UNIVERZITA PARDUBICE
ÚSTAV ELEKTROTECHNIKY
A INFORMATIKY

SÉRIOVÁ KOMUNIKACE V OPERAČNÍM SYSTÉMU MICROSOFT WINDOWS

BAKALÁŘSKÁ PRÁCE

AUTOR PRÁCE: Pavlík Pavel
VEDOUCÍ PRÁCE: Ing. Hájek Martin

2007

UNIVERSITY OF PARDUBICE
INSTITUTE OF ELECTRICAL ENGINEERING
AND INFORMATICS

SERIAL COMMUNICATION IN OPERATION SYSTEM MICROSOFT WINDOWS

BACHELOR WORK

AUTHOR: Pavlík Pavel
SUPERVISOR: Ing. Hájek Martin

2007

Univerzita
Pardubice
Ústav elektrotechniky
a informatiky

Vysokoškolský ústav: Ústav elektrotechniky a informatiky
Katedra/Ústav: Ústav elektrotechniky a informatiky
Akademický rok: 2006/2007

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Pro: Pavlík Pavel

Studijní program: Informační technologie

Studijní obor: Informační technologie

Název tématu: Sériová komunikace v operačním systému Microsoft Windows

Zásady pro zpracování:

Navrhněte univerzální třídu v jazyce C++, která zapouzdří funkce pro komunikaci po sériové lince pod operačním systémem Microsoft Windows NT 4.0 a vyšším. Třída bude napsána tak, aby jí bylo možné využít i v jiných projektech nezávisle na použitém vývojovém prostředí – bude tedy založena pouze na funkcích z Win32 API. Třída by měla umožnit textový i binární způsob komunikace. Funkčnost třídy ověřte v jednoduchém testovacím programu. Osnova práce: • Programátorský model 32-bitového operačního systému Windows • Sériová komunikace v OS Windows • Vývoj třídy pro sériovou komunikaci • Vývoj ukázkového programu a testy funkčnosti třídy

Seznam odborné literatury:

- PETZOLD, Charles: *Programování ve Windows*. 5. vyd. Praha: Computer Press, 2002. ISBN 80-7226-206-8
- ECKEL, Bruce: *Myslíme v jazyku C++*. Praha: Grada Publishing, 2000. ISBN 80-247-9009-2
- ECKEL, Bruce: *Myslíme v jazyku C++ 2. díl – Knihovna zkušeného programátora*. Praha: Grada Publishing, 2005. ISBN 80-247-1015-3

Rozsah: 40 stran

Vedoucí práce: Ing. Hájek Martin

Vedoucí katedry (ústavu): prof. Ing. Pavel Bezoušek, CSc.

Datum zadání práce: 31.11. 2006

Termín odevzdání práce: 18.5. 2007

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně Univerzity Pardubice.

V Pardubicích dne 18.05.2007

Pavel Pavlík
(vlastnoruční podpis)

Děkuji vedoucímu bakalářské práce, Ing. Martinu Hájkovi, za cenné podněty, připomínky a materiály, které mi poskytl při zpracování mé bakalářské práce.

V Pardubicích dne 18. května 2007

Pavel Pavlík

ABSTRAKT

Cílem této bakalářské práce je vytvořit funkční třídu pro komunikaci po sériovém portu v prostředí 32-bitového operačního systému Windows pouze za použití programovacího jazyka C/C++ a funkcí z Windows API. Požadavky kladené zadáním byly splněny vytvořením více vláknové třídy realizující příjem a vysílání dat v samostatných vláknech na pozadí aplikace. Třída byla vytvořena jako vysoce univerzální, lze ji využít v konzolových i aplikacích s grafickým uživatelským prostředím. Funkčnost třídy byla otestována ve dvou aplikacích, přičemž tyto testy potvrdily bezproblémovou funkci třídy.

OBSAH

1. ÚVOD 10

2. PROGRAMÁTORSKÝ MODEL 32-BITOVÉHO OPERAČNÍHO SYSTÉMU WINDOWS 11

2.1 Datové typy 11

2.2 Callback funkce 12

2.3 Architektura operačního systému Windows 12
2.3.1 Handle 12
2.3.2 Okno 13
2.3.3 Zprávy 14

2.4 Architektura programů pro Windows 16
2.4.1 Konzolové aplikace 17
2.4.2 Grafické aplikace 17

2.5 Dynamické knihovny 18

2.6 Windows API 19

2.7 Multitasking a multithreading 20
2.7.1 Synchronizace vláken 20
2.7.2 Použití vláken ve Windows 21

3. POPIS ROZHRANNÍ RS-232 23

3.1 Základní technický popis komunikace 25

3.2 Parita 26

3.3 Řízení toku 26

4. SÉRIOVÁ KOMUNIKACE V OPERAČNÍM SYSTÉMU WINDOWS 27

4.1 Úvod 27
4.1.1 Nonoverlapped I/O 27
4.1.2 Overlapped I/O 28

4.2 Otevření portu 28

4.3 Konfigurace portu 29
4.3.1 Maska 29
4.3.2 Systémové buffery 30
4.3.3 Časové limity 30
4.3.4 DCB struktura 32

4.4 Čtení a zápis na port ... 33
4.4.1 Zápis ... 34
4.4.2 Čtení ... 35

4.5 Reakce na chyby ... 37

4.6 Uzavření portu ... 37

5. VÝVOJ TŘÍDY PRO SÉRIOVOU KOMUNIKACI ... 37

5.1 Návrh architektury třídy ... 37

5.2 Návrh třídy z uživatelského hlediska ... 39

5.3 Implementace metody pro otevření komunikace ... 40

5.4 Implementace bufferu pro odesílání dat ... 40

5.5 Implementace vlákna pro odesílaní dat ... 41

5.6 Implementace metod pro odesílání dat ... 41

5.7 Implementace vlákna pro detekci a příjem dat ... 41

5.8 Implementace logování dat ... 42

5.9 Implementace metody pro ukončení komunikace ... 42

5.10 Implementace ostatních metod třídy ... 42

5.11 Ošetření chyb ... 42

6. VÝVOJ UKÁZKOVÉHO PROGRAMU A TESTY FUNKČNOSTI TŘÍDY ... 43

6.1 Grafická ukázková aplikace ... 43

7. ZÁVĚR ... 47

SEZNAM OBRÁZKŮ A TABULEK

Obr. 1) Hierarchická struktura oken ... str. 14
Obr. 2) Zpracování zpráv ve Win32 ... str. 16
Obr. 3) Zprávy, vlákna a procesy ... str. 22
Obr. 4) Konektor sériového portu ... str. 24
Obr. 5) Přenášení hodnoty 11001101 ... str. 25
Obr. 6) Architektura třídy ... str. 39
Obr. 7) Ukázková GUI aplikace ... str. 45
Tab. 1) Popis signálů RS-232 ... str. 24
Tab. 2) Události portu ... str. 29
Tab. 3) Nejpodstatnější položky struktury DCB ... str. 32

1. ÚVOD

V současnosti se komunikace po sériovém portu v oblasti PC prakticky nepoužívá a byla již plně nahrazena výkonnějším rozhranním USB. Ovšem v průmyslu je tento standart a zvláště jeho modifikace (RS-422, RS-485) stále velice rozšířen a díky své „jednoduchosti" tomu bude pravděpodobně i na dále.

Třídy se zaměřením na sériovou komunikaci v 32-bitovém operačním systému Windows lze získat z internetu, ale jsou buď příliš jednoduché [5][12] (nepodporují práci na pozadí, současný příjem a vysílání větších bloků dat), nebo relativně komplexnější [13], ale s nedostatečnou šíří možností komunikace. Navíc většina tříd byla napsána s využitím knihovny MFC od firmy Microsoft, což snižuje jejich univerzalitu.

Z těchto důvodů bylo rozhodnuto vyvinout třídu zcela novou, nezávislou na jakékoli knihovně. Třída by měla mít následující charakteristiky:

- veškeré operace probíhají na pozadí aplikace
- možnost současného příjmu a vysílání dat
- podpora jak textového, tak i binárního způsobu komunikace

2. Programátorský model 32-bitového operačního systému Windows

Předtím, než je možné se věnovat programování komunikace po sériového portu v operačním systému Windows, je nutné se seznámit se základy programátorského prostředí. V této kapitole byly použity zdroje [2], [4] a [10].

2.1 Datové typy

Programy pro Windows používají kvůli kompatibilitě mezi různými vývojovými prostředími celou řadu svých názvů typů. Tyto typy jsou definovány pomocí `typedef` nebo `#define` a vycházejí ze standardních typů jazyka C. Většina z nich se nachází v souborech windef.h a winnt.h, které jsou pomocí `#include` zahrnuty do souboru windows.h. Některé nejběžnější typy jsou zde uvedeny:

- `CHAR` - ekvivalent pro C klíčové slovo char
- `BYTE` - označuje unsigned char (8-bitové číslo bez znaménka)
- `WORD` - označuje unsigned short (16-bitové číslo bez znaménka)
- `DWORD` - označuje unsigned long (32-bitové číslo bez znaménka)
- `INT` - ekvivalent pro C klíčové slovo int
- `UINT` - označuje unsigned int
- `FLOAT` - ekvivalent pro C klíčové slovo float
- `BOOL, BOOLEAN` - hodnoty `TRUE` nebo `FALSE`
- `LPSTR` - označuje char *
- `LPCSTR` - označuje const char *
- `LPTSTR` - označuje řetězec wchar_t je-li `UNICODE` definován, jinak řetězec unsigned char

Při volbě názvu proměnných se často dodržuje takzvaná „Maďarská notace“, která spočívá jednoduše řečeno v tom, že názvy proměnných začínají malým písmenem nebo písmeny, které naznačující

typ proměnné (např. dwNum – číslo typu `DWORD`, více zdroj [6]) . Často se také používá prefix *m_*, který naznačuje, že daná proměnná je členský atribut třídy.

2.2 Callback funkce

Callback funkce neboli zpětně volaná funkce je volaná přímo operačním systémem při vhodné příležitosti. Např. funkci hlavního okna aplikace neboli proceduru okna deklarovanou s použitím `CALLBACK` nebo `WINAPI` volá operační systém v případě, že potřebuje předat hlavnímu oknu nějakou informaci (např. nový popisek okna)

2.3 Architektura operačního systému Windows

Architektura OS Windows je postavena na struktuře klient-sever. V roli serveru je OS a v roli klientů jsou běžící procesy aplikací a jejich vlákna. Komunikace mezi serverem a klientem probíhá oběma směry. Server, tedy OS Windows, vykonává požadované operace, zprostředkovává distribuci zpráv mezi klienty a hlásí klientům výskyt událostí prostřednictvím zpráv. Klienti, tedy procesy, si od OS odebírají zprávy a reagují na ně podle toho, jak potřebují. Procesy v prostředí Win32 nemají přímý přístup ke vstupním a výstupním periferiím PC, proto musí veškerou manipulaci s nimi provádět přes OS nebo zasíláním zpráv. Programování v OS Windows je tedy především o zasílání, příjmu a reakci na zprávy.

2.3.1 Handle

Každý dynamický objekt, který OS Windows vytvoří má přidělený tzv. „handle“ (manipulátor), který se dá považovat za něco jako evidenční číslo. Handle je potřeba pro jakoukoli manipulaci s daným objektem a bývá většinou reprezentován číslem typu `UINT`. Číselná hodnota handle je jedinečná uvnitř skupiny stejného typu, např. v množině existujících ikon. Aby se vyloučila možnost záměny, jsou zavedeny odlišné typy pro jednotlivé handle skupiny.

Např:

- `HBITMAP` – handle bitmapy
- `HCURSOR` – handle kursoru myši
- `HFONT` – handle fontu
- `HWND` – handle okna
- `HICON` – handle ikony
- `HINSTANCE` – handle „instance“, tedy samotné aplikace

2.3.2 Okno

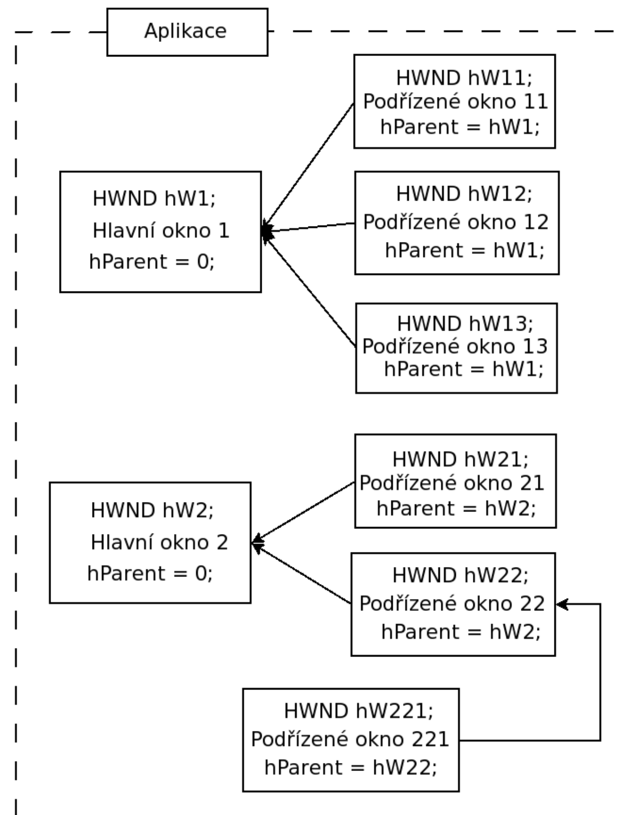
Okno je ve Windows základní grafický prvek. Oknem jsou veškeré aktivní grafické prvky např. tlačítko, scrollbar, editační pole, dialog a dokonce i některé statické texty. Každé okno při vzniku dostane handle typu `HWND` a většinou si zaregistruje funkci callback, kterou OS volá v případě, že má pro okno zprávu. Systém oken tvoří hierarchickou strukturu. Každé okno může i nemusí mít svého vlastníka nebo-li rodiče (parent) a své podřízené prvky nebo-li potomky (children). Vztah podřízenosti rodič - potomek se projeví především při zániku oken. Se zánikem rodiče se automaticky ruší všechna jeho podřízená okna. Okna lze procházet systémovou funkcí *GetWindow()*. Struktura oken v OS Windows je zobrazena na obrázku Obr. 1.

