用 MFC 实现串口编程

龚建伟评论：本文既写了在 Windows 中怎样用 VC 控件 MSComm，又说明了 API 技术编程方法，在写用 MSComm 控件时，数据类型的转换说得不是太明白，初次涉猎串口编程的朋友恐怕看了还是编不出来；直接从底层编写的部分值得一读，说得较为详细，但你得先从 VC 教本上看一看什么是线程。

本文详细介绍了串行通信的基本原理，以及在 Windows NT、Win98 环境下用 MFC 实现串口（COM）通信的方法：使用 ActiveX 控件或 Win API.并给出用 Visual C++6.0 编写的相应 MFC32 位应用程序。关键词：串行通信、VC++6.0、ActiveX 控件、Win API、MFC32 位应用程序、事件驱动、非阻塞通信、多线程.

在 Windows 应用程序的开发中，我们常常需要面临与外围数据源设备通信的问题。计算机和单片机（如 MCS-51）都具有串行通信口，可以设计相应的串口通信程序，完成二者之间的数据通信任务。

实际工作中利用串口完成通信任务的时候非常之多。已有一些文章介绍串口编程的文章在计算机杂志上发表。但总的感觉说来不太全面，特别是介绍 32 位下编 程的更少，且很不详细。笔者在实际工作中积累了较多经验,结合硬件、软件，重点提及比较新的技术，及需要注意的要点作一番探讨。希望对各位需要编写串口通 信程序的朋友有一些帮助。

一．串行通信的基本原理

串行端口的本质功能是作为 CPU 和串行设备间的编码转换器。当数据从 CPU 经过串行端口发送出去时，字节数据转换为串行的位。在接收数据时，串行的位被转换为字节数据。

在 Windows 环境（Windows NT、Win98、Windows2000）下，串口是系统资源的一部分。
应用程序要使用串口进行通信，必须在使用之前向操作系统提出资源申请要求（打开串口），通信完成后必须释放资源（关闭串口）。

二．串口信号线的接法

一个完整的 RS-232C 接口有 22 根线，采用标准的 25 芯插头座（或者 9 芯插头座）。25 芯和 9 芯的主要信号线相同。以下的介绍是以 25 芯的 RS-232C 为例。

①主要信号线定义：

2 脚：发送数据 TXD； 3 脚：接收数据 RXD； 4 脚：请求发送 RTS； 5 脚：清除发送 CTS；

6 脚：数据设备就绪 DSR；20 脚：数据终端就绪 DTR； 8 脚：数据载波检测 DCD；

1 脚：保护地； 7 脚：信号地。

②电气特性：

数据传输速率最大可到 20K bps,最大距离仅 15m.

注：看了微软的 MSDN 6.0，其 Windows API 中关于串行通讯设备（不一定都是串口 RS-232C 或 RS-422 或 RS-449）速率的设置，最大可支持到 RS_256000，即 256K bps! 也不知道到底是什么串行通讯设备？但不管怎样，一般主机和单片机的串口通讯大多都在 9600 bps,可以满足通讯需求。

③接口的典型应用：

大多数计算机应用系统与智能单元之间只需使用 3 到 5 根信号线即可工作。这时，除了 TXD、RXD 以外，还需使用 RTS、CTS、DCD、DTR、DSR 等信号线。（当然，在程序中也需要对相应的信号线进行设置。）

以上接法，在设计程序时，直接进行数据的接收和发送就可以了，不需要对信号线的状态进行判断或设置。（如果应用的场合需要使用握手信号等，需要对相应的信号线的状态进行监测或设置。）

三．16 位串口应用程序的简单回顾

16 位串口应用程序中，使用的 16 位的 Windows API 通信函数:

① OpenComm() 打开串口资源，并指定输入、输出缓冲区的大小（以字节计）；

CloseComm() 关闭串口;

例：int idComDev;

idComDev = OpenComm("COM1", 1024, 128);

CloseComm(idComDev);

② BuildCommDCB() 、setCommState()填写设备控制块 DCB，然后对已打开的串口进行参数配置;

例：DCB dcb;

BuildCommDCB("COM1:2400,n,8,1", &dcb);

SetCommState(&dcb);

③ ReadComm 、WriteComm()对串口进行读写操作，即数据的接收和发送.

例：char *m_pRecieve; int count;

ReadComm(idComDev,m_pRecieve,count);

Char wr[30]; int count2;

WriteComm(idComDev,wr,count2);

16 位下的串口通信程序最大的特点就在于：串口等外部设备的操作有自己特有的 API 函数；而 32 位程序则把串口操作(以及并口等)和文件操作统一起来了，使用类似的操作。

四．在 MFC 下的 32 位串口应用程序

回到页顶

32 位下串口通信程序可以用两种方法实现：利用 ActiveX 控件；使用 API 通信函数。

使用 ActiveX 控件，程序实现非常简单，结构清晰，缺点是欠灵活；使用 API 通信函数的优缺点则基本上相反。

以下介绍的都是在单文档（SDI）应用程序中加入串口通信能力的程序。

㈠ 使用 ActiveX 控件：

VC++ 6.0 提供的 MSComm 控件通过串行端口发送和接收数据，为应用程序提供串行通信功能。使用非常方便，但可惜的是，很少有介绍 MSComm 控件的资料。

⑴. 在当前的 Workspace 中插入 MSComm 控件。

Project 菜单------>Add to Project---->Components and Controls----->Registered

ActiveX Controls--->选择 Components: Microsoft Communications Control, version 6.0 插入到当前的 Workspace 中。

结果添加了类 CMSComm(及相应文件：mscomm.h 和 mscomm.cpp)。

⑵. 在 MainFrm.h 中加入 MSComm 控件。

```
protected:
    CMSComm m_ComPort;
```

在 Mainfrm.cpp::OnCreare()中：

```
    DWORD style=WS_VISIBLE|WS_CHILD;
    if
(!m_ComPort.Create(NULL,style,CRect(0,0,0,0),this,ID_COMMCTRL)){
TRACE0("Failed to create OLE Communications Control\n");
return -1;     // fail to create
        }
```

⑶.初始化串口

```
m_ComPort.SetCommPort(1);    //选择 COM1
m_ComPort. SetInBufferSize(1024); //设置输入缓冲区的大小，Bytes
m_ComPort. SetOutBufferSize(512); //设置输入缓冲区的大小，Bytes//

if(!m_ComPort.GetPortOpen()) //打开串口
m_ComPort.SetPortOpen(TRUE);
m_ComPort.SetInputMode(1); //设置输入方式为二进制方式
m_ComPort.SetSettings("9600,n,8,1"); //设置波特率等参数
```

m_ComPort.SetRThreshold(1); //为 1 表示有一个字符引发一个事件

m_ComPort.SetInputLen(0);

⑷. 捕捉串口事项。

MSComm 控件可以采用轮询或事件驱动的方法从端口获取数据。我们介绍比较使用的事件驱动方法：有事件（如接收到数据）时通知程序。在程序中需要捕获并处理这些通讯事件。

在 MainFrm.h 中：

protected:

afx_msg void OnCommMscomm();

DECLARE_EVENTSINK_MAP()

在 MainFrm.cpp 中：

BEGIN_EVENTSINK_MAP(CMainFrame,CFrameWnd)

ON_EVENT(CMainFrame,ID_COMMCTRL,1,OnCommMscomm,VTS_NONE)　　//映射 ActiveX 控件事件

END_EVENTSINK_MAP()

⑸. 串口读写.

完成读写的函数的确很简单，GetInput()和 SetOutput()就可。两个函数的原型是：

VARIANT GetInput(); 及 void SetOutput(const VARIANT& newValue);都要使用 VARIANT 类型（所有 Idispatch::Invoke 的参数和返回值在内部都是作为 VARIANT 对象处理的）。

无论是在PC机读取上传数据时还是在PC机发送下行命令时，我们都习惯于使用字符串的形式（也可以说是数组形式）。查阅 VARIANT 文档知道，可以用 BSTR

表示字符串，但遗憾的是所有的 BSTR 都是包含宽字符，即使我们没有定义 _UNICODE_UNICODE 也是这样！ WinNT 支持宽字符，而 Win95 并不支持。为解决上述问题，我们在实际工作中使用 CbyteArray，给出相应的部分程序如下：

```
        void CMainFrame::OnCommMscomm(){

         VARIANT vResponse;     int k;

if(m_commCtrl.GetCommEvent()==2) {

k=m_commCtrl.GetInBufferCount(); //接收到的字符数目

if(k>0) {

vResponse=m_commCtrl.GetInput(); //read

SaveData(k,(unsigned char*) vResponse.parray->pvData);

} // 接收到字符，MSComm 控件发送事件 }

     。。。。。 // 处理其他 MSComm 控件

}

void CMainFrame::OnCommSend() {

。。。。。。。。 // 准备需要发送的命令，放在 TxData[]中

CByteArray array;

array.RemoveAll();

array.SetSize(Count);

for(i=0;i<Count;i++)

array.SetAt(i, TxData[i]);

     m_ComPort.SetOutput(COleVariant(array)); // 发送数据

}
```

请大家认真关注第⑷、⑸中内容，在实际工作中是重点、难点所在。

㈡ 使用 32 位的 API 通信函数:

可能很多朋友会觉得奇怪：用 32 位 API 函数编写串口通信程序，不就是把 16 位的 API 换成 32 位吗？16 位的串口通信程序可是多年之前就有很多人研讨过了……

此文主要想介绍一下在 API 串口通信中如何结合非阻塞通信、多线程等手段，编写出高质量的通信程序。特别是在 CPU 处理任务比较繁重、与外围设备中有大量的通信数据时，更有实际意义。

⑴. 在中 MainFrm.cpp 定义全局变量

```
HANDLE          hCom; // 准备打开的串口的句柄

HANDLE          hCommWatchThread ;//辅助线程的全局函数
```

⑵. 打开串口，设置串口

```
hCom =CreateFile( "COM2", GENERIC_READ | GENERIC_WRITE, // 允许读写

                  0,                    // 此项必须为 0

                  NULL,                 // no security attrs

                  OPEN_EXISTING,        //设置产生方式

                  FILE_FLAG_OVERLAPPED, // 我们准备使用异步通信

                  NULL );
```

请大家注意，我们使用了 FILE_FLAG_OVERLAPPED 结构。这正是使用 API 实现非阻塞通信的关键所在。

```
ASSERT(hCom!=INVALID_HANDLE_VALUE); //检测打开串口操作是否成功

SetCommMask(hCom, EV_RXCHAR|EV_TXEMPTY );//设置事件驱动的类型

SetupComm( hCom, 1024,512) ; //设置输入、输出缓冲区的大小

PurgeComm( hCom, PURGE_TXABORT | PURGE_RXABORT | PURGE_TXCLEAR

                    | PURGE_RXCLEAR ); //清干净输入、输出缓冲区

COMMTIMEOUTS CommTimeOuts ; //定义超时结构，并填写该结构
```

……

```
SetCommTimeouts( hCom, &CommTimeOuts ) ;//设置读写操作所允许的超时

DCB         dcb ; // 定义数据控制块结构

GetCommState(hCom, &dcb ) ; //读串口原来的参数设置

dcb.BaudRate =9600; dcb.ByteSize =8; dcb.Parity = NOPARITY;

dcb.StopBits = ONESTOPBIT ;dcb.fBinary = TRUE ;dcb.fParity = FALSE;

SetCommState(hCom, &dcb ) ; //串口参数配置
```

上述的 COMMTIMEOUTS 结构和 DCB 都很重要，实际工作中需要仔细选择参数。

⑶启动一个辅助线程，用于串口事件的处理。

Windows 提供了两种线程，辅助线程和用户界面线程。区别在于：辅助线程没有窗口，所以它没有自己的消息循环。但是辅助线程很容易编程，通常也很有用。

在次，我们使用辅助线程。主要用它来监视串口状态，看有无数据到达、通信有无错误；而主线程则可专心进行数据处理、提供友好的用户界面等重要的工作。

```
hCommWatchThread=

          CreateThread( (LPSECURITY_ATTRIBUTES) NULL, //安全属性

                    0,//初始化线程栈的大小，缺省为与主线程大小相同

                    (LPTHREAD_START_ROUTINE)CommWatchProc, //线程的全局函数

                    GetSafeHwnd(), //此处传入了主框架的句柄

                    0, &dwThreadID );

    ASSERT(hCommWatchThread!=NULL);
```

⑷为辅助线程写一个全局函数，主要完成数据接收的工作。

请注意 OVERLAPPED 结构的使用，以及怎样实现了非阻塞通信。

```
UINT CommWatchProc(HWND hSendWnd){
    DWORD dwEvtMask=0 ;
    SetCommMask( hCom, EV_RXCHAR|EV_TXEMPTY );//有哪些串口事件需要监视?
    WaitCommEvent( hCom, &dwEvtMask, os );// 等待串口通信事件的发生
    检测返回的 dwEvtMask，知道发生了什么串口事件：
    if ((dwEvtMask & EV_RXCHAR) == EV_RXCHAR){ // 缓冲区中有数据到达
    COMSTAT ComStat ; DWORD dwLength;
    ClearCommError(hCom, &dwErrorFlags, &ComStat ) ;
    dwLength = ComStat.cbInQue ; //输入缓冲区有多少数据?
    if (dwLength > 0) {
BOOL fReadStat ;
    fReadStat = ReadFile( hCom, lpBuffer，dwLength, &dwBytesRead,
                          &READ_OS( npTTYInfo ) ); //读数据
```

注:我们在 CreareFile()时使用了 FILE_FLAG_OVERLAPPED,现在 ReadFile()也必须使用

LPOVERLAPPED 结构.否则,函数会不正确地报告读操作已完成了.

使用 LPOVERLAPPED 结构, ReadFile()立即返回,不必等待读操作完成,实现非阻塞

通信.此时, ReadFile()返回 FALSE, GetLastError()返回 ERROR_IO_PENDING.

```
if (!fReadStat){
     if (GetLastError() == ERROR_IO_PENDING){
```

```
while(!GetOverlappedResult(hCom,
    &READ_OS( npTTYInfo ), & dwBytesRead, TRUE )){
    dwError = GetLastError();
    if(dwError == ERROR_IO_INCOMPLETE) continue;
                //缓冲区数据没有读完，继续
    …… ……
 ::PostMessage((HWND)hSendWnd,WM_NOTIFYPROCESS,0,0);//通知主线程，串口收到数据   }
```

所谓的非阻塞通信，也即异步通信。是指在进行需要花费大量时间的数据读写操作（不仅仅是指串行通信操作）时，一旦调用 ReadFile()、WriteFile()，就能立即返回，而让实际的读写操作在后台运行；相反，如使用阻塞通信，则必须在读或写操作全部完成后才能返回。由于操作可能需要任意长的时间才能完成，于是问题就出现了。

非常阻塞操作还允许读、写操作能同时进行（即重叠操作?），在实际工作中非常有用。

要使用非阻塞通信，首先在 CreateFile()时必须使用 FILE_FLAG_OVERLAPPED；然后在 ReadFile()时 lpOverlapped 参数一定不能为 NULL，接着检查函数调用的返回值，调用 GetLastError()，看是否返回 ERROR_IO_PENDING。如是，最后调用 GetOverlappedResult()返回重叠操作(overlapped operation)的结果;WriteFile()的使用类似。

⑸．在主线程中发送下行命令。

```
BOOL    fWriteStat ; char szBuffer[count];
        ……//准备好发送的数据，放在 szBuffer[]中
fWriteStat = WriteFile(hCom, szBuffer, dwBytesToWrite,
            &dwBytesWritten, &WRITE_OS( npTTYInfo ) ); //写数据
```

注:我们在 CreareFile()时使用了 FILE_FLAG_OVERLAPPED,现在 WriteFile()也必须使用 LPOVERLAPPED 结构.否则,函数会不正确地报告写操作已完成了.

使用 LPOVERLAPPED 结构,WriteFile()立即返回,不必等待写操作完成,实现非阻塞 通信.此时, WriteFile()返回 FALSE, GetLastError()返回 ERROR_IO_PENDING.

```
int err=GetLastError();

if (!fWriteStat) {

     if(GetLastError() == ERROR_IO_PENDING){

        while(!GetOverlappedResult(hCom, &WRITE_OS( npTTYInfo ),

                    &dwBytesWritten, TRUE )) {

            dwError = GetLastError();

            if(dwError == ERROR_IO_INCOMPLETE){

                    // normal result if not finished

                dwBytesSent += dwBytesWritten; continue; }
```

综上，我们使用了多线程技术，在辅助线程中监视串口，有数据到达时依靠事件驱动，读入数据并向主线程报告(发送数据在主线程中，相对说来，下行命令的数据 总是少得多)；并且，WaitCommEvent()、ReadFile()、WriteFile()都使用了非阻塞通信技术，依靠重叠 (overlapped) 读写操作，让串口读写操作在后台运行。

依托 vc6.0 丰富的功能，结合我们提及的技术，写出有强大控制能力的串口通信应用程序。就个人而言，我更偏爱 API 技术，因为控制手段要灵活的多,功能也要强大得多。

Serial Communications in Win32

Allen Denver
Microsoft Windows Developer Support

December 11, 1995

Allen seldom eats breakfast, but if he had to pick a favorite, Win32 serial communications would be the top choice.

Abstract

Serial communications in Microsoft Win32 is significantly different from serial communications in 16-bit Microsoft Windows? Those familiar with 16-bit serial communications functions will have to relearn many parts of the system to program serial communications properly. This article will help to accomplish this. Those unfamiliar with serial communications will find this article a helpful foundation for development efforts.

