Virtual Serial Port
Applications Programming Interface
Programmers Guide and Reference

Applicable to Version 2.31, and beyond of the Virtual Serial Port Framework for Windows XP, Windows 2000, Windows NT, Windows 98Se, and Millennium

Constellation Data Systems, Inc.
www.VirtualPeripherals.com

Table of Contents

1. Introduction

The Virtual Serial Port framework (VSP) is a product of Constellation Data Systems, Inc (CDS). The VSP is a development accelerator, which can cut months or years from a development project, which requires a virtualized Serial or Communications resource.

1.1 What's in This Manual

This manual describes the Applications Programming Interface (API) of the Virtual Serial Port framework.

1.2 Audience

This literature is for use by the programmer who wishes to develop software, which interfaces with a Virtual Serial Port. It is assumed that the reader is a skilled C/C++ programmer, with a basic understanding of Windows programming, and serial communications in the Windows environment.

Applications, which support the VSP API, are simply called "VSP Applications". Examples of some sample VSP Applications distributed by CDS include such useful reference designs as "Virtual To Virtual", "Virtual To Physical", and "Multi Virtual To Physical". You may wish to use one of these frameworks as a starting point for your development.

Additionally, purchasers of the Network Serial Port (NSP) SDK have access to a suite of VSP compliant reference designs, which have powerful network data transmission capabilities.

1.3 Limitations

Use of this software, information, or technology in a system, or as a component of a system, which can through action or inaction, cause damage to life, limb, property, or the environment is not authorized. Use of this software is also subject to the terms and conditions of your properly executed Software License Agreement(s) with CDS.

2. Communications Resources in Windows

Communications resources are typically physical devices (with associated device drivers) that provide a single bi-directional, asynchronous data stream. Serial ports, parallel ports, fax machines, and modems are examples of physical communications resources.

The VSP resembles a standard Windows *communications resource*. However, rather than having physical hardware which sends and receives data, the VSP has a software implementation of that functionality.

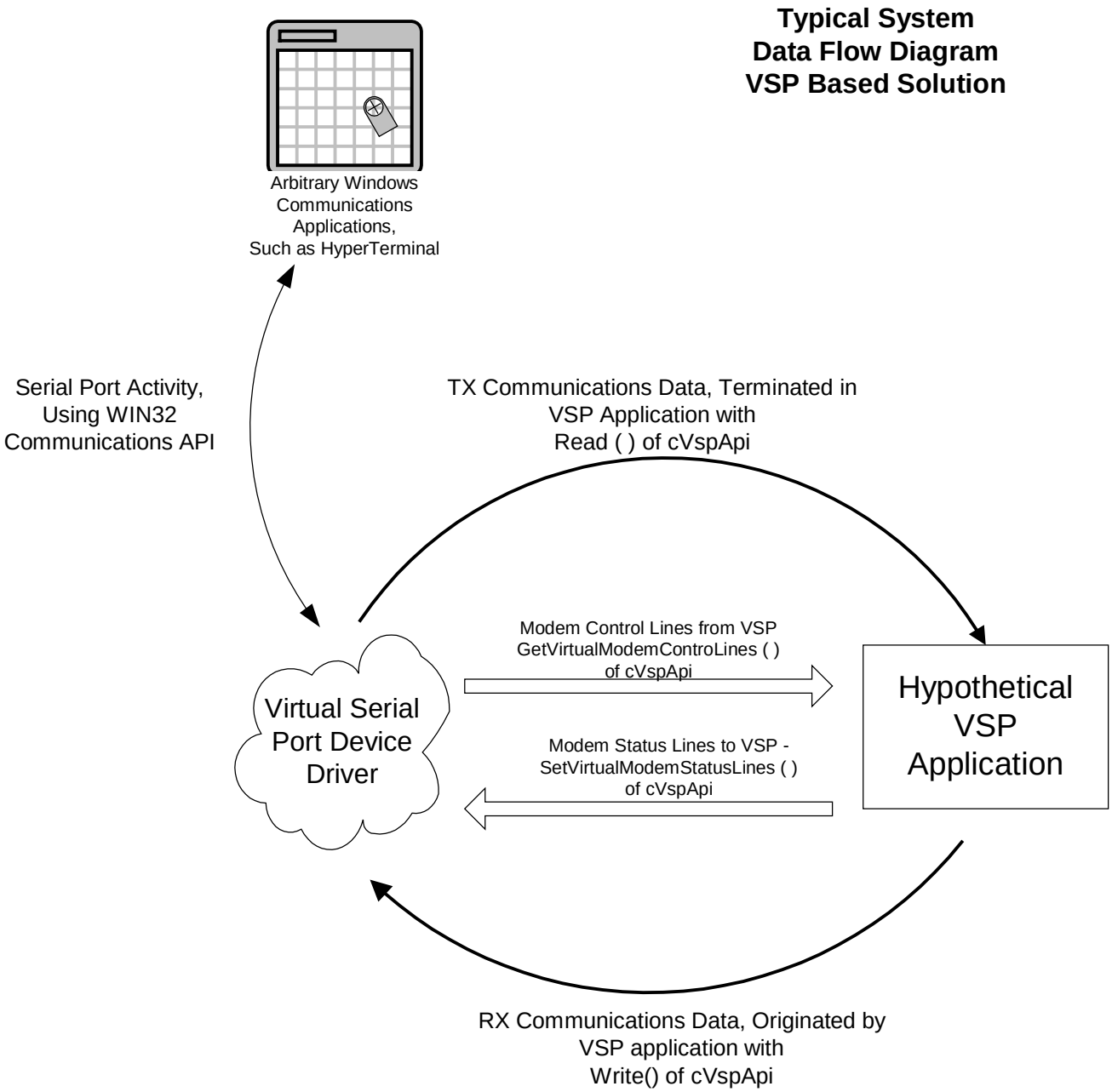
Most Windows applications, which use Microsoft's WIN32 Communications API (see section 7.1) to access *Communications Resources,* can use the VSP in place of physical communications resources. The data transfers to and from the VSP are available to you though the *VSP API* (see section 5).

