[Serial Port Communication In C#](http://www.dreamincode.net/forums/topic/35775-serial-port-communication-in-c%23/)

Welcome to my tutorial on Serial Port Communication in C#. Lately Ive seen a lot of questions on how to send and receive data through a serial port, so I thought it was time to write on the topic. Back in the days of Visual Basic 6.0, you had to use the **[MSComm Control](http://msdn2.microsoft.com/en-us/library/aa259393(vs.60).aspx" \o "External link)** that was shipped with VB6, the only problem with this method was you needed to make sure you included that control in your installation package, not really that big of a deal. The control did exactly what was needed for the task.  
  
We were then introduced to .Net 1.1, VB programmers loved the fact that Visual Basic had finally evolved to an OO language. It was soon discovered that, with all it's OO abilities, the ability to communicate via a serial port wasn't available, so once again VB developers were forced to rely on the **MSComm Control** from previous versions of Visual Basic, still not that big of a deal, but some were upset that an intrinsic way of serial port communication wasn't offered with the .net Framework. Worse yet, C# developers had to rely on a Visual Basic control and Namespace if they wanted to communicate via serial port.  
  
Then along comes .Net 2.0, and this time Microsoft added the **[System.IO.Ports](http://msdn2.microsoft.com/en-us/library/system.io.ports.aspx" \o "External link)** Namespace, and within that was the[**SerialPort**](http://msdn2.microsoft.com/en-us/library/system.io.ports.serialport.aspx) Class. DotNet developers finally had an intrinsic way of serial port communication, without having to deal with the complexities of interoping with an old legacy ActiveX OCX control. One of the most useful methods in the **SerialPort**class is the **[GetPortNames Method](http://msdn2.microsoft.com/en-us/library/system.io.ports.serialport.getportnames.aspx" \o "External link)**. This allows you to retrieve a list of available ports (COM1,COM2,etc.) available for the computer the application is running on.  
  
Now that we have that out of the way, lets move on to programming our application. As with all application I create, I keep functionality separated from presentation, I do this by creating *Manager* classes that manage the functionality for a given process. What we will be looking at is the code in my *CommunicationManager* class. As with anything you write in .Net you need to add the references to the Namespace's you'll be using:

|  |  |
| --- | --- |
| 1 | using System; |
| 2 | using System.Text; | |

|  |  |
| --- | --- |
| 3 | using System.Drawing; |
| 4 | using System.IO.Ports; | |

In this application I wanted to give the user the option of what format they wanted to send the message in, either string or binary, so we have an enumeration for that, and an enumerations for the type of message i.e; Incoming, Outgoing, Error, etc. The main purpose of this enumeration is for changing the color of the text displayed to the user according to message type. Here are the enumerations:

|  |  |  |
| --- | --- | --- |
| 01 | #region Manager Enums | |
| 02 | /// <summary> |

|  |  |  |
| --- | --- | --- |
| 03 | /// enumeration to hold our transmission types | |
| 04 | /// </summary> |

|  |  |  |
| --- | --- | --- |
| 05 | public enum TransmissionType { Text, Hex } | |
| 06 |  |

|  |  |
| --- | --- |
| 07 | /// <summary> |
| 08 | /// enumeration to hold our message types | |

|  |  |
| --- | --- |
| 09 | /// </summary> |
| 10 | public enum MessageType { Incoming, Outgoing, Normal, Warning, Error }; | |

|  |  |
| --- | --- |
| 11 | #endregion |

Next we have our variable list, 6 of them are for populating our class Properties, the other 2 are access throughout the class so they needed to be made global:

|  |  |  |
| --- | --- | --- |
| 01 | #region Manager Variables | |
| 02 | //property variables |

|  |  |  |
| --- | --- | --- |
| 03 | private string \_baudRate = string.Empty; | |
| 04 | private string \_parity = string.Empty; |

|  |  |
| --- | --- |
| 05 | private string \_stopBits = string.Empty; |
| 06 | private string \_dataBits = string.Empty; |

|  |  |  |
| --- | --- | --- |
| 07 | private string \_portName = string.Empty; | |
| 08 | private TransmissionType \_transType; |