This article assumes the reader is familiar with the fundamentals of multiple threading and synchronization in Win32. In addition, a basic familiarity of the Win32 **heap** functions is useful to fully comprehend the memory management methods used by the sample, MTTTY, included with this article. For more information regarding these functions, consult the Platform SDK documentation, the Microsoft Win32 SDK Knowledge Base, or the Microsoft Developer Network Library. Application programming interfaces (APIs) that control user interface features of windows and dialog boxes, though not discussed here, are useful to know in order to fully comprehend the sample provided with this article. Readers unfamiliar with general Windows programming practices should learn some of the fundamentals of general Windows programming before taking on serial communications. In other words, get your feet wet before diving in head first.

Introduction

The focus of this article is on application programming interfaces (APIs) and methods that are compatible with Microsoft?Windows NT?*and* Windows 95; therefore, APIs supported on both platforms are the only ones discussed. Windows 95 supports the Win32?Telephony API (TAPI) and Windows NT 3.*x* does not; therefore, this discussion will not include TAPI. TAPI does deserve mention, however, in that it very nicely implements modem interfacing and call controlling. A production application that works with modems and

makes telephone calls should implement these features using the TAPI interface. This will allow seamless integration with the other TAPI-enabled applications that a user may have. Furthermore, this article does not discuss some of the configuration functions in Win32, such as GetCommProperties.

The article is broken into the following sections: Opening a port, reading and writing (nonoverlapped and overlapped), serial status (events and errors), and serial settings (DCB, flow control, and communications time-outs).

The sample included with this article, MTTTY: Multithreaded TTY, implements many of the features discussed here. It uses three threads in its implementation: a user interface thread that does memory management, a writer thread that controls all writing, and a reader/status thread that reads data and handles status changes on the port. The sample employs a few different data heaps for memory management. It also makes extensive use of synchronization methods to facilitate communication between threads.

Opening a Port

The CreateFile function opens a communications port. There are two ways to call CreateFile to open the communications port: overlapped and nonoverlapped. The following is the proper way to open a communications resource for overlapped operation:

```
HANDLE hComm;
hComm = CreateFile( gszPort,
                    GENERIC_READ | GENERIC_WRITE,
                    0,
                    0,
                    OPEN_EXISTING,
                    FILE_FLAG_OVERLAPPED,
                    0);
if (hComm == INVALID_HANDLE_VALUE)
   // error opening port; abort
```

Removal of the FILE_FLAG_OVERLAPPED flag from the call to CreateFile specifies nonoverlapped operation. The next section discusses overlapped and nonoverlapped operations.

The Platform SDK documentation states that when opening a communications port, the call to CreateFile has the following requirements:

- *fdwShareMode* must be zero. Communications ports cannot be shared in the same manner that files are shared. Applications using TAPI can use the TAPI functions to facilitate sharing resources between applications. For Win32 applications not using TAPI, handle inheritance or duplication is necessary to share the communications port. Handle duplication is beyond the scope of this article; please refer to the Platform SDK documentation for more information.

- *fdwCreate* must specify the OPEN_EXISTING flag.

- *hTemplateFile* parameter must be NULL.

One thing to note about port names is that traditionally they have been COM1, COM2, COM3, or COM4. The Win32 API does not provide any mechanism for determining what ports exist on a system. Windows NT and Windows 95 keep track of installed ports differently from one another, so any one method would not be portable across all Win32 platforms. Some systems even have more ports than the traditional maximum of four. Hardware vendors and serial-device-driver writers are free to name the ports anything they like. For this reason, it is best that users have the ability to specify the port name they want to use. If a port does not exist, an error will occur (ERROR_FILE_NOT_FOUND) after attempting to open the port, and the user should be notified that the port isn't available.

Reading and Writing

Reading from and writing to communications ports in Win32 is very similar to file input/output (I/O) in Win32. In fact, the functions that accomplish file I/O are the same functions used for serial I/O. I/O in Win32 can be done either of two ways: overlapped or nonoverlapped. The Platform SDK documentation uses the terms *asynchronous* and *synchronous* to connote these types of I/O operations. This article, however, uses the terms *overlapped* and *nonoverlapped*.

Nonoverlapped I/O is familiar to most developers because this is the traditional form of I/O, where an operation is requested and is assumed to be complete when the function returns. In the case of *overlapped I/O*, the system may return to the caller immediately even when an operation is not finished and will signal the caller when the operation completes. The program may use the time between the I/O request and its completion to perform some "background?work.

Reading and writing in Win32 is significantly different from reading and writing serial communications ports in 16-bit Windows. 16-bit Windows only has the **ReadComm** and **WriteComm** functions. Win32 reading and writing

can involve many more functions and choices. These issues are discussed below.

Nonoverlapped I/O

Nonoverlapped I/O is very straightforward, though it has limitations. An operation takes place while the calling thread is blocked. Once the operation is complete, the function returns and the thread can continue its work. This type of I/O is useful for multithreaded applications because while one thread is blocked on an I/O operation, other threads can still perform work. It is the responsibility of the application to serialize access to the port correctly. If one thread is blocked waiting for its I/O operation to complete, all other threads that subsequently call a communications API will be blocked until the original operation completes. For instance, if one thread were waiting for a `ReadFile` function to return, any other thread that issued a `WriteFile` function would be blocked.

One of the many factors to consider when choosing between nonoverlapped and overlapped operations is portability. Overlapped operation is not a good choice because most operating systems do not support it. Most operating systems support some form of multithreading, however, so multithreaded nonoverlapped I/O may be the best choice for portability reasons.

Overlapped I/O

Overlapped I/O is not as straightforward as nonoverlapped I/O, but allows more flexibility and efficiency. A port open for overlapped operations allows multiple threads to do I/O operations *at the same time* and perform other work while the operations are pending. Furthermore, the behavior of overlapped operations allows a single thread to issue many different requests and do work in the background while the operations are pending.

In both single-threaded and multithreaded applications, some synchronization must take place between issuing requests and processing the results. One thread will have to be blocked until the result of an operation is available. The advantage is that overlapped I/O allows a thread to do some work between the time of the request and its completion. If no work *can* be done, then the only case for overlapped I/O is that it allows for better user responsiveness.

Overlapped I/O is the type of operation that the MTTTY sample uses. It creates a thread that is responsible for reading the port's data and reading the port's status. It also performs periodic background work. The program creates another thread exclusively for writing data out the port.

Note Applications sometimes abuse multithreading systems by creating too many threads. Although using multiple threads can resolve many difficult problems, creating excessive threads is not the most efficient use of them in an application. Threads are less a strain on the system than processes but still require system resources such as CPU time and memory. An application that creates excessive threads may adversely affect the performance of the entire system. A better use of threads is to create a different request queue for each type of job and to have a worker thread issue an I/O request for each entry in the request queue. This method is used by the MTTTY sample provided with this article.

An overlapped I/O operation has two parts: the creation of the operation and the detection of its completion. Creating the operation entails setting up an OVERLAPPED structure, creating a manual-reset event for synchronization, and calling the appropriate function (ReadFile or WriteFile). The I/O operation may or may not be completed immediately. It is an error for an application to assume that a request for an overlapped operation always yields an overlapped operation. If an operation is completed immediately, an application needs to be ready to continue processing normally. The second part of an overlapped operation is to detect its completion. Detecting completion of the operation involves waiting for the event handle, checking the overlapped result, and then handling the data. The reason that there is more work involved with an overlapped operation is that there are more points of failure. If a nonoverlapped operation fails, the function just returns an error-return result. If an overlapped operation fails, it can fail in the creation of the operation or while the operation is pending. You may also have a time-out of the operation or a time-out waiting for the signal that the operation is complete.

Reading

The ReadFile function issues a read operation. ReadFileEx also issues a read operation, but since it is not available on Windows 95, it is not discussed in this article. Here is a code snippet that details how to issue a read request. Notice that the function calls a function to process the data if the ReadFile returns TRUE. This is the same function called if the operation becomes overlapped. Note the fWaitingOnRead flag that is

defined by the code; it indicates whether or not a read operation is overlapped. It is used to prevent the creation of a new read operation if one is outstanding.

```
DWORD dwRead;
BOOL fWaitingOnRead = FALSE;
OVERLAPPED osReader = {0};

// Create the overlapped event. Must be closed before exiting
// to avoid a handle leak.
osReader.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

if (osReader.hEvent == NULL)
   // Error creating overlapped event; abort.

if (!fWaitingOnRead) {
   // Issue read operation.
   if (!ReadFile(hComm, lpBuf, READ_BUF_SIZE, &dwRead, &osReader)) {
      if (GetLastError() != ERROR_IO_PENDING)     // read not delayed?
         // Error in communications; report it.
      else
         fWaitingOnRead = TRUE;
   }
   else {
      // read completed immediately
      HandleASuccessfulRead(lpBuf, dwRead);
    }
}
```

The second part of the overlapped operation is the detection of its completion. The event handle in the **OVERLAPPED** structure is passed to the **WaitForSingleObject** function, which will wait until the object is signaled. Once the event is signaled, the operation is complete. This does not mean that it was completed successfully, just that it was completed. The **GetOverlappedResult** function reports the result of the operation. If an error occurred, **GetOverlappedResult** returns FALSE and **GetLastError** returns the error code. If the operation was completed successfully, **GetOverlappedResult** will return TRUE.

Note **GetOverlappedResult** can detect completion of the operation, as well as return the operation's failure status. **GetOverlappedResult** returns FALSE and **GetLastError** returns ERROR_IO_INCOMPLETE when the operation is not completed. In addition, **GetOverlappedResult** can be made to block until the operation completes. This effectively turns the

overlapped operation into a nonoverlapped operation and is accomplished by passing TRUE as the *bWait* parameter.

Here is a code snippet that shows one way to detect the completion of an overlapped read operation. Note that the code calls the same function to process the data that was called when the operation completed immediately. Also note the use of the fWaitingOnRead flag. Here it controls entry into the detection code, since it should be called only when an operation is outstanding.

```
#define READ_TIMEOUT      500      // milliseconds

DWORD dwRes;

if (fWaitingOnRead) {
   dwRes = WaitForSingleObject(osReader.hEvent, READ_TIMEOUT);
   switch(dwRes)
   {
      // Read completed.
      case WAIT_OBJECT_0:
          if (!GetOverlappedResult(hComm, &osReader, &dwRead, FALSE))
             // Error in communications; report it.
          else
             // Read completed successfully.
             HandleASuccessfulRead(lpBuf, dwRead);

          //  Reset flag so that another opertion can be issued.
          fWaitingOnRead = FALSE;
          break;

      case WAIT_TIMEOUT:
          // Operation isn't complete yet. fWaitingOnRead flag isn't
          // changed since I'll loop back around, and I don't want
          // to issue another read until the first one finishes.
          //
          // This is a good time to do some background work.
          break;

      default:
          // Error in the WaitForSingleObject; abort.
          // This indicates a problem with the OVERLAPPED structure's
          // event handle.
          break;
   }
```

```
}
```

Writing

Transmitting data out the communications port is very similar to reading in that it uses a lot of the same APIs. The code snippet below demonstrates how to issue and wait for a write operation to be completed.

```
BOOL WriteABuffer(char * lpBuf, DWORD dwToWrite)
{
   OVERLAPPED osWrite = {0};
   DWORD dwWritten;
   DWORD dwRes;
   BOOL fRes;

   // Create this write operation's OVERLAPPED structure's hEvent.
   osWrite.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
   if (osWrite.hEvent == NULL)
      // error creating overlapped event handle
      return FALSE;

   // Issue write.
   if (!WriteFile(hComm, lpBuf, dwToWrite, &dwWritten, &osWrite)) {
      if (GetLastError() != ERROR_IO_PENDING) {
         // WriteFile failed, but isn't delayed. Report error and abort.
         fRes = FALSE;
      }
      else
         // Write is pending.
         dwRes = WaitForSingleObject(osWrite.hEvent, INFINITE);
         switch(dwRes)
         {
            // OVERLAPPED structure's event has been signaled.
            case WAIT_OBJECT_0:
                 if (!GetOverlappedResult(hComm, &osWrite, &dwWritten,
FALSE))
                       fRes = FALSE;
                 else
                  // Write operation completed successfully.
                  fRes = TRUE;
                 break;

            default:
                 // An error has occurred in WaitForSingleObject.
```

```
                // This usually indicates a problem with the
                // OVERLAPPED structure's event handle.
                fRes = FALSE;
                break;
            }
        }
    }
    else
        // WriteFile completed immediately.
        fRes = TRUE;

    CloseHandle(osWrite.hEvent);
    return fRes;
}
```

Notice that the code above uses the **WaitForSingleObject** function with a time-out value of INFINITE. This causes the **WaitForSingleObject** function to wait forever until the operation is completed; this may make the thread or program appear to be "hung?when, in fact, the write operation is simply taking a long time to complete or flow control has blocked the transmission. Status checking, discussed later, can detect this condition, but doesn't cause the **WaitForSingleObject** to return. Three things can alleviate this condition:

- Place the code in a separate thread. This allows other threads to execute any functions they desire while our writer thread waits for the write to be completed. This is what the MTTTY sample does.
- Use COMMTIMEOUTS to cause the write to be completed after a time-out period has passed. This is discussed more fully in the "Communications Time-outs?section later in this article. This is also what the MTTTY sample allows.
- Change the **WaitForSingleObject** call to include a real time-out value. This causes more problems because if the program issues another operation while an older operation is still pending, new **OVERLAPPED** structures and overlapped events need to be allocated. This type of recordkeeping is difficult, particularly when compared to using a "job queue?design for the operations. The "job queue?method is used in the MTTTY sample.

  Note: The time-out values in synchronization functions are not communications time-outs. Synchronization time-outs cause **WaitForSingleObject** or **WaitForMultipleObjects** to return WAIT_TIMEOUT. This is not the same as a read or write operation timing out. Communications time-outs are described later in this article.

Because the **WaitForSingleObject** function in the above code snippet uses an INFINITE time-out, it is equivalent to using **GetOverlappedResult** with TRUE for the *fWait* parameter. Here is equivalent code in a much simplified form:

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

   // Create this writes OVERLAPPED structure hEvent.
   osWrite.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
   if (osWrite.hEvent == NULL)
      // Error creating overlapped event handle.
      return FALSE;

   // Issue write.
   if (!WriteFile(hComm, lpBuf, dwToWrite, &dwWritten, &osWrite)) {
      if (GetLastError() != ERROR_IO_PENDING) {
         // WriteFile failed, but it isn't delayed. Report error and
abort.
         fRes = FALSE;
      }
      else {
         // Write is pending.
         if (!GetOverlappedResult(hComm, &osWrite, &dwWritten, TRUE))
            fRes = FALSE;
         else
            // Write operation completed successfully.
            fRes = TRUE;
      }
   }
   else
      // WriteFile completed immediately.
      fRes = TRUE;

   CloseHandle(osWrite.hEvent);
   return fRes;
}
```

GetOverlappedResult is not always the best way to wait for an overlapped operation to be completed. For example, if an application needs to wait on another event handle, the first code snippet serves as a better model

than the second. The call to WaitForSingleObject is easy to change to WaitForMultipleObjects to include the additional handles on which to wait. This is what the MTTTY sample application does.

A common mistake in overlapped I/O is to reuse an OVERLAPPED structure before the previous overlapped operation is completed. If a new overlapped operation is issued before a previous operation is completed, a new OVERLAPPED structure must be allocated for it. A new manual-reset event for the hEvent member of the OVERLAPPED structure must also be created. Once an overlapped operation is complete, the OVERLAPPED structure and its event are free for reuse.

The only member of the OVERLAPPED structure that needs modifying for serial communications is the hEvent member. The other members of the OVERLAPPED structure should be initialized to zero and left alone. Modifying the other members of the OVERLAPPED structure is not necessary for serial communications devices. The documentation for ReadFile and WriteFile state that the Offset and OffsetHigh members of the OVERLAPPED structure must be updated by the application, or else results are unpredictable. This guideline should be applied to OVERLAPPED structures used for other types of resources, such as files.

Serial Status

There are two methods to retrieve the status of a communications port. The first is to set an event mask that causes notification of the application when the desired events occur. The SetCommMask function sets this event mask, and the WaitCommEvent function waits for the desired events to occur. These functions are similar to the 16-bit functions SetCommEventMask and EnableCommNotification, except that the Win32 functions do not post WM_COMMNOTIFY messages. In fact, the WM_COMMNOTIFY message is not even part of the Win32 API. The second method for retrieving the status of the communications port is to periodically call a few different status functions. Polling is, of course, neither efficient nor recommended.

Communications Events

Communications events can occur at any time in the course of using a communications port. The two steps involved in receiving notification of communications events are as follows:

- **SetCommMask** sets the desired events that cause a notification.

- **WaitCommEvent** issues a status check. The status check can be an overlapped or nonoverlapped operation, just as the read and write operations can be.

  Note: The word *event* in this context refers to communications events only. It does not refer to an event object used for synchronization.

Here is an example of the SetCommMask function:

```
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))
   // error setting communications mask
```

A description of each type of event is in Table 1.

Table 1. Communications Event Flags

Event Flag	**Description**
EV_BREAK	A break was detected on input.
EV_CTS	The CTS (clear-to-send) signal changed state. To get the actual state of the CTS line, **GetCommModemStatus** should be called.
EV_DSR	The DSR (data-set-ready) signal changed state. To get the actual state of the DSR line, **GetCommModemStatus** should be called.
EV_ERR	A line-status error occurred. Line-status errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY. To find the cause of the error, **ClearCommError** should be called.
EV_RING	A ring indicator was detected.
EV_RLSD	The RLSD (receive-line-signal-detect) signal changed state. To get the actual state of the RLSD line, **GetCommModemStatus** should be called. Note that this is commonly referred to as the CD (carrier detect) line.
EV_RXCHAR	A new character was received and placed in the input buffer. See the “Caveat?section below for a discussion of this flag.
EV_RXFLAG	The event character was received and placed in the input buffer. The event character is specified in the **EvtChar** member of the **DCB** structure discussed later. The “Caveat?section below also applies to this flag.
EV_TXEMPTY	The last character in the output buffer was sent to the serial port device. If a hardware buffer is used, this flag only indicates that all data has been sent to the hardware. There is no way to detect when the hardware buffer is empty without talking directly to the hardware with a device driver.

After specifying the event mask, the WaitCommEvent function detects the occurrence of the events. If the port is open for nonoverlapped operation,

then the **WaitCommEvent** function does not contain an **OVERLAPPED** structure. The function blocks the calling thread until the occurrence of one of the events. If an event never occurs, the thread may block indefinitely.

Here is a code snippet that shows how to wait for an EV_RING event when the port is open for nonoverlapped operation:

```
DWORD dwCommEvent;

if (!SetCommMask(hComm, EV_RING))
   // Error setting communications mask
   return FALSE;

if (!WaitCommEvent(hComm, &dwCommEvent, NULL))
   // An error occurred waiting for the event.
   return FALSE;
else
   // Event has occurred.
   return TRUE;
```

Note The Microsoft Win32 SDK Knowledge Base documents a problem with Windows 95 and the EV_RING flag. The above code never returns in Windows 95 because the EV_RING event is not detected by the system; Windows NT properly reports the EV_RING event. Please see the Win32 SDK Knowledge Base for more information on this bug.

As noted, the code above can be blocked forever if an event never occurs. A better solution would be to open the port for overlapped operation and wait for a status event in the following manner:

```
#define STATUS_CHECK_TIMEOUT      500   // Milliseconds

DWORD      dwRes;
DWORD      dwCommEvent;
DWORD      dwStoredFlags;
BOOL      fWaitingOnStat = FALSE;
OVERLAPPED osStatus = {0};

dwStoredFlags = EV_BREAK | EV_CTS | EV_DSR | EV_ERR | EV_RING |\
                EV_RLSD | EV_RXCHAR | EV_RXFLAG | EV_TXEMPTY ;
if (!SetCommMask(comHandle, dwStoredFlags))
   // error setting communications mask; abort
   return 0;

osStatus.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
```

```
   if (osStatus.hEvent == NULL)
      // error creating event; abort
      return 0;

   for ( ; ; ) {
      // Issue a status event check if one hasn't been issued already.
      if (!fWaitingOnStat) {
         if (!WaitCommEvent(hComm, &dwCommEvent, &osStatus)) {
            if (GetLastError() == ERROR_IO_PENDING)
               bWaitingOnStatusHandle = TRUE;
            else
               // error in WaitCommEvent; abort
               break;
         }
         else
            // WaitCommEvent returned immediately.
            // Deal with status event as appropriate.
            ReportStatusEvent(dwCommEvent);
      }

      // Check on overlapped operation.
      if (fWaitingOnStat) {
         // Wait a little while for an event to occur.
         dwRes = WaitForSingleObject(osStatus.hEvent,
STATUS_CHECK_TIMEOUT);
         switch(dwRes)
         {
             // Event occurred.
             case WAIT_OBJECT_0:
                 if (!GetOverlappedResult(hComm, &osStatus, &dwOvRes,
FALSE))
                    // An error occurred in the overlapped operation;
                    // call GetLastError to find out what it was
                    // and abort if it is fatal.
                 else
                    // Status event is stored in the event flag
                    // specified in the original WaitCommEvent call.
                    // Deal with the status event as appropriate.
                    ReportStatusEvent(dwCommEvent);

                 // Set fWaitingOnStat flag to indicate that a new
                 // WaitCommEvent is to be issued.
                 fWaitingOnStat = FALSE;
                 break;
```

```
            case WAIT_TIMEOUT:
                // Operation isn't complete yet.
fWaitingOnStatusHandle flag
                // isn't changed since I'll loop back around and I don't
want
                // to issue another WaitCommEvent until the first one
finishes.
                //
                // This is a good time to do some background work.
                DoBackgroundWork();
                break;

            default:
                // Error in the WaitForSingleObject; abort
                // This indicates a problem with the OVERLAPPED
structure's
                // event handle.
                CloseHandle(osStatus.hEvent);
                return 0;
        }
    }
}

CloseHandle(osStatus.hEvent);
```

The code above very closely resembles the code for overlapped reading. In fact, the MTTTY sample implements its reading and status checking in the same thread using **WaitForMultipleObjects** to wait for either the read event or the status event to become signaled.

There are two interesting side effects of **SetCommMask** and **WaitCommEvent**. First, if the communications port is open for nonoverlapped operation, **WaitCommEvent** will be blocked until an event occurs. If another thread calls **SetCommMask** to set a new event mask, that thread will be blocked on the call to **SetCommMask**. The reason is that the original call to **WaitCommEvent** in the first thread is still executing. The call to **SetCommMask** blocks the thread until the **WaitCommEvent** function returns in the first thread. This side effect is universal for ports open for nonoverlapped I/O. If a thread is blocked on *any* communications function and another thread calls a communications function, the second thread is blocked until the communications function returns in the first thread. The second interesting note about these functions is their use on a port open for overlapped operation. If **SetCommMask** sets a new event mask, any

pending **WaitCommEvent** will complete successfully, and the event mask produced by the operation is NULL.

Caveat

Using the EV_RXCHAR flag will notify the thread that a byte arrived at the port. This event, used in combination with the **ReadFile** function, enables a program to read data only *after* it is in the receive buffer, as opposed to issuing a read that *waits* for the data to arrive. This is particularly useful when a port is open for nonoverlapped operation because the program does not need to poll for incoming data; the program is notified of the incoming data by the occurrence of the EV_RXCHAR event. Initial attempts to code this solution often produce the following pseudocode, including one oversight covered later in this section:

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

if (!SetCommMask(hComm, EV_RXCHAR))
   // Error setting communications event mask.

for ( ; ; ) {
   if (WaitCommEvent(hComm, &dwCommEvent, NULL)) {
      if (ReadFile(hComm, &chRead, 1, &dwRead, NULL))
         // A byte has been read; process it.
      else
         // An error occurred in the ReadFile call.
         break;
   }
   else
      // Error in WaitCommEvent.
      break;
}
```

The above code waits for an EV_RXCHAR event to occur. When this happens, the code calls **ReadFile** to read the one byte received. The loop starts again, and the code waits for another EV_RXCHAR event. This code works fine when one or two bytes arrive in quick succession. The byte reception causes the EV_RXCHAR event to occur. The code reads the byte. If no other byte arrives before the code calls **WaitCommEvent** again, then all is fine; the next byte to arrive will cause the **WaitCommEvent** function to indicate the occurrence of the EV_RXCHAR event. If another single byte arrives before the code has a chance to reach the **WaitCommEvent** function, then

all is fine, too. The first byte is read as before; the arrival of the second byte causes the EV_RXCHAR flag to be set internally. When the code returns to the **WaitCommEvent** function, it indicates the occurrence of the EV_RXCHAR event and the second byte is read from the port in the **ReadFile** call.

The problem with the above code occurs when three or more bytes arrive in quick succession. The first byte causes the EV_RXCHAR event to occur. The second byte causes the EV_RXCHAR flag to be set internally. The next time the code calls **WaitCommEvent**, it indicates the EV_RXCHAR event. Now, a third byte arrives at the communications port. This third byte causes the system to attempt to set the EV_RXCHAR flag internally. Because this has already occurred when the second byte arrived, the arrival of the third byte goes unnoticed. The code eventually will read the first byte without a problem. After this, the code will call **WaitCommEvent**, and it indicates the occurrence of the EV_RXCHAR event (from the arrival of the second byte). The second byte is read, and the code returns to the **WaitCommEvent** function. The third byte waits in the system's internal receive buffer. The code and the system are now out of sync. When a fourth byte finally arrives, the EV_RXCHAR event occurs, and the code reads a single byte. It reads the third byte. This will continue indefinitely.

The solution to this problem seems as easy as increasing the number of bytes requested in the read operation. Instead of requesting a single byte, the code could request two, ten, or some other number of bytes. The problem with this idea is that it still fails when two or more extra bytes above the size of the read request arrive at the port in quick succession. So, if two bytes are read, then four bytes arriving in quick succession would cause the problem. Ten bytes requested would still fail if twelve bytes arrived in quick succession.

The real solution to this problem is to read from the port until no bytes are remaining. The following pseudocode solves the problem by reading in a loop until zero characters are read. Another possible method would be to call **ClearCommError** to determine the number of bytes in the buffer and read them all in one read operation. This method requires more sophisticated buffer management, but it reduces the number of reads when a lot of data arrives at once.

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

if (!SetCommMask(hComm, EV_RXCHAR))
   // Error setting communications event mask
```

```
for ( ; ; ) {
   if (WaitCommEvent(hComm, &dwCommEvent, NULL)) {
      do {
         if (ReadFile(hComm, &chRead, 1, &dwRead, NULL))
            // A byte has been read; process it.
         else
            // An error occurred in the ReadFile call.
            break;
      } while (dwRead);
   }
   else
      // Error in WaitCommEvent
      break;
}
```

The above code does not work correctly without setting the proper time-outs. Communications time-outs, discussed later, affect the behavior of the ReadFile operation in order to cause it to return without waiting for bytes to arrive. Discussion of this topic occurs later in the “Communications Time-outs?section of this article.

The above caveat regarding EV_RXCHAR also applies to EV_RXFLAG. If flag characters arrive in quick succession, EV_RXFLAG events may not occur for all of them. Once again, the best solution is to read all bytes until none remain.

The above caveat also applies to other events not related to character reception. If other events occur in quick succession some of the notifications will be lost. For instance, if the CTS line voltage starts high, then goes low, high, and low again, an EV_CTS event occurs. There is no guarantee of how many EV_CTS events will actually be detected with WaitCommEvent if the changes in the CTS line happen quickly. For this reason, WaitCommEvent cannot be used to keep track of the state of the line. Line status is covered in the “Modem Status?section later in this article.

Error Handling and Communications Status

One of the communications event flags specified in the call to SetCommMask is possibly EV_ERR. The occurrence of the EV_ERR event indicates that an error condition exists in the communications port. Other errors can occur in the port that do not cause the EV_ERR event to occur. In either case, errors associated with the communications port cause all I/O operations

to be suspended until removal of the error condition. ClearCommError is the function to call to detect errors and clear the error condition.

ClearCommError also provides communications status indicating why transmission has stopped; it also indicates the number of bytes waiting in the transmit and receive buffers. The reason why transmission may stop is because of errors or to flow control. The discussion of flow control occurs later in this article.

Here is some code that demonstrates how to call ClearCommError:

```
COMSTAT comStat;
DWORD   dwErrors;
BOOL    fOOP, fOVERRUN, fPTO, fRXOVER, fRXPARITY, fTXFULL;
BOOL    fBREAK, fDNS, fFRAME, fIOE, fMODE;

// Get and clear current errors on the port.
if (!ClearCommError(hComm, &dwErrors, &comStat))
    // Report error in ClearCommError.
    return;

// Get error flags.
fDNS = dwErrors & CE_DNS;
fIOE = dwErrors & CE_IOE;
fOOP = dwErrors & CE_OOP;
fPTO = dwErrors & CE_PTO;
fMODE = dwErrors & CE_MODE;
fBREAK = dwErrors & CE_BREAK;
fFRAME = dwErrors & CE_FRAME;
fRXOVER = dwErrors & CE_RXOVER;
fTXFULL = dwErrors & CE_TXFULL;
fOVERRUN = dwErrors & CE_OVERRUN;
fRXPARITY = dwErrors & CE_RXPARITY;

// COMSTAT structure contains information regarding
// communications status.
if (comStat.fCtsHold)
    // Tx waiting for CTS signal

if (comStat.fDsrHold)
    // Tx waiting for DSR signal

if (comStat.fRlsdHold)
    // Tx waiting for RLSD signal
```

```
if (comStat.fXoffHold)
    // Tx waiting, XOFF char rec'd

if (comStat.fXoffSent)
    // Tx waiting, XOFF char sent

if (comStat.fEof)
    // EOF character received

if (comStat.fTxim)
    // Character waiting for Tx; char queued with TransmitCommChar

if (comStat.cbInQue)
    // comStat.cbInQue bytes have been received, but not read

if (comStat.cbOutQue)
    // comStat.cbOutQue bytes are awaiting transfer
```

Modem Status (a.k.a. Line Status)

The call to **SetCommMask** may include the flags EV_CTS, EV_DSR, EV_RING, and EV_RLSD. These flags indicate changes in the voltage on the lines of the serial port. There is no indication of the actual status of these lines, just that a change occurred. The **GetCommModemStatus** function retrieves the actual state of these status lines by returning a bit mask indicating a 0 for low or no voltage and 1 for high voltage for each of the lines.

Please note that the term *RLSD* (Receive Line Signal Detect) is commonly referred to as the CD (Carrier Detect) line.

Note The EV_RING flag does not work in Windows 95 as mentioned earlier. The **GetCommModemStatus** function, however, does detect the state of the RING line.

Changes in these lines may also cause a flow-control event. The **ClearCommError** function reports whether transmission is suspended because of flow control. If necessary, a thread may call **ClearCommError** to detect whether the event is the cause of a flow-control action. Flow control is covered in the “Flow Control?section later in this article.

Here is some code that demonstrates how to call **GetCommModemStatus**:

```
DWORD dwModemStatus;
BOOL  fCTS, fDSR, fRING, fRLSD;
```

```
if (!GetCommModemStatus(hComm, &dwModemStatus))
   // Error in GetCommModemStatus;
   return;

fCTS = MS_CTS_ON & dwModemStatus;
fDSR = MS_DSR_ON & dwModemStatus;
fRING = MS_RING_ON & dwModemStatus;
fRLSD = MS_RLSD_ON & dwModemStatus;

// Do something with the flags.
```

Extended Functions

The driver will automatically change the state of control lines as necessary. Generally speaking, changing status lines is under the control of a driver. If a device uses communications port control lines in a manner different from RS-232 standards, the standard serial communications driver will not work to control the device. If the standard serial communications driver will not control the device, a custom device driver is necessary.

There are occasions when standard control lines *are* under the control of the application instead of the serial communications driver. For instance, an application may wish to implement its own flow control. The application would be responsible for changing the status of the RTS and DTR lines. EscapeCommFunction directs a communications driver to perform such extended operations. EscapeCommFunction can make the driver perform some other function, such as setting or clearing a BREAK condition. For more information on this function, consult the Platform SDK documentation, the Microsoft Win32 SDK Knowledge Base, or the Microsoft Developer Network (MSDN) Library.

Serial Settings

DCB Settings

The most crucial aspect of programming serial communications applications is the settings in the Device-Control Block (DCB) structure. The most common errors in serial communications programming occur in initializing the DCB structure improperly. When the serial communications functions

do not behave as expected, a close examination of the DCB structure usually reveals the problem.

There are three ways to initialize a DCB structure. The first method is to use the function **GetCommState**. This function returns the current DCB in use for the communications port. The following code shows how to use the **GetCommState** function:

```
DCB dcb = {0};

if (!GetCommState(hComm, &dcb))
   // Error getting current DCB settings
else
   // DCB is ready for use.
```

The second method to initialize a DCB is to use a function called **BuildCommDCB**. This function fills in the baud, parity type, number of stop bits, and number of data bits members of the DCB. The function also sets the flow-control members to default values. Consult the documentation of the **BuildCommDCB** function for details on which default values it uses for flow-control members. Other members of the DCB are unaffected by this function. It is the program's duty to make sure the other members of the DCB do not cause errors. The simplest thing to do in this regard is to initialize the DCB structure with zeros and then set the size member to the size, in bytes, of the structure. If the zero initialization of the DCB structure does not occur, then there may be nonzero values in the reserved members; this produces an error when trying to use the DCB later. The following function shows how to properly use this method:

```
DCB dcb;

FillMemory(&dcb, sizeof(dcb), 0);
dcb.DCBlength = sizeof(dcb);
if (!BuildCommDCB("9600,n,8,1", &dcb)) {
   // Couldn't build the DCB. Usually a problem
   // with the communications specification string.
   return FALSE;
}
else
   // DCB is ready for use.
```

The third method to initialize a DCB structure is to do it manually. The program allocates the DCB structure and sets each member with any value desired. This method does not deal well with changes to the DCB in future implementations of Win32 and is not recommended.

An application usually needs to set some of the DCB members differently than the defaults or may need to modify settings in the middle of execution. Once proper initialization of the DCB occurs, modification of individual members is possible. The changes to the DCB structure do not have any effect on the behavior of the port until execution of the SetCommState function. Here is a section of code that retrieves the current DCB, changes the baud, and then attempts to set the configuration:

```
   DCB dcb;

   FillMemory(&dcb, sizeof(dcb), 0);
   if (!GetCommState(hComm, &dcb))     // get current DCB
      // Error in GetCommState
      return FALSE;

   // Update DCB rate.
   dcb.BaudRate = CBR_9600 ;

