Virtual Serial Port
Software Development Kit
Programmers Guide and Reference

for Windows XP, Windows 2000, and Windows NT

Constellation Data Systems, Inc.
www.VirtualPeripherals.com

Table of Contents

1. Overview

1.1 Capabilities

The Virtual Serial Port (VSP) Software Development Kit (SDK) 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.

While the VSP Core has many powerful pre-developed solutions, it is often necessary that solutions be customized. The process of developing customized VSP solutions is enabled by the VSP SDK. Using the VSP SDK a programmer is capable of developing the following types of software implementations:

- Hardware-less serial port interface emulation
- Easy capture of data from a serial port data.
- Easy generation of data into a serial port.
- High speed data transfers / transmissions.
- Multiplexing multiple data sources on a single serial port.
- Splitting data from a single source onto multiple serial ports.
- Serial device simulation
- Serial port data redirection
- Serial port device simulation
- Serial protocol development and implementation
- Support serial port manipulations in “C/C++”, Visual Basic, MFC and other popular programming paradigms.
- Multiple Virtual Serial Port instances (no theoretical limits) per system.

… and …

- *Almost any imagined serial port manipulation.*

In short, the VSP SDK makes the serial port “impossibilities”, possible. Better yet, it allows the tasks to be implemented quickly and easily. You no longer have to be a serial port expert or device driver “guru” to develop sophisticated serial port system software!

1.2 Typical VSP Implementation

There is no “typical VSP implementation”. The flexibility of the VSP framework allows it to be used in almost endless variety of ways. Among them is GPS data replication, serial port data capture, serial port data reproduction, OEM data communications, telecommunications and data transmissions systems, system debuggers, and many, many more.

1.3 Development Environment

The Virtual Serial Port SDK features software developed in both Microsoft Visual Basic .NET and Microsoft Visual C/C++ (MSVC) Version 6.

The Microsoft Visual Basic .NET Reference Designs can not be accessed without having the .NET Framework installed on the target machine.

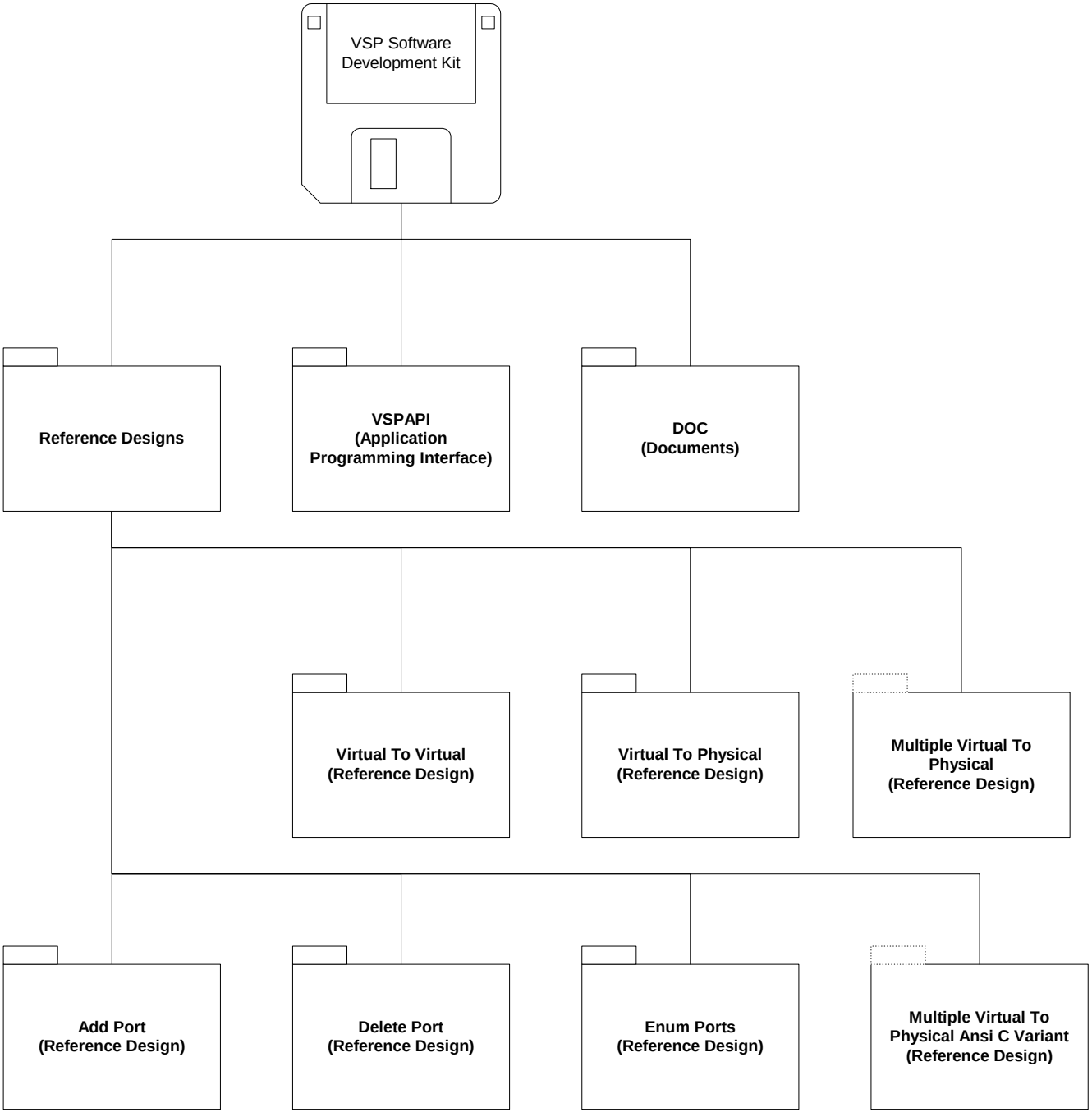
The latest MSVC service packs are also recommended. The reference design implementations are largely WIN32 examples using standard C/C++. To enhance transparency, object oriented ("OO") methods are used very infrequently.

2. SDK Contents

CDS's Virtual Serial Port SDK consists of the following major components.

Component	**Contains**
Reference Designs	Fully documented and working designs which allow data redirection from: Add Virtual Serial Port (VB.NET) Delete Virtual Serial Port (VB.NET) Virtual Serial Port to Physical Serial Port (VB.NET) Add/Create Virtual Serial Port (C/C++) Delete Virtual Serial Port (C/C++) Enumerate Virtual Serial Ports (C/C++) Virtual Serial Port to Physical Serial Port (C/C++) Virtual Serial Port to Virtual Serial Port (C/C++) Multiple Virtual Ports to Physical Serial Port (C/C++)
Applications Programming Interface	The Applications Programming Interface which allow VSP applications access to the heart of the framework.
Documentation	Applications Programming Reference Programmers Guide and Reference, Software Development Kit Programmers Guide and Reference

The SDK is organized as follows:



3. SDK Documentation

The VSP SDK documentation set consists of the following components:

- *Applications Programming Interface Programmers Guide and Reference* (of the Virtual Serial Port)
- *Software Development Kit Programmers Guide and Reference* (of the Virtual Serial Port) -- this document.

The following document, while not formally part of the VSP SDK documentation set, is useful for understanding the VSP framework:

- *Core Users Guide and Reference* (of the Virtual Serial Port)

4. Installation Instructions

Simply unzip the provided ZIP file into a folder of your choice. Be sure to preserve subfolders during the unzip process.

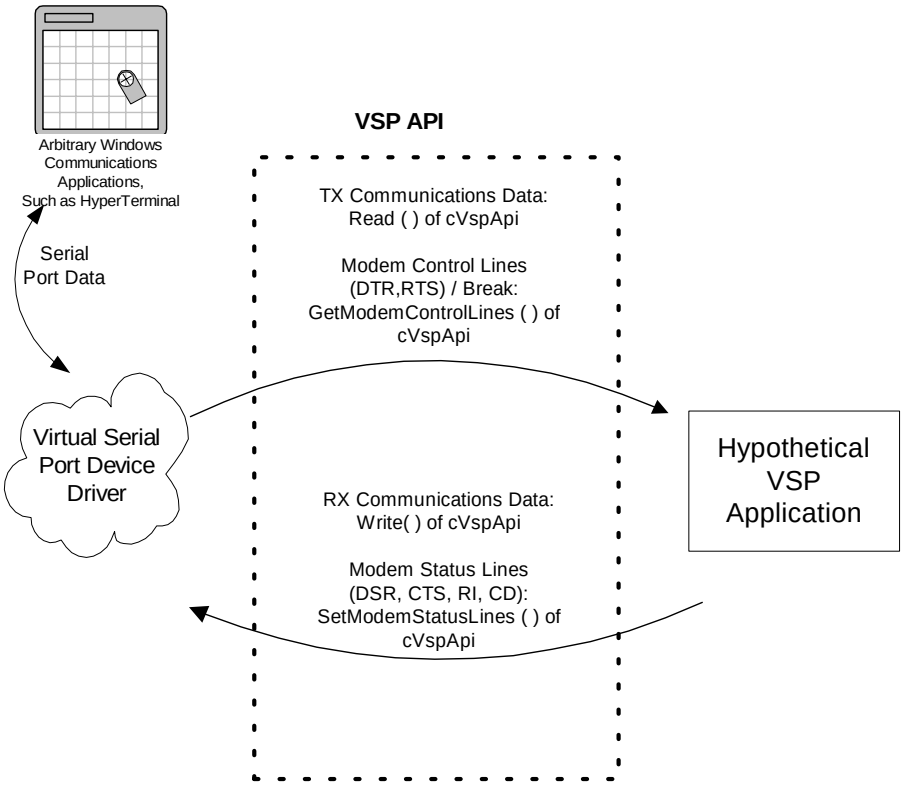
5. VSP Applications Programming Interface

The *VSP Applications Programming Interface* (VSPAPI) facilitates simulation of the hardware oriented functions of a serial port. What this means is that since the VSP has no physical hardware, a software component must take the place of the functions of the hardware. This allows you to create VSP applications, which are specific to your needs and requirements.

