiky s vysokým rozlišením. Jedná se o čítač, jehož časové rozlišení dosahuje na většině počítačů přesnosti kolem $1\mu s$, což je pro naše potřeby dostatečné.

Důvodem proč se podařilo v ProfiMu zajistit generování tak krátkých period je efektivní využití možností přerušení od sériového portu s připojeným převodníkem RS 232/485, případně přerušení od rozšiřující karty. To si můžeme dovolit hlavně díky tomu, že je ProfiM napsaný jako ovladač s přímým přístupem na hardware.

5.3 Ovládání převodníku a jeho využití k časování

Jak již bylo popsáno v části 3.1 u standardního sériového portu na PC nemáme k dispozici přerušení od prázdného výstupního posuvného registru. Musíme si proto pomoci využitím hardwarových vlastností převodníku (obrázek 3.1), jako je loop-back vracející všechna vyslaná data výstupem TxD na vstup RxD (při přepnutém převodníku na vysílání) a propojení mezi výstupem TxD a vstupem DSR.

Využívání sériového portu spolu s převodníkem, které zajišťuje přístup na RS–485 a zároveň dostatečně rychlé časování, řeší následující módy činnosti:

Vysílání znaků

Před začátkem vysílání rámce (bloku dat) přepneme převodník na vysílání (RTS=1). Díky loop-backu v převodníku bude každý vyslaný znak zpětně také přijat (obrázek 5.1a). S příchodem každého přerušení indikujícího příjem znaku zjišťujeme jestli se nejedná o poslední znak rámce. S jeho příchodem můžeme přepnout převodník zpět na příjem (RTS=0). K přepnutí tak nemůže dojít v nevhodnou chvíli (uprostřed posledního vysílaného znaku), neboť ho provádíme až po příjmu přerušení indikujícího zpětný příjem posledního znaku, kdy znak už musel být celý vyslán a nikoliv po přerušení indikujícím prázdný vysílací registr.

Příjem znaků

RTS je v logické nule, čímž jsou výstupy budičů sběrnice ve stavu vysoké impedance a sériový kanál tak může přijímat znaky ze sběrnice (obrázek 5.1b). Tento stav se ovšem při práci ProfiMu téměř vůbec nevyužívá, neboť pokud nevysíláme pak pokaždé běží nějaký time-out, například time-out při čekání na odpověď, čemuž odpovídá následující mód.

Časování se současným příjmem znaků

Nejnáročnější a také nejdůležitější mód. V něm je převodník přepnutý na příjem (RTS=0), takže smyčka loop-backu je přerušena a převodník tak může přijímat znaky ze sběrnice (obrázek 5.1b). Zároveň je však požadováno odměření nějakého časového intervalu (většinou time-out).

Časováním od sériového portu získáme rozlišení přibližně $11t_{bit}$, což odpovídá intervalu potřebnému k vyslání jednoho znaku, který má kromě osmi datových bitů ještě jeden start bit, jeden stop bit a jeden sudý paritní bit. Požadovaný interval v jednotkách t_{bit} tak vydělíme hodnotou 11 a výsledná hodnota pak určuje kolik znaků je potřeba vyslat pro odměření požadovaného intervalu. Jelikož je smyčka loop-backu přerušena, můžeme začít do převodníku vysílat libovolné znaky - pro naše účely je, jak uvidíme později, vysílán znak `0x00`. Ty se nám tak nedostanou na vnější sběrnici a ani nijak nenarušují příjem znaků.

Opět zde musíme vyřešit problém týkající se přepínání směru toku dat na převodníku. V tomto módu sice na vnejší sběrnici nevysíláme, ale po příjmu požadovaných dat nebo vypršení time-outu musíme zajistit korektní přepnutí na vysílání, které obvykle následuje.

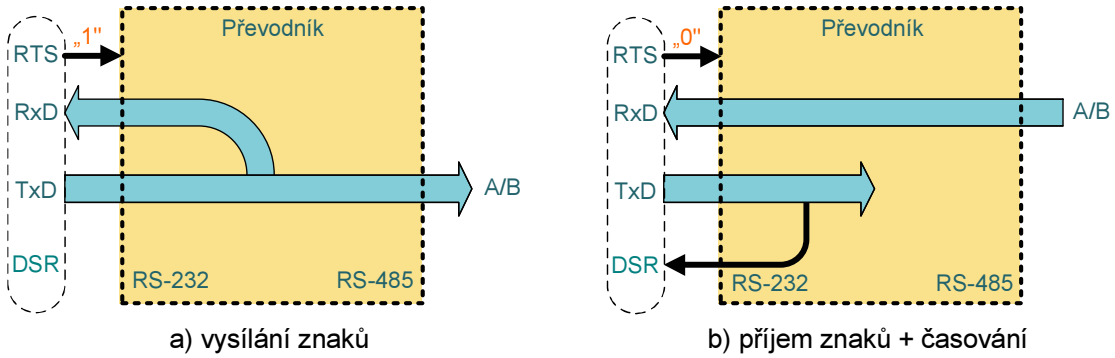
Dokončení vysílání každého znaku je indikováno přerušením oznamující prázdný vysílací registr, tím dostáváme signál k vyslání dalšího znaku. Když nám přijde přerušení oznamující vyslání posledního znaku, máme problém. Vysílací registr je sice prázdný, ale znak je stále vysouván z výstupního posuvného registru. Řešení pomocí loop-backu z módu vysílání znaků je zde nepoužitelné, neboť loop-back je rozpojen. Okamžité přepnutí směru s příchodem posledního znaku by způsobilo to, že se časovací znak dostane na sběrnici, což je nepřijatelné.

K řešení tohoto problému je v zapojení převodníku (obrázek 3.1) propojka mezi výstupem TxD a vstupem DSR. Tento vstup nám umožňuje generovat tzv. „přerušení od modemu“. Toho využijeme následujícím způsobem. Po příjmu přerušení od posledního vysílaného znaku povolíme přerušení od modemu. Toto přerušení, díky tomu, že vysílaný časovací znak má hodnotu `0x00`, přijde v okamžiku, kdy je na převodník vysílána náběžná hrana stop bitu. To už si můžeme dovolit přepnout směr na vysílání. Jedničková hodnota stop bitu znamená klidovou hodnotu na sběrnici se standardem RS–485. Po přepnutí ještě zakážeme přerušení od modemu, aby nám další vysílaný znak nezpůsobil přerušení.

Synchronizační mezera

Využívá se pokud před vysílaný datový rámec potřebujeme vložit časovou prodlevu. Například pro každý rámec dotazu je normou Profibusu vyžadována prodleva o délce $33t_{bit}$. Funguje podobně jako předchozí mód činnosti, pouze neočekává příjem žádných znaků.