   // Set new state.
   if (!SetCommState(hComm, &dcb))
      // Error in SetCommState. Possibly a problem with the
communications
      // port handle or a problem with the DCB structure itself.
```

Here is an explanation of each of the members of the DCB and how they affect other parts of the serial communications functions.

Note Most of this information is from the Platform SDK documentation. Because documentation is the official word in what the members actually are and what they mean, this table may not be completely accurate if changes occur in the operating system.

Table 2. The DCB Structure Members

Member	Description
DCBlength	Size, in bytes, of the structure. Should be set before calling **SetCommState** to update the settings.
BaudRate	Specifies the baud at which the communications device operates. This member can be an actual baud value, or a baud index.
fBinary	Specifies whether binary mode is enabled. The Win32 API does not support nonbinary mode transfers, so this member should be TRUE. Trying to use FALSE will not work.
fParity	Specifies whether parity checking is enabled. If this member is TRUE, parity checking is performed and parity errors are reported. This should not be confused with the **Parity** member, which controls the type of parity used

	in communications.
fOutxCtsFlow	Specifies whether the CTS (clear-to-send) signal is monitored for output flow control. If this member is TRUE and CTS is low, output is suspended until CTS is high again. The CTS signal is under control of the DCE (usually a modem), the DTE (usually the PC) simply monitors the status of this signal, the DTE does not change it.
fOutxDsrFlow	Specifies whether the DSR (data-set-ready) signal is monitored for output flow control. If this member is TRUE and DSR is low, output is suspended until DSR is high again. Once again, this signal is under the control of the DCE; the DTE only monitors this signal.
fDtrControl	Specifies the DTR (data-terminal-ready) input flow control. This member can be one of the following values:

	Value	**Meaning**
	DTR_CONTROL_DISABLE	Lowers the DTR line when the device is opened. The application can adjust the state of the line with **EscapeCommFunction**.
	DTR_CONTROL_ENABLE	Raises the DTR line when the device is opened. The application can adjust the state of the line with **EscapeCommFunction**.
	DTR_CONTROL_HANDSHAKE	Enables DTR flow-control handshaking. If this value is used, it is an error for the application to adjust the line with **EscapeCommFunction**.

fDsrSensitivity	Specifies whether the communications driver is sensitive to the state of the DSR signal. If this member is TRUE, the driver ignores any bytes received, unless the DSR modem input line is high.
fTXContinueOnXoff	Specifies whether transmission stops when the input buffer is full and the driver has transmitted the XOFF character. If this member is TRUE, transmission continues after the XOFF character has been sent. If this member is FALSE, transmission does not continue until the input buffer is within XonLim bytes of being empty and the driver has transmitted the XON character.
fOutX	Specifies whether XON/XOFF flow control is used during transmission. If this member is TRUE, transmission stops when the XOFF character is received and starts again when the XON character is received.
fInX	Specifies whether XON/XOFF flow control is used during reception. If this member is TRUE, the XOFF character is sent when the input buffer comes within XoffLim bytes of being full, and the XON character is sent when the input buffer comes within XonLim bytes of being empty.
fErrorChar	Specifies whether bytes received with parity errors are replaced with the character specified by the **ErrorChar** member. If this member is TRUE and the **fParity** member is TRUE, replacement occurs.
fNull	Specifies whether null bytes are discarded. If this member is TRUE, null

	bytes are discarded when received.	
fRtsControl	Specifies the RTS (request-to-send) input flow control. If this value is zero, the default is RTS_CONTROL_HANDSHAKE. This member can be one of the following values:	
	Value	**Meaning**
	RTS_CONTROL_DISABLE	Lowers the RTS line when the device is opened. The application can use **EscapeCommFunction** to change the state of the line.
	RTS_CONTROL_ENABLE	Raises the RTS line when the device is opened. The application can use **EscapeCommFunction** to change the state of the line.
	RTS_CONTROL_HANDSHAKE	Enables RTS flow-control handshaking. The driver raises the RTS line, enabling the DCE to send, when the input buffer has enough room to receive data. The driver lowers the RTS line, preventing the DCE to send, when the input buffer does not have enough room to receive data. If this value is used, it is an error for the application to adjust the line with **EscapeCommFunction**.
	RTS_CONTROL_TOGGLE	Specifies that the RTS line will be high if bytes are available for transmission. After all buffered bytes have been sent, the RTS line will be low. If this value is set, it would be an error for an application to adjust the line with **EscapeCommFunction**. This value is ignored in Windows 95; it causes the driver to act as if RTS_CONTROL_ENABLE were specified.
fAbortOnError	Specifies whether read and write operations are terminated if an error occurs. If this member is TRUE, the driver terminates all read and write operations with an error status (ERROR_IO_ABORTED) if an error occurs. The driver will not accept any further communications operations until the application has acknowledged the error by calling the **ClearCommError** function.	
fDummy2	Reserved; do not use.	
wReserved	Not used; must be set to zero.	
XonLim	Specifies the minimum number of bytes allowed in the input buffer before	

	the XON character is sent.	
XoffLim	Specifies the maximum number of bytes allowed in the input buffer before the XOFF character is sent. The maximum number of bytes allowed is calculated by subtracting this value from the size, in bytes, of the input buffer.	
Parity	Specifies the parity scheme to be used. This member can be one of the following values:	
	Value	**Meaning**
	EVENPARITY	Even
	MARKPARITY	Mark
	NOPARITY	No parity
	ODDPARITY	Odd
StopBits	Specifies the number of stop bits to be used. This member can be one of the following values:	
	Value	**Meaning**
	ONESTOPBIT	1 stop bit
	ONE5STOPBITS	1.5 stop bits
	TWOSTOPBITS	2 stop bits
XonChar	Specifies the value of the XON character for both transmission and reception.	
XoffChar	Specifies the value of the XOFF character for both transmission and reception.	
ErrorChar	Specifies the value of the character used to replace bytes received with a parity error.	
EofChar	Specifies the value of the character used to signal the end of data.	
EvtChar	Specifies the value of the character used to cause the EV_RXFLAG event. This setting does not actually cause anything to happen without the use of EV_RXFLAG in the **SetCommMask** function and the use of **WaitCommEvent**.	
wReserved1	Reserved; do not use.	

Flow Control

Flow control in serial communications provides a mechanism for suspending communications while one of the devices is busy or for some reason cannot

do any communication. There are traditionally two types of flow control: hardware and software.

A common problem with serial communications is write operations that actually do not write the data to the device. Often, the problem lies in flow control being used when the program did not specify it. A close examination of the DCB structure reveals that one or more of the following members may be TRUE: fOutxCtsFlow, fOutxDsrFlow, or fOutX. Another mechanism to reveal that flow control is enabled is to call ClearCommError and examine the COMSTAT structure. It will reveal when transmission is suspended because of flow control.

Before discussing the types of flow control, a good understanding of some terms is in order. Serial communications takes place between two devices. Traditionally, there is a PC and a modem or printer. The PC is labeled the Data Terminal Equipment (DTE). The DTE is sometimes called the *host*. The modem, printer, or other peripheral equipment is identified as the Data Communications Equipment (DCE). The DCE is sometimes referred to as the *device*.

Hardware flow control

Hardware flow control uses voltage signals on control lines of the serial cable to control whether sending or receiving is enabled. The DTE and the DCE must agree on the types of flow control used for a communications session. Setting the DCB structure to enable flow control just configures the DTE. The DCE also needs configuration to make certain the DTE and DCE use the same type of flow control. There is no mechanism provided by Win32 to set the flow control of the DCE. DIP switches on the device, or commands sent to it typically establish its configuration. The following table describes the control lines, the direction of the flow control, and the line's effect on the DTE and DCE.

Table 3. Hardware Flow-control Lines

Line and Direction	Effect on DTE/DCE
CTS (Clear To Send) Output flow control	DCE sets the line high to indicate that it can receive data. DCE sets the line low to indicate that it cannot receive data. If the fOutxCtsFlow member of the DCB is TRUE, then the DTE will not send data if this line is low. It will resume sending if the line is high. If the fOutxCtsFlow member of the DCB is FALSE, then the

	state of the line does not affect transmission.
DSR (Data Set Ready) Output flow control	DCE sets the line high to indicate that it can receive data. DCE sets the line low to indicate that it cannot receive data. If the **fOutxDsrFlow** member of the DCB is TRUE, then the DTE will not send data if this line is low. It will resume sending if the line is high. If the **fOutxDsrFlow** member of the DCB is FALSE, then the state of the line does not affect transmission.
DSR (Data Set Ready) Input flow control	If the DSR line is low, then data that arrives at the port is ignored. If the DSR line is high, data that arrives at the port is received. This behavior occurs if the **fDsrSensitivity** member of the DCB is set to TRUE. If it is FALSE, then the state of the line does not affect reception.
RTS (Ready To Send) Input flow control	The RTS line is controlled by the DTE. If the **fRtsControl** member of the DCB is set to RTS_CONTROL_HANDSHAKE, the following flow control is used: If the input buffer has enough room to receive data (at least half the buffer is empty), the driver sets the RTS line high. If the input buffer has little room for incoming data (less than a quarter of the buffer is empty), the driver sets the RTS line low. If the **fRtsControl** member of the DCB is set to RTS_CONTROL_TOGGLE, the driver sets the RTS line high when data is available for sending. The driver sets the line low when no data is available for sending. Windows 95 ignores this value and treats it the same as RTS_CONTROL_ENABLE. If the **fRtsControl** member of the DCB is set to RTS_CONTROL_ENABLE or RTS_CONTROL_DISABLE, the application is free to change the state of the line as it needs. Note that in this case, the state of the line does not affect reception. The DCE will suspend transmission when the line goes low. The DCE will resume transmission when the line goes high.
DTR (Data Terminal Ready)	The DTR line is controlled by the DTE. If the **fDtrControl** member of the DCB is set to

Input flow control	DTR_CONTROL_HANDSHAKE, the following flow control is used: If the input buffer has enough room to receive data (at least half the buffer is empty), the driver sets the DTR line high. If the input buffer has little room for incoming data (less than a quarter of the buffer is empty), the driver sets the DTR line low. If the **fDtrControl** member of the DCB is set to DTR_CONTROL_ENABLE or DTR_CONTROL_DISABLE, the application is free to change the state of the line as it needs. In this case, the state of the line does not affect reception. The DCE will suspend transmission when the line goes low. The DCE will resume transmission when the line goes high.

The need for flow control is easy to recognize when the CE_RXOVER error occurs. This error indicates an overflow of the receive buffer and data loss. If data arrives at the port faster than it is read, CE_RXOVER can occur. Increasing the input buffer size may cause the error to occur less frequently, but it does not completely solve the problem. Input flow control is necessary to completely alleviate this problem. When the driver detects that the input buffer is nearly full, it will lower the input flow-control lines. This should cause the DCE to stop transmitting, which gives the DTE enough time to read the data from the input buffer. When the input buffer has more room available, the voltage on flow-control lines is set high, and the DCE resumes sending data.

A similar error is CE_OVERRUN. This error occurs when new data arrives before the communications hardware and serial communications driver completely receives old data. This can occur when the transmission speed is too high for the type of communications hardware or CPU. This can also occur when the operating system is not free to service the communications hardware. The only way to alleviate this problem is to apply some combination of decreasing the transmission speed, replacing the communications hardware, and increasing the CPU speed. Sometimes third-party hardware drivers that are not very efficient with CPU resources cause this error. Flow control cannot completely solve the problems that cause the CE_OVERRUN error, although it may help to reduce the frequency of the error.

Software flow control

Software flow control uses data in the communications stream to control the transmission and reception of data. Because software flow control uses two special characters, XOFF and XON, binary transfers cannot use software flow control; the XON or XOFF character may appear in the binary data and would interfere with data transfer. Software flow control befits text-based communications or data being transferred that does not contain the XON and XOFF characters.

In order to enable software flow control, the **fOutX** and **fInX** members of the DCB must be set to TRUE. The **fOutX** member controls output flow control. The **fInX** member controls input flow control.

One thing to note is that the DCB allows the program to dynamically assign the values the system recognizes as flow-control characters. The **XoffChar** member of the DCB dictates the XOFF character for both input and output flow control. The **XonChar** member of the DCB similarly dictates the XON character.

For input flow control, the **XoffLim** member of the DCB specifies the minimum amount of free space allowed in the input buffer before the XOFF character is sent. If the amount of free space in the input buffer drops below this amount, then the XOFF character is sent. For input flow control, the **XonLim** member of the DCB specifies the minimum number of bytes allowed in the input buffer before the XON character is sent. If the amount of data in the input buffer drops below this value, then the XON character is sent.

Table 4 lists the behavior of the DTE when using XOFF/XON flow control.

Table 4. Software flow-control behavior

Flow-control character	Behavior
XOFF received by DTE	DTE transmission is suspended until XON is received. DTE reception continues. The **fOutX** member of the DCB controls this behavior.
XON received by DTE	If DTE transmission is suspended because of a previous XOFF character being received, DTE transmission is resumed. The **fOutX** member of the DCB controls this behavior.
XOFF sent from DTE	XOFF is automatically sent by the DTE when the receive buffer approaches full. The actual limit is dictated by the **XoffLim** member of the DCB. The **fInX** member of the DCB controls this behavior. DTE transmission is controlled by the **fTXContinueOnXoff** member of the DCB as described below.
XON sent from the DTE	XON is automatically sent by the DTE when the receive buffer

	approaches empty. The actual limit is dictated by the **XonLim** member of the DCB. The **fInX** member of the DCB controls this behavior.

If software flow control is enabled for input control, then the **fTXContinueOnXoff** member of the DCB takes effect. The **fTXContinueOnXoff** member controls whether transmission is suspended after the XOFF character is automatically sent by the system. If **fTXContinueOnXoff** is TRUE, then transmission continues after the XOFF is sent when the receive buffer is full. If **fTXContinueOnXoff** is FALSE, then transmission is suspended until the system automatically sends the XON character. DCE devices using software flow control will suspend their sending after the XOFF character is received. Some equipment will resume sending when the XON character is sent by the DTE. On the other hand, some DCE devices will resume sending after *any* character arrives. The **fTXContinueOnXoff** member should be set to FALSE when communicating with a DCE device that resumes sending after any character arrives. If the DTE continued transmission after it automatically sent the XOFF, the resumption of transmission would cause the DCE to continue sending, defeating the XOFF.

There is no mechanism available in the Win32 API to cause the DTE to behave the same way as these devices. The DCB structure contains no members for specifying suspended transmission to resume when *any* character is received. The XON character is the only character that causes transmission to resume.

One other interesting note about software flow control is that reception of XON and XOFF characters causes pending read operations to complete with zero bytes read. The XON and XOFF characters cannot be read by the application, since they are not placed in the input buffer.

A lot of programs on the market, including the Terminal program that comes with Windows, give the user three choices for flow control: Hardware, Software, or None. The Windows operating system itself does not limit an application in this way. The settings of the DCB allow for Software *and* Hardware flow control simultaneously. In fact, it is possible to separately configure each member of the DCB that affects flow control, which allows for several different flow-control configurations. The limits placed on flow-control choices are there for ease-of-use reasons to reduce confusion for end users. The limits placed on flow-control choices may also be because devices used for communications may not support all types of flow control.

Communications Time-outs

Another major topic affecting the behavior of read and write operations is time-outs. Time-outs affect read and write operations in the following way. If an operation takes longer than the computed time-out period, the operation is completed. There is no error code that is returned by **ReadFile**, **WriteFile**, **GetOverlappedResult**, or **WaitForSingleObject**. All indicators used to monitor the operation indicate that it completed successfully. The only way to tell that the operation timed out is that the number of bytes actually transferred are fewer than the number of bytes requested. So, if **ReadFile** returns TRUE, but fewer bytes were read than were requested, the operation timed out. If an overlapped write operation times out, the overlapped event handle is signaled and **WaitForSingleObject** returns WAIT_OBJECT_0. **GetOverlappedResult** returns TRUE, but dwBytesTransferred contains the number of bytes that were transferred before the time-out. The following code demonstrates how to handle this in an overlapped write operation:

```
BOOL WriteABuffer(char * lpBuf, DWORD dwToWrite)
{
   OVERLAPPED osWrite = {0};
   DWORD dwWritten;
   DWORD dwRes;
   BOOL  fRes;

   // Create this write operation's OVERLAPPED structure hEvent.
   osWrite.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
   if (osWrite.hEvent == NULL)
      // Error creating overlapped event handle.
      return FALSE;

   // Issue write
   if (!WriteFile(hComm, lpBuf, dwToWrite, &dwWritten, &osWrite)) {
      if (GetLastError() != ERROR_IO_PENDING) {
         // WriteFile failed, but it isn't delayed. Report error.
         fRes = FALSE;
      }
      else
         // Write is pending.
         dwRes = WaitForSingleObject(osWrite.hEvent, INFINITE);
         switch(dwRes)
         {
            // Overlapped event has been signaled.
```

```
            case WAIT_OBJECT_0:
                 if (!GetOverlappedResult(hComm, &osWrite, &dwWritten,
FALSE))
                      fRes = FALSE;
                 else {
                  if (dwWritten != dwToWrite) {
                     // The write operation timed out. I now need to
                     // decide if I want to abort or retry. If I retry,
                     // I need to send only the bytes that weren't sent.
                     // If I want to abort, I would just set fRes to
                     // FALSE and return.
                     fRes = FALSE;
                  }
                  else
                     // Write operation completed successfully.
                     fRes = TRUE;
                }
                 break;

            default:
                 // An error has occurred in WaitForSingleObject. This
usually
                // indicates a problem with the overlapped event handle.
                 fRes = FALSE;
                 break;
         }
      }
   }
   else {
      // WriteFile completed immediately.

      if (dwWritten != dwToWrite) {
          // The write operation timed out. I now need to
          // decide if I want to abort or retry. If I retry,
          // I need to send only the bytes that weren't sent.
          // If I want to abort, then I would just set fRes to
          // FALSE and return.
          fRes = FALSE;
      }
      else
          fRes = TRUE;
   }

   CloseHandle(osWrite.hEvent);
```

```
   return fRes;
}
```

The SetCommTimeouts function specifies the communications time-outs for a port. To retrieve the current time-outs for a port, a program calls the GetCommTimeouts function. An applications should retrieve the communications time-outs before modifying them. This allows the application to set time-outs back to their original settings when it finishes with the port. Following is an example of setting new time-outs using SetCommTimeouts:

```
COMMTIMEOUTS timeouts;

