WIN32 串口通信

微软技术文档

摘要

学习 Win32 中的串口通信和 16 位 Windows 操作系统中的串口通信有很大的不同。这篇文章假设读者已经熟悉 Win32 下多线程和同步的基本原理。

Zcube

zcube@qq.com

目录

WIN32 中的串行通信

艾伦 戴夫
微软 Windows 开发者支持中心

1995/12/11

应用于：

- Microsoft® Win32®
- Microsoft Windows®

摘要：学习 Win32 中的串口通信和 16 位 Windows 操作系统中的串口通信有很大的不同。这篇文章假设读者已经熟悉 Win32 下多线程和同步的基本原理。另外，如果对 Win32 中的 **heap** 功能如果有基础的了解，将使读者在完全理解这篇文章中提到的多线程 TTY（MTTTY）例子的内存管理方法上是很有用的。

综述

Win32 中的串口通信和 16 位 Windows 中的串口通信有显著的不同。那些熟悉 16 位串口通信函数的开发人员将不得不重新学习许多系统部分的知识，以便能编写正确的串口通信程序。这篇文章将帮助实现这个目标。那些不熟悉串口通信的人员将发现这篇文章会为他们以后研究发展奠定坚实的基础。

这篇文章假设读者已经熟悉 Win32 下多线程和同步的基本原理。另外，如果对 Win32 中的 **heap** 功能如果有基础的了解，将使读者在完全理解这篇文章中提到的 MTTTY 例子的内存管理方法上是很有用的。

关于这些函数的更多信息，请查阅平台 SDK 文档：微软 Win32 知识库或微软开发者联机文库。虽然那些控制用户界面特性的应用程序接口（APIs）和对话框在这里并不讨论，但是对完全理解这篇文章所提供的例程还是很有用的。不熟悉一般的 Windows 编程的读者在开始处理串行通信前首先应该学习一些 Windows 编程基础。换句话说，冒失地潜水前先沾湿你的脚。

引言

这篇文章主要介绍应用程序接口（APIs）和微软 Windows NT 以及 Windows 95 所兼容的方法。因此，只讨论在 NT 和 95 这两个平台上都被支持的 APIs。Windows 95 支持 Win32 电话 API（TAPI），但

是 Windows NT 3.x 却不支持。因此，这里不对 TAPI 进行讨论。然而，TAPI 值得一提的时，它在调制解调器的连接和调用控制上是非常好的工具。如果一个应用程序产品涉及调制解调器工作和电话拨号，那么利用 TAPI 接口可以实现这些功能。它允许和用户可能有的 TAPI 程序实现无缝结合。此外，这篇文章也不讨论 Win32 中的一些配置函数，像 **GetCommProperties**。

这篇文章包含的例子，MTTTY：多线程 TTY（4918.exe），实现了许多这里所要讨论的功能。在它的实现中使用了 3 个线程：一个用户界面线程实现内存管理、一个写线程实现控制所有的写操作、还有一个读/状态线程实现读数据和处理端口上发生改变的状态。该例子采用一些不同的数据堆实现内存管理。它也广泛使用同步方法促进线程之间的通信。

打开串口

使用 **CreateFile** 函数可以打开一个通信端口。调用 **CreateFile** 打开通信端口有两种方式：重叠的和非重叠的。下面是使用重叠方式打开一个通信资源的例子：

```
HANDLE hComm;
hComm = CreateFile(gszPort,
                   GENERIC_READ | GENERIC_WRITE,
                   0,
                   OPEN_EXISTING,
                   FILE_FLAG_OVERLAPPED,
                   0);
if (hComm == INVALID_HANDLE_VALUE)
    // 打开错误；使中止。
```

移除 **CreateFile** 中的 FILE_FLAG_OVERLAPPED 标志可指定为非重叠方式。下一章节将对重叠和非重叠方式进行讨论

在 Win32 软件开发工具包（SDK）程序员参考手册（概述，窗口管理，系统服务）中规定，当打开一个通信端口时候，调用 **CreateFile** 有如下要求：

- *fdwShareMode* 必须为 0。通信端口不能像文件一样被共享。应用程序使用 TAPI 可以使用 TAPI 函数很容易实现两个应用程序之间的资源共享。对于 Win32 应用程序，不是使用 TAPI。处理继承或副本需要共享通信端口。处理副本超出了本文的范围，请查阅 Win32 SDK 文档获取更多信息。
- *fdwCreate* 必须指定为 OPEN_EXISTING 标志。
- *hTemplateFile* 必须是 NULL。

需要注意一件事，惯例上它们有四个端口分别为：COM1、COM2、COM3 和 COM4。Win32 API 没有提供任何途径去确定系统中存在的端口。Windows NT 和 Windows 95 在配置串口方面互相并不相同，所以任何一种方法都不能确保对所有的 Win 32 平台都是可移植的。一些系统甚至有比惯例上的最大数量四个端口还要多的端口。硬件厂商和串行设备驱动的作者可以用他们所喜欢的方式去自由命名端口。

为此，如果用户可以去指定他们想用的端口名是最好的选择。如果一个端口不存在，在企图打开这个端口的时候一个错误（ERROR_FILE_NOT_FOUND）将会出现，应该警告用户这个端口是不可用的。

读和写

从通信端口读和写在 Win32 中极其类似于 Win32 中文件的输入/输出（I/O）。实际上，实现文件 I/O 的函数和用于串行 I/O 的函数是相同的。Win32 中的 I/O 可以通过两种方式被使用：重叠和非重叠。在 Win32 SDK 文档中用异步和同步这样的术语去暗示这些 I/O 操作的类型。然而，这篇文章中将用重叠和非重叠这样的术语。

非重叠 I/O 对于大多数开发者来说是熟悉的，因为这属于传统的 I/O 操作形式，当函数返回的时候一个被请求的操作将被假设为已完成。就重叠 I/O 来说，即使操作没有完成系统也可以立刻返回给调用者，当操作完成的时候将会用信号通知调用者。程序可以利用 I/O 请求和结束这段时间去执行一些“后台”工作。

在 Win32 中和 16 位 Windows 中对串行通信端口的读和写有显著的不同。16 位 Windows 只有 **ReadComm** 和 **WriteComm** 函数。Win32 中的读写操作可能牵涉更多的函数和选择。这些问题在下文将会被讨论。

非重叠 I/O

非重叠 I/O 非常简单，虽然它还有一些限制。一个操作的执行将导致调用它的线程被阻塞。一旦该操作完成，函数返回，线程继续工作。这种类型的 I/O 对于多线程应用程序来说是非常有用的，因为当 I/O 操作的时候即使一个线程被阻塞，其它线程仍然可以执行工作。应用程序有责任正确无误的处理连续的端口操作。如果一个线程被阻塞去等待 I/O 操作的完成，随后的其它线程如果调用一个通信 API 也将可能被阻塞，直到最初的操作完成。例如，如果一个线程正在等待 **ReadFile** 函数返回，其它线程如果调用 **WriteFile** 函数将会被阻塞。

在非重叠和重叠操作之间做出选择的诸多因素中，其中之一是要考虑到可移植性。重叠操作不是一个好的选择，因为大多数操作系统并不支持它。然而大多数操作系统支持多线程，所以多线程的非重叠 I/O 操作从可移植性上面考虑的话是最好的选择。

重叠 I/O

重叠 I/O 不像非重叠 I/O 那样简单的，但是提供了更多的灵活性和效率。当一个端口打开的时候，对于重叠操作来说，允许线程与此同时执行 I/O 操作和其它的工作，即使这个操作正处于不确定状态。此外，重叠操作允许单线程发出许多不同的请求和执行后台工作，即使这个操作处于不确定状态。

