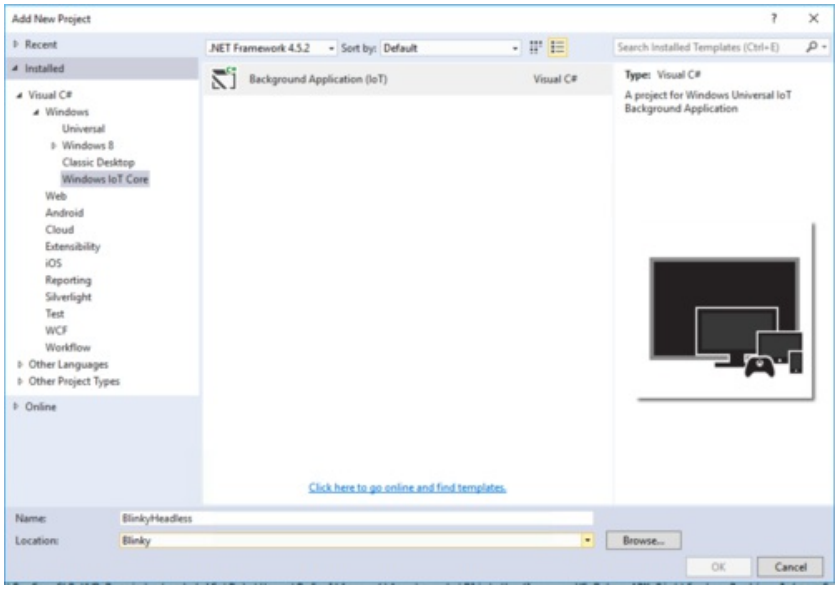
Windows IoT Core Application Development: Headless Blinky

Created by Rick Lesniak



Last updated on 2016-08-18 05:22:16 PM UTC

Guide Contents

Guide Contents 2
Overview 3
- Headed and Headless 3
- C-Sharp (C#) 3

Headless Blinky 5
- Step 1: Create Solution 5
- Step 2: Add IoT framework 7

Program Architecture 10
- Timer thread 10
- Task deferral 11

Pin I/O 12
- Pi Setup 13

BlinkyHeadless Code 14
Running BlinkyHeadless 16
- Remote Machine 16
- Run 18

Debugging 20
- Next steps 21

FAQ 22

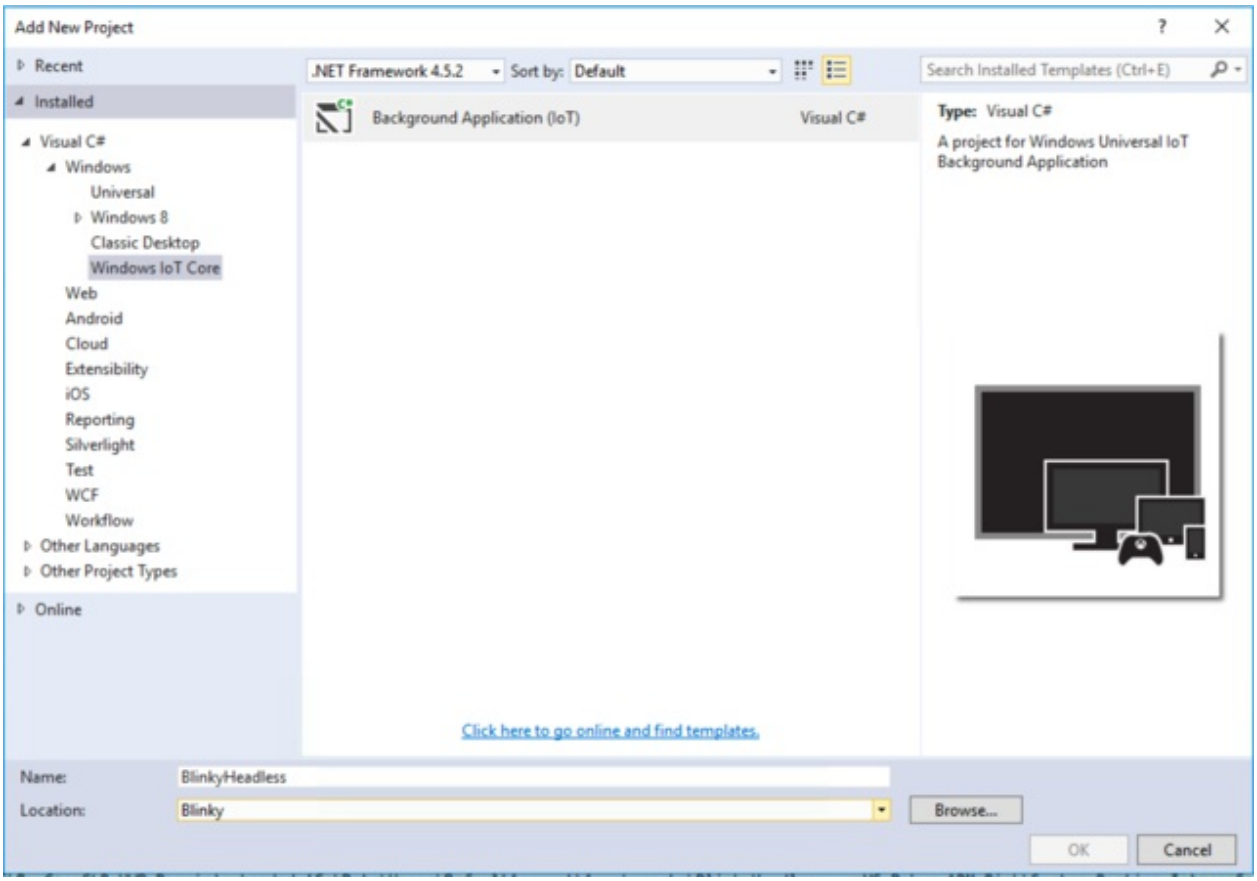
Overview

Headed and Headless

There are two basic types of Windows IoT Core applications: “headed” and “headless”.

“Headless” apps have no graphical user interface (GUI), and are said to run “in the background”. That is, they continue to run even when you start up a “headed” application. Also, they don’t depend on having a monitor attached to the system.

“Headed” applications have a GUI, so you need a display attached to your system in order to use them.



C-Sharp (C#)

Using Visual Studio, you can program your Raspberry Pi in C# or Visual Basic (or even C++). Microsoft has unified the API (Application Programmability Interface) between C# and Visual Basic, so it’s easy to switch between them. We’ll be using C# for this tutorial (just because we

like it better, and hey, it's our tutorial anyway!).

Instructions on the C# and Visual Basic programming languages are wayyy beyond the scope of this tutorial, so we'll assume you already know the basics.

