Windows 10 IoT – Bluetooth Beacons

The use of Bluetooth has made untethered communication over short distances very easy. However, there is generally a manual process for pairing the two devices together before they are able to communicate. This lab demo shows how to automatically pair a Bluetooth beacon to a device and when the beacon is close enough to the device act upon it.

APPROXIMATE TIME (EXCLUDING PREPARATION WORK): 30 Minutes

PREREQUISITES:

- Setup / Install Windows 10 (v 10.0.10240) or higher
- Download Windows 10 IoT Core Dashboard (http://go.microsoft.com/fwlink/?LinkID=708576)
- Setup / Install Visual Studio 2015 Update 1 (You can download Visual Studio Community Edition For Free: http://go.microsoft.com/fwlink/?LinkID=534599)
- Install IoT Core Project Templates (https://visualstudiogallery.msdn.microsoft.com/55b357e1-a533-43ad-82a5-a88ac4b01dec)

PARTS LIST:

- Raspberry Pi 2
- Supported Bluetooth Dongle (http://ms-iot.github.io/content/en-US/win10/SupportedInterfaces.htm#Bluetooth-Dongles)
- Bluetooth Beacon (this lab is programmed to use XY Find It BLE Beacons http://www.xyfindit.com/)
- Speakers (if using audio)

HARDWARE SETUP:

There is no specific hardware setup in this lab, it is primarily code. The Raspberry Pi should be setup and connected to a monitor, network cable plugged in, speakers plugged in, and Bluetooth dongle plugged into an available USB Port

SOFTWARE SETUP:

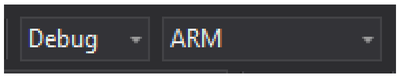
A) Preparing The Device
The device should already have an image of Windows 10 IoT core on it, but if it does not, you can always install it onto the SD Card.
  1. Launch Windows 10 IoT Core Dashboard
  2. Click "Set up a new device" in the left hand menu
  3. Select the appropriate Device Type and OS Image
  4. Insert the Micro SD card into your computer, click "Download and Install" and follow the directions

B) Boot The Device
It may take ~5 minutes for the device to boot up for the first time.

C) Connecting To The Device
When online, the device should appear under the "My Devices" tab on the IoT Core Dashboard.
  1. Note the IP address of the machine and either connect to http://<IP_ADDRESS>:8080 or click the globe under "Open in device Portal" for that specific device
  2. The default user name is Administrator
  3. The default password is p@ssw0rd

D) <u>Deploying To The Device</u>

1. In Visual Studio, on the build bar, make sure that the appropriate build / architecture is selected:



2. Right click the project name and select properties
3. In the settings dialog, click the Debug option
4. Under start options (If the device shows up under IoT Core Dashboard, you should be able to hit “Find” to populate the data
   - i. Target Device: Remote Machine
   - ii. Remote Machine: <IP ADDRESS>
   - iii. Authentication Mode: Universal
5. Build / Deploy the project to the device

APPLICATION

This application uses the DeviceWatcher class to look for and enumerate devices. It constantly looks for the signal strength to determine whether or not to pair the device and speak the text. The sample does have a visual element that binds to a back end collection to show the available devices and their current status.

A) <u>Create A New Project</u>

- A) Launch Visual Studio
- B) Select “New Project”
- C) Under “Templates”, select “Visual C#” and then “Blank App (Universal Windows)”
- D) Give the app a name and location. The code sample below assumes you have called your application “BT-BeaconApp”

B) <u>Adding The Code</u>

Create Class For Displaying Device Information

1. Right click project name -> Add -> Class
2. Name the file DisplayHelper.cs and click Add
3. Remove the scaffolded code and start with the following blank class file

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using Windows.Devices.Enumeration;
using Windows.UI.Xaml.Media.Imaging;

namespace BT_BeaconApp
{

}
```

4. Add a new class to the namespace that will handle the additional properties that we want to get from the endpoint device. In this case, we specifically care about the signal strength

```
public class DeviceInformationDisplay : INotifyPropertyChanged
{
    private DeviceInformation deviceInfo;

    public DeviceInformationDisplay(DeviceInformation deviceInfoIn)
    {
        deviceInfo = deviceInfoIn;
        UpdateGlyphBitmapImage();
    }

}
```

5. Add a new class to the namespace that will manage all of the properties that the UI will display. This class inherits from INotifyPropertyChanged to automatically fire an event when the property changes.

