Serial Comms in C# for Beginners

glennPattonInThePUB, 6 Nov 2013 CPOL

Rate:

4.98 (93 votes)

Serial communication in C#.

Download source

Introduction

Serial Communication between PCs is always seen as the starting point. My first piece of serial comms code was a University assignment getting two PC's, then three, then... you get the idea talking on (what I now know to be) an RS422 with its pair of cables using Borland's Turbo C++ for DOS. Time passed I graduated (somehow!) and I got to be a Design Engineer in the 'real' world. Not looking busy enough got me lumbered with the famous "Glenn you aren't that busy, have a look at this". The 'this' was taking on a project that had been half done and then the guilty party ran. So to make it work the way it was wanted, out came the software (VB6) and there goes any social life I had.

Background

RS-232 is the best know method of PC Communications, the characteristics of RS232 is a logic 1 (true) is can range from -3v to -25v a logic 0 (false) +3v to +25v. The area of -3v to 0 to +3v is taken as not valid to allow for noise and interference on the line. If the port is idle the port is at high level (or -12v), which is why if you look at circuit diagrams of peripherals there is always to a lot of invertors. RS-485 detailed at the end, RS-422 & RS-449 and on to allow very long cable runs and high speeds all follow the same basic path.

VB6 used *MSComm32.ocx*; this was a 32 bit version of *MSComm16.ocx* both of these components worked fairly well. I say fairly as the interrupt handler of the control `CommEvents` never worked as quickly or as well as I was expecting it to. I found the only sure fire way was to poll it with a loop or timer (and possibly lock the system, bless `DoEvents`!) another point worth noting was the limit of 16 Serial Ports (I have used 9 on a test rig using MSComm and was getting a bit worried in case there

was another set of devices!) Thankfully the Net Serial Class lifted it 255. This all changed with the release of .NET1 serial comms were over looked (for reasons that were not clear, I presume it was something to do with security). This gave two ways around it this gave the option of using Dos to create and print to a (virtual) device or import the OCX (this was in the height of .NET-COM war). I did do a program for communicating to a high definition video screen using MScomm in Borland C++ Builder (I got it working in the end Cliff! Ha!!) this led to a large executable so not suitable for most applications. Once .NET 2.0 was released I gave a sigh of relief as this came with a serial port class native to the frame work. I started to use C# as well as VB then dropped VB6 all together.

Serial Ports have now adopted (and in some cases dropped) the 9 way D-type, though I have come across older kit (notably high precision colorimeter cameras from Samsung) use the older 25 D-type.

Hide Copy Code

```
Pin Number (9 way)    Pin Number (25way)   Function
   1                     8                    Carrier Detect
   2                     3                    Receive
   3                     2                    Transmit
   4                     20                   Data Terminal Ready
   5                     7                    Signal Ground
   6                     6                    Data Set Ready
   7                     4                    Request To Send
   8                     5                    Clear To Send
    9                    21                   Ring Indicator
(1,9-19,21,  23-25 are unused for RS232)
```

A common test to see if a cable or port is correct is to use a Terminal Emulator and short pins 2 & 3 to see if you get keyboard presses for characters.

Using the code

Setting Up

Being a hardware guy primarily (I have a selection of burns and scars that cause US immigration fun every time!) I learnt C and Assembly. I didn't do any of this Windows malarkey until I got to the real world so forgive any stumbles I might make.

Hide Copy Code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.IO.Ports;
using System.Windows.Forms;
```

I always treat the using section like I would the #include section of a C program.

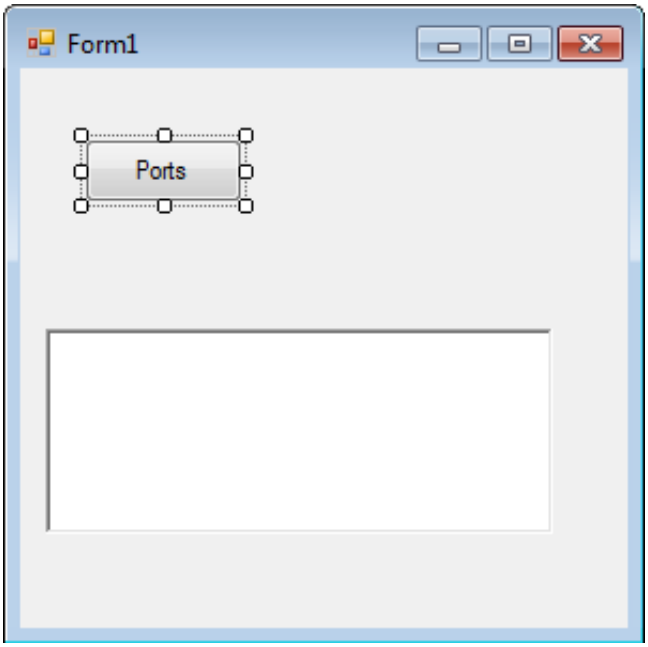
`System.IO.Ports` is the class to use with out resulting to low level hacking. This covers all the serial ports that appear on the machine.

Hide Copy Code

```
SerialPort ComPort = new SerialPort;
```

This will create an object called `ComPort`. This will create a serial port object with the following parameters as default 9600bps, no parity, one stop bit and no flow control.

Shown below is the form:



I have created a standard Windows Forms Application via File menu. To this I have added the button (name Ports) and a Rich Text Box.

The button I have called `btnGetSerialPorts` and the Rich Text Box I have called `rtbIncomingData` (the name will become apparent later).

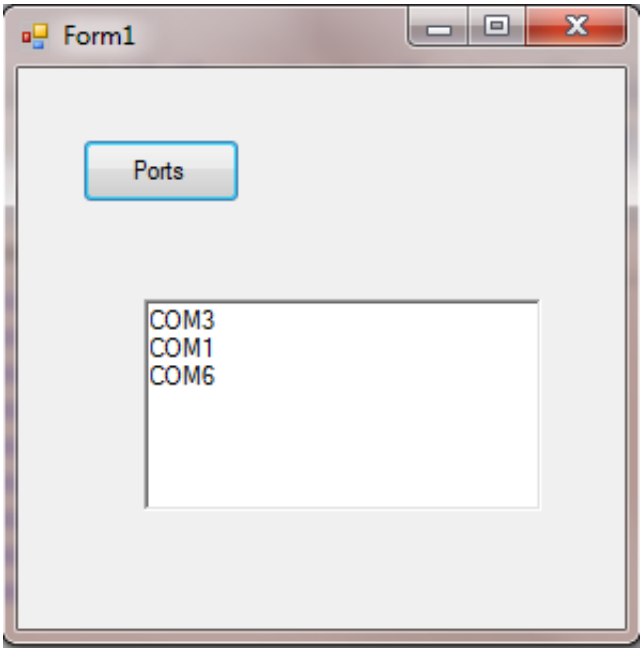
I tend to use the rich text box as it is more flexible than the ordinary text box. Its uses for sorting and aligning text are considerably more than the straight textbox.

To the button's click routine I have added the following code:

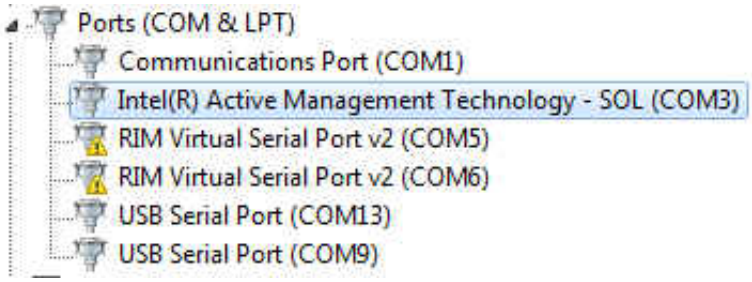
Hide Copy Code

