



Network Serial Port - *Professional* Software Development Kit Programmers Guide and Reference

for Windows XP, Windows 2000, and Windows NT
Constellation Data Systems, Inc.

www.VirtualPeripherals.com

Copyright Ó 2004 Constellation Data Systems, Inc ("CDS"). All rights reserved. Consult your software license agreement. Brand and product names are trademarks of their respective holders. Portions of this manual are © Microsoft Corporation, and are used by permission of the MSDN.

Table of Contents

1. Introduction and Overview	4
1.1 Capabilities	4
1.2 Typical NSP Implementation	4
1.3 Development Environment.....	4
2. SDK Contents	5
3. SDK Documentation	7
4. Installation Instructions	7
5. Application Programming Interfaces	8
5.1 NSP Applications Programming Interface.....	9
5.2 VSP Applications Programming Interface.....	9
6. Network Access Techniques	10
6.1 Sockets and Internet Protocol Addresses.....	10
6.2 Client vs. Server NSP Applications.....	10
7. Reference Designs	11
7.1 <i>Physical On Client to Single</i> Reference Design.....	12
7.1.1 External Data Flow and Construction.....	12
7.1.2 Internal Data Flow and Construction.....	13
7.1.3 Module / Function Software Description.....	14
7.2 <i>Physical On Server To Single</i> Reference Design	15
7.2.1 Client vs. Server Access Techniques in POSTS vs. POCTS.....	15
7.3 <i>Physical On Server To Multiple</i> Reference Design	16
7.3.1 External Data Flow and Construction.....	16
7.3.2 Internal Data Flow and Construction.....	17
7.3.3 Module / Function Software Description.....	18
7.4 <i>Virtual On Client To Single</i> Reference Design	19
7.4.1 External Data Flow and Construction.....	19
7.4.2 Internal Data Flow and Construction.....	20
7.4.3 Module / Function Software Description.....	21
7.5 <i>Virtual On Server To Single</i> Reference Design	22
7.5.1 Client vs. Server Access Techniques in VOSTS vs. VOCTS.....	22
7.6 <i>Virtual On Server To Multiple</i> Reference Design	23
7.6.1 External Data Flow and Construction.....	23
7.6.2 Internal Data Flow and Construction.....	24
7.6.3 Module / Function Software Description.....	25
8. Selected WIN32 References	26
8.1 CreateThread.....	26
8.2 DCB.....	29
8.3 GetCommState	35
8.4 SetCommState	35
9. Notices.....	37
10. Index of Acronyms and Abbreviations	38

**Network Serial Port
Software Development Kit**

1. Introduction and Overview

This manual describes the Network Serial Port - *Professional* (NSP *Pro*) Software Development Kit (SDK).

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

1.1 Capabilities

The NSP *Pro* SDK is a product of Constellation Data Systems, Inc (CDS). This product is a development accelerator, which can cut months or years from a development project that requires a serial/communications resource (physical or virtual) which must be expressed across a computer network.

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

- Easily create custom software to transmit data between computer systems across networks
- Create Seamless connections to remote client and server systems
- Access powerful features native to the VSP Software Development Kit.

1.2 Typical NSP Implementation

There is no "typical NSP *Pro* implementation". The flexibility of the NSP *Pro* framework allows it to be used in almost endless variety of ways. Among them are connecting to remote field instrumentation, Telnet serial port connectivity, 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 NSP software development environment is Microsoft Visual C/C++ (MSVC) Version 6. The latest MSVC service packs are also recommended.

