1601 Pacific Coast Hwy • Suite 290 • Hermosa Beach • CA • 90254
www.rfdigital.com • Tel: 949.610.0008 • sales@rfdigital.com

Abstract

Writing a Windows® application which interfaces with the RFduino has never been easier. An API introduced with Windows 8.1 opens the door for creating exciting for Windows Store applications that interface with the RFduino. While the API is somewhat limited, when coupled with an RFduino rich applications supporting Bluetooth low energy can be created. This tutorial will introduce the API, cover its limitations and provide example code, which communicates with a Bluetooth Smart compass, built utilizing an RFduino.

Windows 8.1 API

The Windows 8.1 API enables reading and writing Characteristic data from an RFduino device. However, before the application is able to perform these operations, the RFduino must first be discovered and paired with from Windows Settings. If it is the first time the application interacts with a specific RFduino, the user will be asked to allow the application to use the RFduino before any data will be forwarded to the application. Once the user has performed these two steps, communication with the RFduino will be available to the application.

The API provides methods for querying RFduino's known to a Windows 8.1 device such as a personal computer or Surface. A query can be filter to only those RFduino's which implement a desired Service or not filtered resulting in the query returning all the RFduino's which are known to the Windows device. When querying by Service, the Service can be either a "well known" Service published by the Bluetooth SIG or a proprietary Service published by the RFduino. A Bluetooth SIG Service will be designated by a 16 bit UUID. The Windows API contains an enumeration consisting of human-readable names to make the source code more readable. A proprietary Service will be a 128 bit UUID defined by the Bluetooth Smart device manufacturer and exposed by the RFduino. There is an "in between" case in which the Bluetooth SIG is selling a limited number of 16 bit UUIDs to be used by Bluetooth SIG members for proprietary Services. In this paper only a 128 bit proprietary UUID is used.

Once a handle to a Service is obtained there are additional API calls to find Characteristics exposed by the Service. Through the Characteristic handles data can be read from and/or written to the device. The format of the data, its access requirements and security requirements are defined by the Bluetooth Smart device manufacturer and the Windows 8.1 API adapts to these requirements to present properly formed Characteristics to the application. However, the application must know the format of the data read from and written to the device. While the API provides methods to simplify formatting the data, the required format can't be determined from the Characteristic itself and must be provided in the documentation from the Bluetooth Smart device manufacturer. As such, an application will typically be tightly couple to one, or a set of, known Bluetooth Smart devices and typically won't work with other Bluetooth Smart devices.

The Windows 8.1 API only allows applications to communicate with paired devices. As such, the API doesn't enable Windows Store application which support the widely popular beacon scenario[1]. The API also doesn't enable background data exchange or receiving notifications while suspended. Thus, many other scenarios envisioned for Bluetooth Smart devices aren't currently possible with the 8.1 API. There is a richer feature set in the Windows Phone 8.1 API. While I haven't seen an official announcement from Microsoft, I expect the Windows Phone enhancements as well as enabling the beacon and background processing scenarios to be enabled in the next release of the Bluetooth low energy API.

The Bluetooth Low Energy API[2] is contained in the `Windows.Devices.Bluetooth`, `Windows.Devices.Bluetooth.Rfcomm` and `Windows.Devices.Bluetooth.GenericAttributeProfile` namespaces. For interacting with Bluetooth Smart devices, we'll focus on the `GenericAttributeProfile` namespace which contains

[1] A nice overview of the AIP limitations can be found here http://channel9.msdn.com/coding4fun/blog/Powering-up-with-BLE-in-Windows-81

[2] See https://msdn.microsoft.com/en-us/library/dn264587.aspx for more information on the API

most of the classes and methods required for this interaction. The Rfcomm namespace is for interacting with classic Bluetooth devices and isn't discussed further in this paper.

Sample Device and Application

As previously described, an application is typically designed to work with one Bluetooth Smart device or a collection of related devices. In this paper, a compass will be implemented on an RFduino which will expose direction to the Windows Store application.

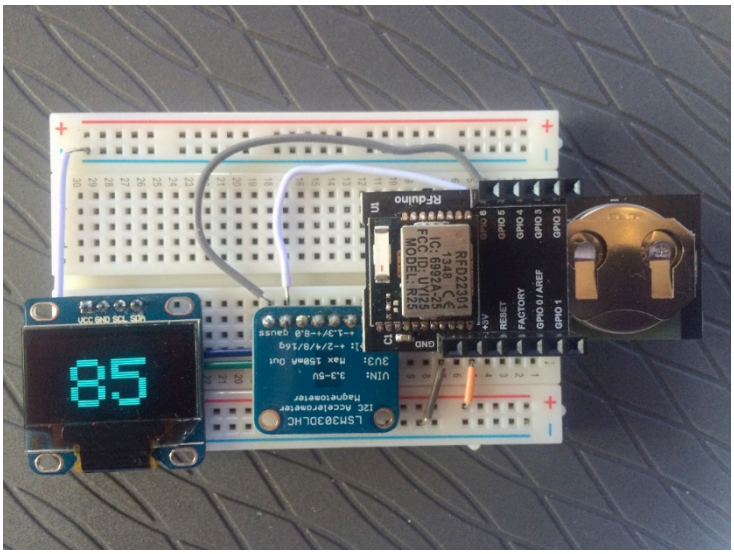
RFduino Compass

A very simple implementation of a compass built with an RFduino will be utilized as the Bluetooth Smart device which sends heading data to the Windows Store application. The compass is built with the following components:

- RFduino with USB and battery shields
- LSM303DLHC I2C 3-axis accelerometer and 3-axis magnetometer
- Optional: I2C OLED display

The LSM303 and OLED display are both capable of running with a 3.3V power supply and data lines, so they are directly compatible with the RFduino without the need for level shifters. This makes it straight forward to connect the devices on the I2C bus using the default SDA and SCL lines of the RFduino, which are 6 and 5 respectfully. The RFduino Compass on a breadboard is show below.

With the hardware wired together the RFduino sketch is straightforward. Periodically the heading is read from the LSM303 and output to the OLED display. If there is an active Bluetooth connection, the heading is also written to the connected device. The LSM303 is much more accurate if it is calibrated. There are elaborate calibration routines which can be used[3], a simple method is implemented in the RFduino compass and can be initiated by sending a calibration command via Bluetooth to the compass.