In most VSP applications, data, which would be transmitted by hardware in a physical serial port implementation, is “read” from a Virtual Serial Port (in a virtual implementation) using this API. In this manner, “transmit data” can be terminated in another component (a “VSP Application”). Similarly, data, which would be received by hardware in a physical serial port implementation, is “written” to the Virtual Serial Port (in a virtual implementation) using this API. In this manner “receive data” can originate from another component (a “VSP Application).

Consider the following data flow diagram, which illustrates a hypothetical VSP application, which originates and terminates communications data, modem status and control lines.

VSP Application DFD



While Constellation Data Systems (CDS) provides a number of pre-built VSP Applications (called “Reference Designs”), programmers and engineers will often find it necessary to write their own VSP Applications. The Reference Designs, documentation and VSPAPI enable this process. The net effect is that your custom requirements are quickly and easily implemented.

Software control of functions normally allocated to hardware is possible using the VSP Applications Programming Interface. These functions include:

- Software capture of transmit serial data using VSP API “*Read” function.*
- Software generation of receive serial data using VSP API *“Write”* function.
- Powerful control of data timing dynamics using VSP API *“SetTimeouts”* function.
- A variety of other data control, synchronization and versioning functions.

For more information on the VSP API, consult the *Applications Programming Interface Programmers Guide and Reference* Manual (of the Virtual Serial Port).

6. VSP Reference Designs

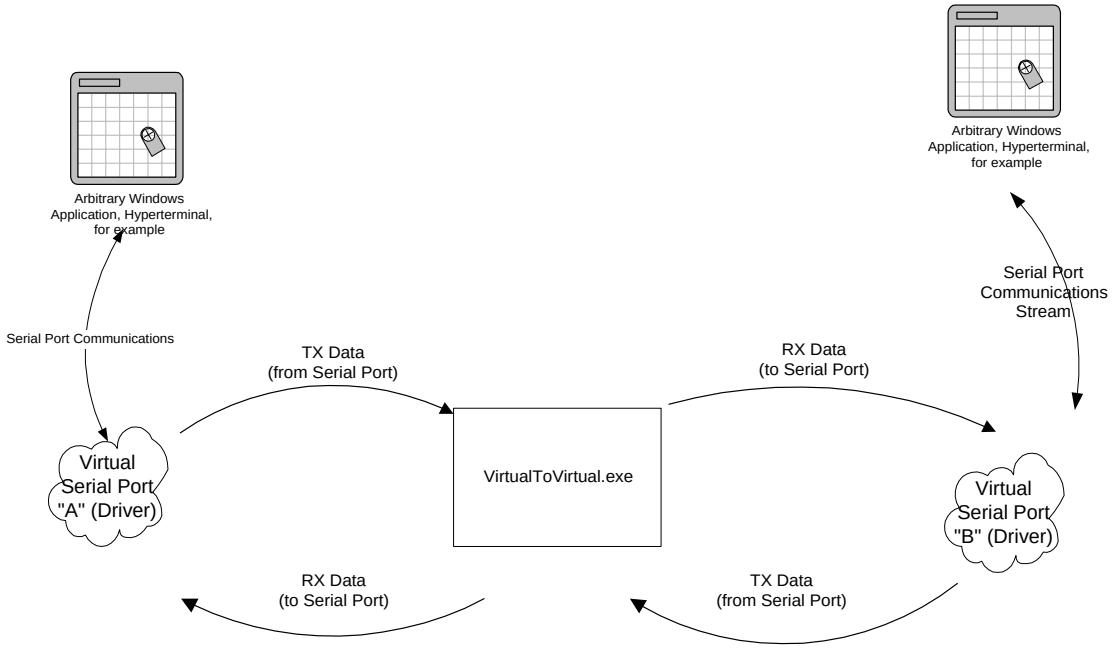
The VSP Product provides a number of Reference Designs in the Software Development Kit. Many OEM's and System Integrators are able to use these Reference Designs to solve real world problems. Often the designs can be used stand-alone, without modification. Where customization is required, the necessary information has been provided to enable that process.

6.1 Virtual to Virtual Serial Port Sample Reference Design (VB.NET)

The *Virtual to Virtual.NET* reference design demonstrates the techniques of taking data to and from a *Virtual Serial Port* and presenting that information to another *Virtual Serial Port* using the Visual Basic.NET development environment. An engineer may wish to use this sample as a reference design (starting point) for a data replicator, debugging interface, or data collection device. Consider the following "external" data flow diagram:

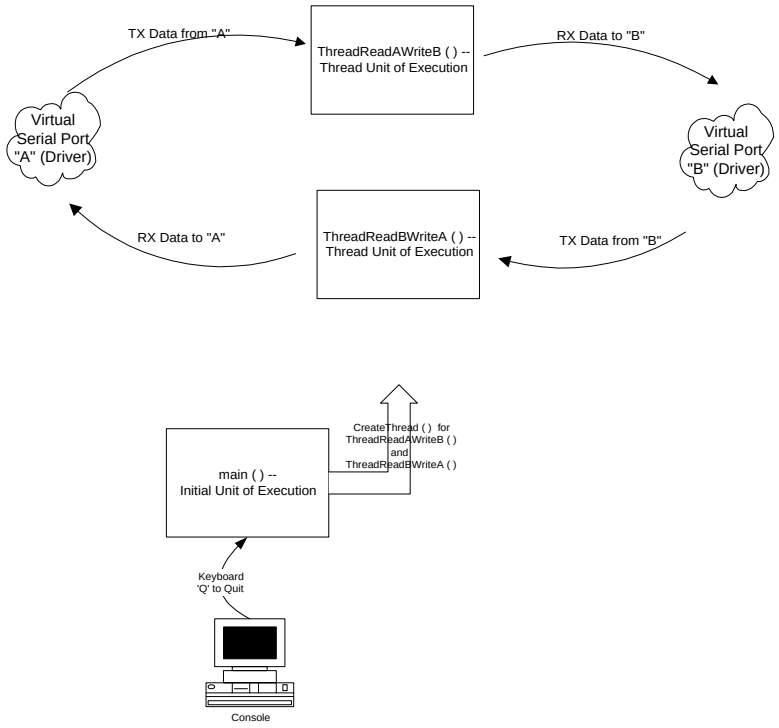
6.1.1 External Data Flow and Construction

Typical Virtual To Virtual Serial Port Data Flow



6.1.2 Internal Data Flow and Construction

The Internal construction of *VirtualToVirtual.NET* is that of a multi-thread application. The following data flow diagram illustrates the "internal" data flow of the *VirtualToVirtual.NET* reference design:



6.1.3 Module / Function Software Description

The primary source module is "VirtualToVirtual.vb", which consists of about 1465 lines of VB.NET code.

"VToVMod.vb" is a secondary module consisting of around 250 lines of VB.NET code where global variables, bit masks, various VSP structures, and VspApi.dll member function declarations are made.

6.1.4 Critical Function Descriptions

The following functions are critical.

Controlling Thread	Description
ThreadAToB.ThreadProc ( )	Reads data from virtual serial port A and writes that data to virtual serial port B.
ThreadBToA.ThreadProc ( )	Reads data from virtual serial port B and writes that data to virtual serial port A.
MclToMsl ( )	Connects the Modem Control Lines (Mcl) of source virtual serial port to destination virtual serial port.
VirtualToVirtual_Load ( )	Entry point; Constructs two virtual serial ports and does version verification.
cmdConnect_Click ( )	Opens handles to desired virtual serial ports and launches above threads.
cmdDisconnect_Click ( )	Closes connection between virtual serial ports and terminates threads.
cmdExit_Click ( )	Exits application.

6.1.5 Narrative from Entry Point

1. From the main entry point “VirtualToVirtual_Load ()”, two Virtual Serial Ports used by this reference design are constructed.

```
'*********************************
' Creates two Virtual Serial Ports.
'*********************************
Call cVspApiConstruct(hVspA)
Call cVspApiConstruct(hVspB)
```

2. A version number comparison for the Underlying Device Driver (for each virtual port), DLL version comparison, and the version of *VirtualToVirtual.NET* is performed. It is strongly suggested that the underlying VSP device driver, the VSP API DLL, and the application software (reference design, utility, or custom VSP application all conform to the same version). The following code fragment typifies the operation of version number validation:

```
'****************************************
' Gets and displays Dll Version information.
'****************************************
```

```
Status = cVspApiDllVersion(hVspA, DllVersion)
If (ERROR_SUCCESS <> Status) Then
    MsgBox("cVspApiDllVersion() failed - Status: " + Str$(Status),
    MsgBoxStyle.Critical + _
    MsgBoxStyle.OKOnly, "DLL Version Error")
    Exit Sub
End If

'*******************************************
' Gets and displays Driver Version information.
'*******************************************
Status = cVspApiDriverVersion(hVspA, DriverVersionA)
If (ERROR_SUCCESS <> Status) Then
    MsgBox("cVspApiDriverVersion() failed - Status: " + Str$(Status),
    MsgBoxStyle.Critical + _
    MsgBoxStyle.OKOnly, "VSP PortA Driver Version Error")
    Exit Sub
End If

Status = cVspApiDriverVersion(hVspA, DriverVersionB)
If (ERROR_SUCCESS <> Status) Then
    MsgBox("cVspApiDriverVersion() failed - Status: " + Str$(Status),
    MsgBoxStyle.Critical + _
    MsgBoxStyle.OKOnly, "VSP PortB Driver Version Error")
    Exit Sub
End If

'*******************************
' Validate the driver version(s).
'*******************************
dblDriverVersionA = DriverVersionA / 100
dblDriverVersionB = DriverVersionB / 100
dblDllVersion = DllVersion / 100
If ((DllVersion <> DriverVersionA) Or (DriverVersionA <>
DriverVersionB)) Then
    MsgBox(" PortA Driver Version: " + dblDriverVersionA + _
    " PortB Driver Version: " + dblDriverVersionB + _
    " Utiltiy DLL Version: " + dblDllVersion, MsgBoxStyle.Information +
_   MsgBoxStyle.OKOnly, "Version Mismatch")
    End
Else
    lblDriverVersion.Text = Str$(dblDriverVersionA)
    lblDllVersion.Text = Str$(dblDllVersion)
End If
```