Ve skutečnosti ProfiM funguje tak, že na převodník jsou neustále vysílány znaky. Buď se jedná o data určená pro vnější sběrnici RS–485, tedy Profibus, nebo jsou to znaky zajišťující časování, které na vnější sběrnici neprojdou.



Obrázek 5.1: Využití převodníku RS 232/485

Použijeme-li rozšiřující kartu s obvodem UART typu 16C450 až 16C750, potom způsob přístupu na sběrnici Profibus a způsob časování je stejný. Rozdíl je pouze v tom, že některé karty již mají výstup ve standardu RS–485, avšak nutnost softwarově ovládat směr zůstává (kvůli možnosti časování).

5.4 Časování s obvodem 16C950

S tímto obvodem je práce výrazně jednodušší, jelikož se jedná o obvod, který dává k dispozici přerušení od prázdného výstupního posuvného registru. Tím nám odpadají všechny složité programové konstrukce, které se snažily tento nedostatek obejít.

Na druhou stranu směr toku dat na sběrnici s RS–485 je stále řízen programově a zůstávají i principy použité pro časování. Pouze není potřeba hardwarový loop-back a propojka mezi výstupem TxD a vstupem DSR, což byly hardwarové prostředky převodníku, které umožnily nahradit přerušení od prázdného výstupního posuvného registru.

Použití tohoto obvodu je velice spolehlivé a rychlostní limit přenosové rychlosti je pouze otázkou výkonu procesoru a maximální přípustné míry zátěže procesoru.

5.5 Princip činnosti ProfiMu

Blokové schéma ProfiMu je na obrázku 5.2. Ústředním blokem je stavový automat FDL vrstvy. Tomu jsou podřízeny všechny ostatní části.

Přístup na sběrnici probíhá přes blok „Řízení Tx/Rx, časování“. Ten tvoří rozhraní k fyzické vrstvě a proto je jeho implementace závislá na použitém hardware, neboť na něj přímo přistupuje a využívá jeho možností. Obecně je funkcí tohoto bloku zajišťovat přístup na sběrnici a to hlavně při nutnosti použít převodník (port se standardem RS–232).

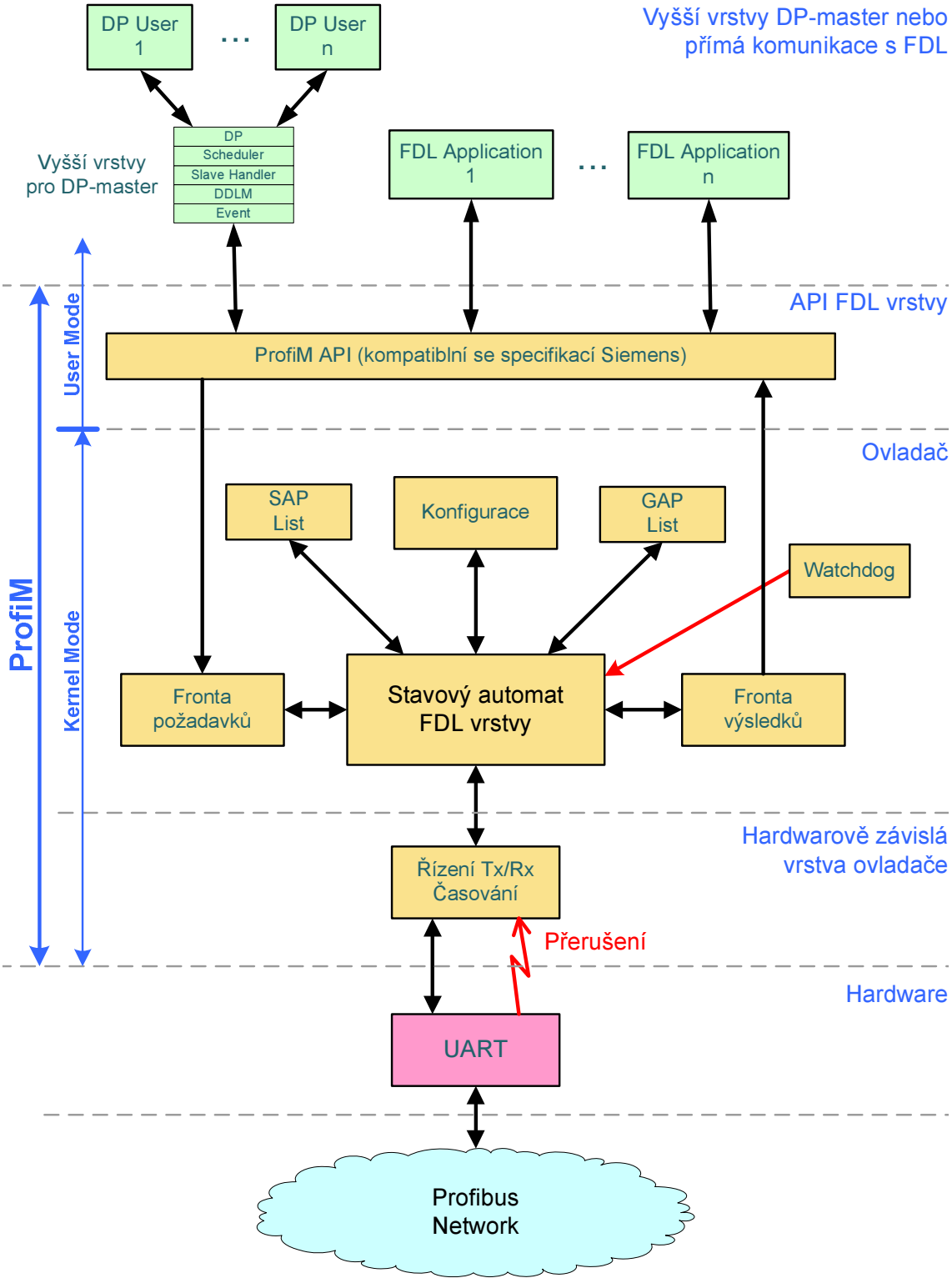
Pro stavový automat zajišťuje vysílání rámců, indikaci události příjmu znaku ze sběrnice a především zajišťuje časování krátkých intervalů, k čemuž využívá možností hardwarové vrstvy a případného převodníku (zejména možností hardwarových přerušení). Ve verzi ProfiMu, která je součástí této diplomové práce podporuje tento blok standardní sériový port RS–232 a PCI rozšiřující karty s obvodem OX16PCI954.

Z druhé strany je směrem k vyšším vrstvám orientované API rozhraní FDL vrstvy. To je tvořeno statickou knihovnou `fdl_rb.lib` (název je zvolen k zachování maximální kompatibility s řešením od Siemensu [6]), která je přilinkována k programům vyžadující komunikaci s FDL vrstvou. Nejprve si program otevře komunikační kanál k FDL vrstvě a potom může začít posílat své požadavky (popis v kapitole 6). Požadavky jsou podle priority (high/low) ukládany do dvou front.