[3] Here is one such example https://learn.adafruit.com/lsm303-accelerometer-slash-compass-breakout/calibration

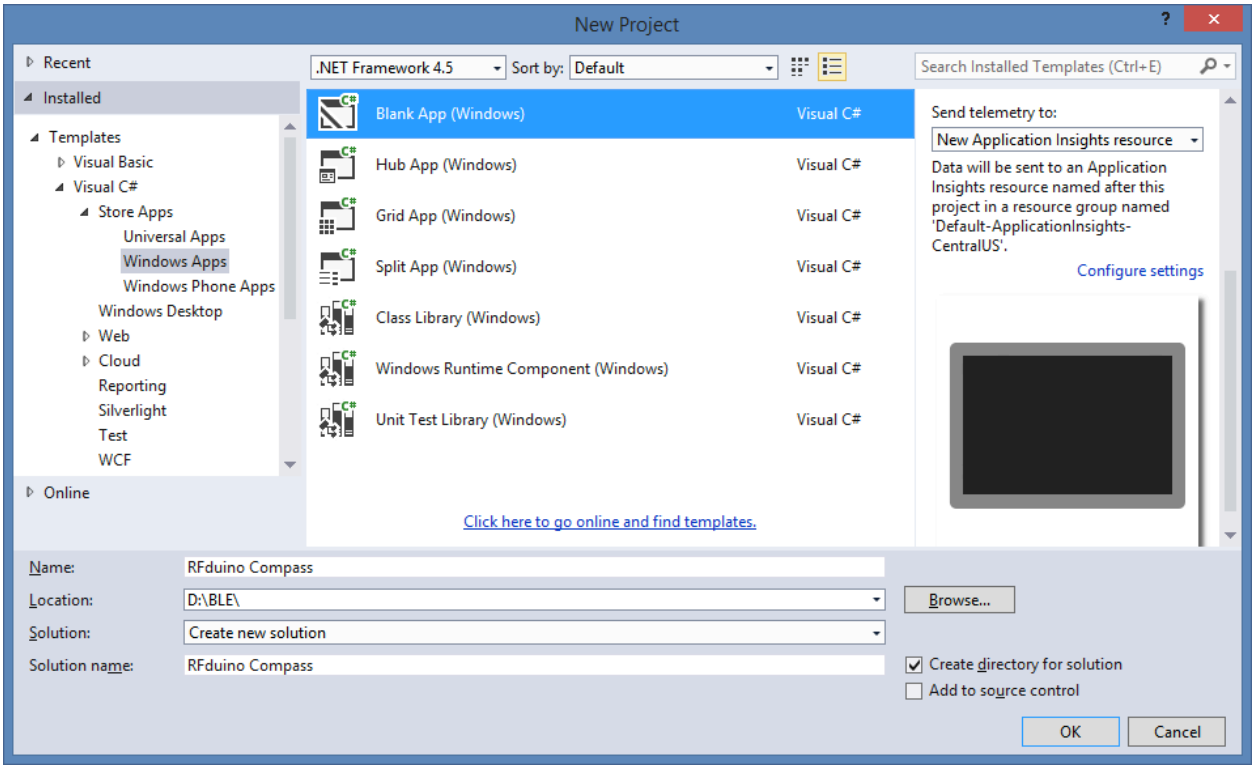
While there are many interesting code elements in the compass, the focus of this paper is the Windows 8.1 API and how a Windows store application can utilize the compass. Thus, further details of the compass aren't provided here. More details and the complete source code for the compass is available on github[4].

Windows Store Compass Application

This section is going to focus on adding Bluetooth low energy connectivity to a Windows Store application. A very simple application is used for the example, but the same principles can be used in a more complex application. Any Windows Store application template[5] can be used. For this paper, the basic template is utilized.

Not to state the obvious, but your development environment must be Visual Studio 2013 (the free Community version was used for this development) on a Windows 8.1 PC with Bluetooth Low Energy installed. The Bluetooth 4.0 radio can be built in or added via a USB dongle. Dongles from CSR and Broadcom were both tested with this example code. To get the dongles with a Broadcom module working properly, the latest driver needed to be downloaded from the Broadcom web site as the one shipped with the dongle on a CD didn't work properly. You should ensure that you Bluetooth low energy working properly on your development machine before proceeding.

To begin this project, in Visual Studio create a new Blank Windows Store application project. Name the project whatever you'd like, "RFduino Compass" was given as the name here.

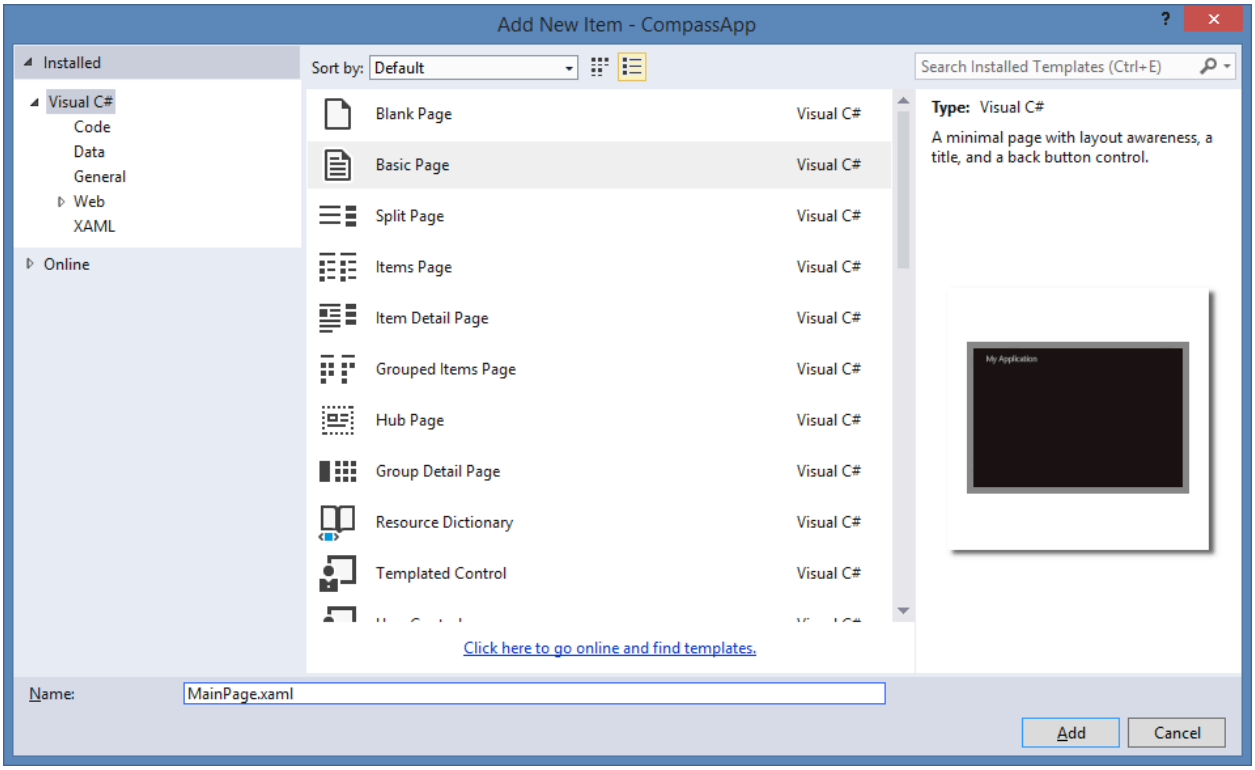


[4] The Compass repository and libraries: https://github.com/RFduino/RFduinoApps/tree/master/Windows%20App

[5] See https://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh768232.aspx for a list of templates