对于单线程和多线程应用程序，在发出请求和得到结果之间必须产生一些同步性。一个线程将会被阻塞，直到一个操作的结果变为有效的。重叠 I/O 的优势所在是允许一个线程在请求和完成之间去做一

些工作。如果没有工作可以被做，然后对于重叠 I/O 只有一种可能，就是它将允许为更好的用户提供响应。

重叠I/O是MTTTY例子中所使用的一种操作类型。它创建一个线程来负责读取端口的数据和状态。它也执行定期的后台工作。程序创建另外一个线程专门用来从端口写出数据。

注意：有时应用程序创建太多的线程，滥用多线程操作系统。虽然利用多线程可以解决很多困难的问题，但是创建过多的线程在应用程序中并不是最有效的方式。在系统中线程没有进程紧张，但是仍然会占用系统资源，像 CPU 时间和内存。如果一个应用程序创建过多的线程，可能对整个系统的性能产生不利的影响。线程的一个更好的使用方式是对每个工作类型创建一个不同的请求队列，有一个工作者线程通过发出一个 I/O 请求使其进入请求队列。上述方法将会被这篇文章中所提到的 MTTTY 这个例子用到。

一个重叠 I/O 操作包含两部分：创建操作和检测是否完成。创建操作必须建立一个 **OVERLAPPED** 结构体，为同步创建一个手工重置事件，然后在调用特定的函数（**ReadFile** 或 **WriteFile**）。I/O 操作可能也可能不会立即的完成。对于一个程序来说，如果假定一个重叠操作请求总是产生一个重叠操作是错误的。如果一个操作完成后，应用程序需要准备继续正常地运行。重叠操作的第二部分是检测它是否完成。检测操作是否完成包含等待事件处理，检查重叠结果和处理数据。有很多工作牵涉到重叠操作的原因是存在很多故障点。如果一个非重叠操作失败了，函数只是会返回一个错误返回的结果。如果一个重叠操作失败了，它可能在创建操作时候失败或是使操作处于等待状态。你也可能有一个超时操作或只是一个超时去等待操作完成的信号。

读

ReadFile 函数将产生一个读的操作。**ReadFileEx** 也产生一个读操作，但是因为它在 Windows 95 上是不可用的，所以在这篇文章中不对它做讨论。这里的代码段详细说明了怎样产生一个读操作。注意，如果 **ReadFile** 函数返回 TRUE，它的功能是将调用一个函数去处理数据。如果操作变成了重叠方式，这个处理数据的函数也是一样的。注意在代码段中定义的 fWaitingOnRead 标记变量，用它来指明一个读操作是否为重叠的。它通常用来防止在一个读操作尚未完成的时候又重新创建一个新的读操作。

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

// 创建重叠事件。必须关闭以前存在的事件以避免句柄泄露。
osReader.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

if (osReader.hEvent == NULL)
      // 创建重叠事件错误；使中止。

if (!fWaitingOnRead) {
      // 执行读操作
      if (!ReadFile(hComm, lpBuf, READ_BUF_SIZE, &dwRead, &osReader)) {
            // 读操作是否处于等待状态?
```

```
            if (GetLastError() != ERROR_IO_PENDING)
                // 通信错误；报告该错误。
            else
                fWaitingOnRead = TRUE;
    }
    else {
        // 完成读操作
        HandleASuccessfulRead(lpBuf, dwRead);
    }
}
```

重叠操作的第二部分是检测它是否完成。**OVERLAPPED** 结构体中的事件句柄会被传递到 **WaitForSingleObject** 函数中进行等待，直到对象被传递信号。一旦事件被传递信号，则代表操作完成了。这并不意味着操作是被成功地完成，仅仅是完成而已。**GetOverlappedResult** 函数将报告操作的结果。如果发生了错误，**GetOverlappedResult** 函数将返回 FALSE，**GetLastError** 函数将返回错误代码。如果操作被成功的完成，**GetOverlappedResult** 将返回 TRUE。

注意：**GetOverlappedResult** 函数可以探测操作是否完成，也可以返回操作的失败状态。如果操作没有完成，**GetOverlappedResult** 将返回 FALSE 且 **GetLastError** 函数将返回 ERROR_IO_INCOMPLETE。此外，**GetOverlappedResult** 可能会被阻塞直到操作完成。实际上，重叠操作转变成非重叠操作可以通过给 **GetOverlappedResult** 函数中的 bWait 参数传递 TRUE 来实现。

这里的代码段展示了一种检查一个重叠读操作是否完成的方法。注意下面的代码也调用的处理数据函数和上面是相同的，该函数在操作完成后立即调用。也要注意使用的 fWaitingOnRead 标记，在这里它用来控制是否执行检测代码，因为它只有在一个读操作尚未完成时候才应该被调用。

```
#define READ_TIMEOUT 500  // 毫秒

DWORD dwRes;

if (fWaitingOnRead) {

    dwRes = WaitForSingleObject(osReader.hEvent, READ_TIMEOUT);

    switch(dwRes) {
        // 读操作完成。
    case WAIT_OBJECT_0:
        if (!GetOverlappedResult(hComm, &osReader, &dwRead, FALSE))
            // 通信错误；报告该错误。
        else
            // 读操作成功地完成。
            HandleASuccessfulRead(lpBuf, dwRead);

        // 重置标记变量，以便其它操作可以被执行。
        fWaitingOnRead = FALSE;
        break;
```

```
        case WAIT_TIMEOUT:
            // 操作还没有完成。fWaitingOnRead 标记变量没有改变，因为我将
            // 继续返回循环，我不想在第一个读操作没完成就去执行另外一个。
            //
            // 在这里比较适合做一些后台工作。
            break;

        default:
            // WaitForSingleObject 函数发生错误；使中止。
            // 这表明了 OVERLAPPED 结构体中的事件句柄发生的一个问题。
            break;
        }
}
```

写

从通信端口向外传输数据和读操作非常相似，而且他们共用很多相同 API 函数。下面的代码段展示了怎样执行和等待写操作的完成。

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

    // 创建写操作，并对相应的 OVERLAPPED 结构体的 hEvent 成员赋值。
    osWrite.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    if (osWrite.hEvent == NULL)
        // 创建重叠事件句柄发生错误。
        return FALSE;

    // 执行写操作。
    if (!WriteFile(hComm, lpBuf, dwToWrite, &dwWritten, &osWrite)) {

        if (GetLastError() != ERROR_IO_PENDING) {
            // WriteFile 失败但不延时。报告错误并使中止。
            fRes = FALSE;
        }
        else {
            // 写等待。
            dwRes = WaitForSingleObject(osWrite.hEvent, INFINITE);

            switch(dwRes) {
                // OVERLAPPED 结构体的事件已经处于有信号状态。
                case WAIT_OBJECT_0:
                    if (!GetOverlappedResult(hComm, &osWrite, &dwWritten, FALSE))
                        fRes = FALSE;
                    else
```

```
                        // 写操作成功地完成。
                        fRes = TRUE;
                    break;

                default:
                    // WaitForSingleObject 发生了一个错误。
                    // 这个通常表明 OVERLAPPED 结构体的事件句柄遇到到了问题。
                    fRes = FALSE;
                    break;
                }
            }
        }
        else
            // WriteFile 执行完毕。
            fRes = TRUE;

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

注意上面代码中使用了 **WaitForSingleObject** 函数和 INFINITE 作为超时值。这将导致 **WaitForSingleObject** 函数永远等待下去，直到操作的完成。这可能使线程或程序被“挂起”。实际上，写操作只不过是花费很长时间去完成，或是流控制传输被阻塞。后面将要介绍的状态检测可以用于检测这种情况，但是不会导致 **WaitForSingleObject** 函数返回。三种办法可以减轻这种情况的发生：

- 放置代码到一个单独的线程。这将允许其他线程执行任何它们想要的功能，而我们的写线程将等待写操作的完成。这便是 MTTTY 例子中所产用的办法。
- 利用 COMMTIMEOUTS 来实现在超时时间过去后完成写操作。这将在稍后这篇文章的“通信的超时”部分做更加详细的讨论。这也是在 MTTTY 例子中所考虑的。
- 改变 **WaitForSingleObject** 去调用包含真正的超时时间值。这将导致更多的问题，因为如果程序开始执行了另外一个操作，而有可能一个早期的操作仍然处于等待状态。这时候新的 **OVERLAPPED** 结构和重叠事件就需要被分配。这种记录保持类型是很麻烦的，尤其是当为操作设计一个“工作队列”去进行比较的时候。

注意：超时值在同步函数中不是通信超时。同步超时将会导致 **WaitForSingleObject** 或 **WaitForMultipleObject** 返回 WAIT_TIMEOUT。这和读写操作的超时还是不同的。通信超时将在这篇文章稍后部分进行描述。

因为 **WaitForSingleObject** 函数在上面代码段中使用了一个 INFINITE 作为超时，它等同于 **GetOverlappedResult** 函数为其 *fWait* 参数赋为 TRUE 的结果。下面是对它进行简化后的等价形式的代码：

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