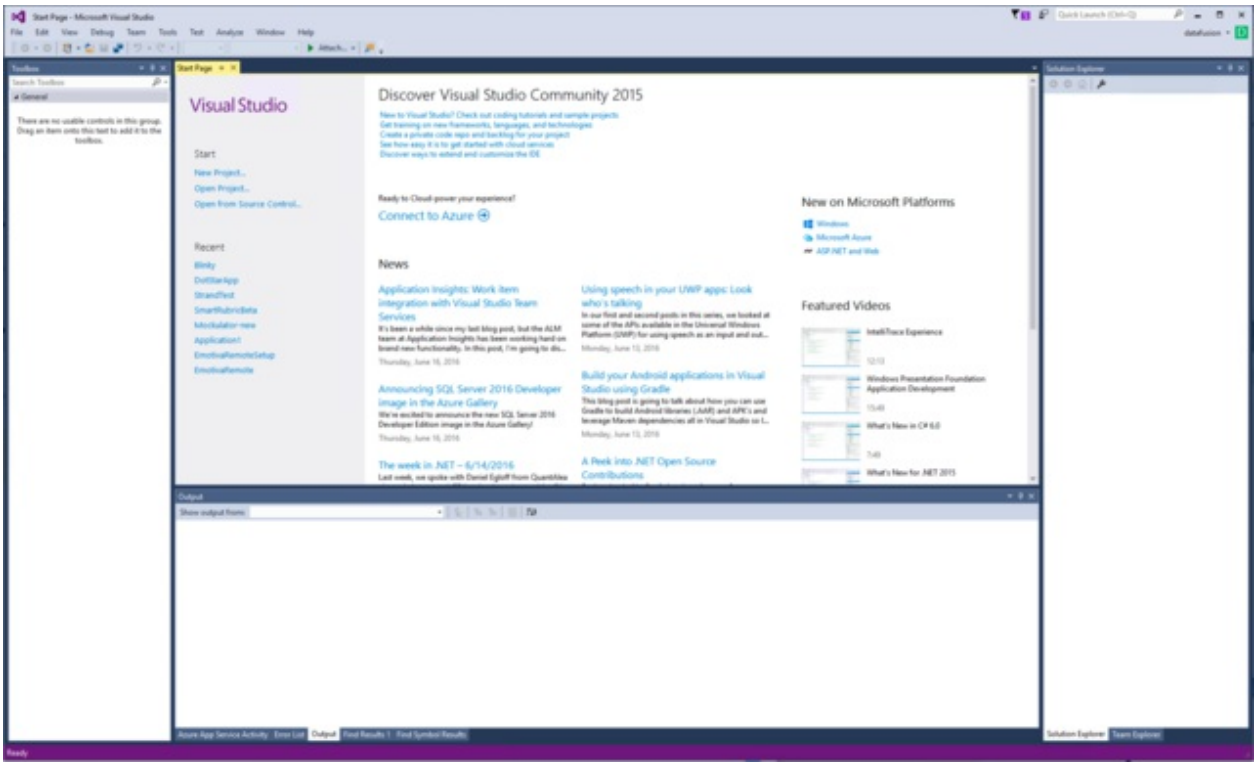
Headless Blinky

Microsoft provides the headless Blinky app as a preconfigured solution (Visual Studio is based on **solutions**. Each **solution** may contain multiple **projects**. You can compile and run each project separately). The Blinky solution contains a single Blinky project.

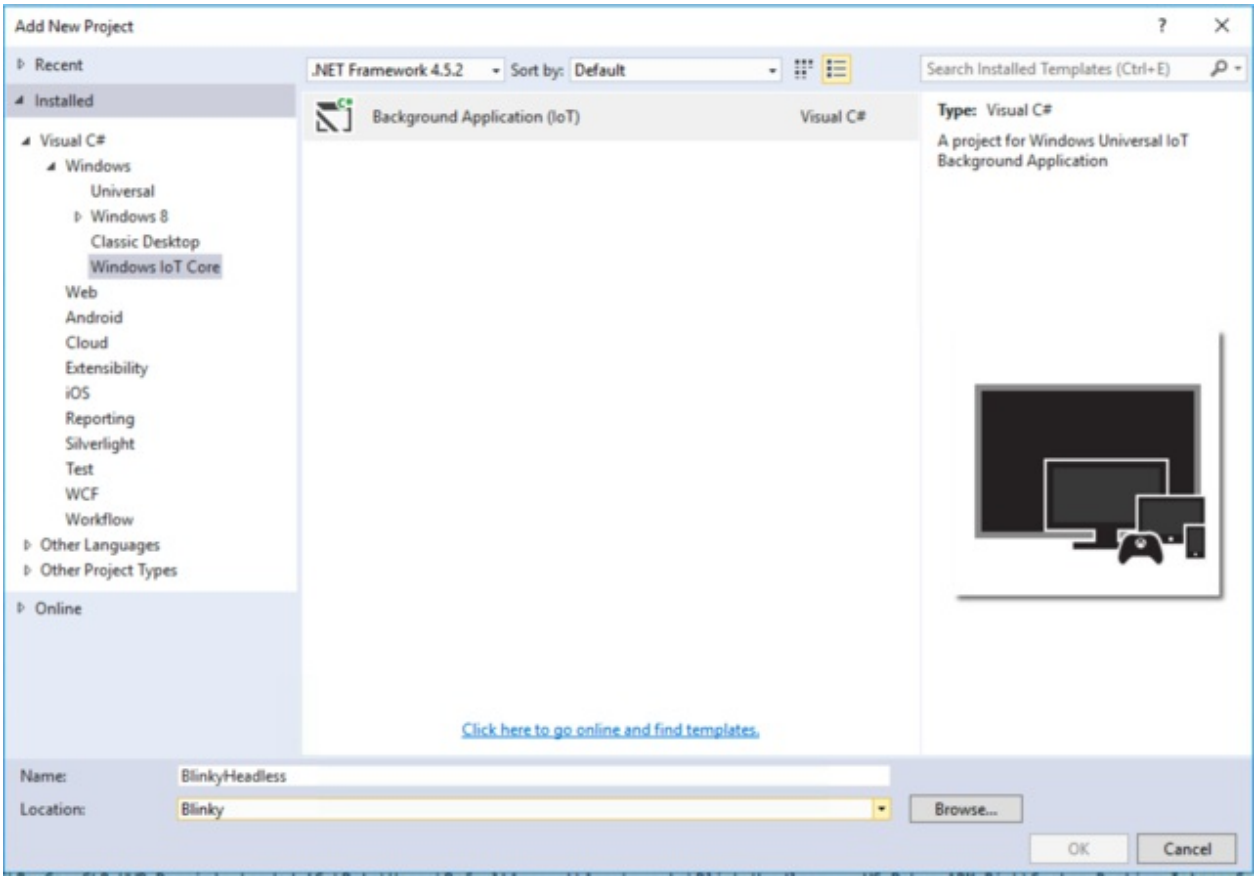
Rather than simply downloading Blinky and running it, we thought it would be more useful to go through the entire process of creating the Blinky solution.