3. Upon execution of “cmdConnect_Click ()” (executed by pressing “Connect” on form), the desired virtual serial ports named on the form are opened. If either port is not found, exit is performed. The following code fragment typifies the operation of preparing a virtual port:

```
'*****************************
' Try to open VSP "PortA" port.
'*****************************
Status = cVspApiOpen(hVspA, txtPortA.Text)
If (ERROR_SUCCESS <> Status) Then
```

```
    MsgBox("cVspApiOpen() failed - Status: " + Str$(Status),
    MsgBoxStyle.Critical + _
    MsgBoxStyle.OKOnly, txtPortA.Text + " Error")
    Exit Sub
End If
```

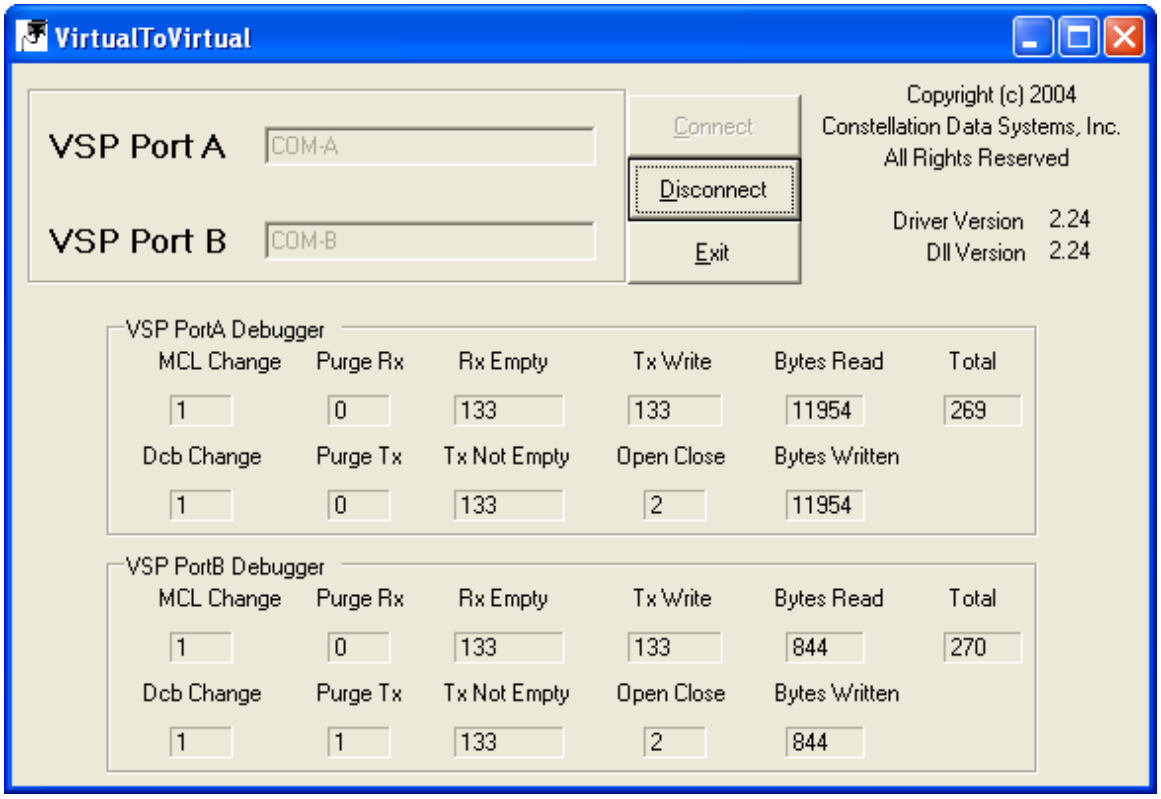
4. Two threads are then created. The “ThreadAToB” thread, and the “ThreadBToA” thread. These threads are responsible for reading from a device and writing to another.

```
'****************
' Create Threads.
'****************
bRunThreadAtoB = True
Static AtoB As New ThreadAtoB(EventArray, New EventCallback(AddressOf
CallbackAtoB))

hThreadAtoB = New System.Threading.Thread(AddressOf AtoB.ThreadProc)

hThreadAtoB.Start()
```

Internally, both these threads are very tight, and consist of strictly the read and writing operations. Consider the following code fragment from “ThreadAToB.ThreadProc ()”:

```
While (bRunThreadAtoB)
    Status = cVspApiRead(hVspA, Buff, BytesToRead, BytesRead)
    If (ERROR_SUCCESS <> Status) Then
      MsgBox("cVspApiRead() failed - Status: " + Str$(Status),
      MsgBoxStyle.Critical + _
      MsgBoxStyle.OKOnly, "VSP PortA Error")
      bDoDataWrite = False
    End If
    EventArray(9) += BytesRead

    '*********************************************
    ' Write to VSP "PortB" port, and validate return.
    '*********************************************
    BytesWritten = 0
    If (bDoDataWrite) Then
        Status = cVspApiWrite(hVspB, Buff, BytesRead, BytesWritten)
        If (ERROR_SUCCESS <> Status) Then
            MsgBox("cVspApiWrite() failed - Status: " + Str$(Status),
            MsgBoxStyle.Critical + _
            MsgBoxStyle.OKOnly, "VSP PortB Error")
            Exit While
        End If
        EventArray(10) += BytesWritten
        If (BytesRead <> BytesWritten) Then
            MsgBox("Read - " + BytesRead + " bytes; Wrote - " +
            BytesWritten + " bytes.", MsgBoxStyle.OKOnly + _
            MsgBoxStyle.Information, "VSP PortA Error")
        End If
    End If
```

5. The data movement is now accomplished by the threading. Termination is initiated by the operator pressing “Disconnect” or “Exit” on the form. Threads are then terminated, and the corresponding devices are closed.



6.2 Add Port Sample Reference Design (VB.NET)

The *Add Port* reference design demonstrates the techniques needed to dynamically add a Virtual Serial Port using the Visual Basic .NET development environment.

6.2.1 External Data Flow and Construction

TBD

6.2.2 Internal Data Flow and Construction

TBD

6.2.3 Module / Function Software Description

The primary source module is “Addport.vb”, which consists of about 300 lines of VB.NET code.

“AddportMod.vb” is a secondary module consisting of around 50 lines of VB.NET code where global variables, bit masks, various VSP structures, and VspApi.dll member function declarations are made.

6.2.4 Critical Function Descriptions

The following functions are critical.

Controlling Thread	Description
Addport_Load ( )	Entry point; does version verification.
cmdAddport_Click ( )	Add/Creates desired virtual serial port.
cmdExit_Click ( )	Exits application.

6.2.5 Narrative from Entry Point

The following code fragment illustrates the techniques used:

```
'***************************
' Try to add/create the VSP.
'***************************
PortName = txtPort.Text
Status = cVspApiAddSerialPort(hVsp, PortName.ToUpper)
If (ERROR_SUCCESS = Status) Then
      MsgBox("Port " + txtPort.Text + " successfully added.",
      MsgBoxStyle.Information + _
      MsgBoxStyle.OKOnly, "Addport")
Else
```

```
      MsgBox("cVspApiAddSerialPort() failed - Status: " + Str$(Status),
      MsgBoxStyle.Critical + _
      MsgBoxStyle.OKOnly, "ERROR - Addport")
End If
```

6.3 Delete Port Sample Reference Design (VB.NET)

The *Delete Port* reference design demonstrates the techniques needed to dynamically delete a Virtual Serial Port using the Visual Basic .NET development environment.

6.3.1 External Data Flow and Construction

TBD

6.3.2 Internal Data Flow and Construction

TBD

6.3.3 Module / Function Software Description

The primary source module is “Deleteport.cpp”, which consists of about 300 lines of VB.NET code.

“DeleteportMod.vb” is a secondary module consisting of around 50 lines of VB.NET code where global variables, bit masks, various VSP structures, and VspApi.dll member function declarations are made.

6.3.4 Critical Functions Description

The following functions are critical.

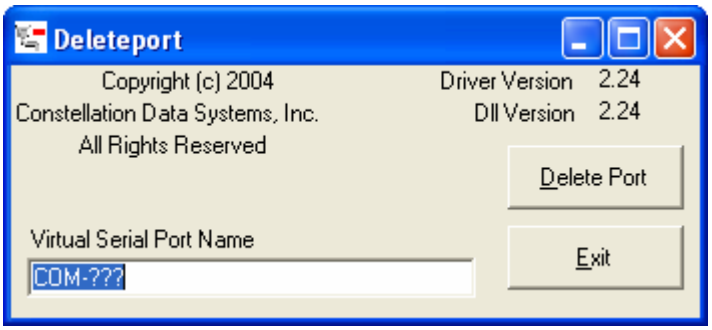
Controlling Thread	Description
Deleteport_Load ()	Entry point; does version verification.
cmdDeleteport_Click ()	Deletes desired virtual serial port.
cmdExit_Click ()	Exits application.