|  |  |  |
| --- | --- | --- |
| 09 | private RichTextBox \_displayWindow; | |
| 10 | //global manager variables |

|  |  |  |
| --- | --- | --- |
| 11 | private Color[] MessageColor = { Color.Blue, Color.Green, Color.Black, Color.Orange, Color.Red }; | |
| 12 | private SerialPort comPort = new SerialPort(); |

|  |  |
| --- | --- |
| 13 | #endregion |

**NOTE:**I always separate my code into sections using the #region ... #endregion to make it easier when scanning my code. It is a design choice so it's not necessary if you don't want to do it.  
  
Now we need to create our class properties. All the properties in this class are public read/write properties. We have properties for the following items of the Serial Port:

* **Baud Rate:** A measure of the speed of serial communication, roughly equivalent to bits per second.
* **Parity:** The even or odd quality of the number of 1's or 0's in a binary code, often used to determine the integrity of data especially after transmission.
* **Stop Bits:** A bit that signals the end of a transmission unit
* **Data Bits:** The number of bits used to represent one character of data.
* **Port Name:** The port with which we're communicating through, i.e; COM1, COM2, etc.

We also have 2 properties that aren't related to the port itself, but with where the data will be displayed, and what transmission type to use:

|  |  |
| --- | --- |
| 01 | #region Manager Properties |

|  |  |
| --- | --- |
| 02 | /// <summary> |

|  |  |
| --- | --- |
| 03 | /// Property to hold the BaudRate |

|  |  |
| --- | --- |
| 04 | /// of our manager class |

|  |  |
| --- | --- |
| 05 | /// </summary> |

|  |  |
| --- | --- |
| 06 | public string BaudRate |

|  |  |
| --- | --- |
| 07 | { |

|  |  |
| --- | --- |
| 08 | get { return \_baudRate; } |

|  |  |
| --- | --- |
| 09 | set { \_baudRate = value; } |

|  |  |
| --- | --- |
| 10 | } |

|  |  |
| --- | --- |
| 11 |  |

|  |  |
| --- | --- |
| 12 | /// <summary> |

|  |  |
| --- | --- |
| 13 | /// property to hold the Parity |

|  |  |
| --- | --- |
| 14 | /// of our manager class |

|  |  |
| --- | --- |
| 15 | /// </summary> |

|  |  |
| --- | --- |
| 16 | public string Parity |

|  |  |
| --- | --- |
| 17 | { |

|  |  |
| --- | --- |
| 18 | get { return \_parity; } |

|  |  |
| --- | --- |
| 19 | set { \_parity = value; } |

|  |  |
| --- | --- |
| 20 | } |

|  |  |
| --- | --- |
| 21 |  |

|  |  |
| --- | --- |
| 22 | /// <summary> |

|  |  |
| --- | --- |
| 23 | /// property to hold the StopBits |

|  |  |
| --- | --- |
| 24 | /// of our manager class |

|  |  |
| --- | --- |
| 25 | /// </summary> |

|  |  |
| --- | --- |
| 26 | public string StopBits |

|  |  |
| --- | --- |
| 27 | { |

|  |  |
| --- | --- |
| 28 | get { return \_stopBits; } |

|  |  |
| --- | --- |
| 29 | set { \_stopBits = value; } |

|  |  |
| --- | --- |
| 30 | } |

|  |  |
| --- | --- |
| 31 |  |

|  |  |
| --- | --- |
| 32 | /// <summary> |

|  |  |
| --- | --- |
| 33 | /// property to hold the DataBits |

|  |  |
| --- | --- |
| 34 | /// of our manager class |

|  |  |
| --- | --- |
| 35 | /// </summary> |

|  |  |
| --- | --- |
| 36 | public string DataBits |

|  |  |
| --- | --- |
| 37 | { |

|  |  |
| --- | --- |
| 38 | get { return \_dataBits; } |

|  |  |
| --- | --- |
| 39 | set { \_dataBits = value; } |

|  |  |
| --- | --- |
| 40 | } |

|  |  |
| --- | --- |
| 41 |  |

|  |  |
| --- | --- |
| 42 | /// <summary> |

|  |  |
| --- | --- |
| 43 | /// property to hold the PortName |