Note: We'll be changing the wiring of the LED from the Microsoft version of Blinky, but otherwise it's all the same.

Step 1: Create Solution

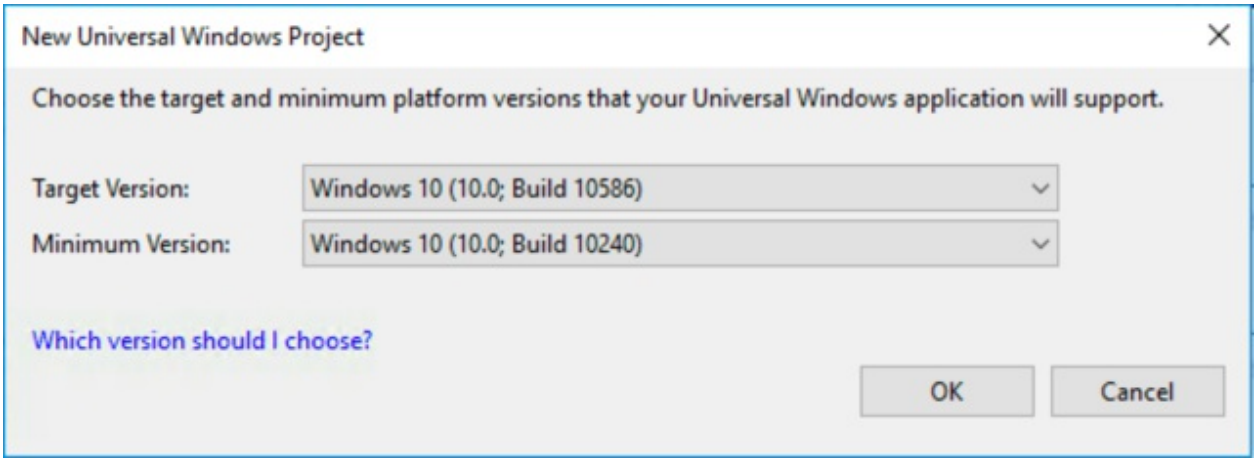


- Start a new instance of **Visual Studio 2015**
- The Start Page will appear. Click on **New Project…**
- This will bring up the New Project window



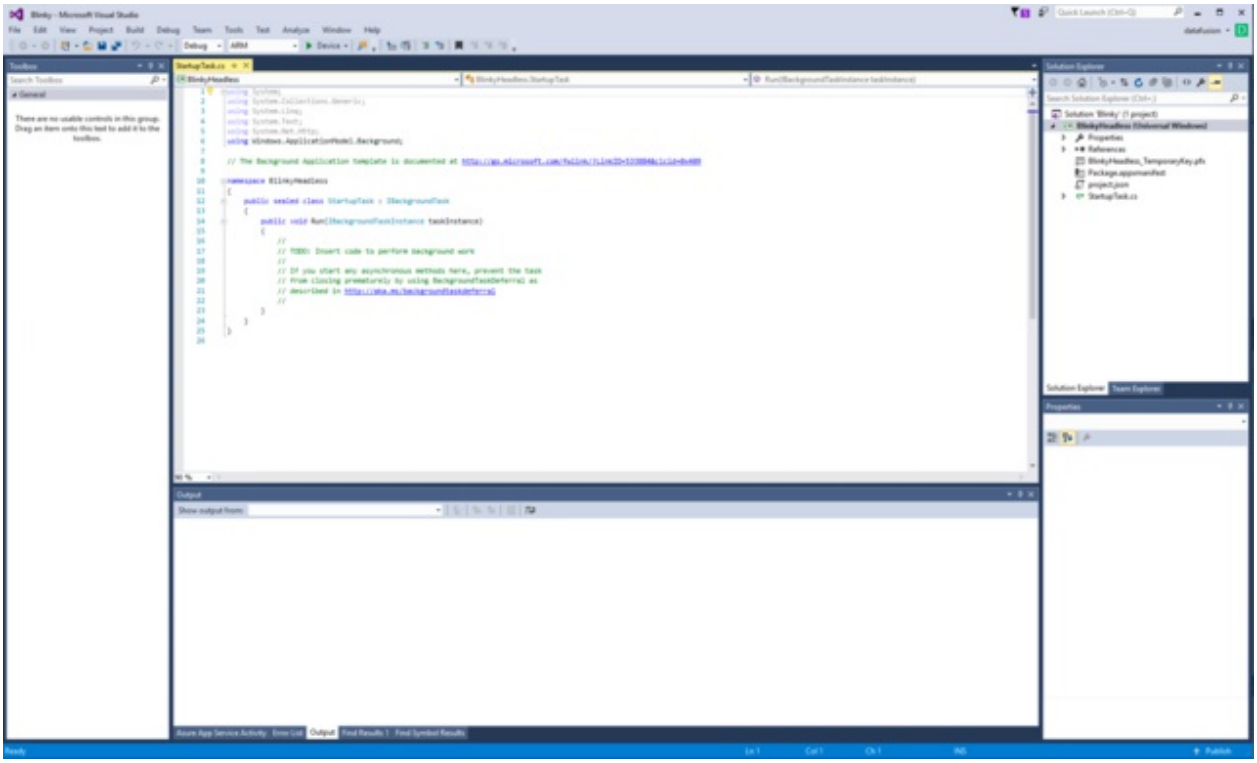
- In the left-hand panel of the New Project window, navigate to**Templates > Visual C# > Windows > Windows IoT Core**
- Select **Background Application (IoT)**
- Give the project a name (“BlinkyHeadless”).
- Enter a location for the **solution**/**project** folder. The default location is fine. Make sure the box labeled **Create directory for solution** is checked.
- Give the solution a name (“Blinky”).
- Click **OK**

A new dialog window will open, asking you to choose the target and minimum versions your Blinky app will support. Choose the target and Minimum versions to match the version of Windows IoT you have running on your Raspberry Pi. If you’re not sure, check the main HDMI display or Device Portal to check the installed version.



Click **OK** and the **Blinky Solution** will open.

