UNIVERSITÉ ANTONINE
Faculté d'ingénieurs en Informatique,
Multimédia, Réseaux & Télécommunications

Using the COM port in VB.NET

Matière : VB.NET et ASP.NET

	NOM Prénom	INF#
Effectué par :	MATTA Elie	Privacy
	HALLAGE Rabih	applied
	Et al.	

Overview

In this part of our project, we will explain in which steps the programming of the COM port will pass.

Please refer to SerialPortCommunication_VBNet.sln to check how all the classes work and the opening/closing the com ports.

In computing, a serial port is a serial communication physical interface through which information transfers in or out one bit at a time (contrast parallel port). Throughout most of the history of personal computers, data transfer through serial ports connected the computer to devices such as terminals and various peripherals. Serial port emulation is useful especially when there is a lack of available physical serial ports. Communication between software and/or devices which would otherwise require extra physical connections, can be benefited by using a virtual COM Port emulator.
A virtual COM port itself is a relatively simple software mechanism that can be implemented by driver software similar to that of a conventional COM port driver. A redirector for the Windows operating system is typically configured using a control-panel style graphical user interface for creating virtual COM ports, configuring settings for individual COM ports, and configuring global settings affecting all COM ports. The redirector GUI typically also includes displays of virtual COM port activity and various diagnostic aids.
And as soon as we finish understanding the whole idea, we can start programming the signal which is going to be emitted from the COM port by using different programming languages such as Pic Basic.
After that, we need to write that code on our board that's why we need a writer on the IC of the board and a reader and so on…

Hardware abstraction

Operating systems usually use a symbolic name to refer to the serial ports of a computer. Unix-like operating systems usually label the serial port devices /dev/tty* (*tty* an abbreviation for *teletype*) where * represents a string identifying the terminal device; the syntax of that string depends on the operating system and the device. The Microsoft MS-DOS and Windows environments refer to serial ports as COM ports: COM1, COM2, etc. On Linux, 8250/16550 UART hardware serial ports are named /dev/ttyS*, USB adapters appear as /dev/ttyUSB* and various types of virtual serial ports do not necessarily have names starting with tty.

History:

Back in the days of Visual Basic 6.0, you had to use the MSComm Control 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 its 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.

Then along comes .Net 2.0, and this time Microsoft added the System.IO.Ports Namespace, and within that was the SerialPort 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. This allows you to retrieve a list of available ports (COM1, COM2, etc.) available for the computer the application is running on.

Settings

Many settings are required for serial connections used for asynchronous start-stop communication, to select speed, number of data bits per character, parity, and number of stop bits per character. Hardware from the 1980s and earlier may require setting switches or jumpers on a circuit board. The speed is either fixed or automatically negotiated. Often if the settings are entered incorrectly the connection will not be dropped; however, any data sent will be received on the other end as nonsense.

Speed

Serial ports use two-level (binary) signaling, so the data rate in bits per second is equal to the symbol rate in bauds. These rates are based on multiples of the rates for electromechanical teleprinters. The port speed and device speed must match. Although some devices may automatically detect popular personal computers, allowing for much higher baud rates, the capability to set a bit rate does not imply that a working connection will result. Not all bit rates are possible with all serial ports. Some special-purpose protocols such as MIDI for musical instrument control, use serial data rates other than the above series.

The speed includes bits for framing (stop bits, parity, etc.) and so the effective data rate is lower than the bit transmission rate. For example with 8-N-1 character framing only 80% of the bits are available for data (for every eight bits of data, two more framing bits are sent).

Data Bits

The number of data bits in each character can be 5 (for Baudot code), 6 (rarely used), 7 (for true ASCII), 8 (for any kind of data, as this matches the size of a byte), or 9 (rarely used). 8 data bits are almost universally used in newer applications. 5 or 7 bits generally only make sense with older equipment such as teleprinters.

Most serial communications designs send the data bits within each byte LSB (Least Significant Bit) first. This standard is also referred to as "little endian". Also

possible, but rarely used, is "big endian" or MSB (Most Significant Bit) first serial communications. The order of bits is not usually configurable, but data can be byte-swapped only before sending.

Parity

Parity is a method of detecting some errors in transmission. Where parity is used with a serial port, an extra data bit is sent with each data character, arranged so that the number of 1 bits in each character, including the parity bit, is always odd or always even. If a byte is received with the wrong number of 1 bits, then it must have been corrupted. If parity is correct there has been an even number of errors. Electromechanical teleprinters were arranged to print a special character when received data contained a parity error, to allow detection of messages damaged by line noise. A single parity bit does not allow implementation of error correction on each character, and communication protocols working over serial data links will have higher-level mechanisms to ensure data validity and request retransmission of data that has been incorrectly received.

