Win32 Communications API

- The **Win32 API** provides a series of specialized functions for accessing and manipulating the serial ports.

- There are several functions available for communicating through the serial and parallel ports.

- The **Win32c** also provides two structures: **COMPROP** & **DCB** (**Device Control Block**) that are used specifically for communications related code.

Opening a Serial port

- Win32 treats serial ports and all other devices as files.

- This means that devices can be manipulated (opened, read/write, close) just like ordinary files.

The CreateFile() Function

- This function will open a serial port and give the calling function exclusive or shared access to the port:

```
HANDLE CreateFile (LPCTSTR  lpFileName,
                        DWORD  dwDesiredAccess,
                        DWORD  dwShareMode,
                        LPSECURITY_ATTRIBUTES   lpSecurityAttributes,
                        DWORD  dwCreationDistribution,
                        DWORD dwFlagsAndAttributes,
                        HANDLE  hTemplateFile);
```

- The function returns a positive integer descriptor that identifies that port and can be used by other communications routines to access the port.

- If the call is unsuccessful, the return value is **INVALID_HANDLE_VALUE.** To get extended error information, call **GetLastError**. a negative integer is returned which can be used to determine the type of error.

- **lpFileName**

  - Points to a null-terminated string that specifies the name of the port, COM1, COM2, etc.

- **dwDesiredAccess**

  - Specifies the type of access to the file or other object. An application can obtain read access, write access, read-write access, or device query access. Both **GENERIC_READ** and **GENERIC_WRITE** must be set to obtain read-write access:

    dwDesiredAccess = GENERIC_READ | GENERIC_WRITE

- **dwShareMode**

  - Specifies how this file can be shared. For serial ports it is set to 0.

- **lpSecurityAttributes**

  - Points to a **SECURITY_ATTRIBUTES** structure that specifies the security attributes for the file. Set to NULL to assign the default security attributes to the port.

- **dwCreationDistribution**

  - Specifies which action to take on files that already exist, and which action to take when files do not exist. Since serial ports always exist it is set to **OPEN_EXISTING** (i.e., don't create but open an existing port).

- **dwFlagsAndAttributes**

  - Specifies the file attributes and flags for the file. For serial ports we use **FILE_FLAG_OVERLAPPED** which specifies Asynchronous I/O.

- **hTemplateFile**

  - Specifies a handle with **GENERIC_READ** access to a template file. Not used with serial ports so set to NULL.

Initializing a Serial port

- The next step is to initialize the port by using the **SetupComm()** to allocate transmit and receive buffers if necessary.

- If the program does not set them, the device uses the default parameters when the first call to another communications function occurs.

- This function initializes the communications parameters for a specified communications device.

```
BOOL SetupComm (HANDLE  hCommDev, DWORD  cbInQueue,
        DWORD cbOutQueue);
```

- If the function succeeds, the return value is **TRUE**.

- If the function fails, the return value is **FALSE**. To get extended error information, call **GetLastError**.

- **hCommDev**
  - Identifies the communications device. The **CreateFile** function returns this handle.

- **cbInQueue**
  - Specifies the recommended size, in bytes, of the port's receive buffer.

- **cbOutQueue**
  - Specifies the recommended size, in bytes, of the port's transmit buffer.

- Note that the buffer sizes are only recommended sizes, Windows may allocate differently.

Configuring a Serial Port

- Win32 provides a number of data structures that integrate serial ports and modems.

- The **COMMPROP** structure provides information on the port's capabilities that can then be used in conjunction with another structure **(DCB**) to configure the port.

The COMMPROP structure

```
typedef struct _COMMPROP {

        WORD  wPacketLength;       // packet size, in bytes
        WORD  wPacketVersion;      // packet version
        DWORD dwServiceMask;       // services implemented
        DWORD dwReserved1;         // reserved
        DWORD dwMaxTxQueue;        // max Tx bufsize, in bytes
        DWORD dwMaxRxQueue;        // max Rx bufsize, in bytes
        DWORD dwMaxBaud;           // max baud rate, in bps
        DWORD dwProvSubType;       // specific provider type
        DWORD dwProvCapabilities;  // capabilities supported
        DWORD dwSettableParams;    // changeable parameters
        DWORD dwSettableBaud;      // allowable baud rates
        WORD  wSettableData;       // allowable byte sizes
        WORD  wSettableStopParity; // stop bits/parity allowed
        DWORD dwCurrentTxQueue;    // Tx buffer size, in bytes
        DWORD dwCurrentRxQueue;    // Rx buffer size, in bytes
        DWORD dwProvSpec1;         // provider-specific data
        DWORD dwProvSpec2;         // provider-specific data
        WCHAR wcProvChar[1];       // provider-specific data
} COMMPROP;
```

- The **GetCommProperties()** function call is used to fill in the structure with the serial port information:

```
BOOL GetCommProperties (HANDLE hCommDev, LPCOMMPROP
        lpCommProp);
```

- If the function succeeds, the return value is **TRUE.**

- If the function fails, the return value is **FALSE.** To get extended error information, call **GetLastError.**

- **hCommDev**

    - Identifies the port handle. The **CreateFile** function returns this handle.

- **lpCommProp**

    - Points to a **COMMPROP** structure in which the communications properties information is returned.

    - This information can be used in subsequent calls to the **SetCommState**, **SetCommTimeouts**, or **SetupComm** function to configure the communications device.

The DCB structure

- The associated structure is the **Device Control Block** (**DCB**). The **DCB** structure is standard mechanism for setting the operating parameters of a serial port.

- The **DCB** Structure for Win32:

```
typedef struct _DCB {

   DWORD DCBlength;           // sizeof(DCB)
   DWORD BaudRate;            // current baud rate
   DWORD fBinary: 1;          // binary mode, no EOF check
   DWORD fParity: 1;          // enable parity checking
   DWORD fOutxCtsFlow:1;      // CTS output flow control
   DWORD fOutxDsrFlow:1;      // DSR output flow control
   DWORD fDtrControl:2;       // DTR flow control type
   DWORD fDsrSensitivity:1;   // DSR sensitivity
   DWORD fTXContinueOnXoff:1; // XOFF continues Tx
   DWORD fOutX: 1;        // XON/XOFF out flow control
   DWORD fInX: 1;         // XON/XOFF in flow control
   DWORD fErrorChar: 1;   // enable error replacement
   DWORD fNull: 1;        // enable null stripping
   DWORD fRtsControl:2;   // RTS flow control
   DWORD fAbortOnError:1; // abort reads/writes on error
   DWORD fDummy2:17;      // reserved
   WORD wReserved;        // not currently used
   WORD XonLim;           // transmit XON threshold
   WORD XoffLim;          // transmit XOFF threshold
   BYTE ByteSize;         // number of bits/byte, 4-8
   BYTE Parity;           // 0-4=no,odd,even,mark,space
   BYTE StopBits;         // 0,1,2 = 1, 1.5, 2
   char XonChar;          // Tx and Rx XON character
   char XoffChar;         // Tx and Rx XOFF character
   char ErrorChar;        // error replacement character
   char EofChar;          // end of input character
   char EvtChar;          // received event character
   WORD wReserved1;       // reserved; do not use
} DCB;
```

Changing Port Settings

- The first step is to read the current **DCB** settings for the port using **GetCommState()** function:

  BOOL GetCommState (HANDLE hCommDev, LPDCB lpDCB);

- **hCommDev**

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

- **lpDCB**

    - Points to the **DCB** structure in which the control settings information is returned.

- The next step is to write the contents of the new **DCB** structure using the **SetCommState()** function:

  BOOL SetCommState (HANDLE hCommDev, LPDCB lpdcb);

- **hCommDev**

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

- **lpdcb**

    - Points to a DCB structure containing the configuration information for the specified communications device.

- Note that it is always a good idea to ensure that the new settings specified by the user can be supported by the serial port by comparing with the allowable settings in the **COMMPROP** structure.

Changing Common Settings

- The **BuildCommDCB()** function is a convenient method for changing the most common port settings:

```
BOOL BuildCommDCB (LPCTSTR szSettings, LPDCB  lpDCB);
```

- **szSettings**
  - Pointer to a null-terminated string that specifies device-control information.

- **lpDCB**
  - Pointer to a **DCB** structure to be filled

- The following example uses the **BuildCommDCB** function to set a port **9600 bps**, **no parity**, **8 data bits**, and **1 stop bit**:

```
err = BuildCommDCB("96,N,8,1", &mydcb);
```

Time-out Settings

- Time-outs are very important in communications programming because they provide a mechanism for ensuring that a program does not "hang" when an unexpected event occurs or an expected event does not occur when sending and receiving data.

- The **COMMTIMEOUTS** structure is used to specify how long a read or write function waits before giving up:

```
typedef struct _COMMTIMEOUTS {

        DWORD ReadIntervalTimeout;
        DWORD ReadTotalTimeoutMultiplier;
        DWORD ReadTotalTimeoutConstant;
        DWORD WriteTotalTimeoutMultiplier;
        DWORD WriteTotalTimeoutConstant;
} COMMTIMEOUTS,*LPCOMMTIMEOUTS;
```

- **ReadIntervalTimeout**

  - Specifies the maximum time, in milliseconds, allowed to elapse between the arrival of two characters on the communications line.

- **ReadTotalTimeoutMultiplier**

  - Specifies the multiplier, in milliseconds, used to calculate the total time-out

  - period for read operations. For each read operation, this value is multiplied by the requested number of bytes to be read.

- **ReadTotalTimeoutConstant**

  - Specifies the constant, in milliseconds, used to calculate the total time-out period for read operations.

  - For each read operation, this value is added to the product of the **ReadTotalTimeoutMultiplier** member and the requested number of bytes.

- **WriteTotalTimeoutMultiplier**

  - Specifies the multiplier, in milliseconds, used to calculate the total time-out period for write operations. For each write operation, this value is multiplied by the number of bytes to be written.

- **WriteTotalTimeoutConstant**

  - Specifies the constant, in milliseconds, used to calculate the total time-out period for write operations.

  - For each write operation, this value is added to the product of the **WriteTotalTimeoutMultiplier** member and the number of bytes to be written.

- Once the structure has been created and initialized to the required values, the **GetCommTimeouts()** and **SetCommTimeouts()** functions are used to implement the time-out settings.

- The **GetCommTimeouts** function retrieves the time-out parameters for all read and write operations on a specified port:

```
BOOL GetCommTimeouts (HANDLE hCommDev, LPCOMMTIMEOUTS
                lpCommTimeouts);
```

- **hCommDev**

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

- **lpCommTimeouts**
  - Points to a **COMMTIMEOUTS** structure in which the time-out information is returned.

- The **SetCommTimeouts** function implements the time-out parameters for all read and write operations on a serial port.

```
BOOL SetCommTimeouts (HANDLE  hCommDev, LPCOMMTIMEOUTS
lpctmo);
```

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

- **Lpctmo**
  - Points to a **COMMTIMEOUTS** structure that contains the new time-out values.

Control Commands

- Sometimes it is necessary to control individual hardware signals on the serial port.
- The **EscapeCommFunction** function directs a specified communications device to perform an extended function.

```
BOOL EscapeCommFunction (HANDLE  hCommDev, DWORD  dwFunc);
```

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

- **dwFunc**
- Specifies the code of the extended function to perform. This parameter can be one of the following values:

Value	Meaning
CLRDTR	Clears the DTR (data-terminal-ready) signal.
CLRRTS	Clears the RTS (request-to-send) signal.
SETDTR	Sends the DTR (data-terminal-ready) signal.
SETRTS	Sends the RTS (request-to-send) signal.

SETXOFF Causes transmission to act as if an XOFF character has been received.

SETXON Causes transmission to act as if an XON character has been received.

SETBREAK Suspends character transmission and places the transmission line in a break state until the ClearCommBreak function is called. Identical to the SetCommBreak function.

CLRBREAK Restores character transmission and places the transmission line in a nonbreak state. Identical to the ClearCommBreak function.

Serial Port I/O

Reading From the Serial Port

- In the simplest case a loop can 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).

- A serious drawback to using ReadFile in a loop (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.

- Later on we shall see how we can improve this using techniques such as multithreading and fully Event Driven I/O.

Writing to the Serial Port

- The techniques are the same as those for reading. The **WriteFile** function is used to send data out the serial port:

```
BOOL WriteFile (HANDLE hCommDev,
                LPCVOID  lpBuffer,
                DWORD nNumberOfBytesToWrite,
                LPDWORD  lpNumberOfBytesWritten,
                LPOVERLAPPED  lpOverlapped);
```

- **hCommDev**

  - The serial port handle returned by the **CreateFile** (created with **GENERIC_WRITE** access to the file function).

- **lpBuffer**

  - Points to the buffer containing the data to be written to the port.

- **nNumberOfBytesToWrite**

  - Specifies the number of bytes to write to the port.

- **lpNumberOfBytesWritten**

  - Points to the number of bytes actually written by this function call. WriteFile sets this value to zero before doing any work or error checking.

- **lpOverlapped**

  - Points to an **OVERLAPPED** structure. This structure is required if **hCommDev** was opened with **FILE_FLAG_OVERLAPPED**.

Closing a Serial port

- The serial port must be closed upon program termination so that other applications can use it.

The CloseHandle() Function

- This function closes the specified communications device and frees any memory allocated for the device's transmission and receiving queues.

- All characters in the output queue are sent before the communications device is closed.

```
BOOL CloseHandle (HANDLE hCommDev);
```

- **hCommDev**
  - Identifies an open port handle.