Okna lze zobrazit v různém stylu podle toho, jakou kombinaci základních prvků okna použijeme. Některé základní prvky okna jsou:

- titulek s názvem okna
- ohraničení
- rám, za nějž lze vzít myší
- systémové menu, křížek na zavření, tlačítko minimalizace atd.

Lze samozřejmě nastavit velikost okna a jeho pozici na obrazovce. Dále také okno může vlastnit prvky typu resource (zdroje) jako – hlavní menu, bitmapu, ikonu atd. V jednom okamžiku může být aktivní pouze jedno okno, což se označuje termínem *focus*. Oknu mající focus se zasílají zprávy od klávesnice a proces, který je jeho vlastníkem

se většinou dostává do popředí. To znamená, že OS mu přidělují více strojového času než procesům běžícím na pozadí. Focus lze přepínat nejen myší či klávesnicí, ale i systémovou funkcí *SetFocus()*.



Obr. 1) Hierarchická struktura oken

2.3.3 Zprávy

OS Windows a jeho aplikace generují zprávy při každé vstupní události jako pohyb kursoru myši, zmáčknutí tlačítka, změna velikosti okna atd. Aplikace mohou posílat zprávy svým oknům, nebo dokonce oknům jiné aplikace. Zpráva je vlastně struktura definovaná v hlavičkovém souboru winuser.h a má tyto atributy:

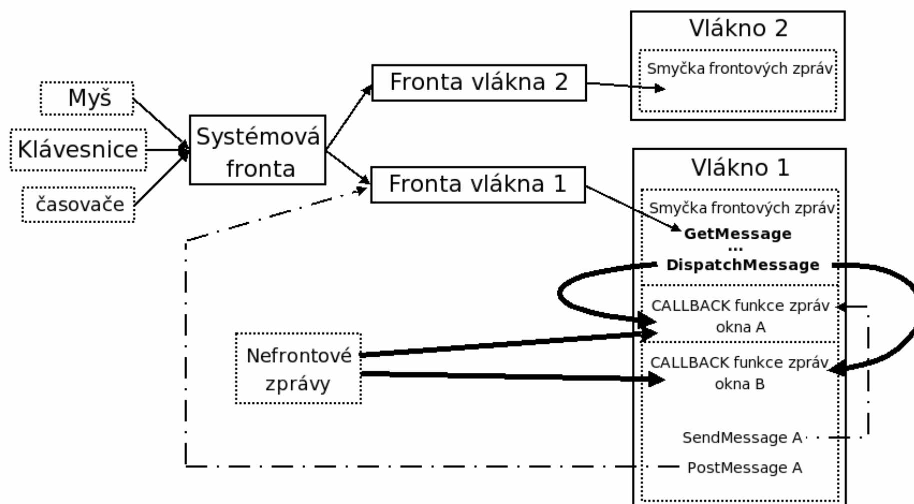
- **`HWND` hwnd** – handle okna, kterému je zpráva směrována
- **`UINT` message** – identifikátor zprávy, číslo, které zprávu označuje, tyto čísla jsou také definovány v souboru winuser.h a začínají předponou `WM` (windows message), je to např. `WM_KEYDOWN`, které se odešle po stisku klávesy

- **WPARAM wParam** – 32-bitový parametr, který záleží na konkrétní zprávě
- **LPARAM lParam** – druhý 32-bitový parametr, také záleží na konkrétní zprávě
- **DWORD time** – čas, ve který byla zpráva umístěna do fronty zpráv
- **POINT pt** – struktura, která obsahuje x a y souřadnici kursoru myši v čas, kdy byla zpráva umístěná do fronty zpráv

Windows používají pro distribuci zpráv dvě metody. První metoda spočívá v zaslání zprávy přímo funkci, která jí obsluhuje, což je vlastně CALLBACK funkce nějakého okna. Druhá metoda spočívá ve vložení zprávy do systémové fronty zpráv, ze které si zprávu vlákno aplikace odebere samo. První metoda je častější, aplikace si zaregistrují CALLBACK funkci a tu pak OS volá v případě, že má pro aplikaci nějakou zprávu.

Ovšem všechny zprávy se CALLBACK funkci neposílají, některé se vkládají do systémové fronty zpráv. Jsou to především vstupní zprávy od klávesnice a myši. Pak jsou to zprávy WM_PAINT (překreslení okna), WM_QUIT (ukončení aplikace) a WM_TIMER (hlášení o uplynutí intervalu časovače). Windows zařazují nové zprávy vždy na konec systémové fronty zpráv s výjimkou WM_PAINT. V případě výskytu této zprávy OS projde celou frontu zpráv, a pokud nalezne předchozí zprávu tohoto typu, pak obě sloučí v jednu, aby urychlil už tak časově náročné překreslení okna.

Windows typu win32 má jednu systémovou frontu a dále pak frontu zpráv pro každé vytvořené vlákno aplikace. Aplikace má vždy alespoň jedno vlákno, a proto má nejméně jednu vlastní frontu zpráv. V případě, že uživatel např. pohne myší, driver příslušného zařízení vloží do systémové fronty zprávu o této události. Až přijde zpráva na řadu Windows ji analyzují a vloží do příslušné fronty vlákna, které pak na zprávu reaguje podle své potřeby. Zpracování zpráv ve win32 je znázorněno na obrázku Obr. 2.



Obr. 2) Zpracování zpráv ve Win32

Vlákno odebírá zprávy ze své fronty v cyklu `while`, který probíhá do té doby než narazí na zprávu `WM_QUIT`. Smyčka zpracovávající frontové zprávy vypadá ve všech programech pro Windows prakticky stejně.

```
while (GetMessage(&msg, NULL, 0, 0))//načtení zprávy
{
      TranslateMessage(&msg);//předzpracovaní zprávy
      DispatchMessage(&msg);//odeslání adresátovi
}
```

Adresát zprávy neboli `CALLBACK` funkce obsahuje `switch`, ve kterém na doručené zprávy nějak reaguje. Samozřejmě funkce nemusí reagovat na všechny zprávy, a z toho důvodu se volá na konci funkce *DefWindowProc()*, která zajistí standardní zpracování zprávy.

Aplikace může posílat zprávy několika funkcemi, nejběžnější z nich jsou:

- **SendMessage()** – pošle zprávu a čeká na její zpracování
- **PostMessage()** – pošle zprávu do systémové fronty a na její zpracování nečeká

2.4 Architektura programů pro Windows

V prostředí 32-bitových Windows lze vytvářet dva základní typy aplikací. Jsou to buď aplikace konzolové, které pracují výhradně pouze s

textovou obrazovkou nebo GUI (Graphic User Interface - uživatelské grafické rozhranní) aplikace, které plnohodnotně využívají grafické prostředí.

2.4.1 Konzolové aplikace

Smyčku zpráv pro konzolové aplikace spravuje sám OS a současně pro ně i emuluje textovou obrazovku, která slouží jak pro vstup (klávesnice), tak i pro výstup. Vstupní bod aplikace je funkce *main()* a lze používat většinu funkcí z knihoven jazyka C/C++.

Konzolové aplikace sice poskytují programátorovi veškeré výhody prostředí Win32 jako jsou 4 GB virtuální paměti, podporu více vláken, dynamické knihovny, sdílení paměti atd., ovšem nedovolují programátorovi využívat grafické možnosti Windows z důvodu toho, že nemohou vytvářet žádná okna.

2.4.2 Grafické aplikace

Grafické aplikace už samozřejmě vytvářet okna a tím využívat grafické možnosti Windows mohou. Ovšem základní struktura programu, která se ve skoro stejné podobě objevuje v každém programu pro Windows, je o dost složitější. Vstupní bod aplikace je funkce *WinMain()*. Důležitým parametrem je *handle instance* aplikace, kterou Windows předají funkci WinMain() při spuštění aplikace. Handle instance se musí později uvádět jako argument pro celou řadu systémových služeb. Struktura GUI aplikace by se dala rozepsat zhruba do těchto kroků:

- **Registrace třídy okna** – nové okno je vždy založeno na nějaké jiné třídě, podle zvolené třídy pak bude vypadat i procedura okna, registrace třídy okna se provádí funkcí *RegisterClass()*
- **Vytvoření okna** – okno se vytvoří voláním funkce *CreateWindow()*, která má za parametry obecné charakteristiky okna, vytvořením Windows alokují část paměti pro uložení okna

- **Zobrazení okna** – po vytvoření okno není stále zobrazené, o zobrazení se starají metody *ShowWindow()*, která přidá okno na displej a následně *UpdateWindow()*, která okno vykreslí
- **Smyčka zpráv** – po zobrazení se už aplikace musí sama postarat o vstup z klávesnice a myši od uživatele
- **Procedura okna** – určuje skutečnou činnost programu, rozhoduje co se zobrazí na displeji i jak se bude reagovat na uživatelský vstup, procedura okna je vždy přiřazena zaregistrované třídě

Jako příloha A jsou přiloženy zdrojové kódy ukázkové aplikace, která uprostřed svého hlavního okna vykreslí nápis „Hello World".

2.5 Dynamické knihovny

OS Windows obsahuje velké množství funkcí, které mohou aplikace volat. Tyto funkce jsou uloženy v dynamických knihovnách, což jsou soubory s koncovkou dll. V dřívějších verzích systému Windows byl celý systém implementován pouze ve třech základních knihovnách. V novějších verzích se sice počet knihoven podstatně zvýšil, ale většina volání stejně směřuje do těchto tří knihoven:

- kernel32.dll – funkce jádra
- user32.dll – funkce uživatelského rozhraní
- gdi32.dll – funkce grafického rozhraní

Funkce v dll knihovnách se volají stejným způsobem např. jako funkce jazyka C/C++. Ovšem základní rozdíl je v tom, že strojový kód těchto funkcí je připojen k programu při jeho sestavování překladačem. Zatímco kódy funkcí z dll knihoven se stále nacházejí mimo přeložený program, a teprve při jeho spuštění programu se kódy funkcí nahrají do paměti, ovšem jen v případě, jestli tam již nejsou. To znamená, že spustitelný soubor obsahuje jen odkazy na různé funkce z dynamických knihoven, které používá.

2.6 Windows API

Z pohledu programátora je Windows určen právě svým programátorským rozhraním (API – Application Programming Interface). API obsahuje všechny funkce a s nimi spojené datové struktury, typy i zprávy, které může programátor využívat ve své aplikaci. Těchto funkcí je velké množství, v řádech tisíce, a jejich počet roste s každou použitou dynamickou knihovnou. Pomocí API lze implementovat vše, co prostředí OS Windows nabízí. Každá funkce API má svůj popisný název např. již zmíněné funkce *CreateWindow(), SendMessage()*. Vzhledem k velké rozsáhlosti API, ale není možné znát vše. Navíc pamatovat si různé obsáhlé datové struktury, názvy a parametry funkcí, by bylo poměrně zbytečné, vše je přehledně popsáno v dokumentaci [9]. Ovšem důležité je znát základní principy fungování Windows a několik nejpoužívanějších funkcí. Tato znalost bude všem určitě platná i v případě, že aplikace nebudou vyvíjené přímo za použití API. Windows API je podle obsahu rozděleno do následujících kategorií:

- **Administrace a Management** – toto rozhraní umožňuje instalovat, nastavovat či spravovat aplikace nebo systém
- **Grafika a Multimédia** – toto rozhraní umožňuje do aplikací vkládat formátovaný text, grafiku, audio nebo video, umožňuje např. i tisk
- **Sítě** – toto rozhraní umožňuje komunikaci počítačů v síti, sdílení, síťové tiskárny atd.
- **Bezpečnost** – toto rozhraní umožňuje autorizaci hesel, správu práv uživatelů, ochranu pro sdílené objekty systému atd.
- **Systémové služby** – toto rozhraní umožňuje aplikacím přístup ke zdrojům počítače jako jsou paměť, souborový systém, zařízení, procesy a vlákna
- **Uživatelské rozhraní –** rozhraní umožňuje vytvářet a zobrazovat okna, tlačítka, dialogy atd.

2.7 Multitasking a multithreading

Multitasking je schopnost OS mít spuštěno více programů současně. OS to řeší tak, že každému běžícímu procesu přiděluje „*časová kvanta*“, tedy určitý čas, po který má k dispozici procesor. V případě, že jsou časová kvanta dostatečně malá, uživatel má dojem, že aplikace běží současně. Ve Windows typu win32 a vyšších je navíc tzv. „*preemptivní*“ multitasking, to znamená, že OS dokáže přepínat úlohy kdykoliv to potřebuje. Výjimkou je Windows 98, který spouští 32-bitové aplikace v preemptivním módu, ale 16-bitové ne.

Multithreading je vlastně multitasking uvnitř programu. Aplikace se mohou rozdělit na více vláken, které zdánlivě pracují současně. Toto rozdělení může být v určitých programech velmi užitečné. V případě, že se v programu spustí nějaká časově náročnější úloha, je vhodné, aby běžela na pozadí, tedy v jiném vláknu, a tak mohla aplikace dále reagovat na vstupní podměty uživatele. Ovšem použití více vláken sebou přináší nové problémy, které nemusí být na první pohled vůbec patrné. Programátor se může např. domnívat, že nějaký výpočet, který zadal jinému vláknu již skončil a snaží se navázat na výsledek výpočtu ve vláknu původním. Ovšem přidělování časových kvant vláknům probíhá zcela nepředvídatelně, to znamená, že jednou již výpočet skončit mohl a jindy zase ne. Z tohoto důvodu je potřeba vlákna v určitých částech kódu synchronizovat.