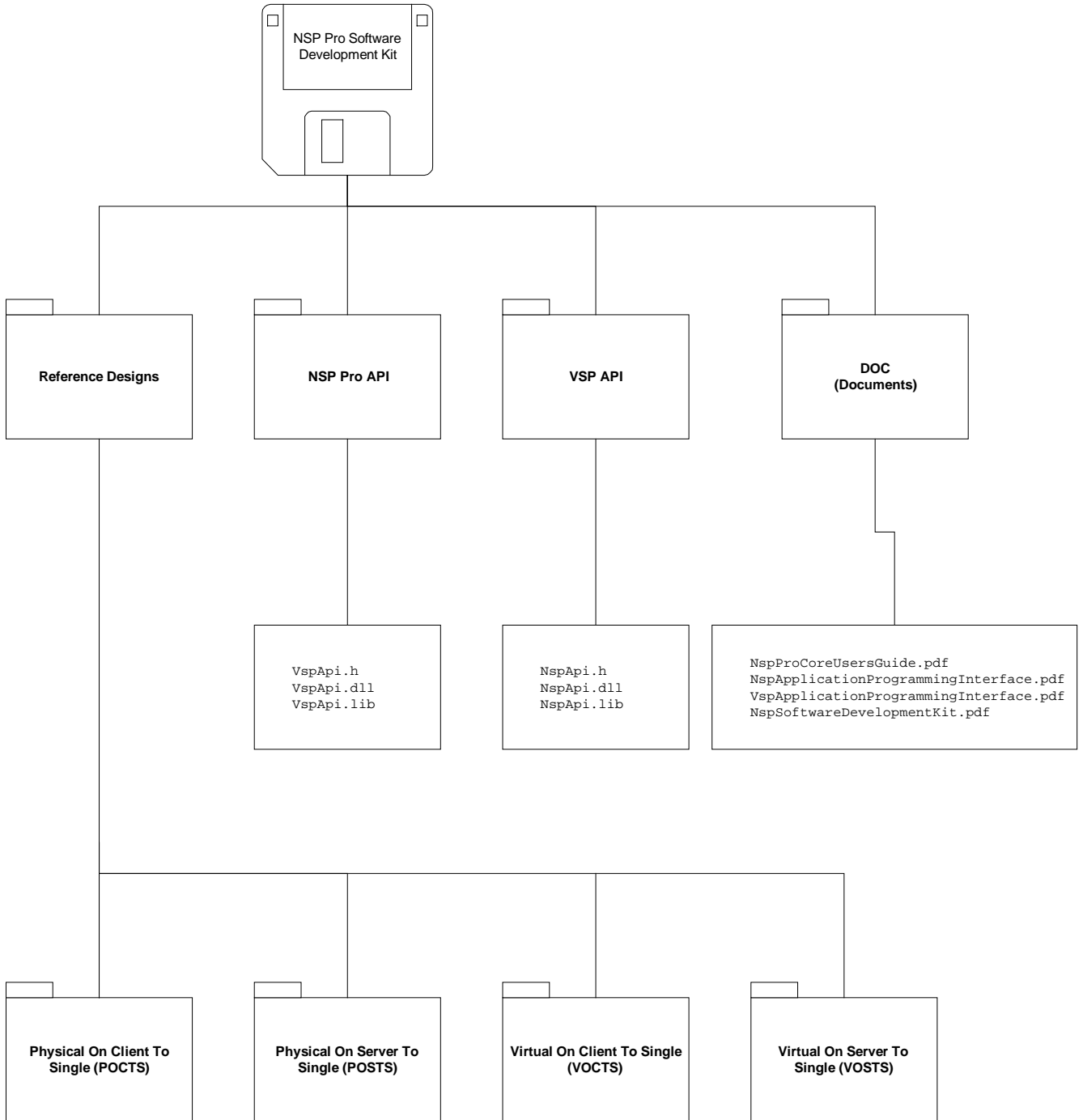
2. SDK Contents

The NSP *Pro* SDK consists of the following major components.

<u>Component</u>	<u>Contains</u>
Reference Designs	Fully documented and working designs which allow serial ports to be expressed in the following manner: <ul style="list-style-type: none">· Physical Serial Port on a “Client” system to a Single TCP/IP Socket Port.· Virtual Serial Port on a “Client” system to a Single TCP/IP Socket Port.· Physical Serial Port on a “Server” system to a Single TCP/IP Socket Port.· Virtual Serial Port on a “Server” system to a Single TCP/IP Socket Port.
NSP <i>Pro</i> Applications Programming Interface (API)	The NSI <i>Pro</i> API allows NSP applications to access other entities across a network, such as other remote NSP applications, Telnet implementations, streaming sockets, etc.
Virtual Serial Port (VSP) Applications Programming Interface (API)	The VSP API allows applications to access virtualized communications resources.
Documentation	<ul style="list-style-type: none">· NSP <i>Pro</i> Software Development Kit Programmers Guide and Reference· NSP <i>Pro</i> API Programmers Guide and Reference· VSP API Programmers Guide and Reference· NSP <i>Pro</i> Core Users Guide and Reference

**Network Serial Port
Software Development Kit**

The SDK is organized as follows:



3. SDK Documentation

The NSP *Pro* SDK documentation set consists of the following components:

- Network Serial Port – *Professional* Applications Programming Interface Programmers Guide and Reference
- Virtual Serial Port Applications Programming Interface Programmers Guide and Reference
- Network Serial Port – *Professional* Software Development Kit Programmers Guide and Reference -- this document.
- Network Serial Port Core Users Guide and Reference

