Handshaking and Flow Control

- There are several members of the **DCB** structure which can be used to implement handshaking and flow control
- The **fDtrControl** member is used to turn **DTR** ON or OFF or to specify **DTR** handshaking, which would prevent another application from changing the line.
- An important method of flow control is the **XON/XOFF** flow control.
- This method relies on special characters being sent back and forth to enable or disable data transmission.
- To stop the receiver buffer from overflowing we can send **XOFF (ASCII 19)** to the other side to suspend further transmissions.
- Later the **XON** character (**ASCII 17**) can be used to tell the other side to resume sending data.
- The **XOFF** and **XON** characters are specified by **XoffChar** and **XonChar**.
- If the **fOutX** member is **TRUE**, transmission stops when the **XoffChar** character is received and starts again when the **XonChar** character is received.
- If the **fInX** member is **TRUE**, the **XoffChar** character is sent when the input buffer comes within **XoffLim** bytes of being full, and the **XonChar** character is sent when the input buffer comes within **XonLim** bytes of being empty.
- If the **fTXContinueOnXoff** member is **TRUE**, transmission continues even after the input buffer has come within **XoffLim** bytes of being full and the driver has transmitted the **XoffChar** character to stop receiving bytes.
- 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 XonChar character to resume reception.
- The **fRtsControl** member can be used to turn the RTS ON or OFF.

Win32 Serial I/O - Receiving and Transmitting Data

- I/O can be implemented as a **Polled,** Synchronous, **Asynchronous**, or a fully **Event driven** system.

Polled Method

- In this method a **thread** (discussed later) must continuously poll for **incoming data** and at the same time allow other Windows to run other applications.
- An advantage of this technique is that it is very simple to design and implement.
- The main disadvantage is that this takes up a lot of CPU time since the thread is always executing.
- A loop is used to read data from the port by continuously calling the ReadFile() function:

```
BOOL ReadFile (HANDLE  hCommDev, LPVOID  lpBuffer, DWORD
      nNumberOfBytesToRead, LPDWORD
      lpNumberOfBytesRead,
      LPOVERLAPPED  lpOverlapped);
```

hCommDev

The serial port handle returned by the **CreateFile** function.

lpBuffer

Points to the buffer that receives the data read from the port.

nNumberOfBytesToRead

Specifies the number of bytes to be read from the file.

lpNumberOfBytesRead

Points to the number of bytes actually read.

lpOverlapped

Points to an **OVERLAPPED** structure (used in Asynchronous I/O).

- When the read loop terminates, the thread will call the function **PurgeComm()** which is a cleanup function.
- **PurgeComm** will terminate all background read and writes on the port and flush the I/O buffers.

BOOL PurgeComm (HANDLE hCommDev, DWORD fdwAction);

hCommDev

The serial port handle returned by the **CreateFile** function.

fdwAction

Specifies the action to take. For example, **PURGE_RXABORT** will terminates all outstanding read operations and returns immediately, even if the read operations have not been completed.

- Another serious drawback to polling is that it always attempts to read exactly **nNumberOfBytesToRead** bytes for every **ReadFile** call which may result in multiple calls to read all the data.
- A better technique is to determine the number of bytes waiting in the port's receive buffer and specify that many bytes to read.

- The **ClearCommError** function retrieves information about a communications error and reports the current status of a port, including the number of bytes in the receiver buffer:

  BOOL ClearCommError (HANDLE hCommDev, LPDWORD lpErrors, LPCOMSTAT lpStat);

  hCommDev

  The serial port handle returned by the **CreateFile** function.

  lpErrors

  Points to a 32-bit variable to be filled with a mask indicating the type of error:

CE_BREAK	The hardware detected a break condition.
CE_DNS	A parallel device is not selected.
CE_FRAME	The hardware detected a framing error.
CE_IOE	An I/O error occurred during communications with the device.
CE_MODE	The requested mode is not supported, or the hFile parameter is invalid. If this value is specified, it is the only valid error.
CE_OOP	A parallel device signaled that it is out of paper.
CE_OVERRUN	A character-buffer overrun has occurred. The next character is lost.
CE_PTO	A time-out occurred on a parallel device.
CE_RXOVER	An input buffer overflow has occurred. There is either no room in the input buffer, or a character was received after the end-of-file (EOF) character.
CE_RXPARITY	The hardware detected a parity error.

CE_TXFULL The application tried to transmit a character, but the output buffer was full.

lpStat

Points to a **COMSTAT** structure in which the device's status information is returned.