6.3.5 Narrative Description

The following code fragment illustrates the techniques used:

```
'***********************
' Try to delete the VSP.
'***********************
PortName = txtPort.Text

'*********************************************************************
' Checks to make sure that the requested port is not currently in use.
'*********************************************************************
Status = cVspApiGetOpenCount(hVsp, OpenCount)
If ((ERROR_SUCCESS = Status) And (OpenCount <> 0)) Then
      MsgBox(txtPort.Text + " is currently in use.",
      MsgBoxStyle.Critical + _
      MsgBoxStyle.OKOnly, "ERROR - Deleteport")
```

```
      txtPort.Focus()
      Exit Sub
End If

Status = cVspApiDeleteSerialPort(hVsp, PortName.ToUpper)
If (ERROR_SUCCESS = Status) Then
      MsgBox("Port " + txtPort.Text + " successfully deleted.",
      MsgBoxStyle.Information + _
      MsgBoxStyle.OKOnly, "Deleteport")
Else
      MsgBox("Make sure " + txtPort.Text + " is a Virtual Serial
      Port.", MsgBoxStyle.Critical + _
      MsgBoxStyle.OKOnly, "ERROR - Port not deleted")
End If
```



6.4 Enum Ports Sample Reference Design (VB.NET)

The Enum Ports reference design demonstrates the techniques needed to enumerate the system Serial Ports, and differentiate between Virtual Serial Ports added using the VSP framework, and other serial ports, using the Virtual Basic .NET development environment.

6.4.1 External Data Flow and Construction

TBD

6.4.2 Internal Data Flow and Construction

TBD

6.4.3 Module / Function Software Description

The primary source module is “Enumports.vb”, which consists of about 300 lines of VB.NET code.

“EnumportsMod.vb” is a secondary module consisting of around 50 lines of VB.NET code where global variables, bit masks, various VSP structures, and VspApi.dll member function declarations are made.

6.4.4 Critical Functions Description

The following functions are critical.

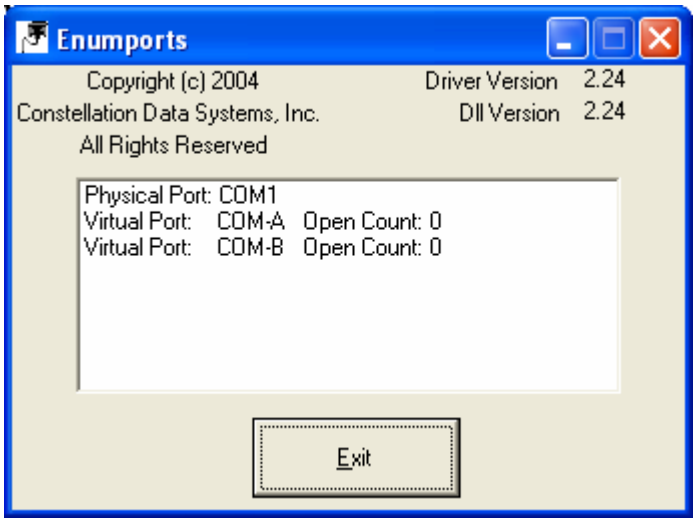
Controlling Thread	Description
Enumports_Load ( )	Entry point. Does version verification and then enumerates all serial ports.
cmdExit_Click ( )	Exits application.

6.4.5 Narrative Description

The following code fragment illustrates the techniques used:

```
'************************************************
' Try to open HKLM/HARDWARE/DEVICEMAP/SERIALCOMM.
'************************************************
RegKey =
Registry.LocalMachine.OpenSubKey("HARDWARE\\DEVICEMAP\\SERIALCOMM")

If (RegKey Is Nothing) Then
      MsgBox("Unable to open registry key HKLM/HARDWARE/",
      MsgBoxStyle.Critical + _
      MsgBoxStyle.OKOnly, "Registry Error")
      End
End If

/*****************************************************
```

```
 ' Spin through all serial ports (virtual and physical).
 '*******************************************************
 For Each Name As String In RegKey.GetValueNames()
     PortName = RegKey.GetValue(Name)
     Status = cVspApiIsVirtualPort(hVsp, PortName, bIsVirtualPort)
     If (ERROR_SUCCESS = Status) Then
         If (bIsVirtualPort) Then

'*******************************************************
' Found a Virtual Serial Port.
' Get port's open count and display on form accordingly.
'*******************************************************
            Call cVspApiGetOpenCount(hVsp, PortName, OpenCount)
            lstEnumports.Items.Add("Virtual Port: " + PortName + "
            Open Count: " + OpenCount.ToString)
         Else

'*********************************************************
' Found a physical serial port. Display on form accordingly.
'*********************************************************
            lstEnumports.Items.Add("Physical Port: " + PortName)
         End If
    End If
Next
```

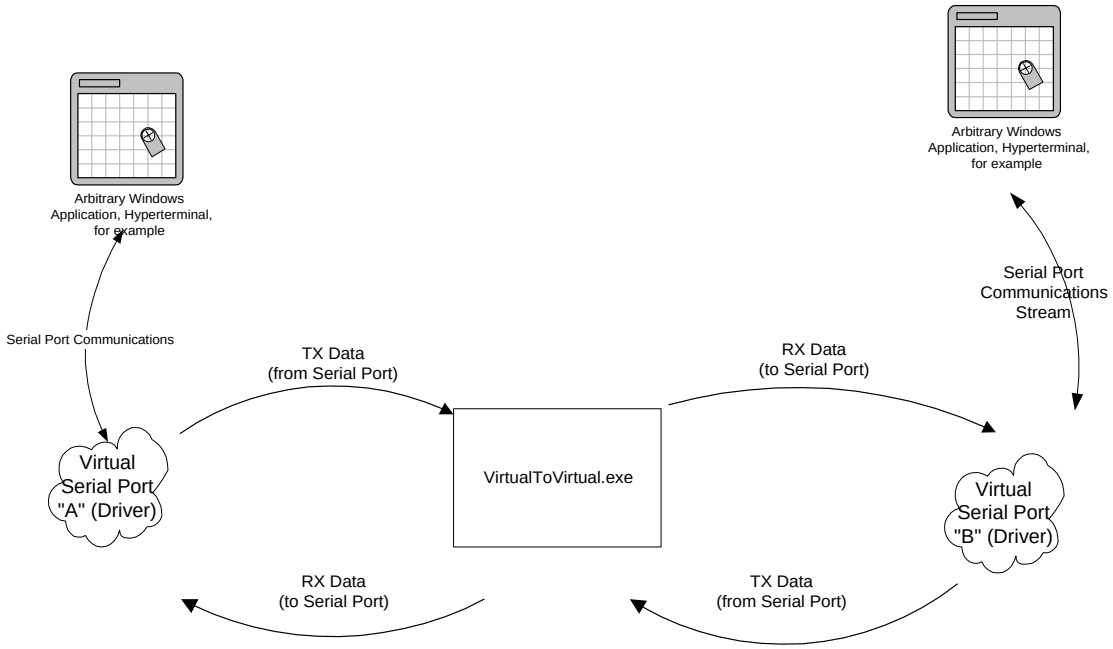


6.5 Virtual to Virtual Serial Port Sample Reference Design (C/C++)

The *Virtual to Virtual* reference design demonstrates the techniques of taking data to and from a *Virtual Serial Port* and presenting that information to another *Virtual Serial Port.* An engineer may wish to use this sample as a reference design (starting point) for a data replicator, debugging interface, or data collection device. Consider the following "external" data flow diagram:

6.5.1 External Data Flow and Construction

Typical Virtual To Virtual Serial Port Data Flow

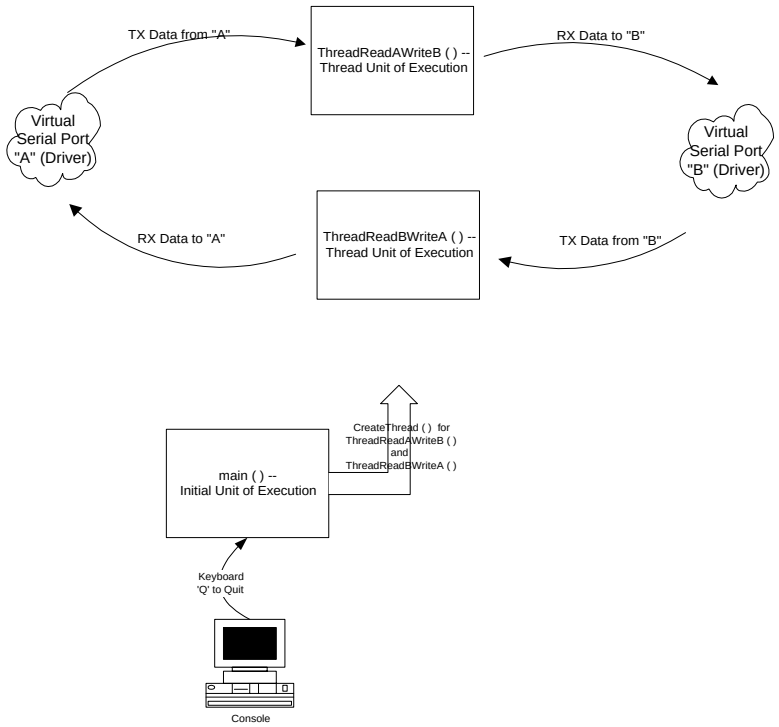


6.5.2 Internal Data Flow and Construction

The Internal construction of *VirtualToVirtual* is that of a multi-thread application using standard WIN32 operations wherever possible. The following programming paradigms are used:

Paradigm	**Usage**
WIN32	Thread creation and synchronization.
VSP API	Access to the Virtual Serial Port.
Stdio	Console operations using printf ( ), misc. string operations, etc.

The following data flow diagram illustrates the "internal" data flow of the *VirtualToVirtual* reference design:



6.5.3 Module / Function Software Description

The primary source module is “VirtualToVirtual.cpp”, which consists of about 400 lines of C/C++ code.