```
    BOOL fRes;

    // 创建写操作，并对相应的 OVERLAPPED 结构体的 hEvent 成员赋值。
    osWrite.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    if (osWrite.hEvent == NULL)
        // 创建重叠事件句柄发生错误。
        return FALSE;

    // 执行写操作。
    if (!WriteFile(hComm, lpBuf, dwToWrite, &dwWritten, &osWrite)) {
        if (GetLastError() != ERROR_IO_PENDING) {
            // WriteFile 失败但不延时。报告错误并使中止。
            fRes = FALSE;
        }
        else {
            // 写等待。
            if (!GetOverlappedResult(hComm, &osWrite, &dwWritten, TRUE))
                fRes = FALSE;
            else
                // 写操作成功地完成。
                fRes = TRUE;
        }
    }
    else
        // 写操作执行完毕。
        fRes = TRUE;

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

GetOverlappedResult 并不是用来等待重叠操作完成的最好方法。例如，如果一个应用程序需要等待另一个事件句柄，第一段代码将比第二段执行的更好。调用 **WaitForSingleObject** 将很容易修改为 **WaitForMultipleObject** 以便添加附加的等待句柄。这也是 MTTTY 例程中所使用的方法。

在使用重叠 I/O 中一个普遍的错误是，先前的重叠操作还没有完成就重用了 **OVERLAPPED** 结构体。如果先前的重叠操作完成之前要执行一个新的重叠操作，那么就应该为其分配一个新的 **OVERLAPPED** 结构体。当然也必须为 **OVERLAPPED** 结构体中 **hEvent** 成员创建一个新的手工重置事件。一旦一个重叠操作完成，那么 **OVERLAPPED** 结构体和它的事件应该被释放，以便再次使用。

对于串口通信来说 **OVERLAPPED** 结构体只需要修改 **hEvent** 成员变量。**OVERLAPPED** 结构体中的其它成员应该初始化为 0，暂且不需要使用。修改 **OVERLAPPED** 结构体中的其它成员对于串行通信设备来说是没有必要的。**ReadFile** 和 **WriteFile** 的相关文档中声明：OVERLAPPED 结构体中的 Offset 和 OffsetHigh 参数必须通过应用程序去更新，否则它们的结果将是不可预知的。应用于 OVERLAPPED 结构体的这一规则也被用在其它资源类型上。例如，文件操作。

串行状态

有两个方法可以得到通信端口的状态。第一种方法是设置一个事件掩码，当期望的事件发生的时候将导致应用程序发出通告消息。使用 **SetCommMask** 函数设置这个事件掩码，调用 **WaitCommEvent** 函数等待事件的发生。这个函数和 16 位的 **SetCommEventMask** 以及 **EnableCommNotification** 函数很相似，除了 Win32 函数不发送 WM_COMMNOTIFY 消息之外。实际上 WM_COMMNOTIFY 消息甚至不是 Win32 API 的一部分。检测通信端口的第二种方法是定期的调用少数的几个状态函数。当然，这种程序查询方式既没有效率也不推荐。

通信事件

使用通信端口可能随时导致通信事件的发生。涉及接受通信事件通告消息的两个步骤如下所示：

- **SetCommMask** 设置引起一个通告消息所要求的事件。
- **WaitCommEvent** 发出一个状态检查。这个状态检查可以是一个重叠的或者是一个非重叠操作，可能仅仅就像读和写操作那样。

注意：上下文中提到的*事件*这个词只是指通信事件。它不是指用于同步的事件对象。

这里是一个关于 SetCommMask 函数的例子：

```
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))
      // 设置通信事件错误
```

有关每个事件类型的描述，如表 1 所示。

表 1　通信事件标记

事件标记	描述
EV_BREAK	检测一个输入中断。
EV_CTS	CTS（clear-to-send）信号状态发生变化。调用**GetCommModemStatus**函数可以得到CTS信号的真实状态。
EV_DSR	DSR（data-set-ready）信号状态发生变化。调用**GetCommModemStatus**函数可以得到DSR信号的真实状态。
EV_ERR	发生一个线状态错误。线状态错误用CE_FRAME，CE_OVERRUN，和CE_RXPARITY来标示。调用**ClearCommError**函数可以得到错误的具体类型。
EV_RING	检测到一个振铃指示符。
EV_RLSD	RLSD（receive-line-signal-detect）信号状态发生变化。调用GetCommModemStatus函数可以的到RLSD信号的真实状态。注意，通常将RLSD信号归类为CD（carrier-detect）信号行。

EV_RXCHAR	一个新字符被接收且被放入了输入缓冲区。下面“告诫”那一章节将对这个标记进行讨论。
EV_RXFLAG	事件字符被接收并放入到输入缓冲区。事件字符通过**DCB**结构体中的**EvtChar**成员变量来指定，我们在后面会介绍该结构体。下面的“告诫”那一章节也对这个标记进行介绍。
EV_TXEMPTY	输出缓冲区中最后一个字符被发送到串行端口设备。如果使用的是硬件缓冲区，这个标记只用来表示所有的数据已经被发送到了该硬件。不过没有办法去检测硬件缓冲区何时为空，因为没有直接和硬件进行交流的设备驱动。

在指定了事件掩码之后，调用 **WaitCommEvent** 函数来检测事件的发生。如果端口是以非重叠操作打开，那么 **WaitCommEvent** 函数将不必包含 **OVERLAPPED** 结构体。该函数将阻塞调用它的线程，直到一个事件的发生。如果一个事件从来没有发生，那么线程将无限期的被阻塞。

这里展示了当端口通过非重叠方式打开后怎样等待一个 EV_RING 事件发生的代码段：

```
DWORD dwCommEvent;

if (!SetCommMask(hComm, EV_RING))
      // 通信掩码设置错误。
      return FALSE;

if (!WaitCommEvent(hComm, &dwCommEvent, NULL))
      // 一个等待事件错误。
      return FALSE;
else
      // 事件成功发生。
      return TRUE
```

注意：在微软 Win32 SDK 基础知识文档中描述了 EV_RING 标记在 Windows 95 中的一个问题。上面的代码在 Windows 95 中将永远不会返回，因为 EV_RING 事件不会被系统所检测。但在 Windows NT 系统中 EV_RING 事件会被正确地报告。请查阅 Win32 SDK 基础知识文档以获取有关于这个漏洞的更多的信息。

