APPENDIX F

Setting up Visual Studio 2017 for IoT development

While I was writing this book, Visual Studio 2017 was released. It has a different installer than Visual Studio 2015, so here I show you how you can set up Visual Studio 2017 and use it to implement the sample apps described throughout this book. Additionally, I describe how to set up and deploy C# apps to the IoT device and how to configure the Portable Class Library using Visual Studio 2017.

Installation

Figure F-1 shows the new installer of Visual Studio 2017.

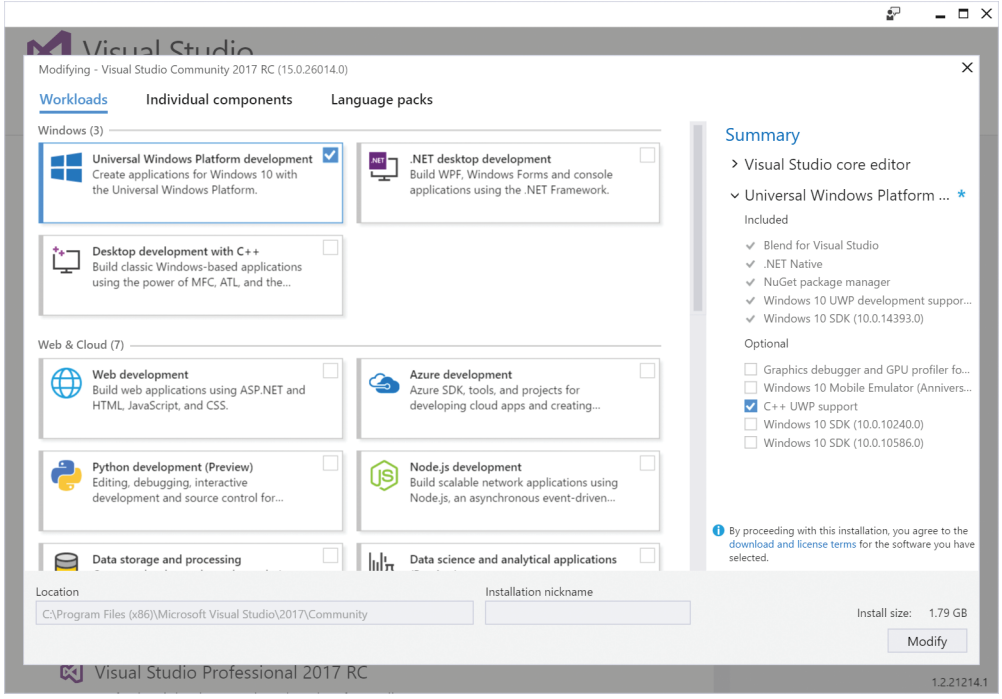

FIGURE F-1 Visual Studio 2017 installer.

For IoT development in this book, you will need the Universal Windows Platform development tools (see Figure F-2) and the C++ UWP support shown in Figure F-3.

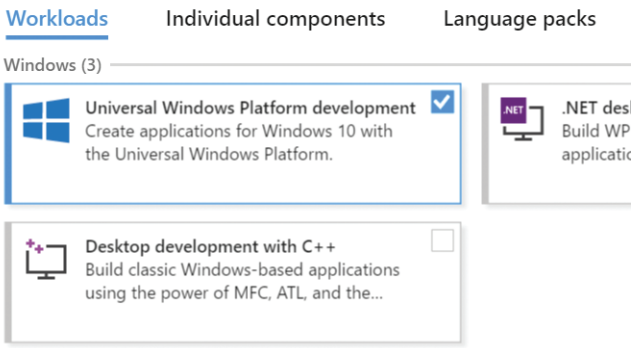

FIGURE F-2 Universal Windows Platform development workload.

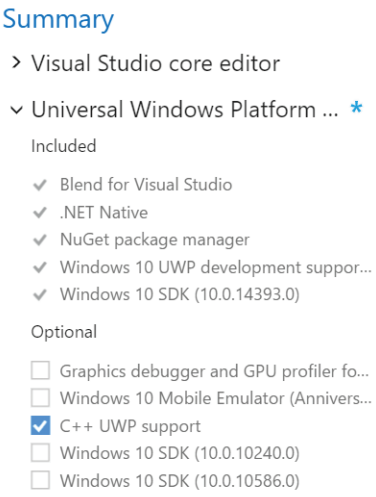

FIGURE F-3 Installing the C++ UWP support package.

Optionally, you can install Windows 10 SDK version 10586. By default, Visual Studio 2017 comes with Windows 10 Anniversary Edition SDK (14393).

To install Windows IoT Core project templates for Visual Studio 2017, open the **Tools/Extensions and Updates** menu in Visual Studio 2017, click the **Online** tab, and type **IoT** in the search box. After a moment, a Windows IoT Core Project Templates for Visual Studio 2017 option is displayed. (See Figure F-4.) Click the **Download** button and wait for the installer to do its job. You will need to close Visual Studio 2017 during the installation.

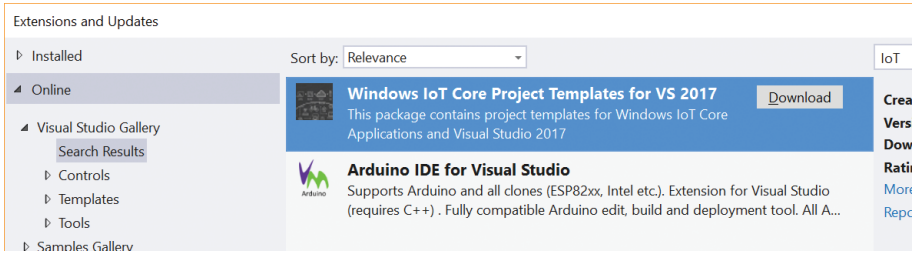

FIGURE F-4 Windows IoT Core Project Templates for Visual Studio 2017.

Visual C# project template

When all the tools are ready, you can write the UWP app by using the Visual C# project template and then deploy it to your IoT device.

Creating a project

You create the project similarly to how you did in Visual Studio 2015, using the New Project dialog box (open the **File** menu and choose **New Project**). Type **Blank App** in the search box, select the **Blank App (Universal Windows) Visual C#** template, and set the project and solution names. (see Figure F-5.)

Note Unlike Visual Studio 2015, the New Project dialog box contains the Open Visual Studio Installer hyperlink. You can use it to open the installer window, where you can add features and templates you cannot find in the New Project dialog box.

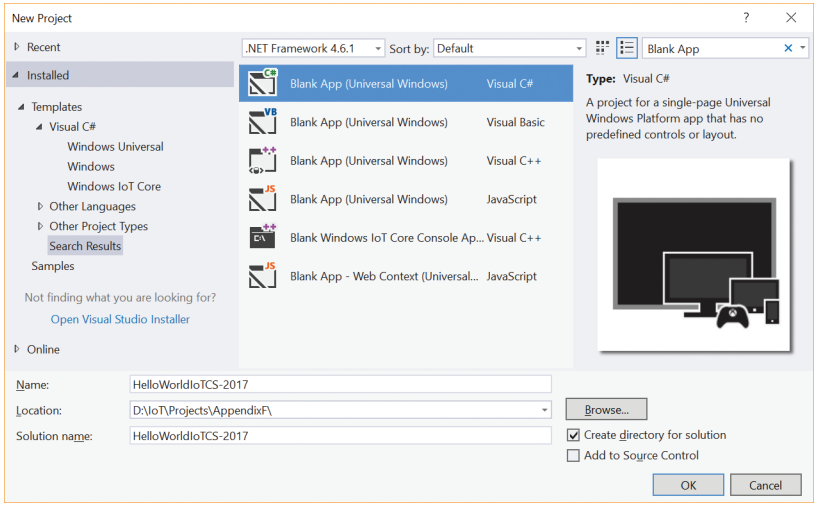

FIGURE F-5 New Project dialog box of Visual Studio 2017.

Click the **OK** button. You'll be given a chance to choose target and minimum platform versions. As shown in Figure F-6, this dialog box looks exactly the same as in Visual Studio 2015 Update 3. Keep the default values: **Windows 10 Anniversary Edition (10.0; Build 14393)** for the Target Version setting and **Windows 10 (10.0; Build 10586)** for the Minimum Version setting.

