# How to Communicate with a Microcontroller or Other Serial Device in VB.Net

## Table of Contents

## Introduction

With today's low-cost electronics and micro-manufacturing techniques the average electronics hobbyist has access to a plethora of powerful little microcontrollers (MCUs), along with a huge assortment of possible peripherals, and nearly all of these require a RS-232 serial interface to communicate with a PC. The accessibility and ease-of-use of VB.Net has made it an attractive platform for developing small purpose-driven applications meant to configure or otherwise communicate with these microcontrollers and microcontroller-based devices. However, as obvious a choice as the SerialPort component is, there are some common pitfalls to its use; and even though these pitfalls are explained in the documentation, a proper implementation of the component can still be elusive for some developers. This article will attempt to explain the common usage of the SerialPort component and offer a sample "SerialDevice" class which can be used to encapsulate the SerialPort component instance along with the data and logic associated with its usage.

↑ Return to Top

## Microcontrollers (MCUs)

There are a few very popular hobby/prototyping MCU platforms, among them are Arduino, BasicStamp, and more recently mBed. But there are also many, many others out there. These typically range from 8-bit to 32-bit processors containing some amount of flash-memory used for storage and operating memory, as well as an array of peripherals such as Digital/Analog I/O, UARTs, I2C/SPI buses, etc. Most are inexpensive (<$100 USD) and several offer rich online communities and/or public code libraries. These devices are popular with electronics designers and hobbyists alike, and once you get past the initial introduction and setup they tend to be quite easy to work with.

Since these devices are popular with electronics designers and hobbyists and since Visual Basic .Net is free, easy to use, and well supported it is natural that people would attempt to write a VB app to configure or communicate with their MCU, having little or no programming experience at the outset. There are typically just a few requirements for this type of application; it needs to connect to the MCU on a serial port (often a USB virtual serial port) and then send and receive what are usually a small number of bytes representing a command with possible parameters and a command execution response. Sometimes the response is expected to contain a long sequence of byte data which will require further processing beyond a simple status result comparison, but most times the result will be a small number of bytes which decode to a status result or requested data value. Due to the nature of the SerialPort component and its multithreaded events the process of sending a command, reading a result, and then processing that result is not as intuitive as the methods of the SerialPort component might make it seem.

↑ Return to Top

## Serial Port Pitfalls

In the VB.Net forums, I have seen many developers fall into the trap of attempting to use the SerialPort.Write method to send a command and then follow it with loop of SerialPort.Read method calls in an attempt to gather the response. This seems like a natural thing to do and is based on a simple assumption. The data being transmitted is small, so you know it happens very quickly, and you might even flush the buffer to be sure it is sent before reading. Once sent, there is nothing to do but wait for the resulting byte stream and the SerialPort.Read method will block until the bytes start to come in. When that happens, the loop will proceed sequentially and some small buffer array will be populated with each read byte. Simple, right? But whenever someone tries this, it almost always ends badly. The buffer they are trying to build doesn't contain the correct bytes, or they are not in the correct sequence, and the results seem sporadic and uncontrollable.

The root of this problem, and the flaw with the simple assumption, is that there are layers of abstraction between the SerialPort component in managed code and the physical hardware which actually performs the serial transmissions. And these layers can vary depending on the type of serial port and underlying hardware. The SerialPort component is designed to handle all of this complexity and notify you when data is accurately available to be consumed in managed code, but this means using the component as its creators intended.

↑ Return to Top

## Using the SerialPort Component

To do this, an application using a SerialPort component must subscribe to the SerialPort.DataReceived event. When data is available to managed code, the SerialPort will raise the DataReceived event and the attached event handler in the application can then read the available data and store for later processing. Since the DataReceived event is raised on a secondary thread the code will have to take multithreading into account when reading and storing the available data. And as an event implies, the code will also have to perform the reading and storing of received data as quickly as possible so as not to block the thread which raises the event. It is because there is very little time for processing during the DataReceived event handler's execution that the bytes read are stored for processing at a later time.