Most modern communications applications use the WIN32 or WIN16 Communications API to access both physical communications resources. The VSP models those resources. Older DOS based communications applications built around hardware direct access is unsupported by the VSP framework.

Using the VSP framework, WIN16 and WIN32 based applications can "believe" that the communications data being sent and received is from a Physical Communications Resource (PCR), when in fact it is being controlled from another application reading and writing that data using the VSP API.

Important Point:	Using VSP framework TX communications data can be read (terminated) by applications through the VSP API, and RX communications data can be written (originated) by applications from the VSP API.

Consider the following data flow diagram, which illustrates a typical communications application, (HyperTerminal in this case), and a VSP implementation which originates and terminates communications data.



3. Binding the VSP API to Your Application

3.1 Introduction – C++ Class form vs. ANSI C form

The VSP API is implemented in two forms: using a C++ class interface, as well as a conventional interface using ANSI C style name decoration. It is suggested that C/C++ programmers should use the C++ class interface described Section 5; VSP API Reference (C++ Class Form).

Program interfaces using languages other that C++, such as Visual Basic, or Borland Delphi should consider using the interface described in Section 6; VSP API Reference (ANSI C Form).

3.2 Name Decoration in C++ ("mangling") vs. ANSI C

Name decoration usually refers to C++ naming conventions, but can apply to a number of C cases as well. By default, C++ uses the function name, parameters, and return type to create a *Linker Symbol Name* for the function. This process in C++ is often also called "name mangling.

Consider the *Linker Symbol Name* generated by the Open ( ) Function of cVspApi in both the C++ vs. the ANSI C form:

Form	**Function Name from vspapi.h**	**Linker Symbol Name**
C++	Open ( )	?Open@cVspApi@@QAEHPAD@Z
ANSI C	cVspApiOpen ( )	_cVspApiOpen

Clearly the *Linker Symbol Name* is a much simpler form. In fact, it is simply the function name prefixed by an underscore. Clearly, to simplify *Linker Symbol Naming,* the ANSI C form of the VSP interface should be used by programmers in environments other than the C or C++ languages.

Other than the ANSI C form (underscore prefaced function naming), there is currently no other standard for C++ naming between compiler vendors or even between different versions of a compiler. Therefore linking object files compiled with other compilers may not produce the same naming scheme and thus causes unresolved externals. **When in doubt use the ANSI C form.**

3.3 Source Code Include ("VspApi.h")

The VSP API class interface is stored in "VspApi.h". Applications typically use the following include identifies the interface:

```
#include "..\VspApi\VspApi.h"
```

3.4 The “VspApi.dll” File

At run time the “VspApi.dll” must be found at the time the VSP Application is started. It is recommended that “VspApi.dll” reside in the same directory as the VSP Application’s executable.

3.5 Binding “VspApi.lib” File

In order for the DLL to bind to the application, the “VspApi.lib” file must be included in the link sequence. For that reason, VSP Applications include “VspApi.lib” in the applications resources. Consider the following screen capture from the Microsoft Visual C/C++ Developer Studio:

4. VSP API Member Function Operation

This section describes the context in which VSP member functions operate. Basically, all member functions can be broken down into two categories: Pre-Open member functions and Post-Open member functions. The distinguishing factor would be opening the VSP via the Open() function. As the names imply, the Pre-Open member functions can be used before the VSP has been opened, as well as after. The Post-Open member functions can only be used while the VSP is open.

4.1 Pre-Open Member Functions

Open()
DllVersion()
DriverVersion()
AddSerialPort()
DeleteSerialPort()
IsVirtualPort()
GetOpenCount()
SetDeviceOptions()
GetDeviceOptions()

4.2 Post-Open Member Functions

Close()
Read()
Write()
SetTimeouts()
SetVirtualModemStatusLines()
GetVirtualModemControlLines()
GetBufferStatus()
WaitForChanges()
GetDcb()
SetReadFileTiming()
GetReadFileTiming()
SetWriteFileTiming()
GetWriteFileTiming()

5. VSP API Reference (C++ Class Form)

5.1 Overview of the VSP API

As established in section 2, the VSP API allows TX communications data to be read (terminated) and, RX communications data to be written (originated). Applications, which support the VSP API, are simply called “VSP Applications”.

Examples of some sample VSP Application frameworks, which are distributed with the VSP SDK include “Virtual To Virtual”, “Virtual To Physical”, “Multi Virtual To Physical”. You may wish to use one of these frameworks as a starting point for your development.

Additionally, purchasers of the Network Serial Port (NSP) SDK have access to a suite of VSP compliant reference designs, which have powerful network data transmission capabilities.

VSP Applications are WIN32 compliant applications, and use the cVspApi Class (see section 5.2) to access the VSP.

5.2 The cVspApi Class

This class encapsulates lower level driver and system interface techniques into an environment, which is both simple and powerful.

5.2.1 Constructor / Destructors

The constructor and destructor have the following forms:

```
cVspApi(void);
~cVspApi(void);
```

Constructing an instance of the VSP simply prepares the underlying application data structures. It does not prepare a VSP, or communicate with a VSP, or an underlying driver until an Open operation is performed.

5.2.2 Open ( ) Function of cVspApi

Open function prepares the VSP and application space data structures for use. Ports may be simultaneously open by both communications applications (HyperTerminal for example) using the WIN32 communications API, and by VSP applications using the VSP API.

Prototype

*int Open (char * FileName);*

Parameters

FileName – Name assigned to the VSP when installed.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

5.2.3 Close ( ) Function of cVspApi

Close function releases a Virtual Serial Port previously opened.

Prototype

int Close ( );

Parameters

None.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”. Close requests to ports, which are not “open” succeed, and return a value of zero.