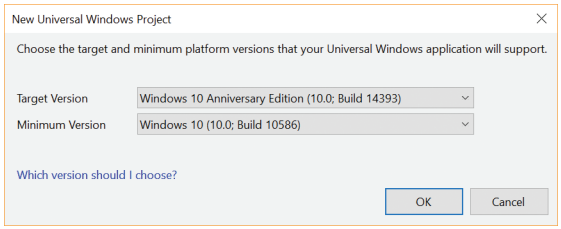

FIGURE F-6 Selecting target and minimum platform versions.

IoT extensions

To access the IoT-specific API of the UWP, reference Windows IoT Extensions for the UWP. This proceeds similarly as in Visual Studio 2015. To open the Reference Manager dialog box, right-click the **References** node in the Solution Explorer and choose **Add Reference**. Next, click the **Universal Windows/Extensions** tab and select **Windows IoT Extensions for the UWP (10.0.14393.0)**. (See Figure F-7.)

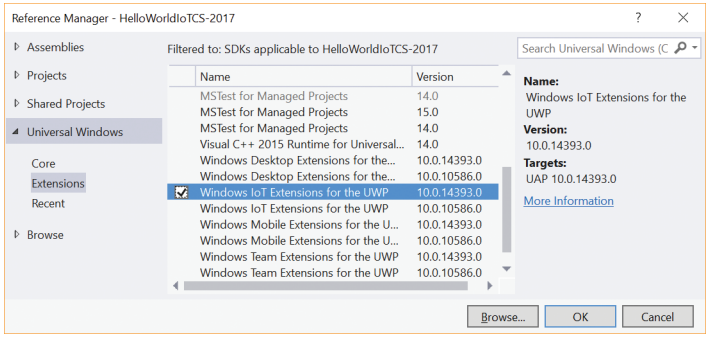

FIGURE F-7 Reference Manager in Visual Studio 2017.

Implementation

I implement the HelloWorldIoTCS-2017 app by first creating the `LedBlinking` class (see the companion code in Appendix F/HelloWorldIoTCS-2017/GpioControl/LedBlinking.cs). It is used to start and stop the background operation, which inverts the selected GPIO pin at specified time intervals. I use this functionality to control the onboard green LED of the RPi2. To control the external LED when using RPi3, you need to use an appropriate GPIO pin number.

`LedBlinking` internally uses methods described in Chapter 2, "Universal Windows Platform on devices," and Chapter 3, "Windows IoT programming essentials." The main element of the `LedBlinking` class is the `InitializeBlinkingTask` method from Listing F-1. It periodically inverts the GPIO pin state to control an LED. To start and stop this blinking, I implement the `Start` and `Stop` methods of the `LedBlinking` class. (See Listing F-2.)

LISTING F-1 Initializing blinking background operation

```
public int MsShineDuration { get; private set; }

private Task blinkingTask;
private CancellationTokenSource blinkingCancellationTokenSource;

private GpioPin gpioPin;

private void InitializeBlinkingTask()
{
    blinkingCancellationTokenSource = new CancellationTokenSource();

    blinkingTask = new Task(() =>
    {
        while (!blinkingCancellationTokenSource.IsCancellationRequested)
        {
            if (IsActive)
            {
                SwitchGpioPin(gpioPin);

                Task.Delay(MsShineDuration).Wait();
            }
        }
    }, blinkingCancellationTokenSource.Token);
}
```

LISTING F-2 Starting and stopping blinking background operation

```
public bool IsActive { get; private set; } = false;

public void Start()
{
    if (!IsActive)
    {
        InitializeBlinkingTask();

        blinkingTask.Start();

        IsActive = true;
    }
}

public void Stop()
{
    if (IsActive)
    {
        blinkingCancellationTokenSource.Cancel();

        IsActive = false;
    }
}
```

Given the `LedBlinking` class, I start to work on the UI. Here, I define a simple UI consisting of three buttons: Initialize, Start Blinking, and Stop Blinking. (See Figure F-8.)