In the center of the window, you'll see the code-editing panel. In the upper right is the **Solution Explorer** panel:

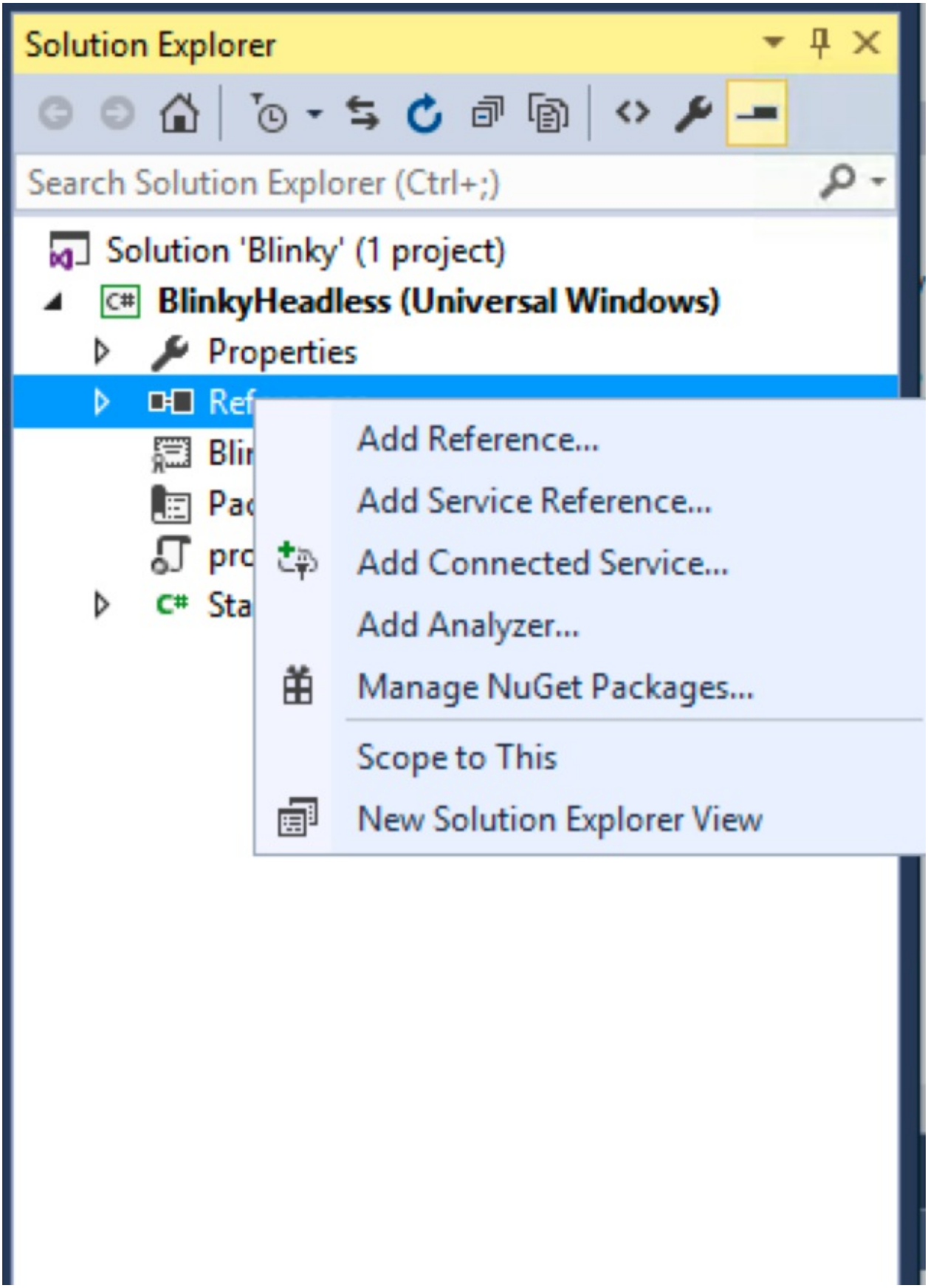


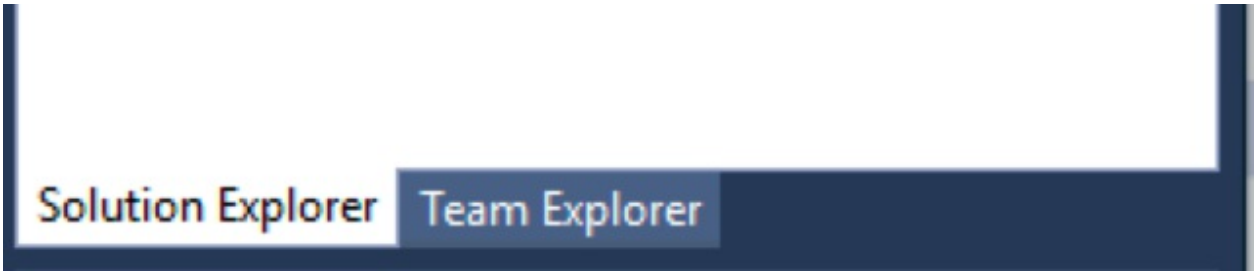
The **Solution Explorer** shows that the Blinky solution contains the BlinkyHeadless project. The code window is open to the StartupTask.cs program file. Except for the **References** item, you can ignore the other files listed in the **Solution Explorer** for now.

Step 2: Add IoT framework

Even though we used a Windows IoT Core template for creating our solution, we still need to add the IoT Framework to our BlinkyHeadless project. In the **Solution Explorer**, right click on

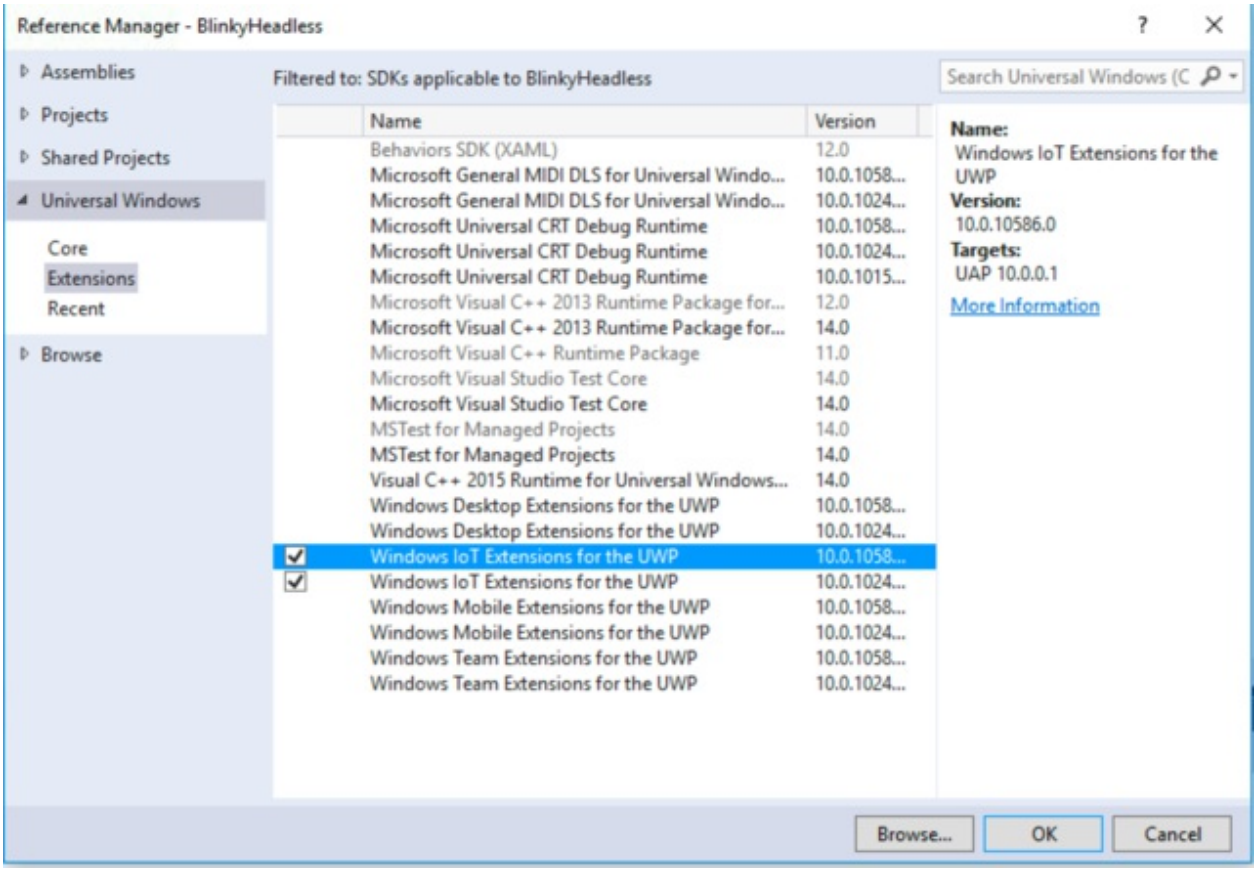
the line that says **References**, and select **Add Reference…**