Získá-li náš master iniciativu na sběrnici (získáním tokenu), vybere z front požadavků jeden požadavek a začne jej zpracovávat. Většina požadavků představuje komunikaci po sběrnici, ale některé jsou pouze lokální (například zjištění konfigurace mastera nebo aktivace SAP). Po zpracování požadavku je do fronty výsledků přidán výsledek. Ten kromě odpovědí nebo potvrzení příznivě zpracovaného požadavku může znamenat chybu nebo time-out.

Výsledky jsou z fronty vybírány aplikací přes API rozhraní FDL vrstvy. Důležitou vlastností při komunikaci mezi aplikací a API rozhraním FDL vrstvy je to, že z hlediska aplikace má každý otevřený komunikační kanál s FDL vrstvou vlastní frontu požadavků a výsledků. To znamená, že jednu FDL vrstvu (jednoho DP mastera) může najednou sdílet více aplikací, aniž by docházelo k vzájemné kolizi mezi požadavky a výsledky. Prostředky, které DP master nabízí, jako jsou například přístupové body SAP, ovšem musejí být sdílené.

Jelikož implementujeme stanici typu master, zajišťuje stavový automat FDL vrstvy kromě služeb pro vyšší vrstvy také svůj podíl na správě a řízení komunikace na Profibusu. K tomu patří, sledování provozu na sběrnici k detekci chybových stavů, vytváření a aktualizaci seznamu aktivních stanic LAS a seznamu GAP, který pro rozsah adres příslušející naší stanici master sleduje obsazenost jednotlivých adres Profibusu.



Obrázek 5.2: Blokové schéma ProfiMu

5.6 Watchdog

Ke zvýšení spolehlivosti byl do ProfiMu přidán watchdog. Ten sleduje aktivitu mastera a v případě jeho delší nečinnosti jej resetuje.

Prakticky je realizován jako rutina, která je nezávislým systémovým časovačem vyvolávána v pravidelných intervalech přibližně jedné sekundy. K zjištění nečinnosti mastera je použito příznaku aktivity (proměnná), který je při činnosti mastera v pravidelných intervalech nastavován. Po vyvolání rutiny watchdogu je tento příznak zkontrolován a ihned resetován. Není-li při kontrole watchdogem tento příznak nastaven, znamená to nějakou chybu a je proveden reset mastera.

Watchdog se hlavně uplatní při chybách v přerušovacím systému nebo v případě použití převodníku, kdy při odpojení převodníku z portu dojde k zastavení činnosti stavového automatu, jehož činnost je na převodníku přímo závislá (neplatí pro zásuvné karty s obvodem 16C950). Watchdog potom zajišťuje opětovné spuštění mastera po opětovném připojení převodníku.

5.7 Zatížení procesoru

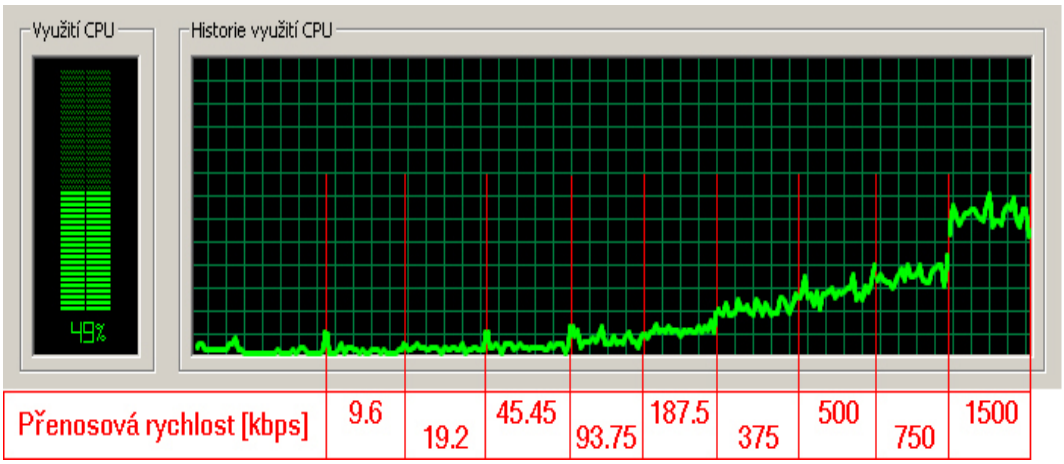
Jelikož ProfiM při implementaci Profibus DP mastera nepoužívá žádný speciální hardware, nese celé zatížení pouze procesor. Specifikem Profibusu je navíc to, že provoz na sběrnici neutichá ani v případech, kdy nejsou přenášena žádná data. Každá stanice DP master se totiž musí účastnit logického kruhu předávání tokenu, vyhledávat změny v jí příslušejícím adresovém GAP prostoru a navíc neustále sledovat provoz a analyzovat všechny rámce.

Graf zatížení procesoru naměřený s ProfiMem při použití různých přenosových rychlostí je na obrázku 5.3, průměrné číselné hodnoty jsou shrnuty v tabulce 5.2. Hodnoty zatížení procesoru byly změřeny na počítači s procesorem AMD Athlon XP 1600+ a rozšiřující kartou sériových portů Tedia PCI-1482 s obvodem OX16PCI954.

Jak je vidět, pro nižší rychlosti nepředstavuje zátěž procesoru větší problém. Především při použití standardního sériového portu, kdy máme k dispozici pouze rychlosti 9600bps a 19200bps, se zvýšení zátěže téměř neprojevuje. Pokud ale chceme pracovat na vysokých přenosových rychlostech, je nutné zvýšenou zátěž vzít v úvahu a případně ještě upravit časové parametry sítě, které dovolí větší časová zpoždění jako například T_{SL} a $maxT_{SDR}$.

Baudrate [kbps]	9,6	19,2	45,45	93,75	187,5	375	500	750	1500
Zatížení CPU [%]	∼1%	∼1%	∼2%	6%	10%	17%	20%	25%	50%

Tabulka 5.2: Zatížení CPU podle přenosové rychlosti



Obrázek 5.3: Zatížení CPU podle přenosové rychlosti

Kapitola 6

Aplikační rozhraní FDL vrstvy

Aplikační rozhraní FDL vrstvy ProfiMu je vytvořeno podle standardu definovaného firmou Siemens [6]. Podle tohoto standardu probíhá komunikace s FDL vrstvou pomocí jednotné datové struktury nazývané Request Block (RB). Tato struktura představuje jakýsi univerzální rámec, který může představovat nejen jakýkoliv požadavek, ale také odpověď na požadavek nebo příznak události. Velikost Request Blocku je přibližně 600B a obsahuje mnoho položek, z nichž pouze některé jsou v daný okamžik využité.