2.7.1 Synchronizace vláken

Jedno z nejčastěji používaných řešení synchronizace vláken je *„kritická sekce“*. Kritická sekce je ve Windows objekt typu `CRITICAL_SECTION`, proces tento objekt nesmí žádným způsobem modifikovat mimo použití těchto API funkcí:

- **InitializeCriticalSection() –** inicializace objektu kritické sekce, musí být zavolána před prvním použitím objektu
- **EnterCriticalSection()** – vstup do kritické sekce, uvnitř sekce může být pouze jedno vlákno, v případě, že už je kritická sekce

vláknem obsazena, je běh ostatních vláken, které chtějí do sekce vstoupit pozastaven do té doby, než první vlákno sekci opustí

- **TryEnterCriticalSection()** – pokusí se vstoupit do kritické sekce, v případě, že je sekce obsazena, vlákno není zablokováno, ale je vrácena hodnota `FALSE`
- **LeaveCriticalSection() –** opuštění kritické sekce
- **DeleteCriticalSection() –** zrušení kritické sekce

Vlákno, které do kritické sekce vstoupí, ji tedy zároveň uzamkne a další vlákna do ní nemohou vstoupit do doby, než vlákno kritickou sekci opustí. Pro praktické použití z toho plyne, že prostředky sdílené mezi vlákny jsou ve zdrojovém kódu uzavřeny mezi volání API funkcí pro vstup a výstup z kritické sekce.

```
EnterCriticalSection(&m_cs);
m_buffer.clear();
LeaveCriticalSection(&m_cs);
```

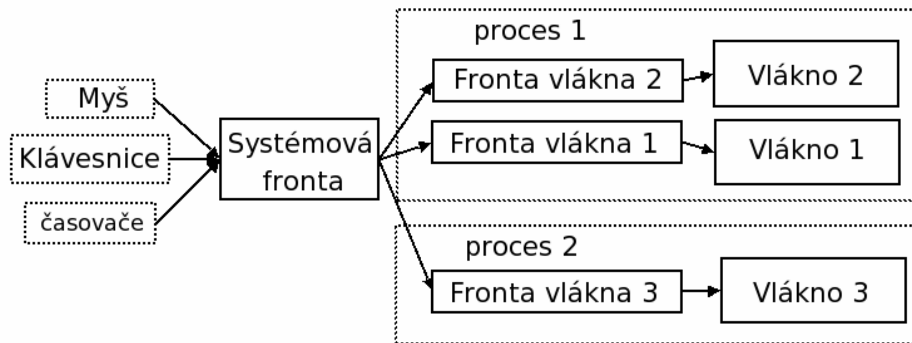
Podobné kritickým sekcím jsou i „*semafory*“, které umožní v určitém místě přerušit běh jednoho vlákna do té doby, než mu jiné vlákno povolí pokračovat. Ovšem synchronizace vláken nemusí být vždy nutná. Je vhodné si uvědomit, že např. příkaz:

```
count++;
```

Kde `count` je 32-bitové číslo se v prostředí win32 provede na jednu instrukci procesoru. Není tedy možné, aby operace vlákna byla přerušena, než bude inkrementace proměnné dokončena. Ale v případě, že by `count` byla 64-bitová hodnota, zpracování by proběhlo na dvě instrukce, a mezi nimi by mohlo být vlákno přerušeno. Více o problematice synchronizace vláken naleznete ve zdrojích [2] a [9].

2.7.2 Použití vláken ve Windows

Jak bylo zmíněno v kapitole o zprávách (2.3.3), má každé vlákno svojí frontu zpráv, a je-li potřeba reagovat na zprávy od klávesnice, myši či časovače, musí mít vlákno vlastní smyčku zpráv. Situace je znázorněna na obrázku Obr. 3.



Obr. 3) Zprávy, vlákna a procesy

Činnost vlákna určuje jeho funkce. Nejedná-li se čistě o pracovní vlákno, které má za úkol provést nějakou činnost a pak skončit, je tělo funkce vlákna tvořeno nekonečnou smyčkou. Většinou je ovšem potřeba, aby bylo vlákno aktivní pouze za určitých okolností, a tím nekonečná smyčka neprobíhala neustále. OS Windows umožňuje vlákno uspat nebo probudit. Ovšem daleko vhodnější je použití funkcí API *WaitForSingleObject(), WaitForMultipleObject()*. Když vlákno narazí na jednu z těchto funkcí, čeká, až nastane zvolená událost nebo události, které mohou být vyvolány v jiném vláknu pomocí volání funkce *SetEvent()*. Nekonečný cyklus ve funkci vlákna by mohl vypadat např. následovně:

```
BOOL bRepeat= TRUE;
while (bRepeat)
{
       dwEvent=WaitForMultipleObject(2,
                                     lpParam->m_hEvents,
                                     FALSE,
                                     INFINITE)
      {
            switch(dwEvent)
            {
                  //nastala událost konec
                  case WAIT_OBJECT_0:
                        bReapeat=FALSE;
                        break;
                  //nastala událost pracuj
                  case WAIT_OBJECT_0+1:
                        //nejaky kod
                        break;
            }
      }
}
```

Následují některé funkce pro práci s vlákny ve Windows:

- **CreateThread()** – funkce API pro vytvoření vlákna
- **SuspendThread(), ResumeThread()** – funkce API pro pozastavení respektive k pokračování činnosti vlákna
- **Sleep()** – funkce API, která uspí vlákno, ve kterém byla vyvolána na zadanou dobu
- **TerminateThread()**- funkce API, která násilně ukončí vlákno
- **CreateEvent()** – funkce API pro vytvoření události
- **SetEvent()** – funkce API, která nastaví zadanou událost
- **WaitForSingleObject()** – funkce API čeká, až nastane zvolená událost
- **WaitForMultipleObject()** – funkce API čeká, až nastane jedna nebo více ze zvolených událostí
- **_beginthread()** – jednodušší alternativa pro vytvoření vlákna z jazyka C, hlavičkový soubor process.h

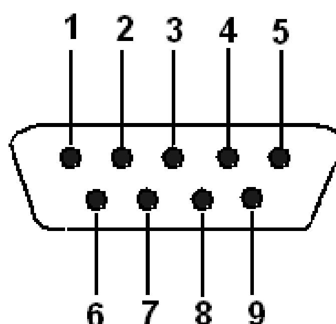
3. Popis rozhranní RS-232

Před programováním komunikace po sériového portu, je také nutné se seznámit se základními informacemi o sériové rozhraní. V této kapitole byly použity zdroje [8] a [11].

Poslední varianta tohoto standartu pochází až z roku 1969 (EIA RS-232-C) a používá se jako rozhranní pro komunikace mezi PC a další elektronikou. Sériový port umožňuje sériovou komunikaci dvou vzájemně propojených zařízení. Jednotlivé bity jsou přenášeny postupně za sebou tedy v sérii, obdobně tomu je i u síťového rozhranní Ethernetu či u novějšího sériového rozhranní USB.

V současnosti se sériový port v oblasti PC prakticky nepoužívá a byl již plně nahrazen výkonnějším rozhranním USB. Ovšem v průmyslu je tento standart a zvláště jeho modifikace (RS-422, RS-485) stále velice rozšířen a díky své „jednoduchosti“ tomu bude pravděpodobně i na dále.

Pro připojení sériovým portem se většinou používá devíti-pinový konektor, ale existuje i pětadvaceti-pinová varianta. Rozhranní bylo navrženo s ohledem pro komunikaci s modemem. Základní tři vodiče (RxD - příjem, TxD - vysílání, GND - zem) jsou doplněny dalšími, které slouží k řízení přenosu dat, a ty mohou, ale nemusí být zapojeny. Na Obr. 4 je znázorněn konektor rozhranní RS-232 a v následující tabulce Tab. 1 je připojen popis jeho signálů.



Obr. 4) Konektor sériového portu

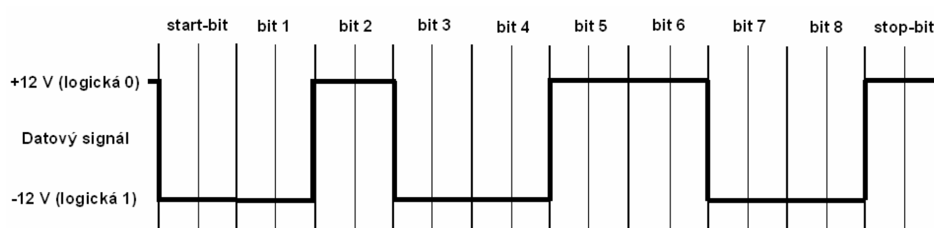
Pin	Signál	Směr	Popis
1	CD	IN	**Data Carrier Detect** Modem oznamuje terminálu, že na telefonní lince detekoval nosný kmitočet
2	RXD	IN	**Receive Data** Tok dat z modemu do terminálu
3	TXD	OUT	**Transmit Data** Tok dat z terminálu do modemu
4	DTR	OUT	**Data Terminal Ready** Terminál tímto signálem oznamuje modemu, že je připraven komunikovat
5	SGND	-	**Signal Ground** Signálová zem
6	DSR	IN	**Data Set Ready** Modem tímto signálem oznamuje terminálu, že je připraven komunikovat
7	RTS	OUT	**Request to Send** Terminál tímto signálem oznamuje modemu, že komunikační cesta je volná
8	CTS	IN	**Clear to Send** Modem tímto signálem oznamuje terminálu, že komunikační cesta je volná
9	RI	IN	**Ring Indicator** Indikátor zvonění. Modem oznamuje terminálu, že na telefonní lince detekoval signál zvonění

Tab. 1) Popis signálů RS-232

3.1 Základní technický popis komunikace

Standart RS-232 definuje asynchronní sériovou komunikaci pro přenos dat. Asynchronní znamená, že přijímač a vysílač jsou synchronizovány na začátek každé zprávy vysláním dohodnuté sekvence (zde tzv. start bitu). Data jsou přenášena za sebou od nejméně významného bitu (LSB) po nejvíce významný bit (MSB). Počet bitů přenášejících data je volitelný. Logické stavy 0 a 1 jsou realizovány pomocí dvou možných úrovní napětí. Podle použitého zařízení mohou nabývat hodnot ±5 V, ±10 V, ±12 V nebo ±15 V. Nejčastěji se používá varianta, kde logická 1 odpovídá napětí -12 V a logická 0 naopak +12 V. V případě, že se nic neděje, je linka v klidovém neboli IDLE stavu, pro který se používá nejčastěji kladné napětí.

Komunikace probíhá pomocí rámců. Každý rámec začíná start bitem (změna z log. 0 na log. 1), následují datové bity (nejčastěji 5-8), případný paritní bit (vysvětleno dále) a zvolený počet stop bitů (poskytují čas pro zpracování přijatých dat). Nejvyužívanější je varianta, kde je zvoleno celkem 10 bitů na rámec (1 start bit, 8 datových bitů, žádný paritní, 1 stop bit). Rámce mohou za sebou ihned následovat, v tom případě, jestli je použita přenosová rychlost 19200 Bd (Baud - udává počet změn stavu přenosového média za 1 s, u RS-232 je to počet přenesených b/s), můžeme za 1 s poslat maximálně 19200/10 tedy 1920 bajtů (platí pro nejvyužívanější variantu). Na obrázku Obr. 5 je po sériové lince přenášen bajt o hodnotě 11001101, velikost t záleží na zvolené rychlosti komunikace.



Obr. 5) Přenášení hodnoty 11001101

3.2 Parita

Parita je nejjednodušší způsob jak bez nároků na výpočetní techniku kontrolovat úspěšný přenos dat. Jestli je parita nastavena, vysílač sečte počet jedničkových bitů a podle dohodnuté podmínky doplní bit paritní. Je možné použit tyto typy parity:

- **Sudá (Even)** – počet jedničkových bitů + paritní bit = sudé číslo
- **Lichá (Odd)** - počet jedničkových bitů + paritní bit = liché číslo
- **Nulová (Space)** – paritní bit je vždy 0
- **Jedničková (Mark) –** paritní bit je vždy 1

Např. je-li posláno 11001101 a je nastavená sudá parita, přidá vysílač na konec paritní bit o hodnotě 1, aby byla hodnota součtu jedničkových bitů sudá. Přijímač pak kontroluje, zda je součet jedničkových bitů včetně paritního také sudý. Samozřejmě, že tento způsob kontroly není stoprocentní, může se stát, že data nebudou kompletní a parita bude v pořádku.

3.3 Řízení toku

Řízení toku (flow control, handshaking) v sériové komunikaci je mechanismus, který přerušuje komunikaci, zatímco jedno ze zařízení je zaneprázdněno nebo z nějakého důvodu nemůže komunikovat. Řízení toku může být hardwarové nebo softwarové.

Při softwarovém řízení toku se přímo s daty posílají ASCII znaky `XON/XOFF`, kterými zařízení oznamují, jestli jsou schopny komunikovat. Tento způsob řízení toku snižuje možnost přetečení dat, ale také zpomaluje samotnou komunikaci.

Pro hardwarové řízení toku jsou použity další vodiče sériového rozhraní, a tím se nesnižuje rychlost komunikace. Musí být ovšem zapojeny piny `4-DTR`, `6-DSR`, `7-RTS`, `8-CTS` (u devíti-pinového kabelu).

4. Sériová komunikace v operačním systému Windows

Informace o sériové komunikace v OS Windows vycházejí ze zdrojů [6] a [10].

4.1 Úvod

Jestliže chceme v prostředí OS Windows při programování využívat sériové rozhranní, musíme použít Windows API. K sériovému portu se ve Windows přistupuje podobně jako k souboru. Nejprve se musí port otevřít, poté se může konfigurovat nebo provádět čtení či zápis a nakonec by se měl port uzavřít. Dokonce se pro tyto vstupní/výstupní (input/output), dále jen I/O, operace používají stejné funkce jako u přístupu k souboru. I/O operace se můžou v prostředí win32 provádět dvěma způsoby, *nonoverlapped* (nepřekrývající se) nebo *overlapped* (překrývající se). Základní funkce pro práci se sériovým portem či souborem jsou:

- **CreateFile()** – otevření portu nebo souboru
- **ReadFile(), WriteFile()** – čtení a zápis na port nebo do souboru
- **CloseHadle()** – uzavření portu nebo souboru

4.1.1 Nonoverlapped I/O

Vlákno, které zavolá nonoverlapped operaci je zablokováno do té doby, než operace skončí. Z tohoto důvodu je vhodné při použití této operace vytvořit více vláken, zatímco je jedno vlákno zablokováno I/O operací, ostatní můžou pokračovat v práci. V případě, že jedno vlákno je zablokováno nonoverlapped I/O operací, ostatní vlákna, která zavolají I/O operaci v posloupnosti zjišťují, jestli původní operace již skončila. V jednu chvíli tedy probíhá pouze jedna I/O operace. Např. když jedno vlákno zavolá ReadFile() a čeká na návrat této funkce, další vlákno, které později zavolá WriteFile(), čeká, než skončí ReadFile(), a teprve poté začne provádět svojí operaci.