也要注意，上面代码可能被永远阻塞如果一个事件从来没有发生。一个好的方法是用重叠操作方式打开端口等待一个状态事件发生，如下代码所示：

```
#define STATUS_CHECK_TIMEOUT 500 // 毫秒

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))
```

```
    // 通信掩码设置错误；使中止
        return 0;

osStatus.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
if (osStatus.hEvent == NULL)
    // 创建事件错误；使中止
        return 0;

for ( ; ; ) {
    // 执行一个状态事件，如果还没有状态事件已经发生的话。
    if (!fWaitingOnStat) {
        if (!WaitCommEvent(hComm, &dwCommEvent, &osStatus)) {
                if (GetLastError() == ERROR_IO_PENDING)
                    fWaitingOnStat = TRUE;
                    // 原文为：bWaitingOnStatusHandle = TRUE;
                else
                    // WaitCommEvent 函数错误；使中止。
                    break;
        }
        else
            // WaitCommEvent 函数执行成功。
            // 适当的处理状态事件。
            ReportStatusEvent(dwCommEvent);
    }

    // 检查重叠操作。
    if (fWaitingOnStat) {
        // 等待一个事件发生。
        dwRes = WaitForSingleObject(osStatus.hEvent, STATUS_CHECK_TIMEOUT);
        switch(dwRes)
        {
            // 事件发生。
        case WAIT_OBJECT_0:
            if (!GetOverlappedResult(hComm, &osStatus, &dwOvRes, FALSE))
                // 重叠操作发生了一个错误；调用 GetLastError 函数可以
                // 找出错误的根源，并中止致命的错误。
            else
                // 状态事件被储存在事件标记中，调用原始的 WaitCommEvent 所指定。
                // 适当的处理状态事件。
                ReportStatusEvent(dwCommEvent);
            // 设置 fWaitingOnStat 标记变量，
            // 指明将要执行一个新的 WaitCommEvent。
            fWaitingOnStat = FALSE;
            break;

        case WAIT_TIMEOUT:
            // 操作还没有完成。fWaitingOnStat（原文为：fWaitingOnStatusHandle）
            // 标记变量不会被改变，因为我将返回循环继续执行，
            // 我不想在第一个 WaitCommEvent 还没有结束就去执行另外一个。
            //
```

```
                // 在这里比较适合做一些返回工作。
                DoBackgroundWork();
                break;
            default:
                // WaitForSingleObject 发生错误；使中止。
                // 这表明 OVERLAPPED 结构体中的事件处理发生了问题。
                CloseHandle(osStatus.hEvent);
                return 0;
            }
        }
    }

    CloseHandle(osStatus.hEvent);
```

上面的代码非常像重叠读操作的代码。实际上，在 MTTTY 例子中执行读和状态检查在同一个线程里面，使用 **WaitForMultipleObjects** 函数等待读事件或状态事件变成有线号状态。

关于 **SetCommMask** 和 **WaitCommEvent** 有两个有趣的意外情况。首先，如果通信端口以非重叠操作打开，**WaitCommEvent** 将被阻塞直到一个事件的发生。如果另外一个线程调用 **SetCommMask** 去设置一个新的事件掩码，那么线程将被 **SetCommMask** 函数所阻塞。原因是原先调用 **WaitCommEvent** 的那个线程仍然在执行。调用 **SetCommMask** 的线程将被阻塞，直到第一个线程中的 **WaitCommEvent** 函数被返回。在使用非重叠 I/O 打开端口的时候这种意外情况很常见。如果一个线程被任意一个通信函数所阻塞，这时候另外一个线程如果调用该通信函数也会被阻塞，直到第一个线程中的函数返回。这些函数第二点需要注意的是用重叠方式打开一个端口。如果 **SetCommMask** 设置一个新的事件掩码，任何处于等待状态的 WaitCommEvent 函数将成功地完成，该操作产生的事件掩码将为 NULL。

告诫

使用 EV_RXCHAR 标记将通告给线程一个字节到达了端口。这个事件和 **ReadFile** 函数联合使用，实现当字符到达缓冲区的时候读取数据，而不是执行一个读操作等待数据的到来。当一个端口用非重叠方式打开的时候尤其有用，因为程序不需要用程序查询方式来接收数据，通过发生 EV_RXCHAR 事件程序将被通知接收数据。最初试图解决这个问题通常使用下面的伪代码段，不过有一些疏忽的地方将在这一节后面部分介绍。

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

if (!SetCommMask(hComm, EV_RXCHAR))
    // 设置通信事件掩码错误。

for ( ; ; ) {
    if (WaitCommEvent(hComm, &dwCommEvent, NULL)) {
        if (ReadFile(hComm, &chRead, 1, &dwRead, NULL))
```

```
                // 读取一个字节并处理。
            else
                // 调用 ReadFile 函数发生一个错误。
                break;
        }
        else
            // WaitCommEvent 函数错误。
            break;
}
```

上面代码等待一个 EV_RXCHAR 事件发生。当发生后，代码段会调用 **ReadFile** 函数读取得到的一个字节。然后循环继续执行，等待另一个 EV_RXCHAR 事件。当一到两个字节快速连续的到达时，这段代码能够良好的工作。字节接收 EV_RXCHAR 事件的发生，然后代码段执行读字节。如果在调用 **WaitCommEvent** 函数前没有其它字节到达也没事，下一个字节的到来将引起 **WaitCommEvent** 函数执行，从而表明 EV_RXCHAR 事件的发生。如果另外一个字节到来之前代码段正好执行 **WaitCommEvent** 函数，当然这也正常。第一个字节如前面所述将被读取，那么到来的第二个字节将会导致 EV_RXCHAR 标记被隐式的设置。当代码段返回到 **WaitCommEvent** 函数执行，表明 EV_RXCHAR 事件的发生，调用 **ReadFile** 函数将从端口读取第二个字节。

当三个或更多的字节快速连续的到来时上面代码就会遇到一些问题。第一个字节导致 EV_RXCHAR 事件的发生；第二个字节导致 EV_RXCHAR 标记被隐式的设置。接下来会调用 **WaitCommEvent** 函数，它表示 EV_RXCHAR 事件。现在，第三个字节到达了通信端口。第三个字节将导致系统试图去隐式地设置 EV_RXCHAR 标记。因为这个动作在第二个字节到来的时候就已经发生了，所以到达的第三个字节将不会被通知。这段代码到最后读取第一个字节将没什么问题。之后，这段代码将调用 **WaitCommEvent** 函数，预示着 EV_RXCHAR 事件的发生（来至于第二个字节）。第二个字节被读取，代码返回到 **WaitCommEvent** 函数。第三个字节在系统内部接收缓冲区处于等待状态。这时候代码和系统已经不同步了。当第四个字节最后到来，导致 EV_RXCHAR 事件发生，代码段读取到的字节是第三个字节。这将继续不确定的执行下去。

解决这个问题的简单方式就是增加读操作要求的字节数量。不要只接收一个字节，这段代码可以请求两个、十个或更多数量的字节。当两个或更多额外字节超过读操作所要求的字节快速连续到达端口时候，这个方法依然会失败。所以，如果读取两个字节，然而四个字节快速连续的到来将会导致问题。即使是十个字节，如果十二个字节被快速连续的到达端口时也会失败。

解决这个问题的真正方法是从端口不停地读数据直到没有剩余字节。下面的伪代码解决了这个问题，它通过循环的读取数据直到没有剩余。另外一个可能的方法是调用 **ClearCommError** 去决定缓冲区的字节数，然后通过一次读操作全部读取它们。这个方法要求更加复杂的缓冲管理，但是它减少了读取的次数，当有许多数据一次性到来的时候。

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

if (!SetCommMask(hComm, EV_RXCHAR))
```

```
    // 通信事件掩码设置错误。

for ( ; ; ) {
    if (WaitCommEvent(hComm, &dwCommEvent, NULL)) {
        do {
            if (ReadFile(hComm, &chRead, 1, &dwRead, NULL))
                // 读取一个字节并处理。
            else
                // 调用 ReadFile 函数发生一个错误。
                break;
        } while (dwRead);
    }
    else
        // WaitCommEvent 函数发生错误。
        break;
}
```

如果没有设置适当的超时上面代码将不能正确地工作。通信超时将在后面讨论，它会影响到 **ReadFile** 操作的行为，以便在没有字节到来的时候能够返回。有关于这个主题的讨论将在这篇文章后面“通信超时”那一节来介绍。

上面有关于 EV_RXCHAR 的告诫也适用于 EV_RXFLAG。如果标记字符快速连续的到来，EV_RXFLAG 事件可能也不会对每个字符都做出响应。解决这个问题的方法如同上面，就是读取所有的字节直到没有剩余。

上面告诫也适用于其它和接收字符无关的事件。如果其它事件快速连续的发生，一些通告消息将会被丢失。例如，如果 CTS 线电压开始为高，然后变为低，再变为高到低，将会产生一个 EV_CTS 事件。如果 CTS 线电压变化的非常快，将不能保证 EV_CTS 事件被 **WaitCommEvent** 函数准确的检测到。所以，**WaitCommEvent** 函数不能用来记录线状态。线状态将在这篇文章后面“调制解调器状态”章节进行介绍。

错误处理和通信状态

如果调用 **SetCommMask** 函数需要指定一个通信事件标记的话，可能就是 EV_ERR 了。EV_ERR 事件的发生表明通信端口存在错误的状态。端口也可能会发生其它错误，但是不会引起 EV_ERR 事件的发生。不论是哪种情况，和通信端口关联的错误将会导致所有的 I/O 操作被挂起直到错误状态被排除。**ClearCommError** 函数用来检测错误和排除错误状态。

ClearCommError 函数也提供通信状态用来表明为什么传输被停止，它也可以指明传输和接收缓冲区的字节数。导致传输停止的原因是因为错误或流控制。关于流控制将在后文进行讨论。

下面代码展示了怎样调用 **ClearCommError** 函数：

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

//  获得并清除端口上的错误。
if (!ClearCommError(hComm, &dwErrors, &comStat))
    // 报告 ClearCommError 获得的错误。
    return;

// 得到错误标记。
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 结构体包含了有关于通信状态的信息。
if (comStat.fCtsHold)
    // 传输等待 CTS（clear-to-send）信号被发送。

if (comStat.fDsrHold)
    // 传输等待 DSR（data-set-ready）信号被发送。

if (comStat.fRlsdHold)
    // 传输等待 RLSD（receive-line-signal-detect）信号被发送。

if (comStat.fXoffHold)
    // 传输等待，XOFF 字符被接收。

if (comStat.fXoffSent)
    // 传输等待，XOFF 字符被发送。

if (comStat.fEof)
    // EOF（end-of-file）字符被接收。

if (comStat.fTxim)
    // 字符等待传输；涉及字符队列，通过 TransmitCommChar 函数发送。

if (comStat.cbInQue)
    // comStat.cbInQue 表示接收道德字节数目，但是还没有读取。

if (comStat.cbOutQue)
    // comStat.cbOutQue 表示等待传输的字节数。
```

调制解调器状态（又名：线状态）

调用 **SetCommMask** 函数可能包含的标记有 EV_CTS、EV_DSR、EV_RING、和 EV_RLSD。这些标记预示着串行端口线上电压的改变。当然并没有指明这些线的实际状态，只是发生了一个改变。通过 **GetCommMaskStatus** 函数可以得到这些线状态的实际情况。该函数返回一个掩码位，对于每个线来说，0 表示低电平或没有电平；1 表示高电平。

要注意关于术语 RLSD（receive-line-signal-detect）通常被称为 CD（carrier-detect）线。

注意：如同前面提到的那样，EV_RING 标记在 Windows 95 中不能工作。然而，GetCommModemStatus 函数会检测 RING 线的状态。

这些线状态的改变也会引起一个流控制事件。通过 **ClearCommError** 函数报告是否传输会因为流控制而被挂起。如果有必要，一个线程可以调用 ClearCommError 函数去检测是否是流控制活动导致的事件。流控制将在本文后面“流控制”那一节进行介绍。

下面代码展示了怎样调用 **GetCommModemStatus** 函数：

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

if (!GetCommModemStatus(hComm, &dwModemStatus))
      // GetCommModemStatus 函数发生错误。
      return;

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

// 用这些标记做一些工作。
```

扩展的功能

驱动程序会根据需要自动地改变控制线的状态。一般而言，改变状态线受驱动的控制。如果一个设备通过通信端口控制线的方式不同于 RS-232 标准，那么标准的串行通信驱动将不能正常地控制设备。如果标准的串行通信驱动不能控制设备，就需要使用定制的设备驱动。

有时候标准控制线受应用程序的控制，而不是串行通信驱动。例如，一个应用程序可能希望它自己实现流控制。这个应用程序将负责，处理 RTS 和 DTR 线状态的改变。**EscapeCommFunction** 被通信驱动所应用去实现那样的扩展的操作。利用 **EscapeCommFunction** 可以使驱动实现一些其它的功能，就像

设置或清除一个 BREAK 状态。关于这个函数的更多信息，请查阅平台 SDK 文档、微软 Win32 SDK 基础知识、或微软开发者网络文库（MSDN）。

串行设置

DCB 设置

编写串口通信应用最重要的方面就是设置设备控制块（DCB）结构体。在串行通信编程中最普遍的错误来源于不适当地初始化 DCB 结构体。当串行通信函数不能够按照预期执行的时候，通常最有可能就是 DCB 结构体设置出了问题。

有三种方法初始化 DCB 结构体。第一种方法是使用 **GetCommState** 函数。这个函数返回通信端口现在正使用的 DCB 结构。下面代码展示了怎样使用 **GetCommState** 函数：

```
DCB dcb = {0};

if (!GetCommState(hComm, &dcb))
      // 得到当前 DCB 设置错误。
else
      // 准备使用 DCB 结构。
```

第二种方法通过调用 **BuildCommDCB** 去初始化一个 DCB 结构。这个函数用于填充 DCB 结构的成员波特率、奇偶校验类型、停止位的个数、和数据位的个数。这个函数同时也设置流控制成员为默认值。查阅 **BuildCommDCB** 函数帮助文档可获得有关于流控制成员默认值的细节内容。DCB 结构的其它成员不受这个函数的影响。你编写的程序有责任确保 DCB 结构其它成员不会引起错误。最简单需要注意的事情就是用 0 去初始化 DCB 结构，然后以字节为单位设置 DCB 结构的大小。如果没有用 0 去初始化 DCB 结构成员，那么非零值有可能是预设成员，这将产生一个错误当在后面使用 DCB 结构的时候。下面代码展示了怎样正确的使用这个方法：

```
DCB dcb;

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

if (!BuildCommDCB("9600,n,8,1", &dcb)) {
      // 不能创建 DCB。通常是通信字符串不规范引起的问题。
      return FALSE;
}
else
      // 准备使用 DCB 结构。
```

