THORLABS

Application Note

DM713 Digital Micrometer: C# Programming Overview

This application note provides a programming reference for interfacing a computer with the DM713 Digital Micrometer using Visual C#. An example C# application is presented, step-by-step, that connects to the DM713 via a communications port, requests a continuous stream of output data from the DM713, extracts the displacement values and measurement units from the returned data, displays the displacement values, and provides the option to log selected displacement values to a file.

This example program is provided as a reference. The user is encouraged to extend or modify the program to fit the specific needs of the application.

August 14, 2019

1 Preface

This application note provides an introduction to using Visual C# to communicate with the DM713 Digital Micrometer. In Section 2, a step-by-step discussion of an example C# program is presented. The full program text is provided in Section 3 without explanation.

The program connects the computer to the COM port of the DM713, receives the stream of data output by the micrometer, operates on the data, displays the results, and allows the user to log selected results. A secondary thread is spawned to implement continuous communication with the micrometer. The stream of output data collected from the micrometer includes displacement measurements and information about the unit of measurement. This program extracts the displacement measurements and determines their units. These displacement values are then displayed in the program's graphical user interface (GUI) with the appropriate units. Logging a displacement value to a file in response to the click of a button was also implemented.

Supported functionality and procedures may differ from those described in this application note if different hardware, firmware, or software versions are used. The versions used to develop this program example were:

- DM173 Hardware / Firmware Version 350-357-30
- Microsoft® Visual Studio Version 15.5.7
- Microsoft® .NET Framework Version 4.6.1

2 Step-by-Step Instructions for Building the Application

The code presented in Section 2.1 acquires data from the DM713, extracts the embedded displacement values, and displays these values along with their units. A screen shot of the running program is shown in Figure 1. Section 2.2 describes the portion of the program that implements the data logging functionality.

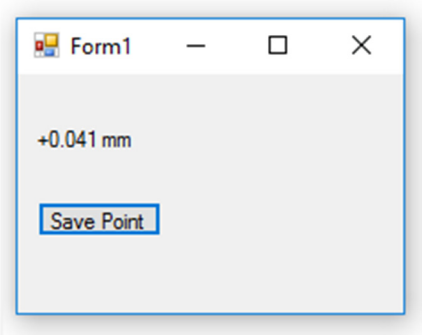

Figure 1 *This screen shot shows the completed example application running.*

2.1 Acquiring and Displaying Measurement Data

The steps required to create a C# application, which receives a stream of output data from the DM713 and displays the displacement values in the UI, are described in this section.

1. Begin a New Project

Connect the DM713 Digital Micrometer to the computer using an RS-232 cable, or an RS-232 to USB converter.

Wait until the computer recognizes the micrometer. If the computer is running a Microsoft® Windows operating system, the DM713 will be listed in the device manager under **Ports (COM and LPT)** with a name that depends on the cable. The cable In this case was a Prolific USB-to-Serial cable, as shown in Figure 2. Make a note of the COM port, which is COM2 in this case.

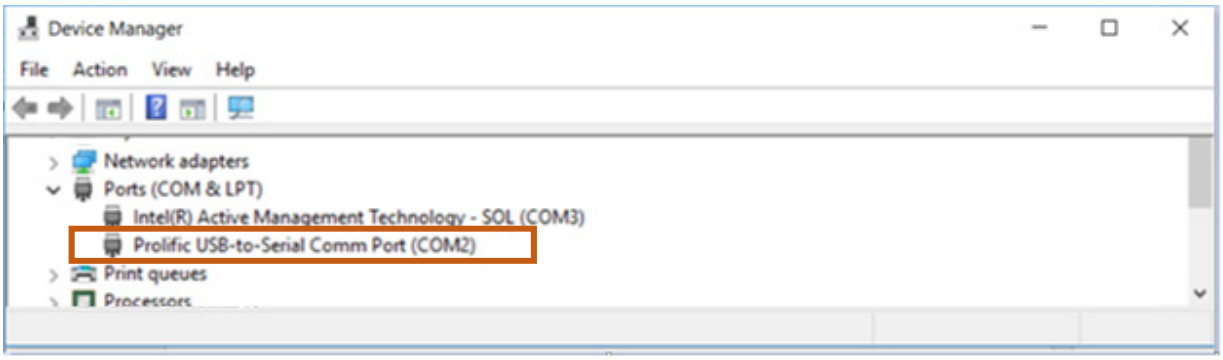

Figure 2 *The serial port used to communicate with the DM713 can be found under the* ***Ports*** *heading in the Windows® Device Manager. In this case, the port is COM2.*

Open Microsoft Visual Studio and start a new project. This can be done by navigating the File menu (**File -> New -> Project**) or by clicking on the **create new project** link shown in the lower right corner of Figure 3.