4.1.2 Overlapped I/O

Tato operace oproti nonoverlapped I/O nabízí větší pružnost a efektivitu. Sériový port otevřený pro overlapped I/O umožňuje více vláknům provádět více I/O operací zároveň, a navíc umožňuje provádět jinou práci, než operace skončí. Díky tomu, že tato I/O operace vlákno nezablokuje, je možné při použití v jedno-vláknové aplikaci mezi tím, než I/O operace skončí, provádět jinou činnost. V případě více-vláknové aplikace není ovšem vhodné pro každou novou I/O operaci vytvářet zvláštní vlákno, bylo by to zbytečně náročné na systémové zdroje (pamět, procesor). Daleko vhodnější je vytvořit vlákno pro každý typ operace (čtení, zápis). Díky tomu, že overlapped operace nezablokuje vlákno, se skládá ze dvou částí:

- vytvoření I/O operace
- detekce, jestli I/O operace již skončila

Volání overlapped I/O operací může i nemusí skončit ihned. V případě, že volání funkcí ReadFile(), WriteFile() skončí chybou „`ERROR_IO_PENDING`“, operace ještě není dokončena. Po dokončení operace se nastaví událost ve struktuře typu „`OVERLAPPED`“, jejíž reference se při volání funkcí ReadFile(), WriteFile() dává jako parametr (v případě overlapped I/O).

4.2 Otevření portu

O otevření portu se stará API funkce *CreateFile().* Při otevírání portu se už musí specifikovat, jestli bude používaná overlapped či nonoverlapped I/O. Následující kód ukazuje, jak správně otevřít port pro overlapped I/O operace.

```
HANDLE hComm;
hComm = CreateFile( gszPort, GENERIC_READ|GENERIC_WRITE,
                    0, 0, OPEN_EXISTING,
                    FILE_FLAG_OVERLAPPED, 0);

if (hComm == INVALID_HANDLE_VALUE)
  //chyba, port se nepodařilo otevřít
```

První parametr obsahuje jméno portu, ke kterému se chceme připojit (např. COM1). Druhý zaručí, že se bude moci na port jak zapisovat, tak z něho číst. Třetí a čtvrtý parametr, které se týkají přístupu a sdílení musí být nastaveny na 0. Pátý parametr musí být nastaven na „OPEN_EXISTING“, port musí v době připojení existovat. Šestý parametr nastavuje, jestli bude port otevřen pro overlapped operace, to je v případě hodnoty „FILE_FLAG_OVERLAPPED“, pro nonoverlapped operaci se nastavuje hodnota 0. Sedmý parametr, týkající se šablony pro vytvoření souboru, musí být při otevírání portu také 0.

Win32 API ovšem neposkytuje žádnou možnost, jak zjistit, jaké sériové porty jsou v systému k dispozici. Prakticky asi jediná možnost jak to zjistit, je daný port zkusit otevřít. Jestli se otevření nepovedlo v důsledku špatně zadaného názvu portu, nastane chyba „ERROR_FILE_NOT_FOUND“.

4.3 Konfigurace portu

Po otevření portu je možné a někdy i nutné měnit různé konfigurace, které ovlivňují průběh komunikace.

4.3.1 Maska

Nastavením masky můžeme ovlivnit, jaké události budou na portu monitorovány. Maska se nastavuje funkcí *SetCommMask(),* následuje ukázka kódu a tabulka Tab. 2, ve které jsou uvedeny možné události portu.

```
DWORD dwStoredFlags;

dwStoredFlags = EV_BREAK | EV_CTS   | EV_DSR |\ EV_ERR |
EV_RING | EV_RLSD | EV_RXCHAR |\ EV_RXFLAG | EV_TXEMPTY;

if (!SetCommMask(hComm, dwStoredFlags))
   //chyba, masku se nepodařilo nastavit
```

Událost	Popis
EV_BREAK	Na vstupu bylo detekováno přerušení.
EV_CTS	CTS (clear-to-send) signál se změnil.
EV_DSR	DSR (data-set-ready) signál se změnil.

EV_ERR	Nastala line-status chyba, může to být CE_FRAME, CE_OVERRUN, nebo CE_RXPARITY. Je důsledkem (ve stejném pořadí), špatného nastavení Bd, ztrátou znaku či znaků, nebo chybou parity.
EV_RING	Byl detekován indikátor zvonění.
EV_RLSD	RLDS (receive-line-signal-detect) signál se změnil
EV_RXCHAR	Znak byl přijat a vložen do vstupního bufferu.
EV_RXFLAG	Událost znaku byla přijata, vyvolá se po nastavení atributu struktury nastavení DCB EvtChar
EV_TXEMTY	Poslední znak z výstupního bufferu byl poslán.

Tab. 2) Události portu

4.3.2 Systémové buffery

Po otevření portu lze nastavit doporučené velikosti vstupních a výstupních bufferů. V případě, že velikosti nebudou do první I/O operace nastaveny, budou použity standardní hodnoty. Velikosti bufferu by měly být voleny podle použitého zařízení, respektive podle velikosti paketů, které zařízení používá. Ovšem velikosti jsou doporučené, a kdyby mělo dojít ke ztrátě dat v důsledku malé velikosti bufferu, OS Windows velikosti zvýší (v případě volných systémových zdrojů) tak, aby k žádné ztrátě dat nedošlo. O nastavení doporučené velikosti bufferů se stará API funkce *SetupComm()*, nejen po zavolání této funkce je vhodné buffery vyprázdnit funkcí *PurgeComm()*. Použití obou API funkcí je ukázáno v následujícím kódu.

```
//nastavení doporučené velikosti bufferu
if(!SetupComm(m_hCOM, 2048, 2048))
      //chyba, velikosti se nepodařilo nastavit

//vyprázdnění bufferu
if(!PurgeComm(m_hCOM,PURGE_TXABORT |PURGE_RXABORT |
             PURGE_TXCLEAR | PURGE_RXCLEAR))
      //chyba, buffery se nepodařilo vyprázdnit
```

4.3.3 Časové limity

Důležité nastavení, které by mělo být provedeno po otevření portu jsou timeouty (časové limity). V určitých případech může špatné nebo žádné nastavení timeotů vést až k nefunkčnosti komunikace. V případě, že nastavení timeoutů nebylo provedeno, je použito již existující

nastavení, které nemusí být vždy vyhovující. Ve Windows API jsou k dispozici pro práci s timeouty dvě funkce:

- **SetCommTimeouts()** – nastaví timeouty
- **GetCommTimeouts()** – vrátí současné nastavení timeoutů

Samotné nastavení timeoutů je zapouzdřeno do struktury s názvem „*COMMTIMEOUTS*“, struktura obsahuje tyto položky:

- **ReadIntervalTimeout** – nastavuje max. čas v ms mezi příchodem dvou znaků v jednom volání funkce ReadFile, v případě hodnoty 0 není interval používán
- **ReadTotalTimeoutMultiplier** – používá se k výpočtu celkového času čtení jednoho volání funkce ReadFile, tato hodnota se násobí počtem znaků, které má ReadFile načíst, v případě hodnoty 0 není použito
- **ReadTotalTimeoutConstant** – používá se k výpočtu celkového času čtení jednoho volání funkce ReadFile, tato hodnota se přičte k výsledku násobení hodnotou ReadTotalTimeoutMultiplier, v případě hodnoty 0 není použito
- **WriteTotalTimeoutMultiplier** – používá se k výpočtu celkového času zápisu jednoho volání funkce WriteFile, tato hodnota se násobí požadovaným počtem znaků, které se mají zapsat do výstupního bufferu, v případě hodnoty 0 není použito
- **WriteTotalTimeoutConstant** - používá se k výpočtu celkového času zápisu jednoho volání funkce WriteFile, tato hodnota se přičte k výsledku násobení hodnotou WriteTotalTimeout Multiplier, v případě hodnoty 0 není použito

Jestli budou všechny položky timeoutu nastaveny na 0, I/O operace neskončí do té doby, než dojde k načtení, respektive zápisu všech požadovaných znaků. V případě, že bude položce ReadIntervalTimeout nastavena hodnota „MAXDWORD“ a položkám ReadTotalTimeoutMultiplier, ReadTotalTimeoutConstatnt nastavena hodnota 0, funkce ReadFile() vezme ze vstupního bufferu již přijaté

znaky (v případě, že tam nějaké jsou) a ihned skončí. To může být užitečné zvlášť u nonoverlapped I/O.

Zda I/O operace neskončily timeoutem lze poznat pouze tím, kolik bylo opravdu načteno či zapsáno znaků. I/O operace jsou v případě, že nastane timeout ukončeny úspěšně (shodně jako když timeout nenastane). Následující kód ukazuje nastavení timeoutů.

```
COMMTIMEOUTS timeouts;

timeouts.ReadIntervalTimeout = 20;
timeouts.ReadTotalTimeoutMultiplier = 10;
timeouts.ReadTotalTimeoutConstant = 100;
timeouts.WriteTotalTimeoutMultiplier = 10;
timeouts.WriteTotalTimeoutConstant = 100;

if (!SetCommTimeouts(hComm, &timeouts))
   //chyba, timeouty se nepodařilo nastavit
```

4.3.4 DCB struktura

Device control block (kontrolní blok zařízení) dále jen DCB je rozsáhlá struktura, která obsahuje nastavení jako rychlost komunikace, parita, řízení toku atd. Správné nastavení DCB je proto nejdůležitějším aspektem pro funkčnost komunikace po sériovém portu. Nejdůležitější položky jsou uvedeny a popsány v následující tabulce Tab. 3.

Datový typ	Nastavení	Popis
DWORD	BaudRate	Nastavuje rychlost komunikace v Bd (u RS-232 to je počet b/s). Může nabývat hodnot: CBR_110, CBR_300, CBR_600, CBR_1200, CBR_2400, CBR_4800, CBR_9600, CBR_14400, CBR_19200, CBR_38400, CBR_57600, CBR_115200, CBR_128000, CBR_256000
BYTE	ByteSize	Nastavuje počet datových bitů, nejčastěji 8, možné až 255.
BYTE	Parity	Nastavuje paritu, možné hodnoty: EVENPARITY – sudá parita ODDPARITY – lichá parita SPACEPARITY – nulová parita MARKPARITY – jedničková parita NONEPARITY – bez paritního bitu
BYTE	StopBit	Nastavuje počet stop bitů, hodnoty: ONESTOPBIT – 1 stop bit ONE5STOPBITS – 1.5 stop bitů TWOSTOPBITS – 2 stop bitů

DWORD	fRtsControl	Nastavuje řízení datového toku RTS on TX, možné hodnoty: RTS_CONTROL_DISABLE – vypne RTS_CONTROL_ENABLE – zapne RTS_CONTROL_HANDSHAKE – zapíná/vypíná auto. podle zaplnění vstupního bufferu RTS_CONTROL_TOGGLE – zapne pouze při odesílání dat

Tab. 3) Nejpodstatnější položky struktury DCB

Nejčasnější chyby při programování komunikace po sériovém portu plynou ze špatného nastavení či inicializace DCB struktury. DCB lze korektně nastavit a inicializovat dvěma způsoby, buď za pomocí API funkce *BuildCommDCB(),* nebo *GetCommState()*. Následují ukázky kódu.

```
DCB dcb = {0};

if (!GetCommState(hComm, &dcb))
      //chyba, při získání DCB
else
      //lze použít získané DCB

DCB dcb2;
FillMemory(&dcb2, sizeof(dcb2), 0);
dcb.DCBlength = sizeof(dcb2);

if (!BuildCommDCB("9600,n,8,1", &dcb2))
{
      //chyba, nelze vytvořit DCB
}
else
      //lze použít získané DCB
```

4.4 Čtení a zápis na port

Čtení a zápis na port si jsou velmi podobné, podobnost je vidět i z totožných parametrů API funkcí, které I/O operace provádějí, tedy *ReadFile()* a *WriteFile().*

Funkce mají pět parametrů a jsou to:

- handle na otevřený port
- ukazatel na buffer, do kterého se data načítají nebo ukládají
- počet bajtů, který se má načíst nebo zapsat

- ukazatel na proměnou, do které bude uložen skutečný počet zapsaných či načtených bajtů
- ukazatel na `OVERLAPPED` strukturu, musí být `NULL` v případě otevření portu pro nonoverlapped I/O operace

Způsob implementace čtení i zápisu se samozřejmě liší podle typu použité I/O operace (overlapped/ nonoverlaped). Pro oba typy operací platí, že by bylo vhodné je volat ze zvláštního vlákna. Overlapped I/O operace sice vlákno nezablokuje, ale zase se musí detekovat, jestli již skončila. Výhoda použití overlapped I/O operací (kapitola 4.1.2) je hlavně v tom, že jich může probíhat více ve stejný okamžik.

4.4.1 Zápis

Při nonoverlapped operaci se zápis provede poměrně jednoduše. Zavolá se funkce WriteFile() s příslušnými parametry, zkontroluje se, zda proběhla v pořádku a následně se případně zkontroluje, jestli nedošlo při zápisu dat k timeoutu. Implementace funkce „WriteABuffer()“, která se pokusí zapsat požadovaná data, by mohla vypadat např. následovně:

```
BOOL WriteABuffer(LPCVOID * lpBuf, DWORD dwToWrite)
{
      DWORD dwWritten=0;
      if(!WriteFile(hComm,lpBuf,dwToWrite,&dwWritten,NULL))
            //chyba, při zápisu dat
            return FALSE;

      if (dwToWrite!=dwWritten)
            //chyba, byl překročen timeout
            return FALSE;

      return TRUE;
}
```

Naopak při použití overlapped I/O operace je situace složitější. V první řadě se nesmí stát, aby `OVERLAPPED` struktura použitá ve volání WriteFile(), byla zároveň použita v jiné I/O operaci. Dále se musí vytvořit v `OVERLAPPED` struktuře událost, která oznámí ukončení operace. Po zavolání WriteFile() s příslušnými parametry se musí zkontrolovat, jestli skončila s chybou nebo bez chyby. V případě, že skončila s chybou „`ERROR_IO_PENDING`“, operace není stále dokončena a musí se tedy čekat

na její dokončení. Nakonec by se opět mohla přidat kontrola, zda nedošlo k timeoutu. Jako příloha B je přiložen zdrojový kód, který ukazuje jak by mohla vypadat implementace funkce „WriteABuffer()“, která se pokusí zapsat požadovaná data při overlapped I/O operaci.