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

if (!SetCommTimeouts(hComm, &timeouts))
   // Error setting time-outs.
```

Note Once again, communications time-outs are not the same as time-out values supplied in synchronization functions. WaitForSingleObject, for instance, uses a time-out value to wait for an object to become signaled; this is not the same as a communications time-out.

Setting the members of the COMMTIMEOUTS structure to all zeros causes no time-outs to occur. Nonoverlapped operations will block until all the requested bytes are transferred. The ReadFile function is blocked until all the requested characters arrive at the port. The WriteFile function is blocked until all requested characters are sent out. On the other hand, an overlapped operation will not finish until all the characters are transferred or the operation is aborted. The following conditions occur until the operation is completed:

- **WaitForSingleObject** always returns WAIT_TIMEOUT if a synchronization time-out is supplied. **WaitForSingleObject** will block forever if an INFINITE synchronization time-out is used.
- **GetOverlappedResult** always returns FALSE and **GetLastError** returns ERROR_IO_INCOMPLETE if called directly after the call to **GetOverlappedResult**.

Setting the members of the **COMMTIMEOUTS** structure in the following manner causes read operations to complete immediately without waiting for any new data to arrive:

```
COMMTIMEOUTS timeouts;

timeouts.ReadIntervalTimeout = MAXDWORD;
timeouts.ReadTotalTimeoutMultiplier = 0;
timeouts.ReadTotalTimeoutConstant = 0;
timeouts.WriteTotalTimeoutMultiplier = 0;
timeouts.WriteTotalTimeoutConstant = 0;

if (!SetCommTimeouts(hComm, &timeouts))
   // Error setting time-outs.
```

These settings are necessary when used with an event-based read described in the “Caveat?section earlier. In order for **ReadFile** to return 0 bytes read, the **ReadIntervalTimeout** member of the **COMMTIMEOUTS** structure is set to MAXDWORD, and the **ReadTimeoutMultiplier** and **ReadTimeoutConstant** are both set to zero.

An application must *always* specifically set communications time-outs when it uses a communications port. The behavior of read and write operations is affected by communications time-outs. When a port is initially open, it uses default time-outs supplied by the driver or time-outs left over from a previous communications application. If an application assumes that time-outs are set a certain way, while the time-outs are actually different, then read and write operations may never complete or may complete too often.

Conclusion

This article serves as a discussion of some of the common pitfalls and questions that arise when developing a serial communications application. The Multithreaded TTY sample that comes with this article is designed using many of the techniques discussed here. Download it and try it out. Learning how it works will provide a thorough understanding of the Win32 serial communications functions.

Bibliography

Brain, Marshall. *Win32 System Services: The Heart of Windows NT*. Englewood Cliffs, NJ: Prentice Hall, 1994.

Campbell, Joe. *C Programmer's Guide to Serial Communications.* 2d ed. Indianapolis, IN: Howard W. Sams & Company, 1994.

Mirho, Charles, and Andy Terrice. "Create Communications Programs for Windows 95 with the Win32 Comm API.?*Microsoft Systems Journal 12* (December 1994). (MSDN Library, Books and Periodicals)

简单而强大的多线程串口编程工具 CserialPort 类(附 VC 基于 MFC 单文档协议通讯源程序及详细编程步骤)

作者：龚建伟  2001.11.09（任意转载，请注明来自啸峰工作室及网址）

老有人觉得 MSComm 通讯控件很土，更有人大声疾呼：忘了它吧。确实当我们对串口编程有了一定的了解后，应该用 API 函数写一个属于自己的串口程序，由 于编程者对程序了解，对程序修改自如。但我一直没有停止过用 MSComm 通讯控件，那么简单的东西，对付简单的任务完全可以，但当我们需要在程序中用多个 串口，而且还要做很多复杂的处理，那么最好不用 MSComm 通讯控件，如果这时你还不愿意自己编写底层，就用这个类：CserialPort 类。

这是 Remon Spekreijse 写的一个串口类， 地址在：

http://codeguru.earthweb.com/network/serialport.shtml

类作者 Remon Spekreijse 已作了一个基于对话框的同时检测 4 个串口示例的程序，在上面的网址和我主页的串口源码下载页也可以找到。我在这儿主要介绍如何将这个 类应用到 VC 中基于文档的程序中。为了加深对串口数据处理的了解，我们利用这个类解决如下问题：

问题：

串口 2（COM2）每隔 1 秒向串口 1（COM1）发送的 NEMA 格式的报文：串头为$，串尾为＊，中间为一个 xxxx 的整数(比如 2345,不足 4 位则前面以 0 代替代)，最后是 hh 校验，规定 hh 为 xxxx 四个数的半 BYTE 校验和,最后加上回车<CR>与换行 <LF>。整个数据包为$xxxx*hh<CR><LF>。
串口 1 收到上述报文后，校验正确后，将发来的数据显示在视窗中，并记下发来的正确帧数和错误帧数，若正确，还向串口 2 发送 Y，串口 2 收到 Y 后将收到的 Y 的计数显示在视窗中。

测试方法：
将三线制串口线联接上同一台计算机的两个串口，编好程序后就可测试。如果没有两个串口的微机，自己改改程序。

好了，你可以先下载源程序： scporttest.zip（大小：49KB，VC6，WIN9X/2000，SerialPort.h SerialPort.cpp 是两个类文件）

编程步骤：

◆1. 建立程序：
建立一个基于单文档的 MFC 应用程序 SCPortTest，所有步骤保持缺省状态。

◆2. 添加类文件：
将 SerialPort.h SerialPort.cpp 两个类文件复制到工程文件夹中，用 Project-Add to Project-Files 命令将上述两个文件加入工程。并在 SCPortTestView.h 中将头文件 SerialPort.h 说明： #include "SerialPort.h"。

◆3. 人工增加串口消息响应函数：OnCommunication(WPARAM ch, LPARAM port)
首先在 SCPortTestView.h 中添加串口字符接收消息 WM_COMM_RXCHAR（串口接收缓冲区内有一个字符）的响应函数声明：

```
//{{AFX_MSG(CSCPortTestView)
afx_msg LONG OnCommunication(WPARAM ch, LPARAM port);
//}}AFX_MSG
```

然后在 SCPortTestView.cpp 文件中进行 WM_COMM_RXCHAR 消息映射：

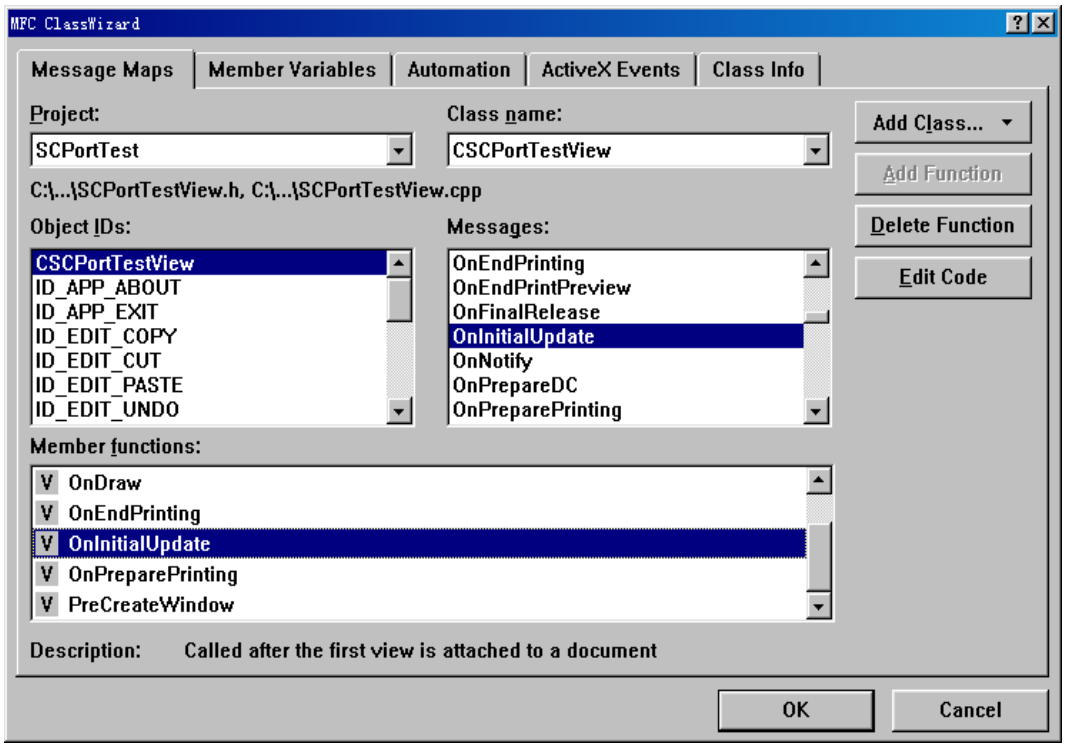
```
BEGIN_MESSAGE_MAP(CSCPortTestView, CView)
//{{AFX_MSG_MAP(CSCPortTestView)
ON_MESSAGE(WM_COMM_RXCHAR, OnCommunication)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

接着在 SCPortTestView.cpp 中加入函数的实现：

```
LONG CSCPortTestView::OnCommunication(WPARAM ch, LPARAM port)
{ …..  }
```

注意：由于这个串口类加入工程后，没有自动的消息映射机制，因此上述步骤均需要手工添加。

◆4 初始化串口
在视创建时初始化串口，首先利用 ClassWizardr 按下图生成 OnInitialUpdate() 函数。



接着在 SerialPort.h 文件中说明我们在程序中要用到的全局变量：
保存两个串口接收数据：

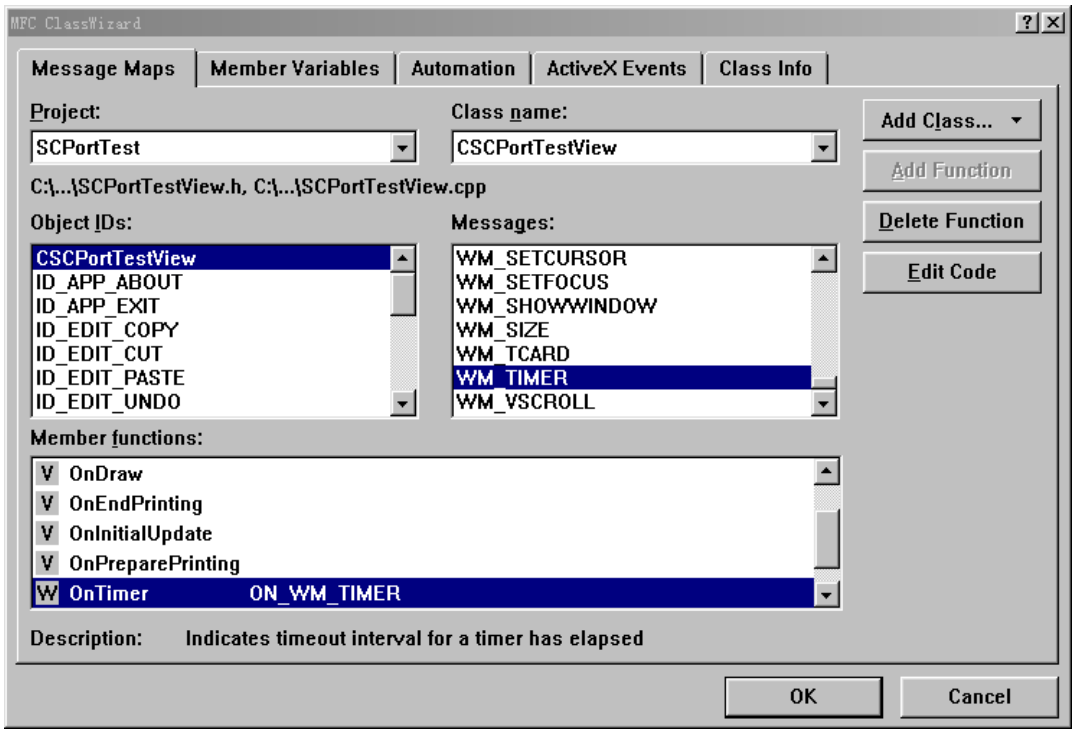
```
char m_chChecksum; //用于 COM1 的校验和计算
CString m_strRXhhCOM1; //用于存放 COM1 接收的半 BYTE 校验字节 hh
CString m_strRXDataCOM1; //COM1 接收数据
CString m_strRXDataCOM2; //COM2 接收数据
UINT m_nRXErrorCOM1; //COM1 接收数据错误帧数
UINT m_nRXErrorCOM2; //COM2 接收数据错误帧数
UINT m_nRXCounterCOM1; //COM1 接收数据错误帧数
UINT m_nRXCounterCOM2; //COM2 接收数据错误帧数 CString
```

再在 SerialPort.h 文件中说明串口类对象：CSerailPort m_ComPort[2]; （public）。
因为要初始化 2 个串口，所以这里用了数组。
下面是初始化串口 1 和串口 2：

```
void CSCPortTestView::OnInitialUpdate()
{
CView::OnInitialUpdate();
// TODO: Add your specialized code here and/or call the base class
m_chChecksum=0; //校验和置 0
m_nRXErrorCOM1=0; //COM1 接收数据错误帧数置 0
m_nRXErrorCOM2=0; //COM2 接收数据错误帧数置 0
m_nRXCounterCOM1=0; //COM1 接收数据错误帧数置 0
```

```
m_nRXCounterCOM2=0; //COM2 接收数据错误帧数置 0
m_strRXhhCOM1.Empty(); //清空半 BYTE 校验 hh 存储变量
for(int i=0;i<2;i++)
{
if (m_ComPort[i].InitPort(this,i+1,9600,'N',8,1,EV_RXFLAG |
EV_RXCHAR,512))
//portnr=1(2),baud=960,parity='N',databits=8,stopsbits=1,
//dwCommEvents=EV_RXCHAR|EV_RXFLAG,nBufferSize=512
{
m_ComPort[i].StartMonitoring(); //启动串口监视线程
if(i==1) SetTimer(1,1000,NULL); //设置定时器，1 秒后发送数据
}
else
{
CString str;
str.Format("COM%d 没有发现，或被其它设备占用",i+1);
AfxMessageBox(str);
}
}
}
```

◆5 利用 ClassWizard 按下图生成 CSCPortTestView 的时间消息 WM_TIMER 响应函数：




```
void CSCPortTestView::OnTimer(UINT nIDEvent)
{
```

```
// TODO: Add your message handler code here and/or call default
int randdata=rand()%9000; //产生 9000 以内的随机数
CString strSendData;
strSendData.Format("%04d",randdata);
SendString(strSendData, 2); //串口 2 发送数据；
CView::OnTimer(nIDEvent);
}
```

上面用到的 SendString()需按如下方式生成：
在 ClassView 中单击鼠标右键，在环境菜单中选择 Add Member Function:


```
void CSCPortTestView::SendString(CString &str, int Port)
{
char checksum=0,cr=CR,lf=LF;
char c1,c2;
for(int i=0;i<str.GetLength();i++)
checksum = checksum^str[i];
c2=checksum & 0x0f; c1=((checksum >> 4) & 0x0f);
if (c1 < 10) c1+= '0'; else c1 += 'A' - 10;
if (c2 < 10) c2+= '0'; else c2 += 'A' - 10;
CString str1;
str1='$'+str+"*"+c1+c2+cr+lf;
m_ComPort[Port-1].WriteToPort((LPCTSTR)str1);
}
```

请注意上面函数中是如何生成校验码的，要切记的是发送的校验码生成方式和对方接收的校验检测方式要一致。

◆6 在 OnCommunication(WPARAM ch, LPARAM port)函数中进行数据处理
说明：WPARAM、 LPARAM 类型是多态数据类型（polymorphic data type），在

WIN32 中为 32 位，支持多种数据类型，根据需要自动适应，这样程序有很强的适应性。在此我们可以分别理解为 char 和 integer 类型数据。
每当串口接收缓冲区内有一个字符时，就会产生一个 WM_COMM_RXCHAR 消息，触发 OnCommunication 函数，这时我们就可以在函数中进行数据处理，所以这个消息就是整个程序的"发动机"。
下面是根据本文最初提出的问题写出的处理函数：

```
LONG CSCPortTestView::OnCommunication(WPARAM ch, LPARAM port)
{
static int count1=0,count2=0,count3=0;
static char c1,c2;
static int flag;
CString strCheck="";

if(port==2) //COM2 接收到数据
{
CString strtemp=(char)ch;
if(strtemp=="Y")
{
m_nRXCounterCOM2++;
CString strtemp;
strtemp.Format("COM2: NO.%06d", m_nRXCounterCOM2);
CDC* pDC=GetDC(); //准备数据显示
pDC->TextOut(10,50,strtemp);//显示接收到的数据
ReleaseDC(pDC);
}
}

if(port==1) //COM1 接收到数据
{
m_strRXDataCOM1 += (char)ch;
switch(ch)
{
case '$':
m_chChecksum=0; //开始计算 CheckSum
flag=0;
break;
case '*':
flag=2;
c2=m_chChecksum & 0x0f; c1=((m_chChecksum >> 4) & 0x0f);
if (c1 < 10) c1+= '0'; else c1 += 'A' - 10;
if (c2 < 10) c2+= '0'; else c2 += 'A' - 10;
break;
case CR:
break;
```

```
case LF:
m_strRXDataCOM1.Empty();
break;
default:
if(flag>0)
{
m_strRXhhCOM1 += ch; //得到收到的校验值 hh
if(flag==1)
{
strCheck = strCheck+c1+c2; //计算得到的校验值 hh
if(strCheck!=m_strRXhhCOM1) //如果校验有错
{
m_strRXDataCOM1.Empty();
m_nRXErrorCOM1++; //串口 1 错误帧数加 1
}
else
{
m_nRXCounterCOM1++;
if(m_strRXDataCOM1.Left(1)=="$") //接收数据的第一个字符是$吗?
{
char tbuf[6];
char *temp=(char*)((LPCTSTR)m_strRXDataCOM1);
tbuf[0]=temp[1]; tbuf[1]=temp[2];
tbuf[2]=temp[3]; tbuf[3]=temp[4];
tbuf[4]=0; //0 表示字符串的结束，必要
int data=atoi(tbuf);
CString strDisplay1,strDisplay2;
strDisplay1.Format("NO. %06d: The reseived data
is %04d",m_nRXCounterCOM1,data);
strDisplay2.Format("Error Counter=%04d.",m_nRXErrorCOM1);
CDC* pDC=GetDC(); //准备数据显示
//int nColor=pDC->SetTextColor(RGB(255,255,0));
pDC->TextOut(10,10,strDisplay1);//显示接收到的数据
pDC->TextOut(30,30,strDisplay2);//显示错误帧数
//pDC->SetTextColor(nColor);
ReleaseDC(pDC);
}
CString str1="Y";
m_ComPort[0].WriteToPort((LPCTSTR)str1);//发送应答信号 Y
}
m_strRXhhCOM1.Empty();
}
flag--;
}
```

```
else
m_chChecksum ^= ch;
break;
}

}
return 0;
}
```