Komunikace probíhá tak, že aplikace (vyšší vrstva DP master nebo přímo FDL aplikace) naplní strukturu Request Block, aby pomocí ní předala požadavek (Request) FDL vrstvě. Podle typu požadavku jsou naplněny pouze určité položky této struktury a ostatní jsou ponechány nevyužité. Následně je struktura poslána FDL vrstvě, ta podle požadavku vykoná danou činnost a jako odpověď (Confirmation) pošle zpět aplikaci opět Request Block. Ten je poslán jak v případě úspěšného vyřízení požadavku, tak i v případě nějaké chyby.

Request Block je také využíván jako indikace události v FDL vrstvě (Indication). Nastane-li nějaká událost (např. příchod dat od jiného mastera) posílá FDL vrstva Request Block aplikaci s příslušně nastavenými položkami podle typu události.

Aplikační rozhraní tvoří pět jednoduchých funkcí `SCP_xxx` pro přímou komunikaci s ovladačem pomocí struktury Request Block:

SCP_open(char *name)
Otevře komunikační kanál mezi aplikací a FDL vrstvou.
Parametry `name` určuje jméno ovladače. První instance ProfiMu má jméno „\\.\ProfiM“ další potom „\\.\ProfiM1“, „\\.\ProfiM2“...
Návratová hodnota Funkce vrací handle ke komunikačnímu kanálu mezi ovladačem a aplikací. Tuto hodnotu používají všechny další funkce k volání ovladače. Nepodaří-li se handle vytvořit, vrací funkce systémem definovanou hodnotu `INVALID_HANDLE_VALUE`.

SCP_send(int handle, UWORD length, char *rb)	
Pošle FDL vrstvě Request Block reprezentující požadavek.	
Parametry	
handle	komunikační kanál otevřený pomocí SCP_open
*rb	ukazatel na posílaný Request Block
length	délka Request Blocku
Návratová hodnota	
SCP_SUCCESS	ovladač přijal Request Block.
SCP_ERROR	při předávání došlo k chybě. Konkrétní kód chyby vrátí funkce SCP_get_errno.

SCP_receive(INT handle, UWORD timeout, UWORD *data_len, UWORD length, CHAR *buffer)	
Tato funkce umožňuje získávat od FDL vrstvy zpět data, výsledky požadavků a indikace událostí. Přenášená data jsou opět ve formě Request Blocku a aplikace na něj může podle nastavení parametru timetou čekat synchronně nebo asynchronně.	
Parametry	
handle	komunikační kanál otevřený pomocí SCP_open
timeout	určuje v milisekundách jak dlouho bude funkce čekat na výsledek. Kromě počtu milisekund lze použít hodnoty SCP_NOWAIT, která způsobí, že data budou vrácena pouze pokud jsou v době volání funkce již k dispozici a nebude se na ně čekat. Jinou možnou hodnotou je SCP_FOREVER, která způsobí, že funkce bude čekat dokud výsledek nebude k dispozici
*data_len	vrátí délku dat zapsaných do bufferu
length	velikost bufferu pro přijímaná data - měl by mít velikost alespoň jako Request Block
*buffer	ukazatel na buffer pro vrácená data
Návratová hodnota	
SCP_SUCCESS	přečtení Request Blocku bylo úspěšné.
SCP_ERROR	při čtení nastala chyba. Konkrétní kód chyby vrátí funkce SCP_get_errno.

SCP_close(int handle)	
Uzavře a uvolní handle komunikačního kanálu k FDL vrstvě.	
Parametry	
handle	určuje komunikační kanál k uzavření
Návratová hodnota	
SCP_SUCCESS	komunikační kanál se úspěšně podařilo uzavřít.
SCP_ERROR	chyba při uzavírání komunikačního kanálu. Konkrétní kód chyby vrátí funkce SCP_get_errno.

SCP_get_errno()
Je volána pokud se některá z předchozích funkcí nezdaří a vrátí hodnotu `SCP_ERROR`.
Parametry -
Návratová hodnota Vrácená hodnota udává konkrétní chybový kód. Tyto chybové kódy jsou popsány v [6].

6.1 Služby linkové vrstvy

FDL vrstva nabízí následující služby datové výměny (blíže [6]):

- SDA
- SDN
- REPLY_UPDATE_SINGLE
- REPLY_UPDATE_MULTIPLE

a následující služby správy a řízení linkové vrstvy:

- FDL_READ_VALUE
- SAP_ACTIVATE
- RSAP_ACTIVATE
- SAP_DEACTIVATE
- LSAP_STATUS
- FDL_LIFE_LIST_CREATE_REMOTE
- FDL_LIFE_LIST_CREATE_LOCAL
- FDL_IDENT
- FDL_READ_STATISTIC_COUNTER
- AWAIT_INDICATION
- FDL_EVENT
- WITHDRAW_INDICATION

pro každou z těchto služeb je určen formát a význam jednotlivých položek Request Blocku podle [6]. Pro zjednodušené vytváření správných Request Blocků pro tyto požadavky přidává ProfiM několik dalších funkcí. Ty podle parametrů vytvoří Request Block odpovídající typu požadavku a pošlou jej FDL vrstvě. Nejpoužívanější z těchto funkcí jsou následující:

SAP_activate(int DriverHandle, BYTE sap_nr, BYTE ACCSAP, BYTE ACCSTAT, BYTE SDA_R, BYTE SDN_R, BYTE SRD_R, BYTE priority)
Aktivuje přístupový bod Service Access Point s číslem `sap_nr`.

Send_SRD_Hex(int DriverHandle, BYTE adr, UBYTE ssap, UBYTE dsap, unsigned char *data, BYTE priority)
Pošle FDL vrstvě požadavek na vyslání datového rámce stanici s adresou `adr` s očekáváním odpovědi - Send and Request Data (SRD). Umožňuje zadávat data v hexadecimálním formátu ve formě řetězce - např. `"B8 12 45"`.

Send_SRD_Bin(int DriverHandle, BYTE addr, BYTE ssap, BYTE dsap, unsigned char *data, int length, BYTE priority)
Pošle FDL vrstvě požadavek Send and Request Data (SRD) s daty v binárním tvaru.

Send_SDN_Hex(int DriverHandle, BYTE addr, BYTE ssap, BYTE dsap, unsigned char *data, BYTE priority)
Pošle FDL vrstvě požadavek na vyslání datového rámce stanici s adresou `adr` bez očekávání odpovědi - Send Data with no Acknowledge (SDN). S daty zadanými v hexadecimálním formátu ve formě řetězce - např. `"B8 12 45"`.

