

WINSOCK 2

WINDOWS SOCKET: STORY PART 3

Program examples if any, compiled using Visual C++ .Net (Visual studio .Net 2003). It is low-level programming and the .Net used is **Unmanaged** (/clr is not set: **Project** menu → **your_project_name Properties...** sub menu → **Configuration Properties** folder → **General** subfolder → **Used Managed Extension** setting set to **No**). This also applied to other program examples in other Modules of tenouk.com Tutorial that mentioned “compiled using Visual C++ .Net”. Other settings are default. Machine’s OS is standalone Windows Xp Pro SP2 except whenever mentioned. Beware the codes that span more than one line. Program examples have been tested for Non Destructive Test. All information compiled for Win 2000 (NT5.0) above and...

1. The program examples were dumped at [Winsock program examples](#).
2. Related functions, structures and macros used in the program examples have been dumped at [Winsock function & structure 1](#) and [Winsock function & structure 2](#).
3. Other related and required information (if any) not available in no. 2 can be found at [MSDN online](#).

Abilities

- Able to understand Winsock implementation and operations through the APIs and program examples.
- Able to gather, understand and use the Winsock [functions](#), [structures](#) and [macros](#) in your programs.
- Able to build programs that use the Winsock APIs.

Socket Connections on Connection-Oriented Protocols

The following paragraphs describe the semantics applicable to socket connections over connection-oriented protocols such as Transmission Control Protocol (TCP).

Binding to a Local Address

Before a socket can be used to set up a connection or receive a connection request, it needs to be bound to a local address. This could be done explicitly by calling `WSPBind()`, or implicitly by `WSPConnect()` if the socket is unbound when this function is called.

Protocol Basics: Listen, Connect, Accept

The basic operations involved with establishing a socket connection can be most conveniently explained in terms of the client-server paradigm. The server side will first create a socket, bind it to a well known local address (so that the client can find it), and put the socket in listening mode, through `WSPListen()`, in order

to prepare for any incoming connection requests and to specify the length of the connection backlog queue. Service providers hold pending connection requests in a backlog queue until they are acted upon by the server or are withdrawn (due to a time-out) by the client. A service provider may silently ignore requests to extend the size of the backlog queue beyond a provider-defined upper limit. At this point, if a blocking socket is being used, the server side may immediately call `WSPAccept()` which will block until a connection request is pending. Conversely, the server may also use one of the network event indication mechanisms discussed previously to arrange for notification of incoming connection requests. Depending on the notification mechanism selected, the provider will either issue a Windows message or signal an event object when connection requests arrive.

The client side will create an appropriate socket, and initiate the connection by calling `WSPConnect()`, specifying the known server address. Clients usually do not perform an explicit `bind()` operation prior to initiating a connection, allowing the service provider to perform an implicit bind on their behalf. If the socket is in blocking mode, `WSPConnect()` will block until the server has received and acted upon the connection request (or until a time-out occurs). Otherwise, the client should use one of the network event indication mechanisms discussed previously to arrange for notification of a new connection being established. Depending on the notification mechanism selected, the provider will indicate this either through a Windows message or by signaling an event object.

When the server side invokes `WSPAccept()`, the service provider calls the application-supplied condition function, using function parameters to pass into the server information from the top entry in the connection request backlog queue. This information includes such things as address of connecting host, any available user data, and any available QOS information. Using this information the server's condition function determines whether to accept the request, reject the request, or defer making a decision until later. This decision is indicated through the return value of the condition function. If the server decides to accept the request, the provider must create a new socket with all of the same attributes and event registrations as the listening socket. The original socket remains in the listening state so that subsequent connection requests can be received. Through output parameters of the condition function, the server may also supply any response user data and assign QOS parameters (assuming that these operations are supported by the service provider).

Determining Local and Remote Names

`WSPGetSockName()` is used to retrieve the local address for bound sockets. This is especially useful when a `WSPConnect()` call has been made without doing a `WSPBind()` first; `WSPGetSockName()` provides the only means to determine the local association which has been set implicitly by the provider. After a connection has been set up, `WSPGetPeerName()` can be used to determine the address of the peer to which a socket is connected.

Enhanced Functionality at Connect Time

Windows Sockets 2 offers an expanded set of operations that can occur coincident to establishing a socket connection. The service provider requirements for implementing these features are described below.