```
public static class DeviceProperties
{
    public static List<string> AssociationEndpointProperties
    {
        get
        {
            List<string> properties = new List<string>();
            properties.Add("System.Devices.Aep.SignalStrength");

            return properties;
        }
    }
}
```

6. Add the public properties to the class

```
public DeviceInformationKind Kind
{
    get { return deviceInfo.Kind; }
}

public string Id
{
    get { return deviceInfo.Id; }
}

public string Name
{
    get { return deviceInfo.Name; }
}

public BitmapImage GlyphBitmapImage
{
    get; private set;
}

public bool IsPairing
{
    get; set;
}

public bool CanPair
{
    get { return deviceInfo.Pairing.CanPair; }
}

public bool IsPaired
{
    get { return deviceInfo.Pairing.IsPaired; }
}

public int SignalStrength
{
    get
    {
        int val = int.MinValue;
        int.TryParse(deviceInfo.Properties["System.Devices.Aep.SignalStrength"]
             .ToString(), out val);

        return val;
    }
}

public IReadOnlyDictionary<string, object> Properties
{
    get { return deviceInfo.Properties; }
}

public DeviceInformation DeviceInformation
{
    get { return deviceInfo; }

    private set { deviceInfo = value; }
}
```

7. Add additional methods and event handlers

```
public void Update(DeviceInformationUpdate deviceInfoUpdate)
    {
        deviceInfo.Update(deviceInfoUpdate);

        OnPropertyChanged("Kind");
        OnPropertyChanged("Id");
        OnPropertyChanged("Name");
        OnPropertyChanged("DeviceInformation");
        OnPropertyChanged("CanPair");
        OnPropertyChanged("IsPaired");
        OnPropertyChanged("SignalStrength");

        UpdateGlyphBitmapImage();
    }

    private async void UpdateGlyphBitmapImage()
    {
        DeviceThumbnail deviceThumbnail = await deviceInfo.GetGlyphThumbnailAsync();
        BitmapImage glyphBitmapImage = new BitmapImage();
        await glyphBitmapImage.SetSourceAsync(deviceThumbnail);
        GlyphBitmapImage = glyphBitmapImage;
        OnPropertyChanged("GlyphBitmapImage");
    }

    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string name)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(name));
        }
    }
```

Add UI Code

1. Open up MainPage.xaml

2. Add the following template to display data about each beacon

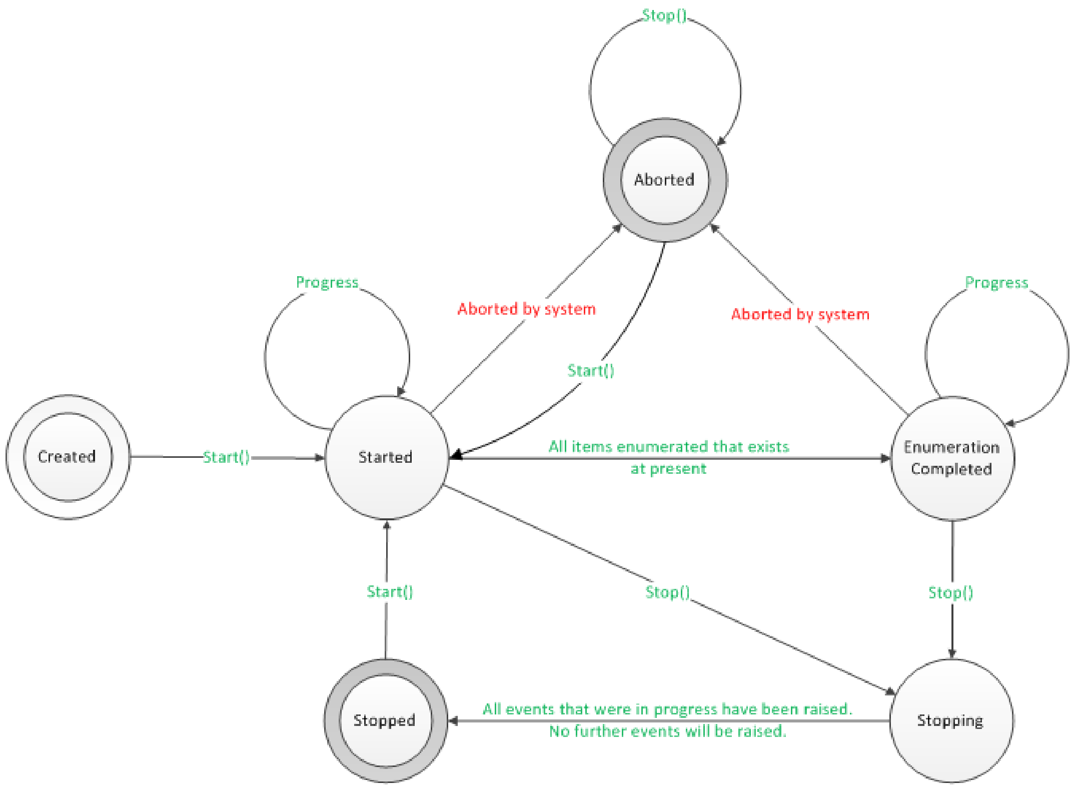
```
<Page.Resources>

