Files and I/0 Technical Articles

Serial Communications in Win32

Allen Denver
Microsoft Windows Developer Support

December 11, 1995

Applies to:
Microsoft® Win32®
Microsoft Windows®

Summary: Learn how serial communications in Microsoft Win32 is significantly different from serial communications in 16-bit Microsoft Windows. This article assumes a familiarity with the fundamentals of multiple threading and synchronization in Win32. In addition, a basic understanding of the Win32 **heap** functions is useful to fully comprehend the memory management methods used by the Multithreaded TTY (MTTTY) sample included with this article. (35 printed pages)

Download the MTTTY sample (4918.exe) for this technical article.

Contents

Overview

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 you are 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. (36 printed pages)

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 sample included with this article, MTTTY: Multithreaded TTY (4918.exe), 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) Input flow control	The DTR line is controlled by the DTE. If the **fDtrControl** member of the DCB is set to 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_O. **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)