在基于单文档（SDI）程序中应用 MSCOMM 串口通讯控件（附源程序）

龚建伟　　（注：本文在网友刘先生的源程序基础上完成）

MSCOMM 串口通讯控件在基于对话框的程序中很好使用(可以参考我写的“串口调试助手源程序及详细编程过程一”），但有时我们需要在基于文档的程序中使用，在“能否在基于单文档的程序中使用串口通讯控件”一文中已说明为什么不能直接使用该控件，基于本文就如何在基于单文档的程序中使用该控件进行了详细说明。

下载源程序：（sdicomm.zip,WIN98/2000，VC6.0，45KB）

1.利用 MFC 向导建立基于单文档应用程序 SDIComm，所有步骤缺省，在项目中插入 MSCOMM 控件(Project->Add to Project->Components and Control...->Registered Active Controls->Mcriosoft Commmunications Control,V6.0，单击 INSERT,OK;

2.SDICommView.h 处理：
首先加入#include "mscomm.h";
然后在//{{AFX_MSG(CSDICommView)和//}}AFX_MSG 之间加入以下两行：
afx_msg void OnComm();
DECLARE_EVENTSINK_MAP()
最后结果是：
//{{AFX_MSG(CSDICommView)

```
afx_msg void OnComm(); //事件处理函数
DECLARE_EVENTSINK_MAP()
//}}AFX_MSG
```
再加入 CMSCOMM 类 PUBLIC 对象定义：
```
CMSComm m_MSComm;
```

3.利用 CLASSWIZARD 为 CSDICommView 类添加 WM_CREATE 函数，该函数在视窗初始化时执行。方法是在 CLASSWIZARD 中选择 MESSAGE MAP 卡，在 OBJECT IDS 中选择 CSDICommView，在 MESSAGES 中选择 WM_CREATE，双击添加 int CSDICommView::OnCreate(LPCREATESTRUCT lpCreateStruct)函数

```
int CSDICommView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CView::OnCreate(lpCreateStruct) == -1)
return -1;

// TODO: Add your specialized creation code here
m_MSComm.Create(NULL,0,CRect(0,0,0,0),this,IDC_MSCOMM1);
if(m_MSComm.GetPortOpen()) //如果串口是打开的，则行关闭串口
m_MSComm.SetPortOpen(FALSE);

m_MSComm.SetCommPort(2); //选择 COM2
m_MSComm.SetInBufferSize(1024); //接收缓冲区
m_MSComm.SetOutBufferSize(1024);//发送缓冲区
m_MSComm.SetInputLen(0);//设置当前接收区数据长度为 0,表示全部读取
m_MSComm.SetInputMode(1);//以二进制方式读写数据
m_MSComm.SetRThreshold(1);//接收缓冲区有 1 个及 1 个以上字符时，将引发接收数据的 OnComm 事件
m_MSComm.SetSettings("9600,n,8,1");//波特率 9600 无检验位，8 个数据位，1 个停止位

if(!m_MSComm.GetPortOpen())//如果串口没有打开则打开
m_MSComm.SetPortOpen(TRUE);//打开串口
else
AfxMessageBox("Open Serial Port Failure!");
m_MSComm.GetInput(); //先预读缓冲区以清除残留数据

return 0;
}
```

在这个函数中 我们应该注意
m_MSComm.Create(NULL,0,CRect(0,0,0,0),this,IDC_MSCOMM1);一句，其功能是初始化 串口类对象 m_MSComm，这在基于对话框中的程序是由 CLASSWIZARD 自动完成的。注意 IDC_MSCOMM1 还是源于对话框中的控件 ID 规 则，而且

IDC_MSCOMM1 必须与串口控件对应，在这里我们最好利用一个对话框，直接将控件拖入对话框中，就可以省不少事，这里我们利用 ABOUT 对 话框，方法如下：

在 ResoureView 中选择 IDD_ABOUTBOX 对话框，将控件图标拖入对话框中。

到目前为止，我们还未完成串口接收数据事件的初始化，还需进行以下步骤：

4.串口接收数据事件的初始化：在 SDICommView.cpp 文件中加入以下事件驱动说明：

```
BEGIN_EVENTSINK_MAP(CSDMSCom1View, CView)
//{{AFX_EVENTSINK_MAP(CAboutDlg)
ON_EVENT(CSDMSCom1View, IDC_MSCOMM1, 1 /* OnComm */, OnComm, VTS_NONE)
//}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()
```

手工添加函数 oncomm()：

在函数中我们作这样的测试处理，每当有 OnComm 事件,就向串口发送数据"OK，I've received some data!\r\n"，顺便提一下，在串口通讯程序中，一般的通讯处理均在 oncomm()函数中处理。

```
void CSDICommView::OnComm()
{
// TODO: Add your control notification handler code here
CString strtemp;
strtemp.Format("OK，I've received some data!\r\n");
m_MSComm.SetOutput(COleVariant(strtemp));//发送数据
}
```

最后是测试程序后，编译运行后，可将串口调试助手 V2.1（或其它调试工具）设置在 COM1，9600，n,8,1，单击手动发送，就可以看到效果了。

用 VC++6.0 实现 PC 机与单片机之间

串行通信的方法

湖南大学（长沙 410082） 于小亿 王 辉 张志学

摘 要 详细介绍了在 Windows 环境下应用 VC++实现 PC 机与单片机的几种串行通信方法，给出了用 Visual C++6.0 编写的 PC 机程序和用 C51 编写的单片机通信程序。经实际应用系统运行稳定可靠。

关键词 Visual C++ 类 串行通信

工业控制领域（如 DCS 系统），经常涉及到串行通信问题。为了实现微机和单片机之间的数据交换，人们用各种不同方法实现串行通信，如 DOS 下采用汇编语言或 C 语言，但在 Windows 环境下却存在一些困难和不足。在 Windows 操作系统已经占据统治地位的情况下（何况有些系统根本不支持 DOS 如 Windows2000）开发 Windows 环境下串行通信技术就显得日益重要。

VC ++6.0 是微软公司于 1998 年推出的一种开发环境，以其强大的功能，友好的界面，32 位面向对象的程序设计及 Active X 的灵活性而受广大软件开发者的青睐，被广泛应用于各个领域。应用 VC++开发串行通信目前通常有如下几种方法：一是利用 Windows API 通信函数；二是利用 VC 的标准通信函数_inp、_inpw、_inpd、_outp、_outpw、_outpd 等直接对串口进行操作；三是使用 Microsoft Visual C++的通信控件（MSComm）；四是利用第三方编写的通信类。以上几种方法中第一种使用面较广，但由于比较复杂，专业化程度较高，使用较困难；第二种 需要了解硬件电路结构原理；第三种方法看来较简单，只需要对串口进行简单配置，但是由于使用令人费解的 VARIANT 类，使用也不是很容易；第四种方法是利用一种用于串行通信的 CSerial 类(这种类是由第三方提供)，只要理解这种类的几个成员函数，就能方便的使用。笔者利用 CSerial 类很方便地实现了在固定式 EBM 气溶胶灭火系统分区启动器（单片机系统）与上位机的通信。以下将结合实例，给出实现串行通信的几种 方法。

1 Windows API 通信函数方法

与通信有关的 Windows API 函数共有 26 个，但主要有关的有：

CreateFile() 用 “com*n*”（*n* 为串口号）作为文件名就可以打开串口。

ReadFile() 读串口。

WriteFile() 写串口。

CloseHandle() 关闭串口句柄。初始化时应注意 CreateFile()函数中串口共享方式应设为 0，串口为不可共享设备，其它与一般文件读写类似。以下给出 API 实现的源代码。

1.1 发送的例程

```
//声明全局变量

HANDLE m_hIDComDev;

OVERLAPPED m_OverlappedRead, m_Over lappedWrite;

//初始化串口

void CSerialAPIView::OnInitialUpdate()

{

CView::OnInitialUpdate();

Char szComParams[50];

DCB dcb;

Memset(&m_OverlappedRead, 0, sizeof (OVERLAPPED));

Memset(&m_OverlappedWrite, 0, sizeof (OVERLAPPED));

m_hIDComDev = NULL;

m_hIDComDev = CreateFile(“COM2”, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, NULL);

if (m_hIDComDev == NULL)

{

AfxMessageBox(“Can not open serial port!”);

goto endd;

}
```

```
memset(&m_OverlappedRead, 0, sizeof (OVERLAPPED));

memset(&m_OverlappedWrite, 0, sizeof (OVERLAPPED));

COMMTIMEOUTS CommTimeOuts;

CommTimeOuts. ReadIntervalTimeout=0×FFFFFFFF;

CommTimeOuts. ReadTotalTimeoutMultiplier = 0;

CommTimeOuts. ReadTotalTimeoutConstant = 0;

CommTimeOuts. WriteTotalTimeoutMultiplier = 0;

CommTimeOuts. WriteTotalTimeoutConstant = 5000;

SetCommTimeouts(m_hIDComDev, &CommTimeOuts);

Wsprintf(szComparams, “COM2:9600, n, 8, 1”);

m_OverlappedRead. hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

m_OverlappedWrite. hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

dcb. DCBlength = sizeof(DCB);

GetCommState(m_hIDComDev, &dcb);

dcb. BaudRate = 9600;

dcb. ByteSize= 8;

unsigned char ucSet;

ucSet = (unsigned char) ((FC_RTSCTS&FC_DTRDSR) != 0);

ucSet = (unsigned char) ((FC_RTSCTS&FC_RTSCTS) ! = 0);

ucSet = (unsigned char) ((FC_RTSCTS&FC_XONXOFF) ! = 0);

if (!SetCommState(m_hIDComDev, &dcb)‖

!SetupComm(m_hIDComDev,10000,10000)‖

m_OverlappedRead. hEvent ==NULL‖

m_OverlappedWrite. hEvent ==NULL)
```

```
{
DWORD dwError = GetLastError();
if (m_OverlappedRead. hEvent != NULL) CloseHandle(m_OverlappedRead. hEvent);
if (m_OverlappedWrite. hEvent != NULL) CloseHandle(m_OverlappedWrite. hEvent);
CloseHandle(m_hIDComDev);
}
endd:
;
}
//发送数据
void CSerialAPIView::OnSend()
{
char szMessage[20] = “thank you very much”;
DWORD dwBytesWritten;
for (int i=0; i＜sizeof(szMessage); i++)
{
WriteFile(m_hIDComDev, (LPSTR)&szMessage[i], 1, &dwBytesWritten, &m_OverlappedWrite);
if (WaitForSingleObject(m_OverlapperWrite, hEvent, 1000))dwBytesWritten = 0;
else{
GentOverlappedResult(m_hIDComDev, &m_OverlappedWrite, &dwBytesWritten, FALSE);
m_OverlappedWrite. Offset += dwBytesWritten;
}
dwBytesWritten++;
}
```

```
}
```

1.2 接收例程

```
DCB ComDcb; //设备控制块

HANDLE hCom; //global handle

hCom = CreateFile ("COM1",GENERIC_READ| GENERIC_WRITE,0,

NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);

if (hCom==INVALID_HANDLE_VALUE)

{

AfxMessageBox("无法打开串行口");

}

else

{

COMMTIMEOUTS CommTimeOuts ;

SetCommMask(hCom, EV_RXCHAR ) ;

SetupComm(hCom, 4096, 4096 ) ; /*设置收发缓冲区 尺寸为 4K */

PurgeComm(hCom, PURGE_TXABORT| PURGE_RXABORT |

PURGE_TXCLEAR| PURGE_RXCLEAR ) ; //清收发缓冲区

//以下初始化结构变量 CommTimeOuts, 设置超时参数 CommTimeOuts.ReadIntervalTimeout =
0×FFFFFFFF ;

CommTimeOuts.ReadTotalTimeoutMultiplier = 0 ;

CommTimeOuts.ReadTotalTimeoutConstant = 4000 ;

CommTimeOuts.WriteTotalTimeoutMultiplier = 0;

CommTimeOuts.WriteTotalTimeoutConstant = 4000 ;
```

```
SetCommTimeouts(hCom, &CommTimeOuts ); //设置超时参数

ComDcb.DCBlength = sizeof( DCB ) ;

GetCommState( hCom, &ComDcb ) ; //获取当前参数

ComDcb.BaudRate =9600; //波特率

ComDcb.ByteSize = 8; //数据位

ComDcb.Parity = 0; /*校验 0~4=no, odd, even, mark, space */

SetCommState(hCom, &ComDcb ) ;

} //设置新的通信参数

接收可用定时器或线程等

DWORD dRead,dReadNum;

unsigned char buff [200];

dRead=ReadFile(hCom, buff, 100, &dReadNum, NULL); //接收 100 个字符，

//dReadNum 为实际接收字节数
```

2 利用端口函数直接操作

这种方式主要是采用两个端口函数_inp(), _outp()实现对串口的读写，其中读端口函数的原型为：

```
int _inp(unsigned shot port)
```

该函数从端口读取一个字节，端口号为 0~65535。

写端口的函数原型为：

```
int _outp(unsigned shot port, int databyte)
```

该函数向指定端口写入一个字节。

不同的计算机串口地址可能不一样，通过向串口的控制及收发寄存器进行读写，可以实现灵活的串口通信功能，由于涉及具体的硬件电路讨论比较复杂，在此不加赘述。

3 MSComm 控件

MSComm 控件是微软开发的专用通信控件，封装了串口的所有功能，使用很方便，但在实际应用中要小心对其属性进行配置。下面详细说明该类应用方法。

3.1 MSComm 控件的属性

CommPort：设置串口号，类型 short :1-comm1 2-comm2.

Settings：设置串口通信参数，类型 CString :B 波特率，P 奇偶性（N 无校验，E 偶校验，O 奇校验），D 字节有效位数，S 停止位。

PortOpen：设置或返回串口状态，类型 BOOL：TURE 打开，FALSE 关闭。

InputMode:设置从接收缓冲区读取数据的格式，类型 long： 0-Text 1-Bin。

Input:从接收缓冲区读取数据，类型 VARIANT。

InBufferCount:接收缓冲区中的字节数，类型：short。

InBufferSize:接收缓冲区的大小，类型：short。

Output:向发送缓冲区写入数据，类型：VARIANT。

OutBufferCount:发送缓冲区中的字节数，类型：short。

OutBufferSize:发送缓冲区的大小，类型：short。

InputLen:设置或返回 Input 读出的字节数，类型：short。

CommEvent:串口事件，类型：short。

3.2 程序示例

串口初始化

```
if (!m_comm.GetPortOpen())

m_comm.SetPortOpen(TURE); /*打开串口*/

m_comm.SetSettings("4800,n,8,1"); /*串口参数设置*/

m_comm.SetInputMode(0); /*设置 TEXT 缓冲区输入方式*/

m_comm.SetRthresHold(1); /*每接收一个字符则激发 OnComm()事件*/
```

接收数据

```
m_comm.SetInputLen(1); /*每次读取一个字符

VARINAT V1=m_comm.GetInput();

/*读入字符*/

m_V1=V1.bstrval;
```

发送字符 m_comm.SetOutput(Colevariant ("Hello"); /*发送 “Hello” */

3.3 注意

SetOutput 方法可以传输文本数据或二进制数据。用 SetOutput 方法传输文本数据，必须定义一个包含一个字符串的 Variant。发送二进制数据，必须传递一个包含字节数组的 Variant 到 Output 属性。正常情况下，如果发送一个 ANSI 字符串到应用程序，可以以文本数据的形式发送。如果发送包含嵌入控制字符、Null 字符等的数据，要以二进制形式发送。此处望引起读者注意，笔者曾经在此犯错。

4 VC++类 CSerial

4.1 串行通信类 CSerial 简介

Cserial 是由 MuMega Technologies 公司提供的一个免费的 VC++类，可方便地实现串行通信。以下为该类定义的说明部分。

```
class CSerial

{

public:

CSerial();

~CSerial();

BOOL Open( int nPort = 2, int nBaud = 9600 );

BOOL Close( void );

int ReadData( void *, int );

int SendData( const char *, int );

int ReadDataWaiting( void );
```

```
BOOL IsOpened( void ){ return( m_bOpened ); }

protected:

BOOL WriteCommByte( unsigned char );

HANDLE m_hIDComDev;

OVERLAPPED m_OverlappedRead, m_OverlappedWrite;

BOOL m_bOpened;

}
```

