WINSOCK 2
WINDOWS SOCKET: STORY PART 2

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 more on TCP/IP.
- 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.

Network Location Awareness Service Provider (NLA)

Personal computers running Microsoft Windows often have numerous network connections, such as multiple network interface cards (NIC) connected to different networks, or a physical network connection and a dial-up connection. Windows Sockets has been capable of enumerating available network interfaces for some time, but certain critical information about network connections - such as the logical network to which a Windows computer is attached, or whether multiple interfaces are connected to the same network -was previously unavailable. The **Network Location Awareness** service provider, commonly referred to as NLA, enables Windows Sockets 2 applications to identify the logical network to which a Windows computer is attached. In addition, NLA enables Windows Sockets applications to identify to which physical network interface a given application has saved specific information. NLA is implemented as a generic Windows Sockets 2 Name Resolution service provider.

About the Winsock SPI

Winsock provides a Service Provider Interface for creating Winsock services, commonly referred to as the Winsock SPI. Two types of service providers exist: **transport providers** and **namespace providers**. Examples of transport providers include protocol stacks such as TCP/IP or IPX/SPX, while an example of a namespace provider would be an interface to the Internet's Domain Naming System (DNS). Separate sections of the service provider interface specification apply to each type of service provider. Transport and namespace service providers must be registered with the ws2_32.dll at the time they are installed. This registration need only be done once for each provider as the necessary information is retained in persistent storage.

Transport Service Providers

A given transport service provider supports one or more protocols. For example, a TCP/IP provider would supply (as a minimum) the TCP and UDP protocols, while an IPX/SPX provider might supply IPX, SPX, and SPX II. Each protocol supported by a particular provider is described by a WSAPROTOCOL_INFO structure, and the total set of such structures can be thought of as the catalog of installed protocols. Applications can retrieve the contents of this catalog and by examining the available WSAPROTOCOL_INFO structures, discover the communications attributes associated with each protocol.

Layered Protocols and Protocol Chains in the SPI

Windows Sockets 2 accommodates the notion of a layered protocol. A layered protocol is one that implements only higher level communications functions, while relying on an underlying transport stack for the actual exchange of data with a remote endpoint. An example of such a layered protocol would be a security layer that adds protocol to the connection establishment process in order to perform authentication and to establish a mutually agreed upon encryption scheme. Such a security protocol would generally require the services of an underlying reliable transport protocol such as TCP or SPX. The term base protocol refers to a protocol such as TCP or SPX which is fully capable of performing data communications with a remote endpoint, and the term layered protocol is used to describe a protocol that cannot stand alone. A protocol chain would then be defined as one or more layered protocols strung together and anchored by a base protocol.

This stringing together of layered protocols and base protocols into chains can be accomplished by arranging for the layered protocols to support the Winsock SPI at both their upper and lower edges. A special WSAPROTOCOL_INFO structure is created which refers to the protocol chain as a whole, and which describes the explicit order in which the layered protocols are joined. This is illustrated in the following image.