```
private void btnGetSerialPorts_Click(object sender, EventArgs e)
{
    string[] ArrayComPortsNames = null;
    int index = -1;
    string ComPortName = null;

    ArrayComPortsNames = SerialPort.GetPortNames();
    do
    {
         index += 1;
        rtbIncoming.Text += ArrayComPortsNames[index]+"\n";
    }
    while (!((ArrayComPortsNames[index] == ComPortName) ||
                        (index == ArrayComPortsNames.GetUpperBound(0))));
}
```



This shows all the devices that appear as com ports, a mistake to make is thinking that a device if plugged into the USB will appear as a COM Port. For instance, Com5 and Com6 are in my phone and below is shown the Device Manager screen with the Ports COM and LPT option expanded:



Devices will appear here there manufactures (RIM, Intel etc.).

COM1 if your machine happens to have one, these days in Net books, Laptops other machines that are transportable the COM1 as a 9way D –type male is getting less common once there was a Com 1 and some times a Com2 fitted (My first Pentium, Windows 95 had two!). As can be seen the ports are not in sequential order these can be ordered. I tend to use Combo Boxes for loading the data into (I think it looks more professional and harder for users to get wrong) and sorting as below:

Hide Copy Code

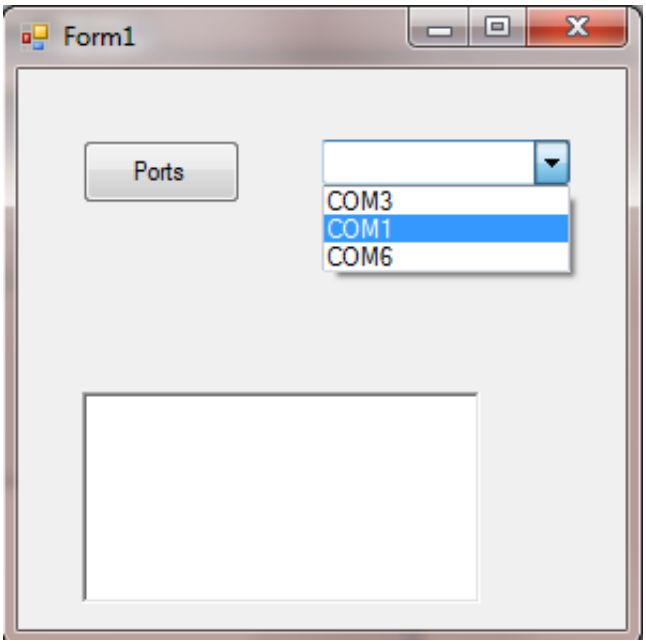
```
ArrayComPortsNames = SerialPort.GetPortNames();
do
{
    index += 1;
    cboPorts.Items.Add(ArrayComPortsNames[index]);
}

while (!((ArrayComPortsNames[index] == ComPortName)
              || (index == ArrayComPortsNames.GetUpperBound(0))));
Array.Sort(ArrayComPortsNames);

//want to get first out
if (index == ArrayComPortsNames.GetUpperBound(0))
{
    ComPortName = ArrayComPortsNames[0];
}
cboPorts.Text = ArrayComPortsNames[0];
```

If you notice the changes that have been made to the click routine a standard Combo Box now accepts the array rather than the Rich Text Box, also the `Array.Sort(ArrayComPortsNames)`.

Sort to get first COM port into the combo box (I have called Ports) these changes now give the below:



The baud rate is the amount of possible events that can happen in a second. It is displays usually as a number of bit per second, the possible number that can be used are 300, 600, 1200, 2400, 9600, 14400, 19200, 38400, 57600, and 115200 (these come from the UAR 8250 chip is used, if a 16650 the additional rates of 230400, 460800 and 921600) .

These are supported by the Serial Port class and come from the teletype machines of old. Non-standard baud rates are a no-no however if you can find it MHComm32.ocx from a company called Elltech did provide a means of creating 'custom' baud rates.

I used it in a application once to give a rate of 9550 when the third party board wouldn't take 9600 without some errors occurring. The standard baud rate for connections is 9600 (the default for the serial port class) the lower baud rates 600 & 300 are for connecting to embedded processors & microcontrollers. Sometimes to limit board complexity and size (and of course cost!) the device has to act as the UART (Universal Asynchronous Receiver & Transmitter). The UART chip handles the bulk of the serial communications for instance the 16550 used in PCs has eight pins for data and various pins for status and control the average embedded system does not have the available pins to control this chip. So to get around this fact the processor is programmed to emulate a UART as the processor could well be busy with other tasks and serial comms is a low priority the slower baud rates were came to be used. Below is the code for the application's Baud Rate called cboBaudRate.

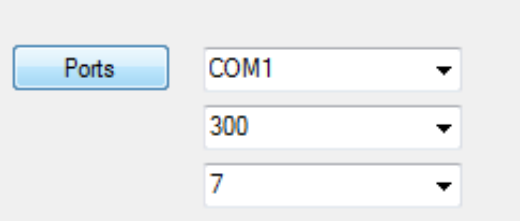
Hide Copy Code