4.4.2 Čtení

Rozdíl mezi čtením a zápisem na port je především v tom, že než se začnou data načítat, je dobré vědět, jestli nějaká data ve vstupním bufferu jsou. Nejlepším řešení je detekovat příchozí data. K detekci příchozích dat, a i ostatních událostí na portu slouží API funkce *WaitCommEvent()*. Funkce oznámí ty události, které byly nastaveny maskou. Obdobně jako ReadFile() nebo WriteFile() funkce WaitCommEvent() při nonoverlapped I/O operaci vlákno zablokuje do té doby, než nastane nějaká událost nastavená v masce. Naopak při overlapped I/O operaci vlákno funkce WaitCommEvent() nezablokuje, ale musí se čekat na konec overlapped I/O operace, která v tomto případě oznámí, že nastala nějaká událost z nastavené masky. Následuje ukázka, jak by mohla vypadat detekce a následné načítání dat při nonoverlapped I/O operaci. Kód bude následně popsán.

```
DWORD dwCommEvent;
DWORD dwRead;
char  chRead;

if (!SetCommMask(hComm, EV_RXCHAR))
   //chyba, pří nastavení masky

for ( ; ; )
{
    if (WaitCommEvent(hComm, &dwCommEvent, NULL))
    {
      do{
            if (ReadFile(hComm, &chRead, 1, &dwRead, NULL))
                  //znak/bajt byl načten, zpracuj ho
            else
                  //nastala chyba při ReadFile
            break;
       } while (dwRead);
   }
   else
      //nastala chyba při WaitCommEvent
      break;
}
```

Po nastavení masky začíná nekonečná smyčka, ve které se nejprve čeká, až nastane událost, která oznamuje příchozí data. Následně se data po jednom bajtu v cyklu načítají. Událost oznamuje příchod jednoho bajtu, proto se na první pohled může jevit, že použitý cyklus je zbytečný. Ovšem při příchodu více bajtů rychle po sobě by bez cyklu nedošlo k načtení všech bajtů. Cyklus probíhá do té doby, než funkce ReadFile() vrátí prázdný bajt, respektive než je vstupní buffer prázdný. Při jakékoli chybě dojde k přerušení nekonečné smyčky. Tento kód bude správně fungovat jen v případě, že budou timeouty nastaveny tak, aby funkce ReadFile() skončila okamžitě s obsahem vstupního bufferu (rozebíráno u timeotů, kapitolo 4.3.3).

Při overlapped I/O operaci je situace o něco složitější, ukázkový kód je přiložen jako příloha C. Po nastavení masky a vytvoření události `OVERLAPPED` struktury, která oznámí, že nastala událost, začíná nekonečná smyčka. Na začátku smyčky je zavolána funkce WaitCommEvent() a následně je zkontrolována její návratová hodnota. V případě chyba „`ERROR_IO_PENDING`" je vše v pořádku, jinak dojde k ukončení nekonečné smyčky. Následně se bude po zavolání WaitForSingleObject() čekat, než nastane událost na portu z nastavené masky. Pokud nenastane událost do určené doby, pokračuje se dalším průchodem nekonečné smyčky, aby se zjistilo, zda nedošlo při čekání na událost na portu k chybě. Jestli událost nastane a bude to událost oznamující data, tak se pomocí funkce *ClearCommError()* a struktury `COMSTAT` zjistí počet bajtů ve vstupním bufferu, které se následně načtou funkcí ReadABuffer(). Funkce ReadABuffer() je obdoba funkce WriteABuffer() z přílohy B. Obě funkce jsou totožné, samozřejmě až na záměnu WriteFile() za ReadFile() a použití nekonstantního bufferu.

Obě varianty ukázkových kódů by bylo vhodné umístit do samostatného vlákna, které by se staralo o detekci a načítaní dat.

4.5 Reakce na chyby

Jedna z možných událostí, které mohou na sériovém portu nastat je „EV_ERR“. Tato událost upozorňuje na chybu vzniklou při komunikaci (můžou nastat i chyby, které EV_ERR nezpůsobí). Když nastane nějaká chyba, je veškerá komunikace přerušena až do zavolání funkce API *ClearCommError(),* díky které nejen zjistíme, jaká chyba nastala, ale také můžeme zjistit zaplnění vstupního a výstupního bufferu portu. Jestli funkce WaitCommEvent() vrátí událost „EV_ERR“ je tedy pro obnovení komunikace zapotřebí zavolat funkci ClearCommError().

4.6 Uzavření portu

Uzavření portu se provede jednoduše voláním systémové funkce *CloseHandel(),* před uzavřením portu by bylo vhodné odstranit nastavení masky (nastavením masky na 0), vyprázdnit buffery a případně obnovit nastavení původních timeoutů (v případě, že bylo uloženo).

5. Vývoj třídy pro sériovou komunikaci

Při vývoji třídy byly použity zdroje [1], [2], [3], [4], [6], [10] a [13].

5.1 Návrh architektury třídy

Před samotným vývojem třídy, je zapotřebí zvážit několik rozhodujících aspektů. Je to zejména typ použité I/O operace, použití vlastních bufferů a použití jednoho nebo více vláken.

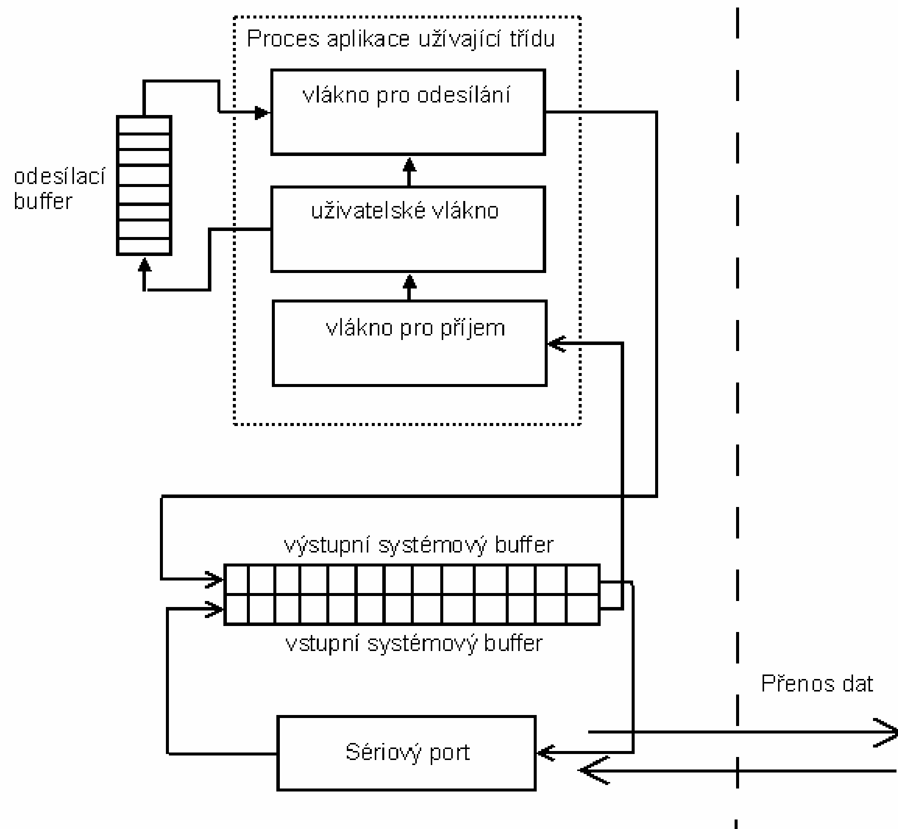
Výhody a nevýhody overlapped/nonoverlapped I/O operací jsou zřejmé. Implementace nonoverlapped operací je jednoduší, ale zablokuje vlákno a není příliš efektivní (neumožňuje více I/O současně). Naopak implementace overlapped I/O operací je složitější, ale nezablokuje vlákno a hlavně umožňuje více I/O operací současně. Vhodnější volbou je tedy použití overlapped I/O operací.

V prostředí OS typu Win32, má každý sériový port k dispozici systémový vstupní a výstupní buffer. Dokonce OS umí měnit v případě potřeby velikosti obou bufferů, aby nedošlo ke ztrátě dat. Je jasné (vzhledem k rychlosti komunikace), že vyprazdňovat, neboli odebírat přijatá data ze systémového vstupního bufferu budeme moci mnohem rychleji, než tam budou přibývat. Z tohoto důvodu by bylo zbytečné použít pro příchozí data další vyrovnávací paměť. Otázkou ovšem zůstává, jakou rychlostí jsme schopni plnit výstupní systémový buffer portu. Dá se tedy říct, že smysl má buffer pouze pro odesílaná data a pro jistotu, že zápis do systémové vyrovnávací paměti při větším objemu dat nebude zdržovat či působit nějaké problémy, je vhodné ho použít.

V případě, že chceme, aby třída v jeden okamžik byla schopna data odesílat, přijímat a ještě přitom reagovat na podměty uživatele, tak je nadmíru jasné, že to nemůže být realizováno pouze pomocí jednoho vlákna. Ani varianta se dvěma vlákny, kde by se jedno vlákno staralo o podměty uživatele a zároveň odesílalo data, zatímco druhé by detekovalo příchozí data a následně je načítalo, není úplně vyhovující. Při rychlostech sériové komunikace může odesílání i menšího objemu dat trvat několik sekund a po dobu odesílání dat by aplikace nebyla schopna reagovat na podměty uživatele (byla by zamrzlá). Nabízí se tedy varianta se třemi vlákny, kde se jedno vlákno bude starat o podměty uživatele, další o příjem dat a další o odesílaní dat. Jiná možnost by byla, vytvářet pro každou novou I/O operaci zvláštní vlákno, které by po skončení operace zaniklo, ale tato možnost je zbytečně náročná na systémové zdroje. Jako nejlepší možnost, která splňuje požadavky na zápis, příjem a reakci se tedy jeví varianta právě se třemi vlákny.

Z předchozích odstavců vyplývá že, třída bude tedy mít tři vlákna, bude použito overlapped I/O operací a bude implementována vlastní vyrovnávací pamět pro odesílaná data. Situace je znázorněna na obrázku Obr. 6. Když uživatel bude odesílat data, uživatelské vlákno je vloží do bufferu pro odesílaná data a oznámí vložení nových dat pomocí události vláknu pro odesílání. Odesílací vlákno odebere data z vyrovnávací

paměti a vloží je do systémového výstupního bufferu, ze kterého je driver zařízení přenese na sériový port. U příjmu dat je situace opačná, driver vloží data z portu do vstupního systémového bufferu, přijímací vlákno příchod dat detekuje, načte je a následně přijatá data odešle pomocí zprávy uživatelskému vláknu, které je např. zobrazí.



Obr. 6) Architektura třídy

Ovšem v návrhu třídy je také nutné myslet na použití v konzolové aplikaci, při které není možné odeslat zprávu s přijatými daty. Právě z tohoto důvodu bude mít třída další buffer, který bude plnit roli úložiště pro příchozí data a bude na uživatelském vláknu, aby data z úložiště odebíralo.

5.2 Návrh třídy z uživatelského hlediska

Třída bude obsahovat veřejné metody, které uživateli umožní:

- zjistit jaké porty jsou v systému k dispozici
- otevřít komunikaci s nastavením základních parametrů komunikace

- změnu konfigurace portu
- posílání ASCII řetězce a binárních dat
- logování komunikace
- test, jestli je port připojen
- zjistit počet a odebrat přijatá data z úložiště při konzolové aplikaci
- uzavření komunikace

Popisy metod umožňující tyto činnosti a příklady použití jsou uvedeny v uživatelské příručce, která je v příloze D.

5.3 Implementace metody pro otevření komunikace

Po samotném otevření portu metoda nastavuje i masku, doporučné velikosti bufferů (následně je i vyprázdní), nastavení timeoutů (vše na hodnotu 0), nastavení DCB podle zadaných nebo standardních parametrů. Po těchto nastaveních jsou vytvořena vlákna a potřebné události. Všechny tyto činnosti byly již popsány v předešlých částech práce. V případě, že nějaká z těchto činností nebude vykonána úspěšně, je vyhozena výjimka, která bude informovat, co přesně selhalo. Po úspěšném otevření portu, je do proměnné, která indikuje zda je port připojen, uložena hodnota `TRUE`. Celá metoda otevření komunikace (stejně jako uzavření) je uvnitř kritické sekce, což umožňuje pouze dva stavy, buď je port připojen se všemi náležitostmi (maska, timeouty, vlákna, atd…) nebo připojen není.

5.4 Implementace bufferu pro odesílání dat

Jako buffer je použita oboustranná fronta (dequeue) z STL (Standart Template Library), která je zapouzdřena do třídy, která umožní vkládat blok dat na konec fronty a odebírat blok dat ze začátku fronty. Součástí třídy bude i kritická sekce, která je zde nutná, vzhledem k tomu, že k bufferu budou přistupovat dvě vlákna (uživatelské a pro odesílání). Použití kontejneru z STL přináší výhodu oproti použití pole či

dynamicky alokovaného pole v tom, že není nutné se starat o alokaci paměti a navíc je použití pohodlnější.

5.5 Implementace vlákna pro odesílaní dat

Procedura vlákna obsahuje nekonečnou smyčku, která na svém začátku čeká pomocí *WaitForMultipleObjects()* na dvě možné události. Je to buď událost, která oznámí, že do bufferu pro odesílání byla vložena nějaká nová data, nebo událost, která oznámí, že se má vlákno ukončit. V případě, že nastane událost, která oznámí nová data pro odesílání, začne cyklus, který bude odesílat data z bufferu pro odesílání do té doby, než bude buffer prázdný nebo nedojde při odesílání dat k chybě. V případě události ukončení vlákna se přeruší nekonečná smyčka a tím vlákno skončí.

5.6 Implementace metod pro odesílání dat

Tyto metody jsou velmi jednoduché, jejich činnost spočívá pouze ve vložení odesílaných dat do bufferu a nastavení události, která odesílacímu vláknu oznámí, že v bufferu jsou nová dat. Rozdíl mezi metodou, která odesílá ACSCII řetězec a metodou odesílající binární data není prakticky žádný. Pouze více prototypů ulehčí uživateli případné přetypování.