- The **COMSTAT** structure:

```
typedef struct _COMSTAT {

   DWORD fCtsHold : 1;   // Tx waiting for CTS signal
   DWORD fDsrHold : 1;   // Tx waiting for DSR signal
   DWORD fRlsdHold : 1;  // Tx waiting for RLSD signal
   DWORD fXoffHold : 1;  // Tx waiting, XOFF char rec'd
   DWORD fXoffSent : 1;  // Tx waiting, XOFF char sent
   DWORD fEof : 1;       // EOF character sent
   DWORD fTxim : 1;      // character waiting for Tx
   DWORD fReserved : 25; // reserved
   DWORD cbInQue;        // bytes in input buffer
   DWORD cbOutQue;       // bytes in output buffer
} COMSTAT, *LPCOMSTAT;
```

Synchronous I/O

- The main difference between **Synchronous I/O** and **Polling** is that in the former, the **ReadFile** function is ***Synchronous***.

- That is, the **ReadFile** call does not return until either the number of bytes specified are read or the total timeout interval expires.

- This is done by setting the interval timeout to 0, and specifying a total timeout of say, 50.

- The disadvantage with this technique is that a thread can potentially block for long time intervals causing delays in other parts of the program which may be waiting to process received data.

- The way around this problem is to use the **cbinQue** member of **COMSTAT** to determine the exact number of bytes in the receive buffer.

Events

- Events are a signaling mechanism used to synchronize processes and threads. For our application here we will use events in overlapped structures.

- We can use an event to cause a thread to wait for a certain action to be completed. An event object is accessed through an even handle.

- An handle can be in one of two states: **signaled** and **nonsignaled**.

- The **CreateEvent** function can be used to create a manual-reset event:

```
HANDLE CreateEvent (LPSECURITY_ATTRIBUTES  lpEventAttributes,
        BOOL bManualReset, BOOL  bInitialState, LPCTSTR  lpName);
```

lpEventAttributes

Points to a **SECURITY_ATTRIBUTES** structure that specifies the security attributes for the event object. Set to **NULL** for our application (default security).

bManualReset

Specifies whether a manual-reset or auto-reset event object is created. If **TRUE**, then you must use the **ResetEvent** function to manually reset the state to nonsignaled. If **FALSE**, Windows automatically resets the state to nonsignaled after a single waiting thread has been released.

bInitialState

Specifies the initial state of the event object. If **TRUE**, the initial state is signaled; otherwise, it is nonsignaled.

lpName

Points to a null-terminated string specifying the name of the event object. The name is limited to **MAX_PATH** characters and can contain any character except the backslash path-separator character (\).

If **lpName** is **NULL**, the event object is created without a name.

- After we create an event we can then wait for the automatic event to occur using the **GetOverlappedResult** and **WaitForSingleObject** functions.
- The **GetOverlappedResult** function returns the results of an overlapped I/O operation on the port:

BOOL GetOverlappedResult (HANDLE hFile, LPOVERLAPPED lpOverlapped, LPDWORD lpNumberOfBytesTransferred, BOOL bWait);

hCommDev

The serial port handle returned by the **CreateFile** function.

lpOverlapped

Points to an **OVERLAPPED** structure that was specified when the overlapped operation was started.

lpNumberOfBytesTransferred

Points to a 32-bit variable that receives the number of bytes that were actually transferred by a read or write operation.

bWait

Specifies whether the function should wait for the pending overlapped operation to be completed. If **TRUE**, the function does not return until the operation has been completed.

If **FALSE** and the operation is still pending, the function returns **FALSE** and the **GetLastError** function returns **ERROR_IO_INCOMPLETE**. Used to additional processing.

- If the function succeeds, the return value is nonzero.
- If the function fails, the return value is zero. GetLastError is used to get extended error information, call.

- The **WaitForSingleObject** function returns when one of the following occurs:
  - The specified object is in the signaled state.
  - The time-out interval elapses.

```
DWORD WaitForSingleObject ( HANDLE hHandle,        // object handle
                            DWORD dwMilliseconds   // time-out interval
);
```

- The second argument specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is nonsignaled.
- If **dwMilliseconds** is zero, the function tests the object's state and returns immediately.
- If **dwMilliseconds** is INFINITE, the function's time-out interval never elapses.
- If the function succeeds, the return value indicates the event that caused the function to return. This value can be one of the following.

- The **WaitForSingleObject** function checks the current state of the specified object. If the object's state is nonsignaled, the calling thread enters an efficient wait state.
- The thread consumes very little processor time while waiting for the object state to become signaled or the time-out interval to elapse.

Asynchronous I/O

- Using **Asynchronous I/O**, an application can read (or write) data in the background while carrying out other tasks in the foreground.

- For example, the application can initiate the reading or writing of 1024 bytes and then continue execution while the I/O occurs in the background.

- After the I/O task has been completed, Windows will generate a signal to indicate that to the application.

- Recall that the **CreateFile** function has a **dwFlagsAndAttributes** parameter which when set to **FILE_FLAG_OVERLAPPED**, results in the port being opened for Asynchronous I/O.

- When you specify **FILE_FLAG_OVERLAPPED**, the **ReadFile** and **WriteFile** functions must specify an **OVERLAPPED** structure so that the I/O occurs in the background.