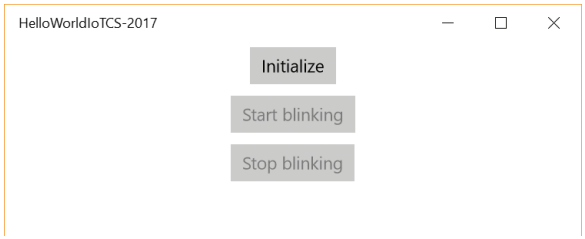

FIGURE F-8 User interface of the HelloWorldIoTCS-2017 app.

The first button, Initialize, is used to instantiate the `LedBlinking` class (see Listing F-3). Note that this event handler uses the `GpioPinNumber` and `MsShineDuration` properties of `BlinkingViewModel`. You can use them to change the GPIO pin number and frequency of LED blinking (GPIO pin switching time intervals). Default values of these properties appear in Listing F-4.

LISTING F-3 Instantiating the LedBlinking class

```
private BlinkingViewModel blinkingViewModel = new BlinkingViewModel();
private LedBlinking ledBlinking;

private void ButtonInitializeGpio_Click(object sender, RoutedEventArgs e)
{
    try
    {
        ledBlinking = new LedBlinking(blinkingViewModel.GpioPinNumber,
            blinkingViewModel.MsShineDuration);
        blinkingViewModel.IsGpioPinAvailable = true;
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex.Message);
    }
}
```

LISTING F-4 The GPIO pin number and LED blinking frequency are configured through appropriate properties of BlinkingViewModel

```
public int GpioPinNumber { get; set; } = 47;
public int MsShineDuration { get; set; } = 100;
```

After successful initialization, I disable the button through a `IsGpioPinAvailable` property of the `BlinkingViewModel` class (see the companion code in Appendix F/HelloWorldIoTCS-2017/ViewModels/BlinkingViewModel.cs) and `LogicalNegationConverter` (see the companion code in Appendix F/HelloWorldIoTCS-2017/Converters/LogicalNegationConverter.cs). I use this converter because the

Initialize button should be enabled only when IsGpioPinAvailable is false. Hence, LogicalNegationConverter negates Boolean values. To use this converter in the entire app, I modify the App.xaml file as indicated in Listing F-5.

LISTING F-5 Application-scoped resources contain LogicalNegationConverter

```
<Application
    x:Class="HelloWorldIoTCS_2017.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:converters="using:HelloWorldIoTCS_2017.Converters"
    RequestedTheme="Light">

    <Application.Resources>
        <converters:LogicalNegationConverter x:Key="LogicalNegationConverter" />
    </Application.Resources>
</Application>
```

The IsGpioPinAvailable property of BlinkingViewModel indicates whether the GPIO pin used to control an LED is available. If so, background operation of LED blinking can be started and then stopped by clicking the Start Blinking and Stop Blinking button, respectively. Whenever these buttons are clicked, event handlers invoke corresponding methods of the LedBlinking class (see Listing F-6).

LISTING F-6 Starting and stopping LED blinking

```
private void ButtonStartBlinking_Click(object sender, RoutedEventArgs e)
{
    ledBlinking.Start();
    blinkingViewModel.IsBlinkingActive = ledBlinking.IsActive;
}

private void ButtonStopBlinking_Click(object sender, RoutedEventArgs e)
{
    ledBlinking.Stop();
    blinkingViewModel.IsBlinkingActive = ledBlinking.IsActive;
}
```

The status of LED blinking (enabled or disabled) is indicated by the IsBlinkingActive property of BlinkingViewModel. Therefore, IsBlinkingActive should change to true after invoking LedBlinking.Start, and to false after invoking LedBlinking.Stop. To set the IsBlinkingActive property of the BlinkingViewModel class instance, I use LedBlinking.IsActive flag.

Inherently, I could use the LedBlinking.IsActive property for data binding. However, within the setting of BlinkingViewModel.IsBlinkingActive, I perform additional logic. Namely, as shown in Listing F-7, I raise the PropertyChanged event to inform the UI about the property change, and then toggle BlinkingViewModel properties (IsStartBlinkingButtonEnabled and IsStopBlinkingButtonEnabled), which control the status of the Start Blinking and Stop Blinking buttons (see Listing F-8).