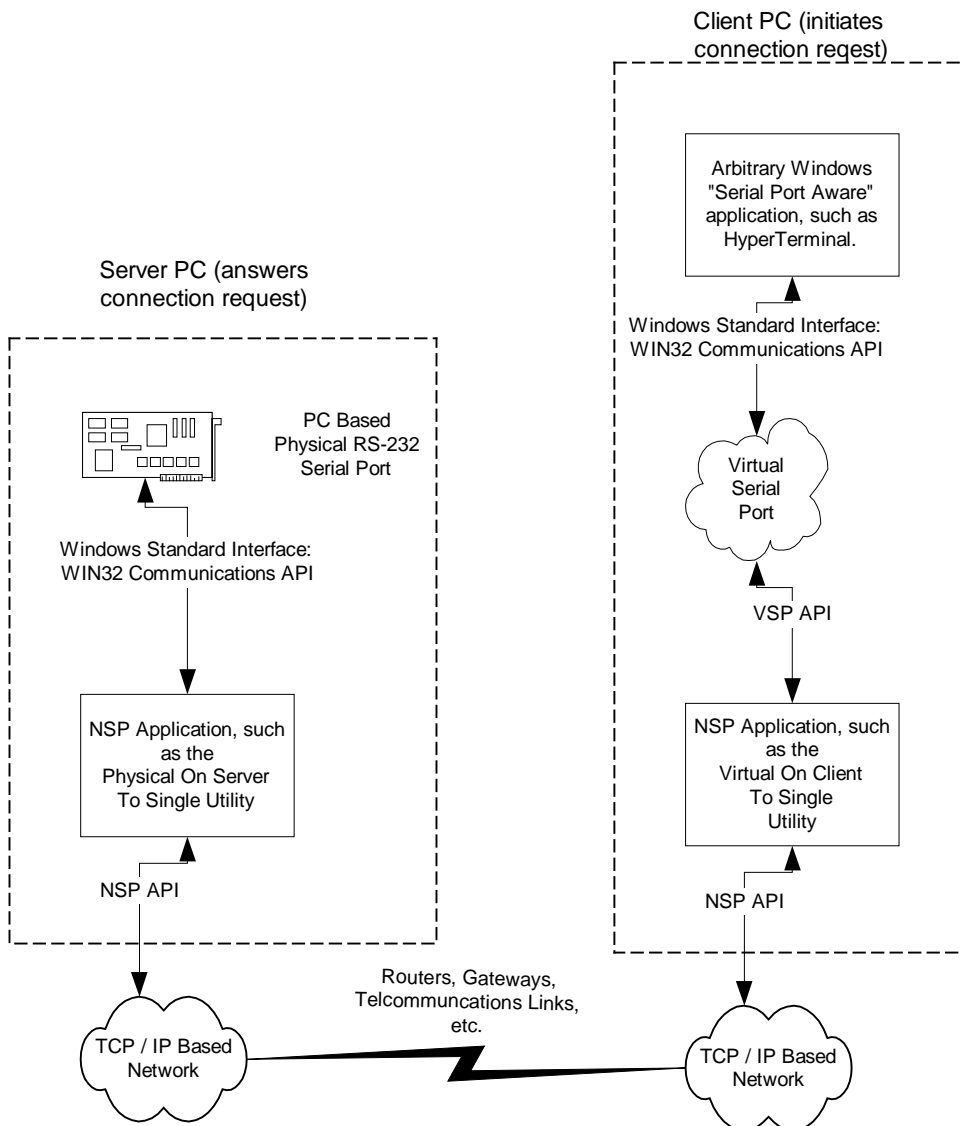
4. Installation Instructions

Review the provided terms and conditions, and then simply unzip (extract) the provided modules into a folder of you're choosing.

5. Application Programming Interfaces

There are two important interfaces used by NSP Applications, they are the NSP API, and the VSP API. The NSP API is needed by applications in order to enable simple network access. The VSP API is only needed by those applications that also need to use a virtualized serial port.

The following data flow diagram illustrates two hypothetical NSP applications, one which has virtualized a serial port (using the VSP framework), and another NSP application which access a physical serial port using the standard WIN32 Communications API. The diagram shows both applications using the NSP *Pro* framework to access the underlying network.



5.1 NSP Applications Programming Interface

The *NSP Applications Programming Interface* (NSPAPI) facilitates simple communication between NSP compliant software applications across a computer network, such as TCP/IP. What this means is that you do not have to be an expert in TCP/IP programming interfaces, and the underlying dynamics of execution.

It is suggested that you become familiar with this manual as you work with the provided reference designs.

5.2 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”).

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

6. Network Access Techniques

6.1 Sockets and Internet Protocol Addresses

IP addresses are typically represented in “dotted quad nomenclature”, where each IP address is shown as four decimal numbers (between 0 and 255), separated by periods (e.g., “61.12.123.11”). Socket numbers are integers between 0 and 65,535. The IP address paired with the socket number comprises an NSP addressable entity.

Important Point	Socket numbers may be used by other applications and system components. To avoid conflicts, you may wish to consult with the Internet Naming Authority’s (“INA”) master lists.
-----------------	--

6.2 Client vs. Server NSP Applications

A minimum of 2 applications is necessary to communicate across a network. Industry convention refers to these applications as the “Client” application and the “Server” application. The Client PC initiates the connection by attempting connection to a known socket at the Server’s Internet Protocol (“IP”) address.

Important Point	<p>The “Client” PC initiates a connection much as a person dialing a telephone initiates a telephone connection. Analogously, a “Server” PC responds to connection request from a “Client” PC in a similar fashion as a person answering a telephone.</p> <p>In order for the “Server” to answer an incoming “Client” connection, an NSP “Server” type utility must be running at the time the connection is attempted.</p>
-----------------	---

7. Reference Designs

The NSP *Pro* 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.

The Internal construction of the reference designs is that of multi-thread applications using standard WIN32 operations wherever possible. The following programming paradigms are also used:

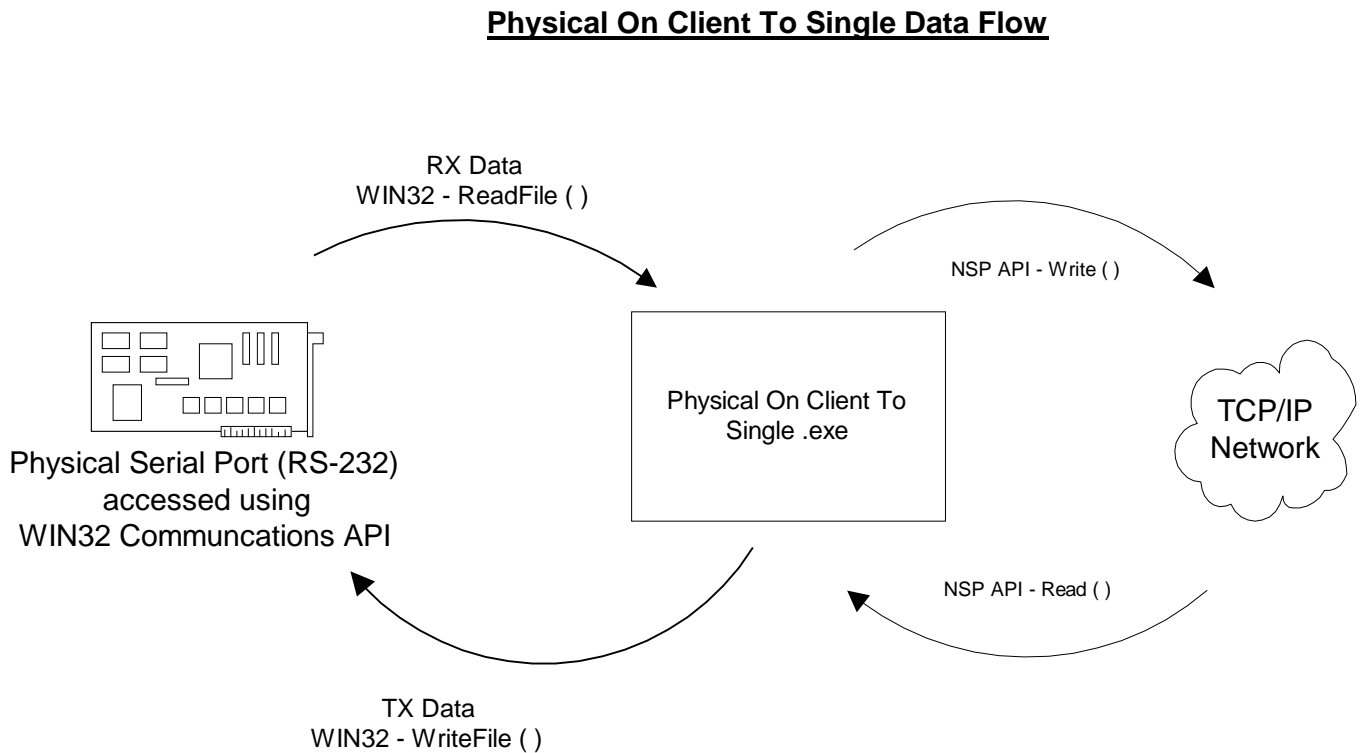
Paradigm	Usage
WIN32	Thread creation and synchronization.
WIN32 Communications API	Access to physical communications device, used by the <i>Physical On Client To Single (POCTS)</i> and the <i>Physical On Server To Single (POSTS)</i> reference designs.
Stdio	Console operations using printf (), misc. string operations, etc.
NSP API	Network Access Techniques.
VSP API	Virtual Serial Port access, used by the <i>Virtual On Client To Single (VOCTS)</i> and the <i>Virtual On Server To Single (VOSTS)</i> reference designs.