The parity bit in each character can be set to none (N), odd (O), even (E), mark (M), or space (S). None means that no parity bit is sent at all. Mark parity means that the parity bit is always set to the mark signal condition (logical 1) and likewise space parity always sends the parity bit in the space signal condition. Aside from uncommon applications that use the 9th (parity) bit for some form of addressing or special signalling, mark or space parity is uncommon, as it adds no error detection information. Odd parity is more common than even, since it ensures that at least one state transition occurs in each character, which makes it more reliable. The most common parity setting, however, is "none", with error detection handled by a communication protocol.

Stop bits

Stop bits sent at the end of every character allow the receiving signal hardware to detect the end of a character and to resynchronise with the character stream. Electronic devices usually use one stop bit. If slow electromechanical teleprinters are used, one-and-one half or two stop bits are required.

Conventional notation

The D/P/S conventional notation specifies the framing of a serial connection. The most common usage on microcomputers is 8/N/1 (8N1). This specifies 8 data bits, no parity, 1 stop bit. In this notation, the parity bit is not included in the data bits. 7/E/1 (7E1) means that an even parity bit is added to the seven data bits for a total of eight bits between the start and stop bits. If a receiver of a 7/E/1 stream is expecting an 8/N/1 stream, half the possible bytes will be interpreted as having the high bit set.

Flow control

A serial port may use signals in the interface to pause and resume the transmission of data. For example, a slow printer might need to handshake with the serial port to indicate that data should be paused while the mechanism advances a line. Common hardware handshake signals use the RS-232 RTS/CTS, DTR/DSR signal circuits. Generally, the RTS and CTS are turned off and on from alternate ends to control data flow, for instance when a buffer is almost full. DTR and DSR are usually on all the time and are used to signal from each end that the other equipment is actually present and powered-up.

Another method of flow control may use special characters such as XON/XOFF to control the flow of data. The XON/XOFF characters are sent by the receiver to the sender to control when the sender will send data, that is, these characters go in the opposite direction to the data being sent. The XON character tells the sender that the receiver is ready for more data. The XOFF character tells the sender to stop sending characters until the receiver is ready again. These are non-printing characters and are interpreted as handshake signals by printers and terminals.

If all possible values of a character must be sent as user data, XON/XOFF handshaking presents difficulties since these codes may appear in user data. Control characters sent as part of the data stream must be sent as part of an escape sequence to prevent data from being interpreted as flow control. Since no extra signal circuits are required, XON/XOFF flow control can be done on a 3 wire interface.

Implementation

Now that we have that out of the way, let's move on to programming our application. As with all application we created, we keep functionality separated from presentation, we do this by creating Manager Classes that manage the functionality or a given process. What we will be looking at is the code in my CommManager class. As with anything you write in .Net you need to add the references to the Namespace's we'll be using:

```
Imports System
Imports System.Text
Imports System.Drawing
Imports System.IO.Ports
Imports System.Windows.Forms
```

In this application we 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:

```
#Region "Manager Enums"
''' <summary>
''' enumeration to hold our transmission types
''' </summary>
Public Enum TransmissionType
    Text
    Hex
End Enum

''' <summary>
''' enumeration to hold our message types
''' </summary>
Public Enum MessageType
    Incoming
    Outgoing
    Normal
    Warning
    [Error]
End Enum
#End Region
```

Next we have our variable list, 6 of them are for populating our class Properties, the others are being access throughout the manager class so they are made Global.

Things are a bit different when sealing with delegates and how objects are access through VB.Net than they are in C#, so in the VB.Net version there are 2 more properties, and an extra Boolean variable used to determine if the buffer is to write the current data to the serial port. Here are the variables needed for the manager class:

```
#Region "Manager Variables"
'property variables
Private _baudRate As String = String.Empty
Private _parity As String = String.Empty
Private _stopBits As String = String.Empty
Private _dataBits As String = String.Empty
Private _portName As String = String.Empty
Private _transType As TransmissionType
Private _displayWindow As RichTextBox
Private _msg As String
Private _type As MessageType
'global manager variables
Private MessageColor As Color() = {Color.Blue, Color.Green, Color.Black,
Color.Orange, Color.Red}
Private comPort As New SerialPort()
Private write As Boolean = True
#End Region
```

NOTE: We will always separate our code into sections using the *#region ... #end* region 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. As stated above we needed to add 2 additional properties for the conversion from C# to VB.Net. We have properties for the following items which we already explained above:

- **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.