```
cboBaudRate.Items.Add(300);
cboBaudRate.Items.Add(600);
cboBaudRate.Items.Add(1200);
cboBaudRate.Items.Add(2400);
cboBaudRate.Items.Add(9600);
cboBaudRate.Items.Add(14400);
cboBaudRate.Items.Add(19200);
cboBaudRate.Items.Add(38400);
cboBaudRate.Items.Add(57600);
cboBaudRate.Items.Add(115200);
cboBaudRate.Items.ToString();
//get first item print in text
cboBaudRate.Text = cboBaudRate.Items[0].ToString();
```

When the button is clicked will give all of the baud rates in the combo box with 300 the lowest in the text of the box.

The next box is the number of Data bits, these represent the total number of transitions of the data transmission (or Tx line) 8 is the standard (8 is useful for reading certain embedded application as it gives two nibbles (4 bit sequences).

The inclusion of 7 bits tends to be used in some RS485 where the extra bit is used for a global message. With the above mods the form represents.



The next several commands require a bit more explanation in my opinion. They are the Stop bit, Parity and Handshaking.

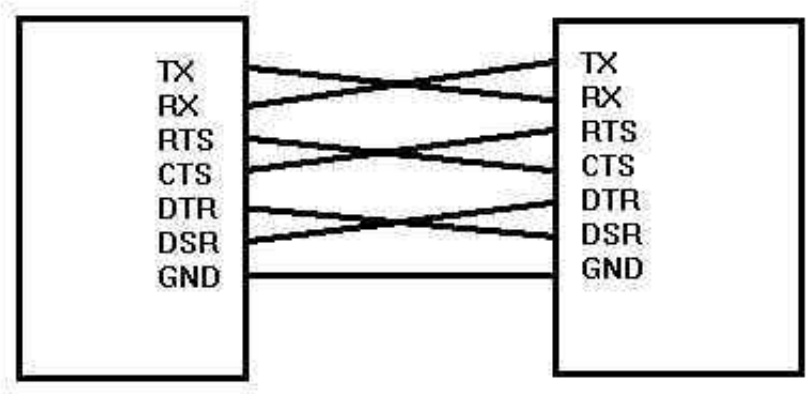
The Handshaking property is used when a full set of connections are used (such as the grey 9 way D-types that litter my desk). It was used originally to ensure both ends lined up with each other and the data was sent and received properly. A common handshake was required between both sender and receiver. Below is the code for the combo box:

Hide Copy Code

```
cboHandShaking.Items.Add("None");
cboHandShaking.Items.Add("XOnXOff");
cboHandShaking.Items.Add("RequestToSend");
cboHandShaking.Items.Add("RequestToSendXOnXOff");
```

I think now would be a suitable time to give a bit more detail on what each property does and how it works. RS-232 is intended to be a standard, but as is the way “rules are for the abeyance of fools and the guidance of the wise”. The transmission of data requires the Handshaking property to be set to one of the settings above. To allow this to work the CTS clear to send (pin 8 of the 9-way), RTS request (ready) to send (pin 7 of the 9 way) which is directly controllable from software, DTR data terminal ready (pin 4 of the 9-way) again this is directly controllable from software and DSR data set ready is similar to the CTS pin but is activate by other end of the connection.

Handshaking can be none (which is the most common these days) the handshaking was originally used when the speeds were lower and it would cost more time to resend the data, it is used also if a large amount of data is to be sent to ensure it has been correctly received. Detailed above are the available settings from the serial port class. The diagram below shows how to connect up two 9 way D-types for full duplex communication.



If the Handshake property is set to None the DTR and RTS pins are then freed up for the common use of Power, the PC on which this is being typed gives +10.99 volts on the DTR pin & +10.99 volts again on the RTS pin if set to true. If set to false it gives -9.95 volts on the DTR, -9.94 volts on the RTS. These values are between +3 to +25 and -3 to -25 volts this give a dead zone to allow for noise immunity.

This switching can be achieved by using the below:

Hide Copy Code

```
ComPort.RtsEnable = true;
ComPort.DtrEnable = true;
```

If the true property is replaced with false the switch over of the voltages can be seen.

These values were measured on the PC being used to write this article using a mulitmeter set to volts mode and Pin 5 of the 9 way as ground. I found when I was learning software communications too much time was spent explaining the theory not enough was doing. The program below shows some modifications to the code I am developing here, four labels called: `lblBreakStatus`, `lblCTSStatus`, `lblDTRStatus`, and `lblRIStatus`.

`lblCTSStatus` and `lblDTRStatus` show the states of the CTS line (which in the above diagram is connected to RTS line) and DTR line (connected to the DTR line in full duplex comms).

A button called `btnTest` is created and the click routine is as follows (this code is included in the listing and example project but commented out uncomment to use!):

Hide Copy Code

```
private void btnTest_Click(object sender, EventArgs e)
{
   SerialPinChangedEventHandler1 = new SerialPinChangedEventHandler(PinChanged);
    ComPort.PinChanged += SerialPinChangedEventHandler1;
    ComPort.Open();
    ComPort.RtsEnable = true;
    ComPort.DtrEnable = true;
    btnTest.Enabled = false;
}
```

I always, as a habit, prevent the user from clicking on the open button again which will cause an exception (generally I change the text and check which it is Open / Closed...) . The RS232 connection has a wire from the DTR or RTS pin and makes contact with the pins (shorting them) it causes the labels back grounds to change like so:

This gives a quick demo of how the pins work. Below is the code:

Hide Shrink ▲ Copy Code

```
internal void PinChanged(object sender, SerialPinChangedEventArgs e)
{
    SerialPinChange SerialPinChange1 = 0;
    bool signalState = false;
```

```
     SerialPinChange1 = e.EventType;
     lblCTSStatus.BackColor = Color.Green;
    lblDSRStatus.BackColor = Color.Green;
    lblRIStatus.BackColor = Color.Green;
    lblBreakStatus.BackColor = Color.Green;

         switch (SerialPinChange1)
    {
        case SerialPinChange.Break:
            lblBreakStatus.BackColor = Color.Red;
            //MessageBox.Show("Break is Set");
            break;
        case SerialPinChange.CDChanged:
            signalState = ComPort.CtsHolding;
       //  MessageBox.Show("CD = " + signalState.ToString());
           break;
        case SerialPinChange.CtsChanged:
             signalState = ComPort.CDHolding;
            lblCTSStatus.BackColor = Color.Red;
          //MessageBox.Show("CTS = " + signalState.ToString());
             break;
        case SerialPinChange.DsrChanged:
            signalState = ComPort.DsrHolding;
            lblDSRStatus.BackColor = Color.Red;
      // MessageBox.Show("DSR = " + signalState.ToString());
            break;
        case SerialPinChange.Ring:
           lblRIStatus.BackColor = Color.Red;
       //MessageBox.Show("Ring Detected");
            break;
    }
}
```

To get this functionality the following lines should be added below the SerialPort declaration:

Hide Copy Code