5.7 Implementace vlákna pro detekci a příjem dat

Procedura vlákna je podobná jako kód přílohy C, který již byl popisován v části o čtení z portu (kapitola 4.4.2). Ovšem neosahuje nastavení masky a vytvoření události, tyty činnosti již byly provedeny při otevírání komunikace. Další změna je ve výměně funkce WaitForSingleObjects() za WaitForMultipleObjects() z důvodu přidání události, která oznámí, že se má vlákno ukončit. V případě, že nastane událost na portu, je zavolána metoda třídy ReportCommEvent(), která ve `switch` reaguje na událost, která na portu nastala.

5.8 Implementace logování dat

Logování dat je implentováno pomocí standardních I/O funkcí jazyka C (fopen(), fclose(), fwrite()). Při logování vstupu a výstupu jsou do souboru jsou ukládána všechny přijatá i odeslaná data (zvlášť). Logování má smysl především u konzolové aplikace, kde není možné uživatelskému vláknu poslat zprávu s přijatými daty.

5.9 Implementace metody pro ukončení komunikace

Před samotným zavoláním API funkce CloseHandle(), která provede zavření portu, jsou ukončena vlákna a jejich události. Dále je odstraněno nastavení masky a jsou vyprázdněny buffery. Po zavření portu je do proměnné, která indikuje, zda je port připojen uložena hodnota `FALSE`. Celá metoda uzavření komunikace je uvnitř kritické sekce.

5.10 Implementace ostatních metod třídy

Implementace ostatních metod třídy je buď velmi jednoduchá, nebo je složená z ukázek kódu z části o sériové komunikaci v OS Windows (kapitola 4). V konstruktoru třídy je mimo počátečního nastavení atributů také inicializována kritická sekce. V destruktoru je ukončena případně otevřená komunikace, logování a je vymazána kritická sekce.

5.11 Ošetření chyb

Při sériové komunikaci může nastat mnoho chyb, je tedy nutné nějakým způsobem na vzniklé chyby reagovat. Reakce je ve většině případů shodná, ukončí se činnost, která byla prováděna a vzniklá chyba je oznámena uživateli třídy, který se sám rozhodne, jestli chce v komunikaci i na dále pokračovat. Nejvhodnější systém ošetření chyb v jazyce C++ je systém výjimek. Oznámení chyby je tedy řešeno vyhozením výjimky. V případě, že nastane chyba, je vyhozen objekt, který zapouzdřuje číselný a textový popis vzniklé chyby.

6. Vývoj ukázkového programu a testy funkčnosti třídy

Při vývinu testovacích aplikací byly použity zdroje [2], [9] a [10].

Pro testovací účely byli vyvinuty dvě jednoduché aplikace, které ověřují funkčnost třídy. Jednodušší konzolová verze je popisovaná v uživatelské příručce třídy, která je přiložena jako příloha D. Složitější aplikace s grafickým rozhraním bude následně popsána. Obě aplikace byly odzkoušeny při spojení dvou PC a fungovali korektně ve Microsoft Windows 98 i XP.

6.1 Grafická ukázková aplikace

Vzhledem k tomu, že třída pro komunikaci byla vyvíjena pouze za pomocí Windows API, byl tento přístup zachován i při vývinu ukázkové aplikace s grafickým rozhraním. Nebyla ovšem použita plnohodnotná aplikace s hlavním oknem a vlastní smyčkou zpráv (jako je příloha A), ale jednodušší varianta aplikace založené pouze na dialogu (již registrovaná třída okna). Takto vytvořená aplikace sice nemůže obsahovat položky jako hlavní menu, status bar, atd., ale za to přináší spoustu ulehčení a pro účely jednodušší ukázkové aplikace plně postačí. Mezi ona ulehčení patří to, že se nemusí okno registrovat ani vytvářet a smyčku zpráv zpracovává sám OS. Tím se zkrátí zdrojový kód a vstupní bod aplikace funkce WinMain() bude obsahovat pouze volání API funkce *DialogBox()*.

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
                   hPrevInst, LPSTR lpCmdLine, int nShow)
{
      return DialogBox(hInstance,
                       MAKEINTRESOURCE(IDD_DIALOG1),
                       NULL,
                       (DLGPROC)DialogProc);
}
```

Další velkou výhodou je, že vzhled a dceřiné ovládací prvky jako tlačítka atd. jsou jednoduchým „klikacím“ způsobem na dialog umístěny v editoru zdrojů (resources), který je např. i součásti Microsoft Visual

Studia. Nebude tedy nutné ruční vytváření a nastavování dceřiných oken. Dialog takto vytvořený v editoru zdrojů je pak pomocí makra MAKEINTRESOURCE vložen do API funkce DialogBox() jako šablona a vytvořené dialogové okno je pak přesně takové, jaké bylo navrženo v editoru zdrojů. Na zprávy lze regovat v proceduře dialogu stejným způsobem jako u aplikace s hlavním oknem, jediný rozdíl je v návratové hodnotě. Když zprávu procedura dialogu nezpracuje, vrací místo volání funkce *DefWindowProc()* jen FALSE a v opačném případě logicky TRUE.

```
INT_PTR CALLBACK DialogProc(HWND hwndDlg, UINT uMsg, WPARAM
                            wParam, LPARAM lParam)
{
      WORD wNotifyCode;//hlaseni od prvku control
      WORD wID;//id pridelene prvku
      switch (uMsg)
      {
            case WM_INITDIALOG:
                  InitDlg(hwndDlg);
                  return TRUE;
            case WM_COMMAND:
                  //zprávy od dceřiných oken
                  WORD wNotifyCode=HIWORD(wParam);
                  WORD wID=LOWORD(wParam);
                  switch (wID)
                  {
                        //zmacknuti tlacitka Connect
                        case IDC_BTN_CONNECT:
                              //činnost
                              break;
                        //zmacknuti tlacitka ReScan
                        case IDC_BTN_RESCAN:
                              //činnost
                              break;
                  }
                  return TRUE;
      }
      return FALSE;
}
```

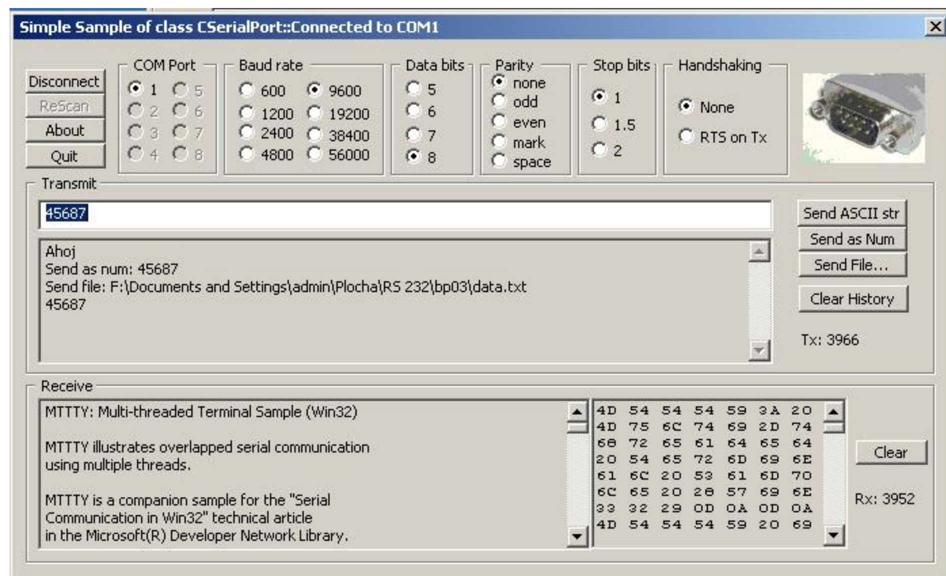
Zpráva WM_COMMAND označuje zprávu od dceřiného prvku okna. Pomocí makra LOWORD je získána identifikace dceřiného prvku a podle ní, je poté ve switch rozhoduto, jak bude na zprávu reagováno.

Další vývoj aplikace (např. zobrazení přijatých dat) probíhal bez větších problémů, ale byl poměrně zdlouhavý. Realizace většiny úkolů spočívala v dohledání vhodné API funkce či zprávy, což např. pro změnu

fontu okna nebylo vůbec snadné. Při realizaci aplikace byly využity především tyto API funkce:

- **GetDlgItem()** – funkce, která vrátí handle okna podle zadaného identifikátoru (define ze resource.h)
- **GetWindowText(), SetWindowText()** – funkce, které nastaví nebo vrátí text zadaného okna, např. popisek tlačítka
- **GetDlgItemInt()** – pokusí se přeložit text okna do `INT` nebo `UINT`
- **EnableWindow()** – vypne nebo zapne zadané okno, tedy nastavuje, jestli je okno aktivní
- **SendDlgItemMessage()** – pošle zprávu oknu diaglogu
- **CheckRadioButton()** – označí rádio button v zadané skupině buttonů
- **IsDlgButtonChecked()** – zjistí, jestli je zadané tlačítko zaškrtlé
- **MessageBox()** – zobrazí okno se zprávou

Na obrázku Obr. 7 je zobrazená vyvinutá GUI aplikace a v následujícím textu bude popsána.



Obr. 7) Ukázková GUI aplikace

Ovládání programu je naprosto intuitivní. Tlačítka v levém horním rohu umožňují připojení k portu respektive odpojení, zjištění dostupných portů (pouze je-li port odpojen), zobrazení informací o

autorovi a ukončení aplikace. Nápis okna (caption) se mění podle toho, jestli je port připojen nebo není. Napravo, vedle tlačítek pro připojení atd. se nachází přepínače pro nastavení konfigurace portu. Aby byla konfigurace zaktivována, musí být nastavena před připojením, tedy před zmáčknutí tlačítka „Connect“.

V části označené „Transmit“ (přenášet) se nachází jeden box pro vstup dat a další, který zobrazuje historii odeslaných dat. Napravo od boxů jsou tlačítka na odeslaní řetězce, čísla nebo souboru a smazání historie. Bude-li v boxu pro vstup např. „125“ a bude stisknuto tlačítko „Send as Num“, bude odeslán jeden bajt, naopak bude-li stisknuto tlačítko „Send ASCII str“ budou poslány tři bajty. V případě, že bude zadané číslo větší než 255 bude posláno čtyřmi bajty (při zmáčknutí „Send as Num). Tlačítka „Send File…“ vyvolá standardní dialog pro výběr souboru, po zvolení souboru se soubor odešle. Pod tlačítky je zobrazen počet odeslaných bajtů.

V části označené „Receive“ (příjem) jsou také dva edit boxy, jeden zobrazuje přijatá data v ASCII a druhý je zobrazuje hexadecimálně. Vedle obou boxů se nachází tlačítko pro smazání příchozích dat a nápis s počtem přijatých bajtů.

7. ZÁVĚR

Cílem práce bylo naprogramovat třídu pro komunikaci po sériovém portu ve 32-bitovém operačním systému Windows pouze za použití C/C++ a Windows API. Důvodem byl požadavek na univerzalitu a také na možnost zápisu i příjmu dat v jeden okamžik.

Vyvinutá třída splňuje všechny požadavky, které na ni byly kladeny. Třída byla naprogramována s využitím vláken – jedno samostatné vlákno je určeno pro příjem, druhé pro vysílání dat. Vlákna umožňují provádět veškeré činnosti spojené s komunikací na pozadí aplikace. Výměna informací mezi jednotlivými vlákny probíhá pomocí zpráv a událostí.

Třídu lze využít v konzolových aplikacích, ale doporučuji třídu využít v programech s grafickým uživatelským rozhraním, kde je možné naplno využít možnosti, které OS Windows nabízí.

Funkčnost třídy byla ověřena jejím použitím ve dvou testovacích aplikacích. První aplikace je pouze konzolová, druhá pak s grafickým uživatelským rozhraním. Obě aplikace během testů potvrdily bezproblémovou funkčnost třídy, a to v operačních systémech Microsoft Windows 98 a XP.

Lze se domnívat, že navrženou třídu bude možné bez problémů používat v praxi, navíc i v řídících aplikacích, neboli programech s vysokým požadavkem na efektivitu jejich práce.

Zdrojové kódy vyvinuté třídy i ukázkových aplikací jsou včetně uživatelské příručky přiloženy na CD.

SEZNAM POUŽITÉ LITERATURY

1. ECKEL, Bruce: *Myslíme v jazyku C++*. Praha: Grada Publishing, 2000. ISBN 80-247-9009-2

2. PETZOLD, Charles: *Programování ve Windows*. 5. vyd. Praha: Computer Press, 2002. ISBN 80-7226-206-8

3. PRATA, Stephan: *Mistrovství v C++*. 2. vyd. Brno: Computer Press, 2004. ISBN 80-251-0098-7

4. ŠUSTA, Richard: *Programování pro řízení ve Windows*. 1. vyd. Praha: ČVUT, 2003. ISBN 80-01-01922-5

5. ARCHER, Tom, LEINECKER, Rick . *CSerial - A C++ Class for Serial Communications* [online]. 1999-08-07 [cit. 2007-05-17]. Dostupný z WWW: <http://www.codeguru.com/cpp/i-n/network/serialcommunications/article.php/c2503/>.

6. DENVER, Allen. *Serial Communications in Win32* [online]. 1995-12-11 [cit. 2007-05-14]. Dostupný z WWW: <http://msdn2.microsoft.com/en-us/library/ms810467.aspx>.

7. *Hungarian notation*. Wikipedia : Otevřená encyklopedie [online]. 2007 [cit. 2007-05-13]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Hungarian_notation>

8. *HW server. HW server představuje - RS-232 : Soubor všech potřebných informací, návody, tipy a triky, katalogové listy aj*. [online]. c2003 [cit. 2007-05-13]. Dostupný z WWW: <http://www.rs232.hw.cz/>

9. CHALUPA, Radek. *Učíme se Win API* . Builder [online]. 2002 [cit. 2007-05-13], s. 1-28. Dostupný z WWW: <http://www.builder.cz/art/cpp/winapi1.html>.

10. Microsoft Corporation. *Win32 and COM Development* [online]. c2007 [cit. 2007-05-13]. Dostupný z WWW: <http://msdn2.microsoft.com/en-us/library/aa139672.aspx>.

11. *RS-232*. Wikipedia : Otevřená encyklopedie [online]. 2007 [cit. 2007-05-13]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/RS-232>

12. SHIBU, K.V..*Implementing Serial Communication in Win9X/2000* [online]. 2002-10-01 [cit. 2007-05-17]. Dostupný z WWW: <http://www.codeguru.com/cpp/i-n/network/serialcommunications/article.php/c5395/>.

13. SPEKREIJSE, Remon . *A communication class for serial port* [online]. 2000-02-08 [cit. 2007-05-17]. Dostupný z WWW: <http://www.codeguru.com/cpp/i-n/network/serialcommunications/article.php/c2483/>.

Příloha A – Hello World