- The **OVERLAPPED** structure contains information used in Asynchronous I/O:

```
typedef struct _OVERLAPPED {
        DWORD  Internal;
        DWORD  InternalHigh;
        DWORD  Offset;
        DWORD  OffsetHigh;
        HANDLE hEvent;
} OVERLAPPED;
```

- The first four members are set to 0 when dealing with serial ports.

- **hEvent** specifies a manual-reset event, which is set to the signaled state when the transfer has been completed. The calling process sets this member before calling the **ReadFile**, **WriteFile.**

- The manual-reset event is created and initialized to a nonsignaled state, and the handle to this event is placed in **hEvent**.

- The **CreateEvent** function can be used to create this manual-reset event as described earlier.

- After the I/O is initiated, the code is put into a loop during which additional foreground processing is performed.

- The function **GetOverlappedResult** is called continuously. The function returns the results of an overlapped I/O operation on the port.

- The function will return **FALSE** as long as the Asynchronous I/O is pending.

- A **TRUE** return value is used in the code to terminate the loop since it indicates that the I/O is complete.

- The loop will also have to be terminated if the timeout threshold has been exceeded.

- When sending large amounts of data, the transmission can be suspended without affecting the contents of the transmit buffer.

- The **SetCommBreak** function suspends character transmission for a specified communications device and places the transmission line in a break state until the **ClearCommBreak** function is called.

  BOOL SetCommBreak (HANDLE hCommDev);

  hCommDev

  The serial port handle returned by the **CreateFile**.

- The **ClearCommBreak** function restores character transmission for a specified communications device and places the transmission line in a nonbreak state.

  BOOL ClearCommBreak (HANDLE hFile);

  hCommDev

  The serial port handle returned by the **CreateFile**.

- Top priority data can be sent using The TransmitCommChar function which transmits a specified character ahead of any pending data in the output buffer of the serial port.

  BOOL TransmitCommChar (HANDLE hCommDev, char chTransmit);

hCommDev

The serial port handle returned by the **CreateFile**.

chTransmit

Specifies the character to be transmitted.

- The **TransmitCommChar** function is useful for sending an **interrupt** character (such as a **CTRL+C**) to a host system.

Event-Driven I/O

- This is the most flexible method of implementing communications programs.
- The idea is to have Win32 notify your application when certain events occur rather than having to check for these events.
- The **GetCommMask** function retrieves the value of the event mask that indicates those events that will be reported to your application.

BOOL GetCommMask (HANDLE hCommDev, LPDWORD lpEvtMask);

hCommDev

The serial port handle returned by the **CreateFile**.

lpEvtMask

Points to the 32-bit variable to be filled with a mask of events that are currently enabled. This parameter can be one or more of the following values:

Value	Meaning
EV_BREAK	A break was detected on input.

EV_CTS The CTS (clear-to-send) signal changed state.

EV_DSR The DSR (data-set-ready) signal changed state.

EV_ERR A line-status error occurred. Line-status errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.

EV_EVENT1 An event of the first provider-specific type occurred.

EV_EVENT2 An event of the second provider-specific type occurred.

EV_PERR A printer error occurred.

EV_RING A ring indicator was detected.

EV_RLSD The RLSD (receive-line-signal-detect) signal changed state.

EV_RX80FULL The receive buffer is 80 percent full.

EV_RXCHAR A character was received and placed in the input buffer.

EV_RXFLAG The event character was received and placed in the input buffer. The event character is specified in the device's DCB structure, which is applied to a serial port by using the SetCommState function.

EV_TXEMPTY The last character in the output buffer was sent.

- The SetCommMask function is used to add to or modify the set of events to be monitored for a communications device.

BOOL SetCommMask (HANDLE hCommDev, DWORD fdwEvtMask);

hCommDev

The serial port handle returned by the **CreateFile**.

fdwEvtMask

Specifies the events to be enabled. A value of zero disables all events. This parameter can be a combination of the same values as those for the GetCommMask function.

- For example, to set up an event when a character is received and placed in receive buffer we can use:

  SetCommMask (hCommDev, EV_RXCHAR);

- After the events of interest have been specified, the **WaitCommEvent** function is called to have the application notification of those events.

- The **WaitCommEvent** function waits for an event to occur for a specified communications device:

  BOOL WaitCommEvent (HANDLE hCommDev, LPDWORD lpfdwEvtMask, LPOVERLAPPED lpo);

  hCommDev

  The serial port handle returned by the **CreateFile**.

  lpfdwEvtMask

  Points to a 32-bit variable that receives a mask indicating the type of event that occurred.

  lpo

  Points to an **OVERLAPPED** structure. This parameter is ignored if the **hCommDev** handle was opened without specifying the **FILE_FLAG_OVERLAPPED** flag. If an overlapped operation is not desired, this parameter can be **NULL**.