5.2.4 Read ( ) Function of cVspApi

This function reads data from a VSP. Data read from a VSP is TX data generated by communications applications such as HyperTerminal. If the number of bytes requested to be read is available in the VSP's "transmit buffer", then the data is returned immediately. If the number of bytes requested is not available, then this function blocks, and the data read is subject to timeouts setup by *SetTimeouts ( ).*

Hint	Use VSPAPI function *GetBufferStatus ( )* to determine how many bytes are available to be read. This allows you to design implementations which return data immediately.

Prototype

*int Read (UCHAR *pBuff, int BytesRequested, ULONG * pBytesRead);*

Parameters

pBuff	Pointer to buffer which will receive data read from the VSP.
BytesRequested	The number of bytes which are requested to be read. This value should never exceed the allocated size of the data at "*pBuff".*
pBytesRead	Returns number of bytes read and places at "*pBuff".*

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: "winerror.h".

Also See

SetTimeouts ( )

5.2.5 Write ( ) Function of cVspApi

This function writes data to a VSP. Data written to a VSP is considered RX data by communications applications such as HyperTerminal.

Prototype

int Write (UCHAR **pBuff*, int *SizeofToWrite*,
ULONG * *pBytesWritten*);

Parameters

pBuff	Pointer to buffer which contains the data to be written to the VSP.
SizeofWrite	The number of bytes to be written to the VSP.
pBytesWritten	Returns number of bytes actually written.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

SetTimeouts ( )

5.2.6 GetBufferStatus ( ) Function of cVspApi

This function returns the VSP's buffer status information.

Hint	Try use VSPAPI function *GetChanges ( )*, to observe when TX data has been written into the VSP buffers, and then use *GetBufferStatus ( )* to determine how many bytes have been written.

Prototype

int GetBufferStatus (VSP_BUFFER_STATUS **pVspBufferStatus*);

Parameters

pVspBufferStatus — Pointer to buffer which receives the buffer status information. The VSP_BUFFER_STATUS structure consists of the following fields:

```
DWORD   SizeofTxBuffer;
DWORD   SizeofRxBuffer;
DWORD   BytesUsedTxBuffer;
DWORD   BytesUsedRxBuffer;
```

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: "winerror.h".

Also See

Read ( )

5.2.7 GetVirtualModemControlLines ( ) Function of cVspApi

This function reads the instantaneous values of the modem control lines, as recorded in the VSP at the time the function is called. The modem control lines are changed by Communications Applications, such as HyperTerminal, using the WIN32 Communications API. Using this function, those changes can then be observed by VSP applications using the VSP API.

The modem control lines are *Data Terminal Ready* (DTR), *Request To Send* (RTS). Also returned is an indication if a BREAK signal has been asserted on the *Transmit Data* (TD) line. In a non-virtualized device (such as an RS-232 port), these signals (DTR, RTS, and TD) originate from the DTE (PC) side of the port.

Hint	To immediately process changes in Modem Control Lines, use VSPAPI function *WaitForChanges ( )*, to determine when virtual modem control, and be able to process them immediately.

Prototype

int GetVirtualModemControlLines (ULONG *pModemControl)

Parameters

PModemControl Pointer to ULONG which will receive a bit mask of the modem control line values. The bit mask may be analyzed using the following bit mask constants:

VSP_MC_DTR_ASSERTED
VSP_MC_RTS_ASSERTED
VSP_MC_BREAK_ASSERTED

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

WaitForChanges ( )

5.2.8 GetOpenCount ( ) Function of cVspApi

This function reads the number of applications which are connected to the target Virtual Serial Port.

Hint	To immediately process changes Open/Close status changes (rather than polling for changes), use VSPAPI function *WaitForChanges ( )*, to determine when then count changes. Then, depending upon the status returned, issue the *GetOpenCount ( )*.

Prototype

int GetOpenCount (char *pPortName, ULONG *pOpenCount)

Parameters

pOpenCount	Pointer to ULONG which will the count of applications which are holding the target port open.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

WaitForChanges ( )

5.2.9 AddSerialPort ( ) Function of cVspApi

This function dynamically adds a Virtual Serial Port.

Prototype

int AddSerialPort (char *pPortName)

Parameters

pPortName	Pointer to ASCII string which contains name of port to be added.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

DeleteSerialPort ( ), IsVirtualPort ( )

Important **Win98Se/Me**	COM1 – COM4 feature robust implementations in the PC peripheral architecture. COM5 – COM99, however, are not legacy devices and their implementations are nebulous.

5.2.10 DeleteSerialPort ( ) Function of cVspApi

This function dynamically deletes and existing Virtual Serial Port.

Prototype

int DeleteSerialPort (char *pPortName)

Parameters

pPortName	Pointer to ASCII string which contains name of the virtual serial port to be deleted.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

AddSerialPort ( ), IsVirtualPort ( )

5.2.11 IsVirtualPort ( ) Function of cVspApi

This function returns and indication of whether a serial port is a Virtual Serial Port.

Prototype

int IsVirtualPort (char *pPortName, BOOL pbIsVirtualPort)

Parameters

pPortName	Pointer to ASCII string which contains name of port to be examined.
pbIsVirtualPort	Pointer to a BOOLEAN, which upon successful return will contain an indication of whether the port is a Virtual Serial Port.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

DeleteSerialPort ( ), AddSerialPort ( )

5.2.12 WaitForChanges ( ) Function of cVspApi

This function waits for certain changes to occur in the Virtual Serial Port. Please take note that the changes observed by this function are controlled by Communications Applications, such as HyperTerminal, which are connected to the corresponding Virtual Serial Port. This function blocks until one of the supported changes is observed.

Only one thread may be waiting using *WaitForChanges ( ).* Should a thread be blocked on this function and a second thread issues a *WaitForChanges ( ),* then the first thread's operation will be cancelled and the second thread shall block on *WaitForChanges ( ).*

To manually release a thread which is waiting, from a second thread issue the *WaitForChanges ( ),* passing in the VSP_EVENT_CANCEL bit using the **pChangeMask* parameter. In this scenario the first thread will be cancelled, and the second thread will return immediately (without blocking).