|  |  |
| --- | --- |
| 44 | /// of our manager class |

|  |  |
| --- | --- |
| 45 | /// </summary> |

|  |  |
| --- | --- |
| 46 | public string PortName |

|  |  |
| --- | --- |
| 47 | { |

|  |  |
| --- | --- |
| 48 | get { return \_portName; } |

|  |  |
| --- | --- |
| 49 | set { \_portName = value; } |

|  |  |
| --- | --- |
| 50 | } |

|  |  |
| --- | --- |
| 51 |  |

|  |  |
| --- | --- |
| 52 | /// <summary> |

|  |  |
| --- | --- |
| 53 | /// property to hold our TransmissionType |

|  |  |
| --- | --- |
| 54 | /// of our manager class |

|  |  |
| --- | --- |
| 55 | /// </summary> |

|  |  |
| --- | --- |
| 56 | public TransmissionType CurrentTransmissionType |

|  |  |
| --- | --- |
| 57 | { |

|  |  |
| --- | --- |
| 58 | get{ return \_transType;} |

|  |  |
| --- | --- |
| 59 | set{ \_transType = value;} |

|  |  |
| --- | --- |
| 60 | } |

|  |  |
| --- | --- |
| 61 |  |

|  |  |
| --- | --- |
| 62 | /// <summary> |

|  |  |
| --- | --- |
| 63 | /// property to hold our display window |

|  |  |
| --- | --- |
| 64 | /// value |

|  |  |
| --- | --- |
| 65 | /// </summary> |

|  |  |
| --- | --- |
| 66 | public RichTextBox DisplayWindow |

|  |  |
| --- | --- |
| 67 | { |

|  |  |
| --- | --- |
| 68 | get { return \_displayWindow; } |

|  |  |
| --- | --- |
| 69 | set { \_displayWindow = value; } |

|  |  |
| --- | --- |
| 70 | } |

|  |  |
| --- | --- |
| 71 | #endregion |

To be able to instantiate any class object we create we need **Constructors**. Constructors are the entry point to your class, and is the first code executed when instantiating a class object. We have 2 constructors for our manager class, one that sets our properties to a specified value, and one that sets our properties to an empty value, thus initializing the variables preventing a **[NullReferenceException](http://msdn2.microsoft.com/en-us/library/system.nullreferenceexception(vs.80).aspx" \o "External link)** from occurring. We also add an **[EventHandler](http://msdn2.microsoft.com/en-us/library/system.io.ports.serialdatareceivedeventhandler.aspx" \o "External link)** in the constructor, the event will be executed whenever there's data waiting in the buffer:

|  |
| --- |
| #region Manager Constructors |

|  |  |
| --- | --- |
| 02 | /// <summary> |

|  |  |
| --- | --- |
| 03 | /// Constructor to set the properties of our Manager Class |

|  |  |
| --- | --- |
| 04 | /// </summary> |

|  |  |
| --- | --- |
| 05 | /// <param name="baud">Desired BaudRate</param> |

|  |  |
| --- | --- |
| 06 | /// <param name="par">Desired Parity</param> |

|  |  |
| --- | --- |
| 07 | /// <param name="sBits">Desired StopBits</param> |

|  |  |
| --- | --- |
| 08 | /// <param name="dBits">Desired DataBits</param> |

|  |  |
| --- | --- |
| 09 | /// <param name="name">Desired PortName</param> |

|  |  |
| --- | --- |
| 10 | public CommunicationManager(string baud, string par, string sBits, string dBits, string name, RichTextBox rtb) |

|  |  |
| --- | --- |
| 11 | { |

|  |  |
| --- | --- |
| 12 | \_baudRate = baud; |

|  |  |
| --- | --- |
| 13 | \_parity = par; |

|  |  |
| --- | --- |
| 14 | \_stopBits = sBits; |

|  |  |
| --- | --- |
| 15 | \_dataBits = dBits; |

|  |  |
| --- | --- |
| 16 | \_portName = name; |

|  |  |
| --- | --- |
| 17 | \_displayWindow = rtb; |

|  |  |
| --- | --- |
| 18 | //now add an event handler |

|  |  |
| --- | --- |
| 19 | comPort.DataReceived += new SerialDataReceivedEventHandler(comPort\_DataReceived); |

|  |  |
| --- | --- |
| 20 | } |

|  |  |
| --- | --- |
| 21 |  |

|  |  |
| --- | --- |
| 22 | /// <summary> |

|  |  |
| --- | --- |
| 23 | /// Comstructor to set the properties of our |

|  |  |
| --- | --- |
| 24 | /// serial port communicator to nothing |

|  |  |
| --- | --- |
| 25 | /// </summary> |

|  |  |
| --- | --- |
| 26 | public CommunicationManager() |

|  |  |
| --- | --- |
| 27 | { |

|  |  |
| --- | --- |
| 28 | \_baudRate = string.Empty; |

|  |  |
| --- | --- |
| 29 | \_parity = string.Empty; |

|  |  |
| --- | --- |
| 30 | \_stopBits = string.Empty; |

|  |  |
| --- | --- |
| 31 | \_dataBits = string.Empty; |

|  |  |
| --- | --- |
| 32 | \_portName = "COM1"; |

|  |  |
| --- | --- |
| 33 | \_displayWindow = null; |

|  |  |
| --- | --- |
| 34 | //add event handler |

|  |  |
| --- | --- |
| 35 | comPort.DataReceived+=new SerialDataReceivedEventHandler(comPort\_DataReceived); |

|  |  |
| --- | --- |
| 36 | } |

|  |  |
| --- | --- |
| 37 | #endregion |

The first think you need to know about serial port communication is writing data to the port. The first thing we do in our**WriteData** method is to check what transmission mode the user has selected, since binary data needs to be converted into binary, then back to string for displaying to the user. Next we need to make sure the port is open, for this we use the[**IsOpen Property**](http://msdn2.microsoft.com/en-us/library/system.io.ports.serialport.isopen.aspx) of the SerialPort Class. If the port isn't open we open it by calling the [**Open Method**](http://msdn2.microsoft.com/en-us/library/system.io.ports.serialport.open.aspx) of the SerialPort Class. For writing to the port we use the [**Write Method**](http://msdn2.microsoft.com/en-us/library/system.io.ports.serialport.write.aspx):

|  |  |
| --- | --- |
| 01 | #region WriteData |

|  |  |
| --- | --- |
| 02 | public void WriteData(string msg) |

|  |  |
| --- | --- |
| 03 | { |

|  |  |
| --- | --- |
| 04 | switch (CurrentTransmissionType) |

|  |  |
| --- | --- |
| 05 | { |

|  |  |
| --- | --- |
| 06 | case TransmissionType.Text: |

|  |  |
| --- | --- |
| 07 | //first make sure the port is open |

|  |  |
| --- | --- |
| 08 | //if its not open then open it |

|  |  |
| --- | --- |
| 09 | if (!(comPort.IsOpen == true)) comPort.Open(); |

|  |  |
| --- | --- |
| 10 | //send the message to the port |

|  |  |
| --- | --- |
| 11 | comPort.Write(msg); |

|  |  |
| --- | --- |
| 12 | //display the message |

|  |  |
| --- | --- |
| 13 | DisplayData(MessageType.Outgoing, msg + "\n"); |

|  |  |
| --- | --- |
| 14 | break; |

|  |  |
| --- | --- |
| 15 | case TransmissionType.Hex: |

|  |  |
| --- | --- |
| 16 | try |

|  |  |
| --- | --- |
| 17 | { |

|  |  |
| --- | --- |
| 18 | //convert the message to byte array |

|  |  |
| --- | --- |
| 19 | byte[] newMsg = HexToByte(msg); |

|  |  |
| --- | --- |
| 20 | //send the message to the port |

|  |  |
| --- | --- |
| 21 | comPort.Write(newMsg,0,newMsg.Length); |

|  |  |
| --- | --- |
| 22 | //convert back to hex and display |

|  |  |
| --- | --- |
| 23 | DisplayData(MessageType.Outgoing, ByteToHex(newMsg) + "\n"); |

|  |  |
| --- | --- |
| 24 | } |

|  |  |
| --- | --- |
| 25 | catch (FormatException ex) |

|  |  |
| --- | --- |
| 26 | { |

|  |  |
| --- | --- |
| 27 | //display error message |

|  |  |
| --- | --- |
| 28 | DisplayData(MessageType.Error, ex.Message); |

|  |  |
| --- | --- |
| 29 | } |

|  |  |
| --- | --- |
| 30 | finally |

|  |  |
| --- | --- |
| 31 | { |

|  |  |
| --- | --- |
| 32 | \_displaywindow.SelectAll(); |

|  |  |
| --- | --- |
| 33 | } |

|  |  |
| --- | --- |
| 34 | break; |

|  |  |
| --- | --- |
| 35 | default: |

|  |  |
| --- | --- |
| 36 | //first make sure the port is open |

|  |  |
| --- | --- |
| 37 | //if its not open then open it |

|  |  |
| --- | --- |
| 38 | if (!(comPort.IsOpen == true)) comPort.Open(); |

|  |  |
| --- | --- |
| 39 | //send the message to the port |

|  |  |
| --- | --- |
| 40 | comPort.Write(msg); |

|  |  |
| --- | --- |
| 41 | //display the message |

|  |  |
| --- | --- |
| 42 | DisplayData(MessageType.Outgoing, msg + "\n"); |

|  |  |
| --- | --- |
| 43 | break; |

|  |  |
| --- | --- |
| 44 | break; |

|  |  |
| --- | --- |
| 45 | } |

|  |  |
| --- | --- |
| 46 | } |

|  |  |
| --- | --- |
| 47 | #endregion |

You will notice in this method we call three methods:

* HexToByte
* ByteToHex
* DisplayData

These methods are required for this manager. The **HexToByte** method converts the data provided to binary format, then the **ByteToHex** converts it back to hex format for displaying. The last one, **DisplayData** is where we marshal a call to the thread that created the control for displaying the data, since UI controls can only be accessed by the thread that created them. First we'll look at converting the string provided to binary format:

|  |  |  |
| --- | --- | --- |
| 01 | #region HexToByte | |
| 02 | /// <summary> |

|  |  |  |
| --- | --- | --- |
| 03 | /// method to convert hex string into a byte array | |
| 04 | /// </summary> |

|  |  |  |
| --- | --- | --- |
| 05 | /// <param name="msg">string to convert</param> | |
| 06 | /// <returns>a byte array</returns> |

|  |  |  |
| --- | --- | --- |
| 07 | private byte[] HexToByte(string msg) | |
| 08 | { |

|  |  |  |
| --- | --- | --- |
| 09 | //remove any spaces from the string | |
| 10 | msg = msg.Replace(" ", ""); |

|  |  |  |
| --- | --- | --- |
| 11 | //create a byte array the length of the | |
| 12 | //string divided by 2 |

|  |  |
| --- | --- |
| 13 | byte[] comBuffer = new byte[msg.Length / 2]; |
| 14 | //loop through the length of the provided string | |

|  |  |
| --- | --- |
| 15 | for (int i = 0; i < msg.Length; i += 2) |
| 16 | //convert each set of 2 characters to a byte | |

|  |  |
| --- | --- |
| 17 | //and add to the array |
| 18 | comBuffer[i / 2] = (byte)Convert.ToByte(msg.Substring(i, 2), 16); | |

|  |  |  |
| --- | --- | --- |
| 19 | //return the array | |
| 20 | return comBuffer; |

|  |  |
| --- | --- |
| 21 | } |
| 22 | #endregion | |

Here we convert the provided string to a byte array, then the **WriteData** method sends it out the port. For displaying we need to convert it back into string format, so we use the **ByteToHex** method we created:

|  |  |  |
| --- | --- | --- |
| 01 | #region ByteToHex | |
| 02 | /// <summary> |

|  |  |  |
| --- | --- | --- |
| 03 | /// method to convert a byte array into a hex string | |
| 04 | /// </summary> |