        <DataTemplate x:Key="ResultsListViewTemplate">
            <Grid Margin="5">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto"/>
                    <ColumnDefinition Width="*" MinWidth="100"/>
                </Grid.ColumnDefinitions>
                <Border Grid.Column="0" Height="40" Width="40" Margin="5" VerticalAlignment="Top">
                    <Image Source="{Binding Path=GlyphBitmapImage}"
                           Stretch="UniformToFill"/>
                </Border>
                <Border Grid.Column="1" Margin="5">
                    <StackPanel>
                        <StackPanel Orientation="Horizontal">
                            <TextBlock Text="Name:" Margin="0,0,5,0"/>
                            <TextBlock Text="{Binding Path=Name}" FontWeight="Bold" TextWrapping="WrapWholeWords"/>
                        </StackPanel>
                        <StackPanel Orientation="Horizontal">
                            <TextBlock Text="Id:" Margin="0,0,5,0"/>
                            <TextBlock Text="{Binding Path=Id}" TextWrapping="Wrap"/>
                        </StackPanel>
                        <StackPanel Orientation="Horizontal">
                            <TextBlock Text="SignalStrength:" Margin="0,0,5,0"/>
                            <TextBlock Text="{Binding Path=SignalStrength}"/>
                        </StackPanel>
                        <StackPanel Orientation="Horizontal">
                            <TextBlock Text="CanPair:" Margin="0,0,5,0"/>
                            <TextBlock Text="{Binding Path=CanPair}"/>
                        </StackPanel>
                        <StackPanel Orientation="Horizontal">
                            <TextBlock Text="IsPaired:" Margin="0,0,5,0"/>
                            <TextBlock Text="{Binding Path=IsPaired}"/>
                        </StackPanel>
                    </StackPanel>
                </Border>
            </Grid>
        </DataTemplate>

    </Page.Resources>
```

3. Replace the existing Grid code with the following code

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <ScrollViewer Grid.Row="0" VerticalScrollMode="Auto" VerticalScrollBarVisibility="Auto">
        <StackPanel HorizontalAlignment="Left" VerticalAlignment="Top" Margin="8,5,15,0">
            <TextBlock Text="Beacons:" Style="{StaticResource SampleHeaderTextStyle}"/>

            <Border BorderBrush="AntiqueWhite" BorderThickness="1">
                <ListView x:Name="resultsListView"
                          ItemTemplate="{StaticResource ResultsListViewTemplate}"
                          ItemsSource="{Binding Path=ResultCollection}"
                          MaxHeight="450">
                </ListView>
            </Border>
        </StackPanel>
    </ScrollViewer>
</Grid>
```

Add UI Code Behind

The device watcher is the primary process that is running and is alerted to changes within the devices in the range. From MSDN, this is the general behavior that it uses for looking at devices:



1. Open MainPage.xaml.cs

2. Remove the original code and start with the following template

```
using System;
using System.Collections.ObjectModel;
using System.Diagnostics;
using System.Linq;
using Windows.Devices.Enumeration;
using Windows.Media.SpeechSynthesis;
using Windows.UI.Core;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace BT_BeaconApp
{
    public sealed partial class MainPage : Page
    {
        private DeviceWatcher _deviceWatcher = null;
        private bool _restartWatcher = false;

        SpeechSynthesizer _speechSynth = null;
        MediaElement _audio = null;

        // Collect of devices bound to the UI
        public ObservableCollection<DeviceInformationDisplay> ResultCollection
        {
            get;
            private set;
        }

        public MainPage()
        {
            this.InitializeComponent();
        }

        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            // Initialize the objects
            ResultCollection = new ObservableCollection<DeviceInformationDisplay>();
            DataContext = this;

            _speechSynth = new SpeechSynthesizer();
            _speechSynth.Voice =
SpeechSynthesizer.AllVoices.DefaultIfEmpty(SpeechSynthesizer.DefaultVoice).FirstOrDefault(p =>
p.DisplayName.Contains("Mark"));
            _audio = new MediaElement();

            StartDeviceWatcher();
        }

        protected override void OnNavigatedFrom(NavigationEventArgs e)
        {
            StopDeviceWatcher();
        }
    }
}
```

3. Add code to start / stop the watcher

```
        private void StartDeviceWatcher()
        {
            ResultCollection.Clear();

            // Currently Bluetooth APIs don't provide a selector to get ALL devices that are both paired and non-
paired.  Typically you wouldn't need this for common scenarios,
            // but it's convenient to demonstrate the various sample scenarios.
            string selector = "(System.Devices.Aep.ProtocolId:=\"{bb7bb05e-5972-42b5-94fc-76eaa7084d49}\")" + " AND
(System.Devices.Aep.CanPair:=System.StructuredQueryType.Boolean#True OR
System.Devices.Aep.IsPaired:=System.StructuredQueryType.Boolean#True)";

            // Kind is specified in the selector info
            _deviceWatcher = DeviceInformation.CreateWatcher(
                selector,
                DeviceProperties.AssociationEndpointProperties,
                DeviceInformationKind.AssociationEndpoint);

            // Hook up events for the watcher
            _deviceWatcher.Added += OnDeviceAdded;
            _deviceWatcher.Updated += OnDeviceUpdated;
            _deviceWatcher.Removed += OnDeviceRemoved;
            _deviceWatcher.EnumerationCompleted += OnEnumerationCompleted;
            _deviceWatcher.Stopped += OnStopped;

            _deviceWatcher.Start();
        }

        private void StopDeviceWatcher()
        {
            if (null != _deviceWatcher)
            {
                // First unhook all event handlers except the stopped handler. This ensures our
                // event handlers don't get called after stop, as stop won't block for any "in flight"
                // event handler calls.  We leave the stopped handler as it's guaranteed to only be called
                // once and we'll use it to know when the query is completely stopped.
                _deviceWatcher.Added -= OnDeviceAdded;
                _deviceWatcher.Updated -= OnDeviceUpdated;
                _deviceWatcher.Removed -= OnDeviceRemoved;
                _deviceWatcher.EnumerationCompleted -= OnEnumerationCompleted;

                if (DeviceWatcherStatus.Started == _deviceWatcher.Status ||
                    DeviceWatcherStatus.EnumerationCompleted == _deviceWatcher.Status)
                {
                    _deviceWatcher.Stop();
                }
            }
        }
```