6.5.4 Critical Function Descriptions

The following functions are critical.

Controlling Thread	Description
ThreadAToB ( )	Reads data from virtual serial port A and writes that data to virtual serial port B.
ThreadBToA ( )	Reads data from virtual serial port B and writes that data to virtual serial port A.
MclToMsl ( )	Connects the Modem Control Lines (Mcl) of source virtual serial port to destination virtual serial port.
ProcessCommandLineSwitches ( )	Whips through the command line parameters, and sets up global variables based on parameter settings.
main ( )	Entry point.

6.5.5 Narrative from Entry Point

From the “main ( )” entry point, the following operations are performed:

2. The virtual ports named on the command line are opened. If either port is not found, exit is performed. The following code fragment typifies the operation of preparing a virtual port:

```
ErrorCode = hVspA.Open (argv[1]);
    if (ERROR_SUCCESS != ErrorCode) {
        printf("Error: Open () failed on Port:'%s' with code: %d\n",
argv[1], ErrorCode);
        exit (0);
    }
```

3. A version number comparison for the Underlying Device Driver (for each virtual port), DLL version comparison, and the version of *VirtualToVirtual* is performed. It is strongly suggested that the underlying VSP device driver, the VSP API DLL, and the application software (reference design, utility, or custom VSP application all conform to the same version). The following code fragment typifies the operation of version number validation:

```
    int ErrorCode;
        ErrorCode = hVspA.DllVersion(&DllVersion);
        if (ERROR_SUCCESS != ErrorCode) {
            printf("Error: DllVersion () failed code: %d\n", ErrorCode);
            exit (0);
        }
    . . .

    ULONG DriverVersionA, DriverVersionB;
        ErrorCode = hVspA.DriverVersion (&DriverVersionA);
        if (ERROR_SUCCESS != ErrorCode) {
            printf("Error: DriverVersion (), port '%s' failed code: %d\n",
    argv[1], ErrorCode);
            hVspA.Close();
            hVspB.Close();
            exit (0);
        }
    . . .

        while (     (DriverVersionA != DllVersion ) ||
                    (DriverVersionA != VSP_VERSION) ||
                    (DriverVersionB != VSP_VERSION) )
            {
```

Note that once the version number comparison is performed, the operator is given the opportunity to either continue (‘c’ keystroke), or to quit (‘q’ keystroke).

4. Both Virtual Serial Ports are installed with a set of read timeouts. The timeouts chosen cause “ReadFile( )” operations to end after 100ms of no serial data traffic (following the received first byte), OR after 1000ms (1 second) the start of a “ReadFile ( )” operation, whichever occurs first. Consider the following code fragment:

```
VspTimeouts.ReadIntervalTimeout = 100;
    VspTimeouts.ReadTotalTimeoutConstant = 1000;

    ErrorCode = hVspA.SetTimeouts ( &VspTimeouts );
    if (ERROR_SUCCESS != ErrorCode) {
        printf("Error: SetTimeouts () failed on Port:'%s' with code: %d\n",
               argv[1], ErrorCode);
        hVspA.Close ();
        hVspB.Close ();
        exit (0);
    }

    ErrorCode = hVspB.SetTimeouts ( &VspTimeouts );
    if (ERROR_SUCCESS != ErrorCode) {
        printf("Error: SetTimeouts () failed on Port:'%s' with code: %d\n",
               argv[2], ErrorCode);
        hVspA.Close ();
        hVspB.Close ();
        exit (0);
    }
```

5. Two threads are then created -- from main ( ) -- The “ThreadAToB” thread, and the “ThreadBToA” thread. These threads are responsible for reading from a device and writing to another.

Internally, both these threads are very tight, and consist of strictly the read and writing operations. Consider the following code fragment from "ThreadAToB":

```
    //
    // Spin while the thread handle value is valid. The main thread
    // can tear down this thread by invalidating the thread handle variable.
    //

    while (INVALID_THREAD_HANDLE != hThreadAToB)
    {
        int LastError;
        LastError = hVspA.Read ( Buff, sizeof(Buff), &dwBytesRead );
        if (ERROR_SUCCESS != LastError) {
            printf ("Error: hVspA.Read (), code %d\n", LastError);
            break;
        }

        if (0 == dwBytesRead)
            continue;

        // printf ("<RDA-%d>", dwBytesRead);

        LastError = hVspB.Write ( Buff, dwBytesRead, &dwBytesWritten );
        if (ERROR_SUCCESS != LastError) {
            printf ("Error: hVspB.Write (), code %d\n", LastError);
            break;
        }


        if (dwBytesRead != dwBytesWritten)
        {
            printf ("Warning: %d bytes read from COM A, %d bytes were written to
Com B.\n",
                dwBytesRead, dwBytesWritten);
        }

    }  // while (INVALID_THREAD_HANDLE == hThreadAToB)

    printf ("Note: ThreadAToB says goodbye\n");

    ExitThread (0);
    return 0;
```

6. The data movement is now accomplished by the threading. Termination is initiated by the operator typing 'Q' on the keyboard, which causes *main ( )* to begin termination. Threads are then terminated, and the corresponding devices are closed.

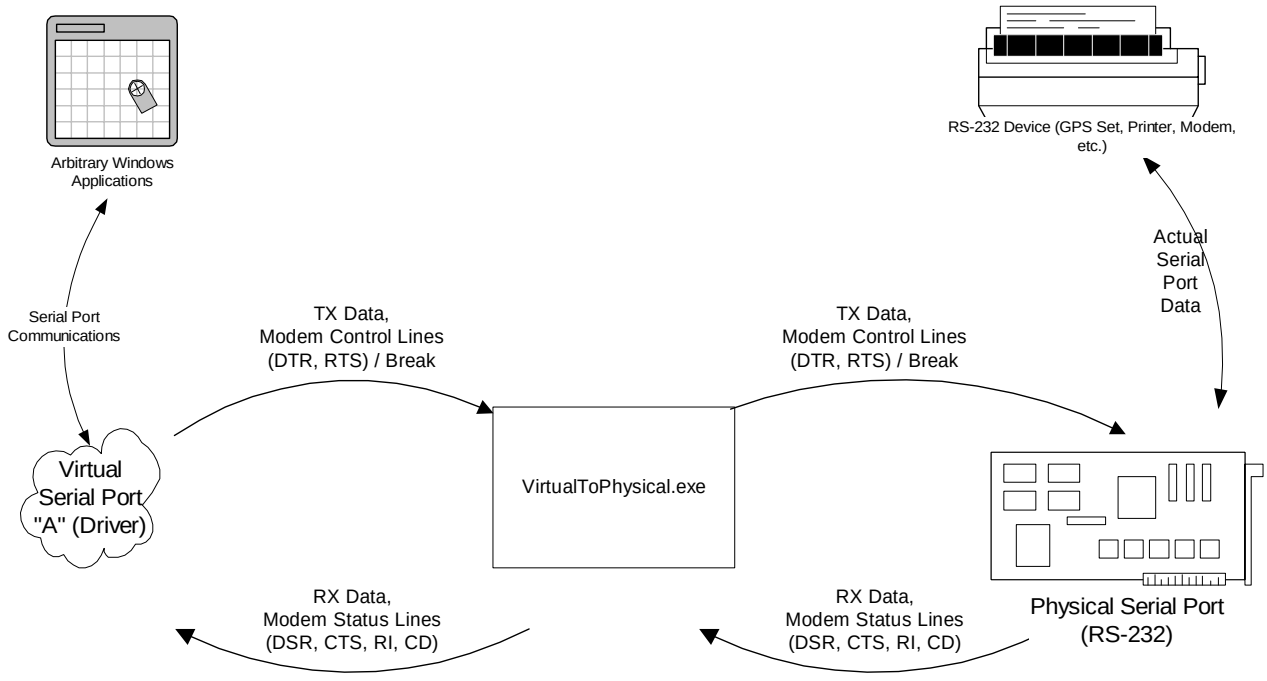
6.6 Virtual to Physical Serial Port Reference Design (C/C++)

The *Virtual to Physical* reference design demonstrates the techniques of taking data to and from a *Virtual Serial Port* and presenting that information to a physical serial port. An engineer may wish to use this sample as a reference design (starting point) for a data redirector, data tap, or a data corrector.

6.6.1 Exterior Data Flow and Construction

Consider the following "external" data flow diagram:

Typical Virtual To Physical Serial Port Data Flow

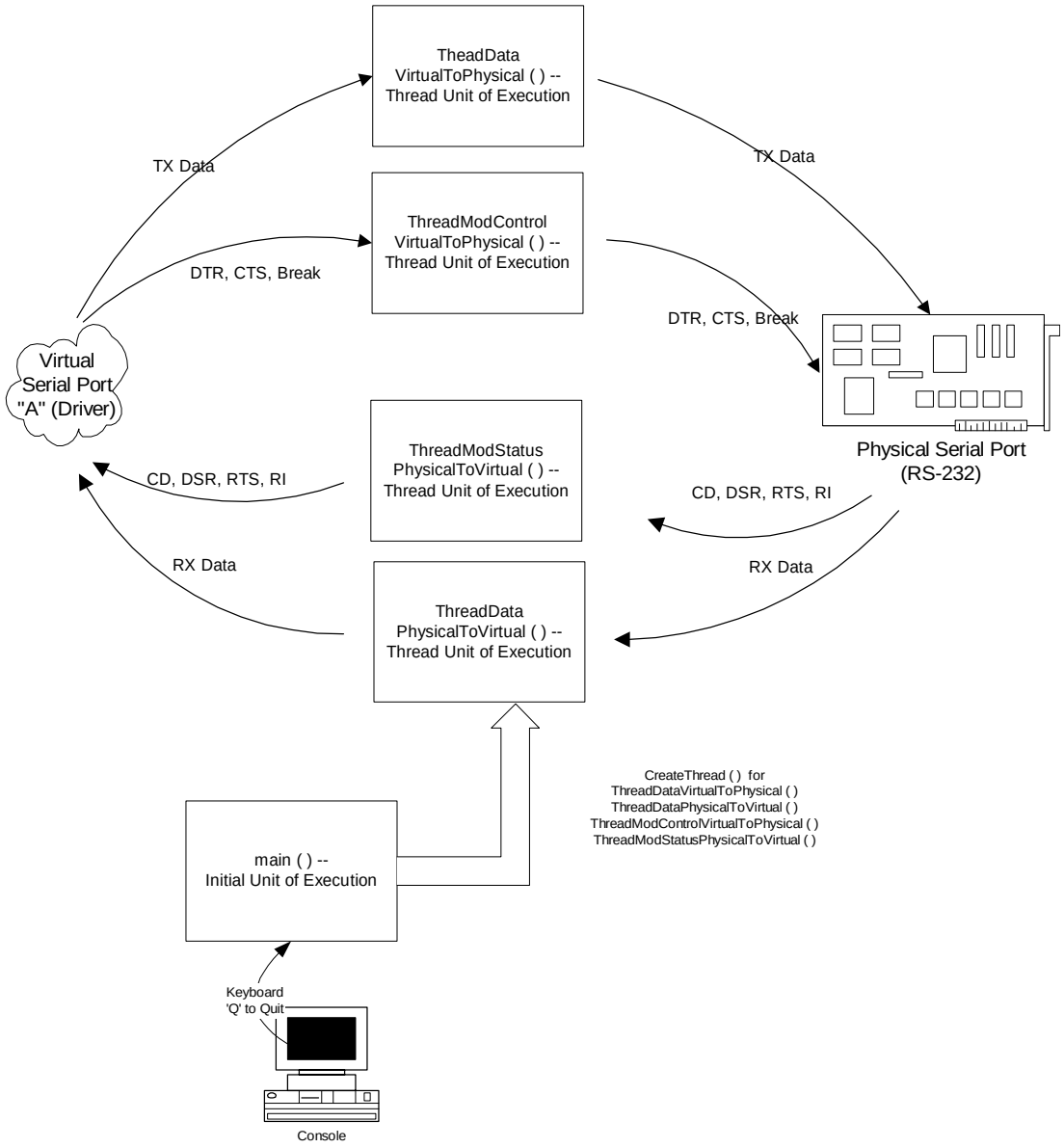


6.6.2 Internal Data Flow and Construction

The Internal construction of *VirtualToPhysical* is that of a multi-thread application using standard WIN32 operations wherever possible. The following programming paradigms are used:

Paradigm	**Usage**
WIN32	Thread creation and synchronization.
WIN32 Communications API	Access to physical communications device.
VSP API	Access to the Virtual Serial Port.
stdio	Console operations using printf ( ), misc. string operations, etc.

The following data flow diagram illustrates the "internal" data flow of the *VirtualToPhysical* reference design:



6.6.3 Module/Function Software Description

The primary source module is “VirtualToPhysical.cpp”, which consists of several hundred lines of C/C++ code.

6.6.4 Critical Function Descriptions

The following functions are critical.

Controlling Thread	Description
ThreadVirtualToPhysical ( )	Reads data from the virtual serial port(s) and writes that data to the physical serial port.
ThreadPhysicalToVirtual ( )	Reads data from the physical serial port and writes that data to the virtual serial port(s).
OpenPhysicalPort ( )	Finds the physical port, gets a handle to the same, and builds a DCB with default settings.
ProcessCommandLineSwitches ()	Whips through the command line parameters, and sets up global variables based on parameter settings.
main ( )	Entry point.

6.6.5 Narrative from Entry Point

From the “main ( )” entry point, the following operations are performed:

1. The physical port named on the command line is examined for the COM1 or COM2. Should another port be found, exit is performed.

2. The virtual device named on the command line is prepared for operation with the following operation (from the VSP API):

```
//
// Get the Virtual serial device up and running...
//

ErrorCode = hVirtual.Open (argv[3]);
if (ERROR_SUCCESS != ErrorCode) {
    printf("\nError: Open () failed on Virtual Serial Port:'%s' with
                    code: %d\n", argv[3], ErrorCode);
    printf("Suggestion: Verify that '%s' a Virtual Serial Port\n", argv[3]);
    exit (0);
}
```

3. Driver and DLL version comparison is performed. It is strongly suggested that the underlying VSP device driver, the VSP API DLL, and the application software (reference design, utility, or custom VSP application all conform to the same version).

4. The Virtual Port is installed with a set of read timeouts. The timeouts chosen cause “ReadFile( )” operations to end after 100ms of no serial data traffic (following the received first byte), OR after 1000ms (1 second) the start of a “ReadFile ( )” operation, whichever occurs first. Consider the following code fragment:

```
    //
    // Give the driver "backdoor" timeouts which make sense.
    //

    VSP_TIMEOUTS VirtualTimeouts = {0};

    //
    // A ReadIntervalTImeout 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.
    //

    VirtualTimeouts.ReadIntervalTimeout = 100;
    VirtualTimeouts.ReadTotalTimeoutConstant = 1000;

    ErrorCode = hVirtual.SetTimeouts ( &VirtualTimeouts );
    if (ERROR_SUCCESS != ErrorCode) {
        printf("\nError: Virtual SetTimeouts () failed on Port:'%s' with code:
%d\n",                                                              argv[3],
ErrorCode);
        hVirtual.Close ();
        exit (0);
```

5 The Physical Serial Port is prepared for operation using the following WIN32 functions: *CreateFile ( ), SetCommTimeOuts ( ), GetCommState ( ), DCB Structure,* and *SetCommState ( ).* For more information, consult Section 7; *Selected WIN32 References*. Please note that the physical port access is done using OVERLAPPED operations. The OVERLAPPED methods insure proper multi thread and multi process functionality.

6 Several threads are then created;

- ThreadDataVirtualToPhysical ( )
- ThreadModControlVirtualToPhysical( )
- ThreadDataPhysicalToVirtual ( )
- ThreadModStatusPhysicalToVirtual ( )

As the naming nomenclature suggests, these threads are responsible for either moving data, or modem status / control lines.

7 After the threads are created, the data movement is in progress, and continues until the operator selects termination. Termination is initiated by the operator typing ‘Q’ on the keyboard. Threads are then terminated, and the corresponding devices are closed.

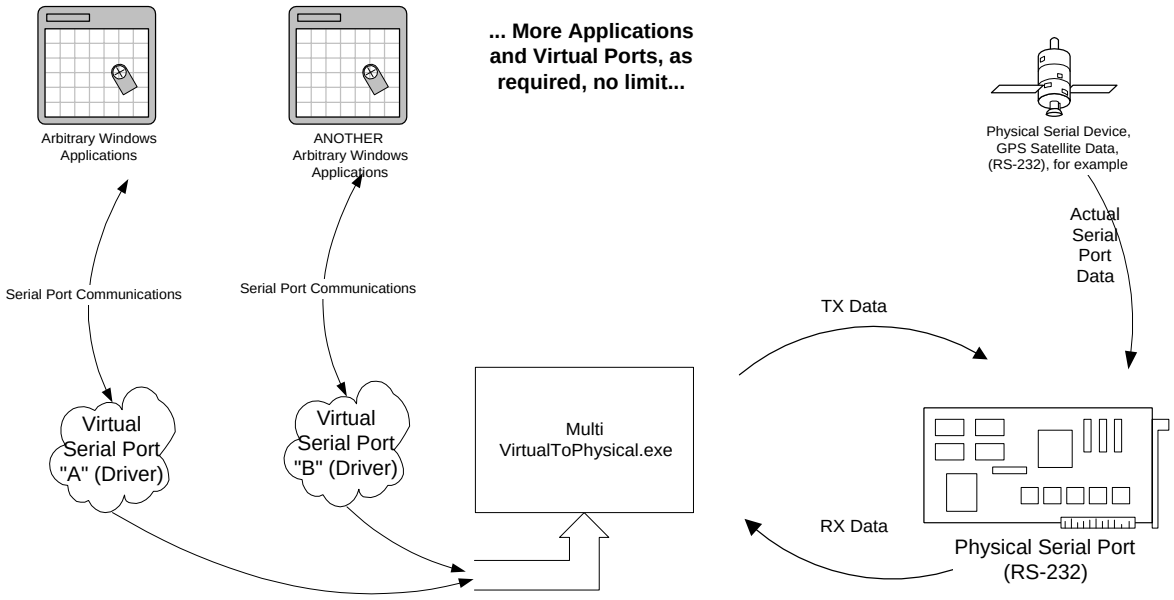
6.7 Physical to MultiVirtual Serial Port Sample Reference Design (C/C++)

The *Physical to Multiple Virtual* reference design demonstrates the techniques of taking data to and from a single physical device and reproducing that data on multiple *Virtual Serial Ports.* An engineer may wish to use this sample as a starting point for a GPS data splitter, data activity monitor, or a "data Y". The following examples show two virtual ports in operation, however the utility is designed to work with an arbitrary number of virtual ports.

6.7.1 External Data Flow and Construction

Consider the following "external" data flow diagram:

Typical Multi Virtual To Physical Serial Port Data Flow

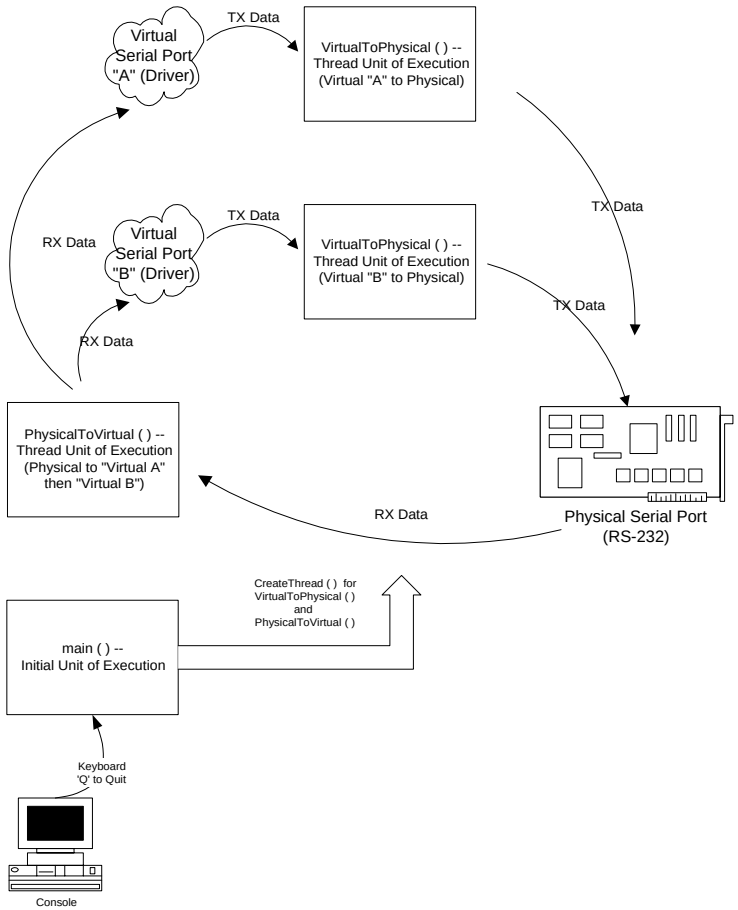


6.7.2 Internal Data Flow and Construction

The Internal construction of *MultiVirtualToPhysical* is that of a multi-thread application using standard WIN32 operations wherever possible. The following programming paradigms are used:

Paradigm	**Usage**
WIN32	Thread creation and synchronization.
WIN32 Communications API	Access to physical communications device.
VSP API	Access to the Virtual Serial Port.
Stdio	Console operations using printf ( ), misc. string operations, etc.

The following data flow diagram illustrates the "internal" data flow of a *MultiVirtualToPhysical* reference design, running with 2 Virtual Ports (hypothetically) named "A" and "B":



As you can see, although there is one "*VirtualToPhysical*" thread for each virtual port, there is only one "*PhysicalToVirtual"* thread. The construction of

the "*PhysicalToVirtual"* thread is such that the thread blocks on a read to the physical port, and once that read is accomplished, the data just read is written to each of the Virtual Ports.

Analogously, each "*VirtualToPhysical*" thread blocks on a read to the corresponding virtual port, and then once that read is accomplished, the data just read is written to the physical port.

6.7.3 Module / Function Software Description

The primary source module is "VirtualToPhysical.cpp", which consists of about 1000 lines of C/C++ code. Note that this reference design is much more complex than its cousin, the simpler *VirtualToPhysical* reference design. The additional complexity stems from two major differences: 1) multiple Virtual Port handling, and 2) Very robust command line parameter handling. Note: consult the *Core Users Guide and Reference* for information on command line parameters and operation.

6.7.4 Critical Function Descriptions

The following functions are critical.

Controlling Thread	**Description**
`ThreadVirtualToPhysical ( )`	Reads data from the virtual serial port(s) and writes that data to the physical serial port.
`ThreadPhysicalToVirtual ( )`	Reads data from the physical serial port and writes that data to the virtual serial port(s).
`OpenPhysicalPort ( )`	Finds the physical port, gets a handle to the same, and builds a DCB with default settings.
`ProcessCommandLineSwitches ()`	Whips through the command line parameters, and sets up global variables based on parameter settings.
`main ( )`	Entry point.

6.7.5 Narrative from Entry Point

From the “main ( )” entry point, the following operations are performed:

1. The physical port is opened by “*OpenPhysicalPort ( )”:*

```
if (!OpenPhysicalPort ( argc, argv)) {
        CleanupPortsAndThreads ();
        exit (0);
}
```

   Initially the port DCB (global: gDCB) is setup for 19200 BPS, 8 data bits, 1 stop bit, and XON / XOFF handshaking. These parameters may be overridden by command line parameters later.

2. Command line parameters are processed by “*ProcessCommandLineSwitches ( )”:*

```
if (!ProcessCommandLineSwitches ( argc, argv )){
        CleanupPortsAndThreads ();
        exit (0);
    }
```

   The majority of the command line switches simply change the global DCB.

3. Now the physical port setup (DCB and timeout) is completed by *“SetupPhysicalPortDcbAndTO ( )”.*

```
// Now that the command line switches have been
// processed, we setup timeouts and DCB in the
// physical port.
//

if (!SetupPhysicalPortDcbAndTO ( ) ) {
    CleanupPortsAndThreads ();
    exit (0);
}
```

   At this point the physical port is completely setup with respect to communications parameters and such which were specified on the command line.

4. Now the following loop runs once for each virtual port indicated on the command line:

```
//
// Start all virtual ports, this loop involves version validation
// and thread creation...
//

for (int VirtualPort = 0; VirtualPort < (argc - 2); VirtualPort++) {

    //
    // Look for a leading slash in the argument, which should indicate
    // then end of the list of virtual ports.
    //
```

```
        if ('/' == *argv[2+VirtualPort])
            break;
```

Each Virtual Port is then opened, “version validated”, and read timeouts are installed (Default: 100ms read interval, and 1 second total). Then a T*hreadVirtualToPhysical ( )* is created for each Virtual Port.

5. Exactly one *ThreadPhysicalToVirtual ( )* thread is then created.

6. The *main ( )* then waits for the operator to select ‘Q’ from the keyboard to indicate its time to quit. During this time the threading performs the desired data movement operations.

6.8 Add Port Sample Reference Design (C/C++)

The *Add Port* reference design demonstrates the techniques needed to dynamically add a Virtual Serial Port. External Data Flow and Construction

6.8.1 External Data Flow and Construction

TBD

6.8.2 Internal Data Flow and Construction

TBD

6.8.3 Module / Function Software Description

The primary source module is “Addport.cpp”, which consists of about 150 lines of C/C++ code.

6.8.4 Critical Function Descriptions

The following functions are critical.

Controlling Thread	Description
main ( )	Entry point.

6.8.5 Narrative from Entry Point

The following code fragment illustrates the techniques used:

```
    //
    // Go out to the VSP API to add the port
    //

    ErrorCode = hVsp.AddSerialPort (argv[1]);

    //
    // Validate the results of that operation...
    //

    if (ERROR_SUCCESS == ErrorCode) {
        printf("Port %s added.\n\n", argv[1]);
    }
    else if (ERROR_INVALID_NAME == ErrorCode) {
        printf("Error - Addport() : Port name too long. Port not created.\n\n");
        exit(ErrorCode);
    }
    else if (ERROR_ALREADY_EXISTS == ErrorCode) {
        printf("Error - Addport() : Port %s has already been created.\n\n", argv[1]);
        exit(ErrorCode);
    }
    else if (ERROR_CANTOPEN == ErrorCode) {
        printf("Error - Addport() : Function failed, code %i\n\n", ErrorCode);
        exit(ErrorCode);
    }
    else{
        printf("Error - Addport() : %s could not be created. Code %i\n\n",
argv[1], ErrorCode);
```

```
        exit(ErrorCode);
    }

    hVsp.Close ();
```

The variable “argv[1]” contains a string representation of the port name to be added; “COM-A”, or “COM4”, for example.

6.9 Delete Port Sample Reference Design (C/C++)

The *Delete Port* reference design demonstrates the techniques needed to dynamically delete a Virtual Serial Port.

6.9.1 External Data Flow and Construction

TBD

6.9.2 Internal Data Flow and Construction

TBD

6.9.3 Module / Function Software Description

The primary source module is “Deleteport.cpp”, which consists of about 175 lines of C/C++ code.

6.9.4 Critical Functions Description

The following functions are critical.

Controlling Thread	Description
main ( )	Entry point.

6.9.5 Narrative Description

The following code fragment illustrates the techniques used:

```
//
// Checks to make sure that the requested port is not currently in use
//

ErrorCode = hVsp.GetOpenCount(argv[1], &OpenCount);

if ((ErrorCode == ERROR_SUCCESS)  && (OpenCount != 0)) {
    printf("Error : Can not delete port %s...port is in use.\n\n", argv[1]);
    exit (3);
}

//
// Delete the Virtual Serial Port
//
ErrorCode = hVsp.DeleteSerialPort (argv[1]);
if (ERROR_SUCCESS == ErrorCode) {
    printf("Port %s deleted.\n\n", argv[1]);
}
else {
    printf("Error: Port %s not found.\n"
           "Make sure %s is a Virtual Port.\n\n", argv[1], argv[1]);
}
```

Note that the variable “argv[1]” contains a string representation of the port name to be deleted; “COM-A”, or “COM4”, for example. Also observe that the “OpenCount” of the port is examined prior to deletion.

6.10 Enum Ports Sample Reference Design (C/C++)

The *Enum Ports* reference design demonstrates the techniques needed to enumerate the system Serial Ports, and differentiate between Virtual Serial Ports added using the VSP framework, and other serial ports.

6.10.1 External Data Flow and Construction

TBD

6.10.2 Internal Data Flow and Construction

TBD

6.10.3 Module / Function Software Description

The primary source module is “Enumports.cpp”, which consists of about 250 lines of C/C++ code.

6.10.4 Critical Functions Description

The following functions are critical.

Controlling Thread	Description
main ( )	Entry point.