|  |  |  |
| --- | --- | --- |
| 05 | /// <param name="comByte">byte array to convert</param> | |
| 06 | /// <returns>a hex string</returns> |

|  |  |  |
| --- | --- | --- |
| 07 | private string ByteToHex(byte[] comByte) | |
| 08 | { |

|  |  |
| --- | --- |
| 09 | //create a new StringBuilder object |
| 10 | StringBuilder builder = new StringBuilder(comByte.Length \* 3); | |

|  |  |  |
| --- | --- | --- |
| 11 | //loop through each byte in the array | |
| 12 | foreach (byte data in comByte) |

|  |  |
| --- | --- |
| 13 | //convert the byte to a string and add to the stringbuilder |
| 14 | builder.Append(Convert.ToString(data, 16).PadLeft(2, '0').PadRight(3, ' ')); | |

|  |  |
| --- | --- |
| 15 | //return the converted value |
| 16 | return builder.ToString().ToUpper(); | |

|  |  |
| --- | --- |
| 17 | } |
| 18 | #endregion | |

The last method that **WriteData** depends on is the **DisplayData** method. Here we use the [**Invoke Method**](http://msdn2.microsoft.com/en-us/library/zyzhdc6b.aspx) of our RichTextBox, the control used to display the data, to create a new EventHandler which creates a new [**Delegate**](http://msdn2.microsoft.com/en-us/library/900fyy8e(VS.80).aspx) for setting the properties we wish for our message, then appending it to the value already displayed:

|  |  |  |
| --- | --- | --- |
| 01 | #region DisplayData | |
| 02 | /// <summary> |

|  |  |  |
| --- | --- | --- |
| 03 | /// method to display the data to & from the port | |
| 04 | /// on the screen |

|  |  |
| --- | --- |
| 05 | /// </summary> |
| 06 | /// <param name="type">MessageType of the message</param> | |

|  |  |  |
| --- | --- | --- |
| 07 | /// <param name="msg">Message to display</param> | |
| 08 | [STAThread] |

|  |  |  |
| --- | --- | --- |
| 09 | private void DisplayData(MessageType type, string msg) | |
| 10 | { |

|  |  |  |
| --- | --- | --- |
| 11 | \_displaywindow.Invoke(new EventHandler(delegate | |
| 12 | { |

|  |  |
| --- | --- |
| 13 | \_displaywindow.SelectedText = string.Empty; |
| 14 | \_displaywindow.SelectionFont = new Font(\_displaywindow.SelectionFont, FontStyle.Bold); | |

|  |  |  |
| --- | --- | --- |
| 15 | \_displaywindow.SelectionColor = MessageColor[(int)type]; | |
| 16 | \_displaywindow.AppendText(msg); |

|  |  |  |
| --- | --- | --- |
| 17 | \_displaywindow.ScrollToCaret(); | |
| 18 | })); |

|  |  |
| --- | --- |
| 19 | } |
| 20 | #endregion | |

**NOTE:** You will notice that we hyave added the **[STAThread Attribute](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemstathreadattributeclasstopic.asp" \o "External link)** to our method. This is used when a single thread apartment is required by a control, like the RichTextBox.  
  
The next method we will look at it used when we need to open the port initially. Here we set the BaudRate, Parity, StopBits, DataBits and PortName Properties of the SerialPort Class:

|  |  |
| --- | --- |
| 01 | #region OpenPort |

|  |  |
| --- | --- |
| 02 | public bool OpenPort() |

|  |  |
| --- | --- |
| 03 | { |

|  |  |
| --- | --- |
| 04 | try |

|  |  |
| --- | --- |
| 05 | { |

|  |  |
| --- | --- |
| 06 | //first check if the port is already open |

|  |  |
| --- | --- |
| 07 | //if its open then close it |

|  |  |
| --- | --- |
| 08 | if (comPort.IsOpen == true) comPort.Close(); |

|  |  |
| --- | --- |
| 09 |  |

|  |  |
| --- | --- |
| 10 | //set the properties of our SerialPort Object |

|  |  |
| --- | --- |
| 11 | comPort.BaudRate = int.Parse(\_baudRate);    //BaudRate |