7.1 Physical On Client to Single Reference Design

The *Physical On Client To Single (POCTS)* reference design demonstrates the techniques of expressing the data flow of a physical serial port to a single network endpoint using “*Client style*” network access techniques. An engineer may wish to use this sample as a reference design (starting point) for a serial port data redirector.

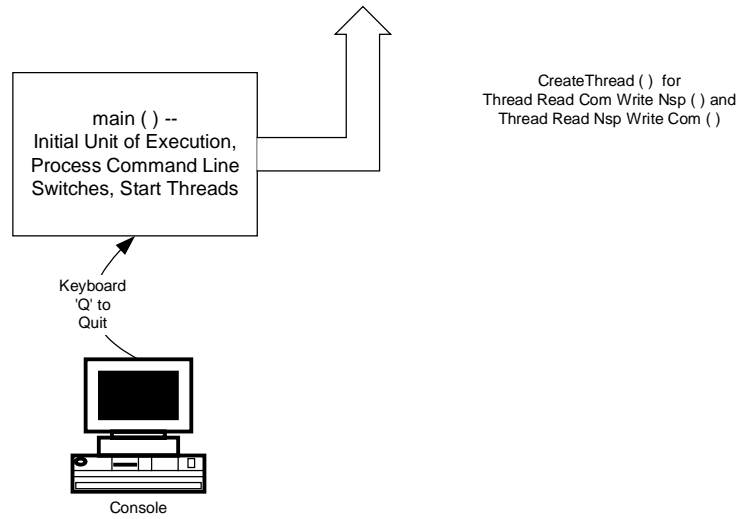
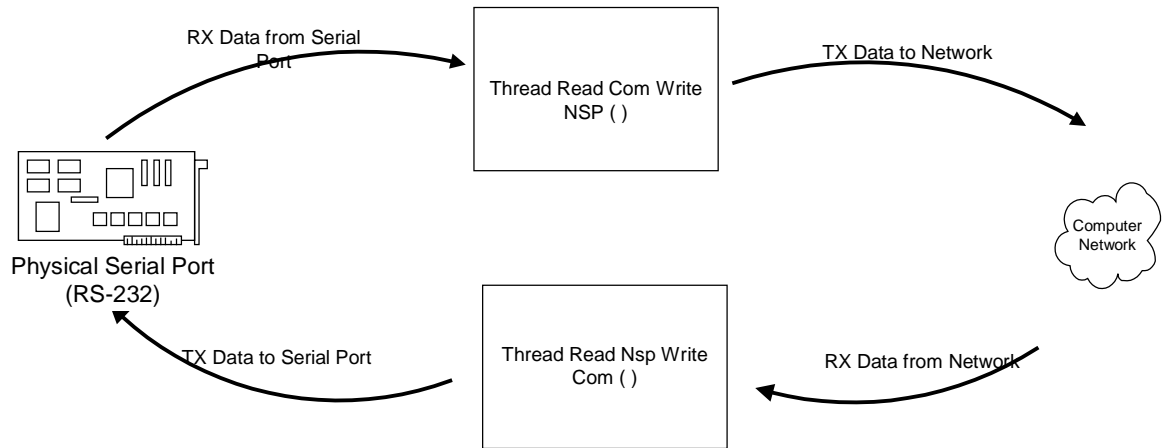
7.1.1 External Data Flow and Construction

Consider the following “external” data flow diagram of *POCTS*:



7.1.2 Internal Data Flow and Construction

The following data flow diagram illustrates the “internal” data flow of the *Physical On Client To Single* reference design:



7.1.3 Module / Function Software Description

The primary source module is “PhysicalOnClientToSingle.cpp”, which consists of about 900 lines of C/C++ code. From the “main ()” entry point, the following operations are performed:

1. The function *OpenPhysicalPort ()* prepares the physical port named on the command line for operation using the following industry standard WIN32 functions: *CreateFile ()*, *SetCommTimeouts ()*, *GetCommState ()*, *SetCommState ()*. The global DCB named *gDcb* is prepared.
2. The *ProcessCommandLineSwitches ()* function processes all command line parameters which are prefaced by a slash (/), and *gDcb* is modified accordingly.
3. The physical device is prepared and timeout data is setup from *gDcb* by the function: *SetupPhysicalPortDcbAndTO ()*.
4. The Network address and socket number is stored in the following global variables: *gNetworkAddress*, *gSocketNumber*.
5. Driver and DLL version comparison is performed. It is strongly suggested that the underlying NSP API DLL, and the application software (reference design, utility, all conform to the same version).
6. Two threads are then created; The *ThreadReadComWriteNsp ()*, and *ThreadReadNspWriteCom ()* thread. The nomenclature suggests what these threads perform. Note: *ThreadReadNspWriteCom ()* also prepares the Network for access using NSPAPI function: *OpenConnectionToServer ()*.
7. After the threads are created, the data movement is in progress, and continues until the operator selects termination. The operator typing ‘Q’ on the keyboard initiates termination. Threads are then terminated, and the corresponding devices are closed.

