Hands-On Lab

Bluetooth Smart Ready Application for Windows Phone 8.1

Lab version: 2.0.0

Last updated: 10/16/2014

Contents

REVISION HISTORY **3**

OVERVIEW **4**

EXERCISE 1: GETTING STARTED **5**

Task 1 – Create a new Project 5

Task 2 – Scanning for peripherals 6

Summary of Exercise 1 8

EXERCISE 2: COMMUNICATING WITH THE BLUETOOTH SMART DEVICE **9**

Task 1 - Adding the PeripheralControlPage 9

Summary of Exercise 2 20

Revision History

Version	Date	Author	Changes
1.0	1st May 2013	Matchbox	Initial version
2.0	16th October 2014	Martin Woolley, Bluetooth SIG	Make a Noise button now triggers the Immediate Alert service instead of a custom service. Switch added to enable/disable sharing of client time data with a new custom service called the Time Monitoring service.

Overview

This document is a step-by-step guide to creating a Bluetooth Smart Ready app for Windows Phone 8.1, using a Bluetooth Smart device implementing a GATT profile. For details on the specific profile and how to configure your Bluetooth Smart Device, please refer to the *Configuring a Bluetooth Smart Device Hands-on Lab* document downloaded with this package. In this lab we will not cover any basics of Windows Phone programming, nor is the resulting code suitable for production – the lab, tasks and code are designed to provide instruction in the principles and practice of Bluetooth Smart.

NOTE: This lab is not a complete commercial solution. All instructions relate directly to the hardware inventory shown below.

Objectives

This lab provides instructions to achieve the following:

- Discover a nearby Bluetooth Smart Device.
- Communicate with a Bluetooth Smart device.
- Implement a "key finder" app, using a Bluetooth Smart device.

To complete this lab you will first need to configure the Arduino Uno and Bluetooth Smart Shield. Please refer to the Hands-on Lab - Configuring a Bluetooth Smart Device document downloaded with this package.

System Requirements

- Latest Windows Phone 8.1 SDK
- Developer Studio 2013
- Windows 8.1

Equipment Requirements

- Arduino Uno
- A-to-B type USB cable
- Red Bear Labs Bluetooth Low Energy Shield
- Windows 8.1 device or better

Exercises

This hands-on lab contains the following exercises:

1. Scanning for Bluetooth Smart devices
2. Communicating with a Bluetooth Smart device

Estimated time to complete this lab: 45 to 60 minutes

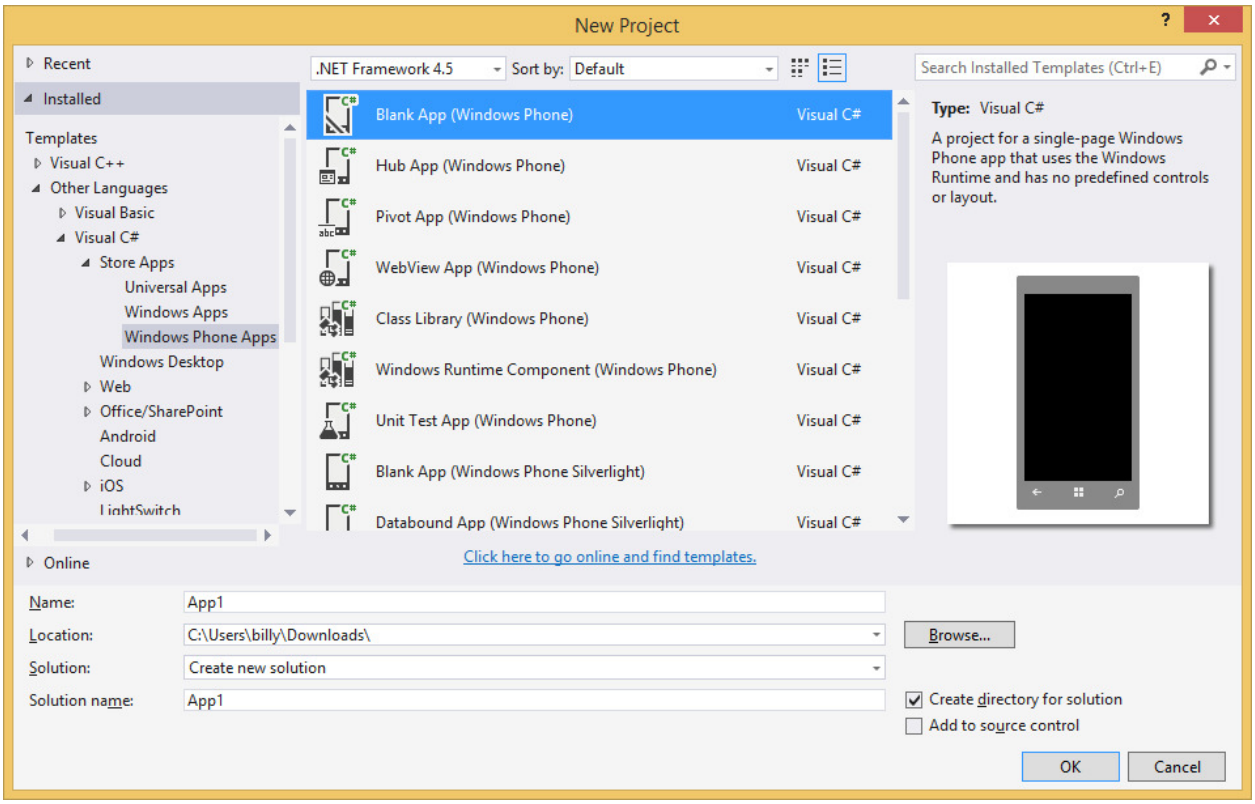
Exercise 1: Getting Started

This lab is all about scanning, connecting and talking to a Bluetooth Smart device. We will begin with a new Windows phone project.

> **NOTE** – you can type the code you see in this document into Visual Studio, or you can simply copy and paste from this document.
>
> Alternatively you can open the completed projects from *[Lab install folder/src/WindowsPhone]*:

Task 1 – Create a new Project

1. Run Visual Studio 2013
2. Select File > New > Project
3. Choose to create a Blank Windows Phone C# application



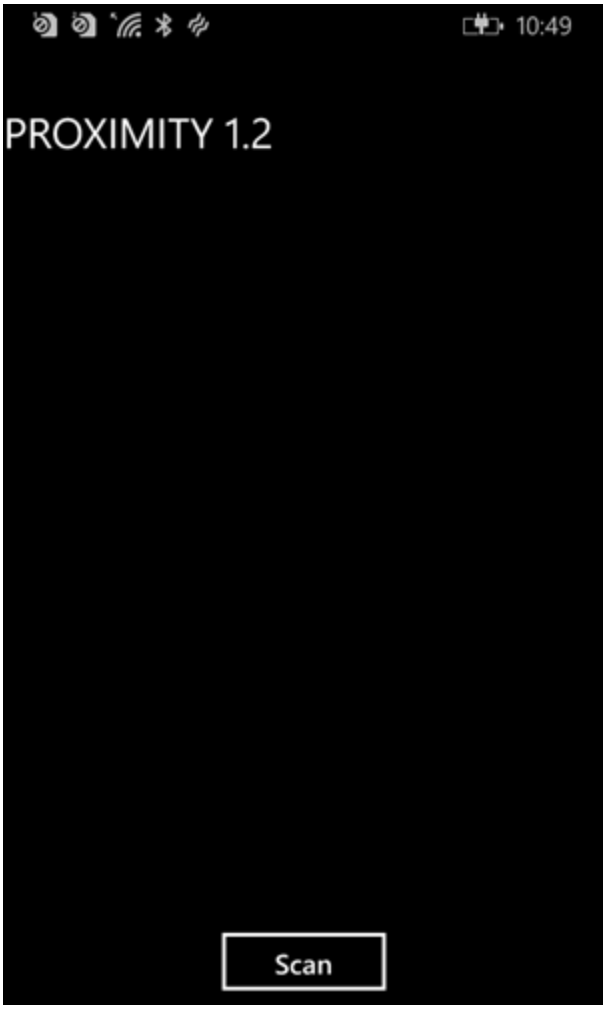
4. Select a suitable location and give the project a name (e.g ex1) and select Finish.

Task 2 – Scanning for peripherals

For this task, you will need an Windows Phone 8.1 device or above, your development machine, and a Bluetooth Smart peripheral powered up and in range.

> **NOTE:** This project will not work on Windows Phone 8.0 or earlier, as it depends on OS improvements made in Windows Phone 8.1

Here is a screenshot of how our page will look once we have scanned and found a device.



In order to use Bluetooth features in an application, we need to declare the Bluetooth permissions.

1. Right click on the *package.appxmanifest* file, and replace the Capabilities tag with the following.

xml

```
<Capabilities>
    <Capability Name="internetClientServer" />
      <m2:DeviceCapability Name="bluetooth.genericAttributeProfile">
        <m2:Device Id="any">
          <m2:Function Type="serviceId:1803" />
        </m2:Device>
      </m2:DeviceCapability>
</Capabilities>
```

`00001803-0000-1000-8000-00805f9b34fb` is the GUID of the Link Loss Service as define by Bluetooth SIG.

Now that the app has the right permissions, we will proceed with the Bluetooth behavior.
We will change the MainPage wizard generated class to display a list view and add a scan button. To start with we will create the required layout xml files.

2. Open MainPage.xaml and add the following between the <Grid> tag.

xml

```
<Grid>

        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <ListView Grid.Row="0" x:Name="deviceList"
SelectionChanged="ListView_SelectionChanged">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding}" FontSize="28" />
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
        <Button Grid.Row="1" x:Name="scanButton" Click="ScanButton_Click" Content="Scan"
HorizontalAlignment="Center" />

</Grid>
```

This adds a listview to the layout with a button aligned to the bottom of the screen.

We will use the button to start a scan, and display the results in the listview.

3. Open the *MainPage.xaml.cs* file, and add the following to the MainPage class above the constructor.

c#

```
public sealed partial class MainPage : Page
{

   Guid LINKLOSSSERVICEGUID = new Guid("00001803-0000-1000-8000-00805f9b34fb");
   DeviceInformationCollection foundDevices = null;

   public MainPage()
   {
      this.InitializeComponent();
      this.NavigationCacheMode = NavigationCacheMode.Required;
   }
```

This defines the GUID of the device we want to find, and a collection of found devices.

4. Add the following function to MainPage class.

C#

```
private void ListView_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
}

private async void ScanButton_Click(object sender, RoutedEventArgs e)
{
     foundDevices = await
DeviceInformation.FindAllAsync(GattDeviceService.GetDeviceSelectorFromUuid
 (LINKLOSSSERVICEGUID));

     int device_count = foundDevices.Count;
     deviceList.Items.Clear();
     for (int i = 0; i < foundDevices.Count; i++)
     {
         DeviceInformation deviceInfo = foundDevices[i];
         deviceList.Items.Add(deviceInfo.Name);
     }
 }
```

We define an empty place holder for the list click, which will be filled in in exercise 2.

The function `ScanButton_Click`() is called when the scan button on the page is pressed. It calls the FindAllAsync function to find all devices that have the SERVICEUUID and adds them to the list.

At this point, you should be able to run the application and scan for devices.

Summary of Exercise 1

And that is the first exercise done. To recap, in this exercise we performed these tasks:

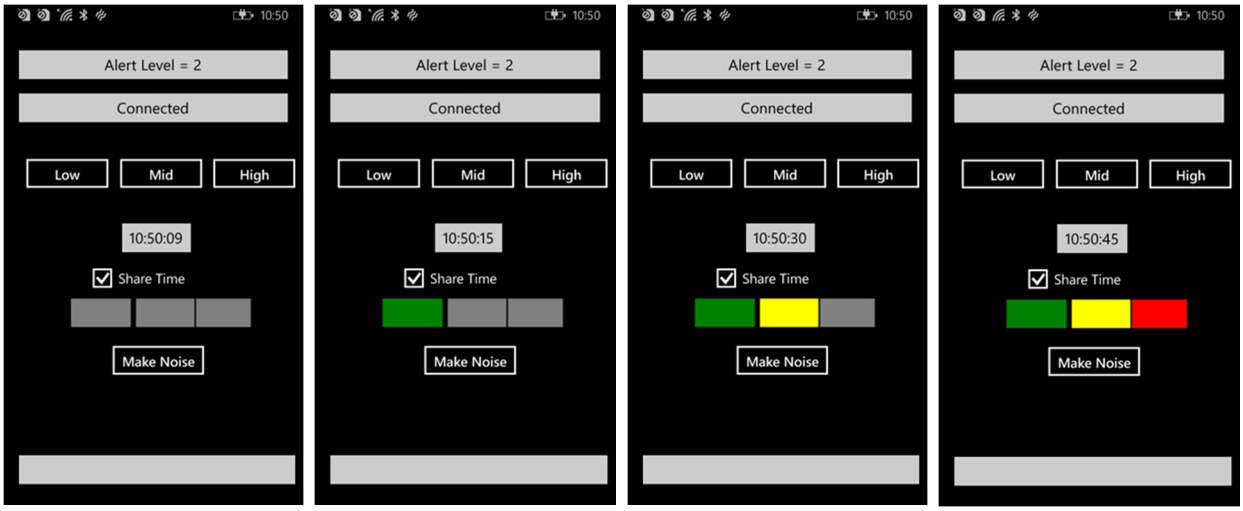
1. Created a blank C# Windows Phone 8.1 project
2. Added a list view and a scan button
3. Implemented the scan functionality and populated the list with devices.

Exercise 2: Communicating with the Bluetooth Smart device

NOTE – you can continue to extend ex1 by following Exercise 2.

Alternatively, the completed ex2 is included the solution from *[Lab install folder/src/WindowsPhone/]*

In this lab we will add a page that will allow us to control the Bluetooth device. Here is a screenshot of how the page will look when we're finished, in various states (with Time Sharing enabled).



Task 1 - Adding the PeripheralControlPage

1. Add a new class to the project called PeripheralControlPage. Once created, open the PeripheralControlPage.xaml and insert the following xml between the <Grid> tag

Xml

```
<Grid>

        <TextBox x:Name="alertLevel" TextAlignment="Center" HorizontalAlignment="Center"
Margin="21,34,21,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top" IsReadOnly="True"
Width="358"/>
         <TextBox x:Name="connectionStatus" TextAlignment="Center"
HorizontalAlignment="Center" Margin="21,92,21,0" TextWrapping="Wrap" Text=""
VerticalAlignment="Top" IsReadOnly="True" Width="358"/>
```

```
        <TextBox x:Name="time_now" TextAlignment="Center" HorizontalAlignment="Center"
Margin="146,265,137,0" TextWrapping="Wrap" Text="HH:MM:SS" VerticalAlignment="Top"
IsReadOnly="True"/>
        <Button x:Name="btn_low" Content="Low" Click="LowButton_Click"
HorizontalAlignment="Left" Margin="31,170,0,0" VerticalAlignment="Top"/>
        <Button x:Name="btn_mid" Content="Mid" Click="MidButton_Click"
HorizontalAlignment="Left" Margin="156,170,0,0" VerticalAlignment="Top"/>
        <Button x:Name="btn_high" Content="High" Click="HighButton_Click"
HorizontalAlignment="Left" Margin="280,170,0,0" VerticalAlignment="Top"/>
        <Button x:Name="btn_noise" Content="Make Noise" Click="MakeNoiseButton_Click"
HorizontalAlignment="Left" Margin="146,419,0,0" VerticalAlignment="Top"/>
        <TextBox x:Name="message" TextAlignment="Center" HorizontalAlignment="Center"
Margin="21,574,11,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="368"
IsReadOnly="True"/>
        <CheckBox x:Name="share_mode" Unchecked="HandleUnchecked" Content="Share Time"
HorizontalAlignment="Left" Margin="119,309,0,0" VerticalAlignment="Top"/>
        <Rectangle x:Name="Q1" HorizontalAlignment="Left" Height="40" Margin="89,364,0,0"
Stroke="Black" VerticalAlignment="Top" Width="82"/>
        <Rectangle x:Name="Q2" HorizontalAlignment="Left" Height="40"
Margin="176,364,0,0" Stroke="Black" VerticalAlignment="Top" Width="80"/>
        <Rectangle x:Name="Q3" HorizontalAlignment="Left" Height="40"
Margin="256,364,0,0" Stroke="Black" VerticalAlignment="Top" Width="75"/>

 </Grid>
```

2. Open *PeripheralControlPage.xaml.cs* and replace all the *using* statements with the following list.

C#

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Devices.Bluetooth;
using Windows.Devices.Bluetooth.GenericAttributeProfile;
using Windows.Storage.Streams;
using Windows.UI.Core;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
using Windows.Foundation;
using Windows.Phone.UI.Input;
using System.Threading.Tasks;
using Windows.UI.Xaml.Shapes;
using Windows.UI;
using Windows.UI.Xaml.Media;
```

3. Add the following declarations to the PeripheralControlPage class.

C#

```
public sealed partial class PeripheralControlPage : Page
{

        Guid LINKLOSSSERVICEGUID = new Guid("00001803-0000-1000-8000-00805f9b34fb");
        Guid IMMEDIATEALERTSERVICEGUID = new Guid("00001802-0000-1000-8000-
00805f9b34fb");
        Guid ALERTLEVELCHARACTERISTICGUID = new Guid("00002a06-0000-1000-8000-
00805f9b34fb");
        Guid PUSHBUTTONSERVICEGUID = new Guid("00009905-0000-1000-8000-00805f9b34fb");
        Guid PUSHBUTTONCHARACTERISTICGUID = new Guid("00009906-0000-1000-8000-
00805f9b34fb");
```

```
        // custom service which we use to send the current time as HH MM SS to the
Arduino Bluetooth Smart device
        Guid TIMEMONITORINGSERVICEUUID = new Guid("3e099912-293f-11e4-93bd-
afd0fe6d1dfd");
        Guid CLIENTTIMECHARACTERISTICUUID = new Guid("3e099913-293f-11e4-93bd-
afd0fe6d1dfd");

        // variable to hold the selected alert level value
        byte alert_level = 0;
```

4. Add the the following functions to the PeripheralControlPage class to match the event handlers we specified in the XAML.

C#

```
 private void LowButton_Click(object sender, RoutedEventArgs e)
 {
 }

 private void MidButton_Click(object sender, RoutedEventArgs e)
 {
 }

 private void HighButton_Click(object sender, RoutedEventArgs e)
 {
 }

 private async void MakeNoiseButton_Click(object sender, RoutedEventArgs e)
 {
}

private void HandleUnchecked(object sender, RoutedEventArgs e)
{
}
```

5. When the device is selected from the list we want to open PeripheralControlPage, passing in the id of the selected device. Open MainPage.xaml.cs and update the ListView_SelectionChanged() function

C#

```
private void ListView_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
     var lv = sender as ListView;
     if (lv != null)
     {
         if (lv.SelectedIndex >= 0 && lv.SelectedIndex < foundDevices.Count)
         {
             DeviceInformation devInfo = foundDevices[lv.SelectedIndex];
             this.Frame.Navigate(typeof(PeripheralControlPage), devInfo.Id);
         }
     }
}
```

This navigates to the PeripheralControlPage page passing the id of the selected device.

6. Add the following declarations to the PeripheralControlPage class, to hold the device we are going to connect to and a reference to the Time Monitoring GATT service which we will obtain shortly.

C#

```
BluetoothLEDevice mDevice = null;
GattDeviceService time_monitoring_service;
```

7. Update the OnNavigatedTo() function in the PeripheralControlPage class to get the device and initialise various Bluetooth GATT resources. We'll also set up a callback function to handle changes status of the connection to our Bluetooth device whilst we're in this area of the code. In addition, display the current time on the UI and set a timer task running so that it's updated every second. Create the TimerTask function too. This function will be called once every second in a background thread.

C#

```
 protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    mDevice = await BluetoothLEDevice.FromIdAsync(e.Parameter.ToString());
    if (mDevice != null)
    {
        // update connect text in the UI
        connectionStatus.Text = mDevice.ConnectionStatus.ToString();

        if (mDevice.ConnectionStatus == BluetoothConnectionStatus.Disconnected)
        {
            btn_low.IsEnabled = false;
            btn_mid.IsEnabled = false;
            btn_high.IsEnabled = false;
            btn_noise.IsEnabled = false;
            share_mode.IsEnabled = false;
            return;
        }

        initialiseGattResources();

    }

    DateTime now = DateTime.Now;
    string date_patt = @"HH:mm:ss";
    string hhmmss = now.ToString(date_patt);
    time_now.Text = hhmmss;

    // Start timer task to collect (and if necessary share using Bluetooth GATT) the
system time
    System.Threading.TimerCallback TimerDelegate = new
System.Threading.TimerCallback(TimerTask);
    TimerItem = new System.Threading.Timer(TimerDelegate, null, 1000, 1000);
}
```

```
        private async void initialiseGattResources()
        {
            // hook up the connection status change callback
            mDevice.ConnectionStatusChanged += ConnectionChanged;

            // get the link loss service
            GattDeviceService service = mDevice.GetGattService(LINKLOSSSERVICEGUID);
            if (service != null)
            {
                // get the characteristic
                GattCharacteristic characteristic =
service.GetCharacteristics(ALERTLEVELCHARACTERISTICGUID)[0];
                if (characteristic != null)
                {
                    // read its current value
                    GattReadResult readValue = await characteristic.ReadValueAsync();

        UpdateAlertLevel(Windows.Storage.Streams.DataReader.FromBuffer(readValue.Value).ReadByte(
));
                }
            }
            // get the time monitoring service
            time_monitoring_service = mDevice.GetGattService(TIMEMONITORINGSERVICEUUID);

        }
        private async void TimerTask(object obj)
        {
            DateTime now = DateTime.Now;
            string date_patt = @"HH:mm:ss";
            string hhmmss = now.ToString(date_patt);
            int hh = now.Hour;
            int mm = now.Minute;
            int ss = now.Second;
            await dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
            {
                time_now.Text = hhmmss;
                // later in this lab, we'll interact with the Time Monitoring Service here
            });
        }
```

The device id is passed into the OnNavigateTo function when it is selected from the list.

The function then gets the BluetoothLEDevice from the Id. There is no need to connect as this happens automatically when we get the device.

If the mDevice is found we then update the connection status text in the UI.

Then we get the link loss service, and the characteristic, and finally read its value and update the UI.

At this stage, all our TimerTask function does is to update the system time in the UI once per second. Later on we'll use it to share the current system time with our Bluetooth Arduino device using its Time Monitoring Service.

8. We used a helper function called updateAlertLevel in the code above to update the alert level. Let's now add that function to the PeripheralControlPage class.

C#

```
private void UpdateAlertLevel(byte value)
```

```
{
        alert_level = value;
        alertLevel.Text = "Alert Level = " + value.ToString();
}
```

This code just displays the alert level on the page.

9. We also referenced a callback function called ConnectionChanged in the code above. Just add a skeleton implementation for now:

```
async void ConnectionChanged(BluetoothLEDevice device, object obj)
{
}
```

10. The click handlers for the three buttons setting the alert level to Low, Mid and High do the same thing, except they set the alert level to a different value. Add the following button handlers to PeripheralControlPage class. They will call the SetLevel() helper function.

C#

```
private void LowButton_Click(object sender, RoutedEventArgs e)
{
    SetLevel(0);
}

private void MidButton_Click(object sender, RoutedEventArgs e)
{
    SetLevel(1);
}

private void HighButton_Click(object sender, RoutedEventArgs e)
{
    SetLevel(2);
}
```

11. Setting the level is similar to reading it, which we did in the OnNavigateTo() function, except we write instead of read. Add the following function to PeripheralControlPage class.

C#

```
private async void SetLevel(byte value)
{
    if (mDevice == null)
    {
        Debug.WriteLine("SetLevel: No device available - ignoring request");
        return;
    }
    // get the link loss service
    GattDeviceService service = mDevice.GetGattService(LINKLOSSSERVICEGUID);
    if (service != null)
    {
```

```
                // get the characteristic
                var characteristic = service.GetCharacteristics(ALERTLEVELCHARACTERISTICGUID)[0];
                if (characteristic != null)
                {
                    // create a data writer to write the value
                    DataWriter writer = new DataWriter();
                    writer.WriteByte(value);
                    // Attempt to write the data to the device
                    // and whist doing so get the status.
                    GattCommunicationStatus status = await
characteristic.WriteValueAsync(writer.DetachBuffer());
                    if (status == GattCommunicationStatus.Success)
                    {
                        UpdateAlertLevel(value);
                    }
                }
            }
        }
```

12. To implement the key finder part of our app we need to tell the Arduino board to make a noise when the 'make noise' button is pressed. To do this we just need to write the current alert level value to the Immediate Alert Service's Alert Level characteristic. Add the implementation for the 'make noise' button to the PeripheralControlPage class.

C#

```
        private async void MakeNoiseButton_Click(object sender, RoutedEventArgs e)
        {
            Debug.WriteLine("MakeNoiseButton_Click");
            // get the immediate alert service
            try
            {
                GattDeviceService service = mDevice.GetGattService(IMMEDIATEALERTSERVICEGUID);
                if (service != null)
                {
                    // get the alert level characteristic
                    var characteristic =
service.GetCharacteristics(ALERTLEVELCHARACTERISTICGUID)[0];
                    if (characteristic != null)
                    {
                        // create a data writer to write the value
                        DataWriter writer = new DataWriter();
                        writer.WriteByte(alert_level);
                        // Attempt to write the data to the device
                        GattCommunicationStatus status = await
characteristic.WriteValueAsync(writer.DetachBuffer(), GattWriteOption.WriteWithoutResponse);
                    }
                }
                else
                {
                    Debug.WriteLine("MakeNoiseButton_Click did NOT find GattDeviceService");
                }
            }
            catch (Exception ex)
            {
                Debug.WriteLine("MakeNoiseButton_Click: exception {0}:{1}", ex.GetType(),
ex.Message);
            }
```

13. We want to detect when the device disconnects and to play an alarm sound. We'll only do this when the alert level is set to 2 (i.e. High). Add a Boolean to PeripheralControlPage class which will be used to tell the code if it should play the sound or not on a disconnect event.

C#

```
Boolean mPlaySoundOnDisconnect = false;
```

14. Add a sound file to the project. Copy the *beep.wav* file from *[Lab install folder/src/assets]* to this projects assets folder, then right click on the asserts folder in Visual Studio solution explorer, choose Add existing item, and select the *beep.wav* file

15. We will set this to *true* if the alert level is set to 2. Edit the UpdateAlertLevel() function with the following code.

C#

```
private void UpdateAlertLevel(byte value)
{
    alert_level = value;
    if (value == 2)
    {
        mPlaySoundOnDisconnect = true;
    }
    else
    {
        mPlaySoundOnDisconnect = false;
    }

    alertLevel.Text = "Alert Level = " + value.ToString();
}
```

16. Add a MediaElement to PeripheralControlPage.xaml. We'll use it to play the sound. Add the following xml between the <Grid> tags.

xml

```
<MediaElement x:Name="mediaPlayer" Volume="1" Source="ms-appx:///assets/beep.wav"
AutoPlay="False"/>
```

We will use this MediaElement to play the sound on the disconnect event.

17. The disconnect event will be implemented as a call back. It is not called on the UI thread. We want to update the connection status and play the *beep.wav* file on the UI thread. We do this using a dispatcher. Let's add the code for the dispatcher in the constructor of the PeripheralControlPage class. Remember to declare the CoreDispatcher object as shown.

C#

```
CoreDispatcher mDispatcher = null;

public PeripheralControlPage()
{
    this.InitializeComponent();
     mDispatcher = Window.Current.Dispatcher;
}
```

18. Finally, we need to complete the call back function, ConnectionChanged(). Note that we enable or disable various UI controls according to the new connection state.

C#

```
async void ConnectionChanged(BluetoothLEDevice device, object obj)
{
      await dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
      {
          connectionStatus.Text = device.ConnectionStatus.ToString();
          if (device.ConnectionStatus == BluetoothConnectionStatus.Disconnected)
          {
              Debug.WriteLine("ConnectionChanged to Disconnected");
              btn_low.IsEnabled = false;
              btn_mid.IsEnabled = false;
              btn_high.IsEnabled = false;
              btn_noise.IsEnabled = false;
              share_mode.IsEnabled = false;
          }
          else
          {
              Debug.WriteLine("ConnectionChanged to Connected");
              btn_low.IsEnabled = true;
              btn_mid.IsEnabled = true;
              btn_high.IsEnabled = true;
              btn_noise.IsEnabled = true;
              share_mode.IsEnabled = true;
          }
          if (mPlaySoundOnDisconnect && device.ConnectionStatus ==
BluetoothConnectionStatus.Disconnected) { mediaPlayer.Play(); }
      });
}
```

This code updates the connect status text, and plays the beep sound if it's a disconnect event and `mPlaySoundOnDisconnect`is true.

19. Our next task is to make use of the Time Monitoring Service whenever the share_mode checkbox is selected. Update the TimerTask function as shown:

```
        private async void TimerTask(object obj)
        {
            DateTime now = DateTime.Now;
            string date_patt = @"HH:mm:ss";
            string hhmmss = now.ToString(date_patt);
            int hh = now.Hour;
            int mm = now.Minute;
            int ss = now.Second;
            await dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
            {
                time_now.Text = hhmmss;
                if (share_mode.IsChecked == true)
                {
                    WriteGattClientTime(hh, mm, ss);
                }
             });
        }


        private void WriteGattClientTime(int hh, int mm, int ss)
        {
            Debug.WriteLine("Sharing time with Bluetooth GATT Time Monitoring service....");
            if (time_monitoring_service != null)
            {
                // get the characteristic
                var characteristic =
time_monitoring_service.GetCharacteristics(CLIENTTIMECHARACTERISTICUUID)[0];
                if (characteristic != null)
                {
                    // create a data writer to write the value
                    DataWriter writer = new DataWriter();
                    writer.WriteByte((byte)hh);
                    writer.WriteByte((byte)mm);
                    writer.WriteByte((byte)ss);
                    if (ss >= 0)
                    {
                        ColorRectangle(Q1, Colors.Gray);
                        ColorRectangle(Q2, Colors.Gray);
                        ColorRectangle(Q3, Colors.Gray);
                    }
                    if (ss >= 15)
                    {
                        ColorRectangle(Q1, Colors.Green);
                    }
                    if (ss >= 30)
                    {
                        ColorRectangle(Q2, Colors.Yellow);
                    }
                    if (ss >= 45)
                    {
                        ColorRectangle(Q3, Colors.Red);
                    }
                    // Attempt to write the data to the device
                    characteristic.WriteValueAsync(writer.DetachBuffer(),
GattWriteOption.WriteWithoutResponse);
                }
            }
            else
            {
                Debug.WriteLine("ERROR: WriteGattClienTime could not obtain client time
characteristic");
                message.Text = "ERROR: WriteGattClienTime could not obtain client time
characteristic";
            }
        }

        private void ColorRectangle(Rectangle rectangle, Color color)
        {
            SolidColorBrush solid_color_brush = new SolidColorBrush();
            solid_color_brush.Color = color;
            rectangle.Fill = solid_color_brush;
```

```
        }
```

20. Finally, we have some miscellaneous taks left to finish our application fully. First, let's make sure all LEDs are switched off on the circuit board our Arduino Bluetooth device is controlling, whenever the share mode checkbox is unchecked. Fill in the body of the HandleUnchecked event handler as shown:

```
        private void HandleUnchecked(object sender, RoutedEventArgs e)
        {
            // send values to GATT Time Monitoring Service which will cause all LEDs to be
switched off
            WriteGattClientTime((byte) 00, (byte) 00, (byte) 00);
            ColorRectangle(Q1, Colors.Black);
            ColorRectangle(Q2, Colors.Black);
            ColorRectangle(Q3, Colors.Black);
        }
```

And finally, when the user selects the back button, we want to make sure that any LEDs which may be lit at that point get switched off and that we dispose of our timer object:

```
    public PeripheralControlPage()

     {

         this.InitializeComponent();

         dispatcher = Window.Current.Dispatcher;

         Windows.Phone.UI.Input.HardwareButtons.BackPressed += HardwareButtons_BackPressed;
     }

       private void HardwareButtons_BackPressed(object sender, BackPressedEventArgs e)
       {
         Debug.WriteLine("HardwareButtons_BackPressed....");
          if (this.Frame.CanGoBack)
          {
              e.Handled = true;
              this.Frame.GoBack();
          }
       }

       protected override void OnNavigatingFrom(NavigatingCancelEventArgs e)
       {
           Debug.WriteLine("OnNavigatingFrom....");
           if (TimerItem != null)
           {
               TimerItem.Dispose();
               Debug.WriteLine("Disposed of timer....");
           }
           if (share_mode.IsChecked == true)
           {
               WriteGattClientTime((byte)00, (byte)00, (byte)00);
           }
           Debug.WriteLine("Switched off LEDs....");
       }
```

Summary of Exercise 2

You now have all the pieces to run the full demonstration scenario.

Build and run the application now and connect to the Bluetooth Smart Device. If you set the alert level to high and keep getting farther away from the board you will hear a siren. In this event, you should note the board will also make a noise until either it has looped around 100 times or the reset button is clicked.

Well done! *By following this hands-on lab you have created an application that scans for Bluetooth Smart devices, scans their services, connects to them, reads and writes characteristic values.*