This will bring up the **References Manager**. In the **References Manager Window**, navigate to **Universal Windows > Extensions**. In the list, find **Windows IoT Extensions for the UWP**. There may be more than one item with this name. To the right of each item, you'll see a version number. Select the items that match the **Target** and **Minimum** versions you entered earlier.

When you select an item, a checkbox will appear to the left of the item. Check the box for each item you select.



Click **OK** to add the references to your project.

Program Architecture

We said we weren't going to turn this into a C# tutorial, but a few words about the program architecture are appropriate.

Windows IoT Core applications are oriented around *tasks*. If you're mostly familiar with Arduino or python programming, you probably aren't too familiar with tasks. Basically, a task is a program than can run at the same time as other programs on the same computer. Arduino only allows one program at a time. Linux supports tasks, but you may not have done any task-oriented programming.

We're not going to go into the intricacies of tasks here, but it's worth noting that the fundamental structure of our BlinkyHeadless application is a task. You can see this from the type of the **StartupTask** class – it inherits from the **IBackgroundTask** interface class. To execute BlinkyHeadless, Windows IoT creates a new task object and uses it to call the **Run** function.

The **Run** function is where we'll put our code.

If you're curious, you can read more about background tasks here:

Background Tasks: More Info
http://adafru.it/psA
For the purposes of this tutorial, we're going to use screen captures to illustrate the code components of the application. The full code will be available at the end of the tutorial.

Timer thread

We're not quite done with tasks. Tasks may also be referred to as **threads**, particularly when they exist *within* a task. One of the standard thread objects in Windows is the **Timer**. We will be using a timer to control the blinking of our LED.

If you are familiar with Arduino, you will be accustomed to using `delay()` statements to control things like blinking LEDs. The drawback of this approach is that the Arduino can't do anything else while it is waiting for the delay to expire.

In Windows, instead of using a delay, we can start a timer thread. Once the thread is started, we can go about doing other useful things, and the timer will alert us when the period has expired. The timer does this through a `Timer Tick` event. All we need to do is to create an event function and associate it with the timer. When it's time to turn the LED on or off, windows will automatically call our tick event function.

```
13  public sealed class StartupTask : IBackgroundTask
14  {
15      public void Run(IBackgroundTaskInstance taskInstance)
16      {
17          ThreadPoolTimer timer = ThreadPoolTimer.CreatePeriodicTimer(Timer_Tick, TimeSpan.FromMilliseconds(500));
18      }
19
20      private void Timer_Tick(ThreadPoolTimer timer)
21      {
22
23      }
24  }
```

There's one other thing we need to do. You may have noticed a series of *using* statements at the top of StartupTask.cs. These are similar to the C *#include* or python *import* statements. They identify libraries we want to refer to in our program. We need to add the threading library to the list:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Net.Http;
6  using Windows.ApplicationModel.Background;
7  using Windows.System.Threading;
```

Don't worry about the *using* statements that are grayed out – it just means that there are no references to those libraries in the code yet.

Task deferral

There is one other thing about tasking in BlinkyHeadless, and this is subtle. When the **Run** function exits, so does our main task. That is, our BlinkyHeadless application terminates. We don't want this to happen.

In BlinkyHeadless, we really don't have anything else to do while we're waiting for the timer tick event. We could add an infinite loop, but that will consume processor resources. Windows provides a better way to do the same thing, and that is a **Task Deferral**.

When we set up a deferral on a background task, such as our main task, the task won't exit until we explicitly tell it that we're done, even if it runs out of other things to do.

You can read more about background task deferral here:

Task Deferral: more info
http://adafru.it/psB

Pin I/O

Digital I/O pin operations are quite simple in Windows 10 IoT Core. Windows provides a GPIO controller object. With this object, we can select a pin, set it for input or output, and read or write the logic state of the pin.

To access a pin, we first need an instance of the *GpioController* object. As the name suggests, the GpioController is where we go to set up GPIO pins. Once we have a reference to the GpioController, we can get a reference to a pin.

The pin reference allows us to set the pin mode to input or output, and to read or write the state of the pin.

Before we can do this, we need to add another *using* statement, to say that we'll be using the Windows.Devices.Gpio library.

Now we can add a little function to set up the pin 5 as an output to blink the LED.

```
30          private void InitGPIO()
31          {
32              GpioController gpio = GpioController.GetDefault();
33              GpioPin pin = gpio.OpenPin(LED_PIN);
34              pin.Write(GpioPinValue.High);
35              pin.SetDriveMode(GpioPinDriveMode.Output);
36          }
```

After we get the pin reference, we write the pin value to `HIGH`, and then set the mode to `output`.

The only thing left to do in BlinkyHeadless is to actually toggle the LED in `Timer_Tick`. We'll keep track of the current state of the pin in a variable, and use that to decide whether to turn the pin to `HIGH` or `LOW`:

```
29          private void Timer_Tick(ThreadPoolTimer timer)
30          {
31              if (GpioPinValue.High == pinValue)
32                  pinValue = GpioPinValue.Low;
33              else
34                  pinValue = GpioPinValue.High;
35
36              pin.Write(pinValue);
37          }
```

Pi Setup

Of course, if we want to blink a LED, we need to attach a LED to the Pi. Use something like a red LED. Connect the long leg of the LED to **GPIO 5** on the Pi. Connect a 180 Ohm resistor to the short leg of the LED, and then connect the other end of the resistor to **Ground** on the Pi.

Connected this way, the LED will light when we set the GPIO pin HIGH, and it will turn off when we set the GPIO pin LOW.

(Note: This is different from the way Microsoft has the LED arranged in their version of Blinky. Microsoft Blinky asks you to wire the LED between 3.3V and the GPIO pin. Wired that way, to turn the LED *on*, you have to set the GPIO pin to *low*. To turn the LED *off*, you have to set the GPIO pin to *high*. There are reasons for doing it this way, but it's kind of logically backwards).

BlinkyHeadless Code

The complete StartupTask.cs program is shown here. In some of the earlier examples, we made some variables local, just to make it clear that they had to be of a particular type. In reality, we want those variables to be global. The final program reflects that, with LED_PIN, pin, pinValue, and timer declared as private global variables.

There are other code modules in the BlinkyHeadless project, but those are all automatically generated. All we need concern ourselves with is the module StartupTask.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net.Http;
using Windows.ApplicationModel.Background;
using Windows.System.Threading;
using Windows.Devices.Gpio;

// The Background Application template is documented at http://go.microsoft.com/fwlink/?LinkID=533884&clcid=0x409

namespace BlinkyHeadless
{
    public sealed class StartupTask : IBackgroundTask
    {
        BackgroundTaskDeferral _deferral;
        private const int LED_PIN = 5;
        private GpioPin pin;
        private GpioPinValue pinValue;
        private ThreadPoolTimer timer;

        public void Run(IBackgroundTaskInstance taskInstance)
        {
            _deferral = taskInstance.GetDeferral();

    InitGPIO();

            timer = ThreadPoolTimer.CreatePeriodicTimer(Timer_Tick,
     TimeSpan.FromMilliseconds(500));
        }

        private void Timer_Tick(ThreadPoolTimer timer)
        {
            if (GpioPinValue.High == pinValue)
                pinValue = GpioPinValue.Low;
            else
                pinValue = GpioPinValue.High;

            pin.Write(pinValue);
```

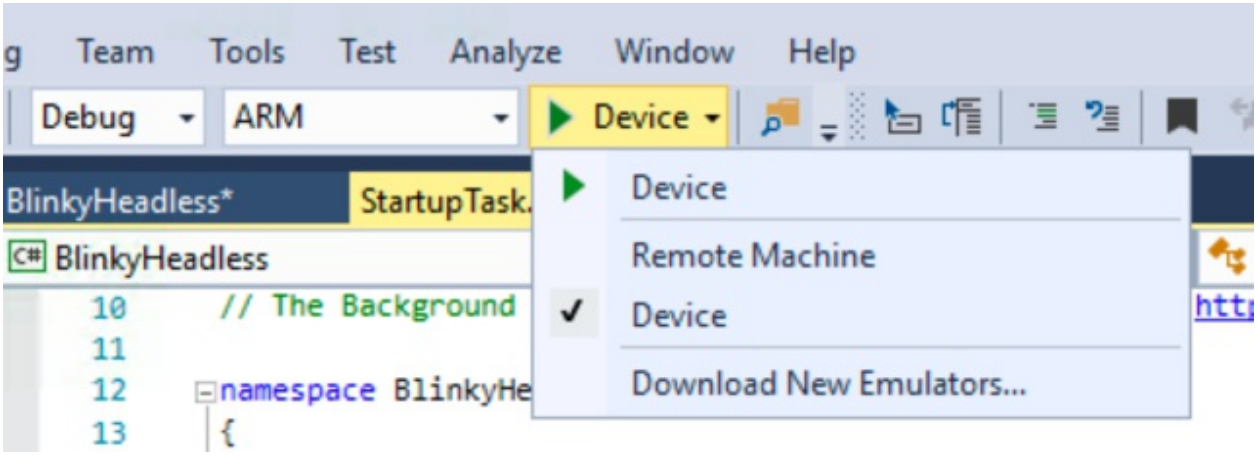
```
        }

        private void InitGPIO()
        {
            GpioController gpio = GpioController.GetDefault();
            pin = gpio.OpenPin(LED_PIN);
            pinValue = GpioPinValue.High;
            pin.Write(pinValue);
            pin.SetDriveMode(GpioPinDriveMode.Output);
        }
    }
}
```

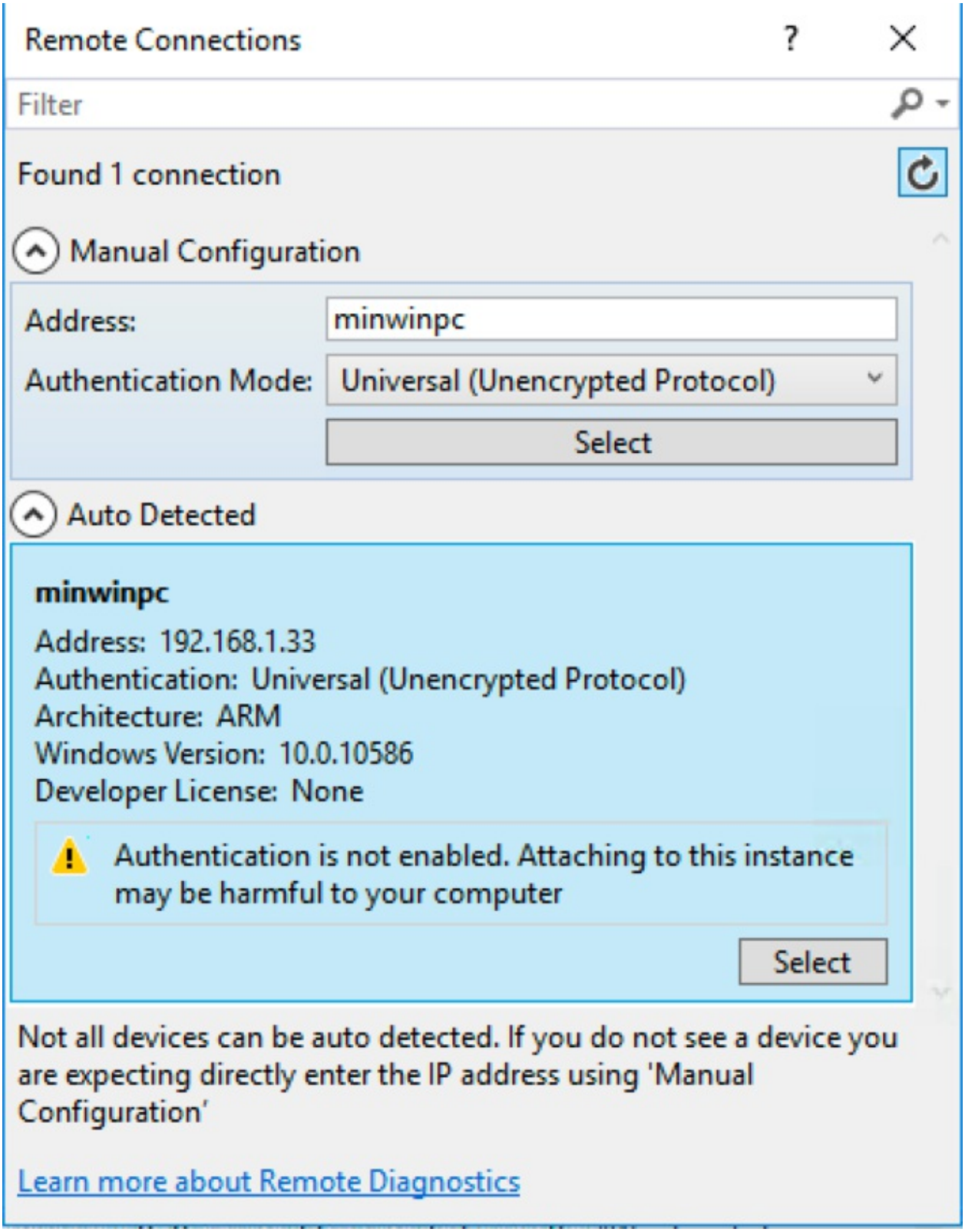
Running BlinkyHeadless

Remote Machine

From the Visual Studio standpoint, the Raspberry Pi is a**Remote Machine.** So, we have to make an association between Visual Studio and the remote machine. In the Visual Studio Toolbar, there should be a button labeled **Device**. On the right-hand side of this button is a little down-arrow. Click the down-arrow and a drop-down will open.



Select **Remote Machine**, and another pop-up window will open. This is the place where you find your Raspberry Pi and make the remote-machine association with Visual Studio. The Raspberry Pi should be automatically found. Click on the arrow next to **Auto Detected** to open up the panel, and click **Select**.



If you have to do a manual configuration for some reason, enter the name of the Raspberry Pi, and select **Universal** for the **Authentication Mode**.

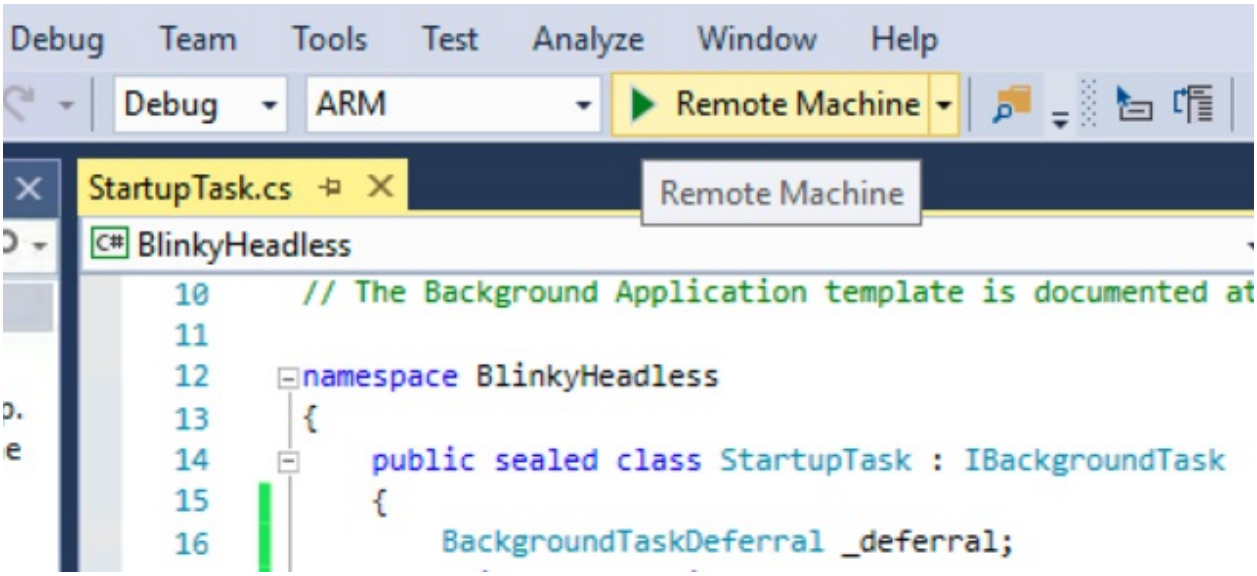
If you have a problem here, it's because your Raspberry Pi isn't visible on the network. Visual

Studio can't see it. Go back and make sure your Pi is configured correctly and is visible to **Device Portal**.

You can also get to this window by going through the **Solution Explorer**. Find the **Properties** entry in your project and double click on it. The **Project Properties** window will open. On the left of the panel, select **Debug**. In **Start Options**, select **Remote Machine**, and then click the **Find…** button.

Run

To run your app, press the large green arrow in the toolbar in the box labeled **Remote Machine**

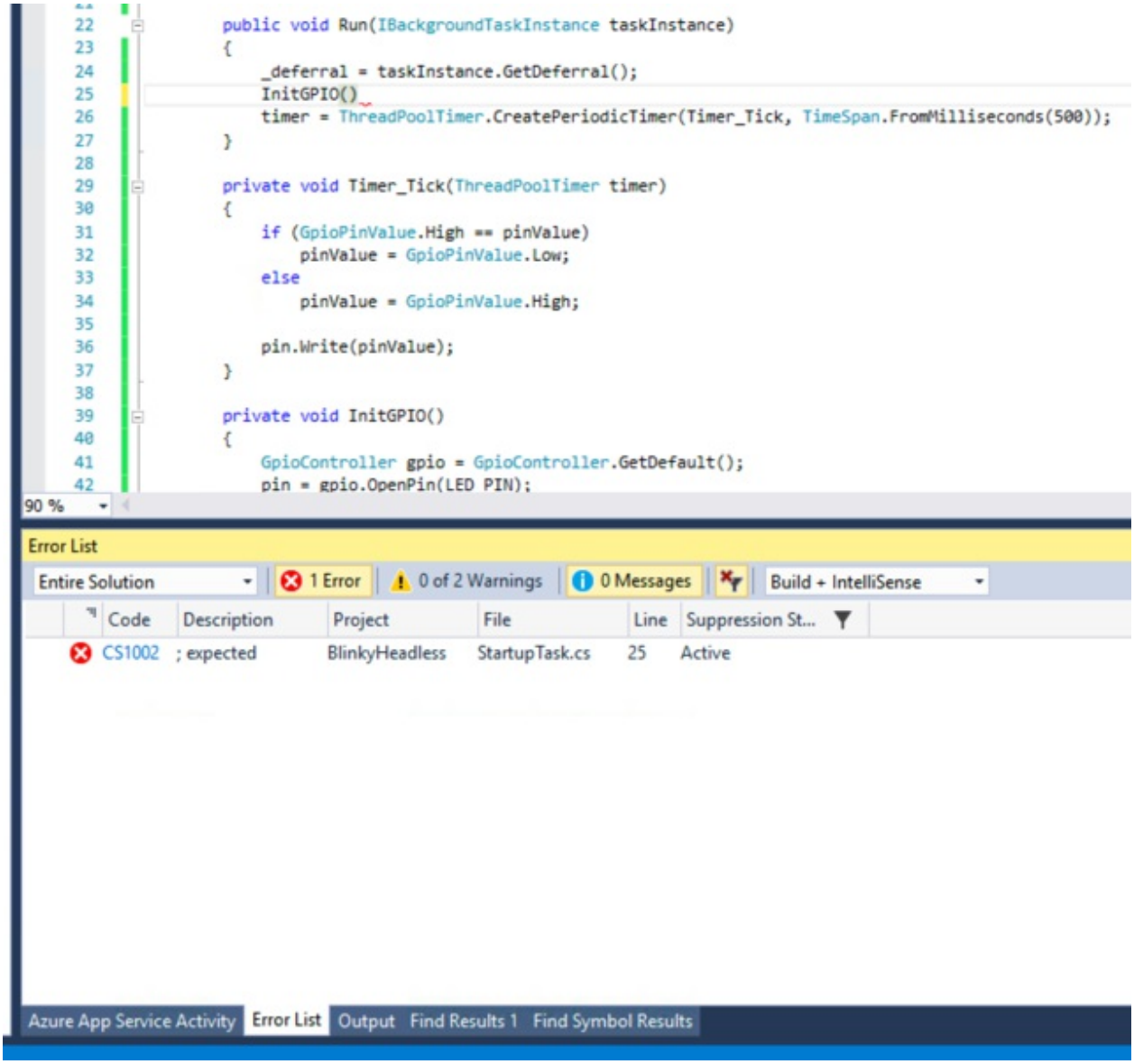


The app will be compiled and downloaded to the Raspberry pi, and it should begin to run. The first time you run your app, this process can take a couple of minutes. Subsequently, your app will download and run much quicker.

Click on the **green arrow** – if all goes well, the LED attached to GPIO 5 will begin to blink!

If your code has syntax errors, then the error list will appear in the panel below the main code panel. Click on the tab titled **Error List** to see the error reports. You'll also see a red underline in the code at the point where the error was detected.

In this example, we forgot to put a semicolon at the end of line 25:



if you reboot your Raspberry Pi, Visual Studio may have trouble re-connecting to the Pi. You may see strange "deployment" errors. If this happens, exit and restart Visual Studio after the Pi has finished rebooting.

Debugging

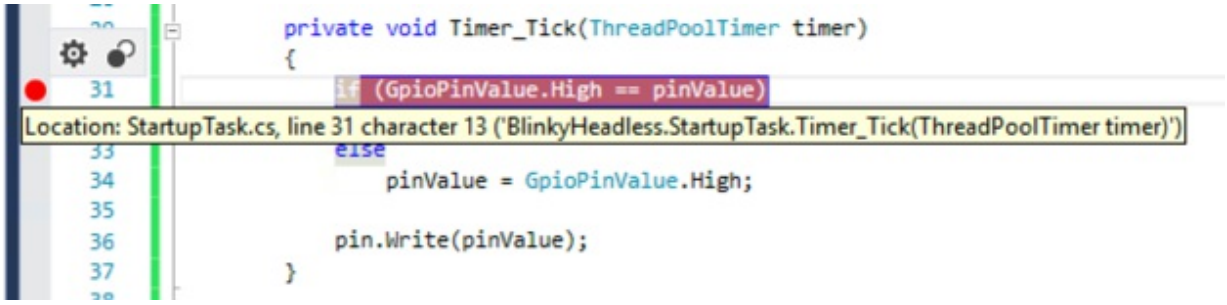
Debugging is a large topic. We won't go into in much detail here.

If you are accustomed to Arduino programming, then you have likely used Serial.print statements to add a debugging trace to your sketches. Visual Studio and IoT Core applications do things differently. Instead of print statements, you set *breakpoints* in your code.

With breakpoints, you tell Visual Studio where to pause in executing your code. Once the code pauses, you can use Visual Studio to look at the values of variables. You can also single-step through your code line-byline, so that you can see the order of operations.

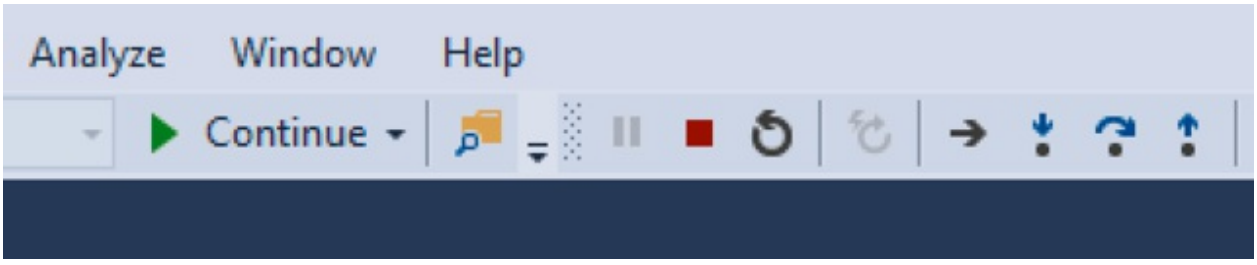
This form of debugging is much quicker and much more effective than using print statements.

Set a breakpoint by clicking in the gray bar to the left of the line you want to pause on:



Here, we've set a breakpoint on the *if* statement in `Timer_Tick`. Visual Studio will pause the application just before we execute this line. We can then do things like look at the value of the variable `pinValue`, simply by hovering the cursor over the word `pinValue`.

We can also single step, using the debugger toolbar:



The stepping icons are to the right of the picture.

- The down arrow over the dot is **step into**. If there is a function call on the line, clicking this button will step into the function and pause again on the first line of the function.
- The curved arrow over the dot is **step over** this will step to the next line in the code, but won't step into any functions called on the line.

- The up arrow over the dot is **step out**. It causes the code to proceed until the current function is exited. It will then pause again on the first line after the function call.
- Other buttons to note are the green **Continue** button, which resumes normal execution from the breakpoint . The red square will immediately terminate your app. The circular arrow will restart your app from the beginning.

Next steps

That's pretty much it for our headless application. If all went well, your LED should be merrily blnking away. If not, help is always available in the forums!

Next, we'll go through the *Headed* version of Blinky

Headed Blinky
http://adafru.it/psC

FAQ

I get a deployment error from Visual Studio: DEP0001 : Unexpected Error: -2145615869

This error typically happens after you reboot your Pi. For whatever reason, Visual Studio is not longer able to communicate with it. The solution is to simply close and restart Visual Studio. Do this every time you reboot your Pi.