Send_SDN_Bin(int DriverHandle, BYTE addr, BYTE ssap, BYTE dsap, unsigned char *data, int length, BYTE priority)
Pošle FDL vrstvě požadavek Send Data with no Acknowledge (SDN) s daty v binárním tvaru.

Read_FDL_value(int DriverHandle, BYTE priority)
Umožňuje zjistit aktuální parametry sítě.

FDL_life_list_remote(int DriverHandle, BYTE priority)
Vytvoří seznam aktivních stanic na sběrnici.

6.2 Komunikace s FDL vrstvou

Způsob komunikace s FDL vrstvou obvykle záleží na složitosti programu (příklady jsou uvedeny v příloze A). V jednoduchých aplikacích si většinou vystačíme se synchronním přístupem. Aplikace vyšle požadavek a pomocí `SCP_receive` s parametrem timeout>0 začne čekat na výsledek. Při tomto volání `SCP_receive` je program v místě volání pozastaven, dokud FDL vrstva nevrátí výsledek pro aplikaci. Tento způsob je přímočarý, ale neumožňuje posílat více požadavků najednou a reagovat na události v FDL vrstvě.

Jiným komplexnějším způsobem je vytvoření samostatného threadu, který slouží pouze pro příjem Request Blocků z FDL vrstvy. Ten po svém spuštění zavolá `SCP_receive` s parametrem timeout=`SCP_FOREVER` a začne čekat na Request Block. S jeho příchodem je Request Block předán ke zpracování a thread opět volá `SCP_receive` s parametrem timeout=`SCP_FOREVER`. Přijímací thread ovšem potřebuje rozlišit pro koho je výsledek určen

(požadavků může být od jedné aplikace zpracováváno více najednou). K tomu se používá položka Request Blocku s názvem `user`, která je určena pro potřeby aplikace a FDL vrstva její hodnotu při zpracování požadavku zachovává.

Pro využití ProfiMu při psaní aplikací je připravena statická knihovna `fdl_rb.lib` a hlavičkový soubor se všemi potřebnými definicemi `fdl_rb.h`. Jména jsou zvolena stejně s originálními knihovnami firmy Siemens. K použití ProfiMu v projektu, který původně používal řešení od Siemensu pak pouze stačí tyto dva původní soubory nahradit soubory ProfiMu, změnit jméno otevíraného zařízení ve funkci `SCP_open` z například obvyklého „/CP_L2_1:/FLC“ na „\\.\ProfiM“ a tím by měla být náhrada hotová.

Kapitola 7

Závěr

Přes nemalé potíže, které provázely vývoj této diplomové práce, se nakonec podařilo dosáhnout cíle a snad i překonat původní očekávání. Výsledkem je realizace Profibus DP Mastera na PC, vytvořeného až po FDL vrstvu, včetně. Práce umožňuje připojit Profibus přes sériový port s přenosovými rychlostmi 9600bps a 19200bps nebo pro dosažení vyšších rychlostí použít PCI karty s obvodem UART (16C950). S použitím rozšiřující karty se podařilo dosáhnout až rychlosti 3Mbps, avšak to byla hranice dána pouze výkonem procesoru (konfigurace testovací sestavy viz. 5.7).

Navíc se podařilo propojit tuto diplomovou práci s jinou diplomovou prací z předchozích let [2], která implementovala vyšší vrstvy Profibus DP Mastera, čímž vznikla zcela plnohodnotná implementace ve všech vrstvách až po aplikační.

Dá se předpokládat, že o tuto práci by mohl vzniknout zájem ze strany firem pracujících v oboru, neboť možnost realizovat Profibus DP Mastera s minimálními náklady, jako umožňuje tato práce, bude pro ně určitě lákavá. Obzvláště proto, že nic podobného ještě pro PC nebylo vytvořeno a to z důvodů zřejmé implementační náročnosti.

Softwarová realizace byla vytvořena jako ovladač pro operační systémy Windows NT, 2000 a XP, který navíc na jednom počítači dokáže obsluhovat až několik připojení Profibusu a může být synchronně nebo asynchronně využíván několika aplikacemi.

Jelikož prostředí operačních systémů Windows nejsou určena pro aplikace reálného času, nelze pro vysoké přenosové rychlosti úplně zaručit, že DP Master tvořený ProfiMem pokaždé dodrží časové intervaly dané normou. Na druhou stranu se to stává výjimečně a i v nejhorších případech je návrhem Profibusu zaručeno bezproblémové znovuobnovení činnosti sběrnice. Z toho také vyplývá nemožnost nasadit ProfiM pro aplikace vyžadující vysokou spolehlivost.

Pro dobrou možnost k využití ovladače aplikacemi bylo vytvořeno aplikační rozhraní FDL vrstvy, která je kompatibilní s aplikačním rozhraním používaným firmou Siemens. U programů, které jej využívají to dovoluje velice jednoduše vyměnit původní hardwarové

řešení za ProfiM. K výměně stačí překopírovat do projektu statickou knihovnu ProfiMu a změnit jméno otevíraného zařízení ve funkci `SCP_open`.

V diplomových pracích z předchozích let, které se zabývaly Profibusem, bylo zvykem jako důkaz funkčnosti přidávat do diplomové práce výpisy provozu na sběrnici z analyzátoru Profibusu. To se mi zdá ovšem nepřehledné a málo průkazné a proto jako ukázka správné činnosti Profibus DP mastera byla ve strojovně na Karlově náměstí vytvořena jednoduchá ukázková aplikace, kde pomocí počítače s nainstalovaným ProfiMem jsou řízeny dva modely pásového dopravníku a manipulátor s krokovými motory (příloha B). Všechny snímače a akční členy jsou připojeny na dva moduly vstupů a výstupů, které jsou jako jednotky slave připojeny na Profibus. Navíc sběrnice přes kterou probíhá komunikace, může být sdílena s dalším masterem, kterým je PLC od firmy Siemens.

7.1 Co by šlo vylepšit nebo přidat

- Přepsat ProfiM pro některý z operačních systémů reálného času, jako je například RT–Linux, nebo použít nějakou real-timovou nástavbu pro Windows. To by mohlo zajistit vysokou spolehlivost chodu a přesné dodržování časových parametrů sběrnice.
- Přidat do ovladače podporu pro Power Management. K správné funkci není nezbytná, avšak měla by být jeho součástí.
- Další optimalizací kódu snížit zátěž procesoru, což by umožňovalo dosahovat vyšších přenosových rychlostí.

Příloha A

Příklad použití ProfiMu

Chceme-li použít ProfiM v programu, který původně využíval pro přístup k Profibusu rozhraní kompatibilní se Siemensem nebo chceme-li vytvořit nový program, přidáme do projektu statickou knihovnu „fdl_rb.lib“ a hlavičkový soubor „fdl_rb.h“.