```
internal delegate void SerialPinChangedEventHandlerDelegate(object sender, SerialPinChangedEventArgs
e);
private SerialPinChangedEventHandler SerialPinChangedEventHandler1;
```

This will create the objects needed the line below should be added to the form declaration below `InitializeComponents();`:

Hide Copy Code

```
SerialPinChangedEventHandler1 = new SerialPinChangedEventHandler(PinChanged);
```

This declares the delegate `PinChanged`. In the button click:

Add:

Hide Copy Code

```
SerialPinChangedEventHandler1 = new SerialPinChangedEventHandler(PinChanged);
```

And attach the delegate as so

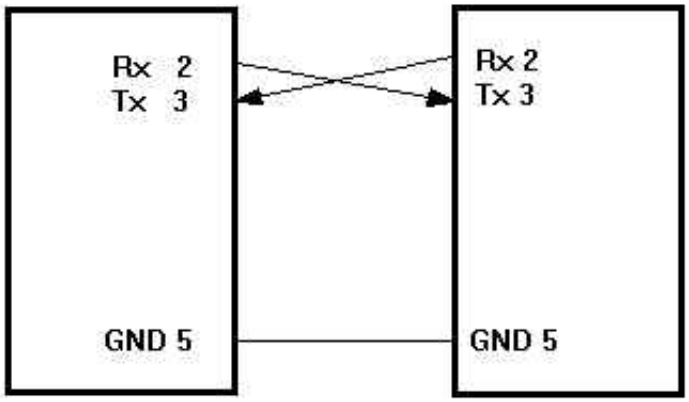
Hide Copy Code

```
ComPort.PinChanged += SerialPinChangedEventHandler1;
```

This now causes the program when either DTR or RTS is connected to the pins to change the labels back colour. I have found it easiest to run this program with a female socket of the end of a 9 way cable. A piece of wire can then be stroked over the pin

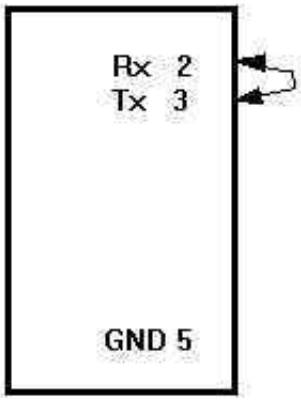
solder buckets (casing the labels to change colour from Green to Red and back), this will give some feeling that you achieving something. It should be noted at this point the PC will try to open the default serial port which is default is 9600 Baud, No Parity, One Stop bit and No flow control, the number is decided by the lowest number serial available Com1 if fitted.

To get the simplest means of comms between two units the below is used:



All this does is connect Pin 2 of one RS-232 port to Pin 3 of another, and the Pin 2 to Pin 3 of the other end, while this is all that is needed a connection between the shield grounds is recommended to prevent errors.

The simplest of all connections is below:



Pins 2 and 3 are directly connected (tip: the pin spacing on a standard 9-way D type is just right to all a jumper from the back of an old CD-Rom to be used!)

This type of connection is used often to see if the program receives and sends data.

To get the demo codes to do this add a button to the form called `PortState` and add the code below:

Hide Copy Code

```
private void btnPortState_Click(object sender, EventArgs e)
{
    if (btnPortState.Text == "Closed")
    {
        btnPortState.Text = "Open";
        ComPort.PortName = Convert.ToString(cboPorts.Text);
        ComPort.BaudRate = Convert.ToInt32(cboBaudRate.Text);
        ComPort.DataBits = Convert.ToInt16(cboDataBits.Text);
        ComPort.StopBits = (StopBits)Enum.Parse(typeof(StopBits), cboStopBits.Text);
        ComPort.Handshake = (Handshake)Enum.Parse(typeof(Handshake), cboHandShaking.Text);
        ComPort.Parity = (Parity)Enum.Parse(typeof(Parity), cboParity.Text);
        ComPort.Open();
    }
    else if (btnPortState.Text == "Open")
    {
        btnPortState.Text = "Closed";
        ComPort.Close();
    }
```

The code once the button is clicked changes the text of the button to “Open” and then sets the various attributes of the com port individually, I set all my Com ports in this way as it saves staring at the code trying to work out was is happening.

A very good practice to get into is to use the `IsOpen` property to check to see if the port can be opened. The main use for this I have found is with USB com ports which can be added and removed at will (and not always placed back in the same socket) an example of this is below:

Hide Copy Code

```
if(!(ComPort.IsOpen))
{
}
```

This if will check the COM port to see if it is open, the not operator is to check if it is closed another useful trick is to use a try... catch as below:

Hide Copy Code

```
try
{
    ComPort.Open();
}
catch(UnauthorizedAccessException ex)
{
    MessageBox.Show(ex.Message);
}
```

This will cause the software to produce a message when a fault is found and not blow up in the users face! Also another thing the Serial Port Class provides is the Timeout Property, to prevent the software sitting waiting in a loop you can do the below:

Hide Copy Code

```
ComPort.ReadTimeout(4000);
ComPort.WriteTimeout(6000);
```

The above sets a time limit on actions to the serial port by default these are set to infinity but can be set. I have used the Read time out with some radio boards to check they are still in range (if the command to get the serial number of board X times out X is not in range).

To received data there are two methods, interrupt using the `DataRecieved` Event and polling with a timer (not recommend) but using a stop watch timer from the tool box with a interval property of 1000 or 1 second (see below for code)...

Hide Copy Code

```
private void tmrPollForRecivedData_Tick(object sender, EventArgs e)
```

```
{
    String RecievedData;
    RecievedData = ComPort.ReadExisting();
    if (!(RecievedData == ""))
    {
        rtbIncoming.Text += RecievedData;
    }
}
```

To start the timer use the line:

Hide Copy Code

```
tmrPollForRecivedData.Enable = true;
```

to stop it use:

Hide Copy Code

```
tmrPollForRecivedData.Enable = false;
```

Generally this method while it works for a single serial port that does not change quickly it works, however I have been bitten by this with a test rig someone (I think I know who!!) altered the values in the software of the intervals to make it run quicker and it went very badly wrong (who got the blame!) as the timer component in can be over taken and ignored by Windows.

These are both valid methods however the `DataReceived` event is acted upon an interrupt basis using a delegate. To get this function to work add the below:

Hide Copy Code

```
delegate void SetTextCallback(string text);
string InputData = String.Empty;
```

The code creates a delegate `SetTextCallback` and an empty string called `InputData`.

Next the event handler is added to the code

Hide Copy Code