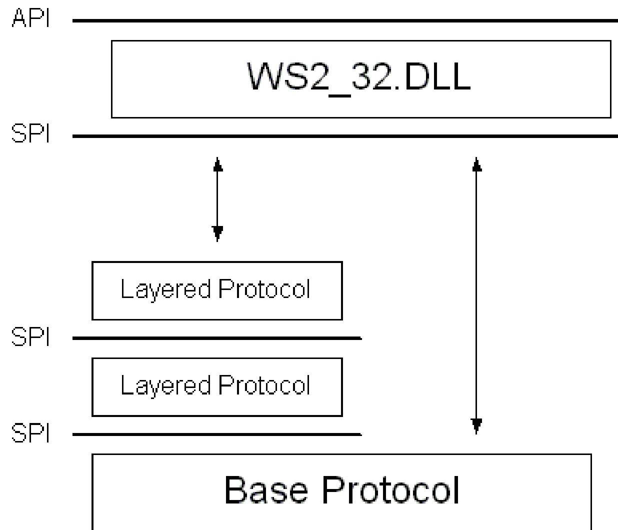

Figure 1

Namespace Service Providers

A namespace provider implements an interface mapping between the Winsock namespace SPI and the native programmatic interface of an existing name service such as DNS, X.500, or NetWare Directory Services (NDS). While a namespace provider supports exactly one namespace, it is possible for multiple providers for a given namespace to be installed. It is also possible for a single DLL to create an instance of multiple namespace providers. As namespace providers are installed, a catalog of WSANAMESPACE_INFO structures is maintained. An application may use WSAEnumNameSpaceProviders() to discover which namespaces are supported on a machine.

Legacy GetXbyY Service Providers

Windows Sockets 2 fully supports the TCP/IP-specific name resolution facilities found in Windows Sockets version 1.1. It does this by including the set of GetXbyY() functions in the SPI. However, the treatment of this set of functions is somewhat different from the rest of the SPI functions. The GetXbyY() functions appearing in the SPI are prefaced with GETXBYYSP_, and are summarized in the following table.

Berkeley Style Functions

SPI function name	Description
GETXBYYSP_gethostbyaddr	Supplies a hostent structure for the specified host address.
GETXBYYSP_gethostbyname	Supplies a hostent structure for the specified host name.
GETXBYYSP_getprotobyname	Supplies a protoent structure for the specified protocol name.

GETXBYYSP_getprotobynumber	Supplies a protoent structure for the specified protocol number.
GETXBYYSP_getservbyname	Supplies a servent structure for the specified service name
GETXBYYSP_getservbyport	Supplies a servent structure for the service at the specified port.
GETXBYYSP_gethostname	Returns the standard host name for the local machine.

Table 1

Async Style Functions

SPI function name	Description
GETXBYYSP_WSAAsyncGetHostByAddr	Supplies a hostent structure for the specified host address.
GETXBYYSP_WSAAsyncGetHostByName	Supplies a hostent structure for the specified host name.
GETXBYYSP_WSAAsyncGetProtoByName	Supplies a protoent structure for the specified protocol name.
GETXBYYSP_WSAAsyncGetProtoByNumber	Supplies a protoent structure for the specified protocol number.
GETXBYYSP_WSAAsyncGetServByName	Supplies a servent structure for the specified service name.
GETXBYYSP_WSAAsyncGetServByPort	Supplies a servent structure for the service at the specified port.
GETXBYYSP_WSACancelAsyncRequest	Cancels an asynchronous GetXbyY() operation.

Table 2

The syntax and semantics of these GetXbyY() functions in the SPI are exactly the same as those documented in the API Specification and are, therefore, not repeated here. The Windows Sockets 2 DLL allows exactly one service provider to offer these services. Therefore, there is no need to include pointers to these functions in the procedure table received from service providers at startup. In Windows environments the path to the DLL that implements these functions is retrieved from the value found in the following registry path. This registry entry does not exist by default:
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\WinSock2\Parameters\GetXByYLibraryPath

Built-In Default GetXbyY() Service Provider

A default GetXbyY() service provider is integrated into the standard Windows Sockets 2 run-time components. This default provider implements all of the above functions, thus it is not required for these functions to be implemented by any namespace provider. However, a namespace provider is free to provide any or all of these functions (and thus override the defaults) by simply storing the string which is the path to the DLL that implements these functions in the indicated registry key. Any of the GetXbyY() functions not exported by the named provider DLL will be supplied through the built-in defaults. Note, however, that if a provider elects to supply any of the async version of the GetXbyY() functions, he should supply all of the async functions so that the cancel operation will work appropriately.
The current implementation of the default GetXbyY() service provider resides within the wsock32.dll. Depending on how the TCP/IP settings have been established through Control Panel, name resolution will occur using either DNS or local host files. When DNS is used, the default GetXbyY() service provider uses standard Windows Sockets 1.1 API calls to communicate with the DNS server. These transactions will occur using whatever TCP/IP stack is configured as the default TCP/IP stack. Two special cases however, deserve special mention.
The default implementation of GETXBYYSP_gethostname() obtains the local host name from the registry. This will correspond to the name assigned to "**My Computer**". The default implementation of GETXBYYSP_gethostbyname() and GETXBYYSP_WSAAsyncGetHostByName() always compares the supplied host name with the local host name. If they match, the default implementation uses a private interface to probe the Microsoft TCP/IP stack in order to discover its local IP address. Thus, in order to be completely independent of the Microsoft TCP/IP stack, a namespace provider must implement both GETXBYYSP_gethostbyname() and GETXBYYSP_WSAAsyncGetHostByName().

Protocol-Independent Out-of-Band Data (OOB)

The stream socket abstraction includes the notion of **out of band** (OOB) data. Many protocols allow portions of incoming data to be marked as special in some way, and these special data blocks can be delivered to the user out of the normal sequence. Examples include expedited data in X.25 and other OSI protocols, and urgent data in BSD UNIX's use of TCP. The following section describes OOB data handling in a protocol-independent manner. A discussion of OOB data implemented using TCP urgent data follows the protocol-independent explanation. In each discussion, the use of recv() also implies

recvfrom(), WSARecv(), and WSARecvFrom(), and references to WSAAsyncSelect() also apply to WSAEventSelect().

Protocol Independent OOB Data

OOB data is a logically independent transmission channel associated with each pair of connected stream sockets. OOB data may be delivered to the user independently of normal data. The abstraction defines that the OOB data facilities must support the reliable delivery of at least one OOB data block at a time. This data block can contain at least one byte of data, and at least one OOB data block can be pending delivery to the user at any one time. For communications protocols that support in-band signaling (such as TCP, where the urgent data is delivered in sequence with the normal data), the system normally extracts the OOB data from the normal data stream and stores it separately (leaving a gap in the normal data stream). This allows users to choose between receiving the OOB 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.
A user can determine if any OOB data is waiting to be read using the ioctlsocket() function with the SIOCATMARK IOCTL. For protocols where the concept of the position of the OOB data block within the normal data stream is meaningful, such as TCP, a Windows Sockets service provider maintains a conceptual marker indicating the position of the last byte of OOB data within the normal data stream. This is not necessary for the implementation of the ioctlsocket(SIOCATMARK) functionality - the presence or absence of OOB data is all that is required.

For protocols where the concept of the position of the OOB data block within the normal data stream is meaningful, an application might process out-of-band data inline, as part of the normal data stream. This is achieved by setting the socket option SO_OOBINLINE with setsockopt(). For other protocols where the OOB data blocks are truly independent of the normal data stream, attempting to set SO_OOBINLINE results in an error. An application can use the SIOCATMARK ioctlsocket command to determine whether there is any unread OOB data preceding the mark. For example, it can use this to resynchronize with its peer by ensuring that all data up to the mark in the data stream is discarded when appropriate.
With SO_OOBINLINE disabled (the default setting):

- Windows Sockets notifies an application of an FD_OOB event, if the application registered for notification with WSAAsyncSelect(), in exactly the same way FD_READ is used to notify of the presence of normal data. That is, FD_OOB is posted when OOB data arrives with no OOB data previously queued. The FD_OOB is also posted when data is read using the MSG_OOB flag while some OOB data remains queued after the read operation has returned. FD_READ messages are not posted for OOB data.
- Windows Sockets returns from select() with the appropriate exceptfds socket set if OOB data is queued on the socket.
- The application can call recv() with MSG_OOB to read the urgent data block at any time. The block of OOB data jumps the queue.
- The application can call recv() without MSG_OOB to read the normal data stream. The OOB data block does not appear in the data stream with normal data. If OOB

data remains after any call to recv(), Windows Sockets notifies the application with FD_OOB or with exceptfds parameter when using select().
- For protocols where the OOB data has a position within the normal data stream, a single recv() operation does not span that position. One recv() returns the normal data before the mark, and a second recv() is required to begin reading data after the mark.

With SO_OOBINLINE enabled:

- FD_OOB messages are not posted for OOB data. OOB data is treated as normal for the purpose of the select() and WSAAsyncSelect() functions, and indicated by setting the socket in readfds or by sending an FD_READ message respectively.
- The application cannot call recv() with the MSG_OOB flag set to read the OOB data block. The error code WSAEINVAL is returned.
- The application can call recv() without the MSG_OOB flag set. Any OOB data is delivered in its correct order within the normal data stream. OOB data is never mixed with normal data. There must be three read requests to get past the OOB data. The first returns the normal data prior to the OOB data block, the second returns the OOB data, and the third returns the normal data following the OOB data. In other words, the OOB data block boundaries are preserved.

The WSAAsyncSelect() routine is particularly well suited to handling notification of the presence of out-of-band-data when SO_OOBINLINE is off.

OOB Data in TCP

The following discussion of out-of-band data, implemented using TCP urgent data, follows the model used in the Berkeley software distribution. Users and implementers should be aware that:

- There are, at present, two conflicting interpretations of RFC 793 (where the concept is introduced).
- The implementation of OOB data in the Berkeley Software Distribution (BSD) does not conform to the Host Requirements laid down in RFC 1122.

Specifically, the TCP urgent pointer in BSD points to the byte after the urgent data byte and an RFC-compliant TCP urgent pointer points to the urgent data byte. As a result, if an application sends urgent data from a BSD-compatible implementation to an RFC-1122 compatible implementation, the receiver reads the wrong urgent data byte (it reads the byte located after the correct byte in the data stream as the urgent data byte).
To minimize interoperability problems, applications writers are advised not to use OOB data unless this is required to interoperate with an existing service. Windows Sockets suppliers are urged to document the OOB semantics (BSD or RFC 1122) that their product implements.
Arrival of a TCP segment with the URG (for urgent) flag set indicates the existence of a single byte of OOB data within the TCP data stream. The OOB data block is one byte in size. The urgent pointer is a positive offset from the current sequence number in the TCP

header that indicates the location of the OOB data block (ambiguously, as noted in the preceding). It might, therefore, point to data that has not yet been received.
If SO_OOBINLINE is disabled (the default) when the TCP segment containing the byte pointed to by the urgent pointer arrives, the OOB data block (one byte) is removed from the data stream and buffered. If a subsequent TCP segment arrives with the urgent flag set (and a new urgent pointer), the OOB byte currently queued can be lost as it is replaced by the new OOB data block (as occurs in Berkeley Software Distribution). It is never replaced in the data stream, however.
With SO_OOBINLINE enabled, the urgent data remains in the data stream. As a result, the OOB data block is never lost when a new TCP segment arrives containing urgent data. The existing OOB data "mark" is updated to the new position.
When the SO_OOBINLINE socket option is set, the SIOCATMARK IOCTL always returns TRUE, and OOB data is returned to the user as normal data.

TCP/IP Raw Sockets

The TCP/IP service providers may support the SOCK_RAW socket type. There are two types of such sockets:

- The first type assumes a known protocol type as written in the IP header. An example of the first type of socket is ICMP.
- The second type allows any protocol number. An example of the second type would be an experimental protocol that is not supported by the service provider.

If a TCP/IP service provider supports SOCK_RAW sockets for the AF_INET family, the corresponding protocol(s) should be included in the list returned by WSAEnumProtocols(). The iProtocol member of the WSAPROTOCOL_INFO structure may be set to zero if the service provider allows an application to specify any value for the protocol parameter for the Socket(), WSASocket(), and WSPSocket() functions. An application may not specify zero (0) as the protocol parameter for the Socket, WSASocket(), and WSPSocket() functions if SOCK_RAW sockets are used. The following rules are applied to the operations over SOCK_RAW sockets:

- When an application sends a datagram it may or may not include the IP header at the front of the outgoing datagrams depending on the IP_HDRINCL option set for the socket.
- An application always gets the IP header at the front of each received datagram regardless of the IP_HDRINCL option.
- Received datagrams are copied into all SOCK_RAW sockets that satisfy the following conditions:

    1. The protocol number specified for the socket should match the protocol number in the IP header of the received datagram.

    2. If a local IP address is defined for the socket, it should correspond to the destination address as specified in the IP header of the received datagram. An application may specify the local IP address by calling bind() functions. If no local IP address is specified for the socket, the datagrams are copied into

the socket regardless of the destination IP address in the IP header of the received datagram.

3. If a foreign address is defined for the socket, it should correspond to the source address as specified in the IP header of the received datagram. An application may specify the foreign IP address by calling connect() functions. If no foreign IP address is specified for the socket, the datagrams are copied into the socket regardless of the source IP address in the IP header of the received datagram.

It is important to understand that SOCK_RAW sockets may get many **unexpected datagrams**. For example, a **PING** program may use SOCK_RAW sockets to send ICMP echo requests. While the application is expecting ICMP echo responses, all other ICMP messages (such as ICMP HOST_UNREACHABLE) may be delivered to this application also. Moreover, if several SOCK_RAW sockets are open on a machine at the same time, the same datagrams may be delivered to all the open sockets. An application must have a mechanism to recognize its datagram and to ignore all others. Such mechanism may include inspecting the received IP header–using unique identifiers in the ICMP header (ProcessID, for example), and so forth. Raw socket support requires administrative privileges. Users running Winsock applications that make use of raw sockets must have administrative privileges on the computer, otherwise raw socket calls will fail with an error code of WSAEACCES. The Microsoft implementation of TCP/IP on Windows is capable of opening a raw UDP socket.

Publishing with Windows Sockets Registration and Resolution

Microsoft® Windows® Sockets services can use the **Registration and Resolution** (RnR) APIs to publish services and look up services published with RnR. RnR publication occurs in **two steps**. The first step **installs** a service class that associates a GUID with a name for the service. The service class can hold service-specific configuration data. Services can then **publish** themselves as instances of the service class. When published, clients can query the directory service for instances of a given class using the RnR APIs and select an instance to bind to. When a class is no longer required, it can be removed.

Nonblocking Input/Output

If a socket is in non-blocking mode, any I/O operation must either complete immediately or return the error code WSAEWOULDBLOCK indicating that the operation cannot be finished right away. In the latter case, a mechanism is needed to discover when it is appropriate to try the operation again with the expectation that the operation will succeed. A set of network events has been defined for this purpose. These events can be polled or waited on by using WSPSelect(), or they can be registered for asynchronous delivery by calling WSPAsyncSelect() or WSPEventSelect().

Blocking Input/Output

The simplest form of I/O in Windows Sockets 2 is blocking I/O. Sockets are created in blocking mode by default. Any I/O operation with a blocking socket will not return until the

operation has been fully completed. Thus, any thread can only execute one I/O operation at a time. For example, if a thread issues a receive operation and no data is currently available, the thread will block until data becomes available and is placed into the thread's buffer. Although this is simple, it is not necessarily the most efficient way to do I/O. Blocking is strongly discouraged due to the non-preemptive nature of the operating system.

Synchronous and Asynchronous I/O

There are two types of file I/O synchronization: **synchronous file I/O** and **asynchronous file I/O**. Asynchronous file I/O is also referred to as **overlapped I/O**. In synchronous file I/O, a thread starts an I/O operation and immediately enters a wait state until the I/O request has completed. A thread performing asynchronous file I/O sends an I/O request to the kernel. If the request is accepted by the kernel, the thread **continues processing another job** until the kernel signals to the thread that the I/O operation is complete. It then interrupts its current job and processes the data from the I/O operation as necessary. These two synchronization types are illustrated in the following figure.

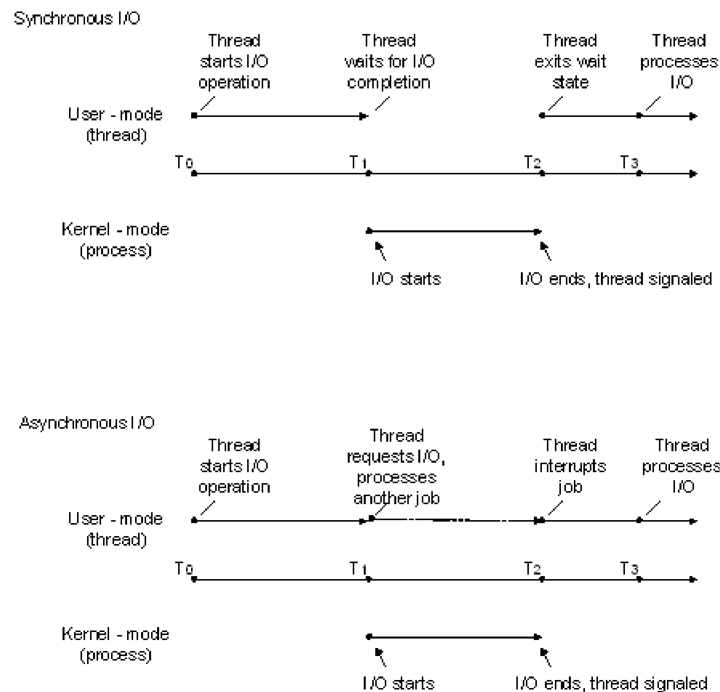

Figure 2

In situations where an I/O request is expected to take a large amount of time, such as a refresh or backup of a large database, asynchronous I/O is generally a good way to optimize processing efficiency. However, for relatively fast I/O operations, the overhead of processing kernel I/O requests and kernel signals may make asynchronous I/O less beneficial, particularly if many fast I/O operations need to be made. In this case, synchronous I/O would be better.
A process opens a file for asynchronous I/O in its call to CreateFile() by specifying the FILE_FLAG_OVERLAPPED flag in the dwFlagsAndAttributes parameter. If

FILE_FLAG_OVERLAPPED is not specified, the file is opened for synchronous I/O. When the file has been opened for asynchronous I/O, a pointer to an OVERLAPPED structure is passed into the call to ReadFile() and WriteFile(). The structure is not passed in calls to ReadFile() and WriteFile() when performing synchronous I/O. Handles to directory objects are obtained by calling CreateDirectory() or CreateDirectoryEx(). Directory handles are almost never used - backup applications are one of the few applications that typically access them.

After opening the file object for asynchronous I/O by calling CreateFile(), an instantiation of the OVERLAPPED structure must be instantiated and passed into each call to ReadFile() and WriteFile(). Keep the following in mind when using this structure in asynchronous read and write operations:

- Do not de-allocate the OVERLAPPED structure or the data buffer until all asynchronous I/O operations to the file object have been completed. If it is de-allocated prematurely, ReadFile or WriteFile may incorrectly report that the I/O operation is complete.
- If you declare your pointer to the OVERLAPPED structure as a local variable, do not exit the local function until all asynchronous I/O operations to the file object have been completed. If the local function is exited prematurely, the OVERLAPPED structure will go out of scope and it will be inaccessible to any ReadFile or WriteFile functions it encounters outside of that function.

You can also create an event and put the handle in the OVERLAPPED structure; the wait functions can then be used to wait for the I/O operation to complete by waiting on the event handle.

An application can also wait on the file handle to synchronize the completion of an I/O operation, but doing so requires extreme caution. Each time an I/O operation is started, the operating system sets the file handle to the non-signaled state. Each time an I/O operation is completed, the operating system sets the file handle to the signaled state. Therefore, if an application starts two I/O operations and waits on the file handle, there is no way to determine which operation is finished when the handle is set to the signaled state. If an application must perform multiple asynchronous I/O operations on a single file, it should wait on the event handle in the OVERLAPPED structure for each I/O operation, rather than on the file handle.

To cancel all pending asynchronous I/O operations, use the CancelIo() function. This function only cancels operations issued by the calling thread for the specified file handle.

The ReadFileEx() and WriteFileEx() functions enable an application to specify a routine to execute when the asynchronous I/O request is completed.