|  |  |
| --- | --- |
| 12 | comPort.DataBits = int.Parse(\_dataBits);    //DataBits |

|  |  |
| --- | --- |
| 13 | comPort.StopBits = (StopBits)Enum.Parse(typeof(StopBits),\_stopBits);    //StopBits |

|  |  |
| --- | --- |
| 14 | comPort.Parity = (Parity)Enum.Parse(typeof(Parity),\_parity);    //Parity |

|  |  |
| --- | --- |
| 15 | comPort.PortName = \_portName;   //PortName |

|  |  |
| --- | --- |
| 16 | //now open the port |

|  |  |
| --- | --- |
| 17 | comPort.Open(); |

|  |  |
| --- | --- |
| 18 | //display message |

|  |  |
| --- | --- |
| 19 | DisplayData(MessageType.Normal, "Port opened at " + DateTime.Now + "\n"); |

|  |  |
| --- | --- |
| 20 | //return true |

|  |  |
| --- | --- |
| 21 | return true; |

|  |  |
| --- | --- |
| 22 | } |

|  |  |
| --- | --- |
| 23 | catch (Exception ex) |

|  |  |
| --- | --- |
| 24 | { |

|  |  |
| --- | --- |
| 25 | DisplayData(MessageType.Error, ex.Message); |

|  |  |
| --- | --- |
| 26 | return false; |

|  |  |
| --- | --- |
| 27 | } |

|  |  |
| --- | --- |
| 28 | } |

|  |  |
| --- | --- |
| 29 | #endregion |

Next lets take a look at our event handler. This event will be executed whenever there's data waiting in the buffer. This method looks identical to our **WriteData** method, because it has to do the same exact work:

|  |  |
| --- | --- |
| 01 | #region comPort\_DataReceived |

|  |  |
| --- | --- |
| 02 | /// <summary> |

|  |  |
| --- | --- |
| 03 | /// method that will be called when theres data waiting in the buffer |

|  |  |
| --- | --- |
| 04 | /// </summary> |

|  |  |
| --- | --- |
| 05 | /// <param name="sender"></param> |

|  |  |
| --- | --- |
| 06 | /// <param name="e"></param> |

|  |  |
| --- | --- |
| 07 | void comPort\_DataReceived(object sender, SerialDataReceivedEventArgs e) |

|  |  |
| --- | --- |
| 08 | { |

|  |  |
| --- | --- |
| 09 | //determine the mode the user selected (binary/string) |

|  |  |
| --- | --- |
| 10 | switch (CurrentTransmissionType) |

|  |  |
| --- | --- |
| 11 | { |

|  |  |
| --- | --- |
| 12 | //user chose string |

|  |  |
| --- | --- |
| 13 | case TransmissionType.Text: |

|  |  |
| --- | --- |
| 14 | //read data waiting in the buffer |

|  |  |
| --- | --- |
| 15 | string msg = comPort.ReadExisting(); |

|  |  |
| --- | --- |
| 16 | //display the data to the user |

|  |  |
| --- | --- |
| 17 | DisplayData(MessageType.Incoming, msg + "\n"); |

|  |  |
| --- | --- |
| 18 | break; |

|  |  |
| --- | --- |
| 19 | //user chose binary |

|  |  |
| --- | --- |
| 20 | case TransmissionType.Hex: |

|  |  |
| --- | --- |
| 21 | //retrieve number of bytes in the buffer |

|  |  |
| --- | --- |
| 22 | int bytes = comPort.BytesToRead; |

|  |  |
| --- | --- |
| 23 | //create a byte array to hold the awaiting data |

|  |  |
| --- | --- |
| 24 | byte[] comBuffer = new byte[bytes]; |

|  |  |
| --- | --- |
| 25 | //read the data and store it |

|  |  |
| --- | --- |
| 26 | comPort.Read(comBuffer, 0, bytes); |

|  |  |
| --- | --- |
| 27 | //display the data to the user |

|  |  |
| --- | --- |
| 28 | DisplayData(MessageType.Incoming, ByteToHex(comBuffer) + "\n"); |