4.2 串行通信类 Cserial 成员函数简介

1. CSerial：：Cserial 是类构造函数，不带参数，负责初始化所有类成员变量。

2. CSerial:: Open 这个成员函数打开通信端口。带两个参数,第一个是埠号，有效值是 1 到 4，第二个参数是波特率，返回一个布尔量。

3. CSerial:: Close 函数关闭通信端口。类析构函数调用这个函数，所以可不用显式调用这个函数。

4. CSerial:: SendData 函数把数据从一个缓冲区写到串行端口。它所带的第一个参数是缓冲区指针，其中包含要被发送的资料；这个函数返回已写到端口的实际字节数。

5. CSerial:: ReadDataWaiting 函数返回等待在通信端口缓冲区中的数据，不带参数。

6. CSerial:: ReadData 函数从端口接收缓冲区读入数据。第一个参数是 void＊缓冲区指针，资料将被放入该缓冲区；第二个参数是个整数值，给出缓冲区的大小。

4.3 应用 VC 类的一个实例

1. 固定式 EBM 气溶胶灭火系统简介

固 定式 EBM 气溶胶灭火装置分区启动器是专为 EBM 灭火装置设计的自动控制设备。可与两线制感温、感烟探测器配套使用，当监测部位发生火情时，探测器发出电 信号给分区启动器，经逻辑判断后发出声、光报警，延时后自动启动 EBM 灭火装置。为了便于火灾事故的事后分析，需对重要的火警事件和关键性操作进行记录， 记录应能从 PC 机读出来；PC 机能控制、协调整个系统的工作，这些都涉及通信。本例中启动器采用 RS-485 通信接口，系统为主从式网络，PC 机为上位 机。具体的通信协议为：（1）下位机定时向上传送记录的事件；（2）应答发送，即 PC 机要得到最新事件记录，而传送时间未到时，PC 机发送命令，下位机接 收命令后，把最新记录传给上位机；（3）上位机发送其它命令如校时、启动、停止、手/自动

等。

2. 通信程序设计

部分上位机程序

（1）发送命令字程序，代码如下

```
void CCommDlg::OnSend()
{
CSerial Serial;
//构造串口类，初始化串行口
if (Serial.Open(2,9600)) //if-1
//打开串行口 2，波特率为 9600bps
{
static char szMessage[]="0";
//命令码(可定义各种命令码)
int nBytesSent;
int count=0;
resend:
nBytesSent=Serial.SendData(szMessage,strlen(szMessage));
//发送命令码
char rdMessage [20];
if (Serial.ReadDataWaiting()) //if-2
{
Serial.ReadData(rdMessage,88);
//rdMessage 定义接收字节存储区，为全局变量//
if ((rdMessage[0]!=0x7f)&&(count<3))
```

```
{
count++;
goto resend
}
if(count>=3)
MessageBox(“发送命令字失败”);
}
else //if-2
MessageBox("接收数据错误");
}
else //if-1
MessageBox("串行口打开失败");
}
```

下位机通信程序：

```
#include<reg51.h>
#include<stdlib.h>
#include<stdio.h>
#define count 9
#define com_code 0x00
#define com_code1 0xff
unsigned char buffer[count];
int po,year,month,date,hour;
int minute,second,recordID ;
int sum;
```

```
main()
{
…
/*初始化串口和定时器*/
TMOD=0×20;
TH1=0×fd;
TR1=0×01;
ET1=0×00;
ES=1;
EA=1;
/*待发送数据送缓冲区*/
buffer[0]=0×ff; //数据特征码
buffer[1]=count+1; //数据长度
buffer[2]=year;  //年
buffer[3]=month; //月
buffer[4]=date; //日
buffer[5]=hour; //时
buffer[6]=minute; //分
buffer[7]=second; //秒
buffer[8]=recordID; //事件号
for(po=0;po<count;po++)
sum+=buffer[po];
buffer[9]=sum; //校验和
…
```

```
}
/*发送中断服务程序*/
void send(void) interrupt 4 using 1
{
int i;
RI=0;
EA=0;
do
{
for(i=0;i<=count;i++)
{
SBUF=buffer[i]; //发送数据和校验和//
while(TI==0);
TI=0;
}
while(RI==0);
RI=0;
}while(SBUF!=0); //主机接收不正确，重新发送//
EA=1;
Return;
}
```

5 应用总结

根据不同需要，选择合适的方法。我们选用的用VC++类实现的上位机和下位机的串行通信方法具有使用简

单、编写程序方便的特点。经过半年多应用于 EBM 灭火系统的情况来看，该方法实现的系统运行稳定可靠，是一种值得推广的简单易行的通信方法。

目次：

在众多网友的支持下，串口调试助手从 2001 年 5 月 21 日发布至今，短短一个月，在全国各地累计下载量近 5000 人次，在近 200 多个电子邮件中，20 多 人提供了使用测试意见，更有 50 多位朋友提出要串口调试助手的源代码，为了答谢谢朋友们的支持，公开推出我最初用 VC 控件 MSComm 编写串口通信程序的 源代码，并写出详细的编程过程，姑且叫串口调试助手源程序 V1.0 或 VC 串口通讯源程序吧，我相信，如果你用 VC 编程，那么有了这个代码，就可以轻而易举地完成串口编程任务了。（也许本文过于详细，高手就不用看）

开始吧：

1.建立项目：打开 VC＋＋6.0，建立一个基于对话框的 MFC 应用程序 SCommTest（与我源代码一致，等会你会方便一点）；

2.在项目中插入 MSComm 控件　　选择 Project 菜单下 Add To Project 子菜单中的 Components and Controls…选项，在弹出的对话框中双击 Registered ActiveX Controls 项（稍等一会，这个过程较慢），则所有注册过的 ActiveX 控件出现在列表框中。 选择 Microsoft Communications Control, version 6.0，，单击 Insert 按钮将它插入到我们的 Project 中来，接受缺省的选项。（如果你在控件列表中看不到 Microsoft Communications Control, version 6.0，那可能是你在安装 VC6 时没有把 ActiveX 一项选上，重新安装 VC6，选上 ActiveX 就可以了），

这时在 ClassView 视窗中就可以看到 CMSComm 类了，（注意：此类在 ClassWizard 中看不到，重构 clw 文件也一样），并且在控件工具栏 Controls 中出现了电话图标（如图 1 所示），现在要做的是用鼠标将此图标拖到对话框中，程序运行后，这个图标 是看不到的。

3.利用 ClassWizard 定义 CMSComm 类控制对象 打开 ClassWizard－>Member Viariables 选项卡，选择 CSCommTestDlg 类，为 IDC_MSCOMM1 添加控制变量：m_ctrlComm，这时你可以看一看，在对 话框头文件中自动加入了 //{{AFX_INCLUDES()　#include "mscomm.h"　//}}AFX_INCLUDES （这时运行程序，如果有错，那就再从头开始）。

4.在对话框中添加控件 向主对话框中添加两个编辑框，一个用于接收显示数据 ID 为 IDC_EDIT_RXDATA，另一个用于输入发送数据，ID 为 IDC_EDIT_TXDATA，再添加一个按钮，功能是按一次就把发送编辑框中的内容发送一次，将其 ID 设为 IDC_BUTTON_MANUALSEND。别忘记了将接收编辑框的 Properties－>Styles 中把 Miltiline 和 Vertical Scroll 属性选上，发送编辑框若你想输入多行文字，也可选上 Miltiline。

再打开 ClassWizard－>Member Viariables 选项卡，选择 CSCommTestDlg 类，为 IDC_EDIT_RXDATA 添加 CString 变量 m_strRXData， 为 IDC_EDIT_TXDATA 添加 CString 变量 m_strTXData。说明： m_strRXData 和 m_strTXData 分别用来放入接收和发送的字符数据。

休息一会吧？我们天天与电脑打交道，要注意保重，我现在在单杠上做引体向上可以来 40 次，可我都 32 了，佩服吗？  。。。。。。好了，再接着来，下面是关键了：

5.添加串口事件消息处理函数 OnComm() 打开 ClassWizard－>Message Maps，选择类 CSCommTestDlg，选择 IDC_MSCOMM1，双击消息 OnComm，将弹出的对话框中将函数名改为 OnComm，（好记而已）OK。

这个函数是用来处理串口消息事件的，如每当串口接收到数据，就会产生一个串口接收数据缓冲区中有字符的消息事件，我们刚才添加的函数就会执行，我们在 OnComm()函数加入相应的处理代码就能实现自己想要的功能了。请你在函数中加入如下代码：

```
void CSCommTestDlg::OnComm()
{
	// TODO: Add your control notification handler code here
	VARIANT variant_inp;
	COleSafeArray safearray_inp;
	LONG len,k;
	BYTE rxdata[2048]; //设置 BYTE 数组 An 8-bit integerthat is not signed.
	CString strtemp;
	if(m_ctrlComm.GetCommEvent()==2) //事件值为 2 表示接收缓冲区内有字符
	{                              ////////以下你可以根据自己的通信协议加入处理代码
		variant_inp=m_ctrlComm.GetInput(); //读缓冲区
		safearray_inp=variant_inp; //VARIANT 型变量转换为 ColeSafeArray 型变量
		len=safearray_inp.GetOneDimSize(); //得到有效数据长度
		for(k=0;k<len;k++)
			safearray_inp.GetElement(&k,rxdata+k);//转换为 BYTE 型数组
		for(k=0;k<len;k++) //将数组转换为 Cstring 型变量
		{
			BYTE bt=*(char*)(rxdata+k); //字符型
			strtemp.Format("%c",bt); //将字符送入临时变量 strtemp 存放
			m_strRXData+=strtemp; //加入接收编辑框对应字符串
		}
	}
	UpdateData(FALSE); //更新编辑框内容
}
```

到目前为止还不能在接收编辑框中看到数据，因为我们还没有打开串口，但运行程序不应该有任何错误，不然，你肯定哪儿没看仔细，因为我是打开 VC6 对照着做一步写一行的，运行试试。没错吧？那么做下一步：

6.打开串口和设置串口参数　你可以在你需要的时候打开串口，例如在程序中做一个开始按钮，在该按钮的处理函数中打开串口。现在我们在主对话框的 CSCommTestDlg::OnInitDialog()打开串口，加入如下代码：

```
// TODO: Add extra initialization here
if(m_ctrlComm.GetPortOpen())
m_ctrlComm.SetPortOpen(FALSE);

m_ctrlComm.SetCommPort(1); //选择 com1
if( !m_ctrlComm.GetPortOpen())
m_ctrlComm.SetPortOpen(TRUE);//打开串口
else
AfxMessageBox("cannot open serial port");

m_ctrlComm.SetSettings("9600,n,8,1"); //波特率 9600，无校验，8 个数据位，1 个停止位

m_ctrlComm.SetInputModel(1); //1：表示以二进制方式检取数据
m_ctrlComm.SetRThreshold(1);
//参数 1 表示每当串口接收缓冲区中有多于或等于 1 个字符时将引发一个接收数据的 OnComm 事件
m_ctrlComm.SetInputLen(0); //设置当前接收区数据长度为 0
m_ctrlComm.GetInput();//先预读缓冲区以清除残留数据
```

现在你可以试试程序了，将串口线接好后(不会接？去看看我写的串口接线基本方法)，打开串口调试助手，并将串口设在 com2，选上自动发送，也可以等会手动发送。再执行你编写的程序，接收框里应该有数据显示了。

7.发送数据　先为发送按钮添加一个单击消息即 BN_CLICKED 处理函数，打开 ClassWizard－>Message Maps，选择类 CSCommTestDlg，选择 IDC_BUTTON_MANUALSEND，双击 BN_CLICKED 添加 OnButtonManualsend()函数，并在函数中添加如下代码：

```
void CSCommTestDlg::OnButtonManualsend()
{
// TODO: Add your control notification handler code here
UpdateData(TRUE); //读取编辑框内容
m_ctrlComm.SetOutput(COleVariant(m_strTXData));//发送数据
}
```

运行程序，在发送编辑框中随意输入点什么，单击发送按钮，啊！看看，在另一端的串口调试助手（或别的调试工具）接收框里出现了什么。

如果你真是初次涉猎串口编程，又一次成功，那该说声谢谢我了，因为我第一次做串口程序时可费劲了，那时网上的资料也不好找。开开玩笑，谢谢你的支持，有什么好东西别忘了给我寄一份。

最后说明一下，由于用到 VC 控件，在没有安装 VC 的计算机上运行时要从 VC 中把 mscomm32.ocx、msvcrt.dll、mfc42.dll 拷到 Windows 目录下的 System 子目录中（win2000 为 System32） 并再进行注册设置，请参考

如何手工注册 MSComm 控件。

龚建伟 2001.6.20

目次：

这是串口调试助手源程序及编程详细过程（一）的续篇，首先谢谢朋友们的支持与鼓励。

8.发送十六进制字符

在主对话框中加入一个复选接钮，ID 为 IDC_CHECK_HEXSEND Caption:十六进制发送，再利用 ClassWizard 为其添加控制变量：m_ctrlHexSend;

在 ClassView 中为 SCommTestDlg 类添加以下两个 PUBLIC 成员函数，并输入相应代码;

```
//由于这个转换函数的格式限制，在发送框中的十六制字符应该每两个字符之间插入一个空隔
//如：A1 23 45 0B 00 29
//CByteArray 是一个动态字节数组，可参看 MSDN 帮助
```

```
int CSCommTestDlg::String2Hex(CString str, CByteArray &senddata)
{
int hexdata,lowhexdata;
int hexdatalen=0;
int len=str.GetLength();
senddata.SetSize(len/2);
for(int i=0;i<len;)
{
char lstr,hstr=str[i];
if(hstr==' ')
{
i++;
continue;
}
i++;
if(i>=len)
break;
lstr=str[i];
hexdata=ConvertHexChar(hstr);
lowhexdata=ConvertHexChar(lstr);
if((hexdata==16)||(lowhexdata==16))
break;
else
hexdata=hexdata*16+lowhexdata;
i++;
senddata[hexdatalen]=(char)hexdata;
hexdatalen++;
}
senddata.SetSize(hexdatalen);
return hexdatalen;
}

//这是一个将字符转换为相应的十六进制值的函数
//好多 C 语言书上都可以找到
//功能：若是在 0-F 之间的字符，则转换为相应的十六进制字符，否则返回-1
char CSCommTestDlg::ConvertHexChar(char ch)
{
if((ch>='0')&&(ch<='9'))
return ch-0x30;
else if((ch>='A')&&(ch<='F'))
return ch-'A'+10;
else if((ch>='a')&&(ch<='f'))
return ch-'a'+10;
```

```
else return (-1);
}
```

再将 CSCommTestDlg::OnButtonManualsend()修改成以下形式：

```
void CSCommTestDlg::OnButtonManualsend()
{
// TODO: Add your control notification handler code here
UpdateData(TRUE); //读取编辑框内容
if(m_ctrlHexSend.GetCheck())
{
CByteArray hexdata;
int len=String2Hex(m_strTXData,hexdata); //此处返回的 len 可以用于计算发送了多少个十六进制数
m_ctrlComm.SetOutput(COleVariant(hexdata)); //发送十六进制数据
}
else
m_ctrlComm.SetOutput(COleVariant(m_strTXData));//发送 ASCII 字符数据

}
```

现在，你先将串口线接好并打开串口调试助手 V2.1，选上以十六制显示，设置好相应串口，然后运行我们这个程序，在发送框中输入 00 01 02 03 A1 CC 等十六进制字符，并选上以十六进制发送，单击手动发送，在串口调试助手的接收框中应该可以看到 00 01 02 03 A1 CC 了。

9.在接收框中以十六进制显示

这就容易多了： 在主对话框中加入一个复选接钮，IDC_CHECK_HEXDISPLAY Caption: 十六进制显示，再利用 ClassWizard 为其添加控制变量：m_ctrlHexDiaplay。 然后修改 CSCommTestDlg::OnComm()函数：

```
void CSCommTestDlg::OnComm()
{
// TODO: Add your control notification handler code here
VARIANT variant_inp;
COleSafeArray safearray_inp;
LONG len,k;
BYTE rxdata[2048]; //设置 BYTE 数组 An 8-bit integerthat is not signed.
CString strtemp;
if(m_ctrlComm.GetCommEvent()==2) //事件值为 2 表示接收缓冲区内有字符
{
variant_inp=m_ctrlComm.GetInput(); //读缓冲区
safearray_inp=variant_inp; //VARIANT 型变量转换为 ColeSafeArray 型变量
```

```
len=safearray_inp.GetOneDimSize(); //得到有效数据长度
for(k=0;k<len;k++)
safearray_inp.GetElement(&k,rxdata+k);//转换为 BYTE 型数组
for(k=0;k<len;k++) //将数组转换为 Cstring 型变量
{
BYTE bt=*(char*)(rxdata+k); //字符型
if(m_ctrlHexDisplay.GetCheck())
strtemp.Format("%02X ",bt); //将字符以十六进制方式送入临时变量 strtemp 存放，注意这里加入一个空隔
else
strtemp.Format("%c",bt); //将字符送入临时变量 strtemp 存放

m_strRXData+=strtemp; //加入接收编辑框对应字符串
}
}
UpdateData(FALSE); //更新编辑框内容
}
```

测试：在串口调试助手发送框中输入 00 01 02 03 A1 CC 等十六进制字符，并选上以十六进制发送，单击手动发送，在本程序运行后选上以十六进制显示，在串口调试助手中单击手动发送或自动发送，则在本程序的接收框中应该可以看到 00 01 02 03 A1 CC 了。

10.如何设置自动发送

最简单的设定自动发送周期是用 SetTimer()函数，这在数据采集中很有用，在控制中指令的传送也可能用到定时发送。

方法是：在 ClassWizard 中选上 MessageMap 卡，然后在 Objects IDs 选中 CSCommTestDlg 类，再在 Messages 框中选上 WM_TIMER 消息，单击 ADD_FUNCTION 加入 void CSCommTestDlg::OnTimer(UINT nIDEvent) 函数，这个函数是放入“时间到”后要处理的代码：

```
void CSCommTestDlg::OnTimer(UINT nIDEvent)
{
// TODO: Add your message handler code here and/or call default
OnButtonManualsend();
CDialog::OnTimer(nIDEvent);
}
```