Jednoduchý příklad

Následující část kódu ukazuje nejjednodušší možnost použití ProfiMu. Jedná se o jednovláknový program, který inicializuje modul vstupů a výstupů Siemens ET-200B a provede jednu iteraci datové výměny. Všechna čekání na výsledek `SCP_receive` jsou blokující a zastavují běh programu.

```
#include "fdl_rb.h"

int DriverHandle;
int SlaveAddress = 13;

void main()
{
  unsigned short  length;
  fdl_rb          rb;
  BYTE            Buffer[256];
  BYTE            out, in1, in2, in3;

  /* Otevreni komunikacniho kanalu k ovladaci */
  DriverHandle = SCP_open ( "\\\\.\\ProfiM" );

  /* Aktivace pristupovych bodu SAP */
  SAP_activate( DEFAULT_SAP, ALL, ALL, SERVICE_NOT_ACTIVATED,
                BOTH_ROLES, BOTH_ROLES, high );
  SCP_receive(DriverHandle,3000,&length,sizeof(rb),(char*) &rb);
  SAP_activate( 62, ALL, ALL, SERVICE_NOT_ACTIVATED,
```

```
                INITIATOR, INITIATOR, high );
  SCP_receive(DriverHandle,3000,&length,sizeof(rb),(char*) &rb);

  /* Zaslani parametrizacniho ramece */
  Send_SRD_Hex(DriverHandle,SlaveAddress,62,61,
               "B0 08 09 0B 00 0F 00 00 00 00 00 00",high);
  SCP_receive(DriverHandle,3000,&length,sizeof(rb),(char*) &rb);

  /* Zaslani konfiguracniho ramece */
  Send_SRD_Hex(DriverHandle,SlaveAddress ,62,62,"20 12",high);
  SCP_receive(DriverHandle,3000,&length,sizeof(rb),(char*) &rb);

  /* Jedna iterace datove vymeny */
  Buffer[0] = out; /* hodnota vystupu */
  Send_SRD_Bin(DriverHandle,SlaveAddress,DEFAULT_SAP,DEFAULT_SAP,
               Buffer,1,high);
  SCP_receive(DriverHandle,3000,&length,sizeof(rb),(char*) &rb);
  Offset=rb.user_data_2[0];
  in1 = rb.user_data_2[Offset];    /* 1.byte vstupu */
  in2 = rb.user_data_2[Offset+1];  /* 2.byte vstupu */
  in3 = rb.user_data_2[Offset+2];  /* 3.byte vstupu */

  /* Uzavreni komunikacniho kanalu */
  SCP_close( DriverHandle );
}
```

Komplexnější příklad

Při řešení komplexnějších komunikačních problémů obvykle nelze použít lineární strukturu programu a většinou se nevyhneme použití více vláken. Základem je vlákno zpracovávající všechny odpovědi, které FDL vrstva posílá aplikaci zpět. Toto vlákno pracuje v nekonečné smyčce, která nejprve čeká na odpověď (timeout=SCP_FOREVER) a potom podle identifikátoru tazatele je každá odpověď zpracována zvlášť. Nakonec přechází vlákno opět v čekání na odpověď.

Vlákno příjmu odpovědí

```
void ReceiverThread()
{
  unsigned short  length;
  fdl_rb          rb;

  while (1)
  {
    /* Cekani na vysledek bez time-outu */
```

```
    SCP_receive( DriverHandle, SCP_FOREVER, &length,
                 sizeof(rb), (char*) &rb);

    /* Zpracovani prijate odpovedi podle identifikatoru tazatele */
    switch( rb.rb2header.user )
    {
      case RequestorID_1:
        ...
      case RequestorID_2:
        ...
      case ...:
        ...
    }
  }
}
```

Posílání požadavků pak může probíhat současně z několika vláken. Důvodem pro přijímání odpovědí pouze v jednom vláknu je možnost rozlišit komu je odpověď adresována. Kdyby každé vlákno posílající požadavky mělo i vlastní příjem odpovědi, mohlo by dojít k jejich záměně mezi vlákny. Odpovědi jsou totiž s požadavky svázány podle použitého DriverHandle a funkce `SCP_receive` vrátí první odpověď se stejným DriverHandle. Jiným řešením potom je, otevřít si pro každé vlákno vlastní handle. Pak již k záměnám nemůže docházet a každé vlákno dostane pouze své odpovědi.

Příloha B

Ukázková aplikace ProfiMu

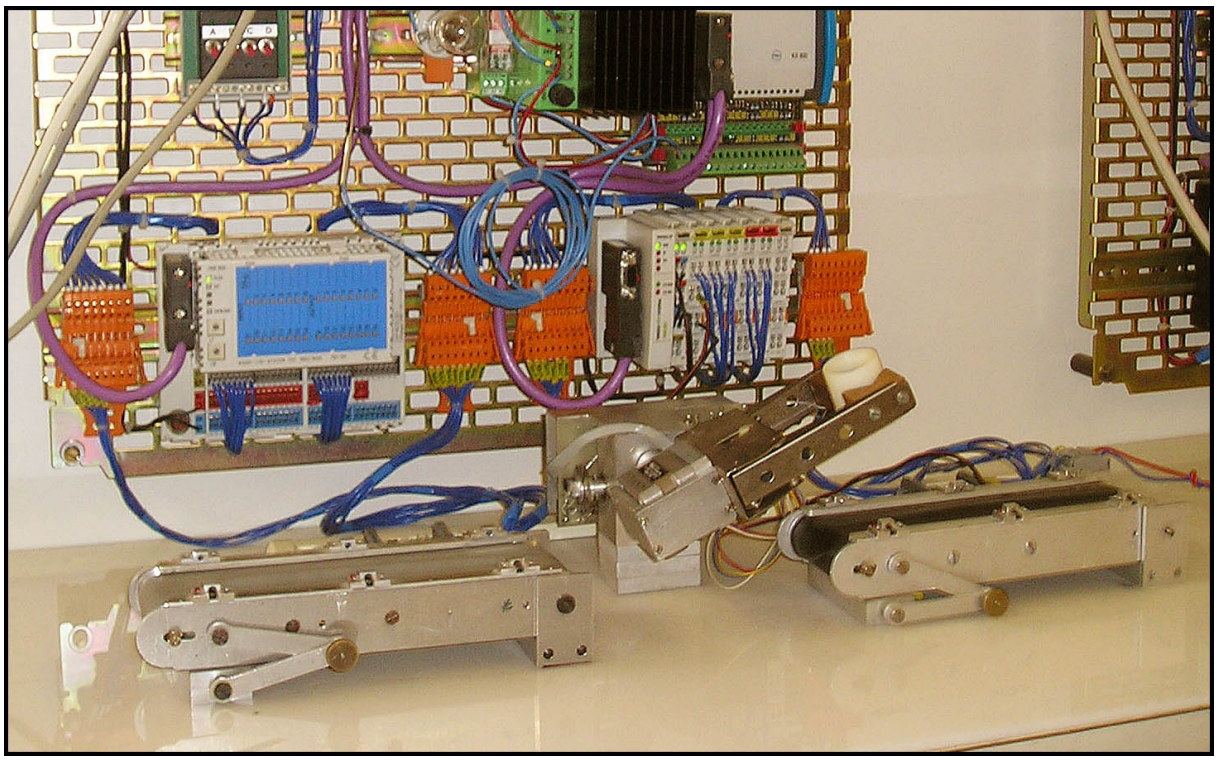
Pro předvedení funkčnosti vytvořeného Profibus DP Mastera, byla ve strojovně na Karlově náměstí vytvořena jednoduchá ukázková aplikace (obrázek B.1). Pomocí počítače s nainstalovaným ProfiMem je řízen jednoduchý model dvou dopravníků a manipulátoru s krokovými motory, který je umístěn mezi nimi. Všechny vstupy a výstupu tohoto modelu jsou připojeny na dvě jednotky vzdálených vstupů a výstupů (obrázek B.2):

