About Communications

The file input and output (I/O) functions (**CreateFile**, **CloseHandle**, **ReadFile**, **ReadFileEx**, **WriteFile**, and **WriteFileEx**) provide the basic interface for opening and closing a communications resource handle and for performing read and write operations. The Microsoft® Win32® API also includes a set of communications functions that provide access to communications resources. This overview describes the use of file I/O and communications functions, which enable applications to perform the following tasks:

- Open a handle to a specified communications resource.
- Set and query the configuration of a serial communications resource.
- Read from or write to a serial communications resource.
- Monitor a specified set of events that might occur for a given serial communications resource.
- Send a control command to the device driver associated with a specified communications resource, causing the driver to execute an extended function.

Communications Resource Handles

A process uses the **CreateFile** function to open a handle to a communications resource. For example, specifying COM1 opens a handle to a serial port, and LPT1 opens a handle to a parallel port. If the specified resource is currently being used by another process, **CreateFile** fails. Any thread of the process can use the handle returned by **CreateFile** to identify the resource in any of the functions that access the resource.

When using **CreateFile** to open a handle directly to a device, an application must use the special charact " \\ .\" to identify the device. For example, to open a handle to drive A, specify " \\ .\a:" for the *lpszName* parameter of **CreateFile**. The calling process can use the handle in the **DeviceIoControl** function to send control codes to the device.

When the process calls **CreateFile** to open a communications resource, it specifies the following attribut

- What type of read-write access exists for the specified resource.
- Whether the handle can be inherited by child processes.
- Whether the handle can be used in overlapped (asynchronous) I/O operations. (For a description of overlapped operations, see Synchronization.)

When the process uses **CreateFile** to open a communications resource, it must specify certain values for the following parameters:

- The *fdwShareMode* parameter must be zero, opening the resource for exclusive access.
- The *fdwCreate* parameter must specify the OPEN_EXISTING flag.
- The *hTemplateFile* parameter must be NULL.

Modification of Communications Resource Settings

When the **CreateFile** function opens a handle to a serial communications resource, the system initializes and configures the resource according to the values set up the last time the resource was opened. Preserving the previous settings enables the user to retain the settings specified through an **mode** comma when the device is reopened. The values inherited from the previous open operation include the configuration settings of the device control block (a **DCB** structure) and the time-out values used in I/O

operations. If the device has never been opened, it is configured with the system defaults.

To determine the initial configuration of a serial communications resource, a process calls the **GetCommState** function, which fills in a serial port **DCB** structure with the current configuration setting To modify this configuration, a process specifies a **DCB** structure in a call to the **SetCommState** functio

Members of the **DCB** structure specify the configuration settings such as the baud rate, the number of da bits per byte, and the number of stop bits per byte. Other **DCB** members specify special characters and enable parity checking and flow control. When a process needs to modify only a few of these configurati settings, it should first call **GetCommState** to fill in a **DCB** structure with the current configuration. The the process can adjust the important values in the **DCB** structure and reconfigure the device by calling **SetCommState** and specifying the modified **DCB** structure. This procedure ensures that the unmodified members of the **DCB** structure contain appropriate values. For example, a common error is to configure a device with a **DCB** structure in which the structure's **XonChar** member is equal to the **XoffChar** membe

The **BuildCommDCB** function provides another way to modify a **DCB** structure. **BuildCommDCB** use a string with the same form as the command-line arguments of the **mode** command to specify the baud rate, parity scheme, number of stop bits, and number of data bits. The remaining members of **DCB** are nc changed by this function, except that the appropriate members are set to disable XON/XOFF and hardwa flow control. **BuildCommDCB** only modifies a **DCB** structure; it does not reconfigure the device.

A process can reconfigure a communications resource by using the **GetCommProperties** function to get information from a device driver about the configuration settings that it supports. The process can use thi information to avoid specifying a configuration that is not supported.

The **SetCommState** function reconfigures the communications resource, but it does not affect the interna output and input buffers of the specified driver. The buffers are not flushed, and pending read and write operations are not terminated prematurely.

A process reinitializes a communications resource by using the **SetupComm** function, which performs th following tasks:

- Terminates pending read and write operations, even if they have not been completed.
- Discards unread characters and frees the internal output and input buffers of the driver associated with the specified resource.
- Reallocates the internal output and input buffers.

A process is not required to call **SetupComm**. If it does not, the resource's driver initializes the device wi the default settings the first time that the communications resource handle is used.

Communications Resource Configuration

The **COMMCONFIG** structure defines the configuration of a communications resource, serial or otherwise. The format of the structure varies depending on the type of communications resource (the provider subtype). The first few structure members are common to all communications resources; additional members are defined for specific provider subtypes. Specific service providers may extend the **COMMCONFIG** structure as well.

An application can get and set the configuration of a communications resource by using the **GetCommConfig** and **SetCommConfig** functions. When opened, a communications resource is initializ using the default configuration for its provider subtype. To get and set the default configuration for a provider subtype, use the **GetDefaultCommConfig** and **SetDefaultCommConfig** functions.

To prompt the user for configuration information, use the **CommConfigDialog** function. This function

displays a dialog box defined by the service provider and fills in a **COMMCONFIG** structure based on user input.

Modem Configuration

Modem configuration functions enable you to configure a modem before making a connection. An application can set modem options and determine the features of a modem without using commands specific to any modem device. Following are the general features an application may set before making a call:

- Primary mode of operation (synchronous, asynchronous, and whether error control is enabled).
- V.42 error control (defined by CCITT recommendation V.42), including specific parameters. CCIT stands for the International Telegraph and Telephone Consultative Committee.
- V.42bis (defined by CCITT recommendation V.42bis) and MNP5 data compression.
- Time-out options, including call setup, inactivity, and buffered data delivery.

Before setting a modem's configuration, an application should determine the capabilities of the modem device by using the **GetCommProperties** function. This function fills in a **COMMPROP** structure. This structure contains both a general portion, which applies to all communications devices, and a portion that specific to each provider subtype. For modem devices, the provider-specific portion of the **COMMPRO** structure is a **MODEMDEVCAPS** structure.

An application can get and set the current configuration of a modem by using the **GetCommConfig** and **SetCommConfig** functions, both of which use a **COMMCONFIG** structure. This structure contains bot a general portion, which applies to all communications devices, and a portion that is specific to each provider subtype. For modem devices, the provider-specific portion of the **COMMCONFIG** structure is **MODEMSETTINGS** structure.

After configuring a modem, an application can use the Telephony Application Programming Interface (TAPI) to actually establish a connection.

The modem configuration functions do not provide for long-term management and maintenance of a modem. Modem service providers should supply modem configuration dialog boxes for this purpose.

Read and Write Operations

The Win32 API supports both synchronous and asynchronous (overlapped) file I/O operations on serial communications resources. Overlapped operations enable the calling thread to perform other tasks while the operation executes in the background. A thread uses the **ReadFile** or **ReadFileEx** function to read from a communications resource, and the **WriteFile** or **WriteFileEx** function to write to a communications resource. **ReadFile** and **WriteFile** can be performed synchronously or asynchronously. **ReadFileEx** and **WriteFileEx** can only be performed asynchronously.

The behavior of these read and write functions is affected by whether the function is executed as an overlapped operation, whether the time-out parameters are associated with the handle, and whether flow control parameters are associated with the handle.

A thread can also write to a communications resource by using the **TransmitCommChar** function, whic transmits a specified character ahead of any pending data in the output buffer. This function is useful for transmitting a high priority signal character to the receiving system. Transmission of the high priority character is still subject to flow control and write time-outs, and the operation is performed synchronousl

A thread can use the **PurgeComm** function to discard all characters in a device's output or input buffer. **PurgeComm** can also terminate pending read or write operations, even if the operations have not been completed. If a thread uses **PurgeComm** to flush an output buffer, the deleted characters are not transmitted. To empty the output buffer while ensuring that the contents are transmitted, a thread can call the **FlushFileBuffers** function (a synchronous operation). Note, however, that **FlushFileBuffers** is subje to flow control but not to write time-outs, and it will not return until all pending write operations have be transmitted.

Overlapped Operations

Overlapped operations enable a thread to execute a time-consuming I/O operation in the background, leaving the thread free to perform other tasks. To enable overlapped I/O operations on a communications resource, the thread must specify the FILE_FLAG_OVERLAPPED flag in the **CreateFile** function when the handle is opened. To execute the **ReadFile** or **WriteFile** function as an overlapped operation, the calling thread must specify a pointer to an **OVERLAPPED** structure.The **OVERLAPPED** structure mus contain a handle to a manual-reset (not an auto-reset) event object. The system sets the state of the event object to not-signaled when a call to the I/O function returns before the operation has been completed. Th system sets the state of the event object to signaled when the operation has been completed. The thread uses a wait function to check the current state of the event object or to wait for its state to be signaled.

The **ReadFileEx** and **WriteFileEx** functions can be performed only as overlapped operations. The callin thread specifies a pointer to the **FileIOCompletionRoutine** function, which is executed when the overlapped operation is completed. The completion routine is executed only if the calling thread perform an alertable operation.

For more information about event objects, wait functions, alertable waits, and completion routines, see Synchronization.

Time-Outs

A handle to a communications resource has an associated set of time-out parameters that affect the behavior of read and write operations. Time-outs can cause a **ReadFile**, **ReadFileEx**, **WriteFile**, or **WriteFileEx** operation to conclude when a time-out interval elapses, even though the specified number c characters have not been read or written. It is not treated as an error when a time-out occurs during a read or write operation (that is, the read or write function's return value indicates success). The count of bytes actually read or written is reported by **ReadFile** or **WriteFile** (or by the **GetOverlappedResult** or **FileIOCompletionRoutine** function, if the I/O was performed as an overlapped operation).

When an application opens a communications resource, the system sets the resource's time-out values to the values in effect when the resource was last used. If the communications resource has never been opened, the system sets the time-out values to some default value. In either case, an application should always determine the current time-out values after opening the resource, and then explicitly set them to meet its requirements. To determine the current time-out values of a communications resource, use the **GetCommTimeouts** function. To change the time-out values, use the **SetCommTimeouts** function.

Two types of time-outs are enabled by the time-out parameters. An interval time-out occurs when the tim between the receipt of any two characters exceeds a specified number of milliseconds. Timing starts whe the first character is received and is restarted when each new character is received. A total time-out occur when the total amount of time consumed by a read operation exceeds a calculated number of millisecond Timing starts immediately when the I/O operation begins. Write operations support only total time-outs. Read operations support both interval and total time-outs, which can be used separately or combined.

The time, in milliseconds, of the total time-out period for a read or write operation is calculated by using the multiplier and constant values from the **COMMTIMEOUTS** structure specified in the **GetCommTimeouts** or **SetCommTimeouts** function. The following formula is used:

```
Timeout = (MULTIPLIER * number_of_bytes) + CONSTANT
```

Using both a multiplier and a constant enables the total time-out period to vary, depending on the amoun of data being requested. An application can use only the constant by setting the multiplier to zero, or use only the multiplier by setting the constant to zero. If both the constant and multiplier are zero, total time-out is not used.

If all read time-out parameters are zero, read time-outs are not used, and a read operation is not complete until the requested number of bytes have been read or an error occurs. Similarly, if all write time-out parameters are zero, a write operation is not completed until the requested number of bytes have been written or an error occurs.

If the read interval time-out parameter is the MAXDWORD value and both read total time-out parameter are zero, a read operation is completed immediately after reading whatever characters are available in the input buffer, even if it is empty.

Interval timing forces a read operation to return when there is a lull in reception. A process using interval time-outs can set a fairly short interval parameter, so it can respond quickly to small, isolated bursts of or or a few characters, yet it can still collect large buffers of characters with a single call when data is receiv in a steady stream.

Time-outs for a write operation can be useful when transmission is blocked by some kind of flow control when the **SetCommBreak** function has been called to suspend character transmission.

The following table summarizes the behavior of read operations based on the values specified for total ar interval time-outs.

Total	Interval	Behavior
0	0	Returns when the buffer is completely filled. Time-outs are not used.
T	0	Returns when the buffer is completely filled or when T milliseconds have elapsed since the beginning of the operation.
0	Y	Returns when the buffer is completely filled or when Y milliseconds have elapsed between the receipt of any two characters. Timing does not begin until the first character is received.
T	Y	Returns when the buffer is completely filled or when either type of time-o occurs.

Note, however, that timing is relative to the system controlling the physical device. For a remote device such as a modem, the timing is relative to the server system to which the modem is attached. Any networ propagation delay is not factored in. For example, a client application might specify a total time-out that computes to be 500 milliseconds. When 500 milliseconds have elapsed at the server, a time-out error is returned to the client. If there is a 50 milliseconds network propagation delay, the client will not be notifi of the time-out until approximately 50 milliseconds after the time-out actually occurred.

The time-out parameters affect the behavior of overlapped read and write operations on a communicatior device. With overlapped I/O, the **ReadFile**, **WriteFile**, **ReadFileEx**, or **WriteFileEx** function can return before the operation has been completed. The time-out parameters can determine when the operation has been completed.

Communications Errors

There are other circumstances where a read or write operation can be completed with fewer than the requested number of characters, even though a time-out has not occurred. Following are some examples:

- Some drivers support the use of special characters, which complete a read operation immediately with only the characters that have been read up to the point when they are received.
- The **PurgeComm** function can be called to prematurely terminate pending read or write operations This function can also delete the contents of the output or input buffers, or both.
- If a communications error occurs during a read or write operation, all I/O operations on the communications resource are terminated. Break conditions, parity errors, or framing errors are examples of such errors. When an error occurs, the process must call the **ClearCommError** function to clear the error flag before it can begin additional I/O operations. **ClearCommError** reports the specific error that occurred and the current status of the device.

Communications Events

A process can monitor a set of events that occur in a communications resource. For example, an application can use event monitoring to determine when the CTS (clear-to-send) and DSR (data-set-ready signals change state.

A process can monitor events on a given communications resource by using the **SetCommMask** functio to create an event mask. To determine the current event mask for a communications resource, a process can use the **GetCommMask** function. The following values specify events that can be monitored.

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_RING	A ring indicator was detected.
EV_RLSD	The RLSD (receive-line-signal-detect) signal changed state.
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 seria port by using the **SetCommState** function.
EV_TXEMPTY	The last character in the output buffer was sent.

After a set of events is specified, a process uses the **WaitCommEvent** function to wait for one of the events to occur. **WaitCommEvent** can be used synchronously or as an overlapped operation. For additional information about executing a function as an overlapped operation, see Synchronization.

When one of the events specified in the event mask occurs, the process completes the wait operation and sets an event mask variable to indicate the type of event detected. If the **SetCommMask** is called for a communications resource while a wait is pending for that resource, **WaitCommEvent** returns an error.

The **WaitCommEvent** function detects events that have occurred since the last call to **SetCommMask** o **WaitCommEvent**. For example, if you specify the EV_RXCHAR event as a wait-satisfying event, a call to **WaitCommEvent** will be satisfied if there are characters in the driver's input buffer that have arrived since the last call to **WaitCommEvent** or **SetCommMask**. Thus, given the following pseudo-code, any characters received between T1 and T2 will satisfy the next call to **WaitCommEvent**.

```
while (we_care)
{
    WaitCommEvent(args)

T1: // Read bytes
    // Process bytes

T2:
}
```

When monitoring an event that occurs when a signal (CTS, DSR, and so on) changes state, **WaitCommEvent** reports the change, but not the current state. To query the current state of the CTS (clear-to-send), DSR (data-set-ready), RLSD (receive-line-signal-detect), and ring indicator signals, a process can use the **GetCommModemStatus** function.

Extended Functions

Some communications functions can be called for a device by using the **EscapeCommFunction** functio This function sends a code to direct the device to execute an extended function. For example, an application can suspend character transmission with the SETBREAK code and resume transmission with the CLRBREAK code. These particular operations can also be started by calling the **SetCommBreak** an **ClearCommBreak** functions. **EscapeCommFunction** can also be used to implement manual modem control. For example, the CLRDTR and SETDTR codes can be used to implement manual DTR (data-terminal-ready) flow control. Note, however, that an error occurs if a process uses **EscapeCommFunctio** to manipulate the DTR line when the device has been configured to enable DTR handshaking, or the RTS (request-to-send) line if RTS handshaking is enabled.

The **DeviceIoControl** function enables a process to send an extended function code directly to a specifie device driver, causing the device to perform a given operation. **DeviceIoControl** gives a device associate with a communications resource capabilities not supported by the standard serial communications functions. It enables an application to configure a device using parameters unique to that device as well a to call any device-specific functions.