```
private void port_DataReceived_1(object sender, SerialDataReceivedEventArgs e)
{
   InputData = ComPort.ReadExisting();
   if (InputData != String.Empty)
   {
      this.BeginInvoke(new SetTextCallback(SetText), new object[] { InputData });
   }
}
```

The `SetText` which is called by the `SetTextCallback`.

Hide Copy Code

```
 private void SetText(string text)
{
    this.rtbIncoming.Text += text;
}
```

A button is created on the form called Hello, the click routine is shown below

Hide Copy Code

```
private void btnHello_Click(object sender, EventArgs e)
{
    ComPort.Write("Hello World!");
}
```

With the above button we can send the famous (or infamous!) message Hello World.

This is done with use of the `.Write()` function it can write a byte array, a character array or subarray or a string. A relation of this is the `WriteLine()` which will send a string and a new line (or \n to C programmers) and the `WriteByte` which sends a Byte out. These methods are 'blocking' in that while they are sending they cannot receive.

The reading method is `ReadExisting()` method this will read text out of the receive buffer until it is empty. This method does not block the port until it gets some data or a timeout is expired.

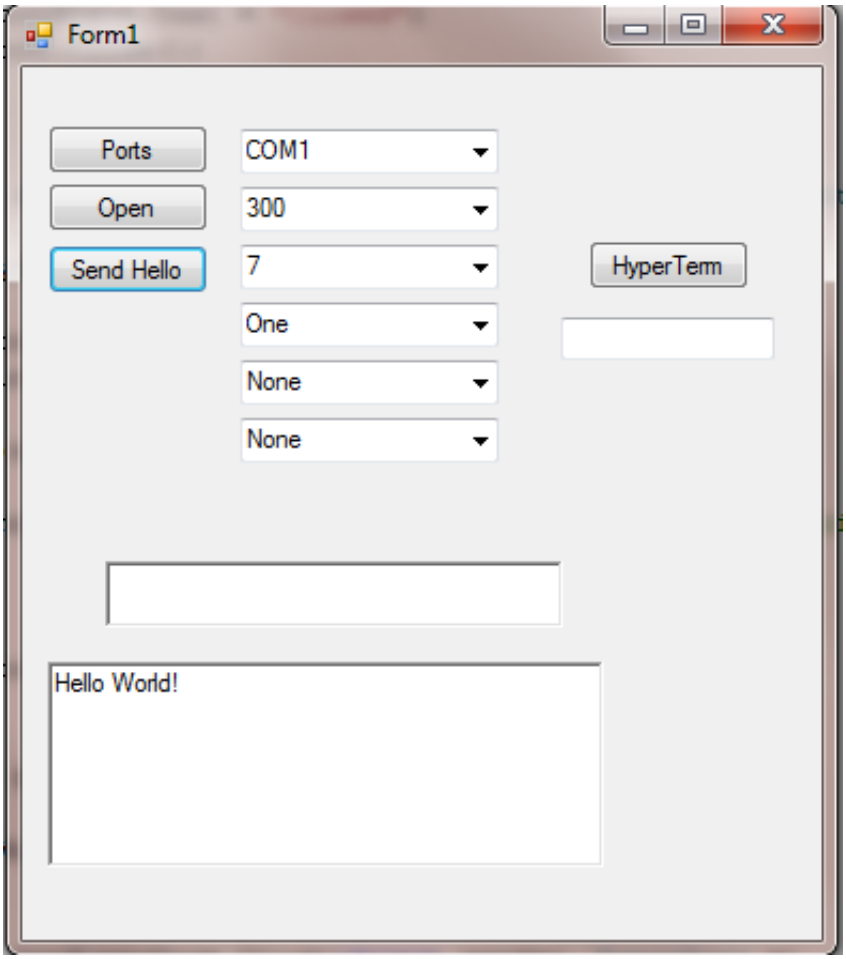
As above related commands are `Read()`, `ReadByte()`, `ReadChar()`, `ReadLine()`, and `ReadTo()`. Often I have found that devices tend to reply with text (some can have the \r\n or carriage return & line feed settings altered) of all of them I have found `ReadExisting()` and `ReadLine()` the most use. `ReadTo()` can be used if checking for a specific character such as "\n" in incoming data the problem with this is if the data is large some times the character can be included by accident such as the "\n", I have used the `ReadExisting()` method for talking to devices that I know will only send a fixed amount and the `ReadLine()` for a situation where the data is quite large as it will read up to a new line (discarding the new line) as follows:

Hide Copy Code

```
private void port_DataReceived(object sender, SerialDataReceivedEventArgs e)
{
     ComPort.ReadLine();
}
```

Placed in the interrupt handler for the port.

Below is the application as it stands:



This achieved by clicking the Ports button to fill the combo boxes and then Open button to open the port chosen with the

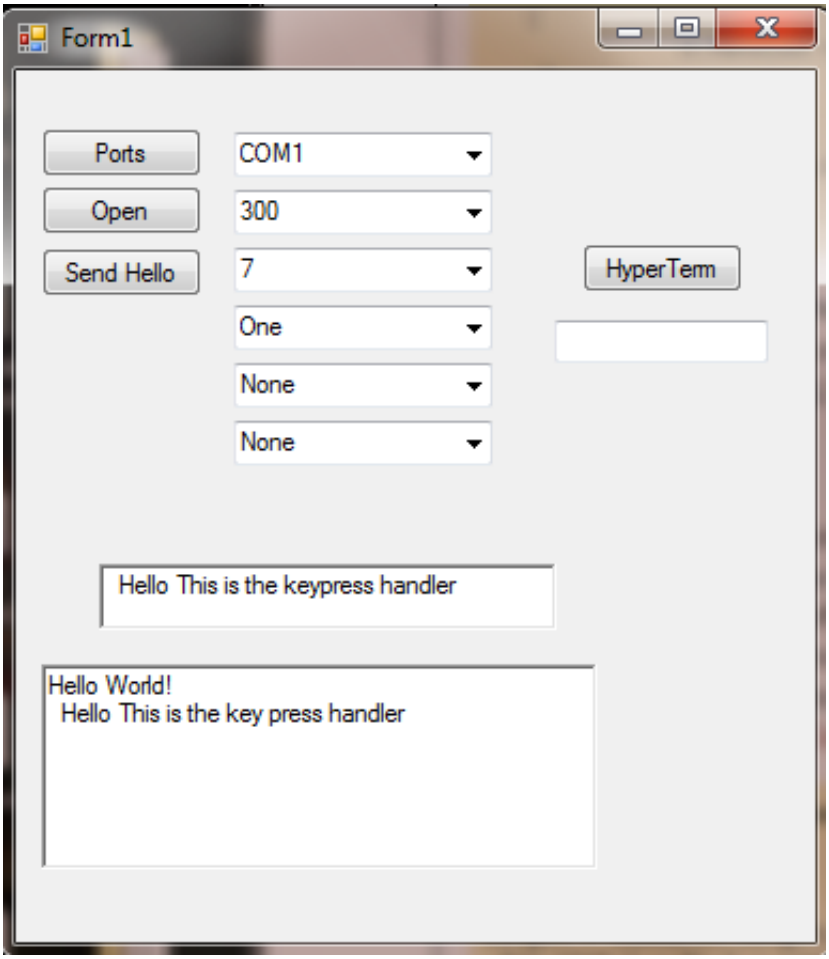
combo boxes the speed of 300 is slow enough to see the message get caught between read cycles of the port.

The next thing to try once you have the above working is to add another rich text box called `rtbOutgoing` add the below code to project:

Hide Copy Code

```
private void rtbOutgoing_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar == (char)13) // enter key
    {
        ComPort.Write("\r\n");
        rtbOutgoing.Text = "";
     }
    else if (e.KeyChar < 32 || e.KeyChar > 126)
   {
            e.Handled = true; // ignores anything else outside printable ASCII range
    }
    else
    {
        ComPort.Write(e.KeyChar.ToString());
    }
}
```

Once a rich text box and the code above is added the KeyPress event must be 'wired' to it by clicking the Event selection (looks like a thunder bolt) going to the KeyPress event and clicking the down arrow and selecting the `rtbOutgoing_KeyPress` this will attach the event.



This will then allow for typing in the outgoing text box to be picked up by the short between pins 2 & 3 from earlier. This will allow text to be sent as a key pressed just like Hyperterminal (no longer comes on Vista up, guessing Microsoft though it wasn't needed...). Hyperterminal for many years was seen as the basis for most serial comms and many devices are expecting

Hyperterminal style commands. I can't say this is the proper way of doing this, however it's ugly but it does work, for this example the text is loaded into a text box called `txtCommand`.

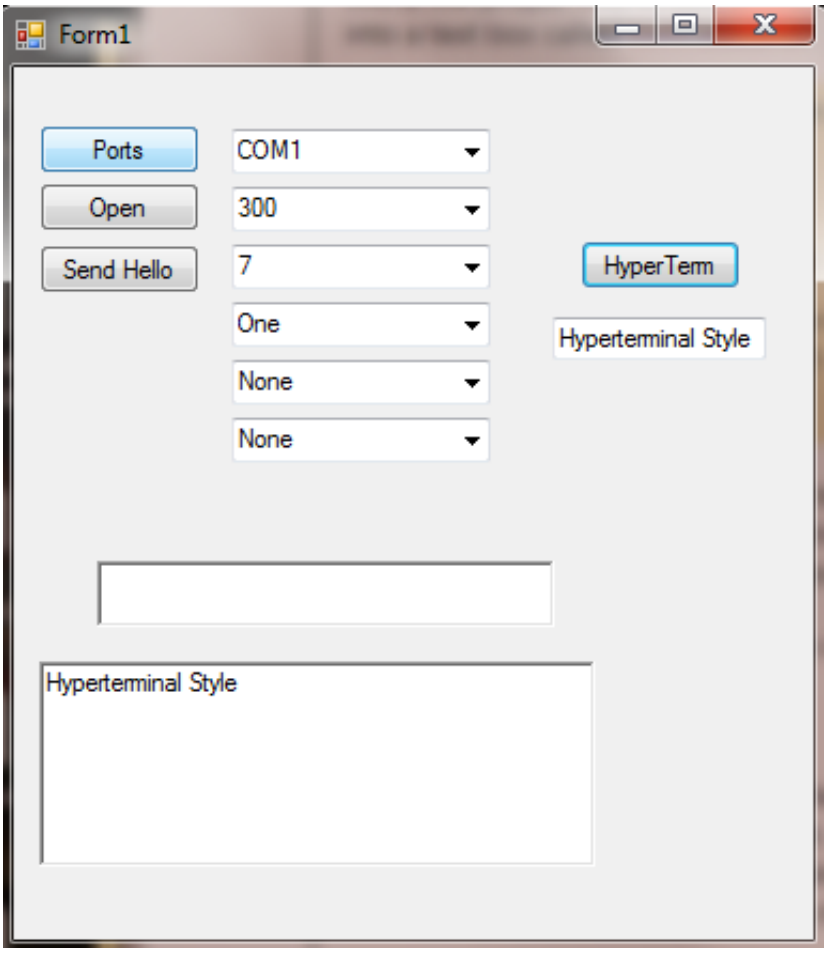
Hide Copy Code

```
string Command1 = txtCommand.Text;
string CommandSent;
int Length, j = 0;

Length = Command1.Length;
 for (int i = 0; i < Length; i++)
{
    CommandSent = Command1.Substring(j, 1);
    myComPort.Write(CommandSent);
    j++;
}
```

All this does in reality is to take the text in `txtCommand` (this could be "RZ04" for example), `Length` an integer is set to the length of the string, a `for` loop then counts from 0 to `Length` each time `CommandSent` is equal to the substring of `Command1` at `j` for one character.

If this is added to the application as a button a textbox names `txtCommand` the form resembles the below:



This example uses the interrupt to handle the data separate from the write, for some applications I have found that this is not really affective. I have found that a method of having one function that you can call and send the data and then it remains in the function while waiting for a response can be useful in debugging hardware (to cure the is that Windows or the hardware frown!).

Hide Copy Code

```
enum REPLY : int { NO_REPLY, YES_REPLY, TIMEOUT_REPLY }
System.Timers.Timer NoDataAtPort = new System.Timers.Timer(10000);
```

By use of the enum `Reply` integer, `Reply` can have three possible values a No, a Yes, and a Timeout. The `NO_REPLY` is for waiting for a reply the timer will stop it waiting for infinity. The `YES_REPLY` is the device at the end has seen the data and has

Acknowledge it, the `TIMEOUT_REPLY` is for the event when the timer expirers and there has been no reply.

Below is an example of a Write / Read routine I wrote and used in some Automatic Test Equipment (hence ATE being used!)

Hide Shrink ▲ Copy Code

```
private string Write_ATE(string Data_To_ATE)
{
    string Data_From_ATE = "";
    Reply_Status = (int)REPLY.NO_REPLY;
    ATEComPort.Write(Data_To_ATE);
    while (Reply_Status == (int)REPLY.NO_REPLY)
    {
        NoDataAtPort.Enabled = true;
    }
    NoDataAtPort.Enabled = false;
    if (Reply_Status == (int)REPLY.TIMEOUT_REPLY)
    {
         Reply_Status = (int)REPLY.NO_REPLY;
        ATEComPort.Write(Data_To_ATE);
        while (Reply_Status == (int)REPLY.NO_REPLY)
        {
            NoDataAtPort.Enabled = true;
        }
        NoDataAtPort.Enabled = false;
        if (Reply_Status == (int)REPLY.TIMEOUT_REPLY)
        {
                    Data_From_ATE = "TIMEOUT";
            return (Data_From_ATE);
        }
        else if (Reply_Status == (int)REPLY.YES_REPLY)
        {
            Data_From_ATE = ATEComPort.ReadTo("\r\n");
           if ((Data_From_ATE.Substring(0, 1)) == (Data_To_ATE.Substring(1, 1)))
           {
             return (Data_From_ATE);
           }
        }
        else
        {
           Data_From_ATE = "SERIOUS_ERROR";
           return (Data_From_ATE);
       }
     }
     else if (Reply_Status == (int)REPLY.YES_REPLY)
     {
       Data_From_ATE = ATEComPort.ReadTo("\r\n");
       if ((Data_From_ATE.Substring(0, 1)) == (Data_To_ATE.Substring(1, 1)))
       {
         return (Data_From_ATE);
       }
       //add hardware replies to this section as below
       /*else if ((Data_From_ATE.Substring(0, 1)) == "E")
       {
            Reply_Status = (int)REPLY.NO_REPLY;
            ATEComPort.Write(Data_To_ATE);
            while (Reply_Status == (int)REPLY.NO_REPLY)
            {
                NoDataAtPort.Enabled = true;
            }
            NoDataAtPort.Enabled = false;
            if (Reply_Status == (int)REPLY.TIMEOUT_REPLY)
```

```
            {
               Data_From_ATE = "TIMEOUT";
               return (Data_From_ATE);
            }
            else if (Reply_Status == (int)REPLY.YES_REPLY)
            {
               Data_From_ATE = ATEComPort.ReadTo("\r\n");
             if ((Data_From_ATE.Substring(0, 1)) == (Data_To_ATE.Substring(1, 1)))
            {
                  return (Data_From_ATE);
             }
             else if ((Data_From_ATE.Substring(0, 1)) == "E")
             {
                         return (Data_From_ATE);
             }
           }
           else
           {
                Data_From_ATE = "SERIOUS_ERROR";
                 return (Data_From_ATE);
           }
      }*/
    }
    else
    {
        Data_From_ATE = "SERIOUS_ERROR";
        return (Data_From_ATE);
    }
    return (Data_From_ATE);
}
```

This also an example of when and how to use `ReadTo()` in it I use `ReadTo("\r\n");` or read to the carriage return, line feed the device from memory used to spit out "\n" for a reason (which I can't think of at the moment!) and the only way of getting a reliable message was to use "\r\n". Time and shipping meant the error section was not used as the unit would return Exx, xx being 00 to 99, E99 for instance was "Command not recognized" there were other codes but these were not required.

The timer code is below:

Hide Copy Code

```
 private void OnTimeOut(object sender, ElapsedEventArgs e)
{
   Reply_Status = (int)REPLY.TIMEOUT_REPLY;
}
```

The Data received handler is below:

Hide Copy Code

```
private void port_DataReceived(object sender, SerialDataReceivedEventArgs e)
{
      Reply_Status = (int)REPLY.YES_REPLY;
}
```

This approach should only be used when and if the unit ends commands with a set string (like this unit did "\r\n") and the program is not required to 'do' any thing else, or some of the data expected is being lost i.e. "Hello World!!" becomes "llo World!".

This method should be called with the following:

Hide Copy Code