7.2 *Physical On Server To Single Reference Design*

The *Physical On Server To Single (POSTS)* reference design demonstrates the techniques of expressing the data flow of a physical serial port to a single network endpoint which uses “*Server style*” network access techniques. An engineer may wish to use this sample as a reference design (starting point) for a serial port data redirector.

Important Point:	The External and Internal data flow and construction of POSTS are virtually identical to that of POCTS. For that reason, the remainder of this discussion shall focus on the differences between the <i>Client</i> vs. <i>Server</i> access techniques utilized.
------------------	--

7.2.1 Client vs. Server Access Techniques in POSTS vs. POCTS

The *Server* NSP applications make use of the *OpenConnectionToClient* () function to make a connection. The *Server* NSP application will block until a *Client* application connects. The following statement from POSTS function *ThreadReadNspWriteCom* () makes the critical connection:

```
ErrorCode = hNsp.OpenConnectionToClient  
            (gSocketNumber, &gClientConnection);
```

Whereas, in POCTS, the following statement makes the critical connection from the *Client* (POCTS) to the remote *Server*:

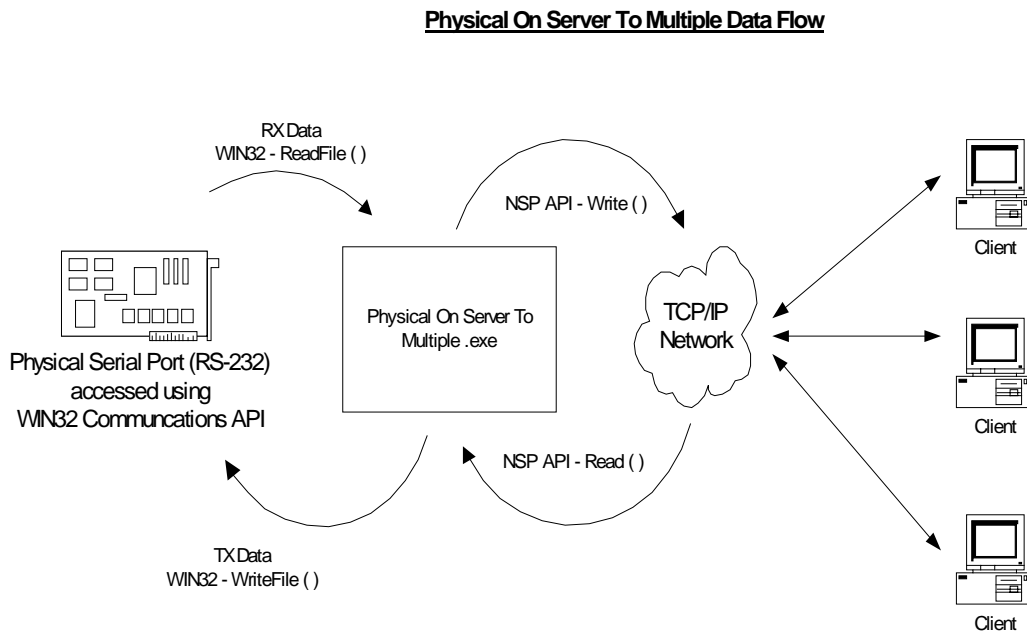
```
ErrorCode = hNsp.OpenConnectionToServer  
            (gNetworkAddress, gSocketNumber,  
            &gClientConnection);
```

7.3 Physical On Server To Multiple Reference Design

The *Physical On Server To Multiple (POSTM)* reference design demonstrates the techniques of expressing the data flow of a physical serial port to a single network endpoint using “*Server style*” network access techniques. An engineer may wish to use this sample as a reference design (starting point) for a serial port data redirector.

7.3.1 External Data Flow and Construction

Consider the following “external” data flow diagram of *POSTM*:

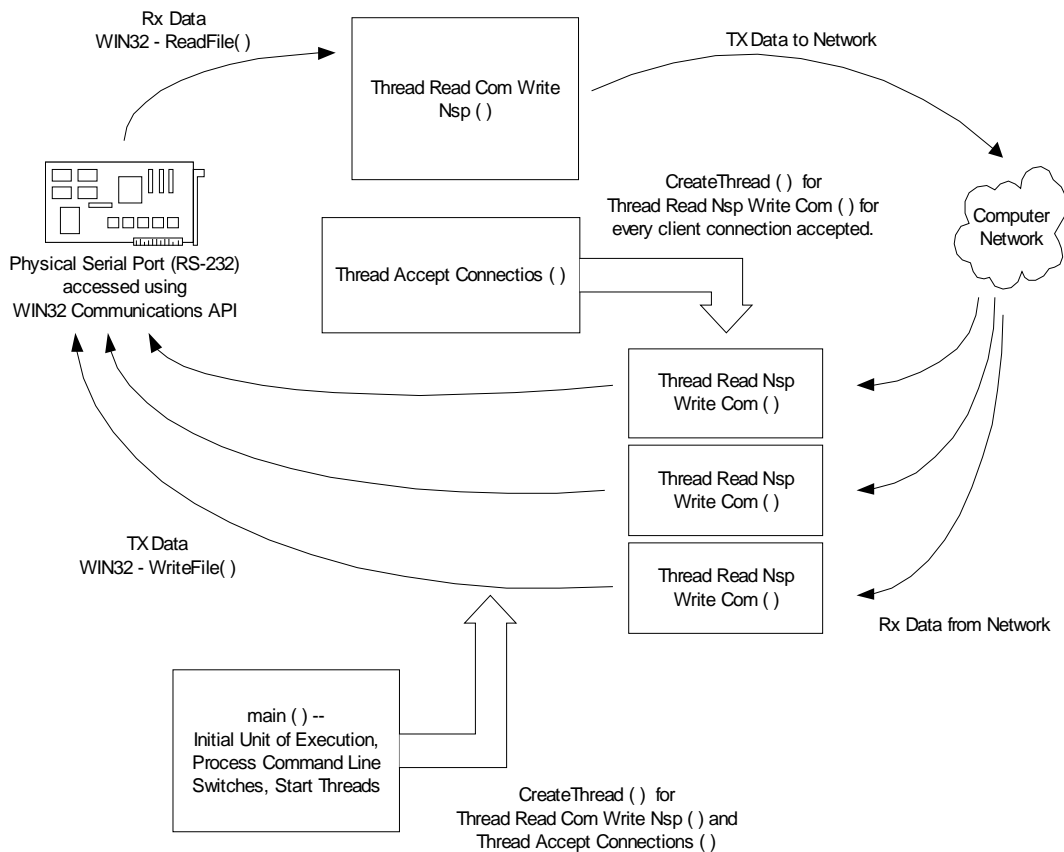


7.3.2 Internal Data Flow and Construction

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

<u>Paradigm</u>	<u>Usage</u>
WIN32	Thread creation and synchronization.
Stdio	Console operations using printf (), misc. string operations, etc.
NSP API	Access to network resources

The following data flow diagram illustrates the “internal” data flow of the *Physical On Server To Multiple* reference design:



7.3.3 Module / Function Software Description

The primary source module is “PhysicalOnServerToMultiple.cpp”, which consists of about 600 lines of C/C++ code. From the “main ()” entry point, the following operations are performed:

1. The function *OpenPhysicalPort ()* prepares the physical port named on the command line for operation using the following industry standard WIN32 functions: *CreateFile ()*, *SetCommTimeouts ()*, *GetCommState ()*, *SetCommState ()*. The global DCB named *gDcb* is prepared.
2. The function *ProcessCommandLineSwitches ()* function processes all command line parameters which are prefaced by a slash (/), and the *gDcb* variable is modified accordingly.
3. The physical device is prepared and timeout data is setup from *gDcb* by the function: *SetupPhysicalPortDcbAndTO ()*.
4. The socket number is stored in the *gSocketNumber* global variable.
5. Driver and DLL version comparison is performed. It is strongly suggested that the underlying NSP API DLL, and the application software (reference design, utility, all conform to the same version).
6. A thread is created; The *ThreadReadComWriteNsp ()* thread. The nomenclature suggests what this thread performs.
7. The Network data path is prepared with the following NSP API call:

```
ErrorCode = hNsp.OpenConnectionToClient (gSocketNumber,  
&gClientConnection);
```

8. A thread is then created for every connection; The *ThreadReadNspWriteCom ()* thread. The nomenclature suggests what this thread performs.

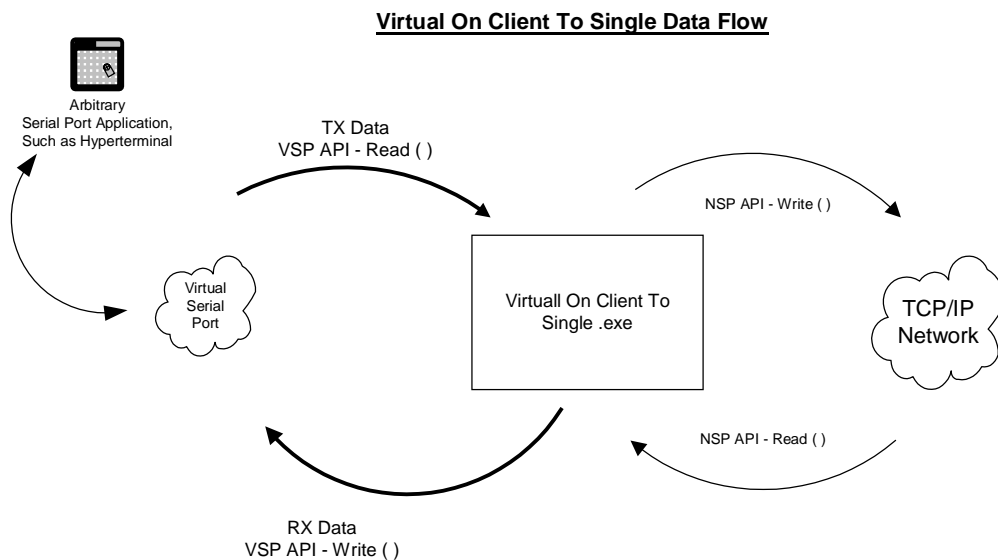
After the threads are created, the data movement is in progress, and continues until the operator selects termination. The operator typing ‘Q’ on the keyboard initiates termination. Threads are then terminated, and the corresponding devices are closed.

7.4 Virtual On Client To Single Reference Design

The *Virtual On Client To Single (VOCTS)* reference design demonstrates the techniques of expressing the data flow of a virtual serial port to a single network endpoint using “*Client style*” network access techniques. An engineer may wish to use this sample as a reference design (starting point) for a serial port data redirector.