```
/*---------------------------------------------------
  Zobrazí uprostřed hlavního okna nápis Hello World
  -------------------------------------------------*/
#include <windows.h>

//definice tridy okna
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE
                   hPrevInstance, PSTR szCmdLine,
                   int iCmdShow)
{
     //okno bude založené na třídě WNDCLASS
     static TCHAR szAppName[] = TEXT ("HelloWorld");

     WNDCLASS wndclass;
     wndclass.style = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc = WndProc ;
     wndclass.cbClsExtra = 0 ;
     wndclass.cbWndExtra = 0 ;
     wndclass.hInstance = hInstance ;
     wndclass.hIcon=LoadIcon (NULL, IDI_APPLICATION);
     wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
     wndclass.hbrBackground =
          (HBRUSH) GetStockObject (WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     //registrace třídy okna
     if (!RegisterClass (&wndclass))
     {
          MessageBox (NULL, TEXT ("Program potřebuje
                          Window NT nebo vyšší."),
                    szAppName, MB_ICONERROR) ;
          return 0 ;
     }

     //vytvoření okna
     HWND hwnd;
     hwnd = CreateWindow (szAppName, //třída okna
          TEXT ("Program Hello World"), //caption
          WS_OVERLAPPEDWINDOW, //styl okna
          CW_USEDEFAULT, //počáteční pozice x
          CW_USEDEFAULT, //počáteční pozice y
          CW_USEDEFAULT, //šířka okna
          CW_USEDEFAULT, //výška okna
          NULL, //handle na rodiče
          NULL, //handle na menu
          hInstance, //handle instance programu
          NULL); // vytvářecí parametr
```

```
	//zobrazení okna
	ShowWindow (hwnd, iCmdShow) ;
	UpdateWindow (hwnd) ;

	//smyčka zpráv
	MSG msg;
	while (GetMessage (&msg, NULL, 0, 0))
	{
		TranslateMessage (&msg) ;
		DispatchMessage (&msg) ;
	}
	return msg.wParam ;
}

//procedura okna, zde se určuje činnost programu
LRESULT CALLBACK WndProc(HWND hwnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
	HDC         hdc ;
	PAINTSTRUCT ps ;
	RECT        rect ;

	//reakce na zprávy
	switch (message)
	{
		case WM_CREATE:
			return 0;
		case WM_PAINT:
			//vykreslení hello world
			hdc = BeginPaint (hwnd, &ps) ;
			GetClientRect (hwnd, &rect) ;
			DrawText (hdc, TEXT ("Hello World"),
			        -1, &rect, DT_SINGLELINE |
			        DT_CENTER | DT_VCENTER);
			EndPaint (hwnd, &ps) ;
			return 0;
		case WM_DESTROY:
			PostQuitMessage (0) ;
			return 0 ;
		}
	return DefWindowProc (hwnd, message, wParam,
	                      lParam) ;
}
```

Příloha B – WriteABuffer s overlapped I/O

```
BOOL WriteABuffer(LPCVOID lpBuf, DWORD dwToWrite)
{
      OVERLAPPED osWrite = {0};
      DWORD dwWritten = 0;
      BOOL fRes=FALSE;

      //vytvoření události OVERLAPPED struktury
      osWrite.hEvent = CreateEvent(NULL, TRUE, FALSE,
                                   NULL);
      if (osWrite.hEvent == NULL)
            //Chyba pri vytvareni udalosti, konec
            return FALSE;


      //Pokus o zapsání
      if (!WriteFile(m_hCOM, lpBuf, dwToWrite,
                        &dwWritten, &osWrite))
      {
            if (GetLastError() != ERROR_IO_PENDING)
                  //jiná chyba než zpoždění
                  fRes = FALSE;
            else
            {
                  //Pokracuje se v zapisu, zapis je
                  //zpožděn
                  if (!GetOverlappedResult(m_hCOM,
                                          &osWrite,
                                          &dwWritten,
                                          TRUE))
                        //Jiná chyba než zpoždění
                        fRes = FALSE;
                  //zápis skončil v pořádku
                  else fRes = TRUE;
            }
       //zápis skončil v pořádku bez zpoždění
       }else fRes = TRUE;

      //Uzavreni handlu udalosti
      CloseHandle(osWrite.hEvent);

      if (dwToWrite!=dwWritten)
            //chyba, byl překročen timeout
            return FALSE;

   return fRes;
}
```

Příloha C – Příjem dat s overlapped I/O

```
OVERLAPPED osStat = {0};
BOOL bRepeat=true;
BOOL bRes;
DWORD dwCommEvent, dwEvent;
const int CHECK_TIMEOUT= 500;

dwStoredFlags = EV_BREAK | EV_CTS | EV_DSR | EV_ERR|\
EV_RING | EV_RLSD | EV_RXCHAR |EV_RXFLAG |EV_TXEMPTY;

if (!SetCommMask(hComm, dwStoredFlags))
	//chyba, masku se nepodařilo nastavit, konec

osStat.hEvent= CreateEvent(NULL, TRUE, FALSE, NULL);
if (osWrite.hEvent == NULL)
	//chyba při vytváření události, konec

while(TRUE)
{
	bRes=WaitCommEvent(hComm,&dwCommEvent,&osStat);
	if (!bRes)//jestli nastala chyba
		if(GetLastError()!=ERROR_IO_PENDING)
			//nastala závažná chyba
			break;

	//čekání na událost po dobu CHECK_TIMEOUT ms
	dwEvent=WaitForSingleObjects(osStat.hEvent,
				     CHECK_TIMEOUT);
	if (dwEvent==WAIT_OBJECT_0)//nastala událost
	{
		if (dwCommEvent==EV_RXCHAR)//nová data
		{
			COMSTAT comStat;
			DWORD dwError;
			BYTE* lpbuffer;
			//zjištění kolik bajtů je ve vstupním
			//bufferu
			if(!ClearCommError(m_hCOM, &dwError,
					     &comStat))
				//chyba pri ClearCommError
				break;
			lpbuffer=new BYTE [comStat.cbInQue];
			if(!ReadABuffer(lpbuffer,
					comStat.cbInQue)
				//chyba při načítání dat
				break;
			//zpracování dat
			delete [] lpbuffer;
		}else //zpracuj událost
	}
}
```

Příloha D – Uživatelská příručka třídy

Úvod

Třída `CSerialPort`, jejíž použití je popisované v této příručce, umožňuje komunikaci po sériovém portu, respektive po rozhraní RS-232, v prostředí win32 (Microsoft Windows 98/Me/Nt/2000/Xp). Třída je kvůli kompatibilitě založena pouze na win32 API a standardních funkcích z knihoven jazyka C/C++. Třída má implementována zvláštní vlákna pro čtení i zápis, aby tyto operace nezdržovaly hlavní vlákno aplikace. Ve třídě jsou zahrnuty tyto soubory:

- rs232.h – definice vlastní třídy `CSerialPort`
- rs232.cpp – implementace metod třídy `CSerialPort`
- CSerialBuffer.h – definice třídy `CSerialPort` použité jako buffer pro odesílání dat a úložiště příchozí dat (při konzolové aplikaci)
- CSerialBuffer.cpp – implementace metod třídy `CSerialPort`
- CSerialException.h – definice třídy výjimky používané ve třídě
- CserialDefines.h – obsahuje definy použité ve třídě `CSerialPort` (zprávy a nastavení systémových bufferů)
- debug.h – definice makra TRACE (alternativa MFC TRACE pro API) pro případné ladění
- debug.cpp – implementace TRACE
- main.cpp – obsahuje ukázkovou aplikace

Použití třídy by se dalo shrnout do těchto kroků:

- případné zjištění dostupných portů – `ScanPort()`
- otevření portu, případné nastavení logování – `OpenCom()`, `StartLogInput()`, `StartLogOutput()`
- případná změna nastavení – `GetPortSettings()`, `UpdatePortSettings()`
- posílaní a příjem dat – funkce `Send()`, příjem dat je u GUI aplikace řešen zprávou a u konzolové funkcemi `ReceivedBytes()`, `GetReceivedBytes()`

- uzavření portu – `CloseCom()`

Nastavení lze samozřejmě měnit i po otevření portu, logování lze ukončit metodami `StopLogInput()` a `StopLogOutput()`. Je také možné otestovat, jestli je port připojen metodou `IsConnected()`. Více detailů je možné nalézt v dalších částech příručky.

Skenování portů

V případě, že je nutné zjistit, jaké porty COM jsou v PC k dispozici, respektive k jakým portům se lze připojit, je možné využít metodu `ScanPort()`.

```
ULONGLONG ScanPort()const throw(serialException);
```

Tato metoda zkusí otevřít porty `COM1` až `COM8` a výsledek uloží do 64-bitového čísla, které následně vrátí. V prvním bajtu (nejméně významný bajt) čísla je uloženo, jestli je možné se připojit ke `COM1`, ve druhém ke `COM2`, ve třetím ke `COM3` atd. Jestliže je možné se připojit, pak je nastavena hodnota 255 (0xFF), v opačném případě je nastavena hodnota 0 (0x00). Takže např., je-li možné se připojit k portu `COM1` a `COM3` metoda vrátí 0xFF00FF. Metoda nesmí být volána, je-li již port připojen. V tomto případě vrátí výjimku `CSerialException`. Následuje ukázkový kód.

```
CSerialPort SP;
unsigned long long ullPorts;
unsigned char abyPorts[8]={0};

try{
      ullPorts= SP.ScanPort();
}catch (CSerialException& SE)
{
      cout << SE.GetErrorCode() << ": "
            << SE.GetErrorText() << endl;
      system("Pause");
      return 0;
}
cout << hex << ullPorts << endl;
memcpy(abyPorts,&ullPorts,8);
for(int i=0; i<8; i++)
      if (abyPorts[i]==255)
      cout << "COM" << i+1 << " je mozne se pripojit."
            << endl;
else cout << "COM" << i+1 << " neni mozne se pripojit."
            << endl;
```

Výstup:

```
ff00ff
COM1 je mozne se pripojit.
COM2 neni mozne se pripojit.
COM3 je mozne se pripojit.
COM4 neni mozne se pripojit.
COM5 neni mozne se pripojit.
COM6 neni mozne se pripojit.
COM7 neni mozne se pripojit.
COM8 neni mozne se pripojit.
Pokračujte stisknutím libovolné klávesy...
```

Test připojení

K ověření, zda je port připojen, slouží metoda `IsConnected()`.

```
BOOL IsConnected()const;
```

Otevření a zavření portu

O otevření portu se stará metoda `OpenCom()`. V této metodě může nastat výjimka.

```
void OpenCom(const HWND Parent=NULL,
const unsigned int ComNumber=1,
const unsigned int BaudRate=CBR_9600,
const unsigned char ByteSize=8,
const unsigned char Parity=NOPARITY,
const unsigned char StopBits=ONESTOPBIT
)throw (CserialException);
```

Jak je vidět z prototypu metody, může být volána i bez parametrů. V tom případě se metoda pokusí otevřít port `COM1` se standardním nastavením. Následuje popis jednotlivých parametrů.

Parent – handel na rodiče (zadejte `NULL` v případě použití v konzoli)

ComNumber – číslo portu, ke kterému se bude připojovat

BaudRate – rychlost komunikace může mít následující hodnoty

```
CBR_110
CBR_300
CBR_600
CBR_1200
CBR_2400
CBR_4800
CBR_9600
CBR_14400
CBR_19200
CBR_38400
CBR_56000
CBR_57600
CBR_115200
```

```
CBR_128000
CBR_256000
```

ByteSize – počet datových bitů

Parity – nastavení parity, která může mít následující hodnoty

`EVENPARITY` - sudá parita (počet 1 bitů + paritní bit = sudé číslo)
`MARKPARITY` – paritní bit je vždy 1
`NOPARITY` – bez parity
`ODDPARITY` – lichá parita (počet 1 bitů + paritní bit = liché číslo)
`SPACEPARITY` – paritní bit je vždy 0

StopBits – počet stop bitů, může mít následující hodnoty

```
ONESTOPBIT
ONE5STOPBITS
TWOSTOPBITS
```

Pro ukončení komunikace neboli k uzavření portu slouží metoda `CloseCom()`. Může u ní nastat výjimka. Není-li komunikace ukončena před koncem programu, je ukončena destruktorem třídy.

```
void CloseCom()throw (CSerialException);
```

Ukázkový kód otevření portu:

```
CSerialPort SP;

try{
	SP.OpenCom();
}
catch (CSerialException& SerialException)
{
	cout << SerialException.GetErrorCode() << ": "
	     << SerialException.GetErrorText() << endl;
	system("Pause");
	return 0;
}
```

Posílání a příjem dat

Pro odesílání dat je ve třídě implementována přetížená metoda `Send()`. Je možné odesílat klasický ASCII C řetězec ukončený nulou, jeden bajt nebo pole bajtů. Aby byla data odeslána, musí být port připojen.

```
void Send(const char* lpzStr);
void Send(const unsigned char byNum);
void Send(const unsigned char* byNums,
          const unsigned int uiCount);
```

Ukázkový kód

```
CSerialPort SP;

try{
      SP.OpenCom();
}catch (CSerialException& SerialException)
{
      cout << SerialException.GetErrorCode() << ": "
            << SerialException.GetErrorText() << endl;
      system("Pause");
      return 0;
}
std::string str("Posilam retezec tridou CSerialPort.");
SP.Send(str.c_str());
```

Jestli jsou přijata nějaká nová data, je rodiči poslána zpráva `WM_COMM_RXCHAR`, parametr zprávy `WPARAM` obsahuje ukazatel na přijatá data a parametr `LPARAM` obsahuje jejich délku neboli počet přijatých bajtů.

Ukázkový kód (vyjmuto z procedury okna), zobrazí `MessageBox()` s přijatými daty.