第三种方法是手动初始化 DCB 结构。程序申请 DCB 结构，然后设置每个成员为任何想要的值。这

个方法在 Win32 系统中实现的时候，将不能很好处理 DCB 结构的改变，所以并不推荐。

应用程序通常需要设置一些和默认值并不相同的 DCB 成员，也可能需要在程序执行的过程中对设置进行修改。一旦正确初始化了 DCB 结构，那么可能会或多或少的对个别成员进行修改。改变 DCB 结构不会对端口的行为有任何影响，直到调用 **SetCommState** 函数。下面代码展示了得到当前 DCB，改变波特率，然后尝试修改配置的操作：

```
DCB dcb;

FillMemory(&dcb, sizeof(dcb), 0);
if (!GetCommState(hComm, &dcb)) //  得到当前 DCB。
      // GetCommState 函数错误。
      return FALSE;

// 更新 DCB 波特率。
dcb.BaudRate = CBR_9600 ;

// 设置新的波特率。
if (!SetCommState(hComm, &dcb))
      // SetCommState 函数错误。
      // 可能是通信端口处理问题或是 DCB 结构本身的问题。
```

这里是有关于 DCB 结构每个成员的解释，还介绍了它们是怎样影响其它的串行通信函数。

注意：这里大部分信息来至于平台 SDK 文档。因为文档是解释这些成员的官方材料，当然由于操作系统不同，这个表也不一定是完全正确无误的。

表 2　DCB结构成员

成员	描述	
DCBlength	以字节为单位的DCB结构的大小。在调用**SetCommState**函数更新设置之前应该首先设置该参数。	
BaudRate	指定通信设备操作的波特率。这个参数可以是实际的波特率，或是波特率索引值。	
fBinary	指定是否使用二进制方式。Win32 API不支持非二进制方式传输，所以这个参数应该设置为TRUE，如果为FALSE将不能工作。	
fParity	指定是否奇偶校验。如果这个参数设为TRUE，将进行奇偶校验并报告错误信息。不要和**Parity**成员混淆了，**Parity**参数用来控制通信中使用的奇偶校验类型。	
fOutxCtsFlow	指定CTS（clear-to-send）信号是否检测输出流控制。如果这个参数为TRUE且CTS为低，则会暂停输出，直到CTS信号变为高。CTS信号受DCE（通常为调制解调器）控制，DTE（通常为PC）仅仅用来检测这个信号的状态，并不会改变它。	
fOutDsrFlow	指定DSR（data-set-ready）信号是否检测输出流控制。如果这个参数为TRUE且DSR为低，则会暂停输出，知道DSR信号变为高。这个信号也受DCE的控制，DTE只用来检测这个信号。	
fDtrControl	指定DTR（data-terminal-ready）输入流控制。这个参数可以设置为下面的值：	
	值	**含义**
	DTR_CONTROL_DISABLE	当设备打开的时候降低DTR线。通过**EscapeCommFunction**函数可以是应用程序调整线状态。
	DTR_CONTROL_ENABLE	当设备打开的时候增高DTR线。通过**EscapeCommFunction**函数可以是应用程序调整线状态。
	DTR_CONTROL_HANDSHAKE	允许DTR流控制进行信息交换。如果使用该值，则应用程序通过**EscapeCommFunction**函数调整线状态会出错。

fDsrSensitivity	指定通信驱动是否蜜柑DSR信号状态。如果这个参数设为TRUE，驱动将忽略任何接收到的字节，除非DSR调制解调器输入线设为高。
fTXContinueOnXoff	指定当输入缓冲区满的时候是否传输停止，且驱动已经将XOFF字符发送出去了。如果这个参数为TRUE，则在XOFF字符被发送后传输继续。如果这个参数为FALSE，传输不会不继续，直到输入缓冲区包含XonLim字节数代表空，且驱动程序已将XON字符发送出去。
fOutX	指定在传输期间是否使用XON/XOFF流控制。如果这个参数设置为TRUE，当收到XOFF字符时候传输停止，当收到XON字符时候传输继续。
fInX	指定在接收期间是否使用XON/XOFF流控制。如果这个参数设置为TRUE，当接收到代表缓冲区已满的XoffLim字节数时候，XOFF字符被发送出去；当接收到代表接收缓冲区已空的XonLim字节数时候，XON字符被发送出去。
fErrorChar	指定是否用**ErrorChar**成员所指定的字符替换掉奇偶校验出错的字节。如果这个参数为TRUE且**fParity**参数也为TRUE时，才会发生替换。
fNull	指定空字节是否被丢弃。当这个参数为TRUE，那么接收到的空字节将会被丢弃。
fRtsControl	指定RTS（request-to-send）输入流控制。缺省值为RTS_CONTROL_HANDSHAKE，值为0。这个参数可以的取值如下所示：
	值 — **含义**
	RTS_CONTROL_DISABLE — 当设备打开后降低RTS线。应用程序可以使用**EscapeCommFunction**函数来改变线状态。
	RTS_CONTROL_ENABLE — 当设备打开时候增高RTS线。应用程序可以使用**EscapeCommFunction**函数来改变线状态。
	RTS_CONTROL_HANDSHAKE — 允许RTS流控制信息交换。驱动增高RTS线，允许DCE被发送，当输入缓冲区有足够的缓冲区来接收数据的时候。驱动降低RTS线，阻止DCE被发送，当缓冲区没有足够的缓冲区去接收数据的时候。如果这个值被使用，那么应用程序在使用**EscapeCommFunction**函数调整线状态的时候会出错。
	RTS_CONTROL_TOGGLE — 如果还有传输的字节，则指定RTS线为高状态。当所有缓冲区字节被发送后，RTS线将变为低状态。如果设置该值，那么应用程序在使用**EscapeCommFunction**函数调整线状态的时候会出错。在Windows95中这个值会被忽略掉，因为只有指定RTS_CONTROL_ENABLE驱动才会起作用。
fAbortOnError	指定如果一个错误产生的时候，是否终止读写操作。如果这个参数值为TRUE，当错误发生的时候，驱动程序将终止所有读写操作，并显示一个错误状态（ERROR_IO_ABORTED）。驱动程序将不会接受任何进一步的通信操作，直到应用程序调用**ClearCommError**函数处理该错误。
fDummy2	保留位，不使用。
wReserved	不被使用，必须设置为0。
XonLim	指定XON字符在被发送前，输入缓冲区所允许的最小字节数。
XoffLim	指定XOFF字符在被发送前，输入缓冲区所允许的最大字节数。最大字节数为接收缓冲区大小减去该值，以字节为单位。
ByteSize	发送和接收字节的位数。（原文遗漏）
Parity	指定使用的奇偶校验方式。这个参数可以的取值如下所示：
	值 — **含义**
	EVENPARITY — 偶校验
	MARKPARITY — 标记校验
	NOPARITY — 无校验
	ODDPARITY — 奇校验
	SPACEPARITY — 空格校验（原文遗漏）
StopBits	指定所使用的停止位个数。这个参数可以得取值如下所示：
	值 — **含义**
	ONESTOPBIT — 1个停止位
	ONE5STOPBITS — 1.5个停止位
	TWOSTOPBITS — 2个停止位
XonChar	指定传输和接收XON字符的值。
XoffChar	指定传输和接收XOFF字符的值。
ErrorChar	指定该字符的值，用来替换奇偶校验发生错误的字节。

EofChar	指定用来表示数据结束的字符。
EvtChar	指定该字符的值，通常使用它引起EV_RXFLAG事件。如果**SetCommMask**函数没有使用EV_RXFLAG事件，且没有使用**WaitCommEvent**函数，则该设置将不起任何作用。
wReserved1	保留位，不适用。

流控制

在串行通信中，当一个设备正忙或是因为一些原因不能够进行任何通讯时候，流控制提供了暂停通信的方法。通常流控制有两种类型：硬件和软件。

串口通信有一个普遍的问题就是，写操作实际上没有把数据写入设备中。通常，当程序没有指定它的时候，这个问题存在于流控制中。如果对 DCB 结构进行检查可以发现下面这些成员变量中的一个或多个参数值可能为 TRUE：fOutxCtsFlow、fOutxDsrFlow、或 fOutX。另外一种方式是调用 ClearCommError 函数去检查 COMSTAT 结构进行检查。当因为流控制传输被暂停的时候，错误会被检查出来。