- **MessageType:** Outgoing, Incoming, Error, Warning, etc.
- **Message:** This is the actual message being sent through the serial port

We also have 4 properties that aren't related to the port itself, but with where the data will be displayed, what transmission type to use, the message type, and the message itself:

```
''' <summary>
''' Property to hold the message being sent
''' through the serial port
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public Property Message() As String
    Get
        Return _msg
    End Get
    Set(ByVal value As String)
        _msg = value
    End Set
End Property

''' <summary>
''' Message to hold the transmission type
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public Property Type() As MessageType
    Get
        Return _type
    End Get
    Set(ByVal value As MessageType)
        _type = value
    End Set
End Property
#End Region
```

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 from occurring. We also add an EventHandler in the constructor, the event will be executed whenever there's data waiting in the buffer:

```
#Region "Manager Properties"
    ''' <summary>
    ''' Property to hold the BaudRate
    ''' of our manager class
    ''' </summary>
    Public Property BaudRate() As String
        Get
            Return _baudRate
        End Get
        Set(ByVal value As String)
            _baudRate = value
        End Set
    End Property

    ''' <summary>
    ''' property to hold the Parity
    ''' of our manager class
    ''' </summary>
    Public Property Parity() As String
        Get
            Return _parity
        End Get
        Set(ByVal value As String)
            _parity = value
        End Set
    End Property

#Region "Manager Constructors"
    ''' <summary>
    ''' Constructor to set the properties of our Manager Class
    ''' </summary>
    ''' <param name="baud">Desired BaudRate</param>
    ''' <param name="par">Desired Parity</param>
    ''' <param name="sBits">Desired StopBits</param>
    ''' <param name="dBits">Desired DataBits</param>
    ''' <param name="name">Desired PortName</param>
Public Sub New(ByVal baud As String, ByVal par As String, ByVal sBits As
String, ByVal dBits As String, ByVal name As

String, ByVal rtb As RichTextBox)
        _baudRate = baud
        _parity = par
        _stopBits = sBits
        _dataBits = dBits
        _portName = name
        _displayWindow = rtb
        'now add an event handler
        AddHandler comPort.DataReceived, AddressOf comPort_DataReceived
    End Sub
```



```
    ''' <summary>
    ''' Comstructor to set the properties of our
    ''' serial port communicator to nothing
    ''' </summary>
    Public Sub New()
        _baudRate = String.Empty
        _parity = String.Empty
        _stopBits = String.Empty
        _dataBits = String.Empty
        _portName = "COM1"
        _displayWindow = Nothing
        'add event handler
        AddHandler comPort.DataReceived, AddressOf comPort_DataReceived
    End Sub
#End Region
```

The first think we 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 of the SerialPort Class. If the port isn't open we open it by calling the Open Method of the SerialPort Class.

For writing to the port we use the Write Method. It is in this method we utilize the new boolean variable, **write** to determine if we want to write the data. This is used to handling byte transmission type when the data is in the incorrect format:

```
#Region "WriteData"
    Public Sub WriteData(ByVal msg As String)
        Select Case CurrentTransmissionType
            Case TransmissionType.Text
                'first make sure the port is open
                'if its not open then open it
                If Not (comPort.IsOpen = True) Then
                    comPort.Open()
                End If
                'send the message to the port
                comPort.Write(msg)
                'display the message
                _type = MessageType.Outgoing
                _msg = msg + "" + Environment.NewLine + ""
                DisplayData(_type, _msg)
                Exit Select
            Case TransmissionType.Hex
                Try
```

```
                    'convert the message to byte array
                    Dim newMsg As Byte() = HexToByte(msg)
                    'Determine if we are goint
                    'to write the byte data to the screen
                    If Not write Then
                        DisplayData(_type, _msg)
                        Exit Sub
                    End If
                    'send the message to the port
                    comPort.Write(newMsg, 0, newMsg.Length)
                    'convert back to hex and display
                    _type = MessageType.Outgoing
                    _msg = ByteToHex(newMsg) + "" + Environment.NewLine + ""
                    DisplayData(_type, _msg)
                Catch ex As FormatException
                    'display error message
                    _type = MessageType.Error
                    _msg = ex.Message + "" + Environment.NewLine + ""
                    DisplayData(_type, _msg)
                Finally
                    _displayWindow.SelectAll()
                End Try
                Exit Select
            Case Else
                'first make sure the port is open
                'if its not open then open it
                If Not (comPort.IsOpen = True) Then
                    comPort.Open()
                End If
                'send the message to the port
                comPort.Write(msg)
                'display the message
                _type = MessageType.Outgoing
                _msg = msg + "" + Environment.NewLine + ""
                DisplayData(MessageType.Outgoing, msg + "" +
Environment.NewLine + "")
                Exit Select
        End Select
    End Sub
#End Region
```