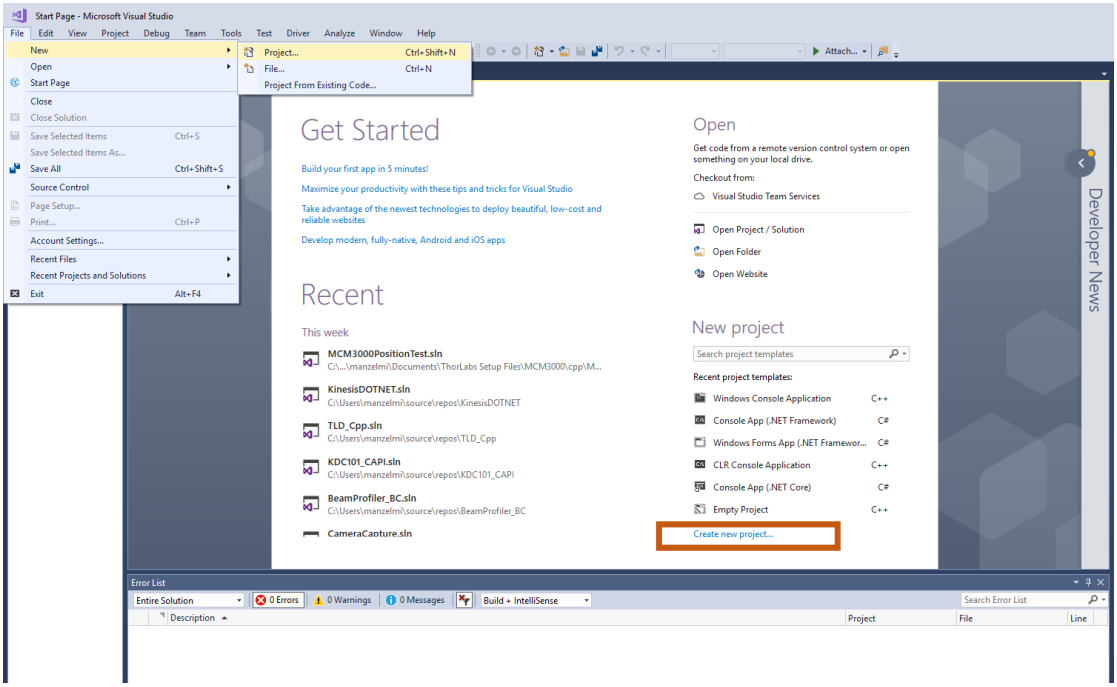

Figure 3 *Visual Studio's starting screen can be used to open a new project by clicking on the* ***create new project*** *link shown in the lower right corner or navigating the* ***File*** *menu.*

2. Start a New Windows Form Application

In the **New Project** window, select the *Windows Forms App (.NET Framework)* template. Enter an appropriate name for the project in the **Name** field. For this example, as shown in Figure 4, the project name *DM713DataLogger* was chosen.

By default, this project name will also be the name of the namespace applied to the class files added to this project.

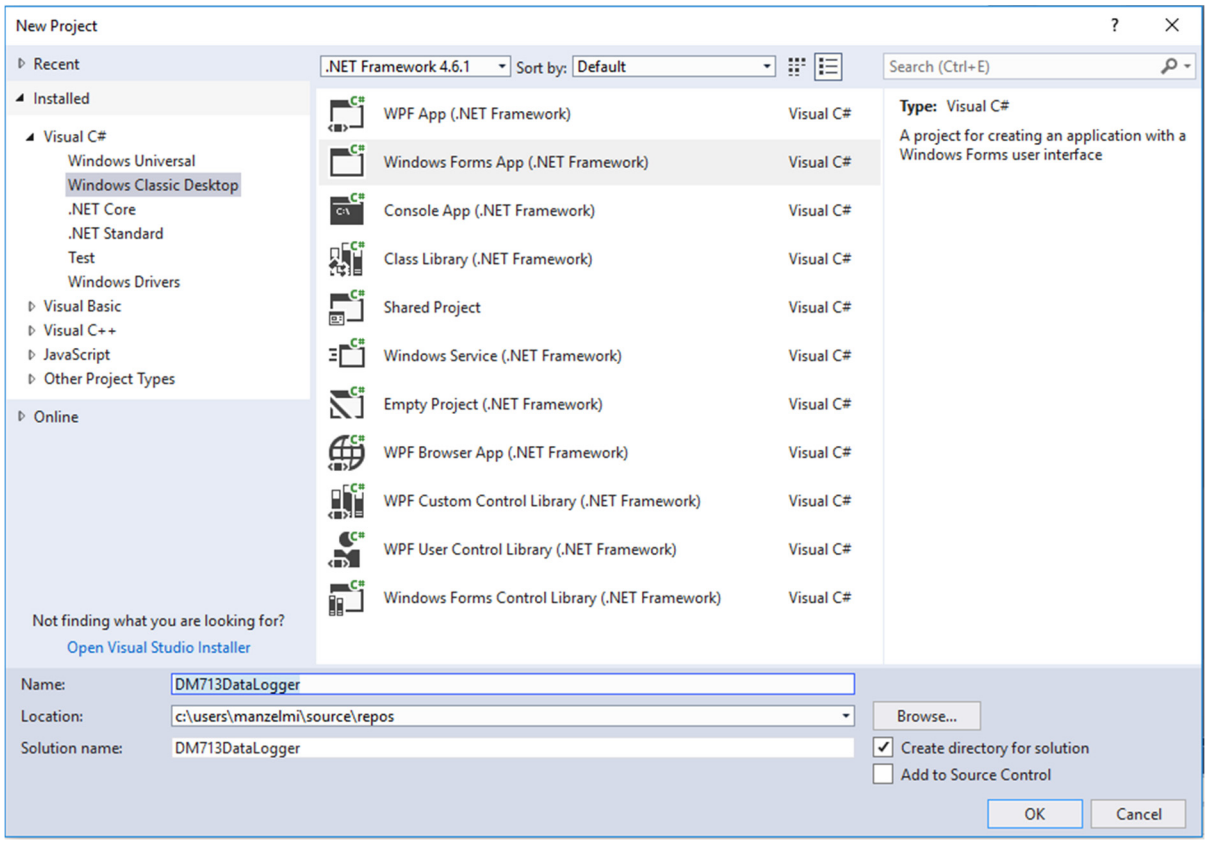

Figure 4 *During the creation of this project, the* Windows Forms App (.NET Framework) *template was selected, and the project was named DM713DataLogger.*

3. Visual Studio Editor

The new project will look similar to the empty project shown in Figure 5. The yellow box on the right encloses the two tabs of the **editor window**. The open tab is a forms designer used to create the form's GUI. The other tab is the corresponding code editor. The tabs are named with the file name, rather than the project name.

The forms designer is used to add all necessary runtime controls to the UI. These elements can be dragged into the application window directly from the toolbox, which is shown on the left of the project window and enclosed in a green box. Controls can only be dragged into the application window, which is enclosed by the purple box, and not into the whitespace outside of the application window.

The size of the form window is the size of the runtime UI window. Adjust the size of the window as desired. In this example, the size of the forms window was adjusted to fit a couple of controls.

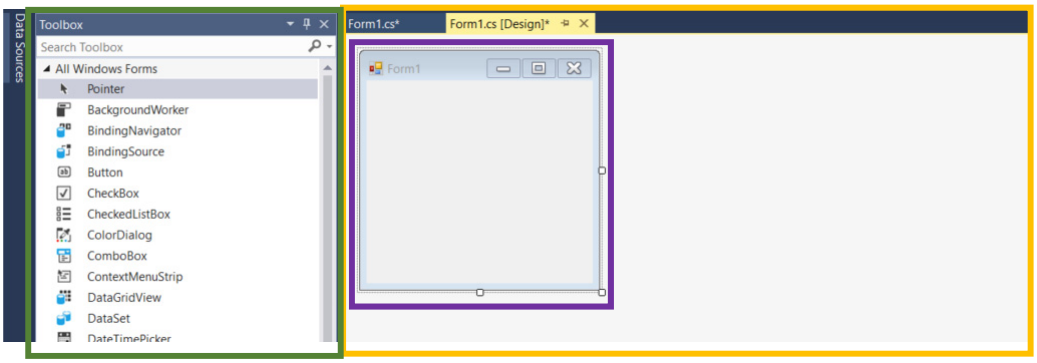

Figure 5 *The project window includes an editor window (on right, enclosed by yellow box) and a toolbox (on left, enclosed by green box). The active tab in the editor window is the forms designer, which is used to create the GUI. The application window (enclosed by purple box) has been sized to accommodate a couple of controls. Controls from the toolbox can be dragged and dropped into the application window.*

4. Insert a Label into the Form

Use the toolbox (green box in Figure 6) to search for the **label** control, and then drag and drop a label into the form window (purple box). A label is a control which can be used to display text, and it will be used in this application to display the displacement values acquired from the micrometer. In the forms designer of Figure 6 the text "label1" is displayed on the label.

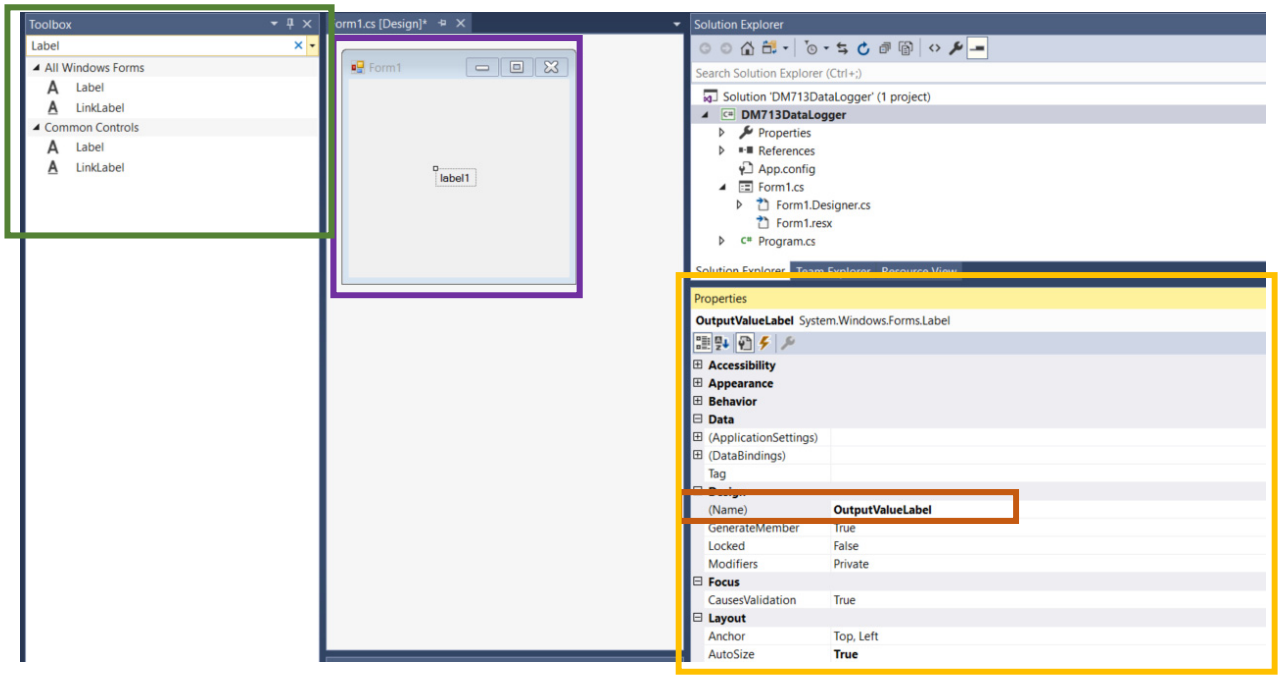

Figure 6 *In this form, an object called a **label** will be used to display the micrometer's displacement values in the form. Drag and drop a label from the toolbox into the form window. Specify an identifier for the label in the code by using the **(Name)** field under the **Design** heading in the **Properties** window.*

The label's identifier in the code can be specified using the **Properties** window (yellow box). Expand the options under the **Design** heading, and enter an appropriate identifier into the **(Name)** field (orange box). In this example, the identifier *OutputValueLabel* was chosen.

5. Open Code Editor and Create Option to Handle User Clicks on the Label

An easy way to open the code editor is to double click the label in the forms designer. This action also auto-creates a **using** block (See Step 6) and couple of methods, including an override method that handles clicks on the label occurring during program runtime. The name of this override method is the name of the identifier entered in Step 4. In this example, since the override method is not used, it is deleted. It should be noted that this override method could be useful for other applications.

A screen shot of the code editor is shown in Figure 7.

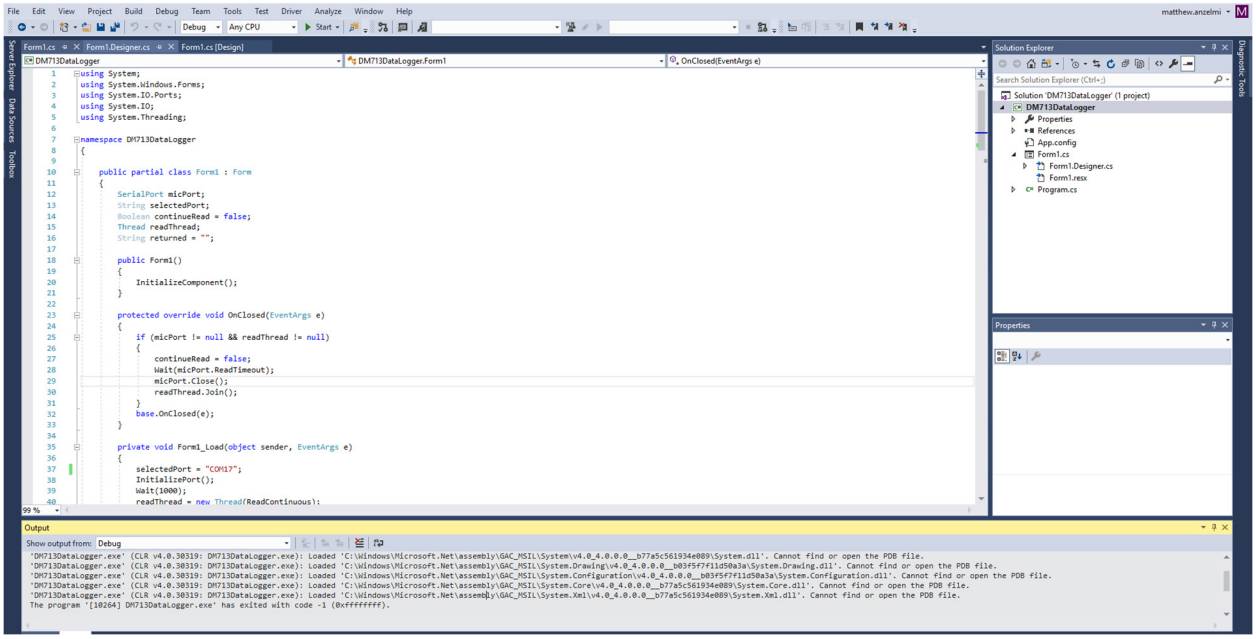

Figure 7 *This screen shot shows the code editor.*

6. Customize the List of **using** *Directives to Include Desired Namespaces*

In the code editor, the block of **using** directives may include some **namespaces** that are not needed and may be missing some that are required. Modify the **using** block to contain:

```
using System;
using System.Windows.Forms;
using System.IO.Ports;
using System.IO;
using System.Threading;
```

Note that

```
namespace DM713DataLogger
```

follows the **using** block. All methods added to this form will be contained in the *DM713DataLogger* **namespace**, whose name defaults from the project name.

7. Make Some Declarations in Class *Form1*

The class *Form1* will contain all methods in this form. At the beginning of the class it is necessary to declare the serial port object (*micPort*) and a string (*selectedPort*) containing the specific COM port designator. After the class definition statement,

```
public partial class Form1 : Form
```

and after the opening parenthesis, add:

```
SerialPort micPort;
String selectedPort;
```

The *SerialPort* class is defined in the *System.IO.Ports* namespace. The *selectedPort* string provides an easy way to set the name in the *initializePort* method discussed in Step 8. This string declaration provides an alternative to hard-coding a port name.

8. Initialize the Serial Port (micPort)

the *InitializePort* method is created in the *Form1* class to help initialize the serial port (*micPort*), which was declared in Step 7. The *Form1_Load* method, which is also in the *Form1* class and should have been auto-generated, will be modified in Step 15 to call this *InitializePort* method and to make a string assignment to *selectedPort*.

The *MessageBox* method, which is included in the *System.Windows.Forms* namespace, is used to notify the user if the port is unavailable. In this case, a pop-up will be displayed with an error message and icon, and then the form will close.

```
private void InitializePort()
{
    micPort = new SerialPort();
    micPort.PortName = selectedPort;
    micPort.DataBits = 8;
    micPort.StopBits = StopBits.One;
    micPort.BaudRate = 2400;
    micPort.Parity = Parity.None;
    micPort.Handshake = Handshake.None;
    micPort.RtsEnable = true;
    micPort.ReadTimeout = 500;
    micPort.WriteTimeout = 500;
    try
    {
        micPort.Open();
    }
    catch (IOException)
    {
        MessageBox.Show("Port could not be opened", "Port Error",
             MessageBoxButtons.OK, MessageBoxIcon.Error);
        this.Close();
    }

}
```

9. Communicate with the Micrometer

Two methods are created in the *Form1* class to enable communication with the micrometer. The first method is *RequestValue*, which requests a data value from the micrometer by sending a byte. The specific value of the byte that is sent does not matter, and the string *1* was chosen for this example. It is good practice to include the try-catch structure to handle any thrown

timeout exceptions, but including the try-catch structure is not strictly necessary for implementing the *RequestValue* method.

```
private void RequestValue()
{
    try
    {
        micPort.Write("1");
    }
    catch (System.TimeoutException)
    { }
}
```

The second method, *ReadValue*, reads the return string from the port. Since there is the risk that all of the bytes may not be immediately available, surrounding the read command with the try-catch structure to handle timeout exceptions is required. The *ReadValue* method passes the acquired data value to the *RemoveUnneededChars* method, which is discussed in Step 10.

```
private String ReadValue()
{
    String output = "";
    try
    {
        output = micPort.ReadExisting();
    }
    catch (System.TimeoutException)
    { }
    return RemoveUnneededChars(output);
}
```

10. Extract Displacement Value from the Output Value

The data value output by the micrometer includes a number of characters not of interest to this application. An example output string,

01A-0002.394\r\n

illustrates the format of the output data. From the left,

- The first three characters (01A) are not needed.
- The fourth character provides the sign value (-), and is needed. The sign will be + or -.
- The number of leading zeros (000) is variable, and these zeros are not needed.
- The fractional number following the zeros (2.394) is needed.
- The terminating characters (\r\n) are not needed.

The number of the characters between the decimal point and the terminating characters is different depending on whether the measurement units are millimeters or inches. The above displacement value is in millimeters. There are more decimal places when the inches are the units of measurement. The unit of measurement is determined in Step 11.

The *RemoveUnneededChars* method concatenates the required characters into a string called *tmp*. It is known that the first three characters in the *output* string are not needed, and the fourth character is always needed. This sign character is immediately assigned to the string *sign* and later concatenated with the value written to *tmp*. The characters in the output data following the sign character are searched to find the first subsequent character that is either

non-zero or a decimal point. When the index of this character is found, the *Substring* method is used to retrieve the substring that starts at this index and concludes at the end of the string. The *Trim* method is then used to remove the trailing terminating characters \r\n. Both *Substring* and *Trim* are included in the *System* namespace.

By selecting the entire remainder of the string, and then removing the terminating characters from to *tmp*, it can be ensured that no digits of the output displacement value will be lost.

```
private String RemoveUnneededChars(String output)
{
    String tmp = "";
    String sign = "";
    if (output.Length > 0) //Check that the returned string from the port is not blank
    {
        sign = output[3].ToString(); //store the sign of the value to above string
        for (int i = 3; i < output.Length; i++)
        // Search through the string until the decimal place or first nonzero number is
        //found
        {
            if (output[i] == '.')
            {
                tmp += output.Substring(i);
                break;
            }
            else
            {
                try
                {
                    int parsed = (int)char.GetNumericValue(output[i]);
                    if (parsed > 0) //if the parsed string is  a number and greater than
                     //0, add it to the output
                    {
                        tmp += output.Substring(i);
                        break;
                    }
                }
                catch (FormatException) { }
            }
        }
    }
    return sign + tmp.Trim('\r','\n'); //add the sign and remove the trailing terminating
    //characters
}
```

11. Determine the Units of Measurement from the Output Value

The *Units* method, created in the *Form1* class, determines the units of measurement by counting the number of characters between the decimal point and the terminating characters in the data value output by the micrometer. The string output by the *RemoveUnneededChars* method, which was discussed in Step 10, is passed to the *Units* method. The *Units* method finds the length of the substring that starts at the index following the decimal place and concludes at the last character of the string, which includes no terminating characters. If the length of this substring is greater than four, the displacement reading was in inches. Otherwise, millimeters are the units of measurement.

```
private String Units(String output)
{
    int indexOfDecimal = output.IndexOf(".");
    String tmp = "";
    if (indexOfDecimal != -1)
    {
        if (output.Substring(indexOfDecimal+1).Length > 4)
        {
            tmp = " in";
        }
        else {
            tmp = " mm";
        }
    }
    return tmp;
}
```

12. Add Some Declarations to the Beginning of the Class Form1

The application will spawn a new thread to allow continuous communication with the micrometer. New variables declared at the start of the *Form1* class,

```
public partial class Form1 : Form
```

that are associated with this second thread are the *readThread* thread and *continueRead* Boolean. With these two additions shown in bold, the block of declarations is now:

```
SerialPort micPort;
String selectedPort;
Boolean continueRead = false;
Thread readThread;
```

13. Set the Text Shown on the Label

The purpose of the **label** added to the form in Step 4 is to display the displacement value provided by the micrometer. The *SetText* method updates the text on the label. Recall that *OutputValueLabel* is the identifier given to the **label** in Step 4. The *SetText* method must ensure the **label** is not being handled by the UI thread at the same time the text displayed by the **label** is being updated. If *InvokeRequired* returns true, the handling will be passed to the delegate method.

The *Delegate* Method:

```
delegate void StringArgReturningVoidDelegate(string text);
```

The *SetText* Method:

```
private void SetText(string text)
{
    if (this.OutputValueLabel.InvokeRequired)
    {
        StringArgReturningVoidDelegate d = new StringArgReturningVoidDelegate(SetText);
        this.Invoke(d, new object[] { text });
    }
    else
    {
        this.OutputValueLabel.Text = text;
    }
}
```

14. Make Continuous Readings of the Data Output by the Micrometer

Continuous readings of the data output by the micrometer are performed by the *ReadContinuous* method. Since the *continueRead* Boolean variable is set true in the *Form1_Load* method, which is discussed in Step 15, the *ReadContinuous* method will continuously loop until the form is closed.

The *ReadContinuous* method calls the *RequestValue* and *ReadValue* methods described in Step 9 to acquire the data value and extract the displacement reading. The string *value* is set equal to the displacement reading. The units of the displacement reading are determined by calling the *Units* method, which is discussed in Step 11, and concatenated with the value string. The text displayed on the **label** in the form is then updated to *value* by calling The *SetText* method, which is discussed in Step 13.

```
public void ReadContinuous()
{
    while (continueRead)
    {
        Wait(300);
        RequestValue();
        Wait(300);
        String value = ReadValue();
        value += Units(value);
        SetText(value);
    }
}
```

The *Wait* method called by the *ReadContinuous* method sets the time delay between reads and writes. This is implemented using the built-in *Timer* class that is included in the *System.Windows.Forms* namespace. The wait value should be a minimum of 300 ms to allow the device to receive the request and return a data value.

```
public void Wait(int milliseconds)
{
    System.Windows.Forms.Timer timer1 = new System.Windows.Forms.Timer();
    if (milliseconds == 0 || milliseconds < 0) return;
    timer1.Interval = milliseconds;
    timer1.Enabled = true;
    timer1.Start();
    timer1.Tick += (s, e) =>
    {
        timer1.Enabled = false;
        timer1.Stop();
    };
    while (timer1.Enabled)
    {
        Application.DoEvents();
    }
}
```

15. Initialize the Thread and Begin Data Acquisition

The *Form1_Load* method is a part of the *Form1* class. It assigns a name to the port, calls the *InitializePort* method discussed in Step 8 to initialize the serial port, and it initializes the second thread declared in Step 12 so that data acquisition can begin. The name of the specific COM port found in Step 1 is assigned to the *selectedPort* string in this method. After changing the value of the *continueRead* Boolean to true, this method uses the *readThread.Start* method to

begin execution of a new thread. This results in two threads running concurrently, with this new thread executing the *ReadContinuous* method discussed in Step 14. The one second wait is used to allow a proper connection to be established before requesting values.

```
private void Form1_Load(object sender, EventArgs e)
{
    selectedPort = "COM2";
    InitializePort();
    Wait(1000);
    readThread = new Thread(ReadContinuous);
    continueRead = true;
    readThread.Start();
}
```

16. Handle the Addition of Events

This method, which is needed to safely run the program, is included in the *Form1* class and overrides the *OnClosed* method included in the *System.Windows.Forms* namespace. *Override* is a modifier that can be used to modify the implementation of an inherited method. In this case, it is used to add several events, including stopping and joining the thread. The wait timer is used to allow any possible in-progress read or write actions to conclude so that a *Closed Port* exception is not thrown.

```
protected override void OnClosed(EventArgs e)
{
    if (micPort != null && readThread != null)
    {
        continueRead = false;
        Wait(micPort.ReadTimeout);
        micPort.Close();
        readThread.Join();
    }
    base.OnClosed(e);
}
```

17. Run the Program

Running the program should now open up a form, which includes a numerical read out. The value of the readout should be continuously updated with the displacement value read by the micrometer.

2.2 Logging the Displacement Data to a File

In this section, a button is added to the form that allows the user to save selected displacement values to a file. Using a single method to open the file and save a displacement value ensures that the file is never left open, and there is no data lost, when the program exits.

18. Add a Button to the Form

The example program is expanded to enable the displayed displacement value to be saved to a file when a button is clicked. Use the toolbox to search for the **button** control (green box in Figure 8) and then drag and drop a **button** into the form window (purple box). In this example, *Save Point* is displayed on the button. The text displayed on the **button** can be customized via the **Properties** window (yellow box), by expanding the **Appearance** section and updating the **Text** field.

The identifier for the **button** used in the code can also be customized using the **Properties** window. Expand the options under **Design**, and enter an appropriate identifier into the **(Name)** field (orange box). In this example, the object has been named *SavePointButton*.

Double click the **button** in the form designer window to generate the event method in the code. The method name will be *(button name) + _click*, which is *SavePointButton_Click* in this example. This method is modified in Step 21.

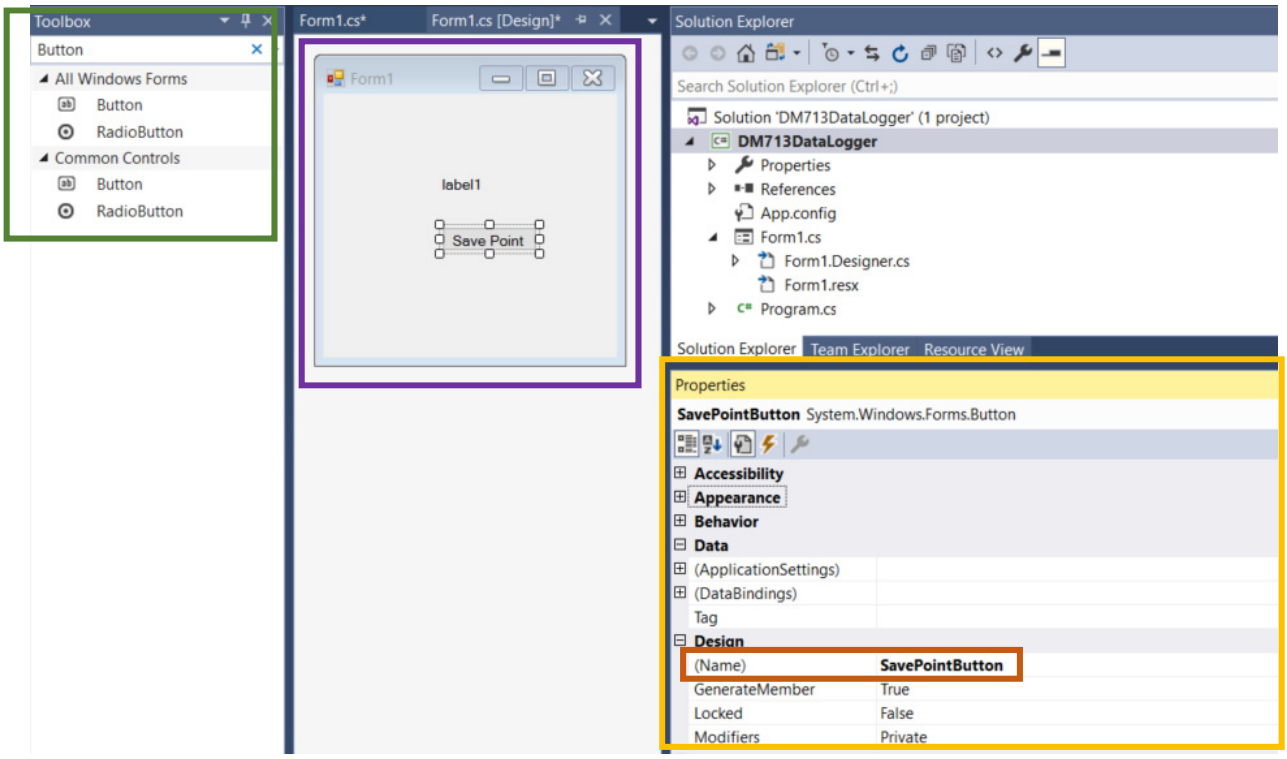

Figure 8 *A button will be used to log individual data points to a file. Drag and drop a* ***button*** *from the toolbox into the form window. Specify the identifier for this* ***button*** *in the code by entering the identifier into the* ***(Name)*** *field under the* ***Design*** *heading in the* ***Properties*** *window.*

19. Declare a String to Hold the Displacement Value

Next, an another declaration is added to the block of variable declarations included at the start of the *Form1* class,

```
public partial class Form1 : Form
```

The new *returned* string is declared and made available to the class so that it can provide a bridge between the *Save Point* **button**, which was added in Step 18, and text displayed on the **label**. The updated block of declarations, in which the new line is shown bolded, is:

```
SerialPort micPort;
String selectedPort;
Boolean continueRead = false;
Thread readThread;
String returned = "";
```

20. Assign the Displacement Value to the returned *String*

Since the value of the *returned* string, which is discussed in Step 19, and the value displayed on the **label** must be the same, the *ReadContinuous* method discussed in Step 14 is modified to set the value of the *returned* string to the displacement value followed by the units of measurement. The *ReadValue* method is called only once in the *ReadContinuous* method, which ensures the value displayed on the **label** and the logged value will be the same at time of save. The revised *ReadContinuous* method, with the new line bolded, is:

```
public void ReadContinuous()
{
    while (continueRead)
    {
        Wait(300);
        RequestValue();
        Wait(300);
        String value = ReadValue();
        value += Units(value);
        returned = value;
        SetText(value);
    }
}
```

21. Save the Displacement Value to a File when the Button is Clicked

The *SaveButton_Click* method was auto-created in the *Form1* class during Step 18, when the **button** in the form designer was double clicked. This method is revised here to define a file path and to check whether the file exists before writing the displacement value. If the file does not exist, it will be created. If the file exists, a new line containing the data value will be added.

In this example, the path and file name are hard coded. Alternatively, a text input and folder chooser option can be added if desired. The *StreamWriter* and *File.Create* classes are included in the *System.IO* namespace.

```
private void SavePointButton_Click(object sender, EventArgs e)
{
    string folderPath = "C:\\Users\\jdoe\\Documents";
    string filename = "TestFile99.txt";
    string fullPath = folderPath + "\\" + filename;

    if (File.Exists(fullPath))
    {
        using (var Tw = new StreamWriter(fullPath, true))
        {
            Tw.WriteLine(returned + "\t" + System.DateTime.Now.ToString("h:mm:ss tt") + "\t" +
                System.DateTime.Now.ToLongDateString());
        }
    }
    else
    {
        FileStream Fs = File.Create(fullPath);
        Fs.Close();

        using (var Tw = new StreamWriter(fullPath, true))
        {
            Tw.WriteLine("Position \t Timestamp \t Date");
            Tw.WriteLine(returned + "\t" + System.DateTime.Now.ToString("h:mm:ss tt") + "\t" +
                System.DateTime.Now.ToLongDateString());
        }
    }
}
```

22 Run the Program

Running the program and clicking the Save Point **button** should write the selected displacement value to the specified file.

3 Complete Program Code

The program code included in this section acquires, processes, and displays a continuous stream of displacement values output by a DM713 Digital Micrometer. In addition, a button is included that allows the user to save selected data points to a log file during runtime.

```
using System;
using System.Windows.Forms;
using System.IO.Ports;
using System.IO;
using System.Threading;

namespace DM713DataLogger
{

    public partial class Form1 : Form
    {
        SerialPort micPort;
        String selectedPort;
        Boolean continueRead = false;
        Thread readThread;
        String returned = "";


        public Form1()
        {
            InitializeComponent();
        }


        protected override void OnClosed(EventArgs e)
        {
            if (micPort != null && readThread != null)
            {
                continueRead = false;
                Wait(micPort.ReadTimeout);
                micPort.Close();
                readThread.Join();
            }
            base.OnClosed(e);
        }


        private void Form1_Load(object sender, EventArgs e)
        {
            selectedPort = "COM2";
            InitializePort();
            Wait(1000);
            readThread = new Thread(ReadContinuous);
            continueRead = true;
            readThread.Start();
        }
```

```
private void InitializePort()
{
    micPort = new SerialPort();
    micPort.PortName = selectedPort;
    micPort.DataBits = 8;
    micPort.StopBits = StopBits.One;
    micPort.BaudRate = 2400;
    micPort.Parity = Parity.None;
    micPort.Handshake = Handshake.None;
    micPort.RtsEnable = true;
    micPort.ReadTimeout = 500;
    micPort.WriteTimeout = 500;

    try
    {
        micPort.Open();
    }
    catch (IOException)
    {
        MessageBox.Show("Port could not be opened", "Port Error",
             MessageBoxButtons.OK, MessageBoxIcon.Error);
        this.Close();
    }

}


public void ReadContinuous()
{
    while (continueRead)
    {
        Wait(300);
        RequestValue();
        Wait(300);
        String value = ReadValue();
        value += Units(value);
        returned = value;
        SetText(value);
    }
}


private void RequestValue()
{
    try
    {
        micPort.Write("1");
    }
    catch (System.TimeoutException)
    { }
}


private String ReadValue()
{
    String output = "";
    try
    {
        output = micPort.ReadExisting();
    }
    catch (System.TimeoutException)
    { }
    return RemoveUnneededChars(output);
}
```

```
private String RemoveUnneededChars(String output)
{
    String tmp = "";
    String sign = "";
    if (output.Length > 0) //Check that the returned string from the port is not blank
    {
        sign = output[3].ToString(); //store the sign of the value to above string
        for (int i = 3; i < output.Length; i++)
        // Search through the string until the decimal place or first nonzero number is            //found
        {
            if (output[i] == '.')
            {
                tmp += output.Substring(i);
                break;
            }
            else
            {
                try
                {
                    int parsed = (int)char.GetNumericValue(output[i]);
                    if (parsed > 0) //if the parsed string is  a number and greater than
                         //0, add it to the output
                    {
                        tmp += output.Substring(i);
                        break;
                    }
                }
                catch (FormatException) { }
            }
        }
    }
    return sign + tmp.Trim('\r','\n'); //add the sign and remove the trailing terminating
    //characters
}


        delegate void StringArgReturningVoidDelegate(string text);


        private void SetText(string text)
        {
            if (this.OutputValueLabel.InvokeRequired)
            {
                StringArgReturningVoidDelegate d = new StringArgReturningVoidDelegate(SetText);
                this.Invoke(d, new object[] { text });
            }
            else
            {
                this.OutputValueLabel.Text = text;
            }
        }
```

```
        private String Units(String output)
        {
            int indexOfDecimal = output.IndexOf(".");
            String tmp = "";
            if (indexOfDecimal != -1)
            {
                if (output.Substring(indexOfDecimal+1).Length > 4)
                {
                    tmp = " in";
                }
                else {
                    tmp = " mm";
                }
            }
            return tmp;
        }


        public void Wait(int milliseconds)
        {
            System.Windows.Forms.Timer timer1 = new System.Windows.Forms.Timer();
            if (milliseconds == 0 || milliseconds < 0) return;
            timer1.Interval = milliseconds;
            timer1.Enabled = true;
            timer1.Start();
            timer1.Tick += (s, e) =>
            {
                timer1.Enabled = false;
                timer1.Stop();
            };
            while (timer1.Enabled)
            {
                Application.DoEvents();
            }
        }


        private void SavePointButton_Click(object sender, EventArgs e)
        {
            string folderPath = "C:\\Users\\jdoe\\Documents";
            string filename = "TestFile99.txt";
            string fullPath = folderPath + "\\" + filename;

            if (File.Exists(fullPath))
            {
                using (var Tw = new StreamWriter(fullPath, true))
                {
                    Tw.WriteLine(returned + "\t" + System.DateTime.Now.ToString("h:mm:ss tt") + "\t" +
                        System.DateTime.Now.ToLongDateString());
                }
            }
            else
            {
                FileStream Fs = File.Create(fullPath);
                Fs.Close();

                using (var Tw = new StreamWriter(fullPath, true))
                {
                    Tw.WriteLine("Position \t Timestamp \t Date");
                    Tw.WriteLine(returned + "\t" + System.DateTime.Now.ToString("h:mm:ss tt") + "\t" +
                        System.DateTime.Now.ToLongDateString());
                }
            }
        }
    }
}
```