SocketClient, Meet SocketServer

The .NET Micro Framework is a great platform for small, connected devices thanks to its built-in sockets-based networking classes. Using these classes, you can create clients for network services or host a server on your device. We'll show you how to get started in this article.

A *socket* is a protocol-independent abstraction of a communications endpoint and an associated set of standardized methods for establishing a network connection and for sending and receiving data through the connection. The socket model is designed to work with any network protocol and address format, although today, TCP/IP is probably the most common network protocol used. Sockets first made their appearance in the Berkeley variety of UNIX in the late 1980s, and have been a standard part of Windows since Windows 98 and Windows NT 4.0.The .NET Micro Framework sockets implementation resides in the .NET Micro Framework Hardware Abstraction Layer (HAL).

The upcoming .NET Micro Framework v2.5 includes a native sockets implementation with TCP/IP protocol support, which should lead to a wider selection of network-capable .NET Micro Framework hardware platforms in the future. Until v2.5, networking is available to .NET Micro Framework applications only on hardware that uses an underlying operating system or kernel with sockets functionality. Digi International's Connect ME is an available development module that provides TCP/IP networking for .NET Micro Framework applications using this approach. You can also use the Socket class with applications running in the .NET Micro Framework emulator, which uses the Windows network stack. This is an ideal way to experiment with sockets and write small client and server applications.

The .NET Micro Framework includes two sample programs, SocketServer and SocketClient, that illustrate how to use the socket classes. SocketServer is a simple TCP/IP server. SocketClient is a simple client that displays the response it gets in the Visual Studio Output window. First, we'll run these programs in the emulator and get them talking to each other. Then we'll take a quick look at the code and discuss possible enhancements to the server application.

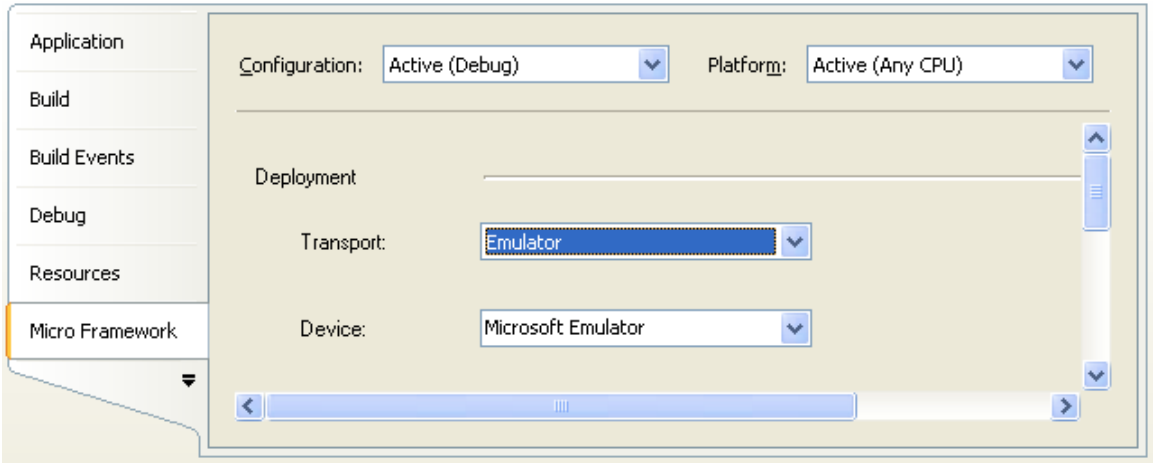
Note This article describes slightly different versions of the SocketClient and SocketServer samples than shipped with the .NET Micro Framework SDK 2.0 SP1. The full source code for both is provided at the end of this article and may be copied into your existing samples to update them. The new versions of the samples will also be provided in .NET Micro Framework v2.5.

Running SocketServer

To try out the SocketServer application, click **Start**, then point to **All Programs**, then **Microsoft .NET Micro Framework**, and finally click **Samples**. The `Samples` folder opens. Open the `SocketServer` folder and then the `SocketServer.sln` Visual Studio solution file to open the application in Visual Studio.

Next, make sure that SocketServer is configured to run in the emulator. In Visual Studio, choose **SocketServer Properties** from the Visual Studio **Project** menu to display the project's

properties. Click the **Micro Framework** tab and make sure the transport is set to **Emulator** and the device is set to **Microsoft Emulator**, as shown here.



Now run SocketServer by choosing **Start Debugging** from the Visual Studio **Debug** menu or by pressing F5. When the emulator window appears, the server is up and running on port 12000 on your workstation. Closing the emulator will stop the server, so leave this window open for now.

Verify that the server is running by opening the URL http://localhost:12000/ in Internet Explorer. You should see the following link in the browser:

> Learn more about the .NET Micro Framework by clicking here

Running SocketClient

Now you can run SocketClient to retrieve the HTML document from SocketServer. As written, SocketClient is hard-coded to retrieve the Microsoft Network home page, so you will need to make a couple of minor modifications to have it connect to your local SocketServer instead.

Go back to the .NET Micro Framework Samples folder and open the `SocketClient` folder, then the `SocketClient.sln` Visual Studio solution file. The application opens in Visual Studio.

Now open the `SocketClient.cs` source file, and, in the `Main` method of the `SocketClientDemo` class, find this line:

```
String html = GetWebPage("www.msn.com");
```

Change it to:

```
String html = GetWebPage("localhost");
```

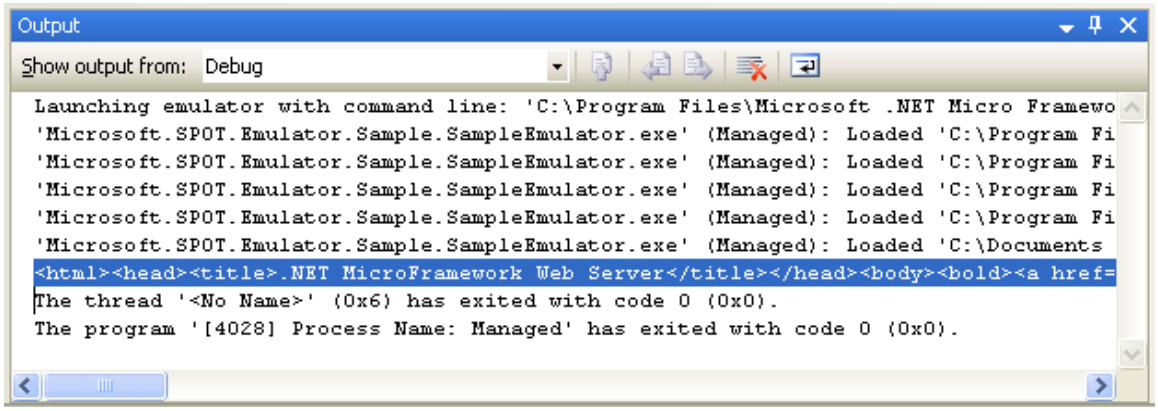
In the method `GetWebPage`, find this line:

```
const Int32 c_httpPort = 80;
```

Change it to:

```
const Int32 c_httpPort = 12000;
```

Make sure that SocketClient is configured to run in the emulator following the procedure described earlier, then run it by choosing **Start Debugging** from the Visual Studio **Debug** menu or pressing F5. If your modifications were successful, the emulator window appears for a moment, then disappears. Check the Output window to see the text retrieved from the SocketServer, as shown here. Choose **Output** from the Visual Studio **View** menu if the Output window is not already visible.



Now that you've seen the network client and server in operation, both running on the .NET Micro Framework, it's time to pop the hood and look at the code, which is simpler than you might expect.

Inside SocketClient

SocketClient's mission is to connect to a network host, send a request, and retrieve the response. This is accomplished by the `GetWebPage` method in the `SocketClientDemo` class and involves four main steps:

1. Create the socket and connect to the desired host
2. Send the request
3. Receive the response
4. Disconnect from the host and dispose of the socket

In the HTTP protocol, we can tell the remote host to disconnect as soon as it finishes responding to the request—in fact, it's the default behavior—so we actually won't need to do all of step 4 ourselves. Instead, we can use the C# `using` construct to have the socket object disposed automatically. If you need to leave the connection open for additional requests in your own applications, you would need to close and dispose of the socket explicitly when you're done with it. This might include non-HTTP clients or HTTP clients that use the HTTP 1.1 keep-alive feature.

Any TCP/IP client needs to know what host it should connect to, and on what port. As shown in the following lines, the `GetWebPage` method accepts the server address as an argument and assumes the server is listening on port 80, the standard HTTP port. Earlier, in our demo, we

changed it to 12000, because that's the port SocketServer listens on by default, and we wanted illustrate how they work together. This is what the code looks like in the unedited version of the sample.

```
private static String GetWebPage(String server) {
    const Int32 c_httpPort = 80;
```

We also need the HTTP request itself. In this case, it's a simple `GET /`, which in HTTP looks something like this:

```
GET / HTTP/1.1
Host: www.msn.com
Connection: Close
```

We will see in a moment how to construct this in C#.

Last but not least, we need a place to store the data returned by the request. It's impossible to know how long the request will be ahead of time—Web pages can be any size—so we can't allocate a static buffer for it. Fortunately, the .NET Micro Framework supports dynamically-allocated **String** objects, so this isn't as big a problem as it would be in C or assembly. You must still, however, read from a socket into a fixed-size buffer, then concatenate the buffer onto a longer string. **SocketClient** uses a 1024-byte buffer, defined as follows:

```
        Byte[] buffer = new Byte[1024];
```

The size of the buffer is entirely up to you, but 1024 bytes is a reasonable size for a small device. The HTTP response will be accumulated from this buffer into a `String` variable called `page`:

```
        String page = String.Empty;
```

The first thing we do is open a socket to the desired host with a call to the provided `ConnectSocket` method, construct the request, and send it using the `Send()` method. As we noted earlier, employing the `using` construct frees us from having to worry about disposing of the `Socket` object when we're done with it.

```
// Create a socket connection to the specified server and port.
using (Socket serverSocket = ConnectSocket(server, c_httpPort)) {
        // Send request to the server.
        String request = "GET / HTTP/1.1\r\nHost: " + server + "\r\nConnection: Close\r\n\r\n";
        Byte[] bytesToSend = Encoding.UTF8.GetBytes(request);
        serverSocket.Send(bytesToSend, bytesToSend.Length, 0);
```

As you can see, an HTTP request's line endings are carriage return/line feed pairs, and the request header is terminated by a blank line and thus an extra CR+LF.

Now that we have opened a socket and sent a request, we can read the data using a loop. The `Socket` methods we'll use are `Poll`, which waits for the socket to have data to read or to time out, and `Receive`, which reads any waiting data from the socket into our buffer. Since we always call `Receive` only after `Poll`, `Receive` will always receive at least one byte of data unless the socket has been closed or timed out; this gives us the terminal condition for our read loop.

```
// Poll for data until 30 second time out - Returns true for data and connection closed
while (serverSocket.Poll(30 * c_microsecondsPerSecond, SelectMode.SelectRead))
{
```

```
        // Zero all bytes in the re-usable buffer
        Array.Clear(buffer, 0, buffer.Length);

        // Read a buffer-sized HTML chunk
        Int32 bytesRead = serverSocket.Receive(buffer);

        // If 0 bytes in buffer, then connection is closed
        if (bytesRead == 0)
                break;

        // Append the chunk to the string
        page = page + new String(Encoding.UTF8.GetChars(buffer));
}

return page;   // Return the complete response
```

This loop begins by polling the socket, waiting up to thirty seconds for data to arrive. If `Poll` returns false, the request has timed out and the loop exits. If `Poll` returns true, there is at least some data waiting, or else the host has closed the connection. In either case, we read the waiting data from the socket into the buffer, then, if any data was read, we append it to the `page` string. If the length of the data read into the buffer was zero, we know `Poll` returned true because the connection was closed, so we exit the loop.

Inside SocketServer

SocketServer, like any network server, uses the `Listen` and `Accept` methods of the Socket class to wait for and accept incoming connections. `Listen` blocks execution until a connection request is received. Additional requests received before the first can be processed are queued and processed in the order they were received. `Accept`, as its name implies, accepts a connection request, returning a new `Socket` object which can be used to receive data from and send data to the remote client (the original `Socket` object continues listening for new connections, immediately processing a queued request if one exists).

A basic TCP/IP server like SocketServer follows these steps:

1. Create a socket and bind it to the local IP address
2. Listen for an incoming connection
3. Accept the connection, creating a session socket and potentially a new thread
4. Read and parse the request
5. Generate and send the response
6. Disconnect and dispose of the session socket
7. Repeat steps 2-6 to process additional connections

Creating and binding the socket, then listening for a connection, takes just a few lines of code:

```
const Int32 c_port = 12000;
Socket server = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
        ProtocolType.Tcp);
IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, c_port);
server.Bind(localEndPoint);
server.Listen(Int32.MaxValue);
```

When a connection is requested by a client, it is accepted as follows:

```
Socket clientSocket = server.Accept();
```

As noted earlier, `Accept()` creates a new socket; we call it `clientSocket` and pass it to our `ProcessClientRequest` method, where it ends up named `m_clientSocket`.

`ProcessClientRequest` in turn calls `ProcessRequest` in a new thread so we can handle multiple client requests at once. As in SocketClient, the `using` construction is used to make sure that the socket is closed and disposed of after the request has been completely processed.