4. Add code to handle pairing / unpairing. In this code, you may need to change parameters to vary behavior based on signal strength, or if you BLE beacons have a different name. Signal strength varies, but -50 on these beacons is ~1 foot.

```
private async void CheckPairing(DeviceInformationDisplay deviceInfoDisplay)
        {
            Debug.WriteLine("{0}- '{1}' Signal Strength - {2}", DateTime.Now.Ticks, deviceInfoDisplay.Name,
deviceInfoDisplay.SignalStrength);

            if (!deviceInfoDisplay.IsPairing)
            {
                // Pair device only if it is close
                if (deviceInfoDisplay.Name.StartsWith("XY") &&
                    !deviceInfoDisplay.IsPaired &&
                    (deviceInfoDisplay.SignalStrength > -50))
                {
                    // Set a flag on the device so that once it begins to pair, it doesn't constantly try to pair
                    deviceInfoDisplay.IsPairing = true;

                    // This is just basic pairing (No PIN or confirmation)
                    DevicePairingKinds ceremonies = DevicePairingKinds.ConfirmOnly;
                    DevicePairingProtectionLevel protectionLevel = DevicePairingProtectionLevel.Default;
                    DeviceInformationCustomPairing customPairing = deviceInfoDisplay.DeviceInformation.Pairing.Custom;

                    // In the cases where more complex pairing is required, user interaction happens in the Pairing
Requested event
                    customPairing.PairingRequested += OnPairingRequested;
                    DevicePairingResult result = await customPairing.PairAsync(ceremonies, protectionLevel);
                    customPairing.PairingRequested -= OnPairingRequested;

                    Debug.WriteLine("{0} pair result - {1}", deviceInfoDisplay.Name, result.Status);

                    // If the pair was successful, act based on the beacon ID
                    if(DevicePairingResultStatus.Paired == result.Status)
                    {
                        var stream = await _speechSynth.SynthesizeTextToStreamAsync("Paired beacon ID " +
deviceInfoDisplay.Name);

                        // Use a dispatcher to play the audio
                        var ignored = Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
                        {
                            //Set the souce of the MediaElement to the SpeechSynthesisStream
                            _audio.SetSource(stream, stream.ContentType);

                            //Play the stream
                            _audio.Play();
                        });

                    }

                    deviceInfoDisplay.IsPairing = false;
                }
                else if (deviceInfoDisplay.Name.StartsWith("XY") &&
                        deviceInfoDisplay.IsPaired &&
                        (deviceInfoDisplay.SignalStrength < -50))
                {
                    // Set a flag on the device so that once it begins to pair, it doesn't constantly try to unpair
                    deviceInfoDisplay.IsPairing = true;

                    // Unpair the beacon
                    DeviceUnpairingResult result = await deviceInfoDisplay.DeviceInformation.Pairing.UnpairAsync();
                    Debug.WriteLine("{0} unpair result - {1}", deviceInfoDisplay.Name, result.Status);

                    deviceInfoDisplay.IsPairing = false;
                }
            }

        }
```

5. Add DeviceAdd event

```
        private async void OnDeviceAdded(DeviceWatcher watcher, DeviceInformation deviceInfoAdded)
        {
            // Since we have the collection databound to a UI element, we need to update the collection on the UI 
thread.
            await Dispatcher.RunAsync(CoreDispatcherPriority.Low, () =>
            {
                DeviceInformationDisplay deviceInfoDisplay = new DeviceInformationDisplay(deviceInfoAdded);

                if (!ResultCollection.Any(p => p.Name == deviceInfoAdded.Name))
                {
                    ResultCollection.Add(deviceInfoDisplay);
                    Debug.WriteLine("{0} devices found.", ResultCollection.Count);
                }

                CheckPairing(deviceInfoDisplay);
            });
        }
```

6. Add DeviceUpdated event

```
        private async void OnDeviceUpdated(DeviceWatcher watcher, DeviceInformationUpdate deviceInfoUpdate)
        {
            // Since we have the collection databound to a UI element, we need to update the collection on the UI 
thread.
            await Dispatcher.RunAsync(CoreDispatcherPriority.Low, () =>
            {
                // Find the corresponding updated DeviceInformation in the collection and pass the update object
                // to the Update method of the existing DeviceInformation. This automatically updates the object
                // for us.
                foreach (DeviceInformationDisplay deviceInfoDisplay in ResultCollection)
                {
                    if (deviceInfoDisplay.Id == deviceInfoUpdate.Id)
                    {
                        CheckPairing(deviceInfoDisplay);
                        deviceInfoDisplay.Update(deviceInfoUpdate);
                        break;
                    }
                }
            });
        }
```

7. Add DeviceRemoved event

```
private async void OnDeviceRemoved(DeviceWatcher watcher, DeviceInformationUpdate deviceInfoRemoved)
        {
            // Since we have the collection databound to a UI element, we need to update the collection on the UI 
thread.
            await Dispatcher.RunAsync(CoreDispatcherPriority.Low, () =>
            {
                // Find the corresponding DeviceInformation in the collection and remove it
                foreach (DeviceInformationDisplay deviceInfoDisplay in ResultCollection)
                {
                    if (deviceInfoDisplay.Id == deviceInfoRemoved.Id)
                    {
                        CheckPairing(deviceInfoDisplay);
                        ResultCollection.Remove(deviceInfoDisplay);
                        break;
                    }
                }

                Debug.WriteLine("{0} devices found.", ResultCollection.Count);
            });
        }
```