```
case WM_COMM_RXCHAR:
      INT count=static_cast<int>(lParam);
      CHAR *lpzBuffer=new CHAR [count+1];
      FillMemory(lpzBuffer, count+1, 0);
      memcpy(lpzBuffer, reinterpret_cast<char*>(wParam),
             count);
      MessageBox(hwndDlg,lpzBuffer,"New
                  data",MB_ICONINFORMATION|MB_OK);
      delete [] lpzBuffer;
      break;
```

V případě, že není použita okenní aplikaci, a je tedy po otevření komunikace nastaven rodič na `NULL`, není možné odeslat zprávu s daty. V tomto případě se musí použít jiný přístup. Není-li možné odeslat zprávu s daty jsou data ve třídě ukládána. Pro zjištění počtu bajtů, které už čekají ve třídě na vyzvednutí slouží metoda `ReceivedBytes()`.

```
ULONG ReceivedBytes()const
```

Pro vyzvednutí určitého počtu přijatých bajtů slouží metoda `GetReceivedBytes()`.

```
ULONG GetReceivedBytes(const ULONG ulCount,unsigned
                        char*lpBuf);
```

Následující kód ukazuje jakým způsobem zjistit, jestli jsou ve třídě uložena nějaká nová přijatá data a poté, jak je získat.

```
ULONG ulCount=SP.ReceivedBytes();
if (ulCount>0)
{
	lpzBuf=new char[ulCount+1];
	FillMemory(lpzBuf, ulCount+1, 0);
	SP.GetReceivedBytes(ulCount,
		reinterpret_cast<unsigned char*>(lpzBuf));
	cout << "Nova data: " << lpzBuf << endl;
	delete [] lpzBuf;
}else cout << "Zadna nova data." << endl;
```

Získaná data jsou ze třídy smazána. Tento přístup lze použít pouze u konzolové aplikace, v případě, že lze odeslat zprávu s daty, nejsou přijatá data ukládána.

Další možností, jak získávat data nejen při konzolovém využití třídy, je logování.

```
void StartLogInput(const char* lpzFileName, bool
                       bBinaryFile=false)throw
                       (CserialException);

void StopLogInput();
```

Od okamžiku, kdy je zavolána metoda `StartLogInput()`, do okamžiku zavolání `StopLogInput()` nebo ukončení komunikace, jsou veškerá přijatá data logována do zadaného souboru. Data jsou čitelná až po ukončení logování. Jestli bude soubor binární nebo textový je možné ovlivnit parametrem `bBinaryFile`. Metoda může vrátit výjimku třídy `CSerialException`. V následujícím ukázkovém kódu jsou veškerá přijatá data logována do textového souboru „Prijato.txt“.

```
CSerialPort SP;
...//nejaky kod
try{
	SP.StartLogInput ("Prijato.txt");
}catch (CSerialException& SerialException)
{
	cout << SerialException.GetErrorCode() << ": "
		<<SerialException.GetErrorText() << endl;
	system("Pause");
	return 0;
}
...//nejaky kod
```

Ekvivalentem jsou metody pro logování odeslaných dat.

```
void StartLogOutput(const char* lpzFileName, bool
                    bBinaryFile=false)
                    throw (CserialException);

void StopLogOutput();
```

Získání a změna aktuálního nastavení portu

V případě, že je nutné zjistit aktuální nastavení portu, je ve třídě implementována metoda `GetPortSettings()`, která vrací kopii struktury `DCB` s aktuálním nastavením portu. Pro změnu nastavení portu je implementována metoda `UpdatePortSettings()`, která jako parametr přijímá `DCB` strukturu, se kterou se pokusí aktuální nastavení portu změnit. Pro obsáhlost `DCB` struktury je vhodné obě metody zkombinovat. Tedy nejprve nastavení získat, poté upravit a na konec nastavení aktualizovat. Pro použití obou metod musí být port připojen, a může v nich nastat výjimka `CSerialException`. Ovšem při změnách konfigurace portu, je třeba pamatovat na to, že nastavení na obou stranách komunikace musí být shodné. Vzhledem k obsáhlosti `DCB` struktury zde její obsah není uveden. Přehlednou dokumentaci lze nalézt v MSDN Library.

```
void UpdatePortSettings(const DCB& dcb)
                        throw(CSerialException);
DCB GetPortSettings()const throw(CSerialException);
```

Ukázkový kód:

```
CSerialPort SP;
DCB myDcb;
try{
      SP.OpenCom();
      //zmena a uprava nastaveni
      myDcb=SP.GetPortSettings();
      myDcb.BaudRate=RTS_CONTROL_TOGGLE;
      SP.UpdatePortSettings(myDcb);
}
catch (CSerialException& SerialException)
{
      cout << SerialException.GetErrorCode() << ": "
            << SerialException.GetErrorText() << endl;
      system("Pause");
      return 0;
}
```

Zprávy zasílané rodiči a nastavení bufferu

V souboru `CserialDefines.h`, který je také zahrnut ve třídě `CserialPort` se nacházejí dvě skupiny maker. První se týká nastavení bufferů a obsahuje tyto makra:

`MAX_WRITE_BUFFER` – nastavení doporučené velikosti výstupního bufferu portu v bajtech, standardně je nastaveno 1024 bajtů

`MAX_READ_BUFFER` – nastavení doporučené velikosti vstupního bufferu portu v bajtech, standardně je nastaveno na 2048 bajtů

Obě tyto hodnoty by měly být voleny podle použitého zařízení, množství najednou posílaných dat a nastavené rychlosti komunikace. Ovšem tyto hodnoty jsou doporučené, OS Windows je rozhodně příliš striktně nedodržuje, a v případě potřeby velikosti bufferů změní. Při testech bylo zjištěno, že při nastavené extrémně malé velikosti bufferu pro příjem (`MAX_READ_BUFFER=2`), rychlosti `BaudRate=9600` a přijímaném souboru o velikosti zhruba 67 KB data byla přijata bez problémů a za stejnou dobu, jako při nastavení `MAX_READ_BUFFER=2048`. Ve většině případů je tedy možné ponechat standardní nastavení. Hodnoty velikosti bufferů musí být sudá čísla.

`MAX_WRITE_BLOCK` – nastavení, kolik bajtů maximálně se najednou zapíše do výstupního bufferu, respektive se najednou pošle, standardně je nastaveno 256 bajtů

`MAX_READ_BLOCK` – nastavení, kolik bajtů maximálně se najednou zapíše do vstupního bufferu, respektive se najednou načte, standardně je nastaveno 256 bajtů

Obě tyto hodnoty by měly být logicky menší než doporučené velikosti bufferů a nejlépe několikrát. Nastavení způsobí, např. při standardně nastaveném `MAX_WRITE_BLOCK` a teoretickém nahromadění 1024 bajtů dat k odeslání, že by data byla odeslána po čtyřech částech. Tyto hodnoty tedy zabraňují případnému posílání či přijímání dat po

zbytečně velkých částech. Ovšem příliš nízké hodnoty těchto údajů negativně působí na rychlost komunikace. Doporučuji hodnoty vyšší než 32 bajtů, naopak nastavení vyšší než 512 bajtů bude poměrně bez efektu (512 B se v bufferech jen tak nenahromadí). I u těchto údajů je možné s klidným svědomím ponechat standardní nastavení.

Druhá skupina maker obsahuje zprávy, které jsou odesílané rodiči (ovšem není-li `NULL`), jako reakce na událost, která nastala na portu. Tato část je pro vás podstatná jen v případě, že třídu používáte v okenní aplikaci a máte tedy možnost na poslané zprávy reagovat. Např. pošle-li se poslední bajt z výstupního bufferu portu, rodiči přijde zpráva `WM_COMM_TXEMPTY_DETECTED`. V případě, že by jste potřebovaly hodnoty těchto maker z jakéhokoli důvodu změnit, upozorňuji, že to musí být hodnoty vyšší než `WM_USER`. Nejpodstatnější zpráva `WM_COMM_RXCHAR`, která oznamuje příjem dat byla popsána v části příručky o posílání a příjmu dat, a zde již uvedena nebude. Pro všechny ostatní zprávy platí, že jejich `WPARAM` je nastaven na `0` a `LPARAM` obsahuje číslo portu, kde k události došlo (např. pro `COM1` to je 1), až na výjimku u zprávy `WM_COMM_ERR_DETECTED`, kde `WPARAM` obsahuje chybu, která nastala.

WM_COMM_BREAK_DETECTED – oznamuje, že bylo na vstupu detekováno přerušení

WM_COMM_CTS_DETECTED – oznamuje, že `CTS(clear-to-send)` signál se změnil

WM_COMM_DSR_DETECTED – oznamuje, že `DSR(data-set-ready)` signál se změnil

WM_COMM_ERR_DETECTED – oznamuje, že nastala line-status chyba, může to být `CE_FRAME`, `CE_OVERRUN`, nebo `CE_RXPARITY`, pravděpodobně je to (ve stejném pořadí), důsledek špatného nastavení baudrate, ztrátou znaku či znaků, nebo chybou parity

WM_COMM_RING_DETECTED – oznamuje, že byl detekován ring indikátor

WM_COMM_RLSD_DETECTED – oznamuje, že RLSD(receive-line-signal-detect) signál se změnil

WM_COMM_RXFLAG_DETECTED – oznamuje, že událost znaku byla přijata, vyvolá se po nastavení atributu struktury DCB EvtChar

WM_COMM_TXEMPTY_DETECTED – oznamuje, že byl z výstupního bufferu odeslán poslední znak a nyní je buffer prázdný.

Při běžném využití třídy v nějaké jednodušší aplikaci není nutné na tyto zprávy nějakým způsobem reagovat.

Návratové hodnoty funkcí vláken

Jak bylo zmíněno v úvodu, třída má implementována zvláštní vlákna pro čtení i zápis, aby tyto operace nezdržovaly hlavní vlákno aplikace. Tyto vlákna mohou mít různé návratové hodnoty:

Vlákno pro odesílání:

10 – vlákno skončilo po normálním průběhu

5 – vlákno skončilo po chybě v odesílání dat

Vlákno pro příjem:

20 – vlákno skončilo po normálním průběhu

17 – vlákno skončilo po chybě ve zpracování události na portu, pravděpodobně v důsledku chyby v příjmu dat

15 – vlákno skončilo po vážné chybě při čekání na událost na portu

Ukázka jednoduché konzolové aplikace

Tato jednoduchá aplikace se pokusí otevřít port se standardním nastavením a začne logovat přijatá i odeslaná data. V zakomentované části je ukázána případná změna nastavení již otevřeného portu. Případné výjimky při inicializaci i ukončení aplikace jsou odchyceny. V menu aplikace je na výběr možnost poslání řetězce (s možností zadat i mezery) a možnost zjistit, jestli nebyla přijata nějaká nová data. Po zadání jiné volby než jsou znaky 0 až 9 je aplikace ukončena. Následuje zdrojový kód.

```
#include <iostream>
#include "rs232.h"

using namespace std;

int main(int argc,  char* argv[])
{
	CSerialPort SP;
	DCB myDcb;
	try
	{
		SP.OpenCom();//otevření portu
		//začátek logování
		SP.StartLogInput("prijato.txt");
		SP.StartLogOutput("poslano.txt");
		//případná změna nastaveni
		//např. nastaveni rizeni toku TRS on TX
		/*myDcb=SP.GetPortSettings();
		myDcb.BaudRate=RTS_CONTROL_TOGGLE;
		SP.UpdatePortSettings(myDcb);*/
	}
	catch (CSerialException& SE)
	{
		cout << SE.GetErrorCode() << ": "
			<< SE.GetErrorText() <<endl;
		system("Pause");
		return 0;
	}
	//jednoduchý program
	if (SP.IsConnected())
	{
		bool bPokracuj=true;
		while(bPokracuj)
		{
			cout << "******** MENU ********" << endl;
			cout << "1- posli retezec" << endl;
			cout << "2- zjisti nova data" << endl;
			cout << "jine znaky nez 0..9 konec \n\n"
			cout << "Zadejte volbu : ";
			int volba;
			cin >> volba;
			if (cin.fail())
			{
				cin.clear();
				bPokracuj=false;
			}else
			{
				char *lpzBuf=0;
				ULONG ulCount;
				switch (volba)
				{
					case 1: lpzBuf=new char[100];
					FillMemory(lpzBuf, 100, 0);
					cout<<endl<<"Zadejte text: ";
					cin.get();
					cin.getline(lpzBuf,99);
					SP.Send(lpzBuf);
					cout << "Odeslano... "<<endl;
```

```
                                delete [] lpzBuf;
                                break;

                                case 2:
                                ulCount=SP.SizeOfReceived();
                                if (ulCount>0)
                                {
                                 lpzBuf=new char[ulCount+1];
                                 FillMemory(lpzBuf,
                                            ulCount+1, 0);
                                 SP.GetReceived(ulCount,
                                            reinterpret_cast
                                            <unsigned char*>
                                            (lpzBuf));
                                 cout << endl <<"Nova data: "
                                      << lpzBuf << endl;
                                 delete [] lpzBuf;
                                }else
                                  cout << "Zadna nova data."
                                       << endl;
                                 break;
                        }
                        system("Pause");
                        system("CLS");
                }
        }
        try
        {
                //uzavření portu,
                //automaticky se ukončí i logování
                SP.CloseCom();
        }
        catch (CSerialException& SE)
        {
                cout << SE.GetErrorCode() << ": "
                        << SE.GetErrorText() << endl;
        }
    }

    system("Pause");
    return 0;
}
```

ÚDAJE PRO KNIHOVNICKOU DATABÁZI

Název práce	Sériová komunikace v operačním systému Microsoft Windows
Autor práce	Pavlík Pavel
Obor	Informační technologie
Rok obhajoby	2007
Vedoucí práce	Ing. Hájek Martin
Anotace	Cílem této bakalářské práce je vytvořit funkční třídu pro komunikaci po sériovém portu v prostředí 32-bitového operačního systému Microsoft Windows. Třída musí být kvůli lepší kompatibilitě s ostatními verzemi operačního systému Windows vytvořena pouze za použití programovacích jazyků C/C++ a Windows API. Další požadavky kladené na tuto třídu je vytvoření zvláštních vláken pro operace čtení a zápis na port z důvodu zamezení zdržování hlavního vlákna aplikace těmito operacemi. Již vytvořené třídy pro komunikaci po sériovém portu nesplňují všechny výše zmíněné požadavky, a proto je cílem navrhnout novou třídu, která bude dané požadavky splňovat.
Klíčová slova	RS-232, sériový port, sériové rozhraní, řízení toku, parita, sériová komunikace, Windows, architektura Windows, Windows API, multithreading, multitasking, synchronizace vláken