```
private void btnTest_Click(object sender, EventArgs e)
{
```

```
    //to send command use below routine
    string ATE_Reply = null;
    Reply_Status = (int)REPLY.NO_REPLY;
    ATE_Reply = Write_ATE("Hello World!\r\n");
    rtbIncoming.Text += ATE_Reply + "\r\n";
}
```

This will send "Hello World!" out of Com 1, this method was originally developed for an Automatic Test Equipment that was being difficult (to use a polite word!) in that some of the devices sent data at a high rate others sent data a low rate and the interrupt method could be called and possibly block the port before it was done sending the command (hence the Enum array and there being no reading done in the interrupt handler).

I have focused on R-232 in the examples; however this should be equally applicable to other serial networks such as the RS-422 and RS-485.

RS-485 does not come as a standard on the PC a third party device such as those by the company Amplicon (http://www.amplicon.com/), the settings are dictated by the devices connected together on the bus the PC listens in via the device attached to the comm. port, to open the port follow the products data sheet and open the port. The commands and formatting of them should all be detailed in the documentation.

Below is a full version of the code:

Hide Shrink ▲ Copy Code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.IO.Ports;
using System.Windows.Forms;
//CodeProjectSerialComms program

//23/04/2013   16:29

namespace CodeProjectSerialComms
{
   public partial class Form1 : Form
    {
        SerialPort ComPort = new SerialPort();

        internal delegate void SerialDataReceivedEventHandlerDelegate(
                 object sender, SerialDataReceivedEventArgs e);

        internal delegate void SerialPinChangedEventHandlerDelegate(
                 object sender, SerialPinChangedEventArgs e);
        private SerialPinChangedEventHandler SerialPinChangedEventHandler1;
        delegate void SetTextCallback(string text);
         string InputData = String.Empty;
        public Form1()
        {
            InitializeComponent();
            SerialPinChangedEventHandler1 = new SerialPinChangedEventHandler(PinChanged);
            ComPort.DataReceived +=
              new System.IO.Ports.SerialDataReceivedEventHandler(port_DataReceived_1);
       }
      private void btnGetSerialPorts_Click(object sender, EventArgs e)
      {
            string[] ArrayComPortsNames = null;
            int index = -1;
            string ComPortName = null;
```

```
            //Com Ports
            ArrayComPortsNames = SerialPort.GetPortNames();
             do
             {
                index += 1;
                 cboPorts.Items.Add(ArrayComPortsNames[index]);
             } while (!((ArrayComPortsNames[index] == ComPortName) ||
               (index == ArrayComPortsNames.GetUpperBound(0))));
             Array.Sort(ArrayComPortsNames);

            if (index == ArrayComPortsNames.GetUpperBound(0))
             {
                 ComPortName = ArrayComPortsNames[0];
             }

             //get first item print in text
             cboPorts.Text = ArrayComPortsNames[0];

             //Baud Rate
             cboBaudRate.Items.Add(300);
             cboBaudRate.Items.Add(600);
             cboBaudRate.Items.Add(1200);
             cboBaudRate.Items.Add(2400);
             cboBaudRate.Items.Add(9600);
             cboBaudRate.Items.Add(14400);
             cboBaudRate.Items.Add(19200);
             cboBaudRate.Items.Add(38400);
             cboBaudRate.Items.Add(57600);
             cboBaudRate.Items.Add(115200);
             cboBaudRate.Items.ToString();

             //get first item print in text
             cboBaudRate.Text = cboBaudRate.Items[0].ToString();
                       //Data Bits
             cboDataBits.Items.Add(7);
             cboDataBits.Items.Add(8);
             //get the first item print it in the text
             cboDataBits.Text = cboDataBits.Items[0].ToString();

                        //Stop Bits
             cboStopBits.Items.Add("One");
              cboStopBits.Items.Add("OnePointFive");
             cboStopBits.Items.Add("Two");
             //get the first item print in the text
             cboStopBits.Text = cboStopBits.Items[0].ToString();

             //Parity
             cboParity.Items.Add("None");
             cboParity.Items.Add("Even");
             cboParity.Items.Add("Mark");
             cboParity.Items.Add("Odd");
             cboParity.Items.Add("Space");

             //get the first item print in the text

             cboParity.Text = cboParity.Items[0].ToString();

             //Handshake
             cboHandShaking.Items.Add("None");
             cboHandShaking.Items.Add("XOnXOff");
             cboHandShaking.Items.Add("RequestToSend");
             cboHandShaking.Items.Add("RequestToSendXOnXOff");

             //get the first item print it in the text
             cboHandShaking.Text = cboHandShaking.Items[0].ToString();
```

```
        }

        private void port_DataReceived_1(object sender, SerialDataReceivedEventArgs e)
        {
            InputData = ComPort.ReadExisting();
            if (InputData != String.Empty)
            {
               this.BeginInvoke(new SetTextCallback(SetText), new object[] { InputData });
            }
        }

        private void SetText(string text)
        {
            this.rtbIncoming.Text += text;
        }

        internal void PinChanged(object sender, SerialPinChangedEventArgs e)
        {
             SerialPinChange SerialPinChange1 = 0;
             bool signalState = false;
             SerialPinChange1 = e.EventType;
             lblCTSStatus.BackColor = Color.Green;
             lblDSRStatus.BackColor = Color.Green;
             lblRIStatus.BackColor = Color.Green;
             lblBreakStatus.BackColor = Color.Green;

             switch (SerialPinChange1)
            {
                case SerialPinChange.Break:
                     lblBreakStatus.BackColor = Color.Red;
                     //MessageBox.Show("Break is Set");
                  break;
                case SerialPinChange.CDChanged:
                     signalState = ComPort.CtsHolding;
                   //  MessageBox.Show("CD = " + signalState.ToString());
                  break;
                case SerialPinChange.CtsChanged:
                     signalState = ComPort.CDHolding;
                     lblCTSStatus.BackColor = Color.Red;
                    //MessageBox.Show("CTS = " + signalState.ToString());
                   break;
               case SerialPinChange.DsrChanged:
                     signalState = ComPort.DsrHolding;
                     lblDSRStatus.BackColor = Color.Red;
                     // MessageBox.Show("DSR = " + signalState.ToString());
                  break;
               case SerialPinChange.Ring:
                      lblRIStatus.BackColor = Color.Red;
                      //MessageBox.Show("Ring Detected");
                     break;
            }
         }

         private void btnTest_Click(object sender, EventArgs e)
         {

             //SerialPinChangedEventHandler1 = new SerialPinChangedEventHandler(PinChanged);
             //ComPort.PinChanged += SerialPinChangedEventHandler1;
             //ComPort.Open();
             //ComPort.RtsEnable = true;
             //ComPort.DtrEnable = true;
             //btnTest.Enabled = false;

         }
```

```
        private void btnPortState_Click(object sender, EventArgs e)
        {
             if (btnPortState.Text == "Closed")
             {
                 btnPortState.Text = "Open";
                 ComPort.PortName = Convert.ToString(cboPorts.Text);
                 ComPort.BaudRate = Convert.ToInt32(cboBaudRate.Text);
                 ComPort.DataBits = Convert.ToInt16(cboDataBits.Text);
                 ComPort.StopBits = (StopBits)Enum.Parse(typeof(StopBits), cboStopBits.Text);
                 ComPort.Handshake = (Handshake)Enum.Parse(typeof(Handshake), cboHandShaking.Text);
                 ComPort.Parity = (Parity)Enum.Parse(typeof(Parity), cboParity.Text);
                 ComPort.Open();
             }
             else if (btnPortState.Text == "Open")
             {
                 btnPortState.Text = "Closed";
                 ComPort.Close();
             }
         }

         private void rtbOutgoing_KeyPress(object sender, KeyPressEventArgs e)
         {
           if (e.KeyChar == (char)13) // enter key
           {
                ComPort.Write("\r\n");
                rtbOutgoing.Text = "";
            }
            else if (e.KeyChar < 32 || e.KeyChar > 126)
             {
               e.Handled = true; // ignores anything else outside printable ASCII range
             }
             else
             {
                ComPort.Write(e.KeyChar.ToString());
             }
         }
         private void btnHello_Click(object sender, EventArgs e)
         {
             ComPort.Write("Hello World!");
         }
         private void btnHyperTerm_Click(object sender, EventArgs e)
         {
             string Command1 = txtCommand.Text;
             string CommandSent;
             int Length, j = 0;

             Length = Command1.Length;
             for (int i = 0; i < Length; i++)
             {
                 CommandSent = Command1.Substring(j, 1);
                 ComPort.Write(CommandSent);
                 j++;
             }
        }
    }
}
```

Points of Interest

I did find a lack of information about emulating HyperTerminal so I came up with my own version which appears to work as I said it's not pretty but... For further clarification of anything detailed here the book Serial Port Complete (Second Edition) by Jan Axelson (LVR). Also my first submission to CodeProject, so be nice!

History

- 0.1 First primordial upload for submission.

License

Share

About the Author

glennPattonInThePUB

Engineer Contracting
United Kingdom

I never wanted to program on Windows, I got stuck with it! I'm a hardware engineer who does software (at least that what I tell everyone!)

You may also be interested in...

Win32 SDK Serial Comm Made Easy

Top 5 .NET Metrics, Tips & Tricks

Game Programming in C - For Beginners

How-To Intel® IoT Technology Code Samples: