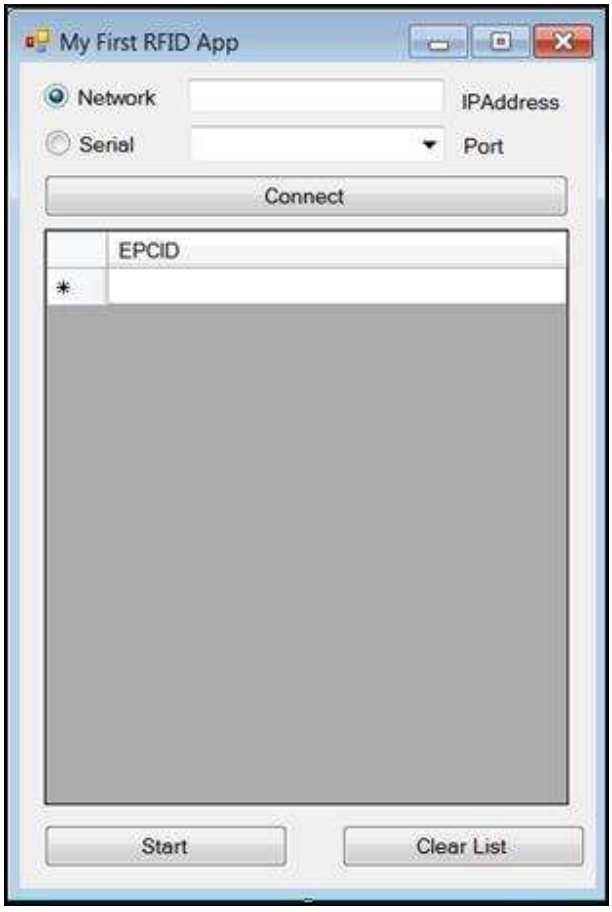
Building a Basic RFID Application in Visual Studio 2008

Introduction

An RFID reader is one of the only products that has no real function unless an application is written to control the reader and read or write data to an RFID tag.

This tutorial will show you how to write an RFID application that can read RFID tags using a reader that has either a Network or serial connection to the host computer.

The application is written in C# .NET using the Intermec RFID Resource Kit and is a Full Framework application. It could have just as easily been written in Visual Basic or as a compact framework application to be used on a mobile device. Very little modification to the source code of this application would have been needed.



Getting Started

Requirements

- Intermec RFID Resource kit
- Visual Studio 2008 or greater (Visual Studio 2008 only for mobile devices)

After installing the RFID Resource Kit, Open visual Studio and create a new Windows forms project targeting .NET Framework 3.5 and call it “Basic RFID”. Add a reference to the project for the following assemblies.

- Intermec.DataCollection.RFID.Basic
- Intermec.DataCollection.RFID.Advanced

On the form add the following controls, and arrange and configure their properties so the look like the controls shown in the screen shot on this page.

Control	Name
Button	btn_Connect
Button	btn_Read
Button	btn_Clear
Radio Button	btn_Network
Radio Button	btn_Serial
TextBox	txt_Address
ComboBox	cmb_Port
DataGrid	dgv_Tags
Label	Address
label	Port

Explaining the Code

Creating the Reader Object

The first thing we need to do is create the reader object. To do this add “using Intermec.DataCollection.RFID;” to your project, an then Create the reader object “BRIReader reader;”

```
using Intermec.DataCollection.RFID;

namespace Basic_RFID
{
    public partial class Form1 : Form
    {
        BRIReader reader;

        public Form1()
        {
            InitializeComponent();
            getAvailableComPorts();
        }

    }

}
```

Getting the Serial Ports

Since our application is going to be able to use either a network or serial connected reader the application needs to know how to communicate with the reader. The network address is not much of an issue as all you need to do is enter the IP address of the reader, but a computer may have more than one serial port available so we need to get a list of the available serial ports and populate the port ComboBox. In our application this method is called from the Forms constructor.

```
/// <summary>
/// Populates the Ports comboBox with a list of available ports
/// </summary>
private void getAvailableComPorts()
{
    cmb_Port.Items.Clear();
    foreach (string portname in System.IO.Ports.SerialPort.GetPortNames())
    {
        cmb_Port.Sorted = true;
        cmb_Port.Items.Add(portname);
    }
}
```

Determining the Connection Type

Since the application is capable of using either type of connection there has to be a means to control which connection type is going to be use. The selected radio button determines which of the connection types is used and the code behind the Network button enables and disables the IPAddress TextBox and Port ComboBox depending on its checked status.

```
/// <summary>
/// We use the one radio button checked event to determine if the application /// is
to connect to the reader using a network or serial connection
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btn_Network_CheckedChanged(object sender, EventArgs e)
{
    if (btn_Network.Checked)
    {
        txt_Address.Enabled = true;
        cmb_Port.Enabled = false;
    }
    else
    {
        txt_Address.Enabled = false;
        cmb_Port.Enabled = true;
    }
}
```

Connecting to the Reader

Up until now we haven't done anything with the RRID reader, which is now going to change as we will need to add code to the Connection button. This button will be used to both make the connection to the reader and disconnect from the reader when done. Again the connection type is controlled by the radio button selected. The code also does some error checking to determine if the user has entered the needed information to make the connection.

The connection is made by constructing a new BRIReader object. Our application is using the "Advanced" assembly to create the reader object. Although the constructor has a number of overloads we are using one of the most basic constructors, passing it the *Owner* and the *readerURI* .

The Owner is a reference to your applications Windows.Forms Object. This can be set to null if you are building a console application (without a user interface). The readerURI specifies the transport type and reader address.

For a TCP connection, use the URI format "tcp://*IPAddress*:*PortNumber*" where *IPAddress* specifies the IP address of the RFID reader in the IPv4 dotted decimal notation and *PortNumber* specifies the port number to open. If PortNumber is not specified, the default BRI port number (2189) will be used. The following are TCP URI examples:

- tcp://111.22.33.44
- tcp://111.22.33.44:2189
- tcp://127.0.0.1 (the localhost address used to access the RFID reader locally)

For a serial connection, use the URI format "serial://*ComPortName*" where *ComPortName* specifies the COM port. The following are serial URI examples.

- serial://COM1
- serial://COM2

Since we are not specifying the read buffer and event queue size in the constructor, the default values of 16KB for the read buffer and 50 events for the event queue will be used. If the application encounters a problem when connecting to the reader a BasicReaderException is thrown and our code displays a message box to the user informing the user that the connection can't be made.

```
/// <summary>
/// Depending on the button caption this button either connects the
/// application to the reader or disconnects it.
/// When making the connection the validity of the IPAdress or Serial port is
/// checked prior to making the connection
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btn_Connect_Click(object sender, EventArgs e)
{
    try
    {
        if (btn_Connect.Text == "Connect")
        {
            string connectionType;
            string readerAddress = txt_Address.Text;
            string msg = string.Empty;
            // Sets the connection type variable to either a network or
            // serial based connection
            if (btn_Network.Checked) connectionType = "TCP://"; else connectionType =
            "Serial://";
            // The switch statement sets the reaader address to the entered IP address
            // or serial port based on the connection type selected. If either the
            // address or port controls contain empty strings a message is created to
            // display to the user
            switch (connectionType)
            {
                case "TCP://":
                    if (txt_Address.Text != string.Empty)
                        readerAddress = txt_Address.Text;
                    else
                        msg = "Could not determine the IPAddress of the reader. Did
                        you enter it in the IPAddress field?";
                    break;
                case "Serial://":
                    if (cmb_Port.Text != string.Empty)
                        readerAddress = cmb_Port.Text;
                    else
                        msg = "Could not determine the serial port. Did you select a
                        port from the drop down menu?";
                    break;
            }
            // if the user message exists it is displayed to the user, otherwise the
            // connection is made to the reader
            if (msg != string.Empty)
            {
                MessageBox.Show(msg);
            }
            else
```

```
            {
                reader = new BRIReader(this, connectionType + readerAddress);
                btn_Connect.Text = "Disconnect";
                btn_read.Enabled = true;
            }
        }
        else
        {
            reader.Dispose();
            btn_Connect.Text = "Connect";
            btn_read.Enabled = false;
        }
    }
    catch (BasicReaderException)
    {
        MessageBox.Show("Can't connect to the reader");
    }
}
```

Reading Tags

The application uses to methods when reading tags, the Read button handler initiates a single shot read, and then sends the list of tags read to the UpdateDataGrid method where each tag in the list is added as a grid item.

Although the Read method has a number of overloads, we are using it in its simplest form which takes no arguments. When executed the method reads all of the tags in range and populates the Tags property with an array of Tag objects. Because we used no arguments the default schema is used and only the tag key is returned for each tag detected. Depending on the type tag being read either the TagID or the EPCID is returned. In this tutorial we are assuming that EPCC1G2 tags are being read so the UpdateDataGrid method populates the DataGrid with the EPCID of each tag read.

```
/// <summary>
/// When the read button is pressed the read command is sent to the reader.
/// Any tag read are placed in the reader objects
/// tag list. Each tag in the tag list is then read into the tag dataGridView
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btn_read_Click(object sender, EventArgs e)
{
        reader.Read();
        UpdateDataGrid(reader.Tags);
}
```

Now that we have read the tags using the code above we need to populate the grid using the code below.

```
/// <summary>
/// Each tag in the tag list is then read into the tag dataGridView
/// </summary>
/// <param name="tags"></param>
private void UpdateDataGrid(Tag[] tags)
{
    if (reader.TagCount > 0)
    {
        for (int i = 0; i < tags.Length; i++)
        {
            DataGridViewRow row = (DataGridViewRow)dgv_Tags.Rows[0].Clone();
            row.Cells[0].Value = tags[i];
            dgv_Tags.Rows.Add(row);
        }
    }
    else
    {
        DataGridViewRow row = (DataGridViewRow)dgv_Tags.Rows[0].Clone();
        row.Cells[0].Value = "No Tags";
        dgv_Tags.Rows.Add(row);
    }
}

/// <summary>
/// Method clears existing tag data from the dataGridView
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btn_Clear_Click(object sender, EventArgs e)
{
    dgv_Tags.Rows.Clear();
}
```

Notice that there is one more method in the code above we haven't talked about, the handler for the Clear button. All this does is clears the EPCIDs from the DataGrid.

Conclusion

Although this is a relatively simple application it gives you the basic information you need to start building an RFID application of your own. More information on the features of the resource kit can be found in the Documents folder of the kit.