In this way the application defines an asynchronous read-write pattern with the serial port. This means that the user can potentially issue multiple commands while the MCU is processing the first command received, possibly resulting in DataReceived events which contain data for more than one command result. There is also now the issue of exactly when and how this "later processing" of the received data will occur. To address these issues, the application can introduce its own secondary thread which can provide the primary work execution for the application.

By utilizing a System.Threading.Tasks.Task object, the application can run three threads to handle all of the functionality requirements. The main thread will host the user interface and will be responsible for responding to user input, communicating that input to the secondary worker thread, updating display controls based on notifications from the secondary worker thread, and writing to the SerialPort component. The SerialPort component will raise its DataReceived event on its own secondary thread, and that thread will be responsible for reading data and communicating that data to the secondary worker thread. The secondary worker thread will have the responsibility of waiting for command results, processing the results, and notifying the main thread of updated display content.

↑ Return to Top

## "SerialDevice" Code Sample

The following code sample defines a SerialDevice class which encapsulates the SerialPort component and worker Task, along with the functionality outlined above. This class is meant to provide a lot of flexibility in usage and can benefit from performance improvements by redesigning it to fit a specific purpose. But as a general sample the performance should be adequate for most purposes. First we'll look at the code and then we'll go over the members in greater detail.

```
Option Strict On
Option Explicit On
Option Infer Off

Imports System.IO.Ports
Imports System.Collections.Concurrent
Imports System.Threading

''' <summary>
''' Provides a wrapper for a SerialPort which automatically reads data from the port and processes it via assigned Func(Of Byte(),
Integer, Integer) delegates.
''' Also provides "Send" methods for writing character, byte, hex, and string data to the port.
''' </summary>
''' <remarks></remarks>
Public Class SerialDevice
    Private WithEvents _Port As SerialPort
    Private _DataQueue As New ConcurrentQueue(Of Byte)
    Private _CancelSource As CancellationTokenSource
    Private _Task As System.Threading.Tasks.Task
    Private _Waiter As System.Threading.ManualResetEventSlim

    ''' <summary>
    ''' Gets a value determining if the serial port is open.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public ReadOnly Property Connected As Boolean
        Get
            Return _Port.IsOpen
        End Get
    End Property

    ''' <summary>
    ''' Gets or sets the COM Port name that the physical device is connected to, or virtual COM Port name created by the physical
device.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property ComPort As String
        Get
            Return _Port.PortName
        End Get
        Set(value As String)
            _Port.PortName = value
        End Set
    End Property

    ''' <summary>
    ''' Gets or sets a value determining whether or not the serial port will use data-terminal-ready signals (default is False).
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property DtrEnable As Boolean
        Get
            Return _Port.DtrEnable
        End Get
        Set(value As Boolean)
            _Port.DtrEnable = value
        End Set
    End Property

    ''' <summary>
    ''' Gets or sets the text encoding used to convert between character strings and byte data (default is ASCII).
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property Encoding As System.Text.Encoding
        Get
            Return _Port.Encoding
        End Get
        Set(value As System.Text.Encoding)
            _Port.Encoding = value
        End Set
    End Property

    ''' <summary>
    ''' Gets or sets the string used to represent a line terminator (default CrLf).
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property NewLine As String
        Get
            Return _Port.NewLine
        End Get
        Set(value As String)
            _Port.NewLine = value
        End Set
    End Property

    ''' <summary>
    ''' Gets or sets a value specifying the baud rate to use.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property Baud As Integer