In a real server, you would want to use a read loop similar to the one in SocketClient to read the request, then parse it and construct an appropriate response. SocketServer sends the same response regardless of what you sent, though, so we just read the first chunk of the HTTP request (if any) into a buffer and leave it at that. We will take a look at how you might start making this code into a real HTTP server soon.

```
// From the client, read the HTTP request and ignore it.
Byte[] buffer = new Byte[1024];
if (!m_clientSocket.Poll(5 * c_microsecondsPerSecond, SelectMode.SelectRead)) {
        // We waited 5 seconds and no bytes are ready to read, ignore this client
        return;
        }
Int32 bytesRead = m_clientSocket.Receive(buffer);
```

Now that we know we have a request, we can send the response:

```
// Return a static HTML response to the client
String s = "HTTP/1.1 200 OK\r\nContent-Type: text/html; charset=utf-8\r\n\r\n<html><head><title>.NET
MicroFramework Web Server</title></head><body><bold><a
href=\"http://msdn.microsoft.com/embedded/netmf/\">Learn more about the .NET Micro Framework by clicking
here</a></bold></body></html>";
m_clientSocket.Send(Encoding.UTF8.GetBytes(s));
```

The `using` construct then takes care of tearing down the client connection, and we're done. The `while (true)` loop in `Main` keeps SocketServer listening for connections forever—or at least until you close the emulator window.

Toward an Embedded Web Server

SocketServer and SocketClient are simple example programs. Although they use HTTP requests, they are not intended to be full-fledged HTTP clients or servers. The SocketClient does not include any code to deal with HTTP response codes or, for that matter, *any* part of the response header. As we have seen, SocketServer does not even make sure it has read the entire request, let alone try to parse it; since it always returns the same response, there's no need to.

To give SocketServer a bit more of the functionality of a real HTTP server, we need to use a read loop similar to the one in SocketClient to read the request, then parse the header and construct an appropriate response. In SocketClient, the read loop simply called `Receive()` until we couldn't read any more because the server closed the socket or the request timed out. However, this approach won't work with SocketServer because the client doesn't close the connection after sending the request. The connection must remain open so the server can send back a response.

Instead, we must test the request after receiving each chunk to know when we have received the end of the HTTP header. An HTTP request header can contain multiple lines, each with a carriage return and linefeed, and is terminated by a blank line (two CR+LF pairs). This may be followed by a request body in some types of requests, such as `POST`. We therefore read from the socket until our `request` string contains the HTTP header terminator (CR, LF, CR, LF). The following code reads the entire HTTP header and prints it to the Visual Studio Output window. It is a drop-in replacement for the original `using` block.

```
using (m_clientSocket)
{

        // Buffer for reading the HTTP request
        Byte[] buffer = new Byte[1024];

        // The HTTP request
        String request = String.Empty;

        // Read chunks until we reach the end of the HTTP header
        while (m_clientSocket.Poll(5 * c_microsecondsPerSecond, SelectMode.SelectRead))
        {
                Array.Clear(buffer, 0, buffer.Length);
                m_clientSocket.Receive(buffer);
                request = request + new String(Encoding.UTF8.GetChars(buffer));

                if (request.IndexOf("\r\n\r\n") > -1)
                        break;

           }

        // Do something with the HTTP request here
        Debug.Print(request);

        // Return a static HTML response to the client
        String s = "HTTP/1.1 200 OK\r\nContent-Type: text/html; charset=utf-8\r\n\r\n<html><head><title>.NET
MicroFramework Web Server</title></head><body><bold><a
href=\"http://msdn.microsoft.com/embedded/netmf/\">Learn more about the .NET Micro Framework by clicking
here</a></bold></body></html>";
        m_clientSocket.Send(Encoding.UTF8.GetBytes(s));

}
```

After we have successfully read the HTTP header, we might split the header lines and put the individual header fields into a collection, then look for a message body and read it if it exists. The HTTP header will contain a field indicating the length of the body, so we can tell when to stop reading and can reject in advance a body longer than we are prepared to deal with. We will leave these embellishments as the proverbial “exercise for the reader,” as header parsing is fairly straightforward string manipulation.

Conclusion

Don’t let the fact that you’re developing for small devices keep you from adding client-server functionality. Though they are bare-bones, the SocketServer and SocketClient samples contain useful code you can adapt for your own projects. Experiment freely and discover how easy it is to write TCP/IP clients and servers using the .NET Micro Framework.