- WAGO-I/O-SYSTEM 752–323 16DI/16DO
- WAGO-I/O-SYSTEM 750–301 8DI/8DO

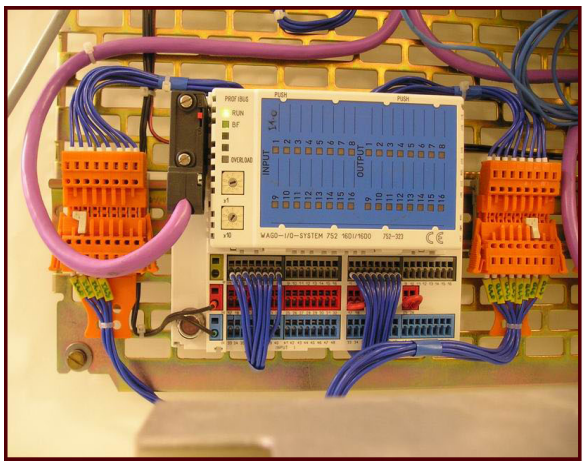
Tyto jednotky jsou jako stanice typu slave připojeny na sběrnici Profibus, která je propojuje s řídícím počítačem. Úkolem ProfiMu jako Profibus DP Mastera, je inicializovat a konfigurovat obě jednotky vzdálených vstupů a výstupů a posléze začít datovou výměnu, při které přenáší do jednotek obrazy výstupů a zároveň nazpět čte obrazy jejich vstupů. Navíc zajišťuje správu provozu na Profibusu a případné sdílení sběrnice s další řídící stanicí master. Obrazy vstupů a výstupů jsou aplikační vrstvou ProfiMu poskytovány aplikaci, která tak má vytvořeny podmínky podobné programům v PLC a zajišťuje řízení modelu.

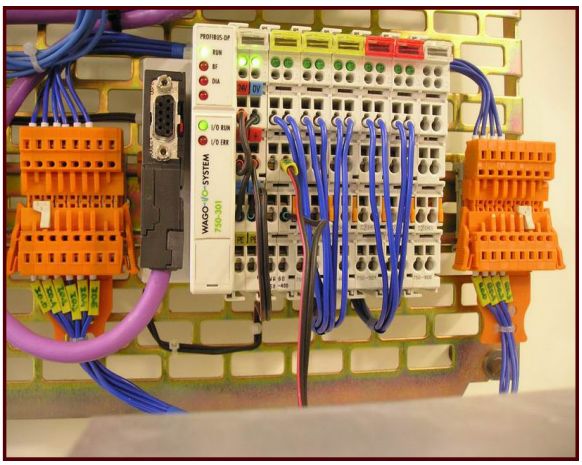
Aplikační interface ProfiMu může být současně využíván více aplikacemi. To je demonstrováno další stanicí vzdálených vstupů a výstupů Siemens ET-200B 24DI/8DO, která je připojena na stejnou sběrnici jako dvě předchozí stanice a řízena z jiné aplikace. Není však na ní připojen žádný model a tak řídící aplikace pouze v určité sekvenci nastavuje výstupy, což je možné vidět na indikačních LED diodách.

Tato aplikace je velmi jednoduchá, avšak úkolem této práce nebylo řídit nějaký složitý model. Pro demonstraci komunikačních schopností vytvořeného Profibus DP Mastera je vcelku dostatečná.



Obrázek B.1: Model dopravníků s manipulátorem





Obrázek B.2: Stanice vzdálených vstupů a výstupů připojující model na Profibus

Příloha C

Seznam zkratek

C.1 Zkratky pro Profibus

CSRD	Cyclic Send and Request Data with reply (FDL Service)
DA	Destination Address of a frame
DAE	Destination Address Extension(s) of a frame conveys DSAP and/or destination Bus ID
DDML	Direc Data Link Mapper
DSAP	Destination Service Access Point a LSAP which identifies the remote FDL User
ED	End Delimiter of a frame
FC	Frame Control (frame type) of a frame
FCS	Frame Check Sequence (checksum) of a frame used to detect corrupted frames
FDL	Fieldbus Data Link layer, OSI layer 2
FMA1/2	Fieldbus MAnagement layer 1 and 2
G	GAP update factor the number of token rounds between GAP maintance (update) cycles
GAP	Range of station addresses from This Station (TS) to its successor (NS) in the logical token ring, excluding stations above HSA
GAPL	GAP List containing the status of all stations in this station's GAP
HSA	Highest Stations Address installed (configured) on this Profibus segment
LAS	List of Active Stations
LE	field giving LEngth of frame beyond fixed part
LEr	field that repeats LEngh to increase frame integrity
LSAP	Link Service Access Point identifies one FDL User in a particular station
NS	Next Station (FDL), the station to which this Master will pass the token
OSI	Open System Interconnection
PHY	PHYsical layer, OSI layer 1

PS	Previous Station (FDL), the station which passes the token to this Master station
RSAP	Reply Service Access Point an LSAP at which Request Data may be obtained
SA	Source Address of a frame
SAE	Source Address Extension(s) of a frame conveys SSAP and/or source Bus ID
SAP	Service Access Point, the point of interaction between entities in different protocol layers
SD	Start Delimiter of a frame
SDA	Send Data with Acknowledge FDL Service)
SDN	Send Data with No acknowledge (FDL Service)
SRD	Send and Request Data with reply (FDL Service)
SSAP	Source Service Access Point, an LSAP which identifies the local FDL User which initiates a transaction
SYN	SYNchronization bits of a frame (period of IDLE) it guarantees the specified frame integrity and allow for receiver synchronization
t_{BIT}	BIT Time, FDL symbol period of the time to transmit one bit on this Profibus System

C.2 Zkratky pro DDK

DDK	Drivers Development Kit
DIRQL	Device Interrupt ReQuest Level - the IRQL at which a given device interrupts.
DPC	Deferred Procedure Call - a Kernel-defined control object type which represents a procedure that is to be called later. DPC usually finishes time consuming operations initialized by ISR.
GUID	Globally Unique IDentifier
HAL	Hardware Abstraction Layer - a Windows NT/Windows 2000 executive component that provides platform-specific support for the Kernel, I/O Manager, kernel-mode debuggers, and lowest-level device drivers.
ISR	Interrupt Service Routine - a routine whose function is to service a device when it generates an interrupt.
IRP	I/O Request Packet - is the basic I/O Manager structure used to communicate with drivers and to allow drivers to communicate with each other
IRQ	Interrupt ReQuest line - a hardware line over which a peripheral device, bus controller, other processor, or the Kernel signals a request for service to the microprocessor.
IRQL	Interrupt ReQuest Level - the hardware priority level at which a given kernel-mode routine runs, thereby masking off interrupts with equivalent and lower IRQL on the processor.
PnP	Plug and Play
WDM	Windows Driver Model
WMI	Windows Management Instrumentation

Příloha D

Obsah přiloženého CD

\src	Zdrojové kódy ProfiMu
\Ovladac	Zdrojové kódy ovladače a instalačních souboru (INF file)
\Knihovny	Zdrojové kódy knihoven aplikačního rozhraní
\Property Page	Zdrojové kódy pro Property Page - panelu pro nastavení vlastností ve Správci zařízení
\Ukazky pouziti	Jednoduché programy v C++ Builder ukazující použití aplikačního rozhraní ProfiMu
\bin	Výsledné soubory připravené k použití
\Ovladac	Přeložený ovladač spolu s dalšími soubory připravený k instalaci. Rozděleno podle operačního systému na verzi pro Windows NT 4.0 a Windows 2000/XP
\Knihovny	Statické knihovny aplikačního rozhraní ProfiMu
\Podklady	Veškerá dokumentace v elektronické podobě, kterou se převážně z Internetu podařilo získat o Profibusu, o standardu RS-485, o psaní ovladačů apod.
\HTML	Tato diplomová práce převedená do formátu HMTL pro vystavení na Internetu.
\LaTeX	Kompletní „zdrojový kód“ tohoto dokumentu v LaTeXu spolu s obrázky. Dobrá inspirace pro ty, kteří chtějí také napsat svoji diplomovou práci v LaTeXu.

Seznam obrázků

2.1 ISO/OSI model a Profibus DP 4
2.2 Formát znaku Profibusu 7
2.3 Formát rámce pověření (Token Frame) 7
2.4 Formát rámce bez dat 8
2.5 Formát rámce s daty fixní délky 8
2.6 Formát rámce s daty proměnné délky 8
2.7 Stavový automat FDL vrstvy 9

3.1 Jednoduchý interface RS–232/RS–485 17
3.2 Přerušení vysílače UARTu 18
3.3 Redukce pro připojení karet Tedia na Profibus 20

4.1 Kernel Mode 22
4.2 Zpracování IRP 26
4.3 Struktura Legacy ovladače 28
4.4 Struktura PnP ovladače 32
4.5 Property Page 36

5.1 Využití převodníku RS 232/485 42
5.2 Blokové schéma ProfiMu 44
5.3 Zatížení CPU podle přenosové rychlosti 46

B.1 Model dopravníků s manipulátorem 58
B.2 Stanice vzdálených vstupů a výstupů připojující model na Profibus 58

Seznam tabulek

3.1 Porovnání přenosových rychlostí RS–232 v PC a Profibusu (hodnoty jsou v bps) 19
3.2 Vlastnosti UART obvodů 20

4.1 Prioritní úrovně 27

5.2 Zatížení CPU podle přenosové rychlosti 45

Literatura

[1] RŮŽIČKA, Pavel. *Realizace vrstvy FDL protokolu Profibus*. Diplomová práce. Praha: České vysoké učení technické, Elektrotechnická fakulta, Katedra řídicí techniky, 2002. 73 s.

[2] SMEJKAL, Radek. *Realizace Profibus DP master*. Diplomová práce. Praha: České vysoké učení technické, Elektrotechnická fakulta, Katedra řídicí techniky, 2002. 79 s.

[3] BARTOSIŃKI, Roman. *Implementace USB interface pro počítačové periferie*. Diplomová práce. Praha: České vysoké učení technické, Elektrotechnická fakulta, Katedra řídicí techniky, 2003. 82 s.

[4] ONEY, WALTER. *Programming the Microsoft Windows Driver Model*. Second Edition. Microsoft Press, 2002. 800 p. ISBN 07-3561-803-8.

[5] *PROFIBUS Specification*. Edition 1.0. Karlsruhe: PROFIBUS International, 1997. 924 p. European Standard EN 50 170.

[6] *FDL Programming Interface*. Release 4. Karlsruhe: Siemens AG, 1995. 126 p.

[7] BURGET, Pavel. *Implementace zařízení DP Slave*. Praha: České vysoké učení technické, Elektrotechnická fakulta, Katedra řídicí techniky, 1999. 5 s.

[8] *RS–232 and RS–485 Application Note*. Octomber 1997 Revision. Ottawa: B&B Electronics Ltd., 1997. 44 p.

[9] *OX16PCI954 Data Sheet*. Revision 1.3. Oxfordshire: Oxford Semiconductor LTD., 1999. 72 p.
URL: <http://www.oxsemi.com>

[10] *PCI-COM komunikační karty pro sběrnici PCI*. Revize 04.2003. Plzeň: TEDIA spol. s.r.o., 2003. 28 s.
URL: <http://www.tedia.cz>

[11] VIRIUS, Miroslav. *Programování v C++*. Praha: České vysoké učení technické, 2001. 364 s. ISBN 80-01-01874-1.

[12] *Microsoft Developer Network - Device Drivers Developement*. Microsoft. URL: <http://msdn.microsoft.com>

[13] BOLDIŠ, Petr. *Bibliografické citace dokumentů podle ČSN ISO 690 a ČSN ISO 690-2 (01 0197): Část 1 Citace: metodika a obecná pravidla*. Verze 3.2. ©1999–2002, poslední aktualizace 3.9. 2002. URL: <http://www.boldis.cz/citace/citace1.pdf>