Conditional Acceptance

As described previously, `WSPAccept()` invokes a client-supplied condition function that uses input parameters to supply information about the pending connection request. This information can be used by the client to accept or reject a connection request based on caller information such as caller identifier, Quality-of-Service (QOS), etc. If the condition function returns `CF_ACCEPT`, a new socket is created with the same properties as the listening socket, and a handle to the new socket is returned. If the condition function returns `CF_REJECT`, the connection request should be rejected. If the condition function returns `CF_DEFER`, the accept/reject decision cannot be made immediately, and the service provider must leave the connection request on the backlog queue. The client must call `WSPAccept()` again, when it is ready to make a decision, and arrange for the condition function to return either `CF_ACCEPT` or `CF_REJECT`. While a deferred connection request is at the top of the backlog queue, the service provider does not issue any further indications for pending connection requests.

Exchanging User Data at Connect Time

Some protocols allow a small amount of user data to be exchanged at connect time. If such data has been received from the connecting host, it is placed in a service provider buffer, and a pointer to this buffer along with a length value are supplied to the Winsock SPI client through input parameters to the `WSPAccept()` condition function. If the Winsock SPI client has response data to return to the connecting host, it may copy this into a buffer that is supplied by the service provider. A pointer to this buffer and an integer indicating buffer size are also supplied as condition function input parameters (if supported by the protocol).

Establishing Socket Groups

All use of socket groups is reserved.

Connection Shutdown

The following describes operations incident to shutting down an established socket connection.

Initiating Shutdown Sequence

A socket connection can be taken down in one of several ways. `WSPShutdown()` (with `how` equal to `SD_SEND` or `SD_BOTH`), and `WSPSendDisconnect()` may be used to initiate a graceful connection shutdown. `WSPCloseSocket()` can be used

to initiate either a **graceful** or **abortive shutdown**, depending on the linger options invoked by the closing a socket.

Indicating Remote Shutdown

Service providers indicate connection teardown that is initiated by the remote party through the FD_CLOSE network event. Graceful shutdown is also be indicated through WSPRecv() when the number of bytes read is 0 for byte-stream protocols, or through a return error code of WSAEDISCON for message-oriented protocols. In any case, a WSPRecv() return error code of WSAECONNRESET indicates an abortive shutdown.

Exchanging User Data at Shutdown Time

At connection teardown time, it is also possible (for protocols that support this) to exchange user data between the endpoints. The end that initiates the teardown can call WSPSendDisconnect() to indicate that no more data is to be sent and cause the connection teardown sequence to be initiated. For certain protocols, part of this teardown sequence is the delivery of disconnect data from the teardown initiator. After receiving notice that the remote end has initiated the teardown sequence (typically through the FD_CLOSE indication), the WSPRecvDisconnect() function may be called to receive the disconnect data (if any).

To illustrate how disconnect data might be used, consider the following scenario. The client half of a client/server application is responsible for terminating a socket connection. Coincident with the termination it provides (through disconnect data) the total number of transactions it processed with the server. The server in turn responds with the cumulative grand total of transactions that it has processed with all clients. The sequence of calls and indications might occur as shown in the following table.

Client side	Server side
(1) Invokes WSPSendDisconnect() to conclude session and supply transaction total.	(2) Gets FD_CLOSE, or WSPRecv() with a return value of zero or WSAEDISCON indicating graceful shutdown in progress.
	(3) Invokes WSPRecvDisconnect() to get client's transaction total.
	(4) Computes cumulative grand total of all transactions.
	(5) Invokes

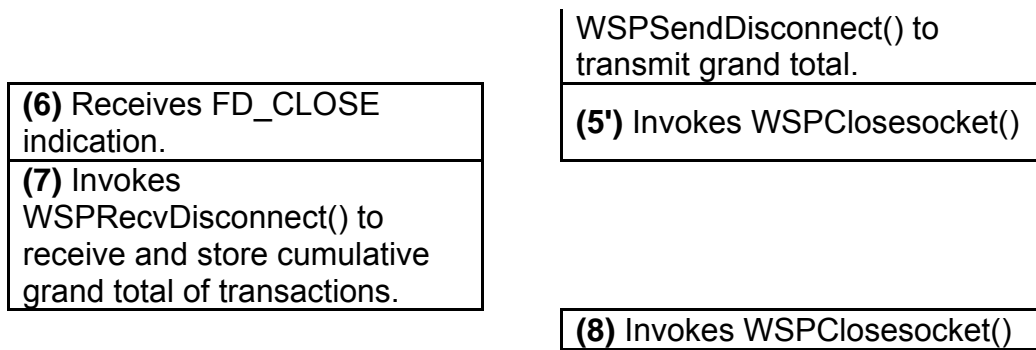


Figure 1

Step **(5')** must follow step **(5)**, but has no timing relationship with steps **(6)**, **(7)**, or **(8)**.

What Else?

IrDA

Infrared Data Association, a multifaceted term commonly referred to as IrDA, can be described as:

- A protocol suite designed to provide wireless, walk-up, line-of-sight connectivity between devices.
- The semi-transparent red window found on many laptops and some desktop computers.
- An organization that creates, promotes, and standardizes IrDA technology (go to irda.org for more information on the IrDA organization).

Microsoft® provides supports IrDA on many of its Windows® versions: Windows XP, Windows 2000, Windows Me, Windows 98, and Windows CE. IrDA is comprised of numerous core protocols and services; Windows provides more thorough support for certain core protocols and services, based on factors including inherent performance characteristics of certain protocols, serialization of data, and programmability. Get more IrDA information in [MSDN](#).

IrDA Programming with Windows Sockets

The recommended approach to programming IrDA applications is through the Windows Sockets API. Legacy versions of Microsoft Windows used the serialized **IrCOMM** programming method, which is not supported in Windows Server 2003, Windows XP or Windows 2000, nor is it the recommended approach. Note that an IrCOMM kernel-mode driver is provided in Windows XP and Windows 2000 to facilitate IrDA-enabled mobile phones using PC-initiated IrCOMM connections. The various Windows operating systems that support IrDA have specific requirements for programming IrDA applications.

IrDA Core Protocols and Services

IrDA on Windows provides core services similar to those exposed by Transmission Control Protocol (the TCP part of TCP/IP). Like TCP, applications on different devices can open multiple, reliable IrDA connections to send and receive data. Client IrDA applications connect to a server application by specifying a device address (similar to a TCP host IP address), and further specify their connection using an application address (similar to a TCP port). The protocols and services exposed by IrDA are explained in the following sections.

IrDA and Windows Sockets Reference

Full descriptions for IrDA functions or structures can be found in the Windows Sockets section of the [IrDA Platform SDK](#).

Bluetooth

Bluetooth is an industry-standard protocol that enables wireless connectivity for a multitude of devices: computers, mobile phones, handheld devices, and many others. Bluetooth touts the following strengths and capabilities:

- A low-cost, low-power consumption wireless protocol with industry-standard support and worldwide acceptance.
- A defined and familiar programming interface, enabling developers to leverage existing network programming knowledge for quick Bluetooth application porting or development.
- An official Web site and an industry-wide cooperative organization that explains, promotes, and standardizes Bluetooth technology (see bluetooth.com for more information).

Microsoft® provides support for Bluetooth on Microsoft® Windows® XP SP1 or later and Windows® CE. For more information about Windows® CE programming, consult the Windows® CE SDK. Bluetooth on Windows provides core services similar to those exposed by Transmission Control Protocol (the TCP part of TCP/IP). Like many networking protocols and services, Bluetooth connectivity and data transfer are programmed through Windows Sockets function calls, using common sockets programming techniques and specific Bluetooth extensions. However, since significant differences exist between a wired, fixed network and a wireless ad-hoc network, Bluetooth provides extensions such as service/device discovery and notification that enable applications to operate properly in the wireless environment. These extensions also pave the way for simple porting to similar technologies, such as IrDA, or future wireless transports.

Extensible Authentication Protocol

Extensible Authentication Protocol (EAP) is a standard supported by many Microsoft networking components. EAP is crucial for protecting the security of

wireless (802.1x) LANs, wired LANs and dial-up and Virtual Private Networks (VPNs). The following components support the EAP protocol:

- Microsoft Remote Access Clients and Servers for both dial-up and VPN (PPTP and L2TP/IPSec) - implemented as an extension to PPP. For more information, see [RFC 2284](#).
- Microsoft IEEE 802.1x compatible wireless and wired LAN clients - implemented as defined in the IEEE 802.1x draft standard.
- Microsoft RADIUS server, called Internet Authentication Service (IAS) - implemented as defined in [RFC 2058](#).

All of the above components support this extensible architecture through the Windows Platform Software Development Kit (SDK), making it possible to integrate third-party authentication modules. This extensible mechanism can be used to support token cards, Kerberos, Public Key, and S/Key authentication protocols. In addition to EAP, the wireless (802.1x) implementation and RADIUS server also support an emerging standard called **Protected EAP (PEAP)**. PEAP provides several key services to other EAP authentication protocols, including the following:

- PEAP encrypts EAP packets of other EAP authentication protocols using Transport Layer Security (TLS), a Secure Socket Layer (SSL) based technology. For more information, see [RFC 2716](#).
- PEAP authenticates the server-side using a Server's Certificate, with Remote Authentication Dial In User Service (RADIUS) server.
- PEAP offers fast re-authentication capability that supports efficient roaming between wireless devices.
- PEAP manages the fragmentation and re-assembly of EAP packets.
- PEAP generates keys used for encrypting wireless traffic.

PEAP makes it much easier to write an EAP protocol because the programmer does not have to address these issues. Get more EAP/PEAP information in [MSDN](#).