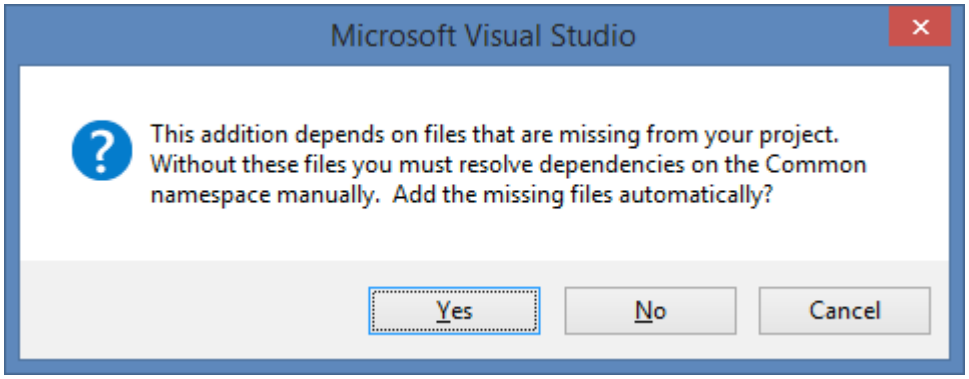
This template creates a minimal store application with references, a few resources and an App.xaml, MainPage.xaml with code behind files as well and a manifest file for the application (Package.appxmanifest). In this paper, we'll focus primarily on the manifest and MainPage files. The application created from the template runs as created but doesn't do much. To add functionality to the application, we are going to update the manifest file to tell Windows what capabilities the application contains, add C# code to interact with the RFduino compass and finally add a simple UI via Xaml. If you are unfamiliar with building Windows Store applications, an introductory tutorial which creates the standard "Hello World" application is available from Microsoft[6].

We'll follow the advice from the Hello World example above and replace the default MainPage created by the Blank template with a Basic Page. Other Page types could be used in a more advanced application. Following those instructions, delete MainPage.xaml form the project and add a new MainPage.xaml based on the Basic Page template.



After adding the page, you will be notified of missing dependencies for the project. Allow Visual Studio to add these to the project by selecting "Yes" in the dialog.

[6] The Hello World application: https://msdn.microsoft.com/en-us/library/windows/apps/hh986965.aspx



Now the project is ready to adding the Bluetooth capability.

Application Manifest

The application needs to let the Windows operating system know of its intended use of Bluetooth devices. How and what these intentions are is defined in the application manifest. The manifest is contained in an XML file named "Package.appxmanifest." Many items available for customization in the manifest can be update via UI in Visual Studio. However, this isn't the case for Bluetooth support. Thus, open the manifest in the editor by right clicking on the manifest file name (Package.appxmanifest) in the Solution Explorer and select "View Code" in the menu.

The default manifest created with the template contains a single capability; internet client. This default capability section won't allow the application to communicate with an RFduino. The capabilities must be expanded to state which RFduino devices are supported as well as which Services are of interest to the application.

```
<Capabilities>
    <Capability Name="internetClient" />
  </Capabilities>
```

The Bluetooth capability is added to the Capabilities section as shown below.

```
<Capabilities>
    <Capability Name="internetClient" />
    <m2:DeviceCapability Name="bluetooth.genericAttributeProfile">
      <m2:Device Id="any">
        <m2:Function Type="serviceId:b329392a-fbcd-49aa-a823-3e87680ac33b" />
      </m2:Device>
    </m2:DeviceCapability>
  </Capabilities>
```

This defines that the Bluetooth.genericAttributeProfile is going to be used by the application. When using this profile, the application is interested in "any" device which contains the serviceId defined by the Function tag. There are named Function[7] attributes for the standard Services published by the Bluetooth SIG. The RFduino compass utilizes a proprietary Service. Thus, the 128 bit UUID of that service is specified here. This UUID ***must exactly*** match the UUID exposed by the RFduino.

[7] The list can be found here:

https://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.bluetooth.genericattributeprofile.gattserviceuuids.aspx

If more than one device type or Service are desired, these sections can be repeated for each Service and/or Device the application utilizes.

Bluetooth Code

The Bluetooth code to interface with the RFduino Compass can be thought of in two basic steps: setup and active use. During setup, the compass is found and the application registers for updates. During active use, the compass continually sends updates to the application which displays them.

The setup code has been placed in a single asynchronous method, findBLEServiceAsync. The function is asynchronous to allow the UI to remain responsive as the RFduino may need to be queried for Service and Characteristic information.

```
using Windows.Devices.Enumeration;
using System.Threading.Tasks;
using Windows.Devices.Bluetooth.GenericAttributeProfile;

/// <summary>
/// Find a device which exposed the given UUID.
///
/// The device must already be attached and paired with the device
/// this application is running on.
/// If more than one currently connected device exposes the given
/// Service, the first one returned from the System is returned to
/// the caller.
/// </summary>
/// <param name="serviceUUID">The UUID of a Service to find in a connected device</param>
/// <returns>A GattDeviceService for the given Service or null if none found</returns>
private async Task<GattDeviceService> findBLEServiceAsync(string serviceUUID)
{
    //
    // Find known devices which implement the given service UUID
    // The returned value is a device enumeration, but it really
    // seems to be a list of Services.
    //
    var devices = await Windows.Devices.Enumeration.DeviceInformation.FindAllAsync(
                GattDeviceService.GetDeviceSelectorFromUuid(new Guid(serviceUUID)));

    if (devices.Count == 0)
        return null;

    var service = await GattDeviceService.FromIdAsync(devices[0].Id);
    return service;
}
```

The method simply calls the Windows DeviceInformation enumeration to find all Bluetooth Smart devices connected which implement the given Service UUID. The UUID may be a Bluetooth SIG specified 16 bit ID or a proprietary 128 bit UUID. With the Service reference in hand, Characteristics contained in the Service can be queried. A callback method can be registered on a Characteristic which notifies when it changes. The “read” Characteristic exposed by the RFduino is one such example. Code to register such an event handler is completed in the registerCharacteristicChangedCallback method.

```
/// <summary>
/// Register a ValueChanged callback on a device Characteristic.
/// The Characteristic must be readable and send Notifications.
/// </summary>
```

```
/// <param name="service">The Service containing the Characteristic</param>
/// <param name="readCharacteristicUUID">The UUID of the Characteristic</param>
/// <param name="valueChangedHandler">The event handler callback</param>
/// <returns>A GattCharacteristic for the which the event handler was
/// registered or null if the Characteristic wasn't found</returns>
private async Task<GattCharacteristic> registerCharacteristicCallback(
    GattDeviceService service,
    string readCharacteristicUUID,
    TypedEventHandler<GattCharacteristic, GattValueChangedEventArgs> valueChangedHandler)
{
    Debug.Assert(service != null, "Null service passed as argument.");
    Debug.Assert(readCharacteristicUUID != null,
                                        "Null Characteristic UUID pass as argument.");

    //Obtain the characteristic we want to interact with
    var characteristics = service.GetCharacteristics(new Guid(readCharacteristicUUID));
    if (characteristics.Count == 0) return null;

    var characteristic = characteristics[0];

    //Subscribe to value changed event
    characteristic.ValueChanged += valueChangedHandler;

    //Set configuration to notify
    await characteristic.WriteClientCharacteristicConfigurationDescriptorAsync(
        GattClientCharacteristicConfigurationDescriptorValue.Notify);
    return characteristic;
}
```