        Get
            Return _Port.BaudRate
        End Get
        Set(value As Integer)
```

```
            _Port.BaudRate = value
        End Set
    End Property

    ''' <summary>
    ''' Gets or sets a value specifying the data bits to use.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property DataBits As Integer
        Get
            Return _Port.DataBits
        End Get
        Set(value As Integer)
            _Port.DataBits = value
        End Set
    End Property

    ''' <summary>
    ''' Gets or sets a value specifying the stop bits to use.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property StopBits As StopBits
        Get
            Return _Port.StopBits
        End Get
        Set(value As StopBits)
            _Port.StopBits = value
        End Set
    End Property

    ''' <summary>
    ''' Gets or sets a value specifying the parity to use.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property Parity As Parity
        Get
```

```
            Return _Port.Parity
        End Get
        Set(value As Parity)
            _Port.Parity = value
        End Set
    End Property

    ''' <summary>
    ''' Gets or sets a value specifying the hardware handshaking to use.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property FlowControl As Handshake
        Get
            Return _Port.Handshake
        End Get
        Set(value As Handshake)
            _Port.Handshake = value
        End Set
    End Property

    ''' <summary>
    ''' Gets or sets a value determining whether or not the serial port uses ready-to-send signals (default False).
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property RtsEnable As Boolean
        Get
            Return _Port.RtsEnable
        End Get
        Set(value As Boolean)
            _Port.RtsEnable = value
        End Set
    End Property

    ''' <summary>
    ''' Gets or sets the processing mode. In ByteMode, the CheckMessageComplete delegate will be invoked for every byte received.
    ''' In BlockMode, available data will be read at once and sent to CheckMessageComplete in chunks, as available.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property MessageMode As MessageProcessingMode

    ''' <summary>
    ''' Gets or sets a delegate function which will be called to determine if a complete message has been queued.
    ''' This function must return zero if the bytes received so far do not constitute a complete message, or a positive
    ''' integer value representing the number of bytes which make up the complete message.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks>
    ''' </remarks>
    ''' <example>
    ''' The following example defines a simple check message delegate function and sets it
    ''' to the CheckMessageComplete property.
    ''' <code>
    ''' mySerialDevice.CheckMessageComplete = AddressOf(DoCheckMessage)
    '''
    ''' Function DoCheckMessage(data As Byte(), count As Integer) As Integer
    '''    '1) Determine if data contains complete message
    '''    '2) Return number of bytes in message if complete, or return 0 if not complete
    ''' End Function
    ''' </code>
    ''' </example>
    Public Property CheckMessageComplete As Func(Of Byte(), Integer, Integer)

    ''' <summary>
    ''' An optional Action(Of PortState, SerialError) delegate to be invoked when a serial port error occurs.
    ''' Only as reliable as the SerialPort.ErrorReceived event. See MSDN documentation for details.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property ProcessError As Action(Of PortState, SerialError)
    ''' <summary>
    ''' A required Action(Of Byte(), Integer) delegate to be invoked when the CheckMessageComplete delegate call returns a positive
value.
    ''' This delegate method performs the actual work of processing the message data received from the serial device.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property ProcessMessage As Action(Of Byte(), Integer)
    ''' <summary>
    ''' An optional Action(Of PortState, SerialPinChange) delegate to be invokde when a serial pin change event occurs.
    ''' See the MSDN documentation for SerialPort.PinChanged event for more information.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Property ProcessPinChange As Action(Of PortState, SerialPinChange)

    ''' <summary>
    ''' Configures the serial port with the default baud rate (9800), 8 data bits, no parity, one stop bit, and no flow control.
    ''' </summary>
    ''' <remarks></remarks>
    Public Sub ConfigurePort()
        ConfigurePort(CommonBaudRate.Default, 8, Parity.None, StopBits.One, Handshake.None)
    End Sub

    Public Sub ConfigurePort(baud As CommonBaudRate)
        ConfigurePort(CInt(baud), 8, Parity.None, StopBits.One, Handshake.None)
    End Sub

    Public Sub ConfigurePort(baud As Integer)
        ConfigurePort(baud, 8, Parity.None, StopBits.One, Handshake.None)
    End Sub

    Public Sub ConfigurePort(baud As CommonBaudRate, dataBits As Integer, parity As Parity, stopBits As StopBits, flowControl As
Handshake)
        ConfigurePort(CInt(baud), dataBits, parity, stopBits, flowControl)
    End Sub

    Public Sub ConfigurePort(baud As Integer, dataBits As Integer, parity As Parity, stopBits As StopBits, flowControl As Handshake)
        _Port.BaudRate = baud
        _Port.DataBits = dataBits
        _Port.Handshake = flowControl
        _Port.Parity = parity
        _Port.StopBits = stopBits
    End Sub

    ''' <summary>
    ''' Creates a new instance of SerialDevice using the specified existing SerialPort instance.
    ''' </summary>
    ''' <param name="serialPort"></param>
    ''' <remarks></remarks>
    Public Sub New(serialPort As SerialPort)
        _Port = serialPort
        ConfigurePort()
    End Sub

    ''' <summary>
    ''' Creates a new instance of SerialDevice with its own internal SerialPort.
    ''' </summary>
    ''' <remarks></remarks>
    Public Sub New()
        Me.New(New SerialPort())
    End Sub

    ''' <summary>
    ''' Opens the SerialPort and connects to the physical device if the PortName, CheckMessageComplete delegate, and ProcessMessage
delegate have all been specified.
    ''' </summary>
    ''' <remarks></remarks>
    Public Sub Connect()
        If _Task Is Nothing Then
            If String.IsNullOrEmpty(_Port.PortName) Then Throw New InvalidOperationException("Cannot connect when COM Port is not
specified.")
            If CheckMessageComplete Is Nothing Then Throw New InvalidOperationException("Cannot connect when no IsMessageComplete
delegate has been specified.")
            If ProcessMessage Is Nothing Then Throw New InvalidOperationException("Cannot connect when no ProcessMessage delegate has
been specified.")

            _Port.Open()
            If _Port.IsOpen Then
                _CancelSource = New CancellationTokenSource
                _Waiter = New ManualResetEventSlim
                _Task = System.Threading.Tasks.Task.Factory.StartNew(New Action(Of Object)(AddressOf ProcessData), _MessageMode,
_CancelSource.Token)
            End If
        End If
    End Sub

    ''' <summary>
    ''' Ends processing of the SerialDevice and closes the underlying serial port.
    ''' </summary>
    ''' <remarks></remarks>
    Public Sub Disconnect()
        If _Task IsNot Nothing Then
            _CancelSource.Cancel()
            _Waiter.Set()
            _Task.Wait()
            _Port.Close()
            _Task.Dispose()
            _CancelSource.Dispose()
            _Waiter.Dispose()
            _Task = Nothing
            _CancelSource = Nothing
            _Waiter = Nothing
        End If
    End Sub

    ''' <summary>
    ''' Sends a single character to the serial port.
    ''' </summary>
    ''' <param name="c"></param>
    ''' <remarks></remarks>
    Public Sub SendChar(c As Char)
        _Port.Write({c}, 0, 1)
    End Sub

    ''' <summary>
    ''' Sends a byte or series of bytes to the serial port.
    ''' </summary>
    ''' <param name="bytes"></param>
    ''' <remarks></remarks>
    Public Sub SendData(ParamArray bytes() As Byte)
        If _Port.IsOpen Then
            _Port.Write(bytes, 0, bytes.Length)
        Else
            MessageBox.Show("The port is not open.", "Not Connected", MessageBoxButtons.OK, MessageBoxIcon.Error)
        End If
    End Sub

    ''' <summary>
    ''' Sends a byte or series of bytes in hexadecimal format to the serial port.
    ''' </summary>
    ''' <param name="hexString"></param>
    ''' <remarks></remarks>
    Public Sub SendHex(hexString As String)
```

```
        hexString = hexString.Replace(" ", "").ToUpper
        If Not hexString.Length Mod 2 = 0 Then
            hexString = "0" & hexString
        End If
        Dim result(CInt(hexString.Length / 2) - 1) As Byte
        Dim delta As Byte
        For i As Integer = 0 To result.Length - 1
            If Byte.TryParse(hexString.Substring(i * 2, 2), Globalization.NumberStyles.AllowHexSpecifier, Nothing, delta) Then
                result(i) = delta
            Else
                Throw New ArgumentException("Invalid hex string.")