7.4.1 External Data Flow and Construction

Consider the following “external” data flow diagram of VOCTS:

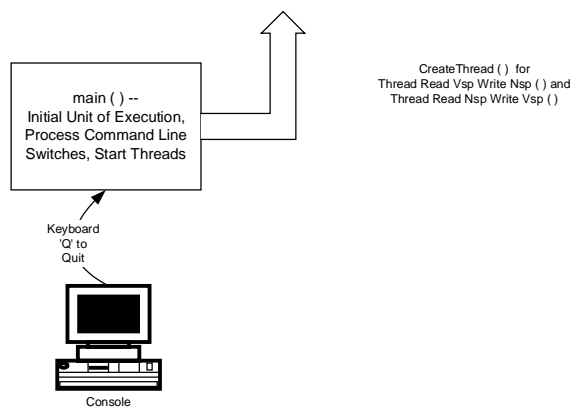
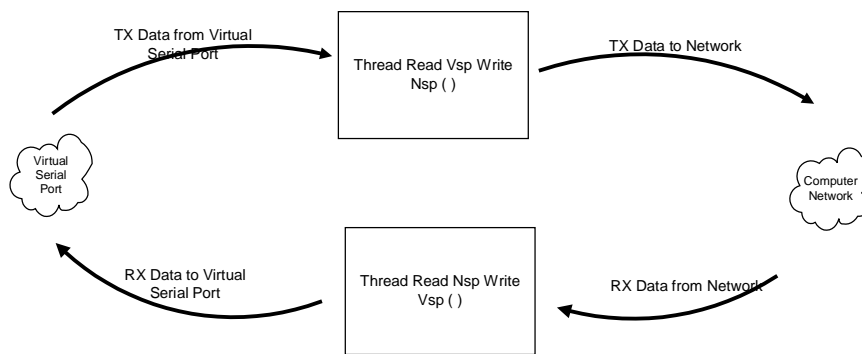


7.4.2 Internal Data Flow and Construction

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

<u>Paradigm</u>	<u>Usage</u>
WIN32	Thread creation and synchronization.
VSP API	Access to virtual communications device.
Stdio	Console operations using printf (), misc. string operations, etc.
NSP API	Access to network resources

The following data flow diagram illustrates the “internal” data flow of the *Virtual On Client To Single* reference design:



7.4.3 Module / Function Software Description

The primary source module is “VirtualOnClientToSingle.cpp”, which consists of about 600 lines of C/C++ code. From the “main ()” entry point, the following operations are performed:

1. The function *ProcessCommandLineSwitches ()* function processes all command line parameters which are prefaced by a slash (/), and several global variables (*gMonitorHex*, *gMonitorAscii*, *gShowInformation*, etc) are modified accordingly.
2. The Network address and socket numbers are stored in the following global variables: *gNetworkAddress*, *gSocketNumber*.
3. Driver and DLL version comparison is performed. It is strongly suggested that the underlying NSP API DLL, and the application software (reference design, utility, all conform to the same version).
4. The Virtual Serial Port is prepared with the following VSP API call:

```
ErrorCode = hVsp.Open (argv[1]);
```

5. The Network data path is prepared with the following NSP API call:

```
ErrorCode = hNsp.OpenConnectionToServer (gNetworkAddress,  
gSocketNumber, &gClientConnection);
```

6. Two threads are then created; The *ThreadReadVspWriteNsp ()*, and *ThreadReadNspWriteVsp ()* thread. The nomenclature suggests what these threads perform.
7. After the threads are created, the data movement is in progress, and continues until the operator selects termination. The operator typing ‘Q’ on the keyboard initiates termination. Threads are then terminated, and the corresponding devices are closed.

7.5 Virtual On Server To Single Reference Design

The *Virtual On Server To Single (VOSTS)* reference design demonstrates the techniques of expressing the data flow of a virtual serial port to a single network endpoint using “*Server style*” network access techniques. An engineer may wish to use this sample as a reference design (starting point) for a serial port data redirector.

Important Point:	The External and Internal data flow and construction of VOSTS are virtually identical to that of VOCTS. For that reason, the remainder of this discussion shall focus on the differences between the <i>Client</i> vs. <i>Server</i> access techniques utilized.
------------------	--

7.5.1 Client vs. Server Access Techniques in VOSTS vs. VOCTS

The *Server* NSP applications make use of the *OpenConnectionToClient* () function to make a connection. The *Server* NSP application will block until a *Client* application connects. The following statement from VOSTS makes the critical connection from the local *Server* to the remote *Client*:

```
ErrorCode = hNsp.OpenConnectionToClient  
            (gSocketNumber, &gClientConnection);
```

Whereas, in VOCTS, the following statement makes the critical connection from the local *Client* (VOCTS) to the remote *Server*:

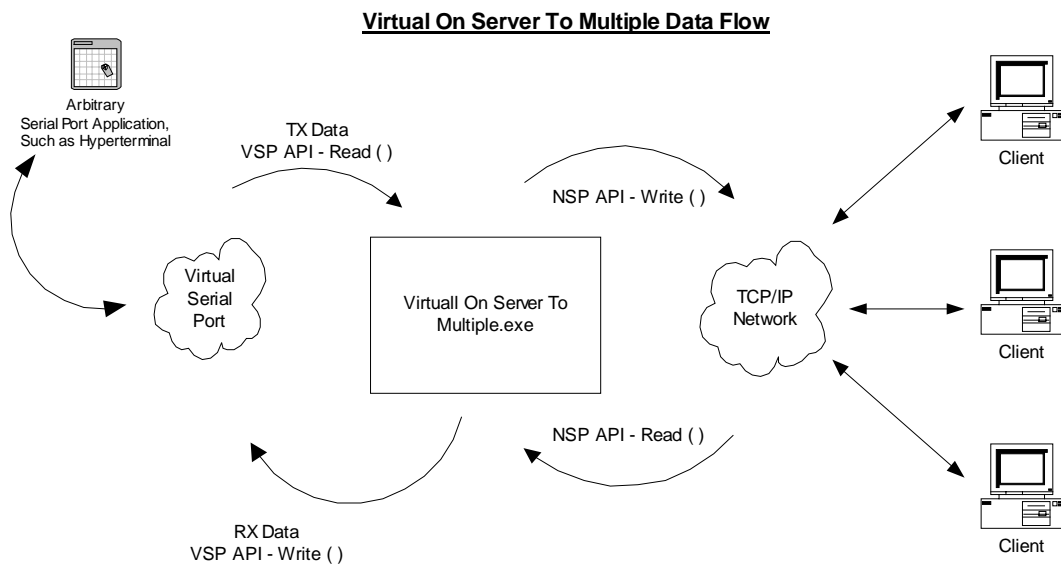
```
ErrorCode = hNsp.OpenConnectionToServer  
            (gNetworkAddress, gSocketNumber,  
            &gClientConnection);
```

7.6 Virtual On Server To Multiple Reference Design

The *Virtual On Server To Multiple (VOSTM)* reference design demonstrates the techniques of expressing the data flow of a virtual serial port to a single network endpoint using “*Server style*” network access techniques. An engineer may wish to use this sample as a reference design (starting point) for a serial port data redirector.