SocketClient.cs

```
// Copyright (C) Microsoft Corporation.  All rights reserved.

using System;
using Microsoft.SPOT;
using System.Net.Sockets;
using System.Text;
using System.Net;
using Socket = System.Net.Sockets.Socket;

public static class SocketClientDemo {
   public static void Main() {
      String html = GetWebPage("www.msn.com");
      Debug.Print(html);
   }

   // This method requests the home page content for the specified server.
   private static String GetWebPage(String server) {
      const Int32 c_httpPort = 80;
      const Int32 c_microsecondsPerSecond = 1000000;


      // Create a socket connection to the specified server and port.
      using (Socket serverSocket = ConnectSocket(server, c_httpPort)) {
         // Send request to the server.
         String request = "GET / HTTP/1.1\r\nHost: " + server + "\r\nConnection: Close\r\n\r\n";
         Byte[] bytesToSend = Encoding.UTF8.GetBytes(request);
         serverSocket.Send(bytesToSend, bytesToSend.Length, 0);

         // Allocate a buffer that we'll keep reusing to receive HTML chunks
         Byte[] buffer = new Byte[1024];

         // 'page' refers to the HTML data as it is built up.
         String page = String.Empty;

         // Poll for data until 30 second time out - Returns true for data and connection closed
         while (serverSocket.Poll(30 * c_microsecondsPerSecond, SelectMode.SelectRead))
         {
             // Zero all bytes in the re-usable buffer
             Array.Clear(buffer, 0, buffer.Length);

             // Read a buffer-sized HTML chunk
             Int32 bytesRead = serverSocket.Receive(buffer);

             // If 0 bytes in buffer, then connection is closed
             if (bytesRead == 0)
                 break;

             // Append the chunk to the string
             page = page + new String(Encoding.UTF8.GetChars(buffer));
         }

         return page;   // Return the complete string
      }
   }

   private static Socket ConnectSocket(String server, Int32 port) {
      // Get server's IP address.
      IPHostEntry hostEntry = Dns.GetHostEntry(server);

      // Create socket and connect to the server's IP address and port
      Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
      socket.Connect(new IPEndPoint(hostEntry.AddressList[0], port));
      return socket;
   }
}
```

SocketServer.cs

```
// Copyright (C) Microsoft Corporation.  All rights reserved.

using System;
using System.Threading;
using System.Net;
using System.Net.Sockets;
using System.Text;
using Microsoft.SPOT;
using Socket = System.Net.Sockets.Socket;

public static class SocketServerDemo {
   public static void Main() {
      // Test this app in your Internet browser by going to http://127.0.0.1:12000/
      const Int32 c_port = 12000;

      // Create a socket, bind it to the server's port and listen for client connections
      Socket server = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
      IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, c_port);
      server.Bind(localEndPoint);
      server.Listen(Int32.MaxValue);

      while (true) {
         // Wait for a client to connect
         Socket clientSocket = server.Accept();

         // Process the client request (synchronously or asynchronously)
         new ProcessClientRequest(clientSocket, true);   // Asynchronous process selected
      }
   }
}

internal sealed class ProcessClientRequest {
   private Socket m_clientSocket;

   public ProcessClientRequest(Socket clientSocket, Boolean asynchronously) {
      m_clientSocket = clientSocket;

      if (asynchronously)
         new Thread(ProcessRequest).Start();
      else ProcessRequest();
   }

   private void ProcessRequest() {
      const Int32 c_microsecondsPerSecond = 1000000;

      // 'using' ensures that the client's socket gets closed
      using (m_clientSocket) {
         // From the client, read the HTTP request and ignore it.
         Byte[] buffer = new Byte[1024];
         if (!m_clientSocket.Poll(5 * c_microsecondsPerSecond, SelectMode.SelectRead)) {
            // We waited 5 seconds and no bytes are ready to read, ignore this client
            return;
         }
         Int32 bytesRead = m_clientSocket.Receive(buffer);

         // Return a static HTML page back to the client
         String s = "HTTP/1.1 200 OK\r\nContent-Type: text/html; charset=utf-8\r\n\r\n<html><head><title>.NET
Micro Framework Web Server</title></head><body><bold><a
href=\"http://msdn.microsoft.com/embedded/netmf/\">Learn more about the .NET Micro Framework by clicking
here</a></bold></body></html>";
         m_clientSocket.Send(Encoding.UTF8.GetBytes(s));
      }
   }
}
```

When updating this sample in Visual Studio, note that the construction of the HTTP response (`String s = "HTTP 1.1 200 OK...";`) must be on a single source line. You may need to manually remove line breaks from this long statement.