Top 5 SerialPort Tips [Kim Hamilton]

The SerialPort class requires some “warming up” time for our users coming from VB6 or other non-.NET backgrounds. One of the main reasons is that SerialPort and its underlying BaseStream are designed to behave similarly to other .NET streams, which (for example) means that Read() may not act as you’d expect. And of course there are other issues that developers occasionally run into.

The following are our top 5 SerialPort tips resulting from bugs we frequently we run into. Listed along with each tip is an example of a bug it applies to. We’re hoping this will help you diagnose problems in your app and save some time when working with this class.

(Note: the code samples below switch between VB and C# code samples, for no other reason than those were the language the bug was written in.)

**Tip 1:** When using SerialPort.Read(buffer, offset, count), where count is the number of bytes you want to read, check the return value, which tells you the number of bytes actually read.

Examples of bugs this tip addresses

* SerialPort.Read(buffer, offset, count) appears to return unexpected data

* SerialPort.Read(buffer, offset, count) does not wait for all of the bytes to be received

Developers sometimes assume that count bytes/chars will be returned when Readfinishes. Consider this code sample:

numBytes = myPort.Read(readBuffer, 0, **readBuffer.Length**)

For i As Integer = 0 To **readBuffer.Length** – 1

        Console.Write(readBuffer(i) & “.”)

    Next i

The above code doesn’t check numBytes, the number of bytes actually read in the call to Read. So if Read returned fewer bytes than the length of readBuffer, the above loop will print out values that were in readBuffer before the call to Read.

Here’s what Read actually does. If there are bytes available on the serial port, Read returns up to count bytes but will not block for the remaining bytes. If there are no bytes available on the serial port, Read will block until at least one byte is available on the port, up until the ReadTimeout milliseconds have elapsed, at which time a TimeoutException will be thrown.

To fix this in your code, check number of bytes actually read and use that value when processing the returned data. In the above code sample, you’d replace the For loop line with this:

    For i As Integer = 0 To **numBytes** – 1

**Tip 2.**When using DataReceived event handlers, don’t assume you’ll receive an event for every byte received (assuming you’re using the default ReceivedBytesThreshold of 1)

Example bug this tip addresses

SerialPort appears to drop data when using DataReceived event handlers

First, assume that you leave the ReceivedBytesThreshold at the default value, which is 1. This means that if you get a DataReceivedevent, you can call Read and there will be at least 1 byte available. It doesn’t mean that you’ll get a DataReceived event for every incoming byte. For example, if several bytes arrived at the serial port at the same time, you’d receive just one event, not one for each byte.

In general, if you set the ReceivedBytesThreshold to some value, say n, the interpretation is the same: you will get a DataReceived event when at least n bytes are available; not one per every n bytes.

If apps don’t account for this, it may appear that the SerialPort is dropping data. The following DataReceived handler reads only one byte per DataReceived event. But if more data were available, it’s possible that the reader won’t get all the data when the app has finished all of its reads.

    private void DataReceivedHandler(object sender,

                 SerialDataReceivedEventArgs e) {

        byte b = (byte)myPort.ReadByte();

    }

To fix this in your code, use the SerialPort methods that let you read or keep reading until you’ve gotten all available data. One way allows you to get the raw bytes and the other way will convert the bytes to a string in the encoding specified (specified on the SerialPort object)

* To get the raw bytes: The BytesToRead property tells you how many bytes are available. Then you can callSerialPort.Read(buffer, offset, count) with a buffer big enough to hold the bytes (or call over several iterations).

* To get a string: ReadExisting reads all available bytes and returns a string using the encoding you specify on the SerialPort. This is more convenient if you want to convert to a string anyway.

