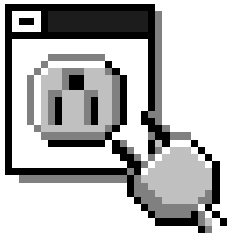
Windows Sockets 2 Service Provider Interface

A Service Provider Interface for Transparent Network Programming under Microsoft Windows

Revision 0.4
December 9, 1994



Winsock 2

Subject to Change Without Notice

Draft

Disclaimer

Microsoft, Intel, and JSB disclaim all warranties and liability for the use of this document and the information contained herein, and assume no responsibility for any errors which may appear in this document. Microsoft, Intel, and JSB make no warranty or license regarding the relationship of this document and the information contained herein to the intellectual property rights of any party. Microsoft, Intel, and JSB make no commitment to update the information contained herein.

Winsock 2.0 Service Provider Interface

TABLE OF CONTENTS

TABLE OF CONTENTS iii
1. INTRODUCTION 4
1.1 Winsock Specification is a WOSA Component 4
1.1.1 Winsock 2 DLLs 5
1.2 Microsoft Windows and Windows-specific extensions 5
1.3 Naming Conventions 5
2. OVERVIEW 7
2.1 Configuration of Winsock Service Providers 7
2.1.1 Registry Layout for Windows NT and Windows 95 7
2.1.1.1 Providers 7
2.1.1.2 Provider-specific Keys 7
2.1.2 WINSOCK2.INI Layout for Windows 3.1 9
2.1.2.1 Providers 9
2.1.2.2 Provider-specific Sections ([Provider-<Provider Key>]) 10
2.2. Service Providers Interface Model 12
2.3 Initialization of Winsock Service Providers 12
2.4 Functionality Implemented Within the Winsock DLL 12
2.5 Functionality Implemented by Service Providers 13
2.6 Differences and Similarities Between Winsock API and SPI 13
2.7 Differences Between 32-bit and 16-bit Winsock SPI 13
2.8 Sockets 14
2.8.1 Out-of-band data 14
2.8.2 Socket Options 14
2.9 Quality of Service (QOS) 16
2.9.1 Overall Approach 16
2.9.2 The Flow Spec Structure 16
3. SERVICE PROVIDER INTERFACE REFERENCE 18
3.1 Socket Routines 18
3.1.1 WSPBind() 19
3.1.2 WSPCloseSocket() 21
3.1.3 WSPGetPeerName() 22
3.1.4 WSPGetSockName() 23
3.1.5 WSPGetSockOpt() 24
3.1.6 WSPIoctlSocket() 28
3.1.7 WSPListen() 30
3.1.8 WSPSelect() 32
3.1.9 WSPSetSockOpt() 34
3.1.10 WSPShutdown() 38
3.1.11 WSPAccept() 40
3.1.12 WSPAsyncSelect32() 43
3.1.13 WSPCallbackSelect16() 49
3.1.14 WSPCancelBlockingCall32() 55
3.1.15 WSPCleanup() 57
3.1.16 WSPConnect() 59
3.1.17 WSPEnumNetworkEvents() 63
3.1.18 WSPEventSelect() 65
3.1.19 WSPIsBlocking32() 70
3.1.20 WSPRecv() 71

3.1.21 WSPRecvFrom() 74
3.1.22 WSPSend() 77
3.1.23 WSPSendTo() 80
3.1.24 WSPSetBlockingHook32() 84
3.1.25 WSPSocket() 86
3.1.26 WSPStartup() 88
3.1.27 WSPUnhookBlockingHook32() 91
4. Upcalls 92
4.1 WPUCreateSocketHandle() 93
4.2 WPUCloseSocketHandle() 94
4.3 WPUQuerySocketHandleContext() 95
4.4 WPUSetSocketHandleContext() 96
4.5 WPUQueueUserAPC32() 97
4.6 WPUGetCurrentThreadId32() 98
5. Installation APIs 99
5.1 WPUInstallProvider() 99
5.2 WPUDeinstallProvider() 100
Appendix A. Error Codes and Header Files 101
A.1 Error Codes 101
A.2 Winsock SPI Header File - ws2spi.h 103
Appendix B. Notes for Winsock Service Providers 118
B.1 Introduction 118
B.2 Winsock SPI Run Time Components 118
B.3 Error Codes 118
Appendix C. Outstanding Issues 119

1. INTRODUCTION

This document defines the Service Provider Interface (SPI) of Windows Socket (Winsock) 2.0. The Winsock SPI specifies the external interface of a service provider to be implemented by vendors of network protocol stacks. Installing a service provider allows Windows applications written to the Winsock 2 API interface to access the service provider's network protocol stacks. This will permit one or more applications to have access to multiple protocol stacks simultaneously.

1.1 Winsock Specification is a WOSA Component

The Winsock network transport services are provided as a WOSA (Windows Open Services Architecture) component. They consist of both an application programming interface (API) used by applications and a service provider interface (SPI) implemented by service providers. This document is designed to be a stand-alone reference for Winsock service provider developers. Readers intending to write network applications should refer to the document "*Windows Socket Interface, revision 2.0*" which describes Winsock 2 API.

Windows Open Service Architecture (WOSA) provides a single-level interface for connecting front-end applications with back-end services. The front-end application and back-end service need not speak each other's language in order to communicate as long as they both know how to talk to the WOSA interface. As a result, WOSA allows application developers and vendors of back-end services to mix and match applications and services to build solutions that shield programmers and users from the underlying complexity of the system. WOSA defines an abstraction layer to heterogeneous computing resources through the WOSA set of APIs. Because this set of APIs is extensible, new services and their corresponding APIs can be added as needed. Applications written to the WOSA APIs have access not only to all the various computing environments supported today, but also to all additional environments as they become available. Moreover, applications don't have to be modified in any way to enjoy this support.

Each service recognized by WOSA also has a set of interfaces that service-provider vendors use to take advantage of the seamless interoperability that WOSA provides. In order to provide transparent access for applications, each implementation of a particular WOSA service simply needs to support the functions defined by its service-provider interface.

WOSA uses a Windows dynamic-link library (DLL) that allows software components to be linked at runtime. In this way, applications are able to connect to services dynamically. An application needs to know only the definition of the interface, not its implementation.

1.1.1 Winsock 2 DLLs

Winsock network services follow the WOSA model. This means that there exists a Winsock Application Programming Interface (API), which is the application programmer's access to network services, a Winsock Service Provider Interface (SPI) which is implemented by network service provider vendors, and the Winsock DLL. For 16 bit applications this DLL is referred to as WINSOCK2.DLL while 32 bit applications use WSOCK32.DLL. Note that in Windows 3 environments, only Winsock2.DLL will be available, while Windows 95 and Windows NT support both DLLs. Winsock 2's WOSA compliant architecture is illustrated below in Figure 1.

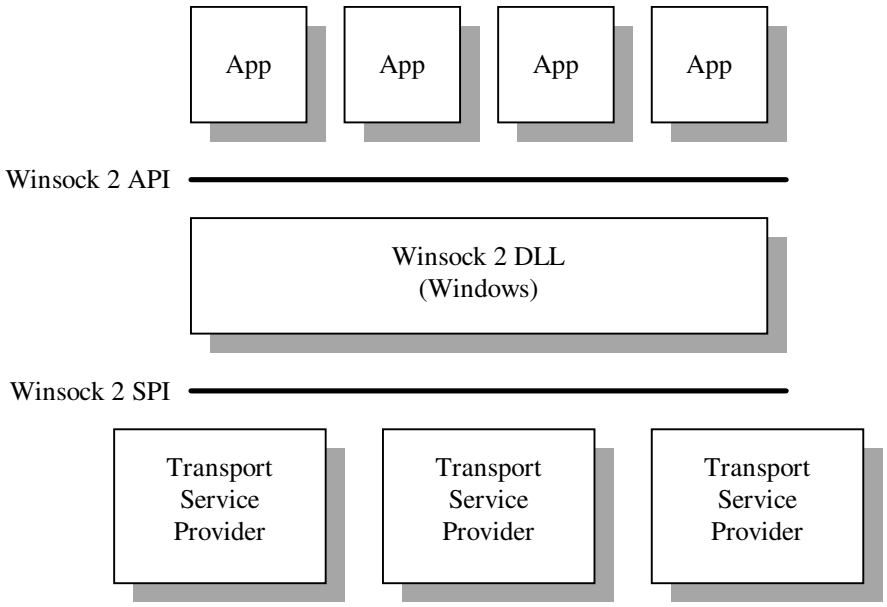


Figure 1

1.2 Microsoft Windows and Windows-specific extensions

This SPI is intended to be usable within all implementations and versions of Microsoft Windows including Windows NT, Windows 95, Windows 3.11, and Windows 3.1.

Winsock makes provisions for multithreaded Windows processes. In the non-preemptive Win16 environment, a task corresponds to a process with a single thread of execution. In the preemptive Win32 environment, a process contains one or more threads of execution.

1.3 Naming Conventions

The Winsock SPI uses the same naming conventions for functions, messages, and parameters as those used by the Winsock API except that all the functions are prefixed by **WSP**, which stands for **W**insock **S**ervice **P**rovider.

SPI functions applicable only to a particular platform end in an identifying suffix. For example, **WSPCallbackSelect16()** is applicable only to 16-bit Windows, and **WSPAsyncSelect32()** is applicable only to 32-bit Windows.

"Upcalls" (utility functions made available by the Winsock DLL for use by SPI DLLs and installation programs) are prefixed by **WPU**, which stands for **W**insock **P**rovider **U**pcall.

2. OVERVIEW

2.1 Configuration of Winsock Service Providers

When a service provider is installed, some configuration information must be added to a configuration database to give the Winsock DLL required information regarding the service provider. A service provider vendor must arrange for this information to be updated as part of the service provider installation.

Under Windows NT and Windows 95, this configuration information is stored in the system registry. Under Windows 3.1 and 3.11, this configuration information is stored in the WINSOCK2.INI file.

2.1.1 Registry Layout for Windows NT and Windows 95

All provider configuration information is stored under the following registry key:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Winsock2
```

Throughout this document, all references to relative subkeys and values are assumed to exist under this main Winsock2 registry key. For example, any reference to the "Providers" subkey is actually a reference to the "HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Winsock2\Providers" key.

2.1.1.1 Providers

The "Providers" subkey contains one subkey for each installed service provider. The maximum length of provider subkey names is 32 characters. These subkeys are created by any software that installs service providers.

The "ProviderOrder" subkey contains zero or more values used to locate the active service providers. These values also establish a priority ordering of service providers. The name of each value represents an active service provider. For example:

```
\ProviderOrder
      Microsoft TCP/IP = REG_SZ ""
      novlwp = REG_SZ ""
      TC&S = REG_SZ ""
```

2.1.1.2 Provider-specific Keys

Each Winsock service provider must have a unique subkey under the "Providers" subkey. It is the responsibility of the service provider to create its provider-specific subkey at the time it is installed. It is also the responsibility of the service provider to delete its provider-specific subkey at the time it is removed. Only the service provider writes values in its provider-specific subkey. The Winsock DLL reads only the following values in the provider-specific subkey. Other values in the subkey are permitted for internal use by the service provider. This subkey consists, at a minimum, of the following:

```
\Provider\<Provider Name>
ProviderFilename = REG_SZ "<filename>"
ProviderDescription = REG_SZ "<Description of the provider>"
ProviderDomain = REG_DWORD <Address family>
ProviderAddrLength = REG_DWORD <length>
NumSockets = REG_DWORD <count>  { PII model implied }
SocketType<index> = REG_DWORD <Socket Type>
```

```
AddressFamily<index> = REG_DWORD <Address Family>
Protocol<index> = REG_DWORD <Protocol>
MinimumAddressLength<index> = REG_DWORD <Minimum Address Length>
MaximumAddressLength<index> = REG_DWORD <Maximum Address Length>
ProtocolAttributes<index> = REG_DWORD <Protocol Attributes>
ProtocolDescription<index> = REG_SZ "<Description of the
protocol>"
```

For example:

```
\Provider\novlwp
ProviderFilename = REG_SZ "novlwp.WSP"
ProviderDescription = REG_SZ "Novell LAN WorkPlace for DOS"
 ProviderAddrLength = REG_DWORD 8
NumSockets = REG_DWORD 3
SocketType0 = REG_DWORD 1
AddressFamily0 = REG_DWORD 2
Protocol0 = REG_DWORD 2
MinimumAddressLength0 = REG_DWORD 16
MaximumAddressLength0 = REG_DWORD 16
ProtocolAttributes0 = REG_DWORD 0x00001066
ProtocolDescription0 = REG_SZ "Stream socket in INET domain"
SocketType1 = REG_DWORD 2
AddressFamily1 = REG_DWORD 2
Protocol1 = REG_DWORD 2
MinimumAddressLength1 = REG_DWORD 16
MaximumAddressLength1 = REG_DWORD 16
ProtocolAttributes1 = REG_DWORD 0x00001609
ProtocolDescription1 = REG_DSZ "Dgram socket in INET domain"
SocketType2 = REG_DWORD 3
AddressFamily2 = REG_DWORD 2
Protocol2 = REG_DWORD 2
MinimumAddressLength2 = REG_DWORD 16
MaximumAddressLength2 = REG_DWORD 16
ProtocolAttributes2 = REG_DWORD 0x00000109
ProtocolDescription2 = REG_SZ"Raw socket in INET domain"
```

The `ProviderFilename` value identifies the filename of the service provider DLL.

The `ProviderDescription` value is the descriptive text of this service provider.

. The `ProviderAddrLength` value is the (maximum) length of addresses (in bytes) used in the address family specified in `ProviderDomain`. The length includes the two-byte address family and the size of the address family specific address.

The `NumSockets` value indicates how many different types of sockets are supported, and how many `SocketType<index>`, `AddressFamily<index>`, `Protocol<index>`, and `ProtocolDescription<index>` values appear in the section.

The `SocketType<index>` value (one per socket type) identifies the type of this socket as defined in "ws2spi.h".

The `AddressFamily<index>` value (one per socket type) identifies the address family of this socket as defined in "ws2spi.h".

The `Protocol<index>` value (one per socket type) identifies the protocol of this socket as defined in "ws2spi.h".

The `MinimumAddressLength<index>` value (one per socket type) specifies the minimum BYTE size of a valid address for this protocol.

The `MaximumAddressLength<index>` value (one per socket type) specifies the maximum BYTE size of a valid address for this protocol.

The `ProtocolAttributes<index>` value (one per socket type) identifies the behavioral characteristics of the protocol. This value consists of any number of XP_xxx values ORed together. (The XP_xxx values are defined in the Registration and Resolution API specification.)

The `ProtocolDescription<index>` value (one per socket type) is the descriptive text for this protocol.

In the `SocketType<index>`, `AddressFamily<index>`, `Protocol<index>`, and `ProtocolDescription<index>` values, `<index>` takes the values from 0 to one less than `<count>` (as specified in `NumSockets`). These values are created by any software that installs the service provider. Although all these values must be present, there is no requirement that they appear in numerical order.

The service provider may include any other values/subkeys in its subkey which are necessary to fulfill the responsibilities of the provider as specified in the Winsock Service Provider Interface Specification, or for other private configuration purposes.

2.1.2 WINSOCK2.INI Layout for Windows 3.1

The WINSOCK2.INI file contains a wide variety of information, and two kinds of identifiers are used:

1. **name-sequence numbers**. These numbers simply provide a convenient way to index through a list of similar entries in a WINSOCK2.INI file section. For example, the list of "ProviderKey" entries in the [Providers] section uses the names ProviderKey0, ProviderKey1, ProviderKey2, ... as a set of easily-indexed names for the entries. The scope of these name-sequence numbers is strictly limited to the section in which they appear. These numbers are unrelated to anything outside the section. They may be completely renumbered whenever the WINSOCK2.INI configuration changes.

2. **permanent string identifiers**. These strings are permanent identifiers for entries. They typically appear in several different WINSOCK2.INI file sections to identify a relationship between entries in those two sections. For example, each provider-specific section ([Provider-<Provider key>]) includes the permanent string identifier called "provider key" of the provider in the section name. This indicates that the entry corresponds to the service provider with the matching permanent string identifier in the [Providers] section. The scope of these string identifiers includes the entire WINSOCK2.INI and persists until such identifiers are explicitly changed. Some of these identifiers can be retrieved through the Winsock SPI interface.

2.1.2.1 Providers

The “[Providers]” section of WINSOCK2.INI specifies the total number of the installed Winsock service providers, and the provider key for each service provider. This information is used so that Winsock2.DLL can identify and load each provider.

These entries in this section have the following format:

```
[Providers]
NumProviders=<count>
ProviderKey<index>="<Provider Key>"
```

For example:

```
[Providers]
NumProviders=2
ProviderKey0="novlwp"
ProviderKey1="FTP"
```

The NumProviders entry indicates how many service providers are installed, and how many ProviderKey<*index*> entries appear in the section. <*count*> is set to 0 when the Winsock implementation is initialized; the values are subsequently updated by any software that installs or removes service providers.

Each ProviderKey<*index*> entry (one per provider) specifies an unique provider key with which the service provider can identify itself. <*index*> takes the values from 0 to one less than <*count*> (as specified in NumProviders). The maximum length of the provider key is 32 characters. These entries are created by any software that installs service providers, and must be renumbered as providers are deleted. Although all these entries must be present, there is no requirement that they appear in numerical order. This provider key is also used to link parameters in other sections for provider-specific information.

2.1.2.2 Provider-specific Sections ([Provider-<Provider Key>])

For each Winsock service provider defined in the [Providers] section, there must be a [Provider-<Provider Key>] section. The <Provider Key> value is the provider key defined for that service provider in the corresponding ProviderKey<index>="<Provider Key>" entry in the [Providers] section. For example, corresponding to an entry such as ProviderKey0="novlwp" there would be a section named [Provider-novlwp].

It is the responsibility of the service provider to create its provider-specific section in WINSOCK2.INI at the time it is installed. It is also the responsibility of the service provider to delete its provider-specific section in WINSOCK2.INI at the time it is removed. Only the service provider writes entries in its provider-specific section. Winsock2.DLL reads only the following entries in the provider-specific section. Other entries in the section are permitted for internal use by the service provider. This section consists, at a minimum, of the following:

```
[Provider-<Provider Key>]
ProviderFilename="<filename>"
ProviderDescription="<Description of the provider>"
 ProviderAddrLength=<length>
NumSockets=<count>
SocketType<index>=<Socket Type>
AddressFamily<index>=<Address Family>
Protocol<index>=<Protocol>
MinimumAddressLength<index>=<Minimum Address Length>
MaximumAddressLength<index>=<Maximum Address Length>
ProtocolAttributes<index>=<Protocol Attributes>
ProtocolDescription<index>="<Description of the protocol>"
```

For example:

```
[Provider-novlwp]
ProviderFilename="novlwp.WSP"
```

```
ProviderDescription="Novell LAN WorkPlace for DOS"
 ProviderAddrLength=8
NumSockets=3
SocketType0=1
AddressFamily0=2
Protocol0=2
MinimumAddressLength0 = 16
MaximumAddressLength0 = 16
ProtocolAttributes0 = 0x00001066
ProtocolDescription0="Stream socket in INET domain"
SocketType1=2
SocketDescription1="Dgram socket in INET domain"
SocketType2=3
MinimumAddressLength0 = 16
MaximumAddressLength0 = 16
ProtocolAttributes0 = 0x00001609
ProtocolDescription2="Raw socket in INET domain"
```

The `ProviderFilename` entry identifies the filename of the service provider DLL.

The `ProviderDescription` entry is the descriptive text of this service provider.

. The `ProviderAddrLength` entry is the (maximum) length of addresses (in bytes) used in the address family specified in `ProviderDomain`. The length includes the two-byte address family and the size of the address family specific address.

The `NumSockets` entry indicates how many different types of sockets are supported, and how many `SocketType<`*`index`*`>`, `AddressFamily<`*`index`*`>`, `Protocol<`*`index`*`>`, and `ProtocolDescription<`*`index`*`>` entries appear in the section.

The `SocketType<`*`index`*`>` entry (one per socket type) identifies the type of this socket as defined in "ws2spi.h".

The `AddressFamily<`*`index`*`>` entry (one per socket type) identifies the address family of this socket as defined in “ws2spi.h”.

The `Protocol<`*`index`*`>` entry (one per socket type) identifies the protocol of this socket as defined in “ws2spi.h”.

The `MinimumAddressLength<`*`index`*`>` value (one per socket type) specifies the minimum BYTE size of a valid address for this protocol.

The `MaximumAddressLength<`*`index`*`>` value (one per socket type) specifies the maximum BYTE size of a valid address for this protocol.

The `ProtocolAttributes<`*`index`*`>` value (one per socket type) identifies the behavioral characteristics of the protocol. This value consists of any number of XP_xxx values ORed together. (The XP_xxx values are defined in the Registration and Resolution API specification.)

The `ProtocolDescription<`*`index`*`>` entry (one per socket type) is the descriptive text for this protocol.

In the `SocketType<`*`index`*`>`, `AddressFamily<`*`index`*`>`, `Protocol<`*`index`*`>`, and `ProtocolDescription<`*`index`*`>` entries, `<`*`index`*`>` takes the values from 0 to one less than `<`*`count`*`>` (as specified in `NumSockets`). These entries are created by any software that installs the

service provider. Although all these entries must be present, there is no requirement that they appear in numerical order.

The service provider may include any other entries in its section which are necessary to fulfill the responsibilities of the provider as specified in the Winsock Service Provider Interface Specification, or for other private configuration purposes.

2.2. Service Providers Interface Model

Winsock service providers are DLLs with EXPORTED procedure entry points for the functions defined by the SPI. Service providers should have their file extension changed from ".DLL" to ".WSP". This requirement is not strict. A service provider will still operate with the Winsock DLL with any file extension.

The SPI defines an entry point for each Winsock-specific function. These procedures are EXPORTED procedures just as in any DLL. They must be exported by ordinal numbers, as defined in "ws2spi.h" for each function. The Winsock DLL calls these Winsock specific entry points once the service provider is loaded using the standard dynamic linkage mechanism for calling DLLs.

The entry points described above cover the instances in which the Winsock DLL invokes functions provided by the service provider. The SPI also defines several circumstances in which the service provider calls back into the Winsock DLL to inform the Winsock DLL of various occurrences.

2.3 Initialization of Winsock Service Providers

Over time, different versions may exist for the Winsock DLLs, applications, and service providers. New versions may define new features, new fields to data structures and bit fields, etc. Version numbers therefore indicate how to interpret various data structures.

To allow optimal mixing and matching of different versions of applications, versions of the Winsock DLL itself, and versions of service providers by different vendors, the SPI provides a version negotiation mechanism for the Winsock DLL and the service provider. This version negotiation is handled by **WSPStartup**(). Basically, the Winsock DLL passes to the service provider the highest version numbers it is compatible with. The service provider compares this with its own supported range of version numbers. If these ranges overlap, the service provider returns a value within the overlapping portion of the range as the result of the negotiation. Usually, this should be the highest possible value. If the ranges do not overlap, the two parties are incompatible and the function returns an error.

2.4 Functionality Implemented Within the Winsock DLL

The major task that the Winsock DLL does is to serve as a sort of "traffic manager" between service providers and applications. Consider several different service providers interacting with the same application. Each Provider interacts strictly with the Winsock DLL. The Winsock DLL takes care of merging streams of events from those service providers into a single stream directed at the application. It hides the details of arbitration and synchronization over the data structures holding this single stream. Service providers are unaware that any of this is happening. They do not need to be concerned about the details of cooperating with one another or even the existence of other service providers. By abstracting the service providers into a consistent DLL interface, the Winsock DLL can interact with a variety of providers regardless of the underlying protocol’s implementation technology.

In addition to its major "traffic manager" service, the Winsock DLL provides a number of other services such as socket descriptor management (in order to avoid conflicts and ambiguities between applications and service providers and among service providers), parameter validation for Winsock API and Winsock SPI,

version negotiation between applications and the Winsock DLL, as well as between the Winsock DLL and service providers. The Winsock DLL also realizes protocol enumeration, event objects, shared sockets, and the pseudo blocking mechanism for Windows 3 environments.. Note that the differences between 16 and 32 bit versions of Windows with respect to blocking, preemption and shared address spaces dictates that the mechanisms used to implement the 16 and 32 bit versions of Winsock 2 DLLs vary significantly as well. This in turn leads to certain SPI functions that are applicable only to either 16 or 32 bit Windows implementations.

2.5 Functionality Implemented by Service Providers

The Winsock DLL has no knowledge about how requests to service providers are realized; this is up to the service provider implementation. Service providers implement the actual transport protocol which includes such functions as . setting up connections, transferring data, exercising flow control and error control, etc. The implementation of such functions may differ greatly from one provider to another. Service providers hide the implementation-specific details of how network operations are accomplished.

To summarize: service providers implement the low-level network-specific protocols. The Winsock DLL provides the medium-level traffic management that interconnects these transport protocols with applications. Applications in turn provide the policy of how these traffic streams and network-specific operations are used to accomplish the functions desired by the user.

2.6 Differences and Similarities Between Winsock API and SPI

The Winsock SPI is similar to the Winsock API in that all the basic socket functions appear. Support functions like **htonl(), htons(), ntohl(), ntohs(), inet_addr()**, and **inet_ntoa()** are implemented in the Winsock DLL, and are not passed down to SPI.

If an extended version of a function and the original version of a function both exist in the API, only the extended version will show up in the SPI. For example, **connect**() and **WSAConnect**() will both map to **WSPConnect**(), **accept**() and **WSAAccept**() to **WSPAccept**(), and **socket**() and **WSASocket**() to **WSPSocket**().

Since error codes are returned along with SPI functions, equivalents of **WSAGetLastError()** and **WSASetLastError()** are not needed in the SPI.

Shared sockets, and Winsock service provider enumeration are both realized in the Winsock DLL, thus **WSADuplicateSocket**and **WSAEnumProtocols()** do not appear as SPI functions.

2.7 Differences Between 32-bit and 16-bit Winsock SPI

In 32-bit environments the service providers are responsible for implementing blocking behavior. All blocking hook related API functions such as **WSAIsBlocking()**, **WSACancelBlockingCall()**, **WSASetBlockingHook()**, and **WSAUnhookBlockingHook()** appear in SPI. All event object related API functions are implemented as #define macros, mapping the Winsock event functions to native Win32 event functions.

In 16-bit environments the Winsock DLL implements all blocking behavior (as pseudo blocking), and all SPI sockets are strictly non-blocking. All the blocking hook related API functions such as **WSAIsBlocking()**, **WSACancelBlockingCall()**, **WSASetBlockingHook()**, and **WSAUnhookBlockingHook()** do not appear in the 16 bit SPI because the blocking hook implementation is realized in Winsock2.DLL. Similarly, all the event object related API functions are also implemented in

Winsock2.DLL, e.g., **WSACreateEvent()**, **WSACloseEvent()**, **WSASetEvent()**, **WSAResetEvent(), WSAWaitForMultipleEvents()**, and **WSAGetOverlappedResult**().

2.8 Sockets 2.8.1 Out-of-band data

Note: The following discussion of out-of-band data, also referred to as TCP Urgent data, follows the model used in the Berkeley software distribution. Users and implementors should be aware of the fact that there are at present two conflicting interpretations of RFC 793 (in which the concept is introduced), and that the implementation of out-of-band data in the Berkeley Software Distribution does not conform to the Host Requirements laid down in RFC 1122. To minimize interoperability problems, applications writers are advised not to use out-of-band data unless this is required in order to interoperate with an existing service. Winsock suppliers are urged to document the out-of-band semantics (BSD or RFC 1122) which their product implements. It is beyond the scope of this specification to mandate a particular set of semantics for out-of-band data handling.

Note that the out-of-band data functionality is mainly inherited from the TCP/IP world, and may not be available to other communication domains supported by this specification.

The stream socket abstraction includes the notion of "out of band" data. Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is delivered to the user independently of normal data. The abstraction defines that the out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e., the urgent data is delivered in sequence with the normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to "peek" at out-of-band data.

An application may prefer to process out-of-band data "in-line", as part of the normal data stream. This is achieved by setting the socket option SO_OOBINLINE (see section 3.1., **WSPSetSockOpt()**). In this case, the application may wish to determine whether any of the unread data is "urgent" (the term usually applied to in-line out-of-band data). To facilitate this, the Winsock service provider will maintain a logical "mark" in the data stream to indicate the point at which the out-of-band data was sent. An application can use the SIOCATMARK **WSPIoctlSocket()** command (see section 3.1.6) to determine whether there is any unread data preceding the mark. For example, it might use this to resynchronize with its peer by ensuring that all data up to the mark in the data stream is discarded when appropriate.

2.8.2 Socket Options

The socket options supported by Winsock SPI are listed and described in the pages describing **WSPSetSockOpt()** and **WSPGetSockOpt()**. A summary of the available options and the default value for each is shown in the following table.

Value	Type	Meaning	Default	Note
SO_ACCEPTCONN	BOOL	Socket is **WSPListen()**ing.	FALSE unless a **WSPListen()** has been performed	get only
SO_BROADCAST	BOOL	Socket is configured for the transmission of broadcast messages.	FALSE	(i)
SO_DEBUG	BOOL	Debugging is enabled.	FALSE	(i)
SO_DONTLINGER	BOOL	If true, the SO_LINGER option is disabled.	TRUE	
SO_DONTROUTE	BOOL	Routing is disabled.	FALSE	(i)
SO_FLOWSPEC	char FAR *	The flow spec of this socket.	NULL	get only
SO_GROUP_FLOWSPEC	char FAR *	The flow spec of the socket group to which this socket belongs.	NULL	get only
SO_GROUP_ID	GROUP	The identifier of the group to which this socket belongs.	NULL	get only
SO_GROUP_PRIORITY	int	The relative priority for sockets that are part of a socket group.	0	(i)
SO_KEEPALIVE	BOOL	Keepalives are being sent.	FALSE	(i)
SO_LINGER	struct linger	Returns the current linger options.	*l_onoff* is 0	
SO_MAX_MSG_SIZE	unsigned int	Maximum size of a message for message-oriented socket types (e.g. DGRAM). Has no meaning for stream-oriented sockets.	Implementation dependent	get only
SO_OOBINLINE	BOOL	Out-of-band data is being received in the normal data stream.	FALSE	
SO_PROTOCOL_INFO	struct PROTOCOL_INFO	Description of protocol info for protocol that is bound to this socket.	Protocol dependent	get only
SO_RCVBUF	int	Buffer size for receives	Implementation dependent	(i)
SO_REUSEADDR	BOOL	The address to which this socket is bound can be used by others.	FALSE	
SO_SNDBUF	int	Buffer size for sends	Implementation dependent	(i)
SO_TYPE	int	The type of the socket (e.g. SOCK_STREAM).	As created via **WSPSocket()**	get only
PVD_CONFIG	char FAR *	An opaque data structure object containing configuration information of the service provider.	Implementation dependent	
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.	Implementation dependent	

Notes:

(i) An implementation may silently ignore this option on **WSPSetSockOpt()** and return a constant value for **WSPGetSockOpt()**, or it may accept a value for **WSPSetSockOpt()**

and return the corresponding value in **WSPGetSockOpt()** without using the value in any way.

2.9 Quality of Service (QOS)

2.9.1 Overall Approach

The basic QOS mechanism in Winsock descends from the flow specification (or "flow spec") as described by Craig Partridge in RFC 1363, dated September 1992. A brief overview of this concept is as follows:

Flow specs describe a set of characteristics about a proposed connection-oriented, unidirectional flow through the network. An application may associate a pair of flow specs with a socket at the time a connection request is made. Flow specs indicate parametrically what level of service is required and also stipulate whether the application is willing to be flexible if the requested level of service is not available. After a connection is established, the application may retrieve the flow specs associated with the socket and examine the contents to discover the level of service that the network is willing and/or able to provide. If the service provided is not acceptable, the application may close the socket and take whatever action is appropriate (e.g. scale back and ask for a lower quality of service, try again later, notify the user and exit, etc.)

Even after a flow is established, conditions in the network may change resulting in a reduction (or increase) in the available service level. A notification mechanism is included which utilizes the usual Winsock 2 notification techniques to indicate to the application that QOS levels have changed. The app should again retrieve the corresponding flow specs and examine them in order to discover what aspect of the service level has changed.

The flow specs proposed for Winsock 2 divide QOS characteristics into the following general areas:

1. Network bandwidth utilization - The manner in which the application's traffic will be injected into the network. This includes specifications for average bandwidth utilization, peak bandwidth, and maximum burst duration.

2. Latency - Upper limits on the amount of delay and delay variation that are acceptable.

3. Level of service guarantee - Whether or not an absolute guarantee is required as opposed to best effort. Note that providers which have no feasible way to provide the level of service requested are expected to fail the connection attempt.

4. Cost - This is a place holder for a future time when a meaningful cost metric can be determined.

5. Provider-specific parameters - The flow spec itself can be extended in ways that are particular to specific providers, and the assumed provider can be identified.

An application indicates its desire for a non-default flow spec at the time a connection request is made (see **WSPConnect ()** and **WSPAccept()**). Since establishing a flow spec'd connection is likely to involve cooperation and/or negotiation between intermediate routers and hosts, the results of a flow spec request cannot be determined until after the connection operation is fully completed. After this time, the application may use **getsockopt()** to retrieve the resulting flow spec structure so that it can determine what the network was willing and/or able to supply.

2.9.2 The Flow Spec Structure

The Winsock 2 flow spec structure is defined in Winsock2.h and is reproduced here.

```
typedef enum
{
       GuaranteedService,
       BestEffortService
} GUARANTEE;

typedef struct _flowparams
{
       int64          AverageBandwith;// In Bytes/sec
       int64          PeakBandwidth; // In Bytes/sec
       int64          BurstLength;   // In microseconds
       int64          Latency;       // In microseconds
       int64          DelayVariation;// In microseconds
       GUARANTEE      levelOfGuarantee;// Guaranteed or
                                     // Best Effort
       int32          CostOfCall;    // Reserved for future
                                     // use, must be set to 0
       int32          ProviderId;    // Provider Identifier
       int32          SizePSP;       // Length of provider
                                     // specific parameters
       UCHAR  ProviderSpecificParams[1];// provider specific
                                     // parameters
} FLOWPARAMS;

typedef struct _QualityOfService
{
       FLOWPARAMS     ForwardFP;     // Caller(Initiator) to callee
       FLOWPARAMS     BackwardFP;    // Callee to caller
} QOS, FAR * LPQOS;
```

3. SERVICE PROVIDER INTERFACE REFERENCE

3.1 Socket Routines

This chapter presents the service provider socket library routines in alphabetical order, and describes each routine in detail.

In each routine it is indicated that the header file **ws2spi.h** must be included. The Windows header file **windows.h** is also needed, but **ws2spi.h** will include it if necessary.

3.1.1 WSPBind()

Description Associate a local address with a socket.

#include <ws2spi.h>

int WSPAPI WSPBind (SOCKET *s,* **const struct sockaddr** FAR * *name*, **int** *namelen*, **int** FAR * *lpErrno* **);**

s A descriptor identifying an unbound socket.

name The address to assign to the socket. The sockaddr structure is defined as follows:

```
struct sockaddr {
      u_short     sa_family;
      char        sa_data[14];
};
```

namelen The length of the *name*.

lpErrno A pointer to the error code.

Remarks This routine is used on an unconnected connectionless or connection-oriented socket, before subsequent **WSPConnect()**s or **WSPListen()**s. When a socket is created with **WSPsocket()**, it exists in a name space (address family), but it has no name assigned. **WSPBind()** establishes the local association of the socket by assigning a local name to an unnamed socket.

As an example, in the Internet address family, a name consists of three parts: the address family, a host address, and a port number which identifies the application. In Winsock 2, the *name* parameter is not strictly interpreted as a pointer to a "sockaddr" struct. It is cast this way for Windows Sockets 1.1 compatibility. Service providers are free to regard it as a pointer to a block of memory of size *namelen*. The first two bytes in this block (corresponding to "sa_family" in the "sockaddr" declaration) must contain the address family that was used to create the socket. Otherwise an error WSAEFAULT will occur.

Return Value If no error occurs, **WSPBind()** returns 0. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The specified address is already in use. (See the SO_REUSEADDR socket option under **WSPSetSockOpt()**.)
WSAEFAULT	The *namelen* argument is too small, the *name* argument contains incorrect address format for the associated address family, or the first two bytes of the memory block specified by *name* does not match the address family associate with the socket descriptor *s*.

WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAEINVAL	The socket is already bound to an address.
WSAENOBUFS	Not enough buffers available, too many connections.
WSAENOTSOCK	The descriptor is not a socket.

See Also **WSPConnect()**, **WSPListen()**, **WSPGetSockName()**, **WSPSetSockOpt()**, **WSPsocket()**, .

3.1.2 WSPCloseSocket()

Description Close a socket.

#include <ws2spi.h>

int WSPAPI WSPCloseSocket (SOCKET *s*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying a socket.
lpErrno	A pointer to the error code.

Remarks This function closes a socket. More precisely, it releases the socket descriptor *s*, so that further references to *s* will fail with the error WSAENOTSOCK.

The semantics of **WSPCloseSocket()** are affected by the socket options SO_LINGER and SO_DONTLINGER as follows:

Option	Interval	Type of close	Wait for close?
SO_DONTLINGER	Don't care	Graceful	No
SO_LINGER	Zero	Hard	No

If SO_LINGER is set (i.e. the *l_onoff* field of the linger structure is non-zero; see sections 3.1.5 and 3.1.) with a zero timeout interval (*l_linger* is zero), **WSPCloseSocket()** is not blocked even if queued data has not yet been sent or acknowledged. This is called a "hard" or "abortive" close, because the socket's virtual circuit is reset immediately, and any unsent data is lost. Any **WSPRecv()** call on the remote side of the circuit will fail with WSAECONNRESET.

If SO_DONTLINGER is set on a stream socket (i.e. the *l_onoff* field of the linger structure is zero; see sections 3.1.5 and 3.1.), the **WSPCloseSocket()** call will return immediately. However, any data queued for transmission will be sent if possible before the underlying socket is closed. This is called a graceful disconnect. Note that in this case the Winsock service provider may not release the socket and other resources for an arbitrary period, which may affect applications which expect to use all available sockets.

Return Value If no error occurs, **WSPCloseSocket()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno***.**

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAENOTSOCK	The descriptor is not a socket.

See Also **WSPAccept**, **WSPsocket()**, **WSPIoctlSocket()**, **WSPSetSockOpt()**.

3.1.3 WSPGetPeerName()

Description Get the address of the peer to which a socket is connected.

#include <ws2spi.h>

int WSPAPI WSPGetPeerName (SOCKET *s*, **struct sockaddr** FAR * *name*, **int** FAR * *namelen*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying a connected socket.
name	The structure which is to receive the name of the peer.
namelen	A pointer to the size of the *name* structure.
lpErrno	A pointer to the error code.

Remarks **WSPGetPeerName()** retrieves the name of the peer connected to the socket *s* and stores it in the struct sockaddr identified by *name*. It is used on a connected socket.

On return, the *namelen* argument contains the actual size of the name returned in bytes.

Return Value If no error occurs, **WSPGetPeerName()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno***.**

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The namelen argument is not large enough.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.

See Also **WSPBind()**, **WSPsocket()**, **WSPGetSockName()**.

3.1.4 WSPGetSockName()

Description Get the local name for a socket.

#include <ws2spi.h>

int WSPAPI WSPGetSockName (SOCKET s**, struct sockaddr** FAR * *name*, **int** FAR * *namelen*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying a bound socket.
name	Receives the address (name) of the socket.
namelen	The size of the *name* buffer.
lpErrno	A pointer to the error code.

Remarks **WSPGetSockName()** retrieves the current name for the specified socket descriptor in *name*. It is used on a bound and/or connected socket specified by the *s* parameter. The local association is returned. This call is especially useful when a **WSPConnect()** call has been made without doing a **WSPBind()** first; as this call provides the only means by which the local association which has been set by the service provider can be determined.

On return, the *namelen* argument contains the actual size of the name returned in bytes.

If a socket was bound to an unspecified address (e.g., ADDR_ANY), indicating that any of the host's addresses within the specified address family should be used for the socket, **WSPGetSockName()** will not necessarily return information about the host address, unless the socket has been connected with **WSPConnect()** or **WSPAccept.** The Winsock DLL must not assume that the address will be specified unless the socket is connected. This is because for a multi-homed host the address that will be used for the socket is unknown unless the socket is connected.

Return Value If no error occurs, **WSPGetSockName()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno***.**

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *namelen* argument is not large enough, or the *name* or *namelen* argument is not part of the user address space.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The socket has not been bound to an address with **WSPBind()**, or ADDR_ANY is specified in **WSPBind()** but connection has not yet occurs.

See Also **WSPBind**(), **WSPsocket(), WSPGetPeerName**().

3.1.5 WSPGetSockOpt()

Description Retrieve a socket option.

#include <ws2spi.h>

int WSPAPI WSPGetSockOpt (SOCKET *s*, **int** *level*, **int** *optname*,
char FAR * *optval*, **int** FAR * *optlen*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying a socket.
level	The level at which the option is defined; the only supported *levels* are SOL_SOCKET and SOL_PROVIDER. (SOL_PROVIDER is defined to be an alias for IPPROTO_TCP for the sake of compatibility with Windows Sockets specification 1.1.)
optname	The socket option for which the value is to be retrieved.
optval	A pointer to the buffer in which the value for the requested option is to be returned.
optlen	A pointer to the size of the *optval* buffer.
lpErrno	A pointer to the error code.

Remarks **WSPGetSockOpt()** retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in *optval*. Options may exist at multiple protocol levels, but they are always present at the uppermost "socket" level. Options affect socket operations, such as the routing of packets, out-of-band data transfer, etc.

The value associated with the selected option is returned in the buffer *optval*. The integer pointed to by *optlen* should originally contain the size of this buffer; on return, it will be set to the size of the value returned. For SO_LINGER, this will be the size of a struct linger; for most other options it will be the size of an integer.

The Winsock DLL is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specifies.

If the option was never set with **WSPSetSockOpt(),** then **WSPGetSockOpt()** returns the default value for the option.

***level* = SOL_SOCKET**		
Value	Type	Meaning
SO_ACCEPTCONN	BOOL	Socket is **WSPListen**()ing.
SO_BROADCAST	BOOL	Socket is configured for the transmission of broadcast messages.
SO_DEBUG	BOOL	Debugging is enabled.
SO_DONTLINGER	BOOL	If true, the SO_LINGER option is disabled.
SO_DONTROUTE	BOOL	Routing is disabled.
SO_FLOWSPEC	char FAR *	The flow spec of this socket.

SO_GROUP_FLOWSPEC	char FAR *	The flow spec of the socket group to which this socket belongs.
SO_GROUP_ID	GROUP	The identifier of the group to which this socket belongs.
SO_GROUP_PRIORITY	int	The relative priority for sockets that are part of a socket group.
SO_KEEPALIVE	BOOL	Keepalives are being sent.
SO_LINGER	struct linger	Returns the current linger options.
SO_MAX_MSG_SIZE	unsigned int	Maximum size of a message for message-oriented socket types (e.g. DGRAM). Has no meaning for stream-oriented sockets.
SO_OOBINLINE	BOOL	Out-of-band data is being received in the normal data stream.
SO_PROTOCOL_INFO	struct PROTOCOL_INFO	Description of protocol info for protocol that is bound to this socket.
SO_RCVBUF	int	Buffer size for receives
SO_REUSEADDR	BOOL	The socket may be bound to an address which is already in use.
SO_SNDBUF	int	Buffer size for sends
SO_TYPE	int	The type of the socket (e.g. SOCK_STREAM).

***level* = SOL_PROVIDER (also aliased to IPPROTO_TCP)**		
PVD_CONFIG	Service Provider Dependent	An "opaque" data structure object from the service provider associated with socket *s*. This object stores the current configuration information of the service provider. The exact format of this data structure is service provider specific.
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.

BSD options not supported for **WSPGetSockOpt()** are:

Value	Type	Meaning
SO_ERROR	int	Get error status and clear
SO_RCVLOWAT	int	Receive low water mark
SO_RCVTIMEO	int	Receive timeout
SO_SNDLOWAT	int	Send low water mark
SO_SNDTIMEO	int	Send timeout
IP_OPTIONS		Get options in IP header.
TCP_MAXSEG	int	Get TCP maximum segment size.

Calling **WSPGetSockOpt()** with an unsupported option will result in an error code of WSAENOPROTOOPT being returned in *lpErrno*.

SO_DEBUG

Winsock service providers are encouraged (but not required) to supply output debug information if the SO_DEBUG option is set by an application. The mechanism for generating the debug information and the form it takes are beyond the scope of this specification.

SO_FLOWSPEC

This is a get-only socket option which indicates the flow spec of the socket. The default flow spec defined in Sec. 2.8.3 will be returned before the application sets the flow spec for this socket. The WSAENOPROTOOPT error code is indicated for service providers which do not support the flow spec option. See **WSPConnect()** about how to set the flow spec.

SO_GROUP_FLOWSPEC

This is a get-only socket option which indicates the flow spec of the group this socket belongs to. The default flow spec defined in Sec. 2.8.3 will be returned before the application sets the flow spec for this socket group. The WSAENOPROTOOPT error code is indicated for service providers which do not support the group flow spec option. If this socket does not belong to an appropriate socket group, the *Flen* and *Blen* fields of the returned QOS struct are set to 0. See **WSPConnect()** about how to set the flow spec.

SO_GROUP_ID

This is a get-only socket option which indicates the identifier of the group this socket belongs to. If this socket is not a group socket, the value is NULL.

SO_GROUP_PRIORITY

Group priority indicates the relative priority of the specified socket relative to other sockets within the socket group. Values are non-negative integers, with zero corresponding to the highest priority. Priority values represent a hint to the underlying service provider about how potentially scarce resources should be allocated. For example, whenever two or more sockets are both ready to transmit data, the highest priority socket (lowest value for SO_GROUP_PRIORITY) should be serviced first, with the remainder serviced in turn according to their relative priorities.

The WSAENOPROTOOPT error code is indicated for non group sockets or for service providers which do not support group sockets.

SO_KEEPALIVE

An application may request that a TCP/IP service provider enable the use of "keep-alive" packets on TCP-connections by turning on the SO_KEEPALIVE socket option. A Winsock provider need not support the use of keep-alives: if it does, the precise semantics are implementation-specific but should conform to section 4.2.3.6 of RFC 1122: *Requirements for Internet Hosts -- Communication Layers*. If a connection is dropped as the result of "keep-alives" the error code WSAENETRESET is returned to any calls in progress on the socket, and any subsequent calls will fail with WSAENOTCONN.

SO_LINGER

SO_LINGER controls the action taken when unsent data is queued on a socket and a **WSPCloseSocket()** is performed. See **WSPCloseSocket()** for a description of the way in which the SO_LINGER settings affect the semantics of **WSPCloseSocket()**. The application gets the desired behavior by creating a *struct linger* (pointed to by the *optval* argument) with the following elements:

```
struct linger {
      int   l_onoff;
      int   l_linger;
}
```

SO_MAX_MSG_SIZE

This is a get-only socket option which indicates the maximum size of a message for message-oriented socket types (e.g. DGRAM) as implemented by a particular service provider. It has no meaning for byte stream oriented sockets

SO_REUSEADDR

By default, a socket may not be bound (see **WSPBind()**) to a local address which is already in use. On occasions, however, it may be desirable to "re-use" an address in this way. Since every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the Winsock provider that a **WSPBind()** on a socket should not be disallowed because the desired address is already in use by another socket, the application should set the SO_REUSEADDR socket option for the socket before issuing the **WSPBind()**. Note that the option is interpreted only at the time of the **WSPBind()**: it is therefore unnecessary (but harmless) to set the option on a socket which is not to be bound to an existing address, and setting or resetting the option after the **WSPBind()** has no effect on this or any other socket.

. . **PVD_CONFIG**

This option retrieves an "opaque" data structure object from the service provider associated with socket *s*. This object stores the current configuration information of the service provider. The exact format of this data structure is service provider specific.

TCP_NODELAY

The TCP_NODELAY option disables the Nagle algorithm. The Nagle algorithm is used to reduce the number of small packets sent by a host by buffering unacknowledged send data until a full-size packet can be sent. However, for some applications this algorithm can impede performance, and TCP_NODELAY may be used to turn it off. Application writers should not set TCP_NODELAY unless the impact of doing so is well-understood and desired, since setting TCP_NODELAY can have a significant negative impact of network performance.

. .

Return Value

If no error occurs, **WSPGetSockOpt()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *optlen* argument was invalid.
WSAEINVAL	No value available for *optname* at the moment.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAENOPROTOOPT	The option is unknown or unsupported by the indicated protocol family.
WSAENOTSOCK	The descriptor is not a socket.

See Also

WSPSetSockOpt(),**WSPsocket()**.

3.1.6 WSPIoctlSocket()

Description Control the mode of a socket.

#include <ws2spi.h>

int WSPAPI WSPIoctlSocket (SOCKET *s*, **long** *cmd*, **u_long** FAR * *argp*,
int FAR * *lpErrno* **);**

s	A descriptor identifying a socket.
cmd	The command to perform on the socket *s*.
argp	A pointer to a parameter for *cmd*.
lpErrno	A pointer to the error code.

Remarks This routine may be used on any socket in any state. It is used to get or retrieve operating parameters associated with the socket, independent of the protocol and communications subsystem. The following commands are supported:

Command	Semantics
FIONREAD	Determine the amount of data which can be read atomically from socket *s*. *argp* points at an **unsigned long** in which **WSPIoctlSocket()** stores the result. If *s* is stream-oriented (e.g., type SOCK_STREAM), FIONREAD returns the total amount of data which may be read in a single **WSPRecv()**; this is normally the same as the total amount of data queued on the socket. If *s* is message-oriented (e.g., type SOCK_DGRAM), FIONREAD returns the size of the first datagram (message) queued on the socket.
SIOCATMARK	Determine whether or not all out-of-band data has been read. This applies only to a socket of stream style (e.g., type SOCK_STREAM) which has been configured for in-line reception of any out-of-band data (SO_OOBINLINE). If no out-of-band data is waiting to be read, the operation returns TRUE. Otherwise it returns FALSE, and the next **WSPRecv()** or **WSPRecvFrom()** performed on the socket will retrieve some or all of the data preceding the "mark"; the application should use the SIOCATMARK operation to determine whether any remains. If there is any normal data preceding the "urgent" (out of band) data, it will be received in order. (Note that a **WSPRecv()** or **WSPRecvFrom()** will never mix out-of-band and normal data in the same call.) *argp* points at a **BOOL** in which **WSPIoctlSocket()** stores the result.

Compatibility This function is a subset of **ioctl()** as used in Berkeley sockets. In particular, there is no command which is equivalent to FIOASYNC.
{where should we break the news about not allowing any new ioctls to be introduced? }
{Not here. This is more of a charter/requirements issue for the API extension group.}

Return Value Upon successful completion, the **WSPIoctlSocket()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno***.**

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	*cmd* is not a valid command, or *argp* is not an acceptable parameter for *cmd*, or the command is not applicable to the type of socket supplied
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAENOTSOCK	The descriptor *s* is not a socket.

See Also **WSPsocket()**, **WSPSetSockOpt()**, **WSPGetSockOpt().**

3.1.7 WSPListen()

Description Establish a socket to listen for incoming connection.

#include <ws2spi.h>

int WSPAPI WSPListen (SOCKET *s*, **int** *backlog*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying a bound, unconnected socket.
backlog	The maximum length to which the queue of pending connections may grow.
lpErrno	A pointer to the error code.

Remarks To accept connections, a socket is first created with **WSPsocket()**, a backlog for incoming connections is specified with **WSPListen()**, and then the connections are accepted with **WSPAccept**. **WSPListen()** applies only to sockets that are connection-oriented (e.g., SOCK_STREAM). The socket *s* is put into "passive" mode where incoming connections are acknowledged and queued pending acceptance by the process.

This function is typically used by servers that could have more than one connection request at a time: if a connection request arrives with the queue full, the client will receive an error with an indication of WSAECONNREFUSED.

Compatibility *backlog* is currently limited (silently) to SOMAXCONN, which is defined to be 5 in the header file. As in 4.3BSD, illegal values (less than 1 or greater than 5) are replaced by the nearest legal value.
{Why do we still enforce this? NT limits the backlog to 100, not 5, because BIG servers (like ftp.microsoft.com) often receive periodic floods of connection requests, easily overrunning a backlog of 5 connections.}

Return Value If no error occurs, **WSPListen()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	An attempt has been made to **WSPListen()** on an address in use.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAEINVAL	The socket has not been bound with **WSPBind()**.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.

WSAEOPNOTSUPP	The referenced socket is not of a type that supports the **WSPListen()** operation.

See Also **WSPAccept**, **WSPConnect**(), **WSPsocket**().

3.1.8 WSPSelect()

Description Determine the status of one or more sockets.

#include <ws2spi.h>

int WSPAPI WSPSelect (int *nfds***, fd_set** FAR * *readfds***, fd_set** FAR * *writefds***, fd_set** FAR * *exceptfds***, const struct timeval** FAR * *timeout*, **int** FAR * *lpErrno* **);**

nfds	This argument is ignored and included only for the sake of compatibility.
readfds	An optional pointer to a set of sockets to be checked for readability.
writefds	An optional pointer to a set of sockets to be checked for writability
exceptfds	An optional pointer to a set of sockets to be checked for errors.
timeout	. The maximum time for WSPSelect() to wait, or NULL for a blocking operation. Note that the 16-bit Winsock DLL will always pass in a timeout of zero to perform a non-blocking poll on the socket set(s). The 32-bit Winsock DLL may pass in any timeout value.
lpErrno	A pointer to the error code.

Remarks This function is . . used to determine the status of one or more sockets. For each socket, the caller may request information on read, write or error status. The set of sockets for which a given status is requested is indicated by an fd_set structure. All entries in an fd_set correspond to sockets created by the service provider. Upon return, the structure is updated to reflect the subset of these sockets which meet the specified condition, and **WSPSelect()** returns the total number of sockets meeting the conditions. A set of macros is provided for manipulating an fd_set. These macros are compatible with those used in the Berkeley software, but the underlying representation is completely different.

The parameter *readfds* identifies those sockets which are to be checked for readability. If the socket is currently **WSPListen()**ing, it will be marked as readable if an incoming connection request has been received, so that an **WSPAccept** is guaranteed to complete immediately. For other sockets, readability means that queued data is available for reading or, for connection-oriented sockets, that the virtual circuit corresponding to the socket has been closed, so that a **WSPRecv()** or **WSPRecvFrom()** is guaranteed to complete immediately. If the virtual circuit was closed gracefully, then a **WSPRecv()** will return immediately with 0 bytes read; if the virtual circuit was reset, then a **WSPRecv()** will complete immediately with the error code WSAECONNRESET. The presence of out-of-band data will be checked if the socket option SO_OOBINLINE has been enabled (see **WSPSetSockOpt()**).

The parameter *writefds* identifies those sockets which are to be checked for writability. If a socket is **WSPConnect()**ing, writability means that the connection establishment successfully completed. If the socket is not in the process of **WSPConnect()**ing, writability means that a **WSPSend()** or **WSPSendTo()** will complete immediately. [It is not specified how long this guarantee can be assumed to be valid, particularly in a multithreaded environment.]

The parameter *exceptfds* identifies those sockets which are to be checked for the presence of out-of-band data or any exceptional error conditions. Note that out-of-band data will only be reported in this way if the option SO_OOBINLINE is FALSE. For a connection-oriented socket, the breaking of the connection by the peer or due to KEEPALIVE failure will be indicated as an exception. This specification does not define which other errors will be included. If a socket is **WSPConnect()**ing, failure of the connect attempt is indicated in *exceptfds.*

Any of *readfds*, *writefds*, or *exceptfds* may be given as NULL if no descriptors are of interest.

Four macros are defined in the header file **ws2spi.h** for manipulating the descriptor sets. The variable FD_SETSIZE determines the maximum number of descriptors in a set. (The default value of FD_SETSIZE is 64, which may be modified by #defining FD_SETSIZE to another value before #including **ws2spi.h**.) Internally, an fd_set is represented as an array of SOCKETs; the last valid entry is followed by an element set to INVALID_SOCKET. The macros are:

FD_CLR(*s***,** ******set***)**	Removes the descriptor *s* from *set*.
FD_ISSET(*s***,** ******set***)**	Nonzero if *s* is a member of the *set*, zero otherwise.
FD_SET(*s***,** ******set***)**	Adds descriptor *s* to *set*.
FD_ZERO(**set***)**	Initializes the *set* to the NULL set.

Return Value **WSPSelect()** returns the total number of descriptors which are ready and contained in the fd_set structures, or SOCKET_ERROR if an error occurred. If the return value is SOCKET_ERROR, a specific error code is available in *lpErrno*.

Comments **WSPSelect ()** has no effect on the persistence of socket events registered with **WSPEventSelect()**.

Error Codes

WSAEFAULT	The Winsock service provider was unable to allocated needed resources for its internal operations, or the *readfds*, *writefds*, or *exceptfds* parameters are not part of the user address space.
WSAENETDOWN	The network subsystem has failed.
.	
WSAENOTSOCK	One of the descriptor sets contains an entry which is not a socket.

See Also **WSPAccept()**, **WSPConnect()**, **WSPRecv()**, **WSPRecvFrom()**, **WSPSend(), WSPSendTo(), WSPEventSelect()**

3.1.9 WSPSetSockOpt()

Description Set a socket option.

#include <ws2spi.h>

int WSPAPI WSPSetSockOpt (SOCKET *s*, **int** *level*, **int** *optname*,
const char FAR * *optval*, **int** *optlen*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying a socket.
level	The level at which the option is defined; the only supported *level*s are SOL_SOCKET and SOL_PROVIDER. (SOL_PROVIDER is defined to be an alias for IPPROTO_TCP for the sake of compatibility with Windows Sockets specification 1.1.)
optname	The socket option for which the value is to be set.
optval	A pointer to the buffer in which the value for the requested option is supplied.
optlen	The size of the *optval* buffer.
lpErrno	A pointer to the error code.

Remarks **WSPSetSockOpt()** sets the current value for a socket option associated with a socket of any type, in any state. Although options may exist at multiple protocol levels, they are always present at the uppermost "socket" level. Options affect socket operations, such as whether broadcast messages may be sent on the socket, etc.

There are two types of socket options: Boolean options that enable or disable a feature or behavior, and options which require an integer value or structure. To enable a Boolean option, *optval* points to a nonzero integer. To disable the option *optval* points to an integer equal to zero. *optlen* should be equal to sizeof(int) for Boolean options. For other options, *optval* points to the an integer or structure that contains the desired value for the option, and *optlen* is the length of the integer or structure.

***level* = SOL_SOCKET**		
Value	Type	Meaning
SO_BROADCAST	BOOL	Allow transmission of broadcast messages on the socket.
SO_DEBUG	BOOL	Record debugging information.
SO_DONTLINGER	BOOL	Don't block close waiting for unsent data to be sent. Setting this option is equivalent to setting SO_LINGER with *l_onoff* set to zero.
SO_DONTROUTE	BOOL	Don't route: send directly to interface.
SO_GROUP_PRIORITY	int	Specify the relative priority to be established for sockets that are part of a socket group.
SO_KEEPALIVE	BOOL	Send keepalives
SO_LINGER	struct linger	Linger on close if unsent data is present

SO_OOBINLINE	BOOL	Receive out-of-band data in the normal data stream.
SO_RCVBUF	int	Specify buffer size for receives
SO_REUSEADDR	BOOL	Allow the socket to be bound to an address which is already in use. (See **bind**().)
SO_SNDBUF	int	Specify buffer size for sends.

***level* = SOL_PROVIDER (aliased to IPPROTO_TCP)**		
PVD_CONFIG	Service Provider Dependent	This object stores the configuration information for the service provider associated with socket *s*. The exact format of this data structure is service provider specific.
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.

BSD options not supported for **WSPSetSockOpt()** are:

Value	Type	Meaning
SO_ACCEPTCONN	BOOL	Socket is listening
SO_ERROR	int	Get error status and clear
SO_RCVLOWAT	int	Receive low water mark
SO_RCVTIMEO	int	Receive timeout
SO_SNDLOWAT	int	Send low water mark
SO_SNDTIMEO	int	Send timeout
SO_TYPE	int	Type of the socket
IP_OPTIONS		Set options field in IP header.

SO_DEBUG

Winsock service providers are encouraged (but not required) to supply output debug information if the SO_DEBUG option is set by an application. The mechanism for generating the debug information and the form it takes are beyond the scope of this specification.

SO_GROUP_PRIORITY

Group priority indicates the relative priority of the specified socket relative to other sockets within the socket group. Values are non-negative integers, with zero corresponding to the highest priority. Priority values represent a hint to the underlying service provider about how potentially scarce resources should be allocated. For example, whenever two or more sockets are both ready to transmit data, the highest priority socket (lowest value for SO_GROUP_PRIORITY) should be serviced first, with the remainder serviced in turn according to their relative priorities.

The WSAENOPROTOOPT error is indicated for non group sockets or for service providers which do not support group sockets.

SO_KEEPALIVE

An application may request that a TCP/IP provider enable the use of "keep-alive" packets on TCP-connections by turning on the SO_KEEPALIVE socket option. A Winsock provider need not support the use of keep-alives: if it does, the precise semantics are implementation-specific but should conform to section 4.2.3.6 of RFC 1122: *Requirements for Internet Hosts -- Communication Layers*. If a connection is dropped as the result of "keep-alives" the error code WSAENETRESET is returned to any calls in progress on the socket, and any subsequent calls will fail with WSAENOTCONN.

SO_LINGER

SO_LINGER controls the action taken when unsent data is queued on a socket and a **WSPCloseSocket()** is performed. See **WSPCloseSocket()** for a description of the way in which the SO_LINGER settings affect the semantics of **WSPCloseSocket()**. The application sets the desired behavior by creating a *struct linger* (pointed to by the *optval* argument) with the following elements:

```
struct linger {
      int   l_onoff;
      int   l_linger;
}
```

To enable SO_LINGER, the application should set *l_onoff* to a non-zero value, set *l_linger* to 0, and call **WSPSetSockOpt()**. To enable SO_DONTLINGER (i.e. disable SO_LINGER) *l_onoff* should be set to zero and **WSPSetSockOpt()** should be called.

SO_REUSEADDR

By default, a socket may not be bound (see **WSPBind()**) to a local address which is already in use. On occasions, however, it may be desirable to "re-use" an address in this way. Since every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the Winsock provider that a **WSPBind()** on a socket should not be disallowed because the desired address is already in use by another socket, the application should set the SO_REUSEADDR socket option for the socket before issuing the **WSPBind()**. Note that the option is interpreted only at the time of the **WSPBind()**: it is therefore unnecessary (but harmless) to set the option on a socket which is not to be bound to an existing address, and setting or resetting the option after the **WSPBind()** has no effect on this or any other socket.

PVD_CONFIG

This object stores the configuration information for the service provider associated with socket *s*. The exact format of this data structure is service provider specific.

TCP_NODELAY

The TCP_NODELAY option is specific to TCP/IP service providers. It is used to disable the Nagle algorithm. The Nagle algorithm is used to reduce the number of small packets sent by a host by buffering unacknowledged send data until a full-size packet can be sent. However, for some applications this algorithm can impede performance, and TCP_NODELAY may be used to turn it off. Application writers should not set TCP_NODELAY unless the impact of doing so is well-understood and desired, since setting TCP_NODELAY can have a significant negative impact of network performance.

Return Value

If no error occurs, **WSPSetSockOpt()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	*optval* is not in a valid part of the process address space.
WSAEINPROGRESS	The function is invoked when a callback is in progress.

WSAEINVAL	*level* is not valid, or the information in *optval* is not valid.
WSAENETRESET	Connection has timed out when SO_KEEPALIVE is set.
WSAENOPROTOOPT	The option is unknown or unsupported for the specified provider.
WSAENOTCONN	Connection has been reset when SO_KEEPALIVE is set.
WSAENOTSOCK	The descriptor is not a socket.

See Also **WSPBind()**, **WSPGetSockOpt()**, **WSPIoctlSocket()**, **WSPsocket()**, **WSPEventSelect()**.

3.1.10 WSPShutdown()

Description Disable sends and/or receives on a socket.

#include <ws2spi.h>

int WSPAPI WSPShutdown (SOCKET *s***, int** *how*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying a socket.
how	A flag that describes what types of operation will no longer be allowed.
lpErrno	A pointer to the error code.

Remarks **WSPShutdown()** is used on all types of sockets to disable reception, transmission, or both.

If *how* is SD_RECEIVE, subsequent receives on the socket will be disallowed. This has no effect on the lower protocol layers. For TCP, the TCP window is not changed and incoming data will be accepted (but not acknowledged) until the window is exhausted. For UDP, incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated.

If *how* is SD_SEND, subsequent sends on the socket are disallowed. For TCP sockets, a FIN will be sent.

Setting *how* to SD_BOTH disables both sends and receives as described above.

Note that **WSPShutdown()** does not close the socket, and resources attached to the socket will not be freed until **WSPCloseSocket()** is invoked.

Comments **WSPShutdown()** does not block regardless of the SO_LINGER setting on the socket. An application should not rely on being able to re-use a socket after it has been shut down. In particular, a Winsock service provider is not required to support the use of **WSPConnect()** on such a socket.

Return Value If no error occurs, **WSPShutdown()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno***.**

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	*how* is not valid, or is not consistent with the socket type, e.g., SD_SEND is used with a UNI_RECV socket type.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.

See Also **WSPConnect**(), **WSPsocket**().

3.1.11 WSPAccept()

Description Conditionally accept a connection based on the return value of a condition function, and optionally create and/or join a socket group.

#include <ws2spi.h>

SOCKET WSPAPI WSPAccept (SOCKET *s*, **struct sockaddr** FAR * *addr,* **int** FAR * *addrlen*, **LPCONDITIONPROC** *lpfnCondition*, **DWORD** *dwCallbackData*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying a socket which is listening for connections after a **WSPListen**().
addr	An optional pointer to a buffer which receives the address of the connecting entity, as known to the communications layer. The exact format of the *addr* argument is determined by the address family established when the socket was created.
addrlen	An optional pointer to an integer which contains the length of the address *addr*.
lpfnCondition	The address of the optional, the Winsock DLL-supplied condition function which will make an accept/reject decision based on the caller information passed in as parameters, and optionally create and/or join a socket group by assigning appropriate value to the result parameter *g* of this function.
dwCallbackData	The callback data passed back to the Winsock DLL in the condition function. This object is not interpreted by the service provider.
lpErrno	A pointer to the error code.

Remarks This routine extracts the first connection on the queue of pending connections on *s*, and checks it against the condition function, provided the condition function is specified (i.e., not NULL). If *lpfnCondition* is set to NULL, a connection is accepted unconditionally, and no socket group is created or joined. If the condition function returns CF_ACCEPT, this routine creates a new socket with the same properties as *s* and returns a handle to the new socket, and then optionally creates and/or joins a socket group based on the value of the result parameter *g* in the condition function. If the condition function returns CF_REJECT, this routine rejects this connection request. The condition function runs in the same thread as this routine does, and should return as soon as possible. If the decision cannot be made immediately, the condition function will return CF_DEFER to indicate that no decision has been made, and no action about this connection request should be taken by the service provider. When the Winsock DLL is ready to take action on the connection request, it may invoke **WSPAccept()** again and return either CF_ACCEPT or CF_REJECT as a return value from the condition function.

If no pending connections are present on the queue, **WSPAccept()** returns an error as described below. The accepted socket may not be used to accept more connections. The original socket remains open.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the address family in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-oriented socket types such as SOCK_STREAM. If *addr* and/or *addrlen* are equal to NULL, then no information about the remote address of the accepted socket is returned. Otherwise, these two parameters will be filled in regardless of whether the condition function is specified or what it returns.

The prototype of the condition function is as follows:

int WSACALLBACK ConditionFunc(LPWSABUF *lpCallerId*, **LPWSABUF** *lpCallerData*, **LPWSABUF** *lpCalleeId*, **LPWSABUF** *lpCalleeData*, **GROUP** FAR * *g*, **DWORD** *dwCallbackData* **);**

LPWSABUF is defined in ws2spi.h as follows:

```
typedef struct _WSABUF {
      int         len;  // the length of the buffer
      char FAR *  buf;  // the pointer to the buffer
} WSABUF, FAR * LPWSABUF;
```

ConditionFunc is a placeholder for a Winsock DLL supplied function. It is invoked in the same thread as **WSPAccept()**, thus no other Winsock functions can be called. The *lpCallerId* and *lpCallerData* are value parameters which contain the address of the connecting entity and any user data that was sent along with the connection request, respectively. The *lpCalleeId* is a value parameter which contains the local address of the connected entity. The *lpCalleeData* is a result parameter in which the condition function fills in the user data passed back to the connecting entity. *lpCalleeData->len* initially contains the length of the buffer allocated by the service provider, and 0 means user data back to the caller is not supported. If *lpCalleeData->len* is set to 0, no user data will be passed back. The exact format of the address and user data is specific to the address family to which the socket belongs.

The result parameter *g* is assigned within the condition function to indicate the following actions:

if *g* is an existing socket group id, add *s* to this group, provided all the requirements set by this group are met; or
if *g* = SG_UNCONSTRAINED_GROUP, create an unconstrained socket group and have *s* as the first member; or
if *g* = SG_CONSTRAINED_GROUP, create a constrained socket group and have *s* as the first member; or
if *g* = NULL, no operation is performed.

For unconstrained groups, any set of sockets may be grouped together as long as they are supported by a single Winsock service provider and are connection-oriented. A constrained socket group requires that connections on all grouped sockets be to the same host. For newly created socket groups, the new group id can be retrieved by *lpGroupAction* or using **WSPGetSockOpt()** with option SO_GROUP_ID, if this operation completes successfully.

Return Value

If no error occurs, **WSPAccept**() returns a value of type SOCKET which is a descriptor for the accepted socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code is available in *lpErrno*.

The integer referred to by *addrlen* initially contains the amount of space pointed to by *addr*. On return it will contain the actual length in bytes of the address returned.

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAECONNREFUSED	The connection request was forcefully rejected as indicated in the return value of the condition function (CF_REJECT).
WSAEFAULT	The *addrlen* argument is too small or the *lpfnCondition* is not part of the user address space.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAEINVAL	**WSPListen()** was not invoked prior to **WSPAccept()**, parameter *g* specified in the condition function is not a valid value, the return value of the condition function is not a valid one, or any case where the specified socket is in an invalid state.
WSAEMFILE	The queue is non-empty upon entry to **WSPAccept()** and there are no socket descriptors available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The referenced socket is not a type that supports connection-oriented service.
WSATRY_AGAIN	The acceptance of the connection request was deferred as indicated in the return value of the condition function (CF_DEFER).
WSAEWOULDBLOCK	No connections are present to be accepted, or the connection request that was deferred has timed out or been withdrawn.

See Also **WSPAccept(), WSPBind(), WSPConnect(), WSPGetSockOpt(), WSPListen(), WSPSelect(), WSPsocket(), WSPEventSelect().**

3.1.12 WSPAsyncSelect32()

Description Request event notification for a socket.

#include <ws2spi.h>

int WSPAPI WSPAsyncSelect32 (SOCKET *s*, **HWND** *hWnd*,
unsigned int *wMsg*, **long** *lEvent*, **int FAR *** *lpErrno* **);**

s	A descriptor identifying the socket for which event notification is required.
hWnd	A handle identifying the window which should receive a message when a network event occurs.
wMsg	The message to be received when a network event occurs.
lEvent	A bitmask which specifies a combination of network events in which the application is interested.
lpErrno	A pointer to the error code.

Remarks This function is only applicable to the 32-bit SPI. The 16 bit Winsock2.DLL uses **WSPCallbackSelect()** to implement the API function **WSAAsyncSelect().**

This function is used to request that the service provider should send a message to the window *hWnd* whenever it detects any of the network events specified by the *lEvent* parameter. The message which should be sent is specified by the *wMsg* parameter. The socket for which notification is required is identified by *s*.

This function automatically sets socket *s* to non-blocking mode, regardless of the value of *lEvent*. See **WSPIoctlSocket()** about how to set the socket back to blocking mode.

The *lEvent* parameter is constructed by or'ing any of the values specified in the following list.

Value	Meaning
FD_READ	Want to receive notification of readiness for reading
FD_WRITE	Want to receive notification of readiness for writing
FD_OOB	Want to receive notification of the arrival of out-of-band data
FD_ACCEPT	Want to receive notification of incoming connections
FD_CONNECT	Want to receive notification of completed connection
FD_CLOSE	Want to receive notification of socket closure
FD_QOS	Want to receive notification of socket Quality of Service (QOS) changes
FD_GROUP_QOS	Want to receive notification of socket group Quality of Service (QOS) changes

Issuing a **WSPAsyncSelect32()** for a socket cancels any previous **WSPAsyncSelect32()** for the same socket. For example, to receive notification for both reading and writing, the

Winsock DLL will call **WSPAsyncSelect32()** with both FD_READ and FD_WRITE, as follows:

```
rc = WSPAsyncSelect32(s, hWnd, wMsg, FD_READ|FD_WRITE,
            &error);
```

It is not possible to specify different messages for different events. The following code will not work; the second call will cancel the effects of the first, and only FD_WRITE events will be reported with message wMsg2:

```
rc = WSPAsyncSelect32(s, hWnd, wMsg1, FD_READ, &error);
rc = WSPAsyncSelect32(s, hWnd, wMsg2, FD_WRITE, &error);
```

To cancel all notification – i.e., to indicate that the service provider should send no further messages related to network events on the socket – *lEvent* should be set to zero.

```
rc = WSPAsyncSelect32(s, hWnd, 0, 0, &error);
```

Although in this instance **WSPAsyncSelect32()** immediately disables event message posting for the socket, it is possible that messages may be waiting in the application's message queue. The application must therefore be prepared to receive network event messages even after cancellation. Closing a socket with **WSPCloseSocket()** also cancels **WSPAsyncSelect32()** message sending, but the same caveat about messages in the queue prior to the **WSPCloseSocket()** still applies.

Since a **WSPAccept()'**ed socket has the same properties as the listening socket used to accept it, any **WSPAsyncSelect32()** events set for the listening socket apply to the accepted socket. For example, if a listening socket has **WSPAsyncSelect32()** events FD_ACCEPT, FD_READ, and FD_WRITE, then any socket accepted on that listening socket will also have FD_ACCEPT, FD_READ, and FD_WRITE events with the same *wMsg* value used for messages. If a different *wMsg* or events are desired, the application should call **WSPAsyncSelect32()**, passing the accepted socket and the desired new information.[1]

When one of the nominated network events occurs on the specified socket *s*, the application's window *hWnd* receives message *wMsg*. The *wParam* argument identifies the socket on which a network event has occurred. The low word of *lParam* specifies the network event that has occurred. The high word of *lParam* contains any error code. The error code be any error as defined in **Winsock2.h**.

The error and event codes may be extracted from the *lParam* using the macros WSAGETSELECTERROR and WSAGETSELECTEVENT, defined in **Winsock2.h** as:

```
#define WSAGETSELECTERROR(lParam)             HIWORD(lParam)
#define WSAGETSELECTEVENT(lParam)             LOWORD(lParam)
```

The use of these macros will maximize the portability of the source code for the application.

---

[1]Note that there is a timing window between the **accept()** call and the call to **WSAAsyncSelect()** to change the events or *wMsg*. An application which desires a different *wMsg* for the listening and **accept()**'ed sockets should ask for only FD_ACCEPT events on the listening socket, then set appropriate events after the **accept()**. Since FD_ACCEPT is never sent for a connected socket and FD_READ, FD_WRITE, FD_OOB, and FD_CLOSE are never sent for listening sockets, this will not impose difficulties.

The possible network event codes which may be returned are as follows:

Value	Meaning
FD_READ	Socket *s* ready for reading
FD_WRITE	Socket *s* ready for writing
FD_OOB	Out-of-band data ready for reading on socket s.
FD_ACCEPT	Socket *s* ready for accepting a new incoming connection
FD_CONNECT	Connection initiated on socket *s* completed
FD_CLOSE	Connection identified by socket *s* has been closed
FD_QOS	Quality of Service associated with socket *s* has changed.
FD_GROUP_QOS	Quality of Service associated with the socket group to which *s* belongs has changed.

Return Value The return value is 0 if the application's declaration of interest in the network event set was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

Comments Although **WSPAsyncSelect32()** can be called with interest in multiple events, the application window will receive a single message for each network event.

As in the case of the **WSPSelect()** function, **WSPAsyncSelect32()** will frequently be used to determine when a data transfer operation (**WSPSend()** or **WSPRecv()**) can be issued with the expectation of immediate success. Nevertheless, a robust application must be prepared for the possibility that it may receive a message and issue a Winsock2 call which returns WSAEWOULDBLOCK immediately. For example, the following sequence of events is possible:

(i)	data arrives on socket **s**; Winsock2 posts **WSPAsyncSelect32()** message
(ii)	application processes some other message
(iii)	while processing, application issues an **WSPIoctlSocket(s, FIONREAD...)** and notices that there is data ready to be read
(iv)	application issues a **WSPRecv(s,...)** to read the data
(v)	application loops to process next message, eventually reaching the **WSPAsyncSelect32()** message indicating that data is ready to read
(vi)	application issues **WSPRecv(s,...)**, which fails with the error WSAEWOULDBLOCK.

Other sequences are possible.

The Winsock DLL will not continually flood an application with messages for a particular network event. Having successfully posted notification of a particular event to an application window, no further message(s) for that network event will be posted to the application window until the application makes the function call which implicitly reenables notification of that network event.

Event	Re-enabling function
FD_READ	**WSPRecv()** or **WSPRecvFrom()**
FD_WRITE	**WSPSend()** or **WSPSendTo()**
FD_OOB	**WSPRecv()**
FD_ACCEPT	**WSPAccept()** unless the error code returned is WSATRY_AGAIN indicating that the condition function returned CF_DEFER

FD_CONNECT	NONE
FD_CLOSE	NONE
FD_QOS	**WSPGetSockOpt()** with option SO_FLOWSPEC
FD_GROUP_QOS	**WSPGetSockOpt()** with option SO_GROUP_FLOWSPEC

Any call to the reenabling routine, even one which fails, results in reenabling of message posting for the relevant event.

For FD_READ, FD_OOB, FD_ACCEPT, FD_QOS and FD_GROUP_QOS events, message posting is "level-triggered." This means that if the reenabling routine is called and the relevant event is still valid after the call, a **WSPAsyncSelect32()** message is posted to the application. This allows an application to be event-driven and not be concerned with the amount of data that arrives at any one time. Consider the following sequence:

(i)	network transport stack receives 100 bytes of data on socket **s** and causes Winsock2 to post an FD_READ message.
(ii)	The application issues **WSPRecv(s, buffptr, 50, 0)** to read 50 bytes.
(iii)	another FD_READ message is posted since there is still data to be read.

With these semantics, an application need not read all available data in response to an FD_READ message--a single **WSPRecv()** in response to each FD_READ message is appropriate. If an application issues multiple **WSPRecv()** calls in response to a single FD_READ, it may receive multiple FD_READ messages. Such an application may wish to disable FD_READ messages before starting the **WSPRecv()** calls by calling **WSPAsyncSelect32()** with the FD_READ event not set.

If an event has already happened when the application calls **WSPAsyncSelect32()** or when the reenabling function is called, then a message is posted as appropriate. All the events have persistence beyond the occurrence of their respective events. For example, consider the following sequence: 1) an application calls **WSPListen()**, 2) a connect request is received but not yet accepted, 3) the application calls **WSPAsyncSelect32()** specifying that it wants to receive FD_ACCEPT messages for the socket. Due to the persistence of events, Winsock2 posts an FD_ACCEPT message immediately.

The FD_WRITE event is handled slightly differently. An FD_WRITE message is posted when a socket is first connected with **WSPConnect()** or accepted with **WSPAccept()**, and then after a **WSPSend()** or **WSPSendTo()** fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, an application can assume that sends are possible starting from the first FD_WRITE message and lasting until a send returns WSAEWOULDBLOCK. After such a failure the application will be notified that sends are again possible with an FD_WRITE message.

The FD_OOB event is used only when a socket is configured to receive out-of-band data separately. If the socket is configured to receive out-of-band data in-line, the out-of-band (expedited) data is treated as normal data and the application should register an interest in, and will receive, FD_READ events, not FD_OOB events. An application may set or inspect the way in which out-of-band data is to be handled by using **WSPSetSockOpt()** or **WSPGetSockOpt()** for the SO_OOBINLINE option.

The error code in an FD_CLOSE message indicates whether the socket close was graceful or abortive. If the error code is 0, then the close was graceful; if the error code is WSAECONNRESET, then the socket's virtual circuit was reset. This only applies to connection-oriented sockets such as SOCK_STREAM.

The FD_CLOSE message is posted when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the FD_CLOSE is posted when the connection goes into the FIN WAIT or CLOSE WAIT states. This results from the remote end performing a **WSPShutdown()** on the send side or a **WSPCloseSocket().**

Please note your application will receive ONLY an FD_CLOSE message to indicate closure of a virtual circuit, and only when all the received data has been read if this is a graceful close. It will NOT receive an FD_READ message to indicate this condition.

The FD_QOS or FD_GROUP_QOS message is posted when any field in the flow spec associated with socket *s* or the socket group that *s* belongs to has changed, respectively. Applications might use **WSPGetSockOpt()** with option SO_FLOWSPEC or SO_GROUP_FLOWSPEC to get the current QOS for socket *s* or for the socket group *s* belongs to, respectively.

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid, or the specified socket is in an invalid state.
WSAEINPROGRESS	A blocking Winsock2 call is in progress, or the service provider is still processing a callback function (see section ???).
WSAENOTSOCK	The descriptor is not a socket.

Additional error codes may be set when an application window receives a message. This error code is extracted from the *lParam* in the reply message using the WSAGETSELECTERROR macro. Possible error codes for each network event are:

Event: FD_CONNECT

Error Code	Meaning
WSAEADDRINUSE	The specified address is already in use.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAENETUNREACH	The network can't be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection

Event: FD_CLOSE

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.

WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was aborted due to timeout or other failure.

Event: FD_READ
Event: FD_WRITE
Event: FD_OOB
Event: FD_ACCEPT
Event: FD_QOS
Event: FD_GROUP_QOS

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.

See Also **WSPSelect()**

3.1.13 WSPCallbackSelect16()

Description Request event notification for a socket via the specified callback function.

#include <ws2spi.h>

int WSPAPI WSPCallbackSelect16 (**SOCKET** *s*, **LPSELECTPROC** *lpfnCallback*, **DWORD** *dwCallbackData*, **long** *lEvent*, **int** FAR * *lpErrno*);

s	A descriptor identifying the socket for which event notification is required.
lpfnCallback	The procedure instance address of the callback function to be invoked by Winsock service providers whenever a registered network event happens.
dwCallbackData	The callback data which is passed back to the application as a parameter to the callback function. This object is not interpreted by the Winsock service provider.
lEvent	A bitmask which specifies a combination of network events in which the Winsock DLL is interested.
lpErrno	A pointer to the error code.

Remarks This function is only applicable to the 16-bit SPI. The 16 bit Winsock2.DLL uses this function to implement the **select()**, **WSAAsyncSelect()**, and **WSPEventSelect()** API functions.

This function enables the function-based callback mechanism for the specified socket. The callback function will be invoked whenever the service provider detects any of the network events specified by the *lEvent* parameter. The socket for which notification is required is identified by *s*. The value of *dwCallbackData* will be passed back to the caller as a parameter to the callback function.

The prototype of the callback function is as follows:

> **VOID WSACALLBACK CallbackFunc(SOCKET** *s*, **long** *lEvent*, **int** *ErrorCode,* **DWORD** *dwCallbackData* **);**

CallbackFunc is a placeholder for a Winsock DLL supplied function. The callback function is written in such a way that it can be called by the provider from within interrupt context[2].**WSPSend(), WSPSendTo(), WSPRecv(), WSPRecvFrom()**

The *lEvent* parameter is constructed by or'ing any of the values specified in the following list.

---

[2] Because the callback is accessed at interrupt time, it must reside in a DLL, and its code segment must be specified as FIXED in the module-definition file for the DLL. Any data that the callback accesses must be in a FIXED data segment as well. The callback may not make any system calls except for PostMessage, timeGetSystemTime, timeGetTime, timeSetEvent, timeKillEvent, midiOutShortMsg, midiOutLongMsg, and OutputDebugStr.

Value	Meaning
FD_READ	Want to receive notification of readiness for reading
FD_WRITE	Want to receive notification of readiness for writing
FD_OOB	Want to receive notification of the arrival of out-of-band data
FD_ACCEPT	Want to receive notification of incoming connections
FD_CONNECT	Want to receive notification of completed connection
FD_CLOSE	Want to receive notification of socket closure
FD_QOS	Want to receive notification of Quality of Service (QOS) changes
FD_GROUP_QOS	Want to receive notification of socket group Quality of Service (QOS) changes

Issuing a **WSPCallbackSelect16()** for a socket cancels any previous **WSPCallbackSelect16()** for the same socket. For example, to receive notification for both reading and writing, the Winsock DLL must call **WSPCallbackSelect16()** with both FD_READ and FD_WRITE, as follows:

```
rc = WSPCallbackSelect16(s, lpfnCallback, dwCallbackData,
            FD_READ|FD_WRITE, lpErrno);
```

It is not possible to specify different callback data for different events. The following code will not work; the second call will cancel the effects of the first, and only FD_WRITE events will be reported with dwCallbackData2:

```
rc = WSPCallbackSelect16(s, lpfnCallback, dwCallbackData1,
            FD_READ, lpErrno);
rc = WSPCallbackSelect16(s, lpfnCallback, dwCallbackData2,
            FD_WRITE, lpErrno);
```

To cancel all notification – i.e., to indicate that the service provider should no long invoke the callback function related to network events on the socket – *lEvent* should be set to zero. In this case, *lpfnCallback* and *dwCallbackData* are ignored.

```
rc = WSPCallbackSelect16(s, lpfnCallback, dwCallbackData, 0,
            lpErrno);
```

Closing a socket with **WSPCloseSocket()** also cancels **WSPCallbackSelect16()** mechanism on the socket.

A **WSPAccept()**'ed socket has the same properties as the listening socket used to accept it except that no network events and callback function are associated with it. Winsock2.DLL will subsequently invoke **WSPCallbackSelect16()** to register interest in the appropriate set of events for the **WSPAccept**()'ed socket[3].

When one of the nominated network events occurs on the specified socket *s*, the callback function will be invoked. The *s* argument identifies the socket on which a network event has occurred. The *lEvent* argument specifies the event that has occurred. The *ErrorCode* argument contains any error code. The error code can be any error as defined in **ws2spi.h**.

The possible network event codes which may be returned are as follows:

[3] Note that there is a potential race condition here: WINSOCK 2..DLL may get FD_ACCEPT callback in interrupt time before it returns from **WSPAcceptEx()**. Similar situations may happen to any of the reenabling functions if they are not invoked from within callback context.

Value	Meaning
FD_READ	Socket *s* ready for reading
FD_WRITE	Socket *s* ready for writing
FD_OOB	Out-of-band data ready for reading on socket s.
FD_ACCEPT	Socket *s* ready for accepting a new incoming connection
FD_CONNECT	Connection initiated on socket *s* completed
FD_CLOSE	Connection identified by socket *s* has been closed
FD_QOS	Quality of Service associated with socket *s* or the socket group to which *s* belongs has been changed.
FD_GROUP_QOS	Quality of Service associated with the socket group to which *s* belongs has changed.

Return Value The return value is 0 if the Winsock's declaration of interest in the network event set was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number is available in *lpErrno*.

Comments Although **WSPCallbackSelect16()** can be called with interest in multiple events, the Winsock DLL will get a callback for each network event.

As in the case of **WSPSelect()**, **WSPCallbackSelect16()** is frequently used to determine when a data transfer operation (**WSPSend()** or **WSPRecv()**) can be issued with the expectation of immediate success.

The Winsock service provider will not continually flood the Winsock DLL with callbacks for a particular network event on a given socket. Having successfully invoked a callback indicating a particular event to the Winsock DLL, no further callback(s) for that network event on the socket will be invoked to the Winsock DLL until the Winsock DLL makes the function call which implicitly reenables notification of that network event, and returns from the callback function[4].

Event	Re-enabling function
FD_READ	**WSPRecv()** or **WSPRecvFrom()**
FD_WRITE	**WSPSend()** or **WSPSendTo()**
FD_OOB	**WSPRecv()**
FD_ACCEPT	**WSPAccept()** unless the error code returned is WSATRY_AGAIN indicating that the condition function returned CF_DEFER
FD_CONNECT	NONE
FD_CLOSE	NONE
FD_QOS	**WSPGetSockOpt()** with option SO_FLOWSPEC
FD_GROUP_QOS	**WSPGetSockOpt()** with option SO_GROUP_FLOWSPEC

Any call to the reenabling routine, even one which fails (except in the case of FD_ACCEPT when **WSPAccept()** returns with error code WSATRY_AGAIN), results in reenabling of callback invocation for the relevant event.

[4] However, due to the independence feature of sockets, WINSOCK2.DLL may get a callback for one socket before the callback for another socket has returned. WINSOCK2.DLL may need to coordinate access to any common memory area shared amongst these callback driven sockets.

For FD_READ, FD_OOB, FD_ACCEPT, FD_QOS and FD_GROUP_QOS events, callback invocation is "level-triggered." This means that if the reenabling routine is called and the relevant event is still valid after the call, an additional callback is invoked after the previous callback returns. This allows applications using Winsock2.DLL to be event-driven and not be concerned with the amount of data that arrives at any one time. Consider the following sequence:

(i) The service provider receives 100 bytes of data on socket **s** and invokes an FD_READ callback.

(ii) The Winsock2.DLL issues **WSPRecv(s, buffptr, 50, 0, lpErrno)** to read 50 bytes.

(iii) The service provider invokes another FD_READ callback since there is still data to be read.

With these semantics, the Winsock DLL need not read all available data in response to an FD_READ callback--a single **WSPRecv()** in response to each FD_READ callback is appropriate. If the Winsock DLL issues multiple **WSPRecv()** calls in response to a single FD_READ, it may receive multiple FD_READ callbacks.

If an event has already happened when the Winsock DLL calls **WSPCallbackSelect16()**[5] or when the reenabling function is called, then a callback is invoked as appropriate. All the events have persistence beyond the occurrence of their respective events. For example, if the Winsock DLL calls **WSPListen()**, a connect attempt is made, then the Winsock DLL calls **WSPCallbackSelect16()** specifying that it wants to receive FD_ACCEPT callbacks for the socket, the service provider invokes an FD_ACCEPT callback immediately.

The FD_WRITE event is handled slightly differently. An FD_WRITE callback is invoked when a socket is first connected with **WSPConnect()** or accepted with **WSPAccept**, and then after a **WSPSend()** or **WSPSendTo()** fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, the Winsock DLL can assume that sends are possible starting from the first FD_WRITE callback and lasting until a send returns WSAEWOULDBLOCK. After such a failure the Winsock DLL will be notified that sends are again possible with an FD_WRITE callback.

The FD_OOB event is used only when a socket is configured to receive out-of-band data separately. If the socket is configured to receive out-of-band data in-line, the out-of-band (expedited) data is treated as normal data and the Winsock DLL should register an interest in, and will receive, FD_READ events, not FD_OOB events. The Winsock DLL may set or inspect the way in which out-of-band data is to be handled by using **WSPSetSockOpt()** or **WSPGetSockOpt()** for the SO_OOBINLINE option.

The error code in an FD_CLOSE callback indicates whether the socket close was graceful or abortive. If the error code is 0, then the close was graceful; if the error code is WSAECONNRESET, then the socket's virtual circuit was reset. This only applies to connection-oriented sockets such as SOCK_STREAM.

The FD_CLOSE callback is invoked when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the FD_CLOSE callback is invoked when the connection goes into the FIN WAIT or CLOSE WAIT

[5] Note that there is a potential race condition here: WINSOCK2.DLL may get a callback in interrupt time even before returning from **WSPCallbackSelect()**, in which case WINSOCK2.DLL should proceed with the callback and assume that **WSPCallbackSelect()** will return 0.

states. This results from the remote end performing a **WSPShutdown()** on the send side or a **WSPCloseSocket().**

Please note the Winsock DLL should receive ONLY an FD_CLOSE callback to indicate closure of a virtual circuit. It should NOT receive an FD_READ callback to indicate this condition.

The FD_QOS or FD_GROUP_QOS callback is invoked when any field in the flow spec associated with socket *s* or the socket group to which *s* belongs has changed, respectively.
WSPGetSockOpt

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid, or the specified socket is in an invalid state.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAEFAULT	The *lpfnCallback* is not part of the user address space.
WSAENOTSOCK	The descriptor is not a socket.

Additional error codes may be set when the Winsock DLL receives a callback. Possible error codes for each network event are:

Event: FD_CONNECT

Error Code	Meaning
WSAEADDRINUSE	The specified address is already in use.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAENETUNREACH	The network can't be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection

Event: FD_CLOSE

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was aborted due to timeout or other failure.

Event: FD_READ
Event: FD_WRITE
Event: FD_OOB
Event: FD_ACCEPT
Event: FD_QOS
Event: FD_GROUP_QOS

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.

Notes For Winsock Service Providers

When a socket is closed, the Winsock service provider should cancel any pending callbacks to the Winsock DLL.

3.1.14 WSPCancelBlockingCall32()

Description Cancel a blocking call which is currently in progress.

#include <ws2spi.h>

int WSPAPI WSPCancelBlockingCall32 (**int** FAR * *lpErrno*);

lpErrno A pointer to the error code.

Remarks This function is only applicable to the 32-bit SPI.

This function cancels any outstanding blocking operation for the current thread. It is normally used in two situations:

1. An application is processing a message which has been received while a blocking call is in progress. In this case, **WSPIsBlocking32()** will be TRUE.
2. A blocking call is in progress, and Winsock has called back to the applications "blocking hook" function (as established by **WSPSetBlockingHook32()**).

In each case, the original blocking call will terminate as soon as possible with the error WSAEINTR. (In (1), the termination will not take place until Windows message scheduling has caused control to revert to the blocking routine in Windows Sockets. In (2), the blocking call will be terminated as soon as the blocking hook function completes.)

In the case of a blocking **WSPConnect()** operation, the Windows Sockets implementation will terminate the blocking call as soon as possible, but it may not be possible for the socket resources to be released until the connection has completed (and then been reset) or timed out. This is likely to be noticeable only if the application immediately tries to open a new socket (if no sockets are available), or to **WSPConnect()** to the same peer.

Cancelling an **WSPAccept()** or a **WSPSelect()** call does not adversely impact the sockets passed to these calls. Only the particular call fails; any operation that was legal before the cancel is legal after the cancel, and the state of the socket is not affected in any way.

Cancelling any operation other than **WSPAccept()** and **WSPSelect()** can leave the socket in an indeterminate state. If an application cancels a blocking operation on a socket, the only operation that the application can depend on being able to perform on the socket is a call to **WSPCloseSocket()**, although other operations may work on some service provider implementations. If an application desires maximum portability, it must be careful not to depend on performing operations after a cancel. An application may reset the connection by setting the timeout on SO_LINGER to 0.

If a cancel operation compromised the integrity of a SOCK_STREAM's data stream in any way, the Windows Sockets implementation must reset the connection and fail all future operations other than **WSPCloseSocket()** with WSAECONNABORTED.

Return Value The return value is 0 if the blocking operation has been successfully cancelled. Otherwise the value SOCKET_ERROR is returned, and a specific error number is available in *lpErrno*.

Comments Note that it is possible that the network operation completes before the **WSPCancelBlockingCall32()** is processed, for example if data is received into the user buffer at interrupt time while the application is in a blocking hook. In this case, the blocking operation will return successfully as if **WSPCancelBlockingCall32()** had never been called. Note that the **WSPCancelBlockingCall32()** still succeeds in this case; the only way to know with certainty that an operation was actually canceled is to check for a return code of WSAEINTR from the blocking call.

Error Codes

WSANOTINITIALISED	A successful **WSPStartup()** must occur before using this SPI.
WSAENETDOWN	The Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINVAL	Indicates that there is no outstanding blocking call.

See Also **WSPIsBlocking32(), WSPSetBlockingHook32(), WSPUnhookBlockingHook32()**

3.1.15 WSPCleanup()

Description Terminate use of the Winsock service provider.

#include <ws2spi.h>

int WSPAPI WSPCleanup (LPCLEANUPPROC *lpfnCallback*, **DWORD** *dwCallbackData*, **int** FAR * *lpErrno* **);**

lpfnCallback The address of the callback function to be invoked by the service provider when the cleanup operation completed.

dwCallbackData The callback data passed back to the Winsock DLL along with the callback. This object is not interpreted by the Winsock service provider.

lpErrno A pointer to the error code.

Remarks The Winsock DLL is required to perform a (successful) **WSPStartup()** call before it can use Winsock service providers. When it has completed the use of Winsock service providers, the Winsock DLL must call **WSPCleanup()** to deregister itself from a Winsock service provider and allow the service provider to free any resources allocated on behalf of the Winsock DLL. Any open connection-oriented sockets that are connected when **WSPCleanup()** is called are reset; sockets which have been closed with **WSPCloseSocket()** but which still have pending data to be sent are not affected--the pending data is still sent. When the cleanup operation is finished, the specified callback function will be invoked. The value of *dwCallbackData* will be passed back to the Winsock DLL along with the callback.

The prototype of the callback function is as follows:

VOID WSACALLBACK CallbackFunc(int *ErrorCode*, **DWORD** *dwCallbackData* **);**

CallbackFunc is a placeholder for a Winsock DLL-supplied function. The callback function is only called when the client thread that invoked this SPI is blocked in an alertable wait. The callback data is passed back to the Winsock DLL along with the callback. This object is not interpreted by the Winsock service provider.

Return Value The return value is 0 if the operation has been successfully initiated. Otherwise the value SOCKET_ERROR is returned, and a specific error number is available in *lpErrno*.

Notes For Winsock Service Providers The Winsock DLL will make one and only one **WSPCleanup()** call to indicate deregistration from a Winsock service provider. This function can thus, for example, be utilized to free up allocated resources. {Yeah, but since each protocol is represented as a separate service provider, it's possible that the service provider's DLL which implements multiple protocols (e.g. TCP and UDP) could be called with WSPCleanup() multiple times} {We have basically two choices here. We could either a) make separate WSPStartup() and WSPCleanup() calls for each protocol/address_family/socket_type triple supported by a given service provider DLL, or b) provide a single WSPStartup() call when a service provider is first loaded, and a single WSPCleanup() when its time to

close shop. I tend to favor b), as it places the burden of responsibility on the Winsock DLL, where it's easier to control & get right (once).}

In a multithreaded environment, **WSPCleanup()** terminates Winsock operations for all threads.

A Winsock service provider must ensure that **WSPCleanup()** leaves things in a state in which the Winsock DLL can invoke **WSPStartup()** to re-establish Winsock usage.

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *lpfnCallback* is not part of the user address space.

See Also **WSPStartup()**

3.1.16 WSPConnect()

Description Establish a connection to a peer, create and/or join a socket group, and specify needed quality of service based on the supplied flow spec.

#include <ws2spi.h>

int WSPAPI WSPConnect (SOCKET *s*, **const struct sockaddr** FAR * *name*, **int** *namelen*, **LPWSABUF** *lpCallerData*, **LPWSABUF** *lpCalleeData*, **GROUP** *g*, **LPQOS** *lpSFlowspec*, **LPQOS** *lpGFlowspec*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying an unconnected socket.
name	The name of the peer to which the socket is to be connected.
namelen	The length of the *name*.
lpCallerData	A pointer to the user data that is to be transferred to the peer during connection establishment.
lpCalleeData	A pointer to the user data that is to be transferred back from the peer during connection establishment.
g	The identifier of the socket group.
lpSFlowspec	A pointer to the flow specs for socket *s*, one for each direction, if *s* is a DSTREAM type socket. Otherwise, this parameter is ignored.
lpGFlowspec	A pointer to the flow specs for the socket group to be created, one for each direction, if the value of parameter *g* is SG_CONSTRAINED_GROUP. Otherwise, this parameter is ignored.
lpErrno	A pointer to the error code.

Remarks This function is used to create a connection to the specified destination, and to perform a number of other ancillary operations that occur at connect time as well. For connection-oriented sockets (e.g., type SOCK_STREAM), an active connection is initiated to the foreign host using *name* (an address in the name space of the socket; for a detailed description, please see **WSPBind()**). When this call completes successfully, the socket is ready to send/receive data.

For a connectionless socket (e.g., type SOCK_UNREL_DGRAM), the operation performed by **WSPConnect()** is merely to establish a default destination address which will be used on subsequent **WSPSend**() and **WSPRecv**() calls. Exchange of user to user data and QOS specification are not possible and the corresponding parameters will be ignored.

If the socket, *s*, is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound. Note that if the address field of the *name* structure is all zeroes, **WSPConnect()** will return the error WSAEADDRNOTAVAIL.

The Winsock DLL is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specified. **LPWSABUF** and **LPQOS** are defined in ws2spi.h as follows:

```
typedef struct _WSABUF {
      int         len;  // the length of the buffer
      char FAR *  buf;  // the pointer to the buffer
} WSABUF, FAR * LPWSABUF;

typedef enum
{
      GuaranteedService,
      BestEffortService
} GUARANTEE;

typedef struct _flowparams
{
      int64       AverageBandwidth;// In Bytes/sec
      int64       PeakBandwidth;  // In Bytes/sec
      int64       BurstLength;    // In microseconds
      int64       Latency;        // In microseconds
      int64       DelayVariation; // In microseconds
      GUARANTEE   levelOfGuarantee;// Guaranteed or
                                  // Best Effort
      int32       CostOfCall;     // Reserved for future
                                  // use, must be set to 0
      int32       ProviderId;     // Provider Identifier
      int32       SizePSP;        // Length of provider
                                  // specific parameters
      UCHAR ProviderSpecificParams[1];// provider specific
                                  // parameters
} FLOWPARAMS;

typedef struct _QualityOfService
{
      FLOWPARAMS ForwardFP;       // Caller(Initiator) to callee
      FLOWPARAMS BackwardFP;      // Callee to caller
} QOS, FAR * LPQOS;
```

The *lpCallerData* is a value parameter which contains any user data that is to be sent along with the connection request. If *lpCallerData* is NULL, no user data will be passed to the peer. The *lpCalleeData* is a result parameter which will contain any user data passed back from the peer as part of the connection establishment. If *lpCalleeData->len* is 0, no user data has been passed back. The *lpCalleeData* information will be valid when the connection operation is complete, i.e., after the FD_CONNECT notification has occurred. If *lpCalleeData* is NULL, no user data will be passed back. The exact format of the user data is specific to the address family to which the socket belongs.

Parameter *g* is used to indicate the appropriate actions on socket groups:

if *g* is an existing socket group id, add *s* to this group, provided all the requirements set by this group are met; or

if *g* = SG_UNCONSTRAINED_GROUP, create an unconstrained socket group and have *s* as the first member; or

if *g* = SG_CONSTRAINED_GROUP, create a constrained socket group and have *s* as the first member; or

if *g* = NULL, no operation is performed, and is equivalent to connect().
For unconstrained group, any set of sockets may be grouped together as long as they are supported by a single Winsock service provider and are connection-oriented. A constrained socket group requires that connections on all grouped sockets be to the same host. For newly created socket groups, the new group id can be retrieved by using **WSPGetSockOpt()** with option SO_GROUP_ID, if this operation completes successfully.

lpSFlowspec specifies two blocks of memory containing the flow specs for socket *s*, one for each direction. . The forward QOS or backward QOS values will be ignored as appropriate for unidirectional sockets. . . The first part of each memory block is struct FLOWSPEC, optionally followed by any service provider specific portion. Thus, *lpSFlowspec->Flen* and *lpSFlowspec->Blen* must be larger than or equal to the size of struct FLOWSPEC. A NULL value for *lpSFlowspec* indicates no application supplied flow spec.

lpGFlowspec specifies two blocks of memory containing the flow specs for the socket group to be created, one for each direction, provided that the value of parameter *g* is SG_CONSTRAINED_GROUP. Otherwise, these values are ignored. The first part of each memory block is struct FLOWSPEC, optionally followed by any service provider specific portion. Thus, *lpGFlowspec->Flen* and *lpGFlowspec->Blen* must be larger than or equal to the size of struct FLOWSPEC. A NULL value for *lpGFlowspec* indicates no application supplied flow spec.

Comments

When connected sockets break (i.e. become closed for whatever reason), they should be discarded and recreated. It is safest to assume that when things go awry for any reason on a connected socket, the Winsock DLL must discard and recreate the needed sockets in order to return to a stable point.

Return Value

If no error occurs, **WSPConnect()** returns 0. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno***.**

Error Codes

The following errors may occur at the time of the function call, and indicate that the connect operation could not be initiated.

WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The specified address is already in use.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEDESTADDREQ	A destination address is required.
WSAEFAULT	The *namelen* argument is incorrect, the buffer length for *lpCalleeData*, *lpSFlowspec*, and *lpGFlowspec* are too small, or the buffer length for *lpCallerData* is too large.

WSAEINVAL	The parameter *g* specified in the condition function is not a valid value, or the parameter *s* is a listening socket.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more socket descriptors are available.
WSAENETUNREACH	The network can't be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEPROTONOSUPPORT	The *lpCallerData* augment is not supported by the service provider.

The following error codes may be returned by **WSPEnumNetworkEvents()** after the connect has completed.

WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The specified address is already in use.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAENETUNREACH	The network can't be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAEOPNOTSUPP	The flow specs specified in *lpSFlowspec* and *lpGFlowspec* cannot be satisfied.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection

See Also **WSPAccept(), WSPBind(), WSPGetSockName(), WSPGetSockOpt(), WSPsocket(), WSPSelect(), WSPEventSelect(), WSPEnumNetworkEvents().**

3.1.17 WSPEnumNetworkEvents()

Description Discover occurrences of network events for the indicated socket.

#include <ws2spi.h>

int WSPAPI WSPEnumNetworkEvents (SOCKET *s*, **WSAEVENT** *hEventObject*, **LPWSANETWORKEVENT** *lpNetworkEvents*, **LPINT** *lpiCount*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying the socket.
hEventObject	An optional handle identifying an associated event object to be reset.
lpNetworkEvents	An array of WSANETWORKEVENT structs, each of which records an occurred network event and the associated error code.
lpiCount	The number of elements in the array. Upon returning, this parameter indicates the actual number of elements in the array, or the minimum number of elememts needed to retrieve all the network events if the return value is WSAENOBUFS.
lpErrno	A pointer to the error code.

Remarks This function is used to discover which network events have occurred for the indicated socket since the last invocation of this function. It is intended for use in conjunction with **WSPEventSelect(),** which associates an event object with one or more network events. The socket's internal record of network events is copied to *lpNetworkEvents*, whereafter the internal network events record is cleared. If *hEventObject* is non-null, the indicated event object is also reset. The Winsock2 DLL guarantees that the operations of copying the network event record, clearing it and resetting any associated event object are atomic, such that the next occurrence of a nominated network event will cause the event object to become set.

The following error codes may be returned along with the respective network event:

Event: FD_CONNECT

Error Code	Meaning
WSAEADDRINUSE	The specified address is already in use.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAENETUNREACH	The network can't be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection

Event: FD_CLOSE

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was aborted due to timeout or other failure.

Event: FD_READ
Event: FD_WRITE
Event: FD_OOB
Event: FD_ACCEPT
Event: FD_QOS
Event: FD_GROUP_QOS

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.

Return Value

The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number is available in *lpErrno*.

Error Codes

WSANOTINITIALISED	A successful **WSPStartup()** must occur before using this SPI.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section ???).
WSAENOBUFS	The supplied buffer is too small.

See Also

WSPEventSelect()

3.1.18 WSPEventSelect()

Description Specify an event object to be associated with the supplied set of FD_XXX network events.

#include <ws2spi.h>

int WSPAPI WSPEventSelect (SOCKET *s*, **WSAEVENT** *hEventObject*, **long** *lNetworkEvents*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying the socket.
hEventObject	A handle identifying the event object to be associated with the supplied set of FD_XXX network events.
lNetworkEvents	A bitmask which specifies the combination of FD_XXX network events in which the application has interest.
lpErrno	A pointer to the error code.

Remarks This function is used to specify an event object, *hEventObject*, to be associated with the selected FD_XXX network events, *lNetworkEvents*. The socket for which an event object is specified is identified by *s*. The event object is set when any of the nominated network events occur.

WSPEventSelect() operates very similarly to **WSPAsyncSelect32()**, the difference being in the actions taken when a nominated network event occurs. Whereas **WSPAsyncSelect32()** causes an application-specified Windows message to be posted, **WSPEventSelect()** sets the associated event object and records the occurrence of this event by setting the corresponding bit in an internal network event record. An application can use **WSAWaitForMultipleEvents()** or **WSAGetOverlappedResult()** to wait or poll on the event object, and use **WSAEnumNetworkEvents()** to retrieve the contents of the internal network event record and thus determine which of the nominated network events have occurred.

This function automatically sets socket *s* to non-blocking mode, regardless of the value of *lNetworkEvents*. See **WSPIoctlSocket()** about how to set the socket back to blocking mode.

The *lNetworkEvents* parameter is constructed by or'ing any of the values specified in the following list.

Value	Meaning
FD_READ	Want to receive notification of readiness for reading
FD_WRITE	Want to receive notification of readiness for writing
FD_OOB	Want to receive notification of the arrival of out-of-band data
FD_ACCEPT	Want to receive notification of incoming connections
FD_CONNECT	Want to receive notification of completed connection
FD_CLOSE	Want to receive notification of socket closure
FD_QOS	Want to receive notification of socket Quality of Service (QOS) changes
FD_GROUP_QOS	Want to receive notification of socket group Quality of Service (QOS) changes

Issuing a **WSPEventSelect()** for a socket cancels any previous **WSPAsyncSelect32()** or **WSPEventSelect()** for the same socket and clears all bits in the internal network event record. For example, to associate an event object with both reading and writing network events, the application must call **WSPEventSelect()** with both FD_READ and FD_WRITE, as follows:

```
rc = WSPEventSelect(s, hEventObject, FD_READ|FD_WRITE);
```

It is not possible to specify different event objects for different network events. The following code will not work; the second call will cancel the effects of the first, and only FD_WRITE network event will be associated with hEventObject2:

```
rc = WSPEventSelect(s, hEventObject1, FD_READ);
rc = WSPEventSelect(s, hEventObject2, FD_WRITE);
```

To cancel the association and selection of network events on a socket, *lNetworkEvents* should be set to zero, in which case the *hEventObject* parameter will be ignored.

```
rc = WSPEventSelect(s, hEventObject, 0);
```

Closing a socket with **WSPCloseSocket()** also cancels the association and selection of network events specified in **WSPEventSelect()** for the socket. The application, however, still needs to call **WSACloseEvent()** to explicitly close the event object and free any resources.

Since a **WSAAccept()'**ed socket has the same properties as the listening socket used to accept it, any **WSPEventSelect()** association and network events selection set for the listening socket apply to the accepted socket. For example, if a listening socket has **WSPEventSelect()** association of *hEventOject* with FD_ACCEPT, FD_READ, and FD_WRITE, then any socket accepted on that listening socket will also have FD_ACCEPT, FD_READ, and FD_WRITE network events associated with the same *hEventObject*. If a different *hEventObject* or network events are desired, the application should call **WSPEventSelect()**, passing the accepted socket and the desired new information.[6]

Return Value The return value is 0 if the application's specification of the network events and the associated event object was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number is available in *lpErrno*.

Comments As in the case of the **WSPSelect()** and **WSPAsyncSelect32()** functions, **WSPEventSelect()** will frequently be used to determine when a data transfer operation (**WSPSend()** or **WSPRecv()**) can be issued with the expectation of immediate success. Nevertheless, a robust application must be prepared for the possibility that the event object is set and it issues a Winsock call which returns WSAEWOULDBLOCK immediately. For example, the following sequence of operations is possible:

---

[6]Note that there is a timing window between the **accept()** call and the call to **WSAEventSelect()** to change the network events or *hEventObject*. An application which desires a different *hEventObject* for the listening and **accept()**'ed sockets should ask for only FD_ACCEPT network event on the listening socket, then set appropriate network events after the **accept()**. Since FD_ACCEPT never happens to a connected socket and FD_READ, FD_WRITE, FD_OOB, and FD_CLOSE never happen to listening sockets, this will not impose difficulties.

(i) data arrives on socket **s**; Winsock sets the **WSPEventSelect** event object
(ii) application does some other processing
(iii) while processing, application issues an **WSPIoctlSocket(s, FIONREAD...)** and notices that there is data ready to be read
(iv) application issues a **WSPRecv(s,...)** to read the data
(v) application eventually waits on event object specified in **WSPEventSelect**, which returns immediately indicating that data is ready to read
(vi) application issues **WSPRecv(s,...)**, which fails with the error WSAEWOULDBLOCK.

Other sequences are possible.

Having successfully recorded the occurrence of the network event (by setting the corresponding bit in the internal network event record) and signaled the associated event object, no further actions are taken for that network event until the application makes the function call which implicitly reenables the setting of that network event and signaling of the associated event object.

Network Event	Re-enabling function
FD_READ	**WSPRecv()** or **WSPRecvFrom()**
FD_WRITE	**WSPSend()** or **WSPSendTo()**
FD_OOB	**WSPRecv()**
FD_ACCEPT	**WSPAccept()** unless the error code returned is WSATRY_AGAIN indicating that the condition function returned CF_DEFER
FD_CONNECT	NONE
FD_CLOSE	NONE
FD_QOS	**WSPGetSockOpt()** with option SO_FLOWSPEC
FD_GROUP_QOS	**WSPGetSockOpt()** with option SO_GROUP_FLOWSPEC

Any call to the reenabling routine, even one which fails, results in reenabling of recording and setting for the relevant network event and event object, respectively.

For FD_READ, FD_OOB, FD_ACCEPT, FD_QOS and FD_GROUP_QOS network events, network event recording and event object setting are "level-triggered." This means that if the reenabling routine is called and the relevant network condition is still valid after the call, the network event is recorded and the associated event object is set . This allows an application to be event-driven and not be concerned with the amount of data that arrives at any one time. Consider the following sequence:

(i) transport provider receives 100 bytes of data on socket **s** and causes Winsock2 DLL to record the FD_READ network event and set the associated event object.
(ii) The application issues **WSPRecv(s, buffptr, 50, 0)** to read 50 bytes.
(iii) The transport provider causes WINSOCK DLL to record the FD_READ network event and sets the associated event object again since there is still data to be read.

With these semantics, an application need not read all available data in response to an FD_READ network event --a single **WSPRecv()** in response to each FD_READ network event is appropriate.

If a network event has already happened when the application calls **WSPEventSelect()** or when the reenabling function is called, then a network event is recorded and the associated event object is set as appropriate. All the network events have persistence beyond the occurrence of their respective events. For example, consider the following sequence: 1) an application calls **WSPListen()**, 2) a connect request is received but not yet accepted, 3) the application calls **WSPEventSelect()** specifying that it is interested in the FD_ACCEPT network event for the socket. Due to the persistence of network events, Winsock records the FD_ACCEPT network event and sets the associated event object immediately.

The FD_WRITE network event is handled slightly differently. An FD_WRITE network event is recorded when a socket is first connected with **WSPConnect()** or accepted with **WSPAccept()**, and then after a **WSPSend()** or **WSPSendTo()** fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, an application can assume that sends are possible starting from the first FD_WRITE network event setting and lasting until a send returns WSAEWOULDBLOCK. After such a failure the application will find out that sends are again possible when an FD_WRITE network event is recorded and the associated event object is set .

The FD_OOB network event is used only when a socket is configured to receive out-of-band data separately. If the socket is configured to receive out-of-band data in-line, the out-of-band (expedited) data is treated as normal data and the application should register an interest in, and will get, FD_READ network event, not FD_OOB network event. An application may set or inspect the way in which out-of-band data is to be handled by using **WSPSetSockOpt()** or **WSPGetSockOpt()** for the SO_OOBINLINE option.

The error code in an FD_CLOSE network event indicates whether the socket close was graceful or abortive. If the error code is 0, then the close was graceful; if the error code is WSAECONNRESET, then the socket's virtual circuit was reset. This only applies to connection-oriented sockets such as SOCK_STREAM.

The FD_CLOSE network event is recorded when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the FD_CLOSE is recorded when the connection goes into the FIN WAIT or CLOSE WAIT states. This results from the remote end performing a **WSPShutdown()** on the send side or a **WSPCloseSocket().**

Please note Winsock will record ONLY an FD_CLOSE network event to indicate closure of a virtual circuit. It will NOT record an FD_READ network event to indicate this condition.

The FD_QOS or FD_GROUP_QOS network event is recorded when any field in the flow spec associated with socket *s* or the socket group that *s* belongs to has changed, respectively. Applications should use **WSPGetSockOpt()** with option SO_FLOWSPEC or SO_GROUP_FLOWSPEC to get the current QOS for socket *s* or for the socket group *s* belongs to, respectively.

Error Codes

WSANOTINITIALISED	A successful **WSPStartup()** must occur before using this SPI.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid, or the specified socket is in an invalid state.

WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section ???).
WSAENOTSOCK	The descriptor is not a socket.

See Also **WSPEnumNetworkEvents()**

3.1.19 WSPIsBlocking32()

Description Determine if a blocking call is in progress.

#include <ws2spi.h>

BOOL WSPAPI WSPIsBlocking32 (**VOID**);

Remarks This function is only applicable to the 32-bit SPI.

This function allows the Winsock DLL to determine if it is executing while waiting for a previous blocking call to complete.

Return Value The return value is TRUE if there is an outstanding blocking function awaiting completion. Otherwise, it is FALSE.

See Also **WSPCancelBlockingCall32(), WSPSetBlockingHook32(), WSPUnhookBlockingHook32()**

3.1.20 WSPRecv()

Description Receive data from a socket. {warning! Since both the recv() and WSARecv() API functions map to this function the description needs to be more generic with respect to overlapped I/O This also applies to WSPRecvFrom, WSPSend, WSPSendTo}

#include <ws2spi.h>

int WSPAPI WSPRecv (SOCKET *s*, **LPVOID** *lpBuffer*, **DWORD** *nNumberOfBytesToRecv*, **LPDWORD** *lpNumberOfBytesRecvd*, **LPINT** *lpFlags,* **LPWSAOVERLAPPED** *lpOverlapped*, **LPWSAOVERLAPPED_COMPLETION_ROUTINE** *lpCompletionRoutine*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying a connected socket.
lpBuffer	A pointer to the buffer for the incoming data.
nNumberOfBytesToRecv	The number of bytes to receive from the network.
lpNumberOfBytesRecvd	A pointer to the number of bytes received by this call.
lpFlags	A pointer to flags.
lpOverlapped	A pointer to a WSAOVERLAPPED structure.
lpCompletionRoutine	A pointer to the completion routine called when the receive operation has been completed.
lpErrno	A pointer to the error code.

Remarks This function is used on a connection-oriented socket specified by *s* to post a buffer into which incoming data will be placed as it becomes available. It can also be used on connectionless sockets which have a stipulated default peer address established via the **WSPConnect()** function.

For non-overlapped socket, this function behaves like the standard **recv()** API with identical blocking semantics and the *lpOverlapped* and *lpCompletionRoutine* parameters are ignored. The completion status of this SPI is the final completion status of the receive operation.

For overlapped sockets, the final completion status is retrieved via the **WSAGetOverlappedResult()** API.

For byte stream style sockets (e.g., type SOCK_STREAM), incoming data is placed into the buffer until the buffer is filled. For message-oriented sockets (e.g., type SOCK_DGRAM), an incoming message is placed into the supplied buffer, up to the size of the buffer supplied. If the message is larger than the buffer supplied, the buffer is filled with the first part of the message, the excess data is lost, and **WSPRecv()** indicates the error WSAEMSGSIZE.

If the socket is connection-oriented and the remote side has shut down the connection gracefully, **WSPRecv()** will complete immediately indicating zero bytes received. If the connection has been reset, a **WSPRecv()** will fail with the error WSAECONNRESET.

This function may be called from within the completion routine of a previous **WSPRecv()**, **WSPRecvFrom()**, **WSPSend()** or **WSPSendTo()** function.

When called with an overlapped socket, the *lpOverlapped* parameter much be valid for the duration of the overlapped operation. The WSAOVERLAPPED structure has the following form:

```
typedef struct _WSAOVERLAPPED {
      DWORD       Internal;          // reserved
      DWORD       InternalHigh;      // reserved
      DWORD       Offset;            // ignored
      DWORD       OffsetHigh;        // ignored
      WSAEVENT    hEvent;
} WSAOVERLAPPED, LPWSAOVERLAPPED;
```

The *lpCompletionRoutine* must be non-NULL and the *hEvent* field is used to pass context information to the completion routine.

The prototype of the completion routine is as follows:

VOID CALLBACK CompletionRoutine(DWORD *dwError***,**
DWORD *cbTransferred***, LPWSAOVERLAPPED** *lpOverlapped* **);**

CompletionRoutine is a placeholder for an application-defined or library-defined function. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes received. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order they are supplied

Upon the completion of the overlapped operation, the *lpNumberOfBytesRecvd* parameter is filled with the number of bytes received, and, for message-oriented sockets, the MSG_PARTIAL bit is set in the *lpFlags* parameter if a partial message is received. If a complete message is received, MSG_PARTIAL is cleared in *lpFlags*.

Return Value

If no error occurs, **WSPRecv()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

Error Codes

WSANOTINITIALISED	A successful **WSPStartup()** must occur before using this SPI.
WSAENETDOWN	The network subsystem has failed.
WSAENOTCONN	The socket is not connected.
WSAENETRESET	The connection must be reset because the service provider dropped it.

WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **WSPRecv()** on a socket after **WSPShutdown()** has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	There are too many outstanding overlapped I/O requests.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated. Any trailing portion of the message that did not fit into the buffer has been discarded. {Does this make sense?}
WSAEINVAL	The socket has not been bound with **WSPBind()**, or the socket is not created with the overlapped flag**.**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.

See Also **WSPSocket()**

3.1.21 WSPRecvFrom()

Description Receive a datagram and store the source address.

#include <ws2spi.h>

int WSPAPI WSPRecvFrom (SOCKET *s*, **LPVOID** *lpBuffer*, **DWORD** *nNumberOfBytesToRecv*, **LPDWORD** *lpNumberOfBytesRecvd*, **LPINT** *lpFlags,* **LPVOID** *lpFrom*, **LPINT** *lpFromlen,* **LPWSAOVERLAPPED** *lpOverlapped*, **LPWSAOVERLAPPED_COMPLETION_ROUTINE** *lpCompletionRoutine*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying a socket.
lpBuffer	A pointer to the buffer for the incoming data.
nNumberOfBytesToRecv	The number of bytes to receive from the network.
lpNumberOfBytesRecvd	A pointer to the number of bytes received by this call.
lpFlags	A pointer to flags.
lpFrom	An optional pointer to a buffer which will hold the source address upon the completion of the overlapped operation.
lpFromlen	An optional pointer to the size of the *from* buffer.
lpOverlapped	A pointer to a WSAOVERLAPPED structure.
lpCompletionRoutine	A pointer to the completion routine called when the receive operation has been completed.
lpErrno	A pointer to the error code.

Remarks This function is used to post a buffer into which incoming data will be placed as it becomes available on a (possibly connected) socket. For connectionless socket types, the address from which the data originated is copied to the buffer pointed by *lpFrom*. The value pointed to by *lpFromlen* is initialized to the size of this buffer, and is modified on return to indicate the actual size of the address stored there. The *lpFrom* and *lpFromlen* parameters are ignored for connection-oriented sockets.

For non-overlapped socket, this function behaves like the standard **recvfrom()** API with identical blocking semantics. The *lpOverlapped* and *lpCompletionRoutine* parameters are ignored. The completion status of this SPI is the final completion status of the receive operation.

For overlapped sockets, the final completion status is retrieved via the **WSAGetOverlappedResult()** API.

For byte stream style sockets (e.g., type SOCK_STREAM), incoming data is placed into the buffer until the buffer is filled. For message-oriented sockets, an incoming message is placed into the supplied buffer, up to the size of the buffer supplied. If the message is

larger than the buffer supplied, the buffer is filled with the first part of the message, the excess data is lost, and **WSPRecvFrom()** indicates the error code WSAEMSGSIZE.

If the socket is connection-oriented and the remote side has shut down the connection gracefully, a **WSPRecvFrom()** will complete immediately with 0 bytes received. If the connection has been reset **WSPRecvFrom()** will fail with the error WSAECONNRESET.

This function may be called from within the completion routine of a previous **WSPRecv()**, **WSPRecvFrom()**, **WSPSend()** or **WSPSendTo()** function.

When called with an overlapped socket, the *lpOverlapped* parameter much be valid for the duration of the overlapped operation. The WSAOVERLAPPED structure has the following form:

```
typedef struct _WSAOVERLAPPED {
      DWORD       Internal;          // reserved
      DWORD       InternalHigh;      // reserved
      DWORD       Offset;            // ignored
      DWORD       OffsetHigh;        // ignored
      WSAEVENT    hEvent;
} WSAOVERLAPPED, LPWSAOVERLAPPED;
```

The *lpCompletionRoutine* must be non-NULL and the *hEvent* field is used to pass context information to the completion routine.

The prototype of the completion routine is as follows:

VOID CALLBACK CompletionRoutine(DWORD *dwError***,**
DWORD *cbTransferred***, LPWSAOVERLAPPED** *lpOverlapped* **);**

CompletionRoutine is a placeholder for an application-defined or library-defined function. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes received. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order they are supplied

Upon the completion of the overlapped operation, the *lpNumberOfBytesRecvd* parameter is filled with the number of bytes received, and, for message-oriented sockets, the MSG_PARTIAL bit is set in the *lpFlags* parameter if a partial message is received. If a complete message is received, MSG_PARTIAL is cleared in *lpFlags*.

Return Value

If no error occurs, **WSPRecvFrom()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

Error Codes

WSANOTINITIALISED	A successful **WSPStartup()** must occur before using this SPI.
WSAENETDOWN	The network subsystem has failed.

WSAEFAULT	The *lpFromlen* argument was invalid: the *lpFrom* buffer was too small to accommodate the peer address.
WSAEINVAL	The socket has not been bound with **WSPBind()**, or the socket is not created with the overlapped flag**.**
WSAENETRESET	The connection must be reset because the Winsock provider dropped it.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **WSPRecvFrom()** on a socket after **shutdown()** has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	There are too many outstanding overlapped I/O requests.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated. Any trailing portion of the message that did not fit into the buffer has been discarded. {Does this make sense?}
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.

See Also **WSPSocket()**

3.1.22 WSPSend()

Description Send data on a connected socket.

#include <ws2spi.h>

int WSPAPI WSPSend (SOCKET *s*, **LPVOID** *lpBuffer*, **DWORD** *nNumberOfBytesToSend*, **LPDWORD** *lpNumberOfBytesSent*, **int** *nFlags,* **LPWSAOVERLAPPED** *lpOverlapped*, **LPWSAOVERLAPPED_COMPLETION_ROUTINE** *lpCompletionRoutine,* **int** FAR * *lpErrno* **);**

s	A descriptor identifying a connected socket.
lpBuffer	A pointer to the buffer for the outgoing data.
nNumberOfBytesToSend	The number of bytes to send to the network.
lpNumberOfBytesSent	A pointer to the number of bytes sent by this call.
iFlags	Flags.
lpOverlapped	A pointer to a WSAOVERLAPPED structure.
lpCompletionRoutine	A pointer to the completion routine called when the send operation has been completed.
lpErrno	A pointer to the error code.

Remarks **WSPSend**() is used to write outgoing data on a connected socket. For message-oriented sockets, it is an error to exceed the maximum packet size of the underlying transport, which can be obtained by getting the value of socket option SO_MAX_DG_SIZE. If the data is too long to pass atomically through the underlying protocol the error WSAEMSGSIZE is returned, and no data is transmitted.

Note that the successful completion of a **WSPSend()** does not indicate that the data was successfully delivered.

For non-overlapped socket, this function behaves like the standard **send()** API with identical blocking semantics. The *lpOverlapped* and *lpCompletionRoutine* parameters are ignored. The completion status of this SPI is the final completion status of the receive operation.

For overlapped sockets, the final completion status is retrieved via the **WSAGetOverlappedResult()** API.

This function may be called from within the completion routine of a previous **WSPRecv()**, **WSPRecvFrom()**, **WSPSend()** or **WSPSendTo()** function.

When called with an overlapped socket, the *lpOverlapped* parameter much be valid for the duration of the overlapped operation. The WSAOVERLAPPED structure has the following form:

```
typedef struct _WSAOVERLAPPED {
      DWORD       Internal;         // reserved
      DWORD       InternalHigh;     // reserved
      DWORD       Offset;           // ignored
      DWORD       OffsetHigh;       // ignored
      WSAEVENT    hEvent;
} WSAOVERLAPPED, LPWSAOVERLAPPED;
```

The *lpCompletionRoutine* must be non-NULL and the *hEvent* field is used to pass context information to the completion routine.

The prototype of the completion routine is as follows:

VOID CALLBACK CompletionRoutine(DWORD *dwError***,**
DWORD *cbTransferred***, LPWSAOVERLAPPED** *lpOverlapped* **);**

CompletionRoutine is a placeholder for an application-defined or library-defined function. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes received. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be sent in the same order they are supplied
Flags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Winsock service provider may choose to ignore this flag; see also the discussion of the SO_DONTROUTE option in section 2.8..
MSG_PARTIAL	Specifies that *lpBuffer* only contains a partial message. Note that this flag will be ignored by transports which do not support partial message transmissions.

Upon the completion of the overlapped operation, the *lpNumberOfBytesSent* parameter is filled with the number of bytes sent.

Return Value

If no error occurs, **WSPSend()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

Error Codes

WSANOTINITIALISED	A successful **WSPStartup()** must occur before using this SPI.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.

WSAEFAULT	The *lpBuffer* argument is not in a valid part of the user address space.
WSAENETRESET	The connection must be reset because the Winsock provider dropped it.
WSAENOBUFS	The Winsock provider reports a buffer deadlock.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **WSPSend()** on a socket after **WSPShutdown()** has been invoked with how set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	There are too many outstanding overlapped I/O requests.
WSAEMSGSIZE	The socket is message-oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	The socket has not been bound with **WSPBind()**, or the socket is not created with the overlapped flag**.**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.

See Also **WSPSocket()**

3.1.23 WSPSendTo()

Description Send data to a specific destination using overlapped I/O.

#include <ws2spi.h>

int WSPAPI WSPSendTo (SOCKET *s*, **LPVOID** *lpBuffer*, **DWORD** *nNumberOfBytesToSend*, **LPDWORD** *lpNumberOfBytesSent*, **int** *nFlags,* **LPVOID** *lpTo,* **int** *nTolen,* **LPWSAOVERLAPPED** *lpOverlapped*, **LPWSAOVERLAPPED_COMPLETION_ROUTINE** *lpCompletionRoutine*, **int** FAR * *lpErrno* **);**

s	A descriptor identifying a socket.
lpBuffer	A pointer to the buffer for the outgoing data.
nNumberOfBytesToSend	The number of bytes to send to the network.
lpNumberOfBytesSent	A pointer to the number of bytes sent by this call.
iFlags	Flags.
lpTo	An optional pointer to the address of the target socket.
iTolen	The size of the address in *lpTo*.
lpOverlapped	A pointer to a WSAOVERLAPPED structure.
lpCompletionRoutine	A pointer to the completion routine called when the send operation has been completed.
lpErrno	A pointer to the error code.

Remarks **WSPSendTo**() is used to write outgoing data on a socket. For message-oriented sockets, it is an error to exceed the maximum packet size of the underlying transport, which can be obtained by getting the value of socket option SO_MAX_DG_SIZE. If the data is too long to pass atomically through the underlying protocol the error WSAEMSGSIZE is returned, and no data is transmitted.

Note that the successful completion of a **WSPSendTo()** does not indicate that the data was successfully delivered.

For non-overlapped socket, this function behaves like the standard **sendto()** API with identical blocking semantics. The *lpOverlapped* and *lpCompletionRoutine* parameters are ignored. The completion status of this SPI is the final completion status of the receive operation.

For overlapped sockets, the final completion status is retrieved via the **WSAGetOverlappedResult()** API.

WSPSendTo() is normally used on a connectionless socket to send a datagram to a specific peer socket identified by the *lpTo* parameter. On a connection-oriented socket,

the *lpTo* and *iTolen* parameters are ignored; in this case the **WSPSendTo()** is equivalent to **WSPSend**().

This function may be called from within the completion routine of a previous **WSPRecv()**, **WSPRecvFrom()**, **WSPSend()** or **WSPSendTo()** function.

When called with an overlapped socket, the *lpOverlapped* parameter much be valid for the duration of the overlapped operation. The WSAOVERLAPPED structure has the following form:

```
typedef struct _WSAOVERLAPPED {
      DWORD       Internal;          // reserved
      DWORD       InternalHigh;      // reserved
      DWORD       Offset;            // ignored
      DWORD       OffsetHigh;        // ignored
      WSAEVENT    hEvent;
} WSAOVERLAPPED, LPWSAOVERLAPPED;
```

The *lpCompletionRoutine* must be non-NULL and the *hEvent* field is used to pass context information to the completion routine.

The prototype of the completion routine is as follows:

VOID CALLBACK CompletionRoutine(**DWORD** *dwError***,**
DWORD *cbTransferred***, LPWSAOVERLAPPED** *lpOverlapped* **);**

CompletionRoutine is a placeholder for an application-defined or library-defined function. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes received. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be sent in the same order they are supplied
Flags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A WINSOCK service provider may choose to ignore this flag; see also the discussion of the SO_DONTROUTE option in section 2.8..
MSG_PARTIAL	Specifies that *lpBuffer* only contains a partial message. Note that this flag will be ignored by transports which do not support partial message transmissions.

Upon the completion of the overlapped operation, the *lpNumberOfBytesSent* parameter is filled with the number of bytes sent.

Return Value If no error occurs, **WSPSendTo()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

Error Codes

WSANOTINITIALISED	A successful **WSPStartup()** must occur before using this SPI.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.
WSAEFAULT	The *lpBuffer* or *lpTo* parameters are not part of the user address space, or the *lpTo* argument is too small.
WSAENETRESET	The connection must be reset because the Winsock provider dropped it.
WSAENOBUFS	The Winsock provider reports a buffer deadlock.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only)
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **WSPSendTo()** on a socket after **WSPShutdown()** has been invoked with how set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	There are too many outstanding overlapped I/O requests.
WSAEMSGSIZE	The socket is message-oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	The socket has not been bound with **WSPBind()**, or the socket is not created with the overlapped flag**.**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.

WSAEDESTADDRREQ	A destination address is required.
WSAENETUNREACH	The network can't be reached from this host at this time.

See Also **WSPSocket()**

3.1.24 WSPSetBlockingHook32()

Description Establish an application-specific blocking hook function.

#include <ws2spi.h>

LPBLOCKINGPRC WSPAPI WSPSetBlockingHook32 (**LPBLOCKINGPROC** lpBlockFunc, **int** FAR * *lpErrno*);

lpBlockFunc A pointer to the procedure instance address of the blocking function to be installed.

lpErrno A pointer to the error code.

Remarks This function is only applicable to the 32-bit SPI.

This function installs a new function which the Winsock Service Provider should use to implement blocking socket function calls.

Winsock Service Providers include a default mechanism by which blocking socket functions are implemented. The function **WSPSetBlockingHook32()** gives the application the ability to execute its own function at "blocking" time in place of the default function.

When an application invokes a blocking Windows Sockets API operation, the Winsock Service Provider initiates the operation and then enters a loop which is similar to the following pseudocode:

```
for(;;) {
     /* flush messages for good user response */
     while(BlockingHook())
          ;
     /* check for WSPCancelBlockingCall32() */
     if(operation_cancelled())
          break;
     /* check to see if operation completed */
     if(operation_complete())
          break;     /* normal completion */
}
```

Note that Winsock Service Providers may perform the above steps in a different order; for example, the check for operation complete may occur before calling the blocking hook. The default **BlockingHook()** function is equivalent to:

```
BOOL DefaultBlockingHook(void) {
     MSG msg;
     BOOL ret;
     /* get the next message if any */
     ret = (BOOL)PeekMessage(&msg,NULL,0,0,PM_REMOVE);
     /* if we got one, process it */
     if (ret) {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }
     /* TRUE if we got a message */
     return ret;
```

```
}
```

The **WSPSetBlockingHook32()** function is provided to support those applications which require more complex message processing - for example, those employing the MDI (multiple document interface) model. It is not intended as a mechanism for performing general applications functions. In particular, the only Windows Sockets SPI function which may be issued from a custom blocking hook function is **WSPCancelBlockingCall32()**, which will cause the blocking loop to terminate.

This function must be implemented on a per-thread basis for multithreaded versions of Windows such as Windows NT. It thus provides for a particular thread to replace the blocking mechanism without affecting other threads.

In multithreaded versions of Windows, there is no default blocking hook--blocking calls block the thread that makes the call. However, an application may install a specific blocking hook by calling **WSPSetBlockingHook32().** This allows easy portability of applications that depend on the blocking hook behavior.

Return Value

The return value is a pointer to the procedure-instance of the previously installed blocking function. The Winsock DLL should save this return value so that it can be restored if necessary. (If "nesting" is not important, the Winsock DLL may simply discard the value returned by **WSPSetBlockingHook32()** and eventually use **WSPUnhookBlockingHook32()** to restore the default mechanism.) If the operation fails, a NULL pointer is returned, and a specific error code is available in *lpErrno*.

Error Codes

WSANOTINITIALISED	A successful **WSPStartup()** must occur before using this SPI.
WSAENETDOWN	The Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.

See Also

WSPCancelBlockingCall32(), WSPIsBlocking32(), WSPUnhookBlockingHook32()

3.1.25 WSPSocket()

Description Create a socket which is bound to a specific transport service provider.

{ May be revised to use PROTOCOL_INFO as an input param in place of *af, type, protocol* parameters}

#include <ws2spi.h>

SOCKET WSPAPI WSPSocket (int *af***, int** *type***, int** protocol, **DWORD** *dwProviderId*, **int** *nFlags*, **int** FAR * *lpErrno* **);**

af	An address family specification.
type	A type specification for the new socket.
protocol	A particular protocol to be used with the socket, or 0 if the caller does not wish to specify a protocol.
dwProviderId	An opaque value returned by the **WSAEnumProtocols()** API that identifies a unique service provider. This information is useful to provider implementations that present multiple service provider appearances to Winsock.
iFlags	The socket attribute specification.
lpErrno	A pointer to the error code.

Remarks **WSPSocket()** causes a socket descriptor and any related resources to be allocated and bound to the transport service provider specified by *dwProviderId*. If *protocol* is not specified (i.e., equal to zero), the default for the specified socket type is used. However, the address family may be given as AF_UNSPEC (unspecified), in which case the *protocol* parameter must be specified. The protocol number to use is particular to the "communication domain" in which communication is to take place.

{The new name res stuff should obviate the need to enumerate the list of supported socket types here}

The *iFlags* parameter may be used to specify the attributes of the socket by or-ing any of the following Flags:

Flag	Meaning
WSA_FLAG_OVERLAPPED	This flag causes an overlapped socket to be created. Overlapped sockets must utilize the overlapped I/O features of the **WSPSend()**, **WSPSendTo()**, **WSPRecv()**, **WSPRecvFrom()** SPI for I/O operations, and allows multiple of these to be initiated and in progress simultaneously. Overlapped sockets are always non-blocking.

Connection-oriented sockets such as SOCK_STREAM provide full-duplex connections, and must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a **connect**() call. Once connected, data may be transferred using **WSPSend**() and **WSPRecv**() calls. When a session has been completed, a **WSPCloseSocket**() must be performed.

The communications protocols used to implement a reliable, connection-oriented socket ensure that data is not lost or duplicated. If data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and subsequent calls will fail with the error code set to WSAETIMEDOUT.

Connectionless, message-oriented sockets allow sending and receiving of datagrams to and from arbitrary peers using **WSPSendTo()** and **WSPRecvFrom()**. If such a socket is **connect()**ed to a specific peer, datagrams may be send to that peer using **WSPSend**() and may be received from (only) this peer using **WSPRecv**().

Return Value If no error occurs, **WSPSocket()** returns a descriptor referencing the new socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code is available in *lpErrno***.**

Error Codes

WSANOTINITIALISED	A successful **WSPStartup()** must occur before using this SPI.
WSAENETDOWN	The network subsystem has failed.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT	The specified protocol is not supported.
WSAEPROTOTYPE	The specified protocol is the wrong type for this socket.
WSAESOCKTNOSUPPORT	The specified socket type is not supported in this address family.

See Also **WSPAccept**, **WSPBind**(), **WSPConnect**(), **WSPGetSockName**(), **WSPGetSockOpt**(), **WSPSetSockOpt**(), **WSPListen**(), **WSPRecv**(), **WSARecvFrom**(), **WSASend**(), **WSASendTo**(), **WSPShutdown**(), **WSPIoctlSocket**().

3.1.26 WSPStartup()

Description

#include <ws2spi.h>

int WSPAPI WSPStartup (WORD *wVersionRequested***,**
LPWSADATA *lpWSAData* **);**

wVersionRequested	The highest version of Winsock SPI support that the caller can use. The high order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.
lpWSAData	A pointer to the **WSADATA** data structure that is to receive details of the Winsock service provider.

Remarks

This function **MUST** be the first Winsock SPI function called by the Winsock DLL. It allows the Winsock DLL to specify the version of Winsock SPI required and to retrieve details of the specific Winsock service provider implementation. The Winsock DLL may only issue further Winsock SPI functions after a successful **WSPStartup()** invocation.

In order to support future Winsock service providers and the Winsock DLL which may have functionality differences from the current Winsock SPI, a negotiation takes place in **WSPStartup()**. The caller of **WSPStartup()** (namely, the Winsock DLL) and the Winsock service provider indicate to each other the highest version that they can support, and each confirms that the other's highest version is acceptable. Upon entry to **WSPStartup()**, the Winsock service provider examines the version requested by the Winsock DLL. If this version is higher than the lowest version supported by the service provider, the call succeeds and the service provider returns in *wHighVersion* the highest version it supports and in *wVersion* the minimum of its high version and *wVersionRequested.* The Winsock service provider then assumes that the Winsock DLL will use *wVersion.* If the *wVersion* field of the **WSADATA** structure is unacceptable to the caller, it should call **WSPCleanup()** and either search for another Winsock service provider or fail to initialize.

This negotiation allows both a Winsock service provider and the Winsock DLL to support a range of Winsock versions. The Winsock DLL can successfully utilize a Winsock service provider if there is any overlap in the version ranges. The following chart gives examples of how **WSPStartup()** works in conjunction with different DLL and Winsock service provider (SP) versions:

DLL versions	SP Versions	*wVersionRequested*	*wVersion*	*wHighVersion*	End Result
1.1	1.1	1.1	1.1	1.1	use 1.1
1.0 1.1	1.0	1.1	1.0	1.0	use 1.0
1.0	1.0 1.1	1.0	1.0	1.1	use 1.0
1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1	1.0	1.1	1.0	1.0	DLL fails
1.0	1.1	1.0	---	---	WSAVERNOTSUPPORTED
1.0 1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1 2.0	1.1	2.0	1.1	1.1	use 1.1
2.0	1.1	2.0	1.1	1.1	DLL fails

The following code fragment demonstrates how a DLL which supports only version 1.1 of Winsock SPI makes a **WSPStartup()** call:

```
WORD wVersionRequested;
WSADATA WSAData;
int err;

wVersionRequested = MAKEWORD( 1, 1 );

err = WSPStartup( wVersionRequested, &WSAData );
if ( err != 0 ) {
    /* Tell the user that we couldn't find a useable */
    /* Winsock service provider.                     */
    return;
}

/* Confirm that the Winsock service provider supports 1.1.*/
/* Note that if the service provider supports versions
greater    */
/* than 1.1 in addition to 1.1, it will still return */
/* 1.1 in wVersion since that is the version we      */
/* requested.                                        */

if ( LOBYTE( WSAData.wVersion ) != 1 ||
         HIBYTE( WSAData.wVersion ) != 1 ) {
    /* Tell the user that we couldn't find a useable */
    /* Winsock service provider.                     */
    WSPCleanup( );
    return;
}

/* The Winsock service provider is acceptable.  Proceed. */
```

And this code fragment demonstrates how a Winsock service provider which supports only version 1.1 performs the **WSPStartup()** negotiation:

```
/* Make sure that the version requested is >= 1.1.   */
/* The low byte is the major version and the high    */
/* byte is the minor version.                        */

if ( LOBYTE( wVersionRequested ) < 1 ||
     ( LOBYTE( wVersionRequested ) == 1 &&
       HIBYTE( wVersionRequested ) < 1 ) {
    return WSAVERNOTSUPPORTED;
}

/* Since we only support 1.1, set both wVersion and  */
/* wHighVersion to 1.1.                              */

lpWSAData->wVersion = MAKEWORD( 1, 1 );
lpWSAData->wHighVersion = MAKEWORD( 1, 1 );
```

Once the Winsock DLL has made a successful **WSPStartup()** call, it may proceed to make other Winsock SPI calls as needed. When it has finished using the services of the Winsock service provider, the Winsock DLL must call **WSPCleanup()** in order to allow the Winsock service provider to free any resources for the Winsock DLL.

Details of the actual Winsock service provider are described in the **WSAData** structure defined as follows:

```
struct WSAData {
      WORD              wVersion;
      WORD              wHighVersion;
      char              szDescription[WSADESCRIPTION_LEN+1];
      char              szSystemStatus[WSASYSSTATUS_LEN+1];

      char FAR *        lpVendorInfo;
};
```

The members of this structure are:

Element	Usage
wVersion	The version of the Winsock SPI specification that the Winsock service provider expects the caller to use.
wHighVersion	The highest version of the Winsock SPI specification that this service provider can support (also encoded as above). Normally this will be the same as *wVersion*.
szDescription	A null-terminated ASCII string into which the Winsock DLL copies a description of the Winsock service provider. The text (up to 256 characters in length) may contain any characters except control and formatting characters: the most likely use that an application will put this to is to display it (possibly truncated) in a status message.
szSystemStatus	A null-terminated ASCII string into which the Winsock service provider copies relevant status or configuration information. The Winsock service provider should use this field only if the information might be useful to the user or support staff: it should not be considered as an extension of the *szDescription* field.
. . **WSPGetSockOpt**. lpVendorInfo	This value should be ignored. It is retained for compatibility with Windows Sockets specification 1.1. The Winsock DLL needing to access vendor-specific configuration information should use **WSPGetSockOpt()** to retrieve the value of option PVD_CONFIG. The definition of this value (if utilized) is beyond the scope of this specification.

Return Value

WSPStartup() returns zero if successful. Otherwise it returns one of the error codes listed below.

Notes For Winsock Service Providers

The Winsock DLL will make one and only one **WSPStartup()** call before issuing any other Winsock SPI calls for each Winsock service provider. This function can thus be utilized for initialization purposes.

Further issues are discussed in the notes for **WSPCleanup()**.

Error Codes

WSASYSNOTREADY	Indicates that the underlying network subsystem is not ready for network communication.
WSAVERNOTSUPPORTED	The version of Winsock SPI support requested is not provided by this particular Winsock service provider.
WSAEFAULT	The *lpWSAData* parameter is invalid.

See Also

WSPSend(), **WSPSendTo()**, **WSPCleanup()**

3.1.27 WSPUnhookBlockingHook32()

Description Restores the default blocking hook function.

#include <ws2spi.h>

int WSPAPI WSPUnhookBlockingHook32 (**int** FAR * *lpErrno*);

lpErrno A pointer to the error code.

Remarks This function is only applicable to the 32-bit SPI.

This function removes any previous blocking hook that has been installed and reinstalls the default blocking mechanism.

WSPUnhookBlockingHook32() will always install the default mechanism, not the previous mechanism. If the Winsock DLL wishes to nest blocking hooks - i.e. to establish a temporary blocking hook function and then revert to the previous mechanism (whether the default or one established by an earlier **WSPSetBlockingHook32()**) - it must save and restore the value returned by **WSPSetBlockingHook32()**; it cannot use **WSPUnhookBlockingHook32()**.

In multithreaded versions of Windows such as Windows NT, there is no default blocking hook. Calling **WSPUnhookBlockingHook32()** disables any blocking hook installed by the application and any blocking calls made block the thread which made the call.

Return Value The return value is 0 if the operation has been successfully initiated. Otherwise the value SOCKET_ERROR is returned, and a specific error number is available in *lpErrno*.

Error Codes

WSANOTINITIALISED	A successful **WSPStartup()** must occur before using this SPI.

See Also **WSPCancelBlockingCall32(), WSPIsBlocking32(), WSPSetBlockingHook32()**

4. Upcalls

This section contains the private “upcalls” that service providers may make into the Windows Sockets DLL.

4.1 WPUCreateSocketHandle()

Description Creates a new socket handle.

#include <ws2spi.h>

SOCKET WSPAPI WPUCreateSocketHandle (DWORD *dwProviderId*, **LPVOID** *lpContext*, **int** FAR * *lpErrno* **);**

dwProviderId Identifies the calling service provider. *{We must provide a mechanism for getting this to the provider, perhaps at **WspStartup()** time?}*

lpContext A context value to associate with the new socket handle.

lpErrno A pointer to the error code.

Remarks This routine creates a new socket handle for the specified provider. The handle returned is Winsock specific; it is not a proper system handle.

This routine is only used by providers that do not provide real system handles.

Return Value If no error occurs, **WPUCreateSocketHandle()** returns the new socket handle. Otherwise, it returns INVALID_SOCKET, and a specific error code is available in *lpErrno*.

Error Codes WSAENOBUFS Not enough buffers available, too many sockets.

See Also **WPUCloseSocketHandle()**.

4.2 WPUCloseSocketHandle()

Description Closes an existing socket handle.

#include <ws2spi.h>

int WSPAPI WPUCloseSocketHandle (SOCKET *s*, **int** FAR * *lpErrno* **);**

s	Identifies a socket handle created with **WPUCreateSocketHandle()**.
lpErrno	A pointer to the error code.

Remarks This routine closes an existing socket handle. This function removes the socket from Winsock's internal socket table. The owning service provider is responsible for releasing any resources associated with the socket.

Return Value If no error occurs, **WPUCreateSocketHandle()** returns 0. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

Error Codes

WSAENOTSOCK	The descriptor is not a socket created by **WPUCreateSocketHandle()**.

See Also **WPUCreateSocketHandle()**.

4.3 WPUQuerySocketHandleContext()

Description Queries the context value associated with the specified socket handle.

#include <ws2spi.h>

int WSPAPI WPUQuerySocketHandleContext (SOCKET *s*, **LPVOID** FAR * *lpContext*, **int** FAR * *lpErrno* **);**

s	Identifies the socket whose context is to be queried.
lpContext	A pointer to an LPVOID that will receive the context value.
lpErrno	A pointer to the error code.

Remarks This routine queries the current context value associated with the specified socket handle. Service providers typically use this function to retrieve a pointer to provider-specific data associated with the socket. For example, a service provider may use the socket context to store a pointer to a structure containing the socket's state, local and remote transport addresses, event objects for signaling network events, etc.

Return Value If no error occurs, **WPUQuerySocketHandleContext()** returns 0 and stores the current context value in *lpContext*. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

Error Codes

WSAENOTSOCK	The descriptor is not a socket created by **WPUCreateSocketHandle()**.

See Also **WPUCreateSocketHandle(), WPUSetSocketHandleContext()**.

4.4 WPUSetSocketHandleContext()

Description Sets the context value associated with the specified socket handle.

#include <ws2spi.h>

int WSPAPI WPUSetSocketHandleContext (SOCKET *s*, **LPVOID** *lpContext*, **int** FAR * *lpErrno* **);**

s	Identifies the socket whose context is to be set.
lpContext	The new context value to associate with the socket.
lpErrno	A pointer to the error code.

Remarks This routine sets the current context value associated with the specified socket handle.

Return Value If no error occurs, **WPUSetSocketHandleContext()** returns 0 and stores *lpContext* and the socket's new context value. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

Error Codes

WSAENOTSOCK	The descriptor is not a socket created by **WPUCreateSocketHandle()**.

See Also **WPUCreateSocketHandle(), WPUQuerySocketHandleContext()**.

4.5 WPUQueueUserAPC32()

Description Queues a user-mode APC against the specified thread.

#include <ws2spi.h>

int WSPAPI WPUQueueUserAPC32 (DWORD *dwThreadId*, **LPWSAUSERAPC** *lpfnUserApc*, **DWORD** *dwContext*, **int** FAR * *lpErrno* **);**

dwThreadId	Identifies the target thread for the APC. This is typically a value returned by the **WPUGetCurrentThreadId()** upcall.
lpfnUserApc	Points to the function to be called as a user-mode APC.
dwContext	A context value to be passed in to the user-mode APC function.
lpErrno	A pointer to the error code.

Remarks This routine queues a user-mode APC against the specified thread. The APC will only execute when the specified thread is blocked in an alertable wait. This function is only available in 32 bit versions of the Winsock 2 DLL.

LPWSAUSERAPC is defined as follows:

typedef **VOID** (FAR * **LPWSAUSERAPC**)(**DWORD** dwContext);

Return Value If no error occurs, **WPUQueueUserAPC32()** returns 0 and queues the APC for the specified thread. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

Error Codes

????	dwThreadId does not specify a valid thread.

See Also **WPUGetCurrentThreadId32()**.

4.6 WPUGetCurrentThreadId32()

Description Returns an operating-system specific identifier for the current thread.

#include <ws2spi.h>

DWORD WSPAPI WPUGetCurrentThreadId32 (VOID);

Remarks This routine returns an operating-system specific identifier for the current thread. The returned value is suitable for use in the **WPUQueueUserAPC32()** upcall.

Return Value **WPUGetCurrentThreadId32()** returns the current thread ID.

See Also **WPUQueueUserAPC32()**.

5. Installation APIs

*This section is **super** preliminary. These APIs will be receive additional attention after the December 12th meeting.*

5.1 WPUInstallProvider()

Description Installs the specified provider into the system configuration database.

#include <ws2spi.h>

int WSPAPI WPUInstallProvider(const **char** FAR * *lpszProviderName*, const **char** FAR * *lpszProviderDllPath*, const **PROTOCOL_INFO** FAR * *lpProtocolInfoList*, **DWORD** *dwNumberOfEntries*, **DWORD** FAR * *lpdwProviderId*, **int** FAR * *lpErrno*);

lpszProviderName	Points to locally unique name for this provider. This name must not conflict with any currently installed provider.
lpszProviderDllPath	Points to a fully qualified path to the provider's DLL image. The Winsock DLL passes this path into the LoadLibrary API to load the provider.
lpProtocolInfoList	Points to an array of **PROTOCOL_INFO** structures. Each structure defines a protocol/address_family/socket_type supported by the provider.
dwNumberOfEntries	Contains the number of entries in the *lpProtocolInfoList* array.
lpdwProviderId	Points to a **DWORD** that will receive the locally unique identifier for the newly installed provider.
lpErrno	A pointer to the error code.

Remarks This routine creates the necessary common Winsock 2 configuration information for the specified provider. After this routine completes successfully, the protocol information provided in *lpProtocolInfoList* will be returned by the WSAEnumProtocols API.

Any file installation or service provider specific configuration information must be performed by the provider setup application.

Return Value If no error occurs, **WPUInstallProvider()** returns 0. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

Error Codes ???? ????

See Also **WPUDeinstallProvider(), WSAEnumProtocols()**.

5.2 WPUDeinstallProvider()

Description Removes the specified provider from the system configuration database.

#include <ws2spi.h>

int WSPAPI WPUDeinstallProvider(**DWORD** *dwProviderId*, **int** FAR * *lpErrno*);

dwProviderId	The locally unique identifier of the provider to deinstall. This must be a value previously returned by **WPUInstallProvider()**.
lpErrno	A pointer to the error code.

Remarks This routine removes the common Winsock 2 configuration information for the specified provider. After this routine completes successfully, the protocol information associated with the provider will not be returned by the WSAEnumProtocols API.

Any file removal or service provider specific configuration information removal must be performed by the provider setup application.

Return Value If no error occurs, **WPUDeinstallProvider()** returns 0. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

Error Codes ???? ????

See Also **WPUInstallProvider(), WSAEnumProtocols()**.

Appendix A. Error Codes and Header Files

A.1 Error Codes

The following is a list of possible error codes available in the *lpErrno* parameter of each function, along with their explanations. The error numbers are consistently set across all Winsock-compliant implementations.

Winsock code	Berkeley equivalent	Error	Interpretation
WSAEINTR	EINTR	10004	As in standard C
WSAEBADF	EBADF	10009	As in standard C
WSAEACCES	EACCES	10013	As in standard C
WSAEFAULT	EFAULT	10014	As in standard C
WSAEINVAL	EINVAL	10022	As in standard C
WSAEMFILE	EMFILE	10024	As in standard C
WSAEWOULDBLOCK	EWOULDBLOCK	10035	As in BSD
WSAEINPROGRESS	EINPROGRESS	10036	This error is returned if any Winsock function is called while a blocking function is in progress.
WSAEALREADY	EALREADY	10037	As in BSD
WSAENOTSOCK	ENOTSOCK	10038	As in BSD
WSAEDESTADDRREQ	EDESTADDRREQ	10039	As in BSD
WSAEMSGSIZE	EMSGSIZE	10040	As in BSD
WSAEPROTOTYPE	EPROTOTYPE	10041	As in BSD
WSAENOPROTOOPT	ENOPROTOOPT	10042	As in BSD
WSAEPROTONOSUPPORT	EPROTONOSUPPORT	10043	As in BSD
WSAESOCKTNOSUPPORT	ESOCKTNOSUPPORT	10044	As in BSD
WSAEOPNOTSUPP	EOPNOTSUPP	10045	As in BSD
WSAEPFNOSUPPORT	EPFNOSUPPORT	10046	As in BSD
WSAEAFNOSUPPORT	EAFNOSUPPORT	10047	As in BSD
WSAEADDRINUSE	EADDRINUSE	10048	As in BSD
WSAEADDRNOTAVAIL	EADDRNOTAVAIL	10049	As in BSD
WSAENETDOWN	ENETDOWN	10050	As in BSD. This error may be reported at any time if the Winsock service provider detects an underlying failure.
WSAENETUNREACH	ENETUNREACH	10051	As in BSD
WSAENETRESET	ENETRESET	10052	As in BSD
WSAECONNABORTED	ECONNABORTED	10053	As in BSD
WSAECONNRESET	ECONNRESET	10054	As in BSD
WSAENOBUFS	ENOBUFS	10055	As in BSD
WSAEISCONN	EISCONN	10056	As in BSD
WSAENOTCONN	ENOTCONN	10057	As in BSD
WSAESHUTDOWN	ESHUTDOWN	10058	As in BSD
WSAETOOMANYREFS	ETOOMANYREFS	10059	As in BSD
WSAETIMEDOUT	ETIMEDOUT	10060	As in BSD
WSAECONNREFUSED	ECONNREFUSED	10061	As in BSD
WSAELOOP	ELOOP	10062	As in BSD
WSAENAMETOOLONG	ENAMETOOLONG	10063	As in BSD
WSAEHOSTDOWN	EHOSTDOWN	10064	As in BSD
WSAEHOSTUNREACH	EHOSTUNREACH	10065	As in BSD
WSASYSNOTREADY		10091	Returned by **WSPStartup()** indicating that the network subsystem is unusable.
WSAVERNOTSUPPORTED		10092	Returned by **WSPStartup()** indicating that the Winsock service provider cannot support the Winsock DLL.
WSAHOST_NOT_FOUND	HOST_NOT_FOUND	11001	As in BSD.
WSATRY_AGAIN	TRY_AGAIN	11002	As in BSD
WSANO_RECOVERY	NO_RECOVERY	11003	As in BSD
WSANO_DATA	NO_DATA	11004	As in BSD

The first set of definitions is present to resolve contentions between standard C error codes which may be defined inconsistently between various C compilers.

The second set of definitions provides Winsock versions of regular Berkeley Sockets error codes.

The third set of definitions consists of extended Winsock-specific error codes.

The error numbers are derived from the **ws2spi.h** header file listed in section A.2, and are based on the fact that Winsock error numbers are computed by adding 10000 to the "normal" Berkeley error number.

Note that this table does not include all of the error codes defined in **ws2spi.h**. This is because it includes only errors which might reasonably be returned by a Winsock service provider: **ws2spi.h**, on the other hand, includes a full set of BSD definitions to ensure compatibility with ported software.

A.2 Winsock SPI Header File - ws2spi.h

The **ws2spi.h** header file includes a number of types and definitions from the standard Windows header file **windows.h**. The **windows.h** in the Windows 3.0 SDK (Software Developer's Kit) lacks a `#include` guard, so if you need to include **windows.h** as well as **ws2spi.h**, you should define the symbol _INC_WINDOWS before #including **ws2spi.h**, as follows:

```
#include <windows.h>
#define _INC_WINDOWS
#include <ws2spi.h>
```

Users of the SDK for Windows 3.1 and later need not do this.

A Winsock service provider vendor **MUST NOT** make any modifications to this header file which could impact binary compatibility of Winsock applications. The constant values, function parameters and return codes, and the like must remain consistent across all Winsock service provider vendors.

```
/* WS2SPI.H--definitions to be used with the Winsock service provider.
 *
 * This header file corresponds to version 2.0 of the Winsock SPI specification.
 *
 * This file includes parts which are Copyright (c) 1982-1986 Regents
 * of the University of California.  All rights reserved.  The
 * Berkeley Software License Agreement specifies the terms and
 * conditions for redistribution.
 */

#ifndef _WS2SPI_
#define _WS2SPI_

#ifndef _WS2API_

/*
 * Pull in WINDOWS.H if necessary
 */
#ifndef _INC_WINDOWS
#include <windows.h>
#endif /* _INC_WINDOWS */

/*
 * SPI function linkage.
 */

#define WSPAPI WSPAPI

/*
 * Basic system type definitions, taken from the BSD file sys/types.h.
 */
typedef unsigned char   u_char;
typedef unsigned short  u_short;
typedef unsigned int    u_int;
typedef unsigned long   u_long;

/*
 * The new type to be used in all
 * instances which refer to sockets.
 */
typedef u_int           SOCKET;

/*
 * Select uses arrays of SOCKETs.  These macros manipulate such
 * arrays.  FD_SETSIZE may be defined by the user before including
 * this file, but the default here should be >= 64.
 *
 * CAVEAT IMPLEMENTOR and USER: THESE MACROS AND TYPES MUST BE
 * INCLUDED IN WS2SPI.H EXACTLY AS SHOWN HERE.
 */
#ifndef FD_SETSIZE
#define FD_SETSIZE      64
```

```
#endif /* FD_SETSIZE */

typedef struct fd_set {
        u_short fd_count;               /* how many are SET? */
        SOCKET  fd_array[FD_SETSIZE];   /* an array of SOCKETs */
} fd_set;

#ifdef __cplusplus
extern "C" {
#endif

extern int WSPAPI __WSAFDIsSet(SOCKET, fd_set FAR *);

#ifdef __cplusplus
}
#endif

#define FD_CLR(fd, set) do { \
    u_int __i; \
    for (__i = 0; __i < ((fd_set FAR *)(set))->fd_count ; __i++) { \
        if (((fd_set FAR *)(set))->fd_array[__i] == fd) { \
            while (__i < ((fd_set FAR *)(set))->fd_count-1) { \
                ((fd_set FAR *)(set))->fd_array[__i] = \
                    ((fd_set FAR *)(set))->fd_array[__i+1]; \
                __i++; \
            } \
            ((fd_set FAR *)(set))->fd_count--; \
            break; \
        } \
    } \
} while(0)

#define FD_SET(fd, set) do { \
    if (((fd_set FAR *)(set))->fd_count < FD_SETSIZE) \
        ((fd_set FAR *)(set))->fd_array[((fd_set FAR *)(set))->fd_count++]=fd;\
} while(0)

#define FD_ZERO(set) (((fd_set FAR *)(set))->fd_count=0)

#define FD_ISSET(fd, set) __WSAFDIsSet((SOCKET)fd, (fd_set FAR *)set)

/*
 * Structure used in select() call, taken from the BSD file sys/time.h.
 */
struct timeval {
        long    tv_sec;         /* seconds */
        long    tv_usec;        /* and microseconds */
};

/*
 * Operations on timevals.
 *
 * NB: timercmp does not work for >= or <=.
 */
#define timerisset(tvp)         ((tvp)->tv_sec || (tvp)->tv_usec)
#define timercmp(tvp, uvp, cmp) \
        ((tvp)->tv_sec cmp (uvp)->tv_sec || \
         (tvp)->tv_sec == (uvp)->tv_sec && (tvp)->tv_usec cmp (uvp)->tv_usec)
#define timerclear(tvp)         (tvp)->tv_sec = (tvp)->tv_usec = 0

/*
 * Winsock extension
 */
#define MAKEWORD(low, high) ((WORD)(((BYTE)(low)) | (((WORD)((BYTE)(high))) << 8)))

/*
 * Commands for ioctlsocket(),  taken from the BSD file fcntl.h.
 *
 *
 * Ioctl's have the command encoded in the lower word,
 * and the size of any in or out parameters in the upper
 * word.  The high 2 bits of the upper word are used
 * to encode the in/out status of the parameter; for now
 * we restrict parameters to at most 128 bytes.
 */
#define IOCPARM_MASK    0x7f            /* parameters must be < 128 bytes */
```

```
#define IOC_VOID        0x20000000      /* no parameters */
#define IOC_OUT         0x40000000      /* copy out parameters */
#define IOC_IN          0x80000000      /* copy in parameters */
#define IOC_INOUT       (IOC_IN|IOC_OUT)
                                        /* 0x20000000 distinguishes new &
                                           old ioctl's */
#define _IO(x,y)        (IOC_VOID|(x<<8)|y)

#define _IOR(x,y,t)     (IOC_OUT|(((long)sizeof(t)&IOCPARM_MASK)<<16)|(x<<8)|y)

#define _IOW(x,y,t)     (IOC_IN|(((long)sizeof(t)&IOCPARM_MASK)<<16)|(x<<8)|y)

#define FIONREAD    _IOR('f', 127, u_long) /* get # bytes to read */
#define FIONBIO     _IOW('f', 126, u_long) /* set/clear non-blocking i/o */
#define FIOASYNC    _IOW('f', 125, u_long) /* set/clear async i/o */

/* Socket I/O Controls */
#define SIOCSHIWAT  _IOW('s',  0, u_long)  /* set high watermark */
#define SIOCGHIWAT  _IOR('s',  1, u_long)  /* get high watermark */
#define SIOCSLOWAT  _IOW('s',  2, u_long)  /* set low watermark */
#define SIOCGLOWAT  _IOR('s',  3, u_long)  /* get low watermark */
#define SIOCATMARK  _IOR('s',  7, u_long)  /* at oob mark? */

/*
 * Structures returned by network data base library, taken from the
 * BSD file netdb.h.  All addresses are supplied in host order, and
 * returned in network order (suitable for use in system calls).
 */

struct  hostent {
        char    FAR * h_name;           /* official name of host */
        char    FAR * FAR * h_aliases;  /* alias list */
        short   h_addrtype;             /* host address type */
        short   h_length;               /* length of address */
        char    FAR * FAR * h_addr_list; /* list of addresses */
#define h_addr  h_addr_list[0]          /* address, for backward compat */
};

/*
 * It is assumed here that a network number
 * fits in 32 bits.
 */
struct  netent {
        char    FAR * n_name;           /* official name of net */
        char    FAR * FAR * n_aliases;  /* alias list */
        short   n_addrtype;             /* net address type */
        u_long  n_net;                  /* network # */
};

struct  servent {
        char    FAR * s_name;           /* official service name */
        char    FAR * FAR * s_aliases;  /* alias list */
        short   s_port;                 /* port # */
        char    FAR * s_proto;          /* protocol to use */
};

struct  protoent {
        char    FAR * p_name;           /* official protocol name */
        char    FAR * FAR * p_aliases;  /* alias list */
        short   p_proto;                /* protocol # */
};

/*
 * Constants and structures defined by the internet system,
 * Per RFC 790, September 1981, taken from the BSD file netinet/in.h.
 */

/*
 * Protocols
 */
#define IPPROTO_IP              0               /* dummy for IP */
#define IPPROTO_ICMP            1               /* control message protocol */
#define IPPROTO_GGP             2               /* gateway^2 (deprecated) */
#define IPPROTO_TCP             6               /* tcp */
#define IPPROTO_PUP             12              /* pup */
#define IPPROTO_UDP             17              /* user datagram protocol */
```

```
#define IPPROTO_IDP             22              /* xns idp */
#define IPPROTO_ND              77              /* UNOFFICIAL net disk proto */

#define IPPROTO_RAW             255             /* raw IP packet */
#define IPPROTO_MAX             256

/*
 * Port/socket numbers: network standard functions
 */
#define IPPORT_ECHO             7
#define IPPORT_DISCARD          9
#define IPPORT_SYSTAT           11
#define IPPORT_DAYTIME          13
#define IPPORT_NETSTAT          15
#define IPPORT_FTP              21
#define IPPORT_TELNET           23
#define IPPORT_SMTP             25
#define IPPORT_TIMESERVER       37
#define IPPORT_NAMESERVER       42
#define IPPORT_WHOIS            43
#define IPPORT_MTP              57

/*
 * Port/socket numbers: host specific functions
 */
#define IPPORT_TFTP             69
#define IPPORT_RJE              77
#define IPPORT_FINGER           79
#define IPPORT_TTYLINK          87
#define IPPORT_SUPDUP           95

/*
 * UNIX TCP sockets
 */
#define IPPORT_EXECSERVER       512
#define IPPORT_LOGINSERVER      513
#define IPPORT_CMDSERVER        514
#define IPPORT_EFSSERVER        520

/*
 * UNIX UDP sockets
 */
#define IPPORT_BIFFUDP          512
#define IPPORT_WHOSERVER        513
#define IPPORT_ROUTESERVER      520
                                        /* 520+1 also used */

/*
 * Ports < IPPORT_RESERVED are reserved for
 * privileged processes (e.g. root).
 */
#define IPPORT_RESERVED         1024

/*
 * Link numbers
 */
#define IMPLINK_IP              155
#define IMPLINK_LOWEXPER        156
#define IMPLINK_HIGHEXPER       158

/*
 * Internet address (old style... should be updated)
 */
struct in_addr {
        union {
                struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
                struct { u_short s_w1,s_w2; } S_un_w;
                u_long S_addr;
        } S_un;
#define s_addr  S_un.S_addr
                                /* can be used for most tcp & ip code */
#define s_host  S_un.S_un_b.s_b2
                                /* host on imp */
#define s_net   S_un.S_un_b.s_b1
                                /* network */
#define s_imp   S_un.S_un_w.s_w2
```

```
                                /* imp */
#define s_impno S_un.S_un_b.s_b4
                                /* imp # */
#define s_lh    S_un.S_un_b.s_b3
                                /* logical host */
};

/*
 * Definitions of bits in internet address integers.
 * On subnets, the decomposition of addresses to host and net parts
 * is done according to subnet mask, not the masks here.
 */
#define IN_CLASSA(i)            (((long)(i) & 0x80000000) == 0)
#define IN_CLASSA_NET           0xff000000
#define IN_CLASSA_NSHIFT        24
#define IN_CLASSA_HOST          0x00ffffff
#define IN_CLASSA_MAX           128

#define IN_CLASSB(i)            (((long)(i) & 0xc0000000) == 0x80000000)
#define IN_CLASSB_NET           0xffff0000
#define IN_CLASSB_NSHIFT        16
#define IN_CLASSB_HOST          0x0000ffff
#define IN_CLASSB_MAX           65536

#define IN_CLASSC(i)            (((long)(i) & 0xc0000000) == 0xc0000000)
#define IN_CLASSC_NET           0xffffff00
#define IN_CLASSC_NSHIFT        8
#define IN_CLASSC_HOST          0x000000ff

#define INADDR_ANY              (u_long)0x00000000
#define INADDR_LOOPBACK         0x7f000001
#define INADDR_BROADCAST        (u_long)0xffffffff
#define INADDR_NONE             0xffffffff
/* Winsock extension */
#define ADDR_ANY                INADDR_ANY

/*
 * Socket address, internet style.
 */
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};

#define WSADESCRIPTION_LEN      256
#define WSASYS_STATUS_LEN       128

typedef struct WSAData {
        WORD                    wVersion;
        WORD                    wHighVersion;
        char                    szDescription[WSADESCRIPTION_LEN+1];
        char                    szSystemStatus[WSASYS_STATUS_LEN+1];
        unsigned short          iMaxSockets;
        unsigned short          iMaxUdpDg;
        char FAR *              lpVendorInfo;
} WSADATA;

typedef WSADATA FAR *LPWSADATA;


/*
 * Options for use with [gs]etsockopt at the IP level.
 */
#define IP_OPTIONS      1               /* set/get IP per-packet options */

/*
 * Definitions related to sockets: types, address families, options,
 * taken from the BSD file sys/socket.h.
 */

/*
 * This is used instead of -1, since the
 * SOCKET type is unsigned.
 */
```

```
#define INVALID_SOCKET  (SOCKET)(~0)
#define SOCKET_ERROR            (-1)

/*
 * Types
 */
#define SOCK_STREAM     1               /* stream socket */
#define SOCK_DGRAM      2               /* datagram socket */
#define SOCK_RAW        3               /* raw-protocol interface */
#define SOCK_RDM        4               /* reliably-delivered message */
#define SOCK_SEQPACKET  5               /* sequenced packet stream */

/*
 * Types -- Winsock extensions for socket types with the following convention
 *          SOCK[_REL|_UNREL][_ISOCH][_UNISEND|_UNIRECV][_STREAM|_DGRAM|_DSTREAM]
 */
#define SOCK_REL_STREAM               SOCK_STREAM
#define SOCK_REL_DSTREAM              SOCK_SEQPACKET
#define SOCK_UNREL_DSTREAM            101
#define SOCK_REL_UNISEND_DSTREAM      102
#define SOCK_REL_UNIRECV_DSTREAM      103
#define SOCK_UNREL_UNISEND_DSTREAM    104
#define SOCK_UNREL_UNIRECV_DSTREAM    105
#define SOCK_REL_DGRAM                106
#define SOCK_UNREL_DGRAM              SOCK_DGRAM
#define SOCK_REL_ISOCH_DSTREAM        201
#define SOCK_UNREL_ISOCH_DSTREAM      202
#define SOCK_UNREL_ISOCH_STREAM       203

/*
 * Option flags per-socket.
 */
#define SO_DEBUG        0x0001          /* turn on debugging info recording */
#define SO_ACCEPTCONN   0x0002          /* socket has had listen() */
#define SO_REUSEADDR    0x0004          /* allow local address reuse */
#define SO_KEEPALIVE    0x0008          /* keep connections alive */
#define SO_DONTROUTE    0x0010          /* just use interface addresses */
#define SO_BROADCAST    0x0020          /* permit sending of broadcast msgs */
#define SO_USELOOPBACK  0x0040          /* bypass hardware when possible */
#define SO_LINGER       0x0080          /* linger on close if data present */
#define SO_OOBINLINE    0x0100          /* leave received OOB data in line */

#define SO_DONTLINGER   (int)(~SO_LINGER)

/*
 * Additional options.
 */
#define SO_SNDBUF       0x1001          /* send buffer size */
#define SO_RCVBUF       0x1002          /* receive buffer size */
#define SO_SNDLOWAT     0x1003          /* send low-water mark */
#define SO_RCVLOWAT     0x1004          /* receive low-water mark */
#define SO_SNDTIMEO     0x1005          /* send timeout */
#define SO_RCVTIMEO     0x1006          /* receive timeout */
#define SO_ERROR        0x1007          /* get error status and clear */
#define SO_TYPE         0x1008          /* get socket type */

/*
 * Winsock extension -- options
 */
#define SO_FLOWSPEC       0x2001          /* flow spec for sockets */
#define SO_GROUP_FLOWSPEC 0x2002          /* flow spec for socket groups */
#define SO_GROUP_ID       0x2003          /* sharing the same physical connection */
#define SO_GROUP_PRIORITY 0x2004          /* the relative priority with a group */
#define SO_MAX_DG_SIZE    0x2005          /* maximum datagram size */
#define PVD_CALL_ID       0x3001          /* an opaque Winsock call ID */
#define PVD_CONFIG        0x3002          /* configuration info for service provider */
#define TAPI_DEVICE_ID    0x3003          /* a TAPI line device ID */

/*
 * TCP options.
 */
#define TCP_NODELAY     0x0001

/*
 * Address families.
 */
```

```
#define AF_UNSPEC       0               /* unspecified */
#define AF_UNIX         1               /* local to host (pipes, portals) */
#define AF_INET         2               /* internetwork: UDP, TCP, etc. */
#define AF_IMPLINK      3               /* arpanet imp addresses */
#define AF_PUP          4               /* pup protocols: e.g. BSP */
#define AF_CHAOS        5               /* mit CHAOS protocols */
#define AF_NS           6               /* XEROX NS protocols */
#define AF_ISO          7               /* ISO protocols */
#define AF_OSI          AF_ISO          /* OSI is ISO */
#define AF_ECMA         8               /* european computer manufacturers */
#define AF_DATAKIT      9               /* datakit protocols */
#define AF_CCITT        10              /* CCITT protocols, X.25 etc */
#define AF_SNA          11              /* IBM SNA */
#define AF_DECnet       12              /* DECnet */
#define AF_DLI          13              /* Direct data link interface */
#define AF_LAT          14              /* LAT */
#define AF_HYLINK       15              /* NSC Hyperchannel */
#define AF_APPLETALK    16              /* AppleTalk */
#define AF_NETBIOS      17              /* NetBios-style addresses */
/*
 * Address families -- Winsock extensions.
 */
#define AF_IPX          18              /* IPX/SPX addresses */
#define AF_POTS_IT      19              /* POTS addresses for Intel Transport */
#define AF_ISDNQMUX_IT  20              /* ISDN addresses for Intel Transport */
#define AF_INET_SPE     21              /* Internet (SPE version) */
#define AF_IPX_SPE      22              /* IPX/SPX (SPE version) */
#define AF_NETBIOS_SPE  23              /* NetBios (SPE version) */

#define AF_MAX          24

/*
 * Structure used by kernel to store most
 * addresses.
 */
struct sockaddr {
        u_short sa_family;              /* address family */
        char    sa_data[14];            /* up to 14 bytes of direct address */
};

/*
 * Structure used by kernel to pass protocol
 * information in raw sockets.
 */
struct sockproto {
        u_short sp_family;              /* address family */
        u_short sp_protocol;            /* protocol */
};

/*
 * Protocol families, same as address families for now.
 */
#define PF_UNSPEC       AF_UNSPEC
#define PF_UNIX         AF_UNIX
#define PF_INET         AF_INET
#define PF_IMPLINK      AF_IMPLINK
#define PF_PUP          AF_PUP
#define PF_CHAOS        AF_CHAOS
#define PF_NS           AF_NS
#define PF_ISO          AF_ISO
#define PF_OSI          AF_OSI
#define PF_ECMA         AF_ECMA
#define PF_DATAKIT      AF_DATAKIT
#define PF_CCITT        AF_CCITT
#define PF_SNA          AF_SNA
#define PF_DECnet       AF_DECnet
#define PF_DLI          AF_DLI
#define PF_LAT          AF_LAT
#define PF_HYLINK       AF_HYLINK
#define PF_APPLETALK    AF_APPLETALK
/*
 * Protocol families -- Winsock extension
 */
#define PF_IPX          AF_IPX
#define PF_POTS_IT      AF_POTS_IT
#define PF_ISDNQMUX_IT  AF_ISDNQMUX_IT
```

```
#define PF_INET_SPE     AF_INET_SPE
#define PF_IPX_SPE      AF_IPX_SPE
#define PF_NETBIOS_SPE  AF_NETBIOS_SPE

#define PF_MAX          AF_MAX

/*
 * Structure used for manipulating linger option.
 */
struct  linger {
        u_short l_onoff;                /* option on/off */
        u_short l_linger;               /* linger time */
};

/*
 * Level number for (get/set)sockopt() to apply to socket itself.
 */
#define SOL_SOCKET      0xffff          /* options for socket level */

/*
 * Winsock extension -- level number for (get/set)socketopt() to apply to service provider
 *                  level.
 */
#define SOL_PROVIDER    IPPROTO_TCP     /* options for service provider level */

/*
 * Maximum queue length specifiable by listen.
 */
#define SOMAXCONN       5

#define MSG_OOB         0x1             /* process out-of-band data */
#define MSG_PEEK        0x2             /* peek at incoming message */
#define MSG_DONTROUTE   0x4             /* send without using routing tables */
#define MSG_INTERRUPT   0x8             /* interrupt-time send or recv */

#define MSG_MAXIOVLEN   0x10

/*
 * Define constant based on rfc883, used by gethostbyxxxx() calls.
 */
#define MAXGETHOSTSTRUCT        1024

/*
 * Winsock extension -- WSABUF and QOS struct
 */
typedef struct _WSABUF {
        int         len;     /* the length of the buffer */
        char FAR *  buf;     /* the pointer to the buffer */
} WSABUF, FAR * LPWSABUF;

typedef enum _GUARANTEE {
        GuaranteedService,
        BestEffortService
} GUARANTEE;

typedef struct _flowparams {
        int64           AverageBandwidth;      /* in bytes/sec */
        int64           PeakBandwidth;         /* in bytes/sec */
        int64           BurstLength;           /* in microseconds */
        int64           Latency;               /* in microseconds */
        int64           DelayVariation;        /* in microseconds */
        GUARANTEE       LevelOfGuarantee;      /* guaranteed or best effort */
        int32           CostOfCall             /* reserved for future; must be zero */
        int32           ProviderId;            /* provider identifier */
        int32           SizePSP;               /* length of provider specific parameters */
        UCHAR           ProviderSpecificParams[1]; /* provider specific parameters */
} FLOWPARAMS;

typedef struct _QOS {
        FLOWPARAMS      ForwardFP;             /* caller (initiator) to callee */
        FLOWPARAMS      BackwardFP;            /* callee to caller */
} QOS, FAR * LPQOS;

/*
 * Winsock extension -- WSANETWORKEVENT
 */
```

```
typedef struct _WSANETWORKEVENT {
        BOOL    Fired;
        int     ErrorCode;
} WSANETWORKEVENT, FAR * LPWSANETWORKEVENT;

/*
 * Winsock extension -- manifest constants for the return value of the condition function
 */
#define CF_ACCEPT       0x0000
#define CF_REJECT       0x0001
#define CF_DEFER        0x0002

/*
 * Winsock extension -- manifest constants for shutdown()
 */
#define SD_RECEIVE      0x00
#define SD_SEND         0x01
#define SD_BOTH         0x02

/*
 * Winsock extension -- data type and manifest constants for socket groups
 */
typedef unsigned int            GROUP;

#define SG_UNCONSTRAINED_GROUP   0x01
#define SG_CONSTRAINED_GROUP     0x02

/*
 * Define flags to be used with the WSPSelect() call.
 */
#define FD_READ         0x01L
#define FD_WRITE        0x02L
#define FD_OOB          0x04L
#define FD_ACCEPT       0x08L
#define FD_CONNECT      0x10L
#define FD_CLOSE        0x20L
/*
 * Winsock extension -- new flags for WSPSelect()
 */
#define FD_QOS          0x40L
#define FD_GROUP_QOS    0x80L

/*
 * All Winsock error constants are biased by WSABASEERR from
 * the "normal"
 */
#define WSABASEERR              10000
/*
 * Winsock definitions of regular Microsoft C error constants
 */
#define WSAEINTR                (WSABASEERR+4)
#define WSAEBADF                (WSABASEERR+9)
#define WSAEACCES               (WSABASEERR+13)
#define WSAEFAULT               (WSABASEERR+14)
#define WSAEINVAL               (WSABASEERR+22)
#define WSAEMFILE               (WSABASEERR+24)

/*
 * Winsock definitions of regular Berkeley error constants
 */
#define WSAEWOULDBLOCK          (WSABASEERR+35)
#define WSAEINPROGRESS          (WSABASEERR+36)
#define WSAEALREADY             (WSABASEERR+37)
#define WSAENOTSOCK             (WSABASEERR+38)
#define WSAEDESTADDRREQ         (WSABASEERR+39)
#define WSAEMSGSIZE             (WSABASEERR+40)
#define WSAEPROTOTYPE           (WSABASEERR+41)
#define WSAENOPROTOOPT          (WSABASEERR+42)
#define WSAEPROTONOSUPPORT      (WSABASEERR+43)
#define WSAESOCKTNOSUPPORT      (WSABASEERR+44)
#define WSAEOPNOTSUPP           (WSABASEERR+45)
#define WSAEPFNOSUPPORT         (WSABASEERR+46)
#define WSAEAFNOSUPPORT         (WSABASEERR+47)
#define WSAEADDRINUSE           (WSABASEERR+48)
#define WSAEADDRNOTAVAIL        (WSABASEERR+49)
#define WSAENETDOWN             (WSABASEERR+50)
```

```
#define WSAENETUNREACH          (WSABASEERR+51)
#define WSAENETRESET            (WSABASEERR+52)
#define WSAECONNABORTED         (WSABASEERR+53)
#define WSAECONNRESET           (WSABASEERR+54)
#define WSAENOBUFS              (WSABASEERR+55)
#define WSAEISCONN              (WSABASEERR+56)
#define WSAENOTCONN             (WSABASEERR+57)
#define WSAESHUTDOWN            (WSABASEERR+58)
#define WSAETOOMANYREFS         (WSABASEERR+59)
#define WSAETIMEDOUT            (WSABASEERR+60)
#define WSAECONNREFUSED         (WSABASEERR+61)
#define WSAELOOP                (WSABASEERR+62)
#define WSAENAMETOOLONG         (WSABASEERR+63)
#define WSAEHOSTDOWN            (WSABASEERR+64)
#define WSAEHOSTUNREACH         (WSABASEERR+65)
#define WSAENOTEMPTY            (WSABASEERR+66)
#define WSAEPROCLIM             (WSABASEERR+67)
#define WSAEUSERS               (WSABASEERR+68)
#define WSAEDQUOT               (WSABASEERR+69)
#define WSAESTALE               (WSABASEERR+70)
#define WSAEREMOTE              (WSABASEERR+71)

/*
 * Extended Winsock error constant definitions
 */
#define WSASYSNOTREADY          (WSABASEERR+91)
#define WSAVERNOTSUPPORTED      (WSABASEERR+92)
#define WSANOTINITIALISED       (WSABASEERR+93)

/*
 * Error return codes from gethostbyname() and gethostbyaddr()
 * (when using the resolver). Note that these errors are
 * retrieved via WSAGetLastError() and must therefore follow
 * the rules for avoiding clashes with error numbers from
 * specific implementations or language run-time systems.
 * For this reason the codes are based at WSABASEERR+1001.
 * Note also that [WSA]NO_ADDRESS is defined only for
 * compatibility purposes.
 */

#define h_errno         WSAGetLastError()

/* Authoritative Answer: Host not found */
#define WSAHOST_NOT_FOUND       (WSABASEERR+1001)
#define HOST_NOT_FOUND          WSAHOST_NOT_FOUND

/* Non-Authoritative: Host not found, or SERVERFAIL */
#define WSATRY_AGAIN            (WSABASEERR+1002)
#define TRY_AGAIN               WSATRY_AGAIN

/* Non recoverable errors, FORMERR, REFUSED, NOTIMP */
#define WSANO_RECOVERY          (WSABASEERR+1003)
#define NO_RECOVERY             WSANO_RECOVERY

/* Valid name, no data record of requested type */
#define WSANO_DATA              (WSABASEERR+1004)
#define NO_DATA                 WSANO_DATA

/* no address, look for MX record */
#define WSANO_ADDRESS           WSANO_DATA
#define NO_ADDRESS              WSANO_ADDRESS

/*
 * Winsock errors redefined as regular Berkeley error constants
 */
#define EWOULDBLOCK             WSAEWOULDBLOCK
#define EINPROGRESS             WSAEINPROGRESS
#define EALREADY                WSAEALREADY
#define ENOTSOCK                WSAENOTSOCK
#define EDESTADDRREQ            WSAEDESTADDRREQ
#define EMSGSIZE                WSAEMSGSIZE
#define EPROTOTYPE              WSAEPROTOTYPE
#define ENOPROTOOPT             WSAENOPROTOOPT
#define EPROTONOSUPPORT         WSAEPROTONOSUPPORT
#define ESOCKTNOSUPPORT         WSAESOCKTNOSUPPORT
#define EOPNOTSUPP              WSAEOPNOTSUPP
```

```
#define EPFNOSUPPORT            WSAEPFNOSUPPORT
#define EAFNOSUPPORT            WSAEAFNOSUPPORT
#define EADDRINUSE              WSAEADDRINUSE
#define EADDRNOTAVAIL           WSAEADDRNOTAVAIL
#define ENETDOWN                WSAENETDOWN
#define ENETUNREACH             WSAENETUNREACH
#define ENETRESET               WSAENETRESET
#define ECONNABORTED            WSAECONNABORTED
#define ECONNRESET              WSAECONNRESET
#define ENOBUFS                 WSAENOBUFS
#define EISCONN                 WSAEISCONN
#define ENOTCONN                WSAENOTCONN
#define ESHUTDOWN               WSAESHUTDOWN
#define ETOOMANYREFS            WSAETOOMANYREFS
#define ETIMEDOUT               WSAETIMEDOUT
#define ECONNREFUSED            WSAECONNREFUSED
#define ELOOP                   WSAELOOP
#define ENAMETOOLONG            WSAENAMETOOLONG
#define EHOSTDOWN               WSAEHOSTDOWN
#define EHOSTUNREACH            WSAEHOSTUNREACH
#define ENOTEMPTY               WSAENOTEMPTY
#define EPROCLIM                WSAEPROCLIM
#define EUSERS                  WSAEUSERS
#define EDQUOT                  WSAEDQUOT
#define ESTALE                  WSAESTALE
#define EREMOTE                 WSAEREMOTE

#endif   /* _WS2API_ */

/*
 *  Winsock SPI socket function prototypes
 */

#ifdef __cplusplus
extern "C" {
#endif

typedef BOOL ( WSACALLBACK * LPBLOCKINGPROC )( VOID );

typedef VOID ( WSACALLBACK * LPCLEANUPPROC )( int ErrorCode,
                                              DWORD dwCallbackData );

typedef int ( WSACALLBACK * LPCONDITIONPROC )( LPWSABUF lpCallerId,
                                               LPWSABUF lpCallerData,
                                               LPWSABUF lpCalleeId,
                                               LPWSABUF lpCalleeData,
                                               GROUP FAR * g,
                                               DWORD dwCallbackData );

typedef VOID ( WSACALLBACK LPSELECTPROC )( SOCKET s,
                                           long lEvent,
                                           int ErrorCode,
                                           DWORD dwCallbackData );

typedef VOID ( FAR * LPWSAUSERAPC )( DWORD dwContext );

int WSPAPI WSPBind( SOCKET s,
                    const struct sockaddr FAR *name,
                    int namelen,
                    int FAR * lpErrno );

int WSPAPI WSPCloseSocket( SOCKET s,
                           int FAR * lpErrno );

int WSPAPI WSPGetPeerName( SOCKET s,
                           struct sockaddr FAR *name,
                           int FAR * namelen,
                           int FAR * lpErrno );

int WSPAPI WSPGetSockName( SOCKET s,
                           struct sockaddr FAR *name,
                           int FAR * namelen,
                           int FAR * lpErrno );

int WSPAPI WSPGetSockOpt( SOCKET s,
                          int level,
```

```
                          int optname,
                          char FAR * optval,
                          int FAR *optlen,
                          int FAR * lpErrno );

int WSPAPI WSPIoctlSocket( SOCKET s,
                           long cmd,
                           u_long FAR *argp,
                           int FAR * lpErrno );

int WSPAPI WSPListen( SOCKET s,
                      int backlog,
                      int FAR * lpErrno );

int WSPAPI WSPSelect( int nfds,
                      fd_set FAR *readfds,
                      fd_set FAR *writefds,
                      fd_set FAR *exceptfds,
                      const struct timeval FAR *timeout,
                      int FAR * lpErrno );

int WSPAPI WSPSetSockOpt( SOCKET s,
                          int level,
                          int optname,
                          const char FAR * optval,
                          int optlen,
                          int FAR * lpErrno );

int WSPAPI WSPShutdown( SOCKET s,
                        int how,
                        int FAR * lpErrno );

/* Microsoft Windows Extension function prototypes */

SOCKET WSPAPI WSPAccept( SOCKET s,
                         struct sockaddr FAR *addr,
                         int FAR *addrlen,
                         LPCONDITIONPROC lpfnCondition,
                         DWORD dwCallbackData,
                         int FAR * lpErrno );

int WSPAPI WSPAsyncSelect32( SOCKET s,
                             HWND hWnd,
                             unsigned int wMsg,
                             long lEvnet,
                             int FAR * lpErrno );

int WSPAPI WSPCallbackSelect16( SOCKET s,
                                LPSELECTPROC lpfnCallback,
                                DWORD dwCallbackData,
                                long lEvent,
                                int FAR * lpErrno );

int WSPAPI WSPCancelBlockingCall32( int FAR * lpErrno );

int WSPAPI WSPCleanup( LPCLEANUPPROC lpfnCallback,
                       DWORD dwCallbackData,
                       int FAR * lpErrno );

int WSPAPI WSPConnect( SOCKET s,
                       const struct sockaddr FAR *name,
                       int namelen,
                       LPWSABUF lpCallerData,
                       LPWSABUF lpCalleeData,
                       GROUP g,
                       LPQOS lpSFlowspec,
                       LPQOS lpGFlowspec,
                       int FAR * lpErrno );

int WSPAPI WSPEnumNetworkEvents( SOCKET s,
                                 WSAEVNET hEventObject,
                                 LPWSANETWORKEVENT lpNetworkEvents,
                                 LPINT lpiCount,
                                 int FAR * lpErrno  );

int WSPAPI WSPEventSelect( SOCKET s,
```

```
                          WSAEVENT hEventObject,
                          long lNetworkEvents,
                          int FAR * lpErrno  );

BOOL WSPAPI WSPIsBlocking32( VOID );

int WSPAPI WSPRecv( SOCKET s,
                    LPVOID lpBuffer,
                    DWORD nNumberOfBytesToRecv,
                    LPDWORD lpNumberOfBytesRecvd,
                    LPINT lpFlags,
                    LPWSAOVERLAPPED lpOverlapped,
                    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
                    int FAR * lpErrno );

int WSPAPI WSPRecvFrom( SOCKET s,
                        LPVOID lpBuffer,
                        DWORD nNumberOfBytesToRecv,
                        LPDWORD lpNumberOfBytesRecvd,
                        LPINT lpFlags,
                        LPVOID lpFrom,
                        LPINT lpFromlen,
                        LPWSAOVERLAPPED lpOverlapped,
                        LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
                        int FAR * lpErrno );

int WSPAPI WSPSend( SOCKET s,
                    LPVOID lpBuffer,
                    DWORD nNumberOfBytesToSend,
                    LPDWORD lpNumberOfBytesSent,
                    int nFlags,
                    LPWSAOVERLAPPED lpOverlapped,
                    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
                    int FAR * lpErrno );

int WSPAPI WSPSendTo( SOCKET s,
                      LPVOID lpBuffer,
                      DWORD nNumberOfBytesToSend,
                      LPDWORD lpNumberOfBytesSent,
                      int nFlags,
                      LPVOID lpTo,
                      int nTolen,
                      LPWSAOVERLAPPED lpOverlapped,
                      LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
                      int FAR * lpErrno );

LPBLOCKINGPROC WSPAPI WSPSetBlockingHook32( LPBLOCKINGPROC lpBlockFunc,
                                            int FAR * lpErrno );

SOCKET WSPAPI WSPSocket( int af,
                         int type,
                         int protocol,
                         DWORD dwProviderId,
                         int nFlags,
                         int FAR * lpErrno );

int WSPAPI WSPStartup( WORD wVersionRequired,
                       LPWSADATA lpWSAData );

int WSPAPI WSPUnhookBlockingHook32( int FAR * lpErrno );

/* Upcalls from service providers to the Winsock DLL */

SOCKETWSPAPI WPUCreateSocketHandle( DWORD dwProviderId,
                                    LPVOID lpContext,
                                    int FAR * lpErrno );

int WSPAPI WPUCloseSocketHandle( SOCKET s,
                                 int FAR * lpErrno );

int WSPAPI WPUQuerySocketHandleContext( SOCKET s,
                                        LPVOID FAR * lpContext,
                                        int FAR * lpErrno );

int WSPAPI WPUSetSocketHandleContext( SOCKET s,
                                      LPVOID lpContext,
```

```
                                    int FAR * lpErrno );

int WSPAPI WPUQueueUserAPC32( DWORD dwThreadId,
                              LPWSAUSERAPC lpfnUserApc,
                              DWORD dwContext,
                              int FAR * lpErrno );

DWORD WSPAPI WPUGetCurrentThreadId32( VOID );

#ifdef __cplusplus
}
#endif

#ifndef _WS2API_

/* Microsoft Windows Extended data types */
typedef struct sockaddr SOCKADDR;
typedef struct sockaddr *PSOCKADDR;
typedef struct sockaddr FAR *LPSOCKADDR;

typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr_in *PSOCKADDR_IN;
typedef struct sockaddr_in FAR *LPSOCKADDR_IN;

typedef struct linger LINGER;
typedef struct linger *PLINGER;
typedef struct linger FAR *LPLINGER;

typedef struct in_addr IN_ADDR;
typedef struct in_addr *PIN_ADDR;
typedef struct in_addr FAR *LPIN_ADDR;

typedef struct fd_set FD_SET;
typedef struct fd_set *PFD_SET;
typedef struct fd_set FAR *LPFD_SET;

typedef struct hostent HOSTENT;
typedef struct hostent *PHOSTENT;
typedef struct hostent FAR *LPHOSTENT;

typedef struct servent SERVENT;
typedef struct servent *PSERVENT;
typedef struct servent FAR *LPSERVENT;

typedef struct protoent PROTOENT;
typedef struct protoent *PPROTOENT;
typedef struct protoent FAR *LPPROTOENT;

typedef struct timeval TIMEVAL;
typedef struct timeval *PTIMEVAL;
typedef struct timeval FAR *LPTIMEVAL;

/*
 * Windows message parameter composition and decomposition
 * macros.
 *
 * WSAMAKEASYNCREPLY is intended for use by Winsock
 * when constructing the response to a WSAAsyncGetXByY() routine.
 */
#define WSAMAKEASYNCREPLY(buflen,error)     MAKELONG(buflen,error)
/*
 * WSAMAKESELECTREPLY is intended for use by Winsock
 * when constructing the response to WSAAsyncSelect().
 */
#define WSAMAKESELECTREPLY(event,error)     MAKELONG(event,error)
/*
 * WSAGETASYNCBUFLEN is intended for use by the Winsock application
 * to extract the buffer length from the lParam in the response
 * to a WSAGetXByY().
 */
#define WSAGETASYNCBUFLEN(lParam)           LOWORD(lParam)
/*
 * WSAGETASYNCERROR is intended for use by the Winsock application
 * to extract the error code from the lParam in the response
 * to a WSAGetXByY().
 */
```

```
#define WSAGETASYNCERROR(lParam)            HIWORD(lParam)
/*
 * WSAGETSELECTEVENT is intended for use by the Winsock application
 * to extract the event code from the lParam in the response
 * to a WSAAsyncSelect().
 */
#define WSAGETSELECTEVENT(lParam)           LOWORD(lParam)
/*
 * WSAGETSELECTERROR is intended for use by the Winsock application
 * to extract the error code from the lParam in the response
 * to a WSAAsyncSelect().
 */
#define WSAGETSELECTERROR(lParam)           HIWORD(lParam)

#endif  /* _WS2API_ */

#endif  /* _WS2SPI_ */
```

Appendix B. Notes for Winsock Service Providers

B.1 Introduction

A Winsock service provider must implement ALL of the applicable functionality described in the Winsock SPI documentation.

Certain Winsock SPIs documented above have special notes for Winsock service provider implementors. A Winsock service provider should pay special attention to conforming to the Winsock SPI as documented. The Special Notes are provided for assistance and clarification.

B.2 Winsock SPI Run Time Components

The run time component provided by each Winsock supplier is:

Component	Description
Installation Program	The Winsock service provider installation program
Service Provider DLL	The Winsock service provider implementation DLL

B.3 Error Codes

In order to avoid conflict between various compiler environments Winsock service providers MUST return the error codes listed in the SPI specification, using the manifest constants beginning with "WSA". The Berkeley-compatible error code definitions are provided solely for compatibility purposes for applications which are being ported from other platforms.

Appendix C. Outstanding Issues

1. Does the 32-bit Windows Sockets 2 implementation live in WSOCK32.DLL or WSOCK232.DLL?
2. Do the 16- and 32-bit providers use the same header file (ws2spi.h)?
3. Should SPI functions that take socket descriptors also take a context value? For providers that use “real” system handles, this could always be NULL.
4. How does a DLL-based protocol support graceful socket close if an app immediately exits after calling closesocket()? The provider & protocol DLLs will be detached from the process (whose address space will go away entirely), yet socket state needs to “linger” for the graceful close.
5. Does the Winsock DLL close all sockets before calling WSPCleanup(), or is WSPCleanup() responsible for closing all open sockets?