6.10.5 Narrative Description

The following code fragment illustrates the techniques used:

```
//
// Spin through for each COM port name read from the
// registry at "HKLM\HARDWARE\\DEVICEMAP\\SERIALCOMM\\"
//
do
{
    CHAR    Name [25];
    UCHAR    szPortName[80];
    DWORD    dwName,
            dwSizeofPortName,
            Type;
    dwName = sizeof (Name);
    dwSizeofPortName = sizeof (szPortName);
    ++dwIndex;
    dwNumSubKeys = 0;
    Status = RegEnumValue (hKey, dwIndex, Name, &dwName, NULL, &Type,
             szPortName, &dwSizeofPortName);

    if ((Status == ERROR_SUCCESS) || (Status == ERROR_MORE_DATA)) {

        int ErrorCode;

        //
        // we have found a port to try...
        // so, try opening the "backdoor" to the driver with it
        // and if the port is active then spin off a thread...
        //

        ErrorCode = hVspA.IsVirtualPort ( (char *) szPortName, &bIsVirtualPort);
```

```
            if (ERROR_SUCCESS != ErrorCode) {

                printf ("Error %d VSPAPI IsVirtualPort ( ) failed\n", ErrorCode);
            } else {

                if (bIsVirtualPort) {
                    printf("Virtual Port : %s\n", Name);
                } else  {
                    printf("Physical Port: %s is %s\n", szPortName, Name);
                }
            }
        } // if ((Status == ERROR_SUCCESS) || (Status == ERROR_MORE_DATA)) {
    } while ( (Status == ERROR_SUCCESS) || (Status == ERROR_MORE_DATA) );
    //
    // Mop all local objects.
    //
    RegCloseKey (hKey);
```

7. Selected WIN32 References

Selected WIN32 SDK (Platform SDK) reference pages follow (reproduced by permission of the MSDN).

7.1 CreateThread

The **CreateThread** function creates a thread to execute within the virtual address space of the calling process.
To create a thread that runs in the virtual address space of another process, use the **CreateRemoteThread** function.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
  SIZE_T dwStackSize,                       // initial stack size
  LPTHREAD_START_ROUTINE lpStartAddress,    // thread function
  LPVOID lpParameter,                       // thread argument
  DWORD dwCreationFlags,                    // creation option
  LPDWORD lpThreadId                        // thread identifier
);
```

Parameters

lpThreadAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If *lpThreadAttributes* is NULL, the handle cannot be inherited.

Windows NT/2000/XP: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new thread. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor.

dwStackSize

[in] Specifies the initial size of the stack, in bytes. The system rounds this value to the nearest page. If this parameter is zero, the new thread uses the default size for the executable. For more information, see Thread Stack Size.

lpStartAddress

[in] Pointer to the application-defined function of type **LPTHREAD_START_ROUTINE** to be executed by the thread and

represents the starting address of the thread. For more information on the thread function, see **ThreadProc**.

lpParameter

[in] Specifies a single parameter value passed to the thread.

dwCreationFlags

[in] Specifies additional flags that control the creation of the thread. If the CREATE_SUSPENDED flag is specified, the thread is created in a suspended state, and will not run until the **ResumeThread** function is called. If this value is zero, the thread runs immediately after creation. At this time, no other values are supported.

Windows XP: If the STACK_SIZE_PARAM_IS_A_RESERVATION flag is specified, the *dwStackSize* parameter specifies the initial reserve size of the stack. Otherwise, *dwStackSize* specifies the commit size.

lpThreadId

[out] Pointer to a variable that receives the thread identifier.

Windows NT/2000/XP: If this parameter is NULL, the thread identifier is not returned.

Windows 95/98/Me: This parameter may not be NULL.

Return Values

If the function succeeds, the return value is a handle to the new thread. If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Note that **CreateThread** may succeed even if *lpStartAddress* points to data, code, or is not accessible. If the start address is invalid when the thread runs, an exception occurs, and the thread terminates. Thread termination due to an invalid start address is handled as an error exit for the thread's process. This behavior is similar to the asynchronous nature of **CreateProcess**, where the process is created even if it refers to invalid or missing dynamic-link libraries (DLLs).

Windows 95/98/Me: CreateThread succeeds only when it is called in the context of a 32-bit program. A 32-bit DLL cannot create an additional thread when that DLL is being called by a 16-bit program.

Remarks

The number of threads a process can create is limited by the available virtual memory. By default, every thread has one megabyte of stack space. Therefore, you can create at most 2028 threads. If you reduce the default stack size, you can create more threads. However, your application will have better performance if you create one thread per processor and build queues of requests for which the application maintains the context information. A thread would process all requests in a queue before processing requests in the next queue.

The new thread handle is created with THREAD_ALL_ACCESS to the new thread. If a security descriptor is not provided, the handle can be used in any function that requires a thread object handle. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If the access check denies access, the requesting process cannot use the handle to gain access to the thread. If the thread impersonates a client, then calls **CreateThread** with a NULL security descriptor, the thread object created has a default security descriptor which allows access only to the impersonation token's TokenDefaultDacl owner or members. For more information, see Thread Security and Access Rights.

The thread execution begins at the function specified by the *lpStartAddress* parameter. If this function returns, the **DWORD** return value is used to terminate the thread in an implicit call to the **ExitThread** function. Use the **GetExitCodeThread** function to get the thread's return value.

The thread is created with a thread priority of THREAD_PRIORITY_NORMAL. Use the **GetThreadPriority** and **SetThreadPriority** functions to get and set the priority value of a thread. When a thread terminates, the thread object attains a signaled state, satisfying any threads that were waiting on the object. The thread object remains in the system until the thread has terminated and all handles to it have been closed through a call to **CloseHandle**.

The **ExitProcess**, **ExitThread**, **CreateThread**, **CreateRemoteThread** functions, and a process that is starting (as the result of a call by **CreateProcess**) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means that the following restrictions hold:

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.

- Only one thread in a process can be in a DLL initialization or detach routine at a time.
- **ExitProcess** does not return until no threads are in their DLL initialization or detach routines.

A thread that uses functions from the C run-time libraries should use the **beginthread** and **endthread** C run-time functions for thread management rather than **CreateThread** and **ExitThread**. Failure to do so results in small memory leaks when **ExitThread** is called.

Example Code

For an example, see Creating Threads.

Requirements

Windows NT/2000/XP: Included in Windows NT 3.1 and later.
Windows 95/98/Me: Included in Windows 95 and later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

7.2 DCB

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.

Requirements

Windows NT/2000/XP: Included in Windows NT 3.1 and later.
Windows 95/98/Me: Included in Windows 95 and later.
Header: Declared in Winbase.h; include Windows.h.

7.3 GetCommState

The **GetCommState** function retrieves the current control settings for a specified communications device.

```
BOOL GetCommState(
  HANDLE hFile <>,  // handle to communications device
  LPDCB lpDCB <>    // device-control block
);
```

Parameters

HFile [in] Handle to the communications device. The **CreateFile** function returns this handle.

lpDCB [out] Pointer to a **DCB** structure that receives the control settings information.

Return Values

If the function succeeds, the return value is nonzero.
If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000/XP: Included in Windows NT 3.1 and later.
Windows 95/98/Me: Included in Windows 95 and later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

7.4 SetCommState

The SetCommState function configures a communications device according to the specifications in a device-control block (a **DCB** structure). The function reinitializes all hardware and control settings, but it does not empty output or input queues.

```
BOOL SetCommState(
  HANDLE hFile;
  LPDCB lpDCB;   // device-control block
);
```

Parameters

hFile

[in] Handle to the communications device. The **CreateFile** function returns this handle.

lpDCB

[in] Pointer to a **DCB** structure that contains the configuration information for the specified communications device.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

The **SetCommState** function uses a **DCB** structure to specify the desired configuration. The **GetCommState** function returns the current configuration.
To set only a few members of the **DCB** structure, you should modify a **DCB** structure that has been filled in by a call to **GetCommState**. This ensures that the other members of the **DCB** structure have appropriate values.
The **SetCommState** function fails if the **XonChar** member of the **DCB** structure is equal to the **XoffChar** member.
When **SetCommState** is used to configure the 8250, the following restrictions apply to the values for the **DCB** structure's **ByteSize** and **StopBits** members:
The number of data bits must be 5 to 8 bits.

Example Code

For an example, see Configuring a Communications Resource.

Requirements

Windows NT/2000/XP: Included in Windows NT 3.1 and later.
Windows 95/98/Me: Included in Windows 95 and later.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

8. Notices

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 with CDS.

9. Index of Acronyms and Abbreviations

AKA	Also Known As
API	Applications Programming Interface
CD	Carrier Detect (modem status line)
CDS	Constellation Data Systems
CTS	Clear to Send (modem status line)
cVspApi	Virtual Serial Port API Class
DCE	Data Communications Equipment
DOS	Disk Operating System
DCB	WIN32 Device Control Block
DLL	Dynamic Link Library
DSR	Data Set Ready (modem status line)
DTE	Data Terminal Equipment
DTR	Data Terminal Ready (modem control line)
GPS	Global Positioning System
HyperTerminal	Standard Windows Communications Application
MS	Microsoft
MSDN	MS Developers Network
PCR	Physical Communications Resource (Such as a UART)
RI	Ring Indicate (modem status line)
RLSD	Receive Line Signal Detect (modem status line) aka CD
RTS	Request To Send (modem control line)
RX	Receive
RS-232	Recommended Standard 232 (from the Electronics Industry Association) for data communications
SDK	Software Development Kit
TBD	To Be Described / To Be Determined
TD	Transmit Data
TLA	Three Letter Acronym
TX	Transmit
UART	Universal Asynchronous Receiver / Transmitter
VSP	Virtual Serial Port
VSPAPI	Virtual Serial Port Applications Programming Interface
WIN16	Windows 16 Bit Programming Paradigm (Arguably Obsolete)
WIN32	Windows 32 Bit Programming Paradigm