在讨论流控制类型之前，很好的理解一些术语是很有必要的。串行通信在两个设备之间进行。通常有一个 PC 和一个调制解调器或打印机。PC 被称作 DTE（data-terminal-equipment）。DTE 有时被称作 *host*（主机）。调制解调器、打印机、或可识别其它的外围设备被称作 DCE（data-communications-equipment）。DCE 有时也被称作 *device*（设备）。

硬件流控制

硬件流控制使用串行电缆控制线上的电压信号去控制是否发送和接收数据。在通信的时候，DTE 和 DCE 必须保证流控制类型一致。设置 DCB 结构体使流控制有效只是用于配置 DTE。为了保证 DTE 和 DCE 使用相同类型的流控制，DCE 也需要配置。在 Win32 中没有办法去设置 DCE 的流控制。通常使用 DIP 开关或发送指令来对它进行配置。有关控制线、流控制、和线对 DTE 以及 DCE 影响的描述如表 3 所示：

表 3　硬件流控制线

线以及方向	对DTE/DCE的作用
CTS（Clear To Send）输出流控制	DCE设置线为高表明它可以接收数据。DCE设置线为低表明它不能接收数据。 如果DCB结构中的**fOutxCtsFlow**成员为TRUE，那么DTE将不发送数据，如果这个线为低。如果线为高则发送将重新开始。 如果DCB结构中的**fOutxCtsFlow**成员为FALSE，那么线的状态对传输不会产生影响。
DSR（Data Set Ready）输出流控制	DCE设置线为高表明它可以接收数据。DCE设置线为低表明它不能接收数据。 如果DCB结构中的**fOutxDsrFlow**成员为TRUE，那么DTE将不发送数据，如果这个线为低。如果线为高则发送将重新开始。 如果DCB结构中的**fOutxDsrFlow**成员为FALSE，那么线的状态对传输不会产生影响。
DSR（Data Set Ready）输入流控制	如果DSR线为低，到达端口的数据将被忽略。如果DSR线为高，到达端口的数据将会被接受。 如果DCB结构中的**fDsrSensitivity**成员设置为TRUE，上述情况才会发生。如果它为FALSE，那

	么线的状态将不会对接收造成影响。
RTS（Ready To Send）输入流控制	RTS线被DTE控制。 如果DCB结构中的**fRtsControl**成员被设置为RTS_CONTROL_HANDSHAKE，则被使用的流控制为如下方式：如果输入缓冲区有足够的空间去接收数据（至少有一半为空），RTS线将会被驱动设置为高。如果输入缓冲区没有足够的空间接收数据（缓冲区小于四分之一为空），RTS线将会被驱动设置为低。 如果DCB结构中的**fRtsControl**成员被设置为RTS_CONTROL_TOGGLE，被发送的数据有效时驱动会设置RTS线为高。当被发送的数据无效时，驱动会设置线为低。Windows 95系统忽略了这个值，效果和RTS_CONTROL_ENABLE是一样的。 如果DCB结构中的**fRtsControl**成员被设置为RTS_CONTROL_ENABLE或RTS_CONTROL_DISABLE，应用程序将根据需要自由的改变线的状态。注意如果这样，线的状态将不再对接收造成影响。 当线为低时，DCE将暂停传输。当线为高时，DCE将重新开始传输。
DTR（Data Terminal Ready）输入流控制	DTR线被DTE控制。 如果DCB结构中的**fDtrControl**成员被设置为DTR_CONTROL_HANDSHAKE，则被使用的流控制为如下方式：如果输入缓冲区有足够的空间去接收数据（至少有一半为空），DTR线将会被驱动设置为高。如果输入缓冲区没有足够的空间接收数据（缓冲区小于四分之一为空），DTR线将会被驱动设置为低。 如果DCB结构中的**fDtrControl**成员被设置为DTR_CONTROL_ENABLE或DTR_CONTROL_DISABLE，应用程序将根据需要自由的改变线的状态。注意如果这样，线的状态将不再对接收造成影响。 当线为低时，DCE将暂停传输。当线为高时，DCE将重新开始传输。

当 CE_RXOVER 错误发生的时候，对流控制的需求是很容易被识别的。这个错误表明接收缓冲区发生了溢出且数据被丢失。如果数据到达端口的速度比被读取它的速度快时，CE_RXOVER 错误可能会发生。增加输入缓冲区的大小，可以避免这个错误经常地发生，但是却不能彻底解决这个问题。当驱动检测到输入缓冲区就要满的时候，它将使输入流控制线变为低。这将导致 DCE 停止传输，以便让 DTE 有足够的时间从输入缓冲区读取数据。当输入缓冲区有足够的空间可用的时候，流控制的电压被拉高，DCE 将恢复发送数据。

还有一个类似的错误为 CE_OVERRUN。当通信硬件和串行通信驱动没有完全接收老的数据时候，新数据的到来将引起该错误。如果通信硬件和 CPU 是传输速度太高的类型，这个错误可能发生。当操作系统不能自由的为通信硬件服务时也可能导致这个错误的发生。减轻这个问题发生的唯一方法是：用一些减少传输速度的组合替代那些会使 CPU 速度增高的通信硬件。有时第三方的硬件驱动会因为这个错误不能很好的利用 CPU 资源。流控制虽然能帮助减少错误的发生，但不能完全解决这个问题。

软件流控制

软件流控制使用通信流中的数据去控制传输和接收数据。因为软件流控制使用两个特殊的字符 XOFF 和 XON，所以二进制传输不能使用软件流控制，XOFF 和 XON 字符可能出现在二进制数据中从而和数据传输造成冲突。软件流控制适合于基于文本的通信，或是传输的数据中不包括 XON 和 XOFF

字符。

使用软件流控制，必须把 DCB 结构中的 **fOutX** 和 **fInX** 成员变量置为 TRUE。**fOutX** 成员控制输出流控制；**fInX** 成员控制输入流控制。

要注意一点，DCB 允许程序动态分配值，系统会识别出作为流控制的字符。DCB 的 **XoffChar** 成员用于为输入和输出流控制指示 XOFF 字符。DCB 的 **XonChar** 成员同样用于指示 XON 字符。

对于输入流控制，DCB 结构中的 **XoffLim** 成员用于指定 XOFF 字符被发送前输入缓冲区允许的最小自由空间。如果输入缓冲区中的自由空间小于这个值的时候，XOFF 字符将会被发送。对于输入流控制，DCB 结构中的 **XonLim** 成员用于指定 XON 字符被发送前输入缓冲区允许的最小字节数。如果输入缓冲区中的数据量小于这个值的时，XON 字符将被发送。

表 4 列出了 DTE 使用 XOFF/XON 流控制时候的行为。

表 4　软件流控制行为

流控制字符	行为
XOFF：被DTE接收	DTE传输被暂停直到XON被接收，DTE接收继续。DCB结构的**fOutX**成员变量控制这个行为。
XON：被DTE接收	如果DTE传输被暂停因为提前接收到一个XOFF字符，DTE传输将会重新开始。DCB结构中的**fOutX**成员变量控制这个行为。
XOFF：被DTE发送	当接收缓冲区临近满时OXFF字符会被DTE自动地发送。这个实际的限制由DCB结构中的**XoffLim**成员变量来分配。DCB结构中的**fInX**成员变量控制这个行为。DTE传输受DCB结构中的**fTXContinueOnXoff**成员变量控制，该参数将在下面进行介绍。
XON：被DTE发送	当接收缓冲区临近空时XON字符会被DTE自动地发送。这个实际的限制由DCB结构中的**XonLim**成员变量来分配。DCB结构中的**fInX**成员变量控制这个行为。