**Tip 3.** Take care to avoid deadlock when calling Close on the SerialPort in response to a GUI event.

Examples of bugs this tip addresses

* An app involving the UI and the SerialPort freezes up when closing the SerialPort

* Deadlock can occur if Control.Invoke() is used in serial port event handlers

This is probably the most frustrating SerialPort issue because it may happen so rarely that it’s hard to diagnose. Deadlocks can happen if you have an event handler trying to work with a GUI control while the GUI thread is trying to close the SerialPort. This happens in the following example.

public partial class Form1 : Form

{

        SerialPort m\_port;

        public Form1()

        {

            InitializeComponent();

            m\_port = new SerialPort(“COM2″, 115200);

            m\_port.DataReceived += new

                 SerialDataReceivedEventHandler(m\_port\_DataReceived);

            m\_port.Open();

        }

        void m\_port\_DataReceived(object sender,

                 SerialDataReceivedEventArgs e)

        {

**this.Invoke**(new EventHandler(delegate

            {

                m\_port.Close();

            }));

        }

}

This example looks a bit strange, but it forces the deadlock to occur. The typical scenario we encounter is *occasional* deadlock in an app that has a data received handler trying to update the GUI at the same time the GUI thread is trying to close the SerialPort (for example, in response to the user clicking a Close button).

The reason deadlock happens is that Close() waits for events to finish executing before it closes the port. You can address this problem in your apps in two ways:

* In your event handlers, replace every Control.Invoke call with Control.BeginInvoke, which executes asynchronously and avoids the deadlock condition. This is commonly used for deadlock avoidance when working with GUIs.

* Call serialPort.Close() on a separate thread. You may prefer this because this is less invasive than updating your Invoke calls.

**Tip 4.** .NET 2.0 isn’t letting you get away with some things, such as attempting to cancel a SerialPort read by interrupting the thread accessing the SerialPort. (Note that pre-2.0, canceling a thread reading from the SerialPort resulted in an exception you could catch but you weren’t warned of worse things going on behind the scenes such as leaked objects.)

Example bug this tip addresses

Sudden application shutdown when interrupting a thread accessing the SerialPort

We’ve had a few bugs reporting sudden shutdown in apps – most often it’s caused by performing unsupported behavior such as interrupting a thread accessing the SerialPort. The specific symptom in this case is an ObjectDisposedException thrown on a separate thread that cannot be caught.

This behavior is related to a new managed thread exception handling policy in .NET 2.0, intended to prevent exceptions caused in background threads from being swallowed and slowly degrading application state; instead the program terminates immediately. Note that this is listed as a breaking change of .NET 2.0: <http://msdn2.microsoft.com/en-us/netframework/aa497241.aspx>

We’ll address this issue more fully in a future blog post. We’ll explain why shutting down your application is actually a good thing and provide some more options of how to deal with this in your apps. The ideal way to fix this class of problems is to fix the bug that led to the exception. However, some users need a quick solution to stop their app from breaking until they can fix the bug. A quick workaround is to add the following line to your application’s config file to get the earlier behavior and be able to catch the exception.

<?xml version =“1.0“?>

<configuration>

  <runtime>

**<legacyUnhandledExceptionPolicy enabled=“1“/>**

  </runtime>

</configuration>

**Tip 5.** This isn’t a tip as much as an attempt to clear up some confusion we may have caused. SerialPort.ReadByte will either return abyte or throw an exception if a byte can’t be read; it never returns -1. The exceptions SerialPort can return are:

* InvalidOperationException: port isn’t open

* TimeoutException: timeout occurred

* IOException: general IO errors; will be accompanied by specific error message

* UnauthorizedAccessException: access to port denied

* EndOfStreamException: end of stream

First, some users wonder why ReadByte doesn’t return a byte (it returns an int, and you have to cast to a byte). As background, theint return type of the underlying BaseStream matches the abstract Stream class definition of ReadByte, allowing a return value less than zero in the case that no data is available.

However, SerialPort,ReadByte() will never actually return -1; it will return a byte or one if the exceptions listed above. If you’re fact checking this against our docs right now, you’ll notice an inconsistency, and we’ve just opened up a doc bug on this.

**Bonus Tip:**  
Check out other recent BCL blog posts on SerialPort: