



Search for

MSDN® Magazine
The Microsoft Journal for Developers

MSDN Home > MSDN Magazine > August 2005

MSDN Magazine

Advanced Search

MSDN Magazine Home

August 2005

Search

Source Code

Back Issue Archive

Column Archive

RSS

Subscribe

Order Back Issues

Reader Services

Meet the Staff

Meet the Authors

Media Kit

Special CD and DVD Offers

Podcast

Submit an Article

Write to Us

Corrections

TechNet Magazine



New information has been added to this article since publication. Refer to the [Editor's Update](#) below.

WINSOCK
Get Closer to the Wire with High-Performance Sockets in .NET

[Daryn Kiely](#)



Parts of this article are based on a prerelease version of the .NET Framework 2.0. Those sections are subject to change.

This article discusses:

- Basic socket programming
- Factors in creating scalable socket-based servers
- The System.Net.Sockets namespace
- Debugging socket-based applications

This article uses the following technologies:
C#, .NET Framework

[Get the sample code for this article.](#)

Contents

- [Sockets Primer](#)
- [Making Connections](#)
- [Server Overview](#)
- [Threaded Server](#)
- [Using Select for Multiplexed I/O](#)
- [Asynchronous I/O](#)
- [What About Scalability?](#)
- [Socket Client Applications](#)
- [What's Next?](#)
- [Debugging a Sockets Application](#)

Sockets are the transport mechanism most frequently used in high-performance server applications. Fortunately, the Win32® Windows® Sockets library (Winsock) provides mechanisms to improve the performance of programs that use sockets, and the Microsoft® .NET Framework provides a layer over Winsock so that managed applications can communicate over sockets (see **Figure 1**). So much advanced socket support is great, but using all these layers to write a truly high-performance socket-based application requires a little background information.

I am going to write a trivial chat server app to explore methods for writing a socket-based server and client using the base System.Net.Sockets.Socket class. Although .NET provides higher-level abstractions like the TcpListener and TcpClient classes (also in System.Net.Sockets), these classes are missing some of the advanced features exposed by the lower-level Socket class. That said, they can be useful in many situations. The TcpListener class provides simple methods that listen for and accept incoming connection requests in blocking synchronous mode, while the TcpClient class provides simple methods for connecting, sending, and receiving stream data over a network in synchronous blocking mode.

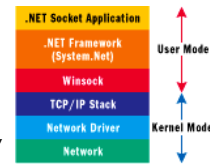


Figure 1 Windows Sockets Layers

Sockets Primer

Sockets define the endpoints for communication, typically across networks. A socket can be used to talk over any of a number of protocols. The most common protocols today are User Datagram Protocol (UDP) and Transmission Control Protocol (TCP).

UDP sockets are connectionless and are generally used for broadcast and multicast communications. UDP has no provision for reliable transport or ordering of messages. It is up to the end application to detect and handle packet loss and packet ordering.

TCP sockets are connection-oriented and provide a reliable communication path between two endpoints. The big advantage of TCP is that it ensures message delivery and proper message ordering. I will focus on TCP sockets in this article.

TCP sockets can either be client or server. A server socket waits for clients to connect; a client socket initiates a connection. Once sockets are connected, both clients and servers can send and receive data or close communications.

To set up a TCP server socket in managed code, the first step is to create an instance of the Socket class. Socket's constructor accepts three parameters: an AddressFamily, a SocketType, and a ProtocolType. The AddressFamily indicates the addressing scheme used by this socket. The two most common AddressFamily values used are InterNetwork for IPV4 addresses and InterNetworkV6 for IPV6 addresses. The SocketType indicates the type of communication that will take place over the socket, the two most common types being Stream (for connection-oriented sockets) and Dgram (for connectionless sockets). The ProtocolType specifies which protocol the socket uses and can have values such as Tcp, Udp, Idp, Gpp, and a host of others. As an example, to create a socket for TCP communication, you could instantiate a Socket as follows:

```
Socket s = new Socket(
    AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
```

Once a socket is created, it can be bound to an address. Binding is optional for client sockets but necessary for server sockets. To bind a socket to an address, call the Socket's Bind method. Bind needs to know the address and port that will be associated with the socket, and thus the method accepts as a parameter an instance of a class that derives from EndPoint. Most of the time, this will be an instance of the IPEndPoint class (the only other EndPoint-derived class included in the Framework is the IrDAEndPoint, which is used for infrared port communication).

The IPEndPoint is a logical representation of the endpoint for network communications; it consists of an IP address and a port. The constructor for an IPEndPoint takes the IP address and port number. The port number is simply an integer representing the port to use, while the IP address can be represented as a .NET IPAddress class or as a long integer (where

each of the segments of the IP address are one of the bytes in the integer). There are several predefined IPAddresses available through static properties on the IPAddress class; the two most useful addresses for TCP-based sockets are IPAddress.Loopback and IPAddress.Any.

IPAddress.Loopback is a well-known address (127.0.0.1) that is local to the current machine. This is a network pseudo-address that does not rely on hardware or any network connectivity. It allows testing to be done locally without having to worry about the actual network or networking hardware. This is the same address that's used when you use "localhost" as the host name in a URL, although that can be changed by editing the hosts file at %windir%\system32\drivers\etc.

IPAddress.Any (0.0.0.0) tells a server to listen for client communications on all of the network interfaces rather than just the interface associated with a particular IP address (UDP behaves slightly differently than TCP in this regard). Why wouldn't you use this all the time? First, you need to be able to control who connects to your server. If you have multiple network interface cards (NICs) in a server, a situation referred to as a dual-homed server, you can listen for connections on one of the interfaces (for instance, on your corporate LAN), but not on the other (such as the NIC hooked up to the Internet), by specifying which network address you want to listen on.

The second reason is security. If your socket is set up to reuse an address (meaning that the socket can be bound to an address that is already in use), and someone else connects to the same port in use by your server application, the underlying socket subsystem must decide which bound socket will receive a packet coming in on that port. Typically the server application that is more tightly bound to the port (that specified a more specific address in the call to Bind) gets the data. The System.Net.NetworkInformation namespace contains classes that provide detailed information about each of the network interfaces in the system. The following code will print out each of the unicast IP addresses for each NIC:

```
foreach (NetworkInterface nic in
    NetworkInterface.GetAllNetworkInterfaces())
{
    Console.WriteLine(nic.Name);
    foreach (UnicastIPAddressInformation addrInfo in
        nic.GetIPProperties().UnicastAddresses)
    {
        Console.WriteLine("\t" + addrInfo.Address);
    }
}
```

To bind to a particular NIC, you can retrieve the IP address information for that NIC and use that address in the IPEndPoint.

Making Connections

An application may want to get an address for a server so it can either make a connection to it or bind to the address. One way to get the address of a server is through the System.Net.Dns class. The Dns class provides forward lookup services for IP addresses and reverse lookup services for host names. It consists of static members that perform network address and name resolution. The Dns.GetHostEntry method returns an IPEndPoint that contains lists of all the relevant information about a machine. It can list all the IP addresses the machine has published to the DNS server. For simplicity, in the examples in this article I'll just use the first address in the list.

After Bind is called, the Socket.Listen method sets up internal queues for a socket. Whenever a client attempts to connect, the connection request is put into a queue. The Listen method takes one argument, the maximum number of socket connections that can be in the pending state. The Socket.Accept method pulls the first pending request from the queue and returns a new Socket instance that can be used for communication with the client.

For a client socket, after creating the Socket instance you typically call the Connect method. You can also first use the Bind method if you require your client to use a specific port. If you do not bind your client socket, the Connect method automatically chooses a port for you. The call to Connect will attempt to establish a connection to the server. Connect also takes an EndPoint, which is used to locate the target remote host. Once a connection is established, either end can use the Send and Receive methods to pass data back and forth.

After a connection is established and the applications are finished communicating over the socket, you must close the socket. The Shutdown method is used to politely initiate a socket closure. This allows all the unsent data to be sent and the unreceived data to be received. If you are synchronously reading data from a socket and you receive 0 bytes, the peer has closed the socket.

Figure 2 summarizes the steps needed to create a server socket and a client socket. As you can see, socket basics are actually fairly simple; what's harder is making socket applications perform well.

When a Socket method encounters a network error, a SocketException is thrown. The exception wraps the error code that was passed up from Win32. For those of you familiar with Win32 Winsock development, SocketException.ErrorCode is the same error code you would have received by calling the WSAGetLastError function in the Winsock library. The key Winsock error codes you should be familiar with are listed in the chart in **Figure 3**. In the .NET Framework 2.0, SocketException.SocketErrorCode provides the same error information through a value from the SocketError enumeration.

Server Overview

The initialization of a server socket involves creating the socket, binding it to an address, and setting up the listen queue. Each of the different forms of servers that as I am going to write will utilize the same code to initialize the socket, as shown in **Figure 4**.

I am going to start with a basic threaded server and explore the limitations of the Socket class. Because the basic API consists of blocking calls, when writing a threaded service you need to spawn threads for accepting connections and performing socket I/O.

There are two ways to avoid creating separate threads for dealing with server sockets. The first is the traditional way of dealing with multiple I/O streams in a single thread—the Select method. The second way is to use asynchronous I/O. Both of these methods for writing a server will be explored in more detail next.

Threaded Server

The threaded server requires you to manage threads that deal with the I/O for all the sockets. The first consideration is the call to Accept. In a synchronous server, the call to Accept must run in its own thread to ensure that sockets are accepted in a timely manner. **Figure 5** shows an example for spawning a new thread to deal with accepting socket connections.

As you can see, once the socket is accepted, a new thread is spawned to do the receiving for that socket. This thread does nothing but receive data from the socket and process it (which in this case means sending it out to all the other connected sockets). In a sense, the example in **Figure 5** is a very basic chat server. To test it, just create a simple .NET Framework 2.0 console application with the following code in the Main method:

```

class Server {
    static void Main(string [] args) {
        ThreadedServer ts = new ThreadedServer(int.Parse(args[0]));
        ts.Start();
        Console.ReadLine();
    }
}

```

This code expects you to run the application with the server port to bind to as the first command-line argument. Run the application specifying a port that isn't being used on your machine. While the application is running, open up a few command prompts and at each one use telnet to connect to your IP address and the port you are specifying when running the server. As soon as they've connected, any text you type into one command prompt window should immediately materialize in all of the others.

This threaded approach works quite well for a small server with a handful of clients, and it is very easy to code. Unfortunately, it does not scale well at all. The main drawback is the sheer number of threads that are created and destroyed. On my machine, the server can accept about 1000 simultaneous connections before getting "out of memory" exceptions (the machine quickly runs out of memory because each running thread has its own stack, which by default is a megabyte in size).



Using Select for Multiplexed I/O

There are several ways to avoid having to create a thread for each connection. The first is simply to use a thread pool—an approach that requires only minimal modifications to the previous example. This has the benefit of throttling the number of threads that can exist at any one time and is a viable approach if connections will be ephemeral. However, as is the case with a chat server, connections to servers typically remain open for prolonged periods of time, so limiting the number of connections to the number of threads in the pool is frequently not a practical approach.

A better solution is to use the static `Socket.Select` method. You can pass to `Select` three lists of Sockets you want to be monitored, one for readability, one for writability, and one for error conditions. When `Select` returns, items that remain in the lists are ready for the operation represented by that particular list. The method also accepts a `microSeconds` parameter that tells the method how long it should wait for a response. One way to use `Select` is to create a timer that fires frequently, and within the timer's event handler call `Select` with a timeout value of zero so that the call is non-blocking. However, this is inefficient. How long it takes to service the sockets would depend on how frequently you timer fires, resulting in a lot of thrashing for no good reason. To avoid this, it's typically better to create a thread that does nothing but call `Select` and service the requests.

Let's first look at how to use `Select` when accepting connections (see [Figure 6](#)). A listening socket should be put into the read list for the call to `Select`. If the listening socket is still present in the read list after the call to `Select`, there is a connection to be accepted (using `Select` in this fashion to determine that a request is available guarantees that `Accept` will not block). At this point, you can call `Accept` and deal with the accepted socket. Conveniently enough, the accepted sockets need to be placed in the same list as the `Accept` socket in order to service incoming data. This means that there is only one list to maintain for your little server.

Let's investigate the implications of using `Select`. Every time you call `Select`, you need to create lists of the sockets you want to monitor. When `Select` returns, these lists are modified to contain only sockets that require servicing. Although this sounds good, in practice it is quite inefficient. Imagine if 100 sockets had pending I/O operations at the same time: the 100th socket would be starved until you finished servicing or scheduling the first 99 sockets. The servicing of the sockets also prevents the code from reentering `Select`, which causes more potential for starvation of threads. You must ensure you have processed the socket I/O for all the sockets before you call `Select` again; failing to do so will result in getting notified for the same I/O more than once.

There are other performance drawbacks to using this method as well. You can see the `Select` performance degrade significantly after a thousand or so clients have connected. This is because the kernel has to internally poll every socket to determine whether there is data available.

Although you can connect significantly more sockets with this method than with the thread-per-request model, this is still not very scalable. You need to manage three different lists and iterate through each in order to service the request. This is more efficient from a thread usage standpoint, but it is much less responsive than using a thread-based server. There must be a better way to deal with multiple sockets.



Asynchronous I/O

The next step in scaling up the server is to use asynchronous I/O. Asynchronous I/O alleviates the need to create and manage threads. This leads to much simpler code and also is a more efficient I/O model. Asynchronous I/O utilizes callbacks to handle incoming data and connections, which means there are no lists to set up and scan and there is no need to create new worker threads to deal with the pending I/O.

So how does the .NET Framework deal with these asynchronous calls? Every .NET-based application has a thread pool associated with it. When an asynchronous I/O function has data that is ready to be processed, a thread from the .NET thread pool performs the callback function. When the callback is complete, the thread is released back into the thread pool. This is different from the approach discussed before where a thread pool thread was dedicated to a particular request; in this case, a thread from the pool is used only for a single I/O operation.

.NET uses a fairly simple model for asynchronous operations. All you need to do is call the relevant `Begin` method (`BeginAccept`, `BeginSend`, `BeginReceive`, and so on). with an appropriate callback delegate, and within your callback call the corresponding `End` method (`EndAccept`, `EndSend`, `EndReceive`, and so on) to get the results. All of the asynchronous `Begin` routines allow the user to pass in a context state object, which can be anything you want. When the asynchronous operation is complete, the object is part of the `IAsyncResult` that is passed into the callback.

In this new asynchronous server, we first call `BeginAccept`. With the first two servers presented in this article we were grappling with two different issues. The threaded server was reasonably fast, but could not accept many connections. The `Select`-based server did not suffer from the same connection limitations, but paid the price with a performance hit.

The asynchronous server removes the limitations on the number of connections it can accept (with some minor exceptions, as I'll explain in the scalability section), and it does so without the negative performance implications of the `Select`-based model. In fact, the performance can be better than in the threading model because the code does not bear the overhead associated with creating and tearing down threads.

Once the `accept` operation has completed, the next step is to queue an asynchronous read. This allows the server to read from the socket without having to explicitly poll or create threads. The asynchronous read is started with a call to `BeginRead`, which the results returned by a call to `EndRead`. The resulting server (see [Figure 7](#)) has significantly higher performance than the `Select`-based server and it can also handle far more connections than the thread-based server. This could be a good solution for the scalable server, but the server could still be made to work even better.



What About Scalability?

No matter which of the servers models I use, on my machine I can't top approximately 4000 simultaneous connections. At that point new client requests start to fail with error messages about lack of buffer space or about queues that are too full: not particularly helpful messages. Obviously this is an unacceptable limit if you want a server to do any heavy-duty processing. So why is the code hitting this wall? It turns out that the limiting factor is the machine's available memory resources, specifically the nonpaged memory pool. When the process has exhausted its nonpaged memory pool, no more socket connections can be made. [[Editor's Update - 1/3/2006: The ephemeral port range is also a limiting factor. For more information, see *When you try to connect from TCP ports greater than 5000 you receive the error 'WSAENOBUFFS \(10055\)'*.](#)]



Socket Client Applications

So now that you've written a server that is very responsive and scalable, you need to make the client application efficient as well. Although the code to do so is relatively simple, there are several considerations when writing a socket client.

As with all socket applications, the client socket must first be created. Once created, it may or may not be bound explicitly to an address and port. If you choose not to bind your client socket to an address, it will be given whatever address the network layer determines is best. If a port is not selected, the network layer will provide a unique port within the range 1024 to 5000.

Why then would you want to bind a client socket to an address or port? Consider the situation in which you have a multi-homed client machine and you want to choose the NIC that your client will use. You should bind the client socket to the IP address of the desired NIC. Fortunately, you can call Bind with a port number of 0 and the network layer will still provide you with a unique port within the range of 1024 to 5000. Binding a client socket to a specific port is not recommended, but it is possible.

Once the socket is created, and possibly bound, it is time to connect to the server. Just as with most of the server calls, the connection to the server can be made synchronously or asynchronously. The asynchronous version of Connect is useful if your client needs to establish connections to multiple servers. Instead of having each connection wait for the previous one to complete, the application can start multiple connections at the same time.

There is a very real possibility of getting a WSAECONNREFUSED error when attempting to connect a client socket, even if the server is available. This can happen if the server's listen queue is full. For this reason, a client may want to handle this exception by periodically retrying its connection.



What's Next?

Now that you've determined that the asynchronous socket model is the most scalable, how do you continue to improve performance? Let's break down the functions that a server needs to perform, and see where to make improvements.

The first thing a server needs to do is accept connections. There are three basic ways to do an asynchronous accept: just accept the connection, accept the connection and receive the first bytes of data on that connection, and accept the connection using a specific socket you provide, receiving the first bytes of data on the connection. Only the first of these approaches is available in the .NET Framework 1.x. In the .NET Framework 2.0, all three are available as overloads of BeginAccept and EndAccept.

The code for the servers in this article has been based on the basic asynchronous accept. The advantage of just accepting the connection is that you can separate your accept logic from your message-processing logic. This is a very simple programming model, but it may not provide the best performance.

The advantage of the "accept and receive" approach is a performance boost on your first message, primarily because you have to make fewer expensive calls to the kernel. This method works quite well if you have a known first few bytes, like a protocol header. The down side of this approach is that the callback supplied to BeginAccept is not called until the receive is complete. This may lead to connected sockets that your application does not know about.

What about the "accept and receive" method where you can pass in a socket? This offers a great advantage in the .NET world, especially when combined with the .NET Framework 2.0 Socket.Disconnect call. Historically, when you were finished using the socket, you needed to close the connection and delete the socket object. With Socket.Disconnect, you can reuse Socket instances, which provides you with the means to implement your own Socket object pool. In certain scenarios having this kind of control can result in a long-term performance boost.

Now let's think about how socket connection processing works. First, you tell the operating system to listen for connections, and it creates a queue to hold a certain number of pending connections. Let's assume the maximum size is 200. This means that if more than 200 clients try to connect simultaneously, you will get WSAECONNREFUSED errors (the server actively refused the connection) in each of the clients that overflowed the listen queue. A remedy for this is to queue multiple accepts; the multiple accepts will be able to pull pending connections from the queue even if you are still doing your processing and have not resubmitted the accept request. This effectively increases the depth of your listening queue. If you take another look at [Figure 7](#), you'll see that I start with 10 asynchronous calls to accept.

However, queuing multiple asynchronous operations has a cost. Earlier I discussed why I was getting a limited number of connections on what should be a highly scalable server. These asynchronous calls deplete the same resource pool that was causing the connection problem; in this case, greater performance comes at the cost of scalability.



Debugging a Sockets Application

One new feature in .NET Framework 2.0 is the ability to perform robust network tracing. This feature can make debugging socket applications much easier than in the past, when you typically relied solely on a debugger and a packet sniffer. By setting up some parameters within your configuration file, you can see critical events, errors, warnings, method entry and exits, and actual network traffic. This is all exposed via the standard Trace functionality within .NET.

In the System.Diagnostics section of the configuration file, you need to add a trace source for the System.Net namespace you want to log (valid sources include "System.Net", "System.Net.Sockets", and "System.Net.Cache"). In this case, that would be System.Net.Sockets. An example of the application configuration file is shown in [Figure 8](#).

Once the network tracing has been enabled, you will start to see a lot of internal information about your sockets-based applications. The value attribute for the System.Net-based trace logging is a bit mask with each bit representing a different level of logging. There are five levels you can enable: network traffic (0x10), method entry and exit (0x8), warnings (0x4), errors (0x2), and critical events (0x1). The example in [Figure 8](#) is logging everything, for a combined value of 31.

Tracing method entry and exit is a very useful tool as it details when you are calling different methods within the System.Net namespace. This feature not only helps you to debug your own program, but it can give some insight as to how the API is working internally. For calls that have a return value, the return is shown in the exiting log for that call. Following is an example of the method entry and exit log for an asynchronous accept call; notice that from this log you can tell that a socket was created within the API. The return value from BeginAccept is an AcceptOverlappedAsyncResult, and the object ID for this particular call is 48209832.

```
System.Net.Sockets Information: 0 : Socket#48285313::BeginAccept ()
System.Net.Sockets Information: 0 : Socket#59817589::Socket (InterNetwork#2)
```

```
System.Net.Sockets Information: 0 : Exiting Socket#59817589::Socket()  
System.Net.Sockets Information: 0 : Exiting Socket#48285313::BeginAccept()  
-> AcceptOverlappedAsyncResult#48209832
```

You should keep in mind that not all methods provide the network trace hooks. For instance, `Select` does not show up in the trace logs, but `Bind` does. Check the documentation to determine which methods support the trace options (see [Network Tracing](#)).

The network traffic log is another very useful tool. This log is very straightforward; it is simply a dump of the data buffer from the send or the receive. One thing to be aware of is that your buffers will be dumped in their entirety (up to the maximum specified size), which may be misleading. You may only receive 1 byte in a 512-byte receive buffer, but you will see the contents of the entire buffer. You have to be certain to check the return value of the send or the receive to ensure the bytes you are looking at are relevant. The following trace is from an application sending out 1 byte of data (an "a") on socket 60504909.

```
System.Net.Sockets Information: 0 : Socket#60504909::Send()  
System.Net.Sockets Verbose: 0 : 00000000 : 61 : a  
System.Net.Sockets Information: 0 : Exiting Socket#60504909::Send()  
-> 1#1
```

The other three levels of logging (warnings, errors, and critical events) provide details about failures within the APIs. Each of these logging types is preceded in the log by their level and contains details about the message. For instance, if a call to `Socket.GetSocketOptions` fails, an error is generated and appears in the log as you can see here:

```
System.Net.Sockets Error: 0 : Exception in the  
Socket#48285313::GetSocketOption - An unknown, invalid, or unsupported  
option or level was specified in a getsockopt or setsockopt call
```

The new logging capabilities of System.Net are very powerful and can be extremely useful in debugging your sockets application. The only thing to be wary of when using this debugging facility is the sheer quantity of data it can potentially generate. It can quickly fill a disk with debugging information.

As you can see, there are multiple approaches to improving the performance of socket applications. This article only scratched the surface of the full socket API.



NEW: [Explore the sample code online!](#) - or - **Code download available at:** [HighPerformanceSockets.exe](#)
(169KB)

Daryn Kiely works in the research and development department at International Game Technology in Las Vegas, Nevada. He can be contacted at thekielys@earthlink.net.



From the [August 2005](#) issue of [MSDN Magazine](#).

[Back to top](#)

QJ: 050804

© 2007 Microsoft Corporation and CMP Media, LLC. All rights reserved; reproduction in part or in whole without permission is prohibited.



10-14 March
Barbican
London

The UK's
biggest
conference
for software
developers

Silverlight
.NET Framework 3.5
WCF
Visual Studio 2008
LINQ
ASP.NET AJAX
SQL Server 2008
C# 3.0
Patterns
WPF



Click for full
session details

RESOURCES

Featured Topic

Additional
Topics

More resources from
msdnmagresources.com



Related Articles from MSDN Magazine:

- [C# 3.0: The Evolution Of LINQ And Its Impact On The Design Of C#](#) by Anson Horton
- [Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework](#) by Jeffrey Richter
- [ASP.NET: 10 Tips for Writing High-Performance Web Applications](#) by Rob Howard
- [Concurrent Affairs: Simplified APM with C#](#) by Jeffrey Richter
- [Windows Services: New Base Classes in .NET Make Writing a Windows Service Easy](#) by Ken Getz
- [.NET Tools: Ten Must-Have Tools Every Developer Should Download Now](#) by James Avery
- [Garbage Collection-Part 2: Automatic Memory Management in the Microsoft .NET Framework](#) by Jeffrey Richter
- [C++ -> C#: What You Need to Know to Move from C++ to C#](#) by Jesse Liberty

Print E-Mail Add to Favorites

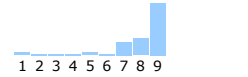
How would you rate the quality of this content?

1 2 3 4 5 6 7 8 9
Poor Outstanding

Tell us why you rated the content this way. (optional)

Submit

Average rating:
8 out of 9



103 people have rated this page

[Manage Your Profile](#) | [Legal](#) | [Contact us](#) | [MSDN Flash Newsletter](#)

© 2007 Microsoft Corporation. All rights reserved. [Terms of Use](#) | [Trademarks](#) | [Privacy Statement](#)