The above function requires an event handler which is called whenever a new value for the Characteristic changes. In this case that will be whenever the heading of the RFduino Compass changes. The handler is a standard event handler and shown next.

```
using System.Diagnostics;
using Windows.ApplicationModel.Core;
using Windows.UI.Core;

/// <summary>
/// Callback when the read characteristic changes.
///
/// This method expects a single 32 bit integer to be received.
/// </summary>
/// <param name="sender">The characteristic upon which that change occurred</param>
/// <param name="args">The event arguments</param>
private async void headingValueChanged(GattCharacteristic sender,
    GattValueChangedEventArgs args)
{
    try
    {
        Debug.Assert(args.CharacteristicValue.Length == 4,
            string.Format("Characteristic length of {0} isn't the expected length of 4",
            args.CharacteristicValue.Length));

        var bytes = args.CharacteristicValue.ToArray();
```

```
        await CoreApplication.MainView.CoreWindow.Dispatcher.RunAsync(
                    CoreDispatcherPriority.Normal,
        () =>
        {
            //
            // Don't read or write the heading property while not on
            // the main thread
            //
            this.Heading = bytes[0] +
                        (bytes[1] << 8) +
                        (bytes[2] << 16) +
                        (bytes[3] << 24);
        });
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex.Message);
    }
}
```

And the Heading property that reflects the last value received form the RFduino Compass is shown next.

```
/// <summary>
/// The last heading reading received from the RFduino Compass
/// </summary>
public static readonly DependencyProperty HeadingProperty = DependencyProperty.Register(
                                            "Heading",
                                             typeof(double),
                                             typeof(MainPage),
                                             new PropertyMetadata(default(double)));
public double Heading
{
    get { return (double)GetValue(HeadingProperty); }
    set { SetValue(HeadingProperty, value); }
}
```

With these three methods in hand, we can use some glue code to enable interaction with the RFduino Compass in the Windows Store application. The glue code is in the configureRFCompass() method. This method also saves the read and write Characteristics discovered in attributes defined in the class.

```
/// Class members
GattCharacteristic readCharacteristic;
GattCharacteristic writeCharacteristic;

using Windows.UI.Popups;

/// <summary>
/// Check if there is a device exposing the Compass Service.
/// If so, register a callback method which is notified
/// whenever the compass sends a new heading value.
/// If a Compass Service isn't found, notify the user.
/// </summary>
private async void configureRFCompass()
{
    const string compassServiceUUID = "b329392a-fbcd-49aa-a823-3e87680ac33b";
```

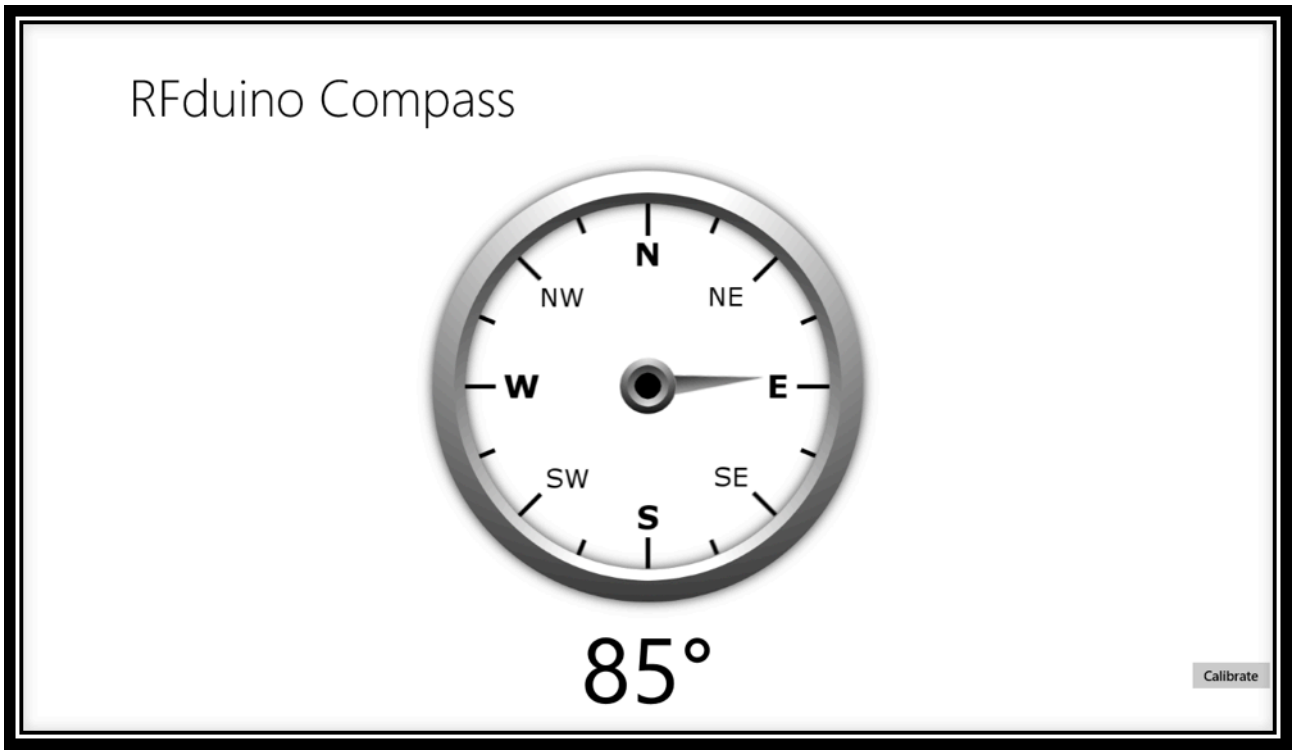
```
    const string readCharacteristicUUID = "b329392b-fbcd-49aa-a823-3e87680ac33b";
    const string writeCharacteristicUUID = "b329392c-fbcd-49aa-a823-3e87680ac33b";

    var rfduinoCompass = await findBLEServiceAsync(compassServiceUUID);

    if(rfduinoCompass != null)
    {
        //
        // Found! Register for notifications of heading changes
        //
        registerCharacteristicChangedCallback(rfduinoCompass,
                    readCharacteristicUUID,
                    headingValueChanged);

        //
        // Save the write characteristic to be used for writing to the compass
        //
        var characteristics = rfduinoCompassService.GetCharacteristics(
                                        new Guid(writeCharacteristicUUID));
        if (characteristics.Count != 0)
        {
            this.writeCharacteristic = characteristics[0];
        }
    }
    else
    {
        //
        // Couldn't find a compass service.
        // Notify User. There may be other error handling you want to do
        //
        var messageDialog = new MessageDialog(
                                    "A compass wasn't found. " +
                                    "Have you paired the compass in Settings?");
        await messageDialog.ShowAsync();
    }
}
```

The glue method can be called from any initialization method. In this example, it will be called from the `navigationHelper_LoadState()` method in `MainPage.xaml.cs` which is part of the Basic Page framework. This method is called whenever the page is navigated to. Other logical places would be in the MainPage() constructor or even in `OnLaunched()` in App.xaml.cs.

With this code in place, the application should be able to connect with the RFdunio Compass, register for notifications and receive heading updates. You could add a Debug.Writeline() statement in the `headingValueChanged()` method to see the updates received in the debugger log. In the next section, we'll add a minimal user interface to display the heading information.

User Interface XAML

The user interface for the RFduino Compass is pretty simple. It consists of two images, two TextBlocks and a Button. The first image is a static image of a compass frame while the second image of the compass needle will be rotated around the center of the frame reflecting the current heading received from the compass. The first TextBlock statically displays the application name while the second TextBlock dynamically displays a numerical representation of the heading. The Button initiates code which sends a calibration request to the RFduino Compass. The user interface is show below.



The MainPage.xaml created by the Basic Page template is, well, basic. It simply contains markup for a title and back arrow arranged in a grid. Replace the entire markup with the following to create the RFduino Compass user interface.

```
<Page
    x:Name="pageRoot"
    x:Class="RFduino_Compass.MainPage"
    DataContext="{Binding RelativeSource={RelativeSource Self}}"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:RFduino_Compass"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
>
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Grid.ChildrenTransitions>
            <TransitionCollection>
                <EntranceThemeTransition/>
            </TransitionCollection>
        </Grid.ChildrenTransitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="140"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="120"/>
        </Grid.RowDefinitions>

        <!-- Back button and page title -->
        <Grid>
            <Grid.ColumnDefinitions>
```

```
            <ColumnDefinition Width="120"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Button x:Name="backButton" Margin="39,59,39,0"
                Command=
                "{Binding NavigationHelper.GoBackCommand, ElementName=pageRoot}"
                Style="{StaticResource NavigationBackButtonNormalStyle}"
                VerticalAlignment="Top"
                AutomationProperties.Name="Back"
                AutomationProperties.AutomationId="BackButton"
                AutomationProperties.ItemType="Navigation Button"/>
        <TextBlock x:Name="pageTitle"
                Text="{StaticResource AppName}"
                Style="{StaticResource HeaderTextBlockStyle}"
                Grid.Column="1"
                IsHitTestVisible="false"
                TextWrapping="NoWrap"
                VerticalAlignment="Bottom"
                Margin="0,0,30,40"/>
    </Grid>
    <Image x:Name="CompassNeedle" Grid.Row="1"
        HorizontalAlignment="Center" Margin="0,0,0,0"
        VerticalAlignment="Top" RenderTransformOrigin="0.5, 0.5"
        Source="Assets/Images/CompassNeedle.png">
        <Image.RenderTransform>
            <RotateTransform x:Name="NeedleTransform" Angle="0" />
        </Image.RenderTransform>
        <Image.Resources>
            <Storyboard x:Key="spin">
                <DoubleAnimation x:Name="CompassNeedleAnimation"
                        Storyboard.TargetName="NeedleTransform"
                        Storyboard.TargetProperty="Angle"
                        By="360"
                        Duration="0:0:0.5"
                        AutoReverse="False"
                        />
            </Storyboard>
        </Image.Resources>
    </Image>
    <Image x:Name="CompassRing" Grid.Row="1"
        HorizontalAlignment="Center" Margin="0,0,0,0"
        VerticalAlignment="Top" Source="Assets/Images/CompassRing.png"/>
    <TextBlock x:Name="CurrentHeadingTextBlock" Grid.Row="2"
        Text="360&#186;" FontSize="100"
        Margin="0,0,0,20" HorizontalAlignment="Center"/>
    <Button x:Name="CalibrateButton" Grid.Row="2"
            HorizontalAlignment="Right" Content="Calibrate"
            Click="Calibrate_ButtonClick" />
</Grid>
</Page>
```

If you used a different name for the project when it was created, you'll need to use that name in the class and using statements above. You'll also have to ensure that the images used in the UI are located in the Assets/Images directory or change the paths above. (The images can be downloaded with the complete source code or you can create better ones!)

The only thing that may be a bit unique about the UI XAML is the use of a Storyboard to animate the compass needle rotations. The Storyboard is defined in the XAML and in the code which updates the UI when a new heading is received utilizes the Storyboard to animate the compass needle moving from the previous heading to the newly received heading. The code to update the UI is placed in the block of code which runs on the main thread in the `headingValueChanged()` method.

```
using Windows.UI.Xaml.Media.Animation;

await CoreApplication.MainView.CoreWindow.Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
() =>
{
    //
    // Don't read or write the heading property while not on the main thread
    //
    //
    var currentHeading = Heading;

    var newHeading = bytes[0] +
                     (bytes[1] << 8) +
                     (bytes[2] << 16) +
                     (bytes[3] << 24);
    Debug.WriteLine("Heading {0}", this.Heading);

    double delta = angleDifference(currentHeading, newHeading);

    double nextHeading = currentHeading + delta;

    //Needed to prevent an error
    this.CompassNeedleAnimation.From = currentHeading;
    this.CompassNeedleAnimation.To = nextHeading;
    Storyboard sb = (Storyboard)CompassNeedle.Resources["spin"];
    sb.Begin();

    // Update the text
    this.CurrentHeadingTextBlock.Text = string.Format("{0}°", newHeading);

    // Save the new heading
    this.Heading = newHeading;
});
```

And the utility method which calculates the smaller of the two angles between the current and new headings. I.e. if the current heading was 45 degrees and the new heading is 30 degrees, the desired movement of the compass is -15 degrees rather than 345 degrees to more accurately reflect how a needle on a physical compass behaves.

```
/// <summary>
/// Return the shortest signed difference between the two given andles.
///
/// The sign of the difference defines the direction traveled to get
/// from the first angle to the second where clockwise is positive.
///    If x = 10 and y = 90, the returned value is 80
///    If x = 90 and y = 10, the returned value is -80
///
/// When it is interesting is when the "origin" (0 0r 360) is crossed
///    if x = 10 and y = 350, the returned value is -20
///    if x = 350 and y = 10, the returned value is 20
```

```
///
/// Note: The method doesn't handle multiple trips around the unit circle.
///       i.e. inpot angles are expected between 0-360.
/// </summary>
/// <param name="x">The first angle in degrees</param>
/// <param name="y">The second angle in degrees</param>
/// <returns>The difference between the two given angles in degrees</returns>
private double angleDifference(double x, double y)
{
    double C360 = 360.000000000;

    double arg;
    arg = Math.IEEERemainder(y - x, C360);
    if (arg < 0) arg = arg + C360;
    if (arg > 180) arg = arg - C360;
    return (arg);
}
```

The XAML also specifies an even handler for when the “Calibration” button is clicked. That method simply writes the value 0xAA to the write Characteristic of the RFduino compass. The value 0xAA is arbitrary, as any value could have been used to trigger a calibration and 0xAA was chosen. More complex data structures can be written by modifying the value passed to the `WriteValueAsync method.`

```
/// <summary>
/// The calibrate button was clicked. Send a calibration request
/// to the compass
/// </summary>
/// <param name="sender">The button which was clicked</param>
/// <param name="e">The event arguments</param>
private async void Calibrate_ButtonClick(object sender, RoutedEventArgs e)
{
    if (this.writeCharacteristic == null) return;

    try
    {
        byte value = 0xAA;
        await this.writeCharacteristic.WriteValueAsync(
                                        (new byte[] { value }).AsBuffer());
    } catch(Exception ex)
    {
        Debug.WriteLine(ex.Message);
    }
}
```

For this application, the light theme looks good. Light is the default template, and can be requested in the App.xaml. The application name used in the MainPage XAML is also specified here. With these two additions, the App.xaml looks like this.

```
<Application
    x:Class="RFduino_Compass.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:RFduino_Compass"
    RequestedTheme="Light">
    <Application.Resources>
        <x:String x:Key="AppName">RFduino Compass</x:String>
```

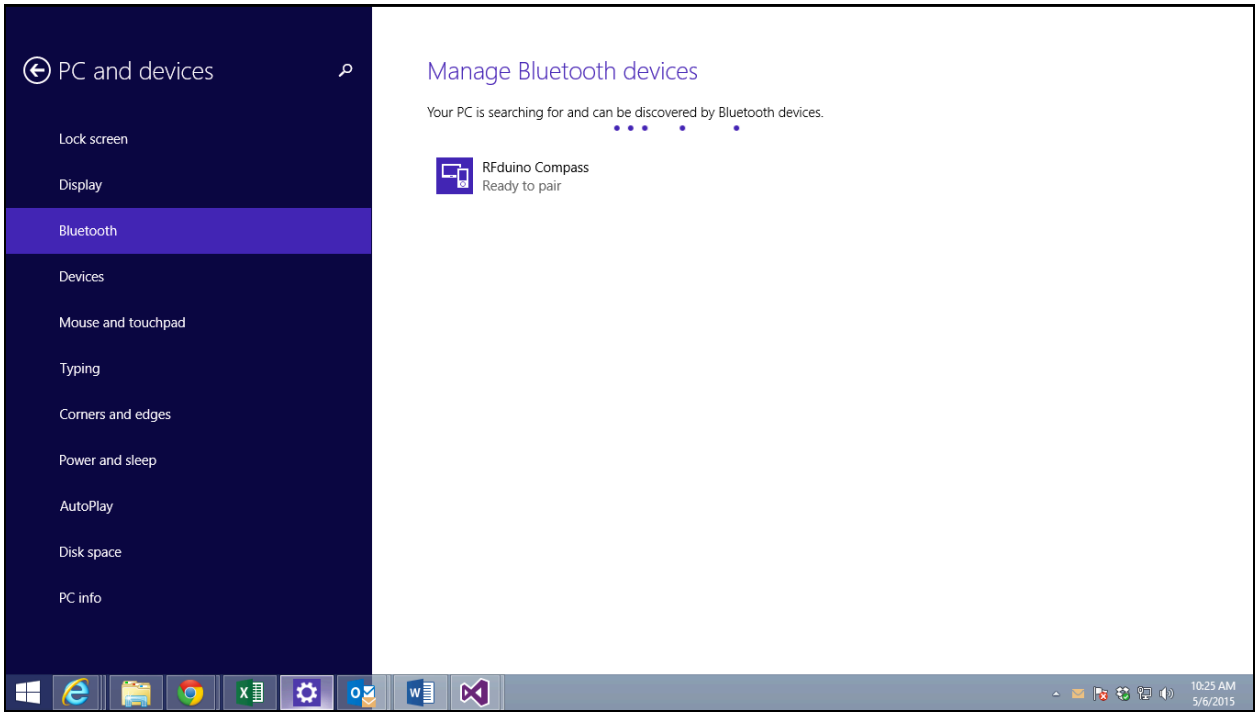
```
    </Application.Resources>
</Application>
```

Just like in the MainPage.xaml, if you used a different name for the project when it was created, you'll need to use that name in the class statement above.

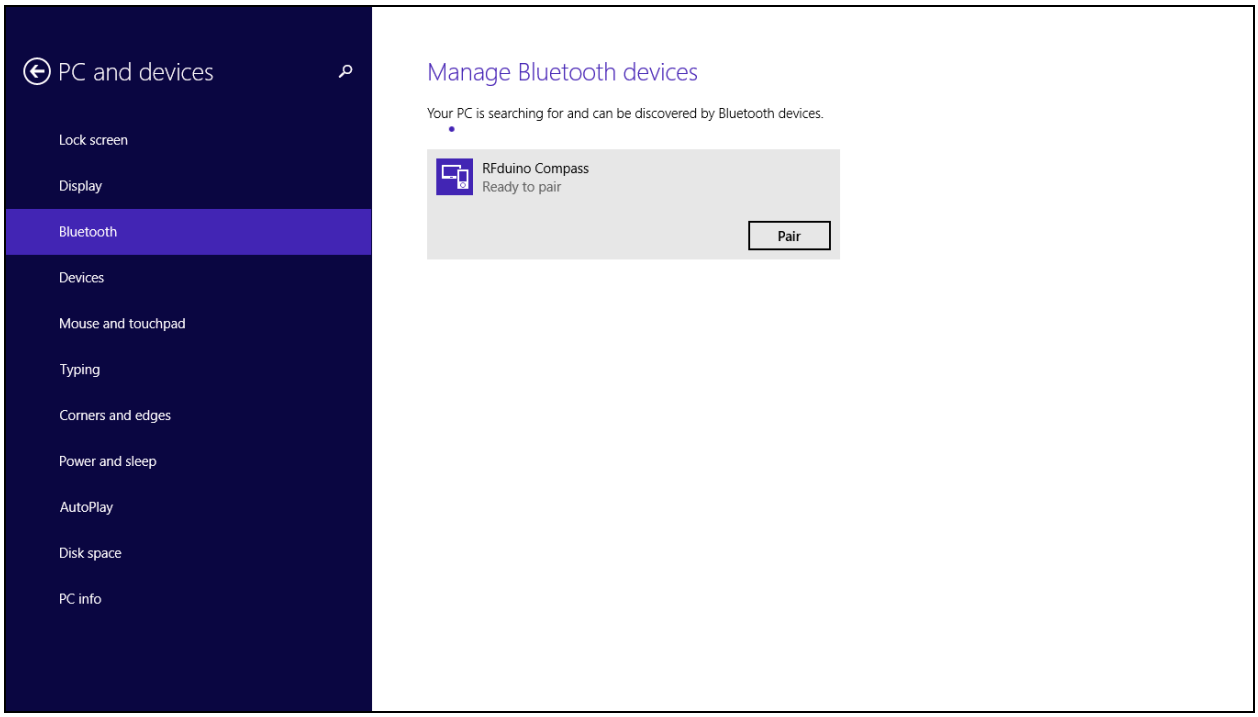
With the above XAML and update code in place the application should work as a fully functional visualization of the heading measured on the RFduino Compass.

User Experience

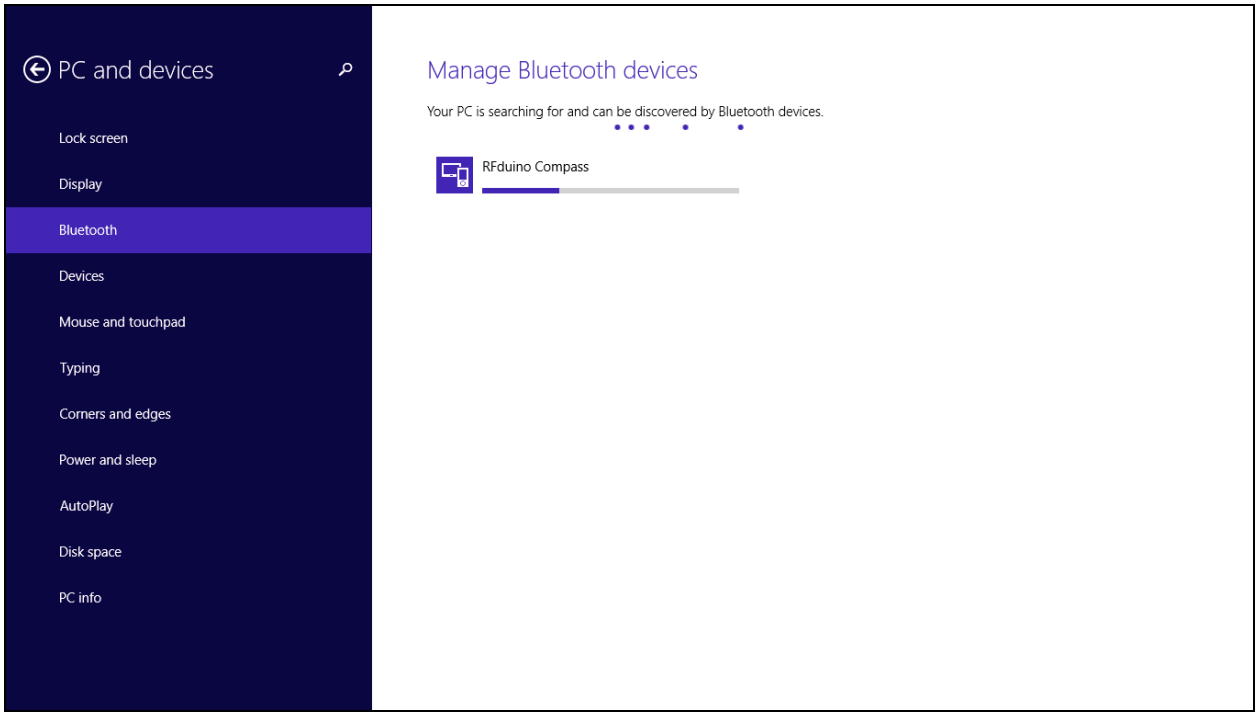
To use the RFduino Compass with the Windows Store application, the compass must first be paired and connected with the device running the application. The application can run on a PC, notebook, Surface or any device running Windows 8.1 with Bluetooth low energy support; also known as a Bluetooth Smart Ready device. To pair the compass with Windows, you must go to PC and devices->Bluetooth in the Settings page. This makes the Windows device discoverable and also starts it searching for other Bluetooth devices. Once your compass device appears on the right portion of the display click on it to pair.



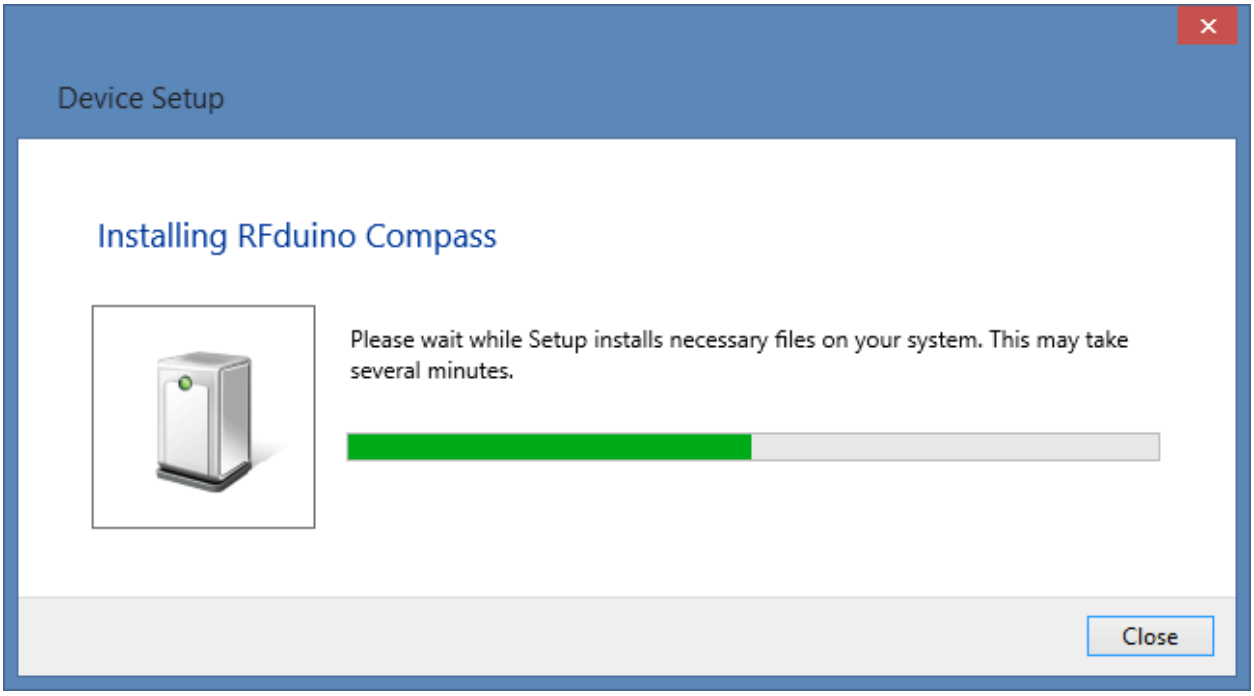
Windows will reply that the device is ready to pair.



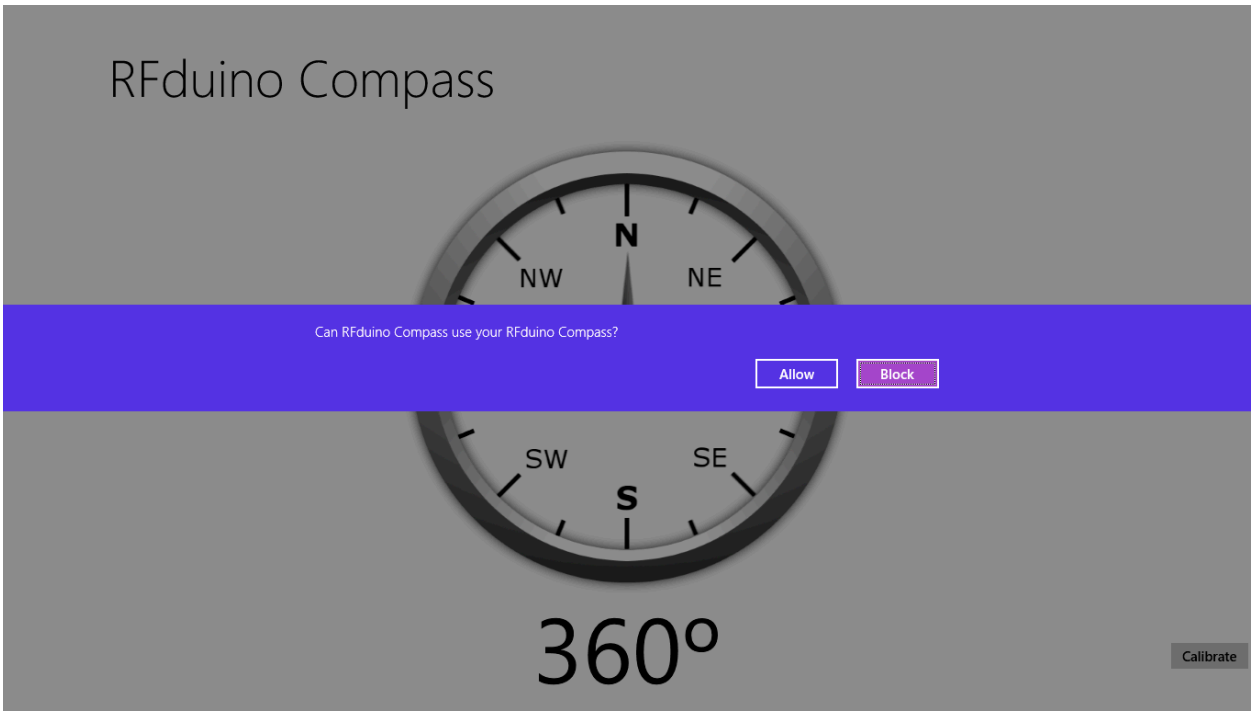
Hitting the "Pair" button will initiate the Bluetooth pairing sequence. At this point you can sit back and let Windows and the RFduino Compass do their magic and create a connection between them. Unfortunately, this magic may take over a minute to complete.



If you switch to the Desktop while pairing is in progress, you'll see the familiar Device Setup dialog.



After pairing is completed, you can run the RFduino Compass application. The first time you run the application, Windows will prompt for permission for the application to access the compass. If you select “Allow” the application will be granted permission to access the compass and it will start receiving updates. If you select “Block” the application will not be allowed to access the compass and will not receive heading updates.



As you can see from the sequence above, the user experience for interacting with Bluetooth Smart device in Windows 8.1 is far from optimal. Connecting and using devices requires far too much knowledge from the user to know where the various steps of connecting and using the device are performed. I'm optimistic this experience will be greatly improved in Windows 10.

Update Stability

The above code functions as desired. For a while… I've found that after a short amount of time, notifications are no longer delivered to the `headingValueChanged()` event handler method. The delay ranges from a few seconds to a few minutes. The RFduino Compass is still shown as connected in the Settings page and the compass itself still reflects being in a connection. Thus, it appears that something "breaks" between the notifications being received by Windows to the event handler being called. I've found this to be 100% consistent and reproducible during testing with Bluetooth radios from three different manufacturers. When using code which writes to a Characteristic, the write fails with a Bad Handle exception being thrown. However, when waiting for notifications, the application silently fails. I haven't found a solution to this and have open questions into Microsoft. In the mean-time, the following workaround is being used to refresh the connection at regular intervals.

*** Note: This issue wasn't experienced when testing with the Windows 10 preview release. I tested with build 10074. Thus, I would suggest upgrading to Windows 10 as soon as it is released.

Windows 8.1 Stability Workaround

After a short period of time after application startup, reads and writes to the Bluetooth Smart device through the Windows 8.1 fail with a Bad Handle exception. Reestablishing the connection at this time resolves the issue for another short period of time. If you have an application which regularly writes to the Bluetooth Smart device you can catch this error and reset the connection when it occurs. However, if the Bluetooth Smart device sends data to Windows more frequently then data is written to it, the data silently stops being delivered to the application. In this case, using a timer to regularly reset the connection keeps the data delivery to the application with minimal loss. The period of the timer will depend upon how tolerant the application is for lost updates and how often the Bluetooth Smart device sends those updates. The other parameter which should be considered in setting the period is how long your application receives data before the notifications stop being received. My testing revealed this to be as quick as 3-5 seconds.

For the RFduino Compass application which receives approximately two updates per seconds from the compass and isn't particularly sensitive to missing a heading update, I use a timer with a 5 second starting delay and then a period of 5 seconds as well. The code which starts the timer should be added to the `configureRFCompass()` method after the connection is established. I.e. there isn't any point in creating the timer if there isn't a compass available. A private reference to the Timer is also maintained such that it can be disposed of at application shutdown or if the page isn't visible. A Delegate to the CharacteristicChanged event handler is passed as the instance data to the timer such that the callback methods can remove it from the read Characteristic, which is saved in the `registerCharacteristicChangedCallback()`, method, before the connection code is called again.

```
using System.Threading;

private Timer Timer;
GattCharacteristic readCharacteristic;

TypedEventHandler<GattCharacteristic, GattValueChangedEventArgs> handler =
                                headingValueChanged;
if(this.Timer == null)
{
    Timer = new System.Threading.Timer(TimerCallback, handler, 5000, 5000);
}
```

```
/// <summary>
/// Callback method to watchdog timer. Reset compass configuration
/// </summary>
/// <param name="state">Not used.</param>
private void TimerCallback(object state)
{
    TypedEventHandler<GattCharacteristic, GattValueChangedEventArgs> valueChangedHandler
              = (TypedEventHandler<GattCharacteristic, GattValueChangedEventArgs>)state;
    if(this.readCharacteristic != null)
    {
        this.readCharacteristic.ValueChanged -= valueChangedHandler;
    }
    configureRFCompass();
}
```

With this workaround in place, the application receives and displays updates from the RFduino Compass for hours on end.

Conclusion

The Windows 8.1 Bluetooth Low Energy API greatly simplifies creating applications which interface with RFduino devices. While the Windows Store application used in the paper isn't a complete application as it lacks such features as icons, robust error handling, etc it demonstrates the key capabilities of the API. There is example code showing how to find devices implementing a particular Service and then how to obtain references to Characteristics within the Service. How data is exchanged with the RFduino through the read and write Characteristics is also shown. Some capabilities contained in the API, such as Descriptors, aren't shown in this paper.

This first generation API still lacks many features to enable a Windows 8.1 device to be a fully functional Bluetooth Smart Ready platform. Hopefully Microsoft will build upon this strong foundation to enable complete functionality and a better user experience for RFduino and other Bluetooth Smart devices in Windows 10.

The complete source code for the RFduino Compass code and Windows 8.1 application code may be found on github[8].

[8] Full source available at: https://github.com/RFduino/RFduinoApps/tree/master/Windows%20App/RFduinoCompassApp