

Ethernet Sample Application with Sample Code

The purpose of this application is to demonstrate how to communicate with a Microcom Model 485 printer using the Ethernet interface. Specific communication features demonstrated here include communication preparation, sending information to the printer, and receiving information from the printer. This document assumes that the Ethernet interface is already selected on the printer as the communication interface being used. All sample code that appears in this document is included in the Microsoft Visual Studio 485EthernetApp solution.

Step 1: Preparing Communication

Before data transfer between the host and the printer can occur, the communication endpoints must be set up. More specifically, before writing to the printer, a TCP/IP socket is required to know where to write data. Likewise, before reading from the printer, a TCP/IP socket is required to know where to read from. In order to create a socket, the IP address and the port number of the printer must be known a priori. The default IP address for the Microcom Model 485 printer is 10.42.0.2 and the default port is 9100. In the sample application, connection to the socket is made via an asynchronous call in a separate thread, shown below. The IP address is passed in as an object so that the thread can be created with parameters passed into it.

```
public void connect(object ip)
{
    threadSafeSetConnectedToText("Connecting...");
    try
    {
        socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
        socket.Connect(ip.ToString(), 9100);
    }
    catch (Exception ex) { }
}
```

Step 2: Sending to the Printer

Once the socket is created and connected, data can be sent to the printer. This can be done using the `socket.Send` function. `Socket.Send` can be called in-line with code as well as resulting from a button press. In the sample application, sending is done as follows:

```
try
{
    if (socket.Connected)
        socket.Send(fileBytes);
}
catch (Exception ex)
{
    threadSafeSetConnectedToText("Connected To: None, Send Failed");
    try
    {
        socket.Close();
    }
    catch (Exception closeEx) { }
}
```

In the sample code above `fileBytes` is a byte array. Because this app uses the TCP/IP Ethernet interface, this byte array can be any size. Sending these bytes should succeed as long as there is a valid connection on the socket.

Step 3: Receiving from the Printer

All receiving done in the Ethernet sample app is done asynchronously, thus a separate thread is used to continuously read. This ensures that data will be read as soon as it is available. The function used to read data from the socket is `socket.Receive`. This function returns the number of bytes read and a byte array is passed in by reference to store the data read. If there is data to be read but less data to be read than the full byte array, it will read the available data into the beginning of the array and the rest of the array will be 0. In the sample code a byte array of size 4096 is passed in. The size is made larger than any expected receive call would return. This is so that if there is ever data available, it will not take more than one read call to read all of the data. This is to decrease overhead that would come with multiple read calls. If there is no data to be read this call will hang until there is data to be read. This is the reason to call `socket.Receive` in a separate thread. The read thread entry is shown below.

```
while (socket.Connected)
{
    byte[] receiveBuffer = new byte[4096];
    int bytesRead = 0;
    try
    {
        bytesRead = socket.Receive(receiveBuffer);
    }
    catch (SocketException ex)
    {
        threadSafeSetConnectedToText("Connected To: None, Connection Reset");
    }
    if (bytesRead > 0)
    {
        string receivedStr = "";
        for (int i = 0; i < receiveBuffer.Length; i++)
            receivedStr += (char)receiveBuffer[i];
        threadSafeAddConsoleText(receivedStr + Environment.NewLine);
    }
}
```

This can only be done after the socket has a valid connection. If the socket connection is closed, any currently hanging or future calls to `socket.Receive` will fail. After that point the while loop condition will be checked and the entire thread will exit.