|  |  |
| --- | --- |
| 29 | break; |

|  |  |
| --- | --- |
| 30 | default: |

|  |  |
| --- | --- |
| 31 | //read data waiting in the buffer |

|  |  |
| --- | --- |
| 32 | string str = comPort.ReadExisting(); |

|  |  |
| --- | --- |
| 33 | //display the data to the user |

|  |  |
| --- | --- |
| 34 | DisplayData(MessageType.Incoming, str + "\n"); |

|  |  |
| --- | --- |
| 35 | break; |

|  |  |
| --- | --- |
| 36 | } |

|  |  |
| --- | --- |
| 37 | } |

|  |  |
| --- | --- |
| 38 | #endregion |

We have 3 small methods left, and these are actually optional, for the lack of a better word. These methods are used to populate my ComboBox's on my UI with the port names available on the computer, Parity values and Stop Bit values. The Parity and Stop Bits are available in enumerations included with the .Net Framework 2.0:

* [**Parity Enumeration**](http://msdn2.microsoft.com/en-us/library/system.io.ports.parity.aspx)
* [**StopBits Enumeration**](http://msdn2.microsoft.com/en-us/library/system.io.ports.stopbits.aspx)

|  |  |
| --- | --- |
| 01 | #region SetParityValues |

|  |  |
| --- | --- |
| 02 | public void SetParityValues(object obj) |

|  |  |
| --- | --- |
| 03 | { |

|  |  |
| --- | --- |
| 04 | foreach (string str in Enum.GetNames(typeof(Parity))) |

|  |  |
| --- | --- |
| 05 | { |

|  |  |
| --- | --- |
| 06 | ((ComboBox)obj).Items.Add(str); |

|  |  |
| --- | --- |
| 07 | } |

|  |  |
| --- | --- |
| 08 | } |

|  |  |
| --- | --- |
| 09 | #endregion |

|  |  |
| --- | --- |
| 10 |  |

|  |  |
| --- | --- |
| 11 | #region SetStopBitValues |

|  |  |
| --- | --- |
| 12 | public void SetStopBitValues(object obj) |

|  |  |
| --- | --- |
| 13 | { |

|  |  |
| --- | --- |
| 14 | foreach (string str in Enum.GetNames(typeof(StopBits))) |

|  |  |
| --- | --- |
| 15 | { |

|  |  |
| --- | --- |
| 16 | ((ComboBox)obj).Items.Add(str); |

|  |  |
| --- | --- |
| 17 | } |

|  |  |
| --- | --- |
| 18 | } |

|  |  |
| --- | --- |
| 19 | #endregion |

|  |  |
| --- | --- |
| 20 |  |

|  |  |
| --- | --- |
| 21 | #region SetPortNameValues |

|  |  |
| --- | --- |
| 22 | public void SetPortNameValues(object obj) |

|  |  |
| --- | --- |
| 23 | { |

|  |  |
| --- | --- |
| 24 |  |

|  |  |
| --- | --- |
| 25 | foreach (string str in SerialPort.GetPortNames()) |

|  |  |
| --- | --- |
| 26 | { |

|  |  |
| --- | --- |
| 27 | ((ComboBox)obj).Items.Add(str); |

|  |  |
| --- | --- |
| 28 | } |

|  |  |
| --- | --- |
| 29 | } |

|  |  |
| --- | --- |
| 30 | #endregion |

That is how you do Serial Port Communication in C#. Microsoft finally gave us intrinsic tools to perform this task, no more relying on legacy objects. I am providing this class and a sample application to show how to implement what we just learned. What I am providing is under the **GNU General Public License** meaning you can modify and distribute how you see fit, but the license header must stay in tact. I hope you found this tutorial useful and informative, thank you for reading.  
  
Happy Coding   
  
**[Attached File](http://www.dreamincode.net/forums/index.php?app=core&module=attach&section=attach&attach_id=5407)**  [**SerialPortCommunication.zip**](http://www.dreamincode.net/forums/index.php?app=core&module=attach&section=attach&attach_id=5407) **(101.85K)**   
Number of downloads: 126343