LISTING F-7 Definition of the IsBlinkingActive property from BlinkingViewModel

```
private bool isBlinkingActive;

public bool IsBlinkingActive
{
    get { return isBlinkingActive; }
    set
    {
        isBlinkingActive = value;
        OnPropertyChanged();

        ToggleStartStopButtons(!value);
    }
}
```

LISTING F-8 The Enabled property of the Start Blinking and Stop Blinking buttons is controlled through data binding

```
public bool IsStartBlinkingButtonEnabled { get; private set; }
public bool IsStopBlinkingButtonEnabled { get; private set; }

private void ToggleStartStopButtons(bool isStartEnabled)
{
    IsStartBlinkingButtonEnabled = isStartEnabled;
    OnPropertyChanged("IsStartBlinkingButtonEnabled");

    IsStopBlinkingButtonEnabled = !isStartEnabled;
    OnPropertyChanged("IsStopBlinkingButtonEnabled");
}
```

Configuring and deploying solutions

To deploy the app to the Raspberry Pi 2 or Pi 3, proceed as shown in Chapter 2. Namely, change the Solution Platform setting to **ARM** and set Target to **Remote Machine**. (See Figure F-9.) If you do it for the first time, the Remote Connections dialog box from Figure F-10 appears. You can later open this dialog box using project properties, where you navigate to the **Debug** tab and then click the **Find** button in the Start Options group.

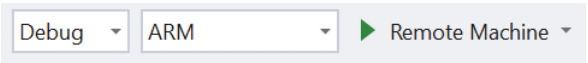

FIGURE F-9 Compilation, solution platform, and target configuration.

After selecting a remote device, run the app by opening the **Debug** menu and choosing either **Start Debugging** or **Start Without Debugging**, or by clicking the **Remote Machine** button (refer to Figure F-9). If you run the app without debugging, then you can stop it from the Processes tab of the Device Portal.

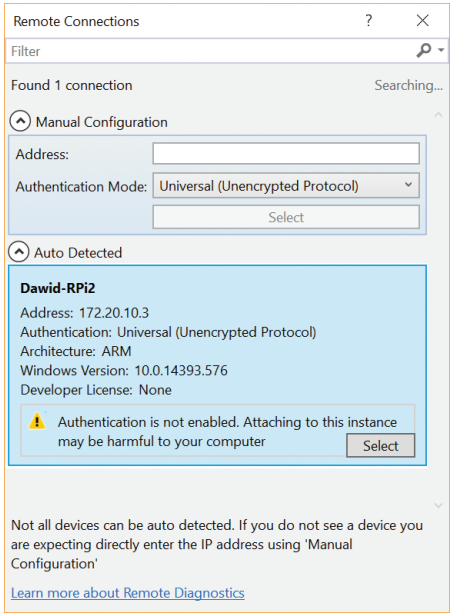

FIGURE F-10 The Remote Connections dialog box can be used to quickly find Windows remote machines in the local network.

I showed how to write the headed app only because headless projects templates are the same as in Visual Studio 2015 Update 2 and 3.

Portable Class Library

To create the Portable Class Library project, you use the Class Library (Portable) project template. You can find it by typing **Portable** in the search box of the New Project dialog box. Use the Add Portable Class Library dialog box to define your targets, as in Visual Studio 2015. (See Figure F-11.)

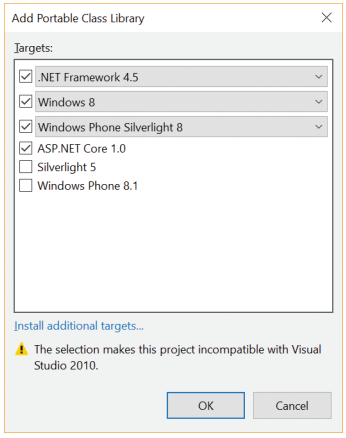

FIGURE F-11 The Add Portable Class Library dialog box of Visual Studio 2017.