如果对于输入控制软件流控制为有效状态，那么 DCB 结构的 **fTXContinueOnXoff** 成员变量将为有效状态。**fTXContinueOnXoff** 成员变量用于控制在 XOFF 字符被系统自动发送后是否暂停传输。如果 **fTXContinueOnXoff** 为 TRUE，当接收缓冲区满且 XOFF 被发送后传输继续。如果 **fTXContinueOnXoff** 为 FALSE，那么传输将会被暂停，直到系统自动地发送 XON 字符。DCE 设备使用软件流控制在收到 XOFF 字符后将暂停他们的发送。当 XON 字符被 DTE 发送后，一些设备将重新开始发送。另外，一些 DCE 设备将在收到任何字符后重新开始发送。如果想让 DCE 设备在通信的时候接收到任何字符都重新开始发送，那么 **fTXContinueOnXoff** 应该设为 FALSE。如果 DTE 自动地发送 XOFF 字符后继续传输，通信的恢复将引起 DCE 继续发送，使 XOFF 无效。

在 Win32 API 中没有有效的技巧去使 DTE 表现的和这些设备一样。DCB 结构中没有相应的变量去指定当收到任何字符传输都能从暂停中恢复。XON 是唯一能导致传输恢复的字符。

另外还需要注意一点，如果读操作完成读取字节数为 0，软件流控制接收 XON 和 XOFF 字符将处于等待状态。应用程序不能读取 XON 和 XOFF 字符，因为输入缓冲区没有多余的空间了。

市场上的大部分程序，包括装有 Windows 的终端设备的程序，在流控制方法为使用者提供三种选择：硬件、软件、或没有。当然 Windows 系统它本身并没有用这种方式来限制应用程序。DCB 结构的

设置允许软件和硬件同时执行流控制。实际上允许多个不同的流控制配置，分别配置 DCB 结构成员可能对流控制产生影响。这样强加于流控制选择的限制，是为了确保易用性和减少对最后使用者造成混乱。这种限制当然也因为通信设备可能不支持所有的流控制类型。

通信超时

另外一个对读和写行为有主要影响的操作是超时。超时会对读和写操作产生下面影响。如果一个操作所用时间超过了原先设定的超时，这个操作将被结束。**ReadFile**、**WriteFile**、**GetOverlappedResualt**、或 **WaitForSingleObject** 将不会返回错误代码。所有用于监视这些操作的指示符都表明它成功的完成了。唯一能知道超时的方式是实际传输的字节少于要求被接收的字节数。所以，如果 **ReadFile** 函数返回 TRUE，但是读取的字节却少于要求的字节，那么有可能发生了超时。如果一个重叠写操作超时，重叠事件处于有信号状态，**WaitForSingleObject** 函数将返回 WAIT_OBJECT_O。**GetOverlappedResult** 函数将返回 TRUE，但是 dwBytesTransferred 包含了在超时之前已被传输的字节数。下面代码展示了在重叠写操作中怎样处理这种情况：

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

    // 为这个写操作创建 OVERLAPPED 结构体的 hEvent 参数。
    osWrite.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    if (osWrite.hEvent == NULL)
        // 创建重叠事件句柄时错误。
        return FALSE;

    // 执行写。
    if (!WriteFile(hComm, lpBuf, dwToWrite, &dwWritten, &osWrite)) {
        if (GetLastError() != ERROR_IO_PENDING) {
            // WriteFile 执行失败，但是它没有被延时。报告错误。
            fRes = FALSE;
        }
        else {
            // 写操作等待。
            dwRes = WaitForSingleObject(osWrite.hEvent, INFINITE);

            switch(dwRes) {
                // 重叠事件已经处于有信号状态。
            case WAIT_OBJECT_0:
                if (!GetOverlappedResult(hComm, &osWrite, &dwWritten, FALSE))
                    fRes = FALSE;
                else {
                    if (dwWritten != dwToWrite) {
                        // 写操作超时。我现在需要决定是否中止程序或是重操作。
```

```
                        // 如果我重操作，我只需要发送没有发送的字节。如果
                        // 我使中止，我只需要置 fRes 参数为 FALSE 然后返回。
                        fRes = FALSE;
                    }
                    else
                        // 写操作成功的完成。
                        fRes = TRUE;
                }
                break;

            default:
                // WaitForSingleObject 发生了一个错误。
                // 这个通常表明重叠事件句柄有问题。
                fRes = FALSE;
                break;
            }
        }
    }
    else {
        // 写操作完成。
        if (dwWritten != dwToWrite) {
            // 写操作超时。我现在需要决定是否中止或重新执行。
            // 如果我想重新执行，我只需要发送没有发送的字节。
            // 如果我想中止，我将只设置 fRes 为 FALSE 然后返回。
            fRes = FALSE;
        }
        else
            fRes = TRUE;
    }

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

SetCommTimeouts 函数用于为端口指定通信超时。调用 **GetCommTimeouts** 函数可以得到端口目前的超时。应用程序应该在修改超时之前保存通信超时。这样就能让应用程序在完成串口通信后恢复它们原先的超时设置。下面是一个使用 **SetCommTimeouts** 设置超时的例子：

```
COMMTIMEOUTS timeouts;

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

if (!SetCommTimeouts(hComm, &timeouts))
    // 设置超时错误。
```

注意：再一次提醒读者，通信超时和应用于同步函数的超时值是不一样的。例如，**WaitForSingleObject** 函数的超时值用于等待对象编程有信号状态，这有别于通信超时。

如果设置 **COMMTIMEOUTS** 结构体为全零状态，将不会引起超时。重叠操作将被阻塞，直到所有要求的字节都被传输。**ReadFile** 函数将被阻塞，直到所有要求的字符到达端口。**WriteFile** 函数将被阻塞，直到所有要求的字符都被发送出去。另外，重叠操作将不会结束，直到所有的字符被传输或操作被中止。可能发生下面的情况在操作完成之前：

- **WaitForSingleObject** 函数总是返回 WAIT_TIMEOUT 如果一个同步的超时被应用。**WaitForSingleObject** 函数将永久的被阻塞，如果使用了一个 INFINITE（永久）的同步超时。
- **GetOverlappedResult** 总是返回 FALSE，且 **GetLastError** 函数会返回 ERROR_IO_INCOMPLETE，如果在调用 **GetOverlappedResult** 函数后直接调用 **GetLastError** 函数的话。

通常用下面的方式来设置 **COMMTIMEOUTS** 结构体会使读操作立即完成，且不需要等待任何新的数据到来：

```
COMMTIMEOUTS timeouts;

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

if (!SetCommTimeouts(hComm, &timeouts))
    // 设置超时错误。
```

当和以读为基础的事件一起使用的时候，这些设置是必须的，在前面“告诫”那一小节介绍过。为了在只有 0 字节的时候能让 **ReadFile** 函数返回，**COMMTIMEOUTS** 结构中的 **ReadIntervalTimeout** 成员变量应设置为 MAXDWORD，且 **ReadTimeoutMultiplier** 和 **ReadTimeoutConstant** 参数都设置为 0。

当应用程序在使用通信端口的时候，总是必须得特别地设置超时。通信超时将会影响读和写操作的行为。如果通信端口打开之前没有设置超时，它将使用驱动提供的默认超时，或是前面的通信程序留下的超时。如果一个应用程序把超时设定成一个特定的值，那么超时实际上可能不同，读和写操作可能永远不能完成或是频繁的完成。

小结

这篇文章讨论了在开发串行通信应用程序时候所普遍存在的缺陷和问题。这篇文章配套的多线程 TTY 例子中的很多技术也在这里进行了讨论。请读者自己下载并测试该程序。学习它是怎么工作的，将

帮助你能更加透彻的理解 Win32 串行通信函数。

参考文献

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)