7.6.1 External Data Flow and Construction

Consider the following “external” data flow diagram of *VOSTM*:

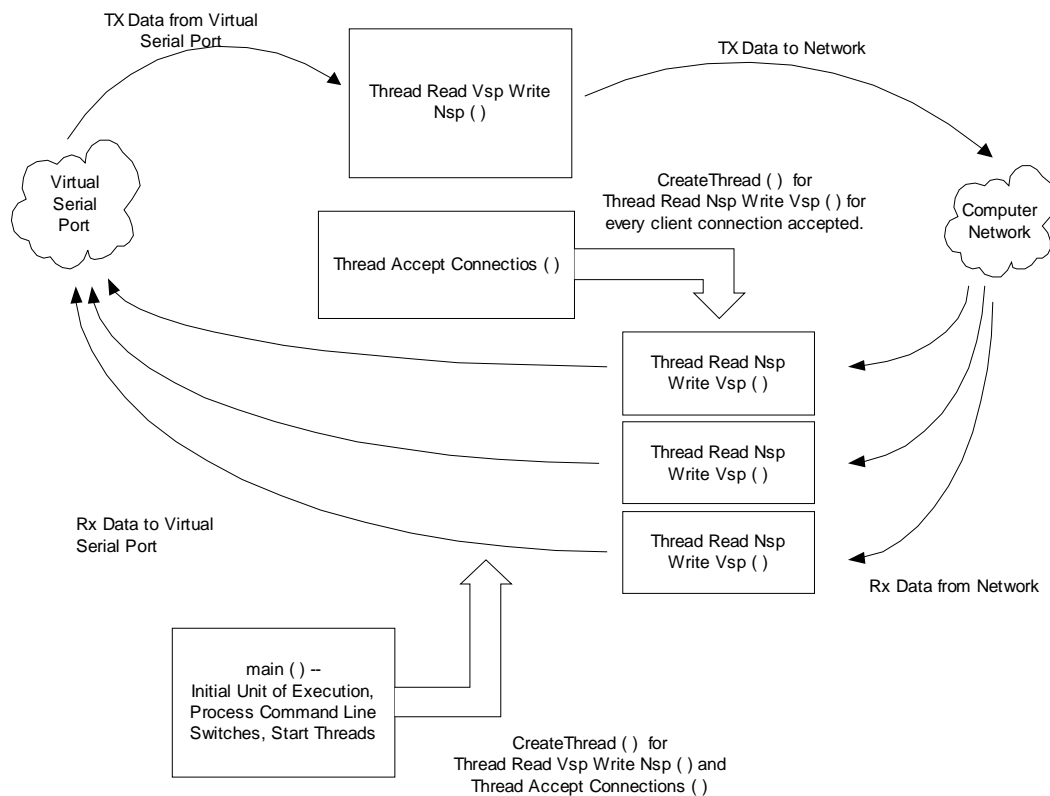


7.6.2 Internal Data Flow and Construction

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

<u>Paradigm</u>	<u>Usage</u>
WIN32	Thread creation and synchronization.
VSP API	Access to virtual communications device.
Stdio	Console operations using printf (), misc. string operations, etc.
NSP API	Access to network resources

The following data flow diagram illustrates the “internal” data flow of the *Virtual On Server To Multiple* reference design:



7.6.3 Module / Function Software Description

The primary source module is “VirtualOnServerToMultiple.cpp”, which consists of about 600 lines of C/C++ code. From the “main ()” entry point, the following operations are performed:

9. The function *ProcessCommandLineSwitches ()* function processes all command line parameters which are prefaced by a slash (/), and several global variables (*gMonitorHex*, *gMonitorAscii*, *gShowInformation*, etc) are modified accordingly.
10. The virtual COM port and socket numbers are stored in the following global variables: *gPortName*, *gSocketNumber*.
11. Driver and DLL version comparison is performed. It is strongly suggested that the underlying NSP API DLL, and the application software (reference design, utility, all conform to the same version).
12. The Virtual Serial Port is prepared with the following VSP API call:

```
ErrorCode = hVsp.Open (argv[1]);
```

13. A thread is created; The *ThreadReadVspWriteNsp ()* thread. The nomenclature suggests what this thread performs.
14. The Network data path is prepared with the following NSP API call:

```
ErrorCode = hNsp.OpenConnectionToClient (gSocketNumber,  
&gClientConnection);
```
15. A thread is then created for every connection; The *ThreadReadNspWriteVsp ()* thread. The nomenclature suggests what this thread performs.

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

8. Selected WIN32 References

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

8.1 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.

8.2 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. 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.
RTS_CONTROL_TOGGLE	

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.

8.3 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.

8.4 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.

9. 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.

This manual, information, technology and software is protected by copyright law and international treaties. Unauthorized reproduction or distribution may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent.

10. Index of Acronyms and Abbreviations

API	Applications Programming Interface
DOS	Disk Operating System
DCB	WIN32 Device Control Block
CDS	Constellation Data Systems
DLL	Dynamic Link Library
GPS	Global Positioning System
HyperTerminal	Standard Windows Communications Application
MS	Microsoft
MSDN	MS Developers Network
NSP	Network Serial Port
PCR	Physical Communications Resource (Such as a UART)
RX	Receive
SDK	Software Development Kit
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