再在在主对话框中加入一个复选接钮，ID 为 IDC_CHECK_AUTOSEND Caption: 自动发送(周期 1 秒)，再利用 ClassWizard 为其添加 BN_CLICK 消息处理函数 void CSCommTestDlg::OnCheckAutosend()：

```
void CSCommTestDlg::OnCheckAutosend()
{
```

```
// TODO: Add your control notification handler code here
m_bAutoSend=!m_bAutoSend;
if(m_bAutoSend)
{
SetTimer(1,1000,NULL);//时间为 1000 毫秒
}
else
{
KillTimer(1);    //取消定时
}
}
```

其中：m_bAutoSend 为 BOOL 型变量，在 CLASSVIEW 中为 CSCommTestDlg 类加入，并在构造函数中初始化：

m_bAutoSen=FALSE;

现在可以运行程序测试了。

11.什么是 VARIANT 数据类型？如何使用 VARIANT 数据类型？

不知如何使用 VARIANT 数据类型， 有不少朋友对 VARIANT 这个新的数据类型大感头疼。SetOutput()函数中 需要的 VARIANT 参数还可以使用 COleVariant 类的构造函数简单生成，现在 GetInput()函数的返回值也成了 VARIANT 类型，那么 如何从返回的值中提取有用的内容。 VARIANT 及由之而派生出的 COleVariant 类主要用于在 OLE 自动化中传递数据。实际上 VARIANT 也只不过是一个新定义的结构罢了，它的 主要成员包括一个联合体及一个变量。该联合体由各种类型的数据成员构成， 而该变量则用来指明联合体中目前起作用的数据类型。我们所关心的接收到的数据就存储在该联合体的某个数据成员中。 该联合体中包含的数据类型很多，从一些简单的变量到非常复杂的数组和指针。由于通过串口接收到的内容常常是一个字节串，我们将使用其中的某个数组或指针来 访问接收到的数据。这里推荐给大家的是指向一个 SAFEARRAY（COleSafeArray)类型变量。新的数据类型 SAFEARRAY 正如其名字一 样，是一个“安全数组”，它能根据系统环境自动调整其 16 位或 32 位的定义，并且不会被 OLE 改变（某些类型如 BSTR 在 16 位或 32 位应用程序间传递时会被 OLE 翻译从而破坏其中的二进制数据）。大家无须了解 SAFEARRAY 的具体定义，只要知道它是另外一个结构，其中包含一个 (void *)类型的指针 pvData，其指向的内存就是存放有用数据的地方。 简而言之，从 GetInput()函数返回的 VARIANT 类型变量中，找出 parray 指针，再从该指针指向的 SAFEARRAY 变量中找出 pvData 指针，就可以向访问数组一样取得所接收到的数据了。具体应用请参见 void CSCommTestDlg::OnComm()函数。

大概我现在也说不清这个问题，我自己从第一次接触这个东西，到现在还是给别人讲不清。

在 VC++中利用 ActiveX 控件开发串行通信程序

上海交通大学 B9703-1 信箱(200030) 黄海荣 田作华

摘　要：探讨在使用 Visual C++编程时利用 Microsoft Communications Control 控件编写串行通信程序的方法,并给出了例程,具有一定的实用意义。

关键词：Visual C++ 串行通信 ActiveX

在开发微机控制系统的过程中，我们经常需要通过 RS-232 串行接口与外部设备进行通信。例如分级控制系统中上位机与下位机的数据交换以及数据采集系统中 计算机与数字仪表的通信等。在 DOS 时代，编写串行通信程序是一件相当复杂的工作，程序员需要具备相当的硬件知识，对可编程串行通信接口芯片的内部寄存器 定义、工作方式、指令字等相关内容有所了解，才有可能着手编写程序，大量的时间和精力都花在了如何与硬件打交道上，而不是花在我们的主要目的——获取与处 理数据上；在 Windows 下，Win32API 提供了使用 CreateFile/WriteFile 等文件 I/O 函数进行串行口操作的方法，但是在实现 上仍然是相当烦琐的。幸运的是，Windows 平台先进的 ActiveX 技术使我们在对串行口编程时不再需要处理烦琐的细节。利用已有的 ActiveX 控 件，我们只需要编写少量的代码，就可以轻松高效地完成任务。本文以 Windows 98 下用 Visual C++6.0 开发 PT650C 秤重显示器的通信模块为例，探讨了使用 Microsoft Communications Control 控件进行串行通信的方法。

1 ActiveX 控件介绍

ActiveX 是 Windows 下进行应用程序开发的崭新技术，它的核心内容是组件对象模型 COM(Component Object Model)。ActiveX 控件包括一系列的属性、方法和事件，使用 ActiveX 控件的应用程序和 ActiveX 控件之间的工作方式是客户 / 服务器方式，即应用程序通过 ActiveX 控件提供的接口来访问 ActiveX 控件的功能。

Microsoft Communications Control（以下简称 MSComm）是 Microsoft 公司提供的简化 Windows 下串行通信编程的 ActiveX 控件，它为应用程序提供了通过 串行接口收发数据的简便方法。具体的来说，它提供了两种处理通信问题的方法：一是事件驱动(Event－driven)方法，一是查询法。

1.1 事件驱动法

在使用事件驱动法设计程序时，每当有新字符到达，或端口状态改变，或发生错误时，MSComm 控件将解发 OnComm 事件，而应用程序在捕获该事件后，通 过检查 MSComm 控件的 CommEvent 属性可以获知所发生的事件或错误，从而采取相应的操作。这种方法的优点是程序响应及时，可靠性高。

1.2 查询法

这种方法适合于较小的应用程序。在这种情况下，每当应用程序执行完某一串行口操作后，将不断检查 MSComm 控件的 CommEvent 属性以检查执行结果 或者检查某一事件是否发生。例如，当程序向串行设备发送了某个命令后，可能只是在等待收到一个特定的响应字符串，而不是对收到的每一个字符都立刻响应并处 理。

MSComm 控件有许多重要的属性，其中首要的几个如表 1 所示。

表 1

属性	说明
CommPort	设置/获取控件对应的串行口
Settings	设置/获取波特率、校验方式、数据位、停止位
PortOpen	打开/关闭通信口
Input	读取数据
Output	发送数据

2 编程实现　　在使用 MSComm 控件开发 PT650C 秤重显示器通信程序时，采用了事件驱动法，主要是在 comEvReceive(接收到数据)事件发生时响应并获取缓冲区中的数据。以下具体介绍实现方法。　　打开 Visual C++6.0 集成开发环境，创建一个基于对话框的 MFC 应用程序项目，命名为 MyCOM，记住在设置项目选项时必须选上 ActiveX Controls，其他的按照缺省设置。完成这一步后，选择菜单项 Project / Add to Project / Components and Controls……，将弹出一个对话框以选择系统中已有的组件(Components)和控件(Controls)。选择 Registered ActiveX Controls 文件夹下的 Microsoft Communications Control 项并按下 Insert 按钮，将 MSComm 控件支持加入到本项目中。这时将生成一个名为 CMSComm 的 C++类，并且在对话框编辑器里的 工具栏将出现 MSComm 控件图标。CMSComm 类是由 MSComm 控件导出的一系列接口函数构成的，利用它将可以访问 MSComm 控件的属性 (Property)和方法(Method)。　　假设 PT650C 秤重显示器接在计算机 COM1 口上，那么打开资源编辑器，在程序主对话框(资源 ID 为 IDD_MYCOM_DIALOG)上面放置一个 MSComm 控件，并用 Class Wizard 为该对话框类添加对应该控件的成员变量 m_wnd COM 1 。由于 PT650C 秤重显示器与计算机进行串行通信时采用 7 个数据位、1 个停止位、偶校验方式，并且波特率为 2400/4800/9600 可选， 这里我采用 9600 波特率，在对话框编辑器中设置 MSComm 控件的属性如下：

ID:IDC_COM1(资源 ID)

CommPort:1 (COM1)

Settings:9600,e,7,1(波特率 9600，偶校验，7 个数据位，1 个停止位)

RThreshold:（每接收到 1 个字符就触发一个接收数据事件）

SThreshold:0（不触发发送缓冲区空事件）

InputLen:1（每次读操作从缓冲区中取一个字符）

其他选项按照缺省设置或者根据具体设备的要求进行设置。如果需要通过多个串行口与多台设备通信，那么每一个串行口对应于一个单独的 MSComm 控件。串行 口的设置参数既可以在对话框编辑器里设定，也可以在程序代码中通过调用 CMSComm 类的成员函数设定。例如，我们可以在 MyCOMDlg 类的 OnInitDialog 成员函数中初始化 MSComm 控件的参数，代码如下：

```
BOOL CMyCOMDlg::OnInitDialog()
{
CDialog:OnInitDialog();
//以上为 MFC 框架自动生成的代码，在此不列出
//TODO:Add extra initialization here
m_wndCOM1.SetCommPort(1);
m_wndCOM1.SetSettings("9600,e,7,1");
m_wndCOM1.SetRThreshold(1);
m_wndCOM1.SetSThreshold(0);
m_wndCOM1.SetInputLen(1);
m_wndCOM1.SetPortOpen(TRUE);//打开通信口
return TRUE;//return TRUE unless you set the focus to a control
}
```

接下来为程序主对话框建立响应MSComm事件的处理函数，每当MSComm控件触发事件时该函数将被调用。在对话框编辑器中用鼠标左键双击MSComm 控件图标，在弹出的对话框中输入函数名OnCommCOM1，该事件处理函数的原型定义和消息映射入口将自动被添加到CMyCOMDlg类中，我们所要做 的只是在OnCommCOM1函数中给出具体的数据处理程序段，代码示例如下：

```
void CMyCOMDlg::OnCommCom1()
{
    //TOD O:Add your control notification handler code here
    CString sInput;
    switch(m_wndCOM1.GetCommEvent())
    {
    case 1: //comEvSend 事件
            /*如有数据要发送，可采用以下代码：
            VARIANT varOut;
            VariantInit(&varOut);
            varOut.vt=VT_BSTR;
            USES_CONVERSION;
            varOut.bstrVal=SysAllocString(T2OLE)("My data"));
            if(varOut.bstrVal){
                m_wndCOM1.SetOutput(varOut);
                SysFreeString(varOut.bstrVal);
            }
            */
            break;
    case 2: //comEvReceiv 事件，有数据到达
            sInput=m_wndCOM1.GetInput().bstrVal;
            //对接收到的数据做必要处理
            break;
    case 1009://comEventRxParity 事件，奇偶校验错误
            //错误处理代码
            break;
    default:
            break;
    }
```

在这里必须注意的一点是在发送字符数据时，必须向MSComm控件提供Unicode格式的字符串，在以上代码中用到了USES_CONVERSION和T2OLE宏进行ANSI字符串到Unicode字符串的转换，具体内容可参考Visual C++6.0所带的MSDN文档，在此不加赘述。 本文对Windows 98下Visual C++ 程序中使用MSComm串行通信ActiveX控件编程的方法做了探讨，显示了ActiveX技术的强大功能、充分的灵活性和易用性，具有一定的实践意义。 **参考文献** 1 Microsoft 公司.Microsoft Development Network. 2 Kate Gregory.Special Edition Using Visual C++5.

如何在串口通讯程序中处理数据包

龚建伟 2001.10.30

在串口通讯程序中，经常要收到数据包，常有网友问及如何从这些数据包中提取需要的数据，如何处理校验等，在这篇文章里我举两个例子予以说明，程序说明为 VC++6.0。关于串口编程建立程序的细节，请参阅我主页上的其它文章。同时，此文也适于其它通讯程序中艰数据报文的处理。

首先，应该指出的是，所有这些处理均在串口事件处理函数 oncommunication()中进行。每当串口缓冲区中有一个或一个以上字符时触发串口通讯 事件，该事件就驱动（调用）串口事件通讯处理函数 oncommunication()，在这里就可以对接收到的数据进行处理，提取需要的数据。

举两个例子，一个是较为简单的位数据格式的处理，另一个是 NMEA 无线通讯格式的处理，最后回答一位网友提出的问题，大家也可以探讨一下。

1. 问题：

一个数据包，其串头为一个字符，字符值为 7EH（16 进制）’~’，其后紧跟一字符‘E’，然后是数据串，串尾也为字符值为 7EH 的一个字符：
即 ~Exxxxxx...~ 如何处理这些数据？
我们仍以串口调试助手源程序及其详细编程过程之一 中的 OnComm()处理为例：

```
void CSCommTestDlg::OnComm()
{
// TODO: Add your control notification handler code here
VARIANT variant_inp;
COleSafeArray safearray_inp;
LONG len,k;
BYTE rxdata[2048]; //设置 BYTE 数组 An 8-bit integerthat is not signed.
CString strtemp;
if(m_ctrlComm.GetCommEvent()==2) //事件值为 2 表示接收缓冲区内有字符
{ ///////以下你可以根据自己的通信协议加入处理代码
variant_inp=m_ctrlComm.GetInput(); //读缓冲区
safearray_inp=variant_inp; //VARIANT 型变量转换为 ColeSafeArray 型变量
len=safearray_inp.GetOneDimSize(); //得到有效数据长度
for(k=0;k<len;k++)
safearray_inp.GetElement(&k,rxdata+k);//转换为 BYTE 型数组
```

```
for(k=0;k<len;k++) //将数组转换为 Cstring 型变量
{
BYTE bt=*(char*)(rxdata+k); //字符型
strtemp.Format("%c",bt); //将字符送入临时变量 strtemp 存放
m_strRXData+=strtemp; //加入接收编辑框对应字符串,在这儿，编辑框不是必须的，可做相应处理
char ch=(char)bt;
if(ch=='E')
{
//在此处设置一个可以接收数据的全局标志，说明接收到数据前的‘E’标志了，下一步可以读数据了，同时将 m_strRXData 清空
                flag=2;
                  m_strRXData.Empty(); //下一次接收的便为有用的数据
}
if(ch==0x7e)
{
           flag=1; //下面可以提取数据了
}
if(flag==1) //标志为 1，
{
         ...//提取数据
               flag=0; //提取完后，置标志为 0
}

}
}
//UpdateData(FALSE); //更新编辑框内容
}
```

2 NMEA 无线通讯格式的处理

2.1 NMEA-0183 报文格式

字符串（ASCII 字符）格式如下：
$XXXX,XX,XX,XX,……＊hh<CR><LF>
$：串头
XXXX: 串头
XX：数据字段，字母或数字
XX：数据字段，字母或数字
XX：数据字段，字母或数字
，：逗号
……
＊：星号，串尾
hh：$与＊之间所有字符代码的校验和，（注意：校验和 h 为半 Byte 校验，*后

第 1 个 h 表示高 4 位校验和，第 2 个 h 表示低 4 位校验和。得到校验值后，再转换成 ASCII 字符。）
<CR>：0DH,回车控制符
<LF>：0AH,换行控制符

2.2 校验处理

由于数据是动态接收，所以数据的处理也是动态进行，尽管有时会收到几个字符才触发一个串口事件，但字符的接收是一个一个接收的，因此就可以在程序中先判断 串头$是否到达，若串头到达，就可以开始计算校验，直至串尾＊到达，这时＊号后面的两个字符就是校验码，收到这两个校验字符，就可以与自己计算的校验值比 较，若不正确，就报错，并继续处理下面的数据，若正确，则处理接收的字符，提取需要的数据。

2.3 程序

```
CString m_strReceived;
CString m_strChecksum;
int flag;
char ch 为每次收到的字符


m_strReceived += (char)ch;
switch(ch)
{
case '$':
checksum=0; //开始计算 CheckSum
flag=0;
break;
case '*':
flag=2;
c2=checksum & 0x0f; c1=((checksum >> 4) & 0x0f);
if (c1 < 10) c1+= '0'; else c1 += 'A' - 10;
if (c2 < 10) c2+= '0'; else c2 += 'A' - 10;
break;
case CR:
break;
case LF:
m_strReceived[port-1].Empty();
break;
default:
if(flag>0)
{
m_strChecksum += ch;
```

```
if(flag==1)
{
strCheck=strCheck+c1+c2;
if(strCheck!=m_strChecksum)
{
m_strReceived.Empty();
}
else
{
strInstruction=m_strReceived[port-1].Left(6);
if(strInstruction=="$QGOKU") //如果串头正确
{
char *temp=(char*)((LPCTSTR)m_strReceived);//转换

int speed=(atoi(temp+7));// 提取 int 型数据
char splevel=*(temp+25); //提取 char 型数据

}

}
m_strChecksum.Empty();
}
flag--;
}
else
checksum=checksum^ch;
break;
}
```

3 网友的问题

另外，我回答了一位网友的问题，大家也可以探讨一下：
问题如下 3：

我用你的串口程序收来的十六进制数据是这个样的:
00 10 10 C0 00 F0 F0 AB AC AD
我现在要将高四位取出来,也就是
011C0FFAAA(这点我不会,但我用 Left 实现了,可得到的是字符,不是我要的数值)
我只要 011C0FF.
我要把 011C0FF 进行如下的处理
011 转化成十进制
C 不变

OFF 也变成十进制
后显示,成 17 C 255

答：右移得到 011COFF 后，可将其放在一个字符型变量 CString m_strReceive 中：
然后将其转换：

```
char *temp=(char*)((LPCTSTR)m_strReceive;

char tbuf[6]; //temporary viable
tbuf[0]=temp[1]; tbuf[1]=temp[2]; tbuf[2]=temp[3]; tbuf[3]=0; //011 最后为 0 表示结束
int data1=atoi(tbuf);
char chdata2==temp[4]; //C
tbuf[0]=temp[5]; tbuf[1]=temp[6]; tbuf[2]=temp[7]; tbuf[3]=0;
int data3=atoi(tbuf); //OFF
```

以上 data1,chdata2,data3 即为你要的数据