            End If
        Next
        SendData(result)
    End Sub

    ''' <summary>
    ''' Sends a string to the serial port using the current text encoding.
    ''' </summary>
    ''' <param name="text"></param>
    ''' <remarks></remarks>
    Public Sub SendString(text As String)
        _Port.Write(text)
    End Sub

    Private Sub _Port_DataReceived(sender As Object, e As System.IO.Ports.SerialDataReceivedEventArgs) Handles _Port.DataReceived
        Dim portRef As SerialPort = DirectCast(sender, SerialPort)
        Dim data(portRef.BytesToRead - 1) As Byte
        Dim count As Integer = portRef.Read(data, 0, data.Length)
        For i As Integer = 0 To count - 1
            _DataQueue.Enqueue(data(i))
        Next
        _Waiter.Set()
    End Sub

    Private Sub _Port_Disposed(sender As Object, e As System.EventArgs) Handles _Port.Disposed
        _CancelSource.Cancel()
    End Sub

    Private Sub _Port_ErrorReceived(sender As Object, e As System.IO.Ports.SerialErrorReceivedEventArgs) Handles _Port.ErrorReceived
        If ProcessError IsNot Nothing Then
```

```
            _ProcessError(New PortState(_Port), e.EventType)
        End If
    End Sub

    Private Sub _Port_PinChanged(sender As Object, e As System.IO.Ports.SerialPinChangedEventArgs) Handles _Port.PinChanged
        If ProcessPinChange IsNot Nothing Then
            _ProcessPinChange(New PortState(_Port), e.EventType)
        End If
    End Sub

    Protected Sub ProcessData(modeObj As Object)
        Dim mode As MessageProcessingMode = DirectCast(modeObj, MessageProcessingMode)
        Dim currentMessage As New List(Of Byte)
        Do While Not _CancelSource.IsCancellationRequested
            Dim delta As Integer = currentMessage.Count
            Dim available As Integer = _DataQueue.Count
            If available > 0 Then
                Dim b As Byte
                If mode = MessageProcessingMode.ByteMode Then
                    If _DataQueue.TryDequeue(b) Then
                        currentMessage.Add(b)
                        CheckAndProcess(currentMessage)
                    End If
                ElseIf mode = MessageProcessingMode.BlockMode Then
                    While available > 0
                        If _DataQueue.TryDequeue(b) Then
                            currentMessage.Add(b)
                            available -= 1
                        End If
                    End While
                    If Not delta = currentMessage.Count Then
                        CheckAndProcess(currentMessage)
                    End If
                End If
            Else
                _Waiter.Wait()
                _Waiter.Reset()
            End If
        Loop
    End Sub
    Protected Sub CheckAndProcess(currentMessage As List(Of Byte))
        Dim removeCount As Integer = _CheckMessageComplete(currentMessage.ToArray, currentMessage.Count)
        If removeCount > 0 Then
            _ProcessMessage(currentMessage.Take(removeCount).ToArray, removeCount)
            currentMessage.RemoveRange(0, removeCount)
        End If
    End Sub
End Class

''' <summary>
''' Provides information about the state of a serial port.
''' </summary>
''' <remarks></remarks>
Public Structure PortState
    Public ReadOnly BreakState As Boolean
    Public ReadOnly CDHolding As Boolean
    Public ReadOnly CTSHolding As Boolean
    Public ReadOnly DSRHolding As Boolean
    Public ReadOnly IsOpen As Boolean

    Public Sub New(port As SerialPort)
        BreakState = port.BreakState
        CDHolding = port.CDHolding
        CTSHolding = port.CtsHolding
        DSRHolding = port.DsrHolding
        IsOpen = port.IsOpen
    End Sub
End Structure

''' <summary>
''' Provides a list of common serial baud rates.
''' </summary>
''' <remarks></remarks>
Public Enum CommonBaudRate
    [Default] = 9600
    bps2400 = 2400
    bps4800 = 4800
    bps9600 = 9600
    bps14400 = 14400
    bps19200 = 19200
    bps28800 = 28800
    bps38400 = 38400
    bps57600 = 57600
    bps115200 = 115200
End Enum

''' <summary>
''' Specifies the processing mode for messages.
''' </summary>
''' <remarks></remarks>
Public Enum MessageProcessingMode
    ''' <summary>
    ''' The CheckMessageComplete delegate will be invoked for every byte received.
    ''' </summary>
    ''' <remarks></remarks>
    ByteMode
    ''' <summary>
    ''' The CheckMessageComplete delegate will be invoked after all available data is read on each data received event.
    ''' </summary>
    ''' <remarks></remarks>
    BlockMode
End Enum
```

↑ Return to Top

## Code Description

The class begins by declaring the SerialPort component, then a thread-safe queue of bytes to use as the primary working data buffer, and finally the Task object along with the related CancellationTokenSource and ManualResetEvent needed to control the secondary thread's execution.  Next come simple properties to expose the primary members of the internal SerialPort instance.  These will be set by the user according to the requirements of their serial device.  The following MessageMode property holds a value from the MessageProcessingMode enumeration defined at the end of the code sample.  This value directs the SerialDevice class on how to process received data.  Next come the delegate properties which define the methods that will perform the actual data processing work.

With this design, when you declare an instance of SerialDevice in your application you will also write methods to do the work of processing messages from the serial port.  You will assign a delegate pointing to the work methods to the related SerialDevice properties.

The CheckMessageComplete delegate will be a method in your program that receives an array of bytes along with the number of bytes in the collection.  This method returns an integer which indicates the position within the collection where a complete message ends.  The secondary worker thread of the SerialDevice will call the CheckMessageComplete delegate and pass it all of the data received so far.  The application code can then analyze the data and decide whether or not a complete reply to the last sent command has been received.  If it has, the method returns the number of bytes in the collection which represent the complete message.  If it has not yet completed, the method should return zero and wait to be called again.

Once the CheckMessageComplete delegate returns a positive value, the indicated number of bytes will be removed from the working data queue and the ProcessMessage delegate will be executed and passed the complete message data.

The ProcessError and ProcessPinChange delegates provide an optional means of executing custom application code should the serial port raise an error event or if the application needs to monitor pin changes for a specific reason (typically this is not required).

Next come a few overrides of a method which can be used to configure the serial port for a connection.  These are followed by the class constructors which either create a new SerialPort instance or use an existing one (in most cases an application will allow the SerialDevice to create the SerialPort instance).

The class then exposes a Connect and Disconnect method which open the serial port and setup the secondary thread, or close the port and tear down the multithreading objects.  These methods are followed by a number of helpers for writing data to the serial port.  These helper methods cover the common requirements of sending characters, bytes, text, and bytes in hex format to the serial port.

We then come to the simple event handlers for the SerialPort object.  In the DataReceived event handler, we simply read all available data and add it to the working data queue.  The ManualResetEvent instance is then set to notify the secondary worker thread that it should start working again if it had been idle.  The other event handlers are rather self-explanatory and simply end the secondary thread should the SerialPort instance become disposed (should not happen) or call the appropriate delegate for the event if one has been defined.

Finally we get to the ProcessData method which represents the secondary thread work, and the CheckAndProcess helper method which calls the defined CheckMessageComplete and ProcessMessage delegates as necessary.

The ProcessData message declares a List(Of Byte) to build up the current MCU response.  This intermediate list holds the data to be checked and processed based on the setting of the MessageMode property.  If the MessageMode is set to ByteMode, then the CheckAndProcess helper is executed for every byte received.  If the MessageMode is set to BlockMode, the CheckAndProcess helper is only executed each time new data is queued and all queued data is sent to be checked.

The CheckAndProcess method then simply calls the CheckMessageComplete delegate and then updates the queue and calls the ProcessMessage delegate if the check returns a positive value.

↑ Return to Top

## Usage Example

A common scenario when working with a MCU is to send a command to configure or interrogate the device.  Many MCU's firmware will use a simple text-based protocol with all data transmitted in ASCII text and control codes implemented using ASCII format codes.  Commonly the user only needs to look for a single character such as Carriage Return (CR) or Line Feed (LF) or Zero (0) to indicate the end of a response.  To that end, we will build a simple program that allows us to build up a message from characters received from an MCU.

The following example will demonstrate the use of a SerialDevice instance in a Windows Forms Application.  The example will use a RichTextBox as a console screen allowing the user to type in characters which are passed to the MCU connected on COM3 (the characters are not displayed in the RichTextBox as they are typed).  The MCU then simply echos any received character back to the PC.  The SerialDevice CheckMessageComplete delegate will look for a CR character (13) to determine when a message is complete and the

ProcessMessage delegate will display the complete message in the RichTextBox. In this way, the program will buffer characters until the Enter key pressed and then the entire string will be written to the RichTextBox. Keep in mind though that each byte is passed through the MCU before being buffered by the SerialDevice for display in the RichTextBox.

```
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1
    Private WithEvents IOTextBox As New RichTextBox With {.BackColor = Color.Black, .ForeColor = Color.Lime, .Dock = DockStyle.Fill,
.ReadOnly = True}
    Private McuDevice As New SerialDevice

    Private Sub Form1_FormClosing(sender As Object, e As System.Windows.Forms.FormClosingEventArgs) Handles Me.FormClosing
        McuDevice.Disconnect()
    End Sub

    Private Sub Form1_Load(sender As System.Object, e As System.EventArgs) Handles MyBase.Load
        Controls.Add(IOTextBox)
        McuDevice.CheckMessageComplete = AddressOf IsMessageComplete
        McuDevice.ProcessMessage = AddressOf ProcessMessage
        McuDevice.ConfigurePort()
        McuDevice.ComPort = "COM3"
        McuDevice.MessageMode = MessageProcessingMode.BlockMode
        McuDevice.Connect()
    End Sub

    Private Function IsMessageComplete(data As Byte(), length As Integer) As Integer
        Dim index As Integer = Array.IndexOf(data, CByte(13))
        If index > -1 Then
            Return index + 1
        End If
        Return 0
    End Function

    Private Sub ProcessMessage(data As Byte(), length As Integer)
        Dim message As String = System.Text.Encoding.ASCII.GetString(data.ToArray)
        Invoke(New Action(Of String)(AddressOf AppendOutput), message)
    End Sub

    Private Sub AppendOutput(message As String)
```

```
        If IOTextBox.TextLength > 0 Then IOTextBox.AppendText(ControlChars.NewLine)
        IOTextBox.AppendText(message)
    End Sub

    Private Sub IOTextBox_KeyDown(sender As Object, e As System.Windows.Forms.KeyEventArgs) Handles IOTextBox.KeyDown
        e.Handled = True
    End Sub

    Private Sub IOTextBox_KeyPress(sender As Object, e As System.Windows.Forms.KeyPressEventArgs) Handles IOTextBox.KeyPress
        McuDevice.SendChar(e.KeyChar)
        e.Handled = True
    End Sub

    Private Sub IOTextBox_KeyUp(sender As Object, e As System.Windows.Forms.KeyEventArgs) Handles IOTextBox.KeyUp
        e.SuppressKeyPress = True
    End Sub
End Class
```

↑ Return to Top

## Summary

With this design, the SerialDevice class offers a versatile framework for communicating with a MCU or other serial device regardless of the particular data protocol employed.  A similar design could be used in a more purpose-specific way to more efficiently analyze and process the received data based on protocol-specifics unique to the application.  That said, this class should still be suitable for many quick-and-simple MCU configuration and/or interrogation utilities.

The example also shows how to implement a SerialPort component, and the basic event handling routines could be used directly on a form without the encapsulating "SerialDevice" class.  The SerialDevice class can be stripped apart into its constituent components in order to apply each concept to a another design.

↑ Return to Top

# See Also

An important place to find a huge amount of Visual Basic related articles is the TechNet Wiki itself. The best entry point is Visual Basic Resources on the TechNet Wiki