Hint	Implement a thread which blocks on *WaitForChanges ( ).* Then process the corresponding changes in the context of that thread. If on the other hand, you desire to process changes in the context of another thread, consider using WIN32 synchronization objects such as *SetEvent ( )*, *WaitForSingleObject ( )* or *WaitForMultipleObjects ( )* to reflect those changes to another thread..

Prototype

int WaitForChanges (ULONG *pChangeMask);

Parameters

pChangeMask	Pointer to ULONG which will receive a bit mask of the change indications observed which differ from the previous (last) values has been observed.

The preceding values (bit masks) may be analyzed using the following bit mask constants:

VSP_EVENT_MCL_CHANGE	A Modem Control Line has been changed.

VSP_EVENT_PURGE_RX	The Receive Buffer has been purged.
VSP_EVENT_PURGE_TX	The Transmit Buffer has been purged.
VSP_EVENT_DCB_CHANGE	The Device Control Block (DCB) has been changed.
VSP_EVENT_RX_EMPTY	The Receive buffer has been read to the point that it was emptied.
VSP_EVENT_TX_NOT_EMPTY	Data has been placed in the Transmit buffer, which was empty before the data was placed in the buffer.
VSP_EVENT_TX_WRITE	Data has been placed in the Transmit buffer.
VSP_EVENT_OPEN_CLOSE	The Virtual Serial Port has either been opened or closed. Also see: *GetOpenCount ( ).*
VSP_EVENT_CANCEL	On return from *WaitForChanges ( )* operation, this bit indicates that the operation has been cancelled by another thread. Note: Should this bit be passed into *WaitForChanges ( ),* then any another thread, which may blocked on *WaitForChanges ( ),* is immediately cancelled. In this scenario the calling thread (which issued the cancel), does not block (it returns immediately).

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

GetDcb ( )
GetOpenCount ( )
GetVirtualModemControlLines ( )

5.2.13 SetVirtualModemStatusLines ( ) Function of cVspApi

This function allows VSP applications to set virtual modem status lines in a Virtual Serial Port. The modem status lines thus setup, can then be accessed by Communications Applications (such as HyperTerminal), using the WIN32 Communications API.

The modem status lines are *Clear To Send* (CTS)*, Data Set Ready* (DSR)*, Ring Indicate* (RING or RI)*, and Receive Line Signal Detect* (RLSD or CD). Note: the *Carrier Detect* (CD) signal is often referred to by Microsoft as *Receive Line Signal Detect* (RLSD). In a non-virtualized device (such as an RS-232 port), these signals originate from the DCE (modem) side of the port.

Prototype

int SetVirtualModemStatusLines (ULONG ModemStatus);

Parameters

ModemStatus	A ULONG value which contains a bit mask of the modem status line values being set into the VSP. The following bit map constants (defined by the WIN32 Communications API), may be used in conjunction with this field: MS_CTS_ON MS_DSR_ON MS_RING_ON MS_RLSD_ON

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

GetVirtualModemControlLines ( )

5.2.14 SetDeviceOptions ( ) Function of cVspApi

This function allows VSP applications to set device options of the target virtual port.

Prototype

int SetDeviceOptions (ULONG DeviceOptions);

Parameters

DeviceOptions	A ULONG value which contains a bit mask of a variety of device options, as described below.

Device Option Descriptions

VSP_DO_SINGLE_WRITE	Device option which sets-up the VSP to return control to the calling application only after the data written by the WIN32 Communications API function "*WriteFile*()" has cleared the device. While this is how Microsoft's Serial Port implementation functions, it will affect throughput through the VSP framework in very high throughput implementations. The tradeoff here is performance vs. compatibility.
VSP_DO_FAST_WRITE	Device option which sets up the VSP to allow "*WriteFile*()" operations maximum use of the VSP buffering. The Microsoft Serial Port implementation generally returns control after the data written has been buffered, and OVERLAPPED operations generally branch toward use of OVERLAPPED techniques. The effect of setting this bit will be to cause the VSP framework to make maximum use of its internal buffering, and make minimum use of OVERLAPPED techniques. Well written applications, such as Hyperterminal, will function well in this mode. Other applications may internally

fault, or behave incorrectly when this option is set.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

GetVirtualModemControlLines ( )

5.2.15 GetDcb ( ) Function of cVspApi

This function reads the Device Control Block of the Virtual Serial Port.

Hint	To immediately process changes in the DCB, use VSPAPI function *WaitForChanges ( )*, to determine when virtual modem control, and be able to process them immediately.

Prototype

int GetDcb (DCB *pDcb)

Parameters

pDcb Pointer to Device Control Block.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

WaitForChanges ( )

5.2.16 DllVersion ( ) Function of cVspApi

This function returns the version number of the “VspApi.dll” file. In the VSP frameworks, it is often expected that the VspApi.dll be of the same version as the underlying VSP device driver.

Prototype

int DllVersion (ULONG **pVersion*);

Parameters

pVersion	Pointer to ULONG which receives the version number of the VspApi.dll file. The version number returned is multiplied by 100. For example, if a 108 is returned, the corresponding VspApi.dll is version “1.08”.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

DriverVersion ( )

5.2.17 DriverVersion ( ) Function of cVspApi

This function returns the version number of the underlying VSP device driver. In the VSP frameworks, it is often expected that the VspApi.dll be of the same version as the underlying VSP device driver.

Prototype

int DriverVersion (ULONG *pVersion);

Parameters

PVersion	Pointer to ULONG which receives the version number of the underlying VSP device driver. The version number returned is multiplied by 100. For example, if a 208 is returned, the corresponding device driver is version “2.08”.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

DllVersion ( )

5.2.18 SetTimeouts ( ) Function of cVspApi

This function sets the time-out parameters for all VSP read and write operations on a specified VSP.

Important:	VSP timeouts are different entity than WIN32 communications timeouts setup by *SetCommTimeouts ( ).* VSP timeouts setup by *SetTimeouts ( )* of cVspApi control VSP *Read ( )* and *Write* ( ) operations used by VSP applications. Whereas *SetCommTimeouts ( )* control *ReadFile ( )* and *WriteFile ( )* operations of communications applications such as HyperTerminal.

Prototype

int SetTimeouts (VSP_TIMEOUTS * *pTimeouts*);

Parameters

pTimeouts Pointer to a structure of type VSP_TIMEOUTS.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

Read ()
Write ()
VSP_TIMEOUTS

5.2.19 VSP_TIMEOUTS Structure used by cVspApi

The VSP_TIMEOUTS structure is used by *SetTimeouts ( )* of cVspApi. The parameters determine the behavior of *Read ( )* and *Write ( )* of cVspApi.

```
typedef struct {
  DWORD ReadIntervalTimeout;
  DWORD ReadTotalTimeoutMultiplier;
  DWORD ReadTotalTimeoutConstant;
  DWORD WriteTotalTimeoutMultiplier;
  DWORD WriteTotalTimeoutConstant;;
} VSP_TIMEOUTS;
```

Members

ReadIntervalTimeout

Specifies the maximum time, in milliseconds, allowed to elapse between the arrival of two characters on the communications line. During a **Read ( )** operation, the time period begins when the first character is received. If the interval between the arrival of any two characters exceeds this amount, the **Read ()** operation is completed and any buffered data is returned. A value of zero indicates that interval time-outs are not used.
A value of MAXDWORD, combined with zero values for both the **ReadTotalTimeoutConstant** and **ReadTotalTimeoutMultiplier** members, specifies that the read operation is to return immediately with the characters that have already been received, even if no characters have been received.

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.

A value of zero for both the **ReadTotalTimeoutMultiplier** and **ReadTotalTimeoutConstant** members indicates that total time-outs are not used for read operations.

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.

A value of zero for both the **WriteTotalTimeoutMultiplier** and **WriteTotalTimeoutConstant** members indicates that total time-outs are not used for write operations.

Remarks

If an application sets **ReadIntervalTimeout** and **ReadTotalTimeoutMultiplier** to MAXDWORD and sets **ReadTotalTimeoutConstant** to a value greater than zero and less than MAXDWORD, one of the following occurs when the **Read ( )** function is called:

If there are any characters in the input buffer, **Read ( )** returns immediately with the characters in the buffer.

If there are no characters in the input buffer, **Read ( )** waits until a character arrives and then returns immediately.

If no character arrives within the time specified by **ReadTotalTimeoutConstant**, **Read ( )** times out.

5.2.20 SetReadFileTiming ( ) Function of cVspApi

This function allows control over the rate of data delivery to connected serial port applications (such as HyperTerminal) when reading data from a Virtual Serial Port. A developer may use this function to control the timing of data read by serial port applications. In simple terms, this function allows the developer to "stall" a connected serial port application (such as HyperTerminal), at the time data is read from a serial port.

On a Physical Communications Device, such as a UART (e.g. PC's RS-232 serial port), the timing of data delivery is a function of the data ("baud") rate. In other words, the slower the baud rate, the slower the rate of data delivery. Of course the faster the baud rate the faster the rate of data delivery. Typically a Virtual Serial Port will deliver data at a rate far in excess of what is generally seen with a Physical Device. There are some serial port aware applications whose functionality depends upon the rate of data delivery. This function allows control over the rate of data delivery when those applications read data from the VSP.

Important:	The delay experienced by a connected serial port aware application when reading data from a VSP is: (Number Bytes Read * ByteToByteDelay) + ConstantDelay Delays experienced are processed by the Operating System in "quantums" (chunks of time) as multiples of generally 10 or 15 milli-seconds. Take for example a case where a delay of 28 milli-seconds is expected; 30 milli-seconds will probably be experienced. Note: The *SetReadFileTiming ( )* interface is supported under Windows 95/98/Millenium, however it has no affect on the timing in that environment.

Prototype

int *SetReadFileTiming* (VSP_RW_FILE_TIMING *ReadFileTiming*);

Parameters

ReadFileTiming	Structure of type VSP_RW_FILE_TIMING, see section 5.2.24. Delay values are specified in milli-seconds.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: "winerror.h".

Also See

GetReadFileTiming ( ), SetWriteFileTiming ( ), GetWriteFileTiming ( ), VSP_RW_FILE_TIMING Structure.

5.2.21 GetReadFileTiming ( ) Function of cVspApi

This function retrieves timing which may be set by VSP API function *SetReadFileTiming ()*. See section 5.2.20 for more information.

Prototype

int GetReadFileTiming(VSP_RW_FILE_TIMING **pReadFileTiming*);

Parameters

pReadFileTiming	Pointer to a structure of type VSP_RW_FILE_TIMING, see section 5.2.24.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

SetReadFileTiming ()
SetWriteFileTiming ()
GetWriteFileTiming()
VSP_RW_FILE_TIMING

5.2.22 SetWriteFileTiming ( ) Function of cVspApi

This function allows control over the rate of data delivery from a connected serial port applications (such as HyperTerminal) when writing data to a Virtual Serial Port. A developer may use this function to control the timing of data written by serial port applications. In simple terms, this function allows the developer to "stall" a connected serial port application (such as HyperTerminal), at the time data is written to a serial port.

On a Physical Communications Device, such as a UART (e.g. PC's RS-232 serial port), the timing of data delivery is a function of the data ("baud") rate. In other words, the slower the baud rate, the slower the rate of data delivery. Of course the faster the baud rate the faster the rate of data delivery. Typically a Virtual Serial Port can consume data at a rate far in excess of what is generally possible with a Physical Device. There are some serial port aware applications whose correct functionality depends upon the rate of data delivery. This function allows control over the rate of data delivery when those applications are writing data to a VSP.

Important:	The delay experienced by a connected serial port aware application when reading data from a VSP is: (Number Bytes Written * ByteToByteDelay) + ConstantDelay Delays experienced are processed by the Operating System in "quantums" (chunks of time) as multiples of generally 10 or 15 milli-seconds. Take for example a case where a delay of 28 milli-seconds is expected; 30 milli-seconds will probably be experienced. Note: The *SetWriteFileTiming ( )* interface is supported under Windows 95/98/Millenium, however it has no affect on the timing in that environment.

Prototype

int SetWriteFileTiming (VSP_RW_FILE_TIMING *WriteFileTiming*);

Parameters

WriteFileTiming Structure of type VSP_RW_FILE_TIMING, see section 5.2.24.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: "winerror.h".

Also See

SetReadFileTiming (), GetReadFileTiming (), GetWriteFileTiming(), VSP_RW_FILE_TIMING

5.2.23 GetWriteFileTiming ( ) Function of cVspApi

This function retrieves timing which may be set by VSP API function *SetWriteFileTiming ()*. See section 5.2.22 for more information.

Prototype

int GetWriteFileTiming (VSP_RW_FILE_TIMING **pWriteFileTiming*);

Parameters

pWriteFileTiming	Pointer to a structure of type VSP_RW_FILE_TIMING.

Return Values

If the function succeeds, the return value is zero. Other status returns are defined by the MS Platform SDK file: “winerror.h”.

Also See

SetReadFileTiming ()
GetReadFileTiming ()
SetWriteFileTiming()
VSP_RW_FILE_TIMING

5.2.24 VSP_RW_FILE_TIMING Structure used by cVspApi

The VSP_RW_FILE_TIMING structure is used by various file timing functions of cVspApi. This structure allows the customer to control the VSP framework in a manner that "Baud Rate Propagation Delays" can be controlled and simulated. The parameters determine the behavior of time delays between byte read and writes of a VSP.

```
typedef struct {
  ULONG   ByteToByteDelay;
  ULONG   ConstantDelay;
  } VSP_RW_FILE_TIMING;
```

Members

ByteToByteDelay

Specifies the amount of time, in milliseconds, to delay between bytes.

ConstantDelay

Specifies the a constant amount of time, in milliseconds, to delay between byte read and writes.

5.2.25 VSP_TIMEOUTS Structure used by cVspApi

The VSP_TIMEOUTS structure is used by *SetTimeouts ( )* of cVspApi. The parameters determine the behavior of *Read ( )* and *Write ( )* of cVspApi.

```
typedef struct {
  DWORD ReadIntervalTimeout;
  DWORD ReadTotalTimeoutMultiplier;
  DWORD ReadTotalTimeoutConstant;
  DWORD WriteTotalTimeoutMultiplier;
  DWORD WriteTotalTimeoutConstant;;
} VSP_TIMEOUTS;
```

Members

ReadIntervalTimeout

Specifies the maximum time, in milliseconds, elapse allowed between the arrival of two characters on the communications line. During a **Read ( )** operation, the time period begins when the first character is received. If the interval between the arrival of any two characters exceeds this amount, the **Read ()** operation is completed and any buffered data is returned. A value of zero indicates that interval time-outs are not used.
A value of MAXDWORD, combined with zero values for both the **ReadTotalTimeoutConstant** and **ReadTotalTimeoutMultiplier** members, specifies that the read operation is to return immediately with the characters that have already been received, even if no characters have been received.

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.

A value of zero for both the **ReadTotalTimeoutMultiplier** and **ReadTotalTimeoutConstant** members indicates that total time-outs are not used for read operations.

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.

A value of zero for both the **WriteTotalTimeoutMultiplier** and **WriteTotalTimeoutConstant** members indicates that total time-outs are not used for write operations.

Remarks

If an application sets **ReadIntervalTimeout** and **ReadTotalTimeoutMultiplier** to MAXDWORD and sets **ReadTotalTimeoutConstant** to a value greater than zero and less than MAXDWORD, one of the following occurs when the **Read ( )** function is called:

If there are any characters in the input buffer, **Read ( )** returns immediately with the characters in the buffer.

If there are no characters in the input buffer, **Read ( )** waits until a character arrives and then returns immediately.

If no character arrives within the time specified by **ReadTotalTimeoutConstant**, **Read ( )** times out.

6. VSP API Reference (ANSI C Form)

6.1 Overview

The VSP API implemented using the ANSI C form is, under the covers, simply a thin layer above the C++ form. Programmers should use the following programming techniques when using this form.

1. "Construct" the Port using *cVspApiConstruct ()*. The port handle is returned, be sure to save the port handle, as all other function of the ANSI C form of the VSP API require this handle. This function should be called once for each Virtual Port which will be accessed. Each Virtual Port is assigned a unique port handle.
2. Manipulate the port using calls to ANSI C functions of the VSP API that are desired; except *cVspApiDestruct ( )* . Be sure to use the correct Virtual Port handle. Note: *cVspApiOpen ( )* and *cVspApiClose ( )* may be called as appropriate.
3. Once access to the port is no longer desired, the port should be "Destructed" using *cVspApiDestruct ( )*.

6.2 Parameter Usage

The first parameter of all functions of the ANSI C form is the VSP port handle (returned by *cVspApiConstruct*). The remaining parameters of each function are the same as its corresponding C++ function. For example, a typical usage of the ANSI C VSP API function cVspApiRead ( ) is;

```
Status = cVspApiRead(hPort, Buff, sizeof(Buff) &BytesRead);
```

A corresponding use of the C++ VSP API function *Read ( )* is:

```
cVspApi hVspApi;
Status = hVspApi.Read(Buff, sizeof(Buff) &BytesRead);
```

Notice that the substantial difference between the two functions lies in the first parameter (the port handle) of the ANSI C form.

6.3 Function Enumeration

ANSI C Form	Comments
cVspApiConstruct	Call once before using any functions of the ANSI C form. Be sure to save the returned handle to the Virtual Port.
cVspApiDestruct	Call once after use of a virtual port is no longer desired. Device should be “Closed” at the time this function is called.
cVspApiOpen	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API Open() function.
cVspApiClose	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API Close() function.
cVspApiDllVersion	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API DllVersion() function.
cVspApiRead	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API Read() function.
cVspApiWrite	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API Write() function.
cVspApiDriverVersion	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API DriverVersion() function.
cVspApiSetTimeouts	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API SetTimeouts() function.
cVspApiSetVirtualMsl	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API SetVirtualModemStatusLines() function.
cVspApiGetVirtualMcl	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API GetVirtualModemControlLines() function.
cVspApiWaitForChanges	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API WaitForChanges() function.
cVspApiIsVirtualPort	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API IsVirtualPort() function.
cVspApiAddSerialPort	Use the port handle from cVspApiConstruct. For other parameters and more information, see the

	VSP API AddSerialPort() function.
cVspApiDeleteSerialPort	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API DeleteSerialPort() function.
cVspApiSetWriteFileDelay	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API SetWriteFileDelay () function.
cVspApiGetWriteFileDelay	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API GetWriteFileDelay () function.
cVspApiSetReadFileDelay	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API SetReadFileDelay () function.
cVspApiGetWriteFileDelay	Use the port handle from cVspApiConstruct. For other parameters and more information, see the VSP API GetReadFileDelay () function.

6.4 Linker Symbol Names

Linker Symbols Names for the ANSI C interface of the VSP API are simply the function name preceded by an underscore. The *Linker Symbol Name* may be necessary to connect programming environments such as Visual Basic.NET, or Delphi, to the VSP API.

Function Name	**Linker Symbol Name**
cVspApiConstruct ( ) cVspApiDestruct ( ) cVspApiOpen ( ) cVspApiClose ( ) cVspApiRead ( ) cVspApiWrite ( ) cVspApiDllVersion ( ) cVspApiDriverVersion ( ) cVspApiSetTimeouts ( ) cVspApiSetVirtualMsl ( ) cVspApiGetVirtualMcl ( ) cVspApiWaitForChanges ( ) cVspApiIsVirtualPort ( ) cVspApiAddSerialPort ( ) cVspApiDeleteSerialPort ( ) etc…	_cVspApiConstruct _cVspApiDestruct _cVspApiOpen _cVspApiClose _cVspApiRead _cVspApiWrite _cVspApiDllVersion _cVspApiDriverVersion _cVspApiSetTimeouts _cVspApiSetVirtualMsl _cVspApiGetVirtualMcl _cVspApiWaitForChanges _cVspApiIsVirtualPort _cVspApiAddSerialPort _cVspApiDeleteSerialPort etc …

7. WIN32 Communications Interfaces of the VSP

7.1 Overview of the WIN32 Communications API

The WIN32 Communications API is described in the MSDN Platform SDK base services documentation. Since this is an "industry standard" interface, a detailed description of this interface is beyond the scope of this document. However, the reader should be aware of the following base capabilities of this interface before proceeding.

In the WIN32 API, the file input and output (I/O) functions -- *CreateFile ( ), CloseHandle ( ) , ReadFile ( ), ReadFileEx ( ), WriteFile ( ),* and *WriteFileEx ( )* -- provide the basic functions for opening and closing a communications resource handle and for performing read and write operations. Additionally, the API provides a set of specific functions that provide access to communications resources.

The VSP framework manages both the basic and specific functions for any given Virtual Serial Port.

Communications Specific Functions of WIN32 API

The following WIN32 Communications Specific Functions are used with communications devices, and are simulated by the VSP. Please consult the MS WIN32 Programmers Reference for detailed programming information.

Function Name	Description
BuildCommDCB	Fills a specified **DCB** structure with values specified in a device-control string.
BuildCommDCBAndTimeouts	Translates a device-definition string into appropriate device-control block codes and places them into a device control block.
ClearCommBreak	Restores character transmission for a specified communications device and places the transmission line in a nonbreak state.
ClearCommError	Retrieves information about a communications error and reports the current status of a communications device.
CommConfigDialog	Displays a driver-supplied configuration dialog box.

EscapeCommFunction	Directs a specified communications device to perform an extended function.
GetCommConfig	Retrieves the current configuration of a communications device.
GetCommMask	Retrieves the value of the event mask for a specified communications device.
GetCommModemStatus	Retrieves modem control-register values.
GetCommProperties	Retrieves information about the communications properties for a specified communications device.
GetCommState	Retrieves the current control settings for a specified communications device.
GetCommTimeouts	Retrieves the time-out parameters for all read and write operations on a specified communications device.
GetDefaultCommConfig	Retrieves the default configuration for the specified communications device.
PurgeComm	Discards all characters from the output or input buffer of a specified communications resource.
SetCommBreak	Suspends character transmission for a specified communications device and places the transmission line in a break state.
SetCommConfig	Sets the current configuration of a communications device.
SetCommMask	Specifies a set of events to be monitored for a communications device.
SetCommState	Configures a communications device according to the specifications in a device-control block.
SetCommTimeouts	Sets the time-out parameters for all read and write operations on a specified communications device.
SetDefaultCommConfig	Sets the default configuration for a communications device.
SetupComm	Initializes the communications parameters for a specified communications device.
TransmitCommChar	Transmits a specified character

	ahead of any pending data in the output buffer of the specified communications device.
WaitCommEvent	Waits for an event to occur for a specified communications device.

7.2 The DCB Structure

The **DCB** structure defines the control setting for a serial communications device.

```
typedef struct _DCB {
 DWORD DCBlength;
  DWORD BaudRate;
  DWORD fBinary: 1;
  DWORD fParity: 1;
  DWORD fOutxCtsFlow:1;
  DWORD fOutxDsrFlow:1;
  DWORD fDtrControl:2;
  DWORD fDsrSensitivity:1;
  DWORD fTXContinueOnXoff:1;
  DWORD fOutX: 1;
  DWORD fInX: 1;
  DWORD fErrorChar: 1;
  DWORD fNull: 1;
  DWORD fRtsControl:2;
  DWORD fAbortOnError:1;
  DWORD fDummy2:17;
  WORD wReserved;
  WORD XonLim;
  WORD XoffLim;
  BYTE ByteSize;
  BYTE Parity;
  BYTE StopBits;
  char XonChar;
  char XoffChar;
  char ErrorChar;
  char EofChar;
  char EvtChar;
  WORD wReserved1;
 DCB;
```

Members

DCBlength

Length, in bytes, of the **DCB** structure.

BaudRate

Baud rate at which the communications device operates. This member can be an actual baud rate value, or one of the following indexes:

CBR_110
CBR_19200
CBR_300
CBR_38400
CBR_600
CBR_56000
CBR_1200
CBR_57600
CBR_2400
CBR_115200
CBR_4800
CBR_128000
CBR_9600
CBR_256000
CBR_14400

fBinary

Indicates whether binary mode is enabled. Windows does not support nonbinary mode transfers, so this member must be TRUE.

fParity

Indicates whether parity checking is enabled. If this member is TRUE, parity checking is performed and errors are reported.

fOutxCtsFlow

Indicates whether the CTS (clear-to-send) signal is monitored for output flow control. If this member is TRUE and CTS is turned off, output is suspended until CTS is sent again.

fOutxDsrFlow

Indicates whether the DSR (data-set-ready) signal is monitored for output flow control. If this member is TRUE and DSR is turned off, output is suspended until DSR is sent again.

fDtrControl

DTR (data-terminal-ready) flow control. This member can be one of the following values.

Value	**Meaning**
DTR_CONTROL_DISABLE	Disables the DTR line when the device is opened and leaves it disabled.
DTR_CONTROL_ENABLE	Enables the DTR line when the device is opened and leaves it on.
DTR_CONTROL_HANDSHAKE	Enables DTR handshaking. If handshaking is

enabled, it is an error for the application to adjust the line by using the **EscapeCommFunction** function.

fDsrSensitivity

Indicates 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

Indicates whether transmission stops when the input buffer is full and the driver has transmitted the **XoffChar** character. If this member is TRUE, transmission continues 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.

fOutX

Indicates whether XON/XOFF flow control is used during transmission. If this member is TRUE, transmission stops when the **XoffChar** character is received and starts again when the **XonChar** character is received.

fInX

Indicates whether XON/XOFF flow control is used during reception. If this 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.

fErrorChar

Indicates 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

Indicates whether null bytes are discarded. If this member is TRUE, null bytes are discarded when received.

fRtsControl

RTS (request-to-send) flow control. This member can be one of the following values.

Value	Meaning
RTS_CONTROL_DISABLE	Disables the RTS line when the device is opened and leaves it disabled.
RTS_CONTROL_ENABLE	Enables the RTS line when the device is opened and leaves it on.
RTS_CONTROL_HANDSHAKE	Enables RTS handshaking. The driver raises the RTS line when the "type-ahead" (input) buffer is less than one-half full and lowers the RTS line when the buffer is more than three-quarters full. If handshaking is enabled, it is an error for the application to adjust the line by using the **EscapeCommFunction** function.
RTS_CONTROL_TOGGLE	**Windows NT/2000/XP:** 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.

fAbortOnError

Indicates 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 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

Reserved; must be zero.

XonLim

Minimum number of bytes allowed in the input buffer before flow control is activated to inhibit the sender. Note that the sender may transmit characters after the flow control signal has been activated, so this value should never be zero. This assumes that either XON/XOFF, RTS, or DTR input flow control is specified in **fInX, fRtsControl,** or **fDtrControl**.

XoffLim

Maximum number of bytes allowed in the input buffer before flow control is activated to allow transmission by the sender. This assumes that either XON/XOFF, RTS, or DTR input flow control is specified in **fInX, fRtsControl,**

or **fDtrControl**. The maximum number of bytes allowed is calculated by subtracting this value from the size, in bytes, of the input buffer.

ByteSize

Number of bits in the bytes transmitted and received.

Parity

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
SPACEPARITY	Space

StopBits

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

Value of the XON character for both transmission and reception.

XoffChar

Value of the XOFF character for both transmission and reception.

ErrorChar

Value of the character used to replace bytes received with a parity error.

EofChar

Value of the character used to signal the end of data.

EvtChar

Value of the character used to signal an event.

wReserved1

Reserved; do not use.

Remarks

When a **DCB** structure is used to configure the 8250, the following restrictions apply to the values specified for the **ByteSize** and **StopBits** members:

- The number of data bits must be 5 to 8 bits.
- The use of 5 data bits with 2 stop bits is an invalid combination, as is 6, 7, or 8 data bits with 1.5 stop bits.

8. Index of Acronyms and Abbreviations

API	Applications Programming Interface
ASCII	American Standard Code for Information Interchange
AKA	Also Known As
BPS	Bits per Second (“baud”)
CD	Carrier Detect (modem status line)
CDS	Constellation Data Systems
CTS	Clear to Send (modem status line)
cVspApi	Virtual Serial Port API Class
DOS	Disk Operating System
DCE	Data Communications Equipment
DLL	Dynamic Link Library
DSR	Data Set Ready (modem status line)
DTE	Data Terminal Equipment
DTR	Data Terminal Ready (modem control line)
MS	Microsoft
MSDN	MS Developers Network
NSP	Network Serial Port
PC	Personal Computer
PCR	Physical Communications Resource (Such as a UART)
RI	Ring Indicate (modem status line)
RS-232	Recommended Standard 232 (from the Electronics Industry Association) for data communications
RLSD	Receive Line Signal Detect (modem status line) AKA Carrier Detect
RTS	Request To Send (modem control line)
RX	Receive
SDK	Systems Development Kit
HyperTerminal	Standard Windows Communications Application
TD	Transmit Data
TLA	Three Letter Acronym
TX	Transmit
UART	Universal Asynchronous Receiver / Transmitter
VSP	Virtual Serial Port
WIN16	Windows 16 Bit Programming Paradigm (Arguably Obsolete)
WIN32	Windows 32 Bit Programming Paradigm
XON	Transmit On
XOFF	Transmit Off