8. Add EnumComplete Event

```
        private void OnEnumerationCompleted(DeviceWatcher watcher, object args)
        {
            Debug.WriteLine("{0} devices found. Enumeration completed. Watching for updates...",
ResultCollection.Count);
            _restartWatcher = true;
            _deviceWatcher.Stop();
        }
```

9. Add Stopped Event

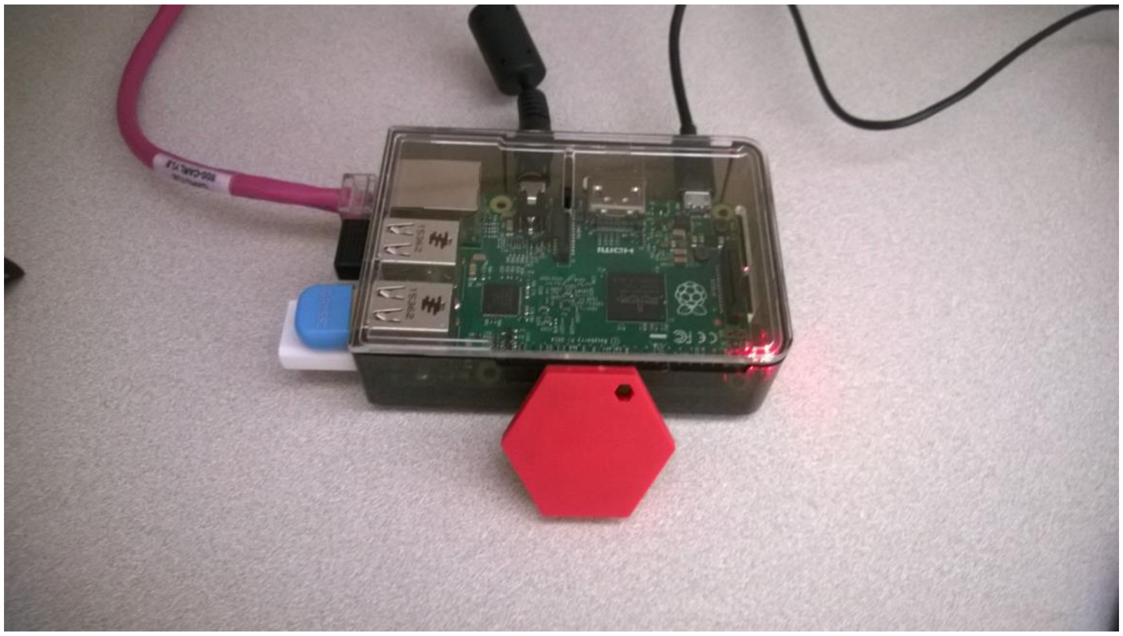
```
        private async void OnStopped(DeviceWatcher watcher, object args)
        {
            Debug.WriteLine("{0} devices found. Watcher {1}.",
                ResultCollection.Count,
                DeviceWatcherStatus.Aborted == watcher.Status ? "aborted" : "stopped");

            if (_restartWatcher)
            {
                await Dispatcher.RunAsync(CoreDispatcherPriority.Low, () =>
                {
                    _restartWatcher = false;
                    _deviceWatcher.Start();
                });
            }
        }
```

10. Add Pairing Event

```
        private async void OnPairingRequested(DeviceInformationCustomPairing sender, DevicePairingRequestedEventArgs
args)
        {
            switch (args.PairingKind)
            {
                case DevicePairingKinds.ConfirmOnly:
                    // Windows itself will pop the confirmation dialog as part of "consent" if this is running on
Desktop or Mobile
                    // If this is an App for 'Windows IoT Core' where there is no Windows Consent UX, you may want to
provide your own confirmation.
                    args.Accept();
                    break;
            }
        }
```

D) Understanding The Code

You should now be able to build and deploy your code. Once it is up and running, the code will start a watcher and look for BLE beacons. If it finds one that starts with the name “XY” and is close enough to the receiver, then it will automatically pair the beacon. When the beacon is successfully paired, the speech engine will say the name of the beacon. Use your imagination to think of other ways this could behavior could be utilized.



http://www.skoonstudios.com

Last Revised: April 4, 2016