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:

```
#Region "HexToByte"
    ''' <summary>
    ''' method to convert hex string into a byte array
    ''' </summary>
    ''' <param name="msg">string to convert</param>
    ''' <returns>a byte array</returns>
    Private Function HexToByte(ByVal msg As String) As Byte()
        'Here we added an extra check to ensure the data
        'was the proper length for converting to byte
        If msg.Length Mod 2 = 0 Then
            'remove any spaces from the string
            _msg = msg
            _msg = msg.Replace(" ", "")
            'create a byte array the length of the
            'divided by 2 (Hex is 2 characters in length)
            Dim comBuffer As Byte() = New Byte(_msg.Length / 2 - 1) {}
            For i As Integer = 0 To _msg.Length - 1 Step 2
                comBuffer(i / 2) = CByte(Convert.ToByte(_msg.Substring(i, 2),
16))
            Next
            write = True
            'loop through the length of the provided string
            'convert each set of 2 characters to a byte
            'and add to the array
            'return the array
            Return comBuffer
        Else
            'Message wasnt the proper length
            'So we set the display message
            _msg = "Invalid format"
            _type = MessageType.Error
            ' DisplayData(_Type, _msg)
            'Set our boolean value to false
            write = False
            Return Nothing
        End If
    End Function
#End Region
```


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:

```
#Region "ByteToHex"
    ''' <summary>
    ''' method to convert a byte array into a hex string
    ''' </summary>
    ''' <param name="comByte">byte array to convert</param>
    ''' <returns>a hex string</returns>
    Private Function ByteToHex(ByVal comByte As Byte()) As String
        'create a new StringBuilder object
        Dim builder As New StringBuilder(comByte.Length * 3)
        'loop through each byte in the array
        For Each data As Byte In comByte
            builder.Append(Convert.ToString(data, 16).PadLeft(2,
"0"c).PadRight(3, " "c))
            'convert the byte to a string and add to the stringbuilder
        Next
        'return the converted value
        Return builder.ToString().ToUpper()
    End Function
#End Region
```

The last method that **WriteData** depends on is the **DisplayData** method. Here we use the Invoke Method of our RichTextBox, the control used to display the data, to create a new EventHandler which creates a new Delegate for setting the properties we wish for our message, then appending it to the value already displayed.

We had to change the format of the **DisplayData** method as VB.Net handles **delegates** completely different than C#. Instead of putting the functionality of the delegate in the method, we had to create a seperate method, then use the AddressOf Method to reference the procedure that will act as our delegate:

```
#Region "DisplayData"
    ''' <summary>
    ''' Method to display the data to and
    ''' from the port on the screen
    ''' </summary>
    ''' <remarks></remarks>
    <STAThread()> _
    Private Sub DisplayData(ByVal type As MessageType, ByVal msg As String)
        _displayWindow.Invoke(New EventHandler(AddressOf DoDisplay))
    End Sub
#End Region
```


NOTE: You will notice that we have added the STAThreadAttribute to our method. This is used when a single thread apartment is required by a control, like the RichTextBox.

Now we will look at our delegate method, which is responsible for setting all of the properties of our display window,
which is a RichTextBox, as it has text formatting options not available to the regular TextBox:

```
#Region "DoDisplay"
    Private Sub DoDisplay(ByVal sender As Object, ByVal e As EventArgs)
        _displayWindow.SelectedText = String.Empty
        _displayWindow.SelectionFont = New Font(_displayWindow.SelectionFont,
FontStyle.Bold)
        _displayWindow.SelectionColor = MessageColor(CType(_type, Integer))
        _displayWindow.AppendText(_msg)
        _displayWindow.ScrollToCaret()
    End Sub
#End Region
```

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:

```
#Region "OpenPort"
    Public Function OpenPort() As Boolean
        Try
            'first check if the port is already open
            'if its open then close it
            If comPort.IsOpen = True Then
                comPort.Close()
            End If

            'set the properties of our SerialPort Object
            comPort.BaudRate = Integer.Parse(_baudRate)
            'BaudRate
            comPort.DataBits = Integer.Parse(_dataBits)
            'DataBits
            comPort.StopBits = DirectCast([Enum].Parse(GetType(StopBits),
_stopBits), StopBits)
            'StopBits
            comPort.Parity = DirectCast([Enum].Parse(GetType(Parity),
_parity), Parity)
            'Parity
            comPort.PortName = _portName
            'PortName
            'now open the port
```



```
            comPort.Open()
            'display message
            _type = MessageType.Normal
            _msg = "Port opened at " + DateTime.Now + "" +
Environment.NewLine + ""
            DisplayData(_type, _msg)
            'return true
            Return True
        Catch ex As Exception
            DisplayData(MessageType.[Error], ex.Message)
            Return False
        End Try
    End Function
#End Region
```

Now that we have opened our port for communication for sending data through, we need to be able to close the port when we are finished without our application. Here we simply call the Close Method of the SerialPort Object, which also disposes of the internal stream being used for the data transmission:

```
#Region " ClosePort "
    Public Sub ClosePort()
        If comPort.IsOpen Then
            _msg = "Port closed at " + DateTime.Now + "" +
Environment.NewLine + ""
            _type = MessageType.Normal
            DisplayData(_type, _msg)
            comPort.Close()
        End If
    End Sub
#End Region
```

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:

```
#Region "comPort_DataReceived"
    ''' <summary>
    ''' method that will be called when theres data waiting in the buffer
    ''' </summary>
    ''' <param name="sender"></param>
    ''' <param name="e"></param>
    Private Sub comPort_DataReceived(ByVal sender As Object, ByVal e As
SerialDataReceivedEventArgs)
        'determine the mode the user selected (binary/string)
        Select Case CurrentTransmissionType
            Case TransmissionType.Text
                'user chose string
                'read data waiting in the buffer
```

```
                Dim msg As String = comPort.ReadExisting()
                'display the data to the user
                _type = MessageType.Incoming
                _msg = msg
                DisplayData(MessageType.Incoming, msg + "" +
Environment.NewLine + "")
                Exit Select
            Case TransmissionType.Hex
                'user chose binary
                'retrieve number of bytes in the buffer
                Dim bytes As Integer = comPort.BytesToRead
                'create a byte array to hold the awaiting data
                Dim comBuffer As Byte() = New Byte(bytes - 1) {}
                'read the data and store it
                comPort.Read(comBuffer, 0, bytes)
                'display the data to the user
                _type = MessageType.Incoming
                _msg = ByteToHex(comBuffer) + "" + Environment.NewLine + ""
                DisplayData(MessageType.Incoming, ByteToHex(comBuffer) + "" +
Environment.NewLine + "")
                Exit Select
            Case Else
                'read data waiting in the buffer
                Dim str As String = comPort.ReadExisting()
                'display the data to the user
                _type = MessageType.Incoming
                _msg = str + "" + Environment.NewLine + ""
                DisplayData(MessageType.Incoming, str + "" +
Environment.NewLine + "")
                Exit Select
        End Select
    End Sub
#End Region
```

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 ComboBoxe'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
- StopBits Enumeration

```
#Region "SetParityValues"
    Public Sub SetParityValues(ByVal obj As Object)
        For Each str As String In [Enum].GetNames(GetType(Parity))
```

```
            DirectCast(obj, ComboBox).Items.Add(str)
        Next
    End Sub
#End Region

#Region "SetStopBitValues"
    Public Sub SetStopBitValues(ByVal obj As Object)
        For Each str As String In [Enum].GetNames(GetType(StopBits))
            DirectCast(obj, ComboBox).Items.Add(str)
        Next
    End Sub
#End Region

#Region "SetPortNameValues"
    Public Sub SetPortNameValues(ByVal obj As Object)

        For Each str As String In SerialPort.GetPortNames()
            DirectCast(obj, ComboBox).Items.Add(str)
        Next
    End Sub

#End Region
```

That is how you do Serial Port Communication in VB.Net. Microsoft finally gave us intrinsic tools to perform this task, no more relying on legacy objects. We are providing this class and a sample application to show how to implement what we just learned.

Please refer to SerialPortCommunication_VBNet.sln to check how all the classes work and the opening/closing the com ports.

Thank you.

