

Programming for the Internet of Things

Using Windows 10 IoT Core
and Azure IoT Suite



Dawid Borycki

Programming for the Internet of Things: Using Windows 10 IoT Core and Azure IoT Suite

Dawid Borycki

PUBLISHED BY

Microsoft Press
A division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2017 by Dawid Borycki

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2016959963

ISBN: 978-1-5093-0206-2

Printed and bound in the United States of America.

First Printing

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Support at msspinput@microsoft.com. Please tell us what you think of this book at <https://aka.ms/tellpress>.

This book is provided “as-is” and expresses the author’s views and opinions. The views, opinions and information expressed in this book, including URL and other Internet website references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

Microsoft and the trademarks listed at <https://www.microsoft.com> on the “Trademarks” webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

Acquisitions Editor: Devon Musgrave

Editorial Production: Polymath Publishing

Technical Reviewer: Chaim Krause

Copy Editor: Traci Cumbay

Layout Services: Shawn Morningstar

Indexing Services: Kelly Talbot Editing Services

Proofreading Services: Corina Lebegioara

Cover: Twist Creative • Seattle

Contents

	<i>Introduction</i>	<i>xi</i>
PART I	ESSENTIALS	1
Chapter 1	Embedded devices programming	3
	What is an embedded device?	3
	Special-purpose firmware	3
	Microcontroller memory	4
	Embedded devices are everywhere	5
	Connecting embedded devices: the Internet of Things	7
	Fundamentals of embedded devices	9
	Embedded device programming vs. desktop, web, and mobile programming	12
	Similarities and user interaction.	12
	Hardware abstraction layer	12
	Robustness	13
	Resources	14
	Security	14
	Benefits of the Windows 10 IoT Core and Universal Windows Platform	14
	Summary	16
Chapter 2	Universal Windows Platform on devices	17
	What is Windows 10 IoT Core?	17
	The power of Universal Windows Platform for devices	18
	Tools installation and configuration	19
	Windows 10	20
	Visual Studio 2015 or later	21
	Windows IoT Core project templates	21
	Windows 10 IoT Core Dashboard	22
	Device setup	23
	Windows 10 IoT Core Starter Pack for Raspberry Pi 2 and Pi 3	23
	Windows 10 IoT Core installation	26
	Configuring the development board	27
	Hello, world! Windows IoT	29
	Circuit assembly	29
	Using C# and C++ to turn the LED on and off	36
	Useful tools and utilities	47
	Device Portal	47
	Windows IoT Remote Client	49
	SSH	51
	FTP	52
	Summary	54

Chapter 3	Windows IoT programming essentials	55
	Connecting Raspberry Pi 2 to an external display and boot configuration	56
	Headed and headless modes.	57
	Headless applications.	58
	C#.	58
	C++.	61
	Summary.	67
	An entry point of the headed application.	67
	C#/XAML	68
	Asynchronous programming	73
	Worker threads and thread pool.	73
	Timers.	76
	Worker threads synchronization with the UI	82
	Blinking the LED using DispatcherTimer	86
	Summary	91
 Chapter 4	 User interface design for headed devices	 93
	UI design of UWP apps	93
	Visual designer.	94
	XAML namespaces	96
	Control declaration, properties, and attributes.	99
	Styles	101
	Style declaration	102
	Style definition.	103
	StaticResource and ThemeResource markup extensions	107
	Visual states and VisualStateManager.	110
	Adaptive and state triggers.	115
	Resource collections	118
	Default styles and theme resources	124
	Layouts	125
	StackPanel.	125
	Grid	127
	RelativePanel	130
	Events	132
	Event handling.	133
	Event handlers and visual designer.	137
	Event propagation	138
	Declaring and raising custom events.	140
	Data binding	143
	Binding control properties	144
	Converters.	146
	Binding to the fields.	147
	Binding to methods	152
	Summary	154

Chapter 5	Reading data from sensors	157
	Bits, bytes, and data types	158
	Decoding and encoding binary data	159
	Bitwise operators	159
	Shift operators, bit masking, and binary representation	160
	Byte encoding and endianness	168
	BitConverter	170
	BitArray	172
	Sense HAT add-on board	175
	User interface	175
	Temperature and barometric pressure	178
	Relative humidity	190
	Accelerometer and gyroscope	193
	Magnetometer	198
	Sensor calibration	205
	Singleton pattern	206
	Summary	208
Chapter 6	Input and output	209
	Tactile buttons	210
	Joystick	213
	Middleware layer	214
	Joystick state visualization	218
	LED array	222
	Joystick and LED array integration	230
	Integrating LED array with sensor readings	233
	Touchscreen and gesture handling	234
	Summary	240
Chapter 7	Audio processing	241
	Speech synthesis	241
	Speech recognition	245
	Background	245
	App capability and system configuration	246
	UI changes	247
	One-time recognition	247
	Continuous recognition	251
	Device control using voice commands	253
	Setting up the hardware	253
	Coding	254
	Waves in time and frequency domain	257
	Fast Fourier Transformation	258
	Sampling rate and frequency scale	264
	Decibel scale	265

Chapter 10	Motors	389
	Motors and device control fundamentals	389
	Motor HATs	390
	Pulse-width modulation	391
	Driver	392
	DC motor	398
	Implementation of the motor control with PWM signals	399
	Headed app	402
	Stepper motor	405
	Full-step mode control	407
	Headed app	412
	Automatic speed adjustment	414
	Micro-stepping	417
	Servo motor.	423
	Hardware assembly	424
	Headed app	424
	Providers.	427
	Lightning providers	428
	PCA9685 controller provider	429
	DC motor control	432
	Summary	434
Chapter 11	Device learning	435
	Microsoft Cognitive Services	436
	Emotions detector	436
	Indicating emotions on the LED array	445
	Computer Vision API	448
	Custom artificial intelligence	450
	Motivation and concepts	450
	Microsoft Azure Machine Learning Studio.	452
	Anomaly detection	461
	Training dataset acquisition	461
	Anomaly detection using a one-class support vector machine	467
	Preparing and publishing a Web service.	471
	Implementing a Web service client	474
	Putting it all together	479
	Summary	482
PART III	AZURE IOT SUITE	483
Chapter 12	Remote device monitoring	485
	Setting up a preconfigured solution	486
	Provisioning a device	488
	Registering a new device	489
	Sending device info	490
	Sending telemetry data.	497

Receiving and handling remote commands	501
Updating device info	501
Responding to remote commands	503
Azure IoT services	506
Summary	506
Chapter 13 Predictive maintenance	507
Preconfigured solution	508
Solution dashboard	509
Machine Learning Workspace	510
Cortana Analytics Gallery	514
Azure resources	514
Azure Storage	517
Predictive maintenance storage	517
Telemetry and prediction results storage	518
DeviceList	519
Azure Stream Analytics	520
Solution source code	523
Event Hub and the machine learning event processor	523
Machine learning data processor	528
Azure Table storage	531
Simulator WebJob	535
Predictive Maintenance web application	539
Simulation service	539
Telemetry service	540
Summary	542
Chapter 14 Custom solution	543
IoT Hub	544
Client app	546
Device registry	548
Sending telemetry data	553
Stream analytics	554
Storage account	554
Azure Tables	556
Event Hub	556
Stream Analytics Job	558
Event processor	566
Data visualization with Power BI	573
Notification Hub	577
Store association	578
Notification client app	579
Notification Hub creation and configuration	585
Sending toast notifications with the event processor	587
Deploying the Event Hub processor to the cloud	590
Summary	593

Appendix A	Code examples for controlling LED using Visual Basic and JavaScript	A-1
	Headed app using Visual Basic/XAML	A-1
	Headless app using Visual Basic/XAML	A-2
	Headed app using JavaScript, HTML, and CSS	A-4
	An entry point of the headed JavaScript app	A-6
Appendix B	Raspberry Pi 2 HDMI modes	B-1
Appendix C	Bits, bytes, and data types	C-1
	Binary encoding: integral types	C-1
	Binary encoding: floating numbers	C-4
	Hexadecimal numeral system	C-4
	Numeral values formatting	C-5
	Binary literals and digit separator	C-6
Appendix D	Class library for Sense HAT sensors	D-1
	Portable Class Library	D-2
	Universal Windows Class Library	D-3
	Universal Windows Platform application	D-5
Appendix E	Visual C++ component extensions	E-1
	User interface and event handling	E-1
	Event declaration and event arguments	E-4
	Concurrency	E-5
	Data binding	E-8
	Value converters	E-10
	Summary	E-11
Appendix F	Setting up Visual Studio 2017 for IoT development	F-1
	Installation	F-1
	Visual C# project template	F-3
	Creating a project	F-3
	IoT extensions	F-4
	Implementation	F-4
	Configuring and deploying solutions	F-8
	Portable Class Library	F-9
	<i>Index</i>	595

This page intentionally left blank

Introduction

Lately, the Internet of Things (IoT), big data, machine learning, and artificial intelligence have become very hot topics. *IoT* is defined as the global network of interconnected devices. These devices can be as small as implantable continuous glucose monitors or wearables, or as big as credit card-sized computers, like the Raspberry Pi. As the number of such devices continues to grow, the amount of data they generate will rapidly increase—and new technological challenges will appear.

The first of these challenges relates to storage. Small devices have physical constraints that do not allow them to store big datasets. Second, big data exceeds the computational capabilities of traditional algorithms and requires different, statistical-based approaches. These are provided by machine learning, a branch of artificial intelligence. Hence, IoT, big data, machine learning, and artificial intelligence are tightly related concepts. Typically, devices are end-points, which send data over the network to the cloud, where data is stored and processed to get new, previously unavailable insights. These insights may help to understand and optimize processes monitored by smart devices.

While this description may sound fascinating, the number of new technologies you need to learn to start implementing custom IoT solutions might seem daunting. Fortunately, Microsoft created Windows 10 IoT Core and Azure IoT Suite, which enable you to program custom IoT solutions fairly quickly. Their functionality is limited only by your imagination. In this book, you will find numerous projects presented in a step-by-step manner. By completing them, you not only obtain the fundamentals of device programming, but you will also be ready to write code to revolutionize devices and robots, which can do the work for you!

This book helps you to master IoT programming in three main parts. Each contains a suitable level of detail and explains how to prepare your development environment, read data from sensors, communicate with other accessories, build artificial vision, build motors, build hearing systems, and incorporate machine learning and artificial intelligence into your device. This book also shows you how to set up remote telemetry and predictive maintenance like Azure IoT solutions and to build custom IoT solutions from scratch.

Audience and expected skills

This book is devoted to students, programmers, engineers, enthusiasts, designers, scientists and researchers who would like to use their existing programming skills to start developing software for custom devices and sensors and also use the cloud to store, process, and visualize remote sensor readings.

I assume the reader knows fundamental aspects of C# programming and is experienced in Windows programming. Therefore, no special discussion is devoted to C# or to programming fundamentals. I do not assume any previous knowledge of audio and image processing, machine learning, or Azure. These topics are explained in detail.

Tools and required hardware

Throughout this book, I use Windows 10 and Visual Studio 2015/2017 as the development environment. Most of the hardware components I use are from the Microsoft IoT Pack for Raspberry Pi, provided by Adafruit Industries. Any additional hardware elements like cameras, add-on boards for Raspberry Pi, communication adapters, or motors will be described in chapters dealing with the particular topic.

Organization of this book

This book is divided into the following three parts:

- Part I: Essentials
- Part II: Device programming
- Part III: Azure IoT Suite

In Part I, I explain the fundamentals of embedded programming and discuss how they differ from desktop, web, and mobile app programming. I also show how to configure the programming environment and write “Hello, world!”-like projects on the Windows 10 IoT Core. Additionally, I describe several fundamental concepts regarding the UWP threading model and XAML markup for declaring the UI. Most experienced developers can skip elements of this part that they are already well-versed in and proceed to the second part.

Part II contains chapters related to device programming with Windows 10 IoT Core and the UWP. I first show you how to acquire data from multiple sensors and control a device. Subsequently, I explain how to acquire and then process signals from a microphone and a camera. Then, I show you how to use various communication protocols, including serial communication, Bluetooth, Wi-Fi, and AllJoyn, to enable your IoT module to communicate with other devices. I also show you how to control motors and use Microsoft Cognitive Services and Azure Machine Learning to make your device really smart and intelligent.

Part III is devoted to the cloud. I show you how to use two preconfigured Azure IoT solutions for remote device telemetry and predictive maintenance. In the last chapter, I present a detailed process of building a custom IoT solution from scratch. This shows

the essence of IoT programming, in which data from remote sensors is transferred to the cloud, where it is stored, processed, and presented. Moreover, I explain how to report abnormal sensor readings directly to the mobile app running on Windows 10.

This material is supplemented by six appendices, which show how to blink an LED with Visual Basic and JavaScript (Appendix A), present HDMI modes of the Raspberry Pi (Appendix B), explain bit encoding (Appendix C), describe code-sharing strategies (Appendix D), introduce Visual C++/Component Extensions (Appendix E), and show how to set up Visual Studio 2017 for IoT development (Appendix F). These appendices are available online here: <https://aka.ms/IoT/downloads>.

Conventions

The following conventions are used in this book:

- **Boldface** type is used to indicate text that you type.
- *Italic* type is used to indicate new terms and URLs.
- Code elements appear in a monospaced font.

About the companion content

I have included companion code to enrich your learning experience. The companion code for this book can be downloaded from the following page:

<https://aka.ms/IoT/downloads>

You may also download the code from GitHub here:

<https://github.com/ProgrammingForTheIoT>

The source code is partitioned into subfolders that correspond to particular chapters and appendices. To improve book readability, in many places, I refer to the companion code rather than showing the full listing, so it is good to have the companion code open while reading this book.

Acknowledgments

This book would not exist without the support of Devon Musgrave, who enthusiastically responded to my book proposal and provided initial comments along with writing guidance.

I'm grateful to Chaim Krause for thoroughly checking every single project discussed in this book and finding all—even the smallest—issues. I'm also very indebted to Kraig Brockschmidt, who comprehensively peer-reviewed every chapter. His wide experience and valuable comments significantly improved the quality of this book. Finally, thanks to Traci Cumbay for her excellent work as copy editor.

Many thanks to Kate Shoup for managing the book's production. I also thank Kim Spilker for shepherding this book to its completion.

Finally, special thanks go to my wife Agnieszka and daughter Zuzanna for their continuous support and patience, shown to me during the writing of this book.

Errata and book support

We have made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at:

<https://aka.ms/loT/errata>

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *msspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority and your feedback our most valuable asset. Please tell us what you think of this book at:

<https://aka.ms/tellpress>

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: @MicrosoftPress.

PART I

Essentials

The first part of this book covers basic aspects of programming for the Internet of Things (IoT) using Windows 10 IoT Core. Chapter 1, “Embedded devices programming,” defines embedded devices, describes their role, and shows how such devices compose the IoT. It explains why embedded programming is challenging and how it differs from desktop, web, and mobile programming.

In Chapter 2, “Universal Windows Platform on devices,” I introduce the Universal Windows Platform (UWP) and Windows IoT Core and show you the advantages and limitations of using these tools for rapid software development for embedded devices. I show you how to install and configure a development environment and implement a “Hello, world!” project for the Windows IoT device using selected programming models available on the UWP.

Chapter 3, “Windows IoT programming essentials,” dives into asynchronous programming—one of the key aspects for IoT programming. I show you the difference between headed and headless modes, and I characterize the `IBackgroundTask` interface and asynchronous programming patterns for UWP apps. In this chapter, I also discuss timers and thread synchronization.

Chapter 4, “User Interface design for headed devices,” runs through the most important aspects of designing a user interface (UI) for headed Windows IoT Core devices using XAML. These include controls used to define UI layout (`Grid`, `StackPanel`, `RelativePanel`), control styling and formatting, events, and data binding.

This page intentionally left blank

Embedded devices programming

Embedded devices work as the control units of a broad range of tools including house accessories, car engines, robots and medical devices. These control units use specially designed software to exchange data with sensors of any kind. Embedded devices apply sophisticated algorithms to sensor data to monitor, control, and automate the specific process. This chapter defines embedded devices, discusses their role, and shows the possibilities that arise from connecting such smart devices. It also describes the structure of embedded devices, their programming aspects, and the common problems and challenges they can present.

What is an embedded device?

An *embedded device (ED)* is a special-purpose computing system for automating a specific process. Unlike a general-purpose computer, which has a fairly standard set of peripherals (input/output devices for display, storage, communication), an ED is designed for a specific purpose. As a consequence, input and output devices of an ED can be far different from those in general-purpose computers. An ED can be fully functional without a keyboard or a monitor; as you can easily imagine, using your laptop or desktop without such fundamental components would be impossible.

Though specialized and general-purpose computers differ by peripherals, their core parts are similar: a central processing unit (CPU or microprocessor) and memory. The microprocessor executes the computer program, which consists of the instructions fetched from memory. Procedures executed by the CPU then control the dedicated hardware. Such a combination of an ED and hardware is called an embedded system.

Special-purpose firmware

In contrast to a typical computing system, an ED is usually dedicated to controlling particular hardware; hence, its form factor and processing capabilities are tailored to the special system. In particular, an ED does not necessarily have to run multiple programs at the same time. Instead, an ED runs specially designed software, called firmware. The firmware functionality is usually not generic and performs tasks devised to the particular hardware. Typically, firmware is loaded to the device in the factory or by the maker during development.

You see an example of firmware specificity in microwave ovens. An embedded device built into a microwave controls the time and heating temperature of the food based on user input provided through the key pad or touch screen. Unlike a keyboard, which is essentially the same for any general-purpose computer, microwave input components strongly differ among devices. Thus the ED of the particular microwave manufacturer cannot be generalized to all microwave ovens. Furthermore, different ovens are equipped with distinct intrinsic sensors and electronic components. Hence, each model has its own specific firmware, adjusted to the hardware capabilities and the system's purpose.

The lifecycle of the firmware loaded to an embedded device is quite different from typical applications of computer systems. The program stored in ED memory is activated whenever the device is switched on and works as long as the device is powered. During this time, firmware communicates with sensors and also input/output (I/O) devices using peripherals. These peripherals constitute interfaces between intrinsic parts of an ED and its environment. Most common peripherals include the following:

- Serial Communication Interface (SCI)
- Serial Peripheral Interface (SPI)
- Inter-Integrated Circuit (I²C)
- Ethernet
- Universal Serial Bus (USB)
- General Purpose Input/Output (GPIO)
- Display Serial Interface (DSI)

Typically, an ED does not require a full-size display. In the extreme case, an ED can even have just a single-pixel display, composed of a single LED used as an indicator. Color or blinking frequency of such an LED can communicate errors or encode monitored values.

Microcontroller memory

Very often, an embedded device has to be accommodated in a very small housing and be power-efficient. To save space and resources, CPU, memory, and peripherals are integrated into a single chip, which is called the microcontroller.

The microcontroller's memory is divided into two main parts: Read Only Memory (ROM), which stores the firmware, and Random Access Memory (RAM), which stores variables used by the software components. ROM memory is non-volatile and can be modified using additional developer tools and (or) a programmer. Non-volatile memory is required to instantly load firmware as soon as the ED is powered up. For example, when you power up your wireless router, it begins execution of the firmware stored in ROM, while your connection settings, including credentials, frequency band, and Service Set Identifier (SSID) are managed in RAM (typically they are loaded from some non-volatile memory to RAM, after the device's boot).

This memory configuration resembles the scenario used in other computer systems, in which ROM stores the special program, known as the basic input/output system (BIOS) or the Unified Extensible Firmware Interface (UEFI). Typically, the BIOS runs immediately after the computer is turned on, and it initializes hardware and loads the operating system, which then creates processes (program instances) and their threads.

ED memory is also supplemented by additional non-volatile storage, termed Electrically Erasable Programmable ROM (EEPROM). Writing to EEPROM is very slow; its main purpose is to store device calibration parameters, which are restored to RAM after a power loss. The data stored in EEPROM depends on the applications and the device type but usually contains calibration parameters used for converting the raw data acquired from sensors into values representing physical parameters such as temperature, humidity, geolocation, or device orientation in three-dimensional space. EEPROM serves as the basis for flash memory, which is used in modern memory sticks and solid state drives (SSD). These newer designs offer much faster speeds than EEPROMs. Figure 1-1 shows a summary of memory types.



FIGURE 1-1 Different purposes require different types of memory.

EEPROM memories usually are designed to store larger amounts of data. Accessing data in large sets can be slow, especially for I/O operations. Hence, to improve I/O, processors also use memory registers—quickly accessible locations for small amounts of fast memory. Registers are especially important for microcontrollers because they control peripherals, as I describe later in this chapter.

Depending on the application, the performance, capabilities, and peripherals may significantly differ among devices. For example, the processing performance of an ED controlling a car engine must be much higher than that of a microcontroller embedded in the simple consumer electronic gadget, like a media receiver. The proper and error-free control of the vehicle is much more critical than that of an electronic gadget.

Embedded devices are everywhere

Embedded devices are everywhere, and are often so hidden that we don't even notice their existence. In the automotive world, numerous internal and external sensors within the vehicle's modules constantly monitor intrinsic systems. Data from these detectors is transferred through peripherals to an appropriate ED, which continually analyzes this input to keep track of the vehicle traction, control the

engine, or display an external temperature or vehicle's location, among other functions. In the financial sector, embedded devices control units of the automated teller machines (ATM) to enable a bank customer to make a financial transaction automatically. Healthcare screening devices are also managed by embedded devices that control the position of a light beam to noninvasively produce an image of the human body or deliver information about diseases. Intelligent buildings, weather stations, and security devices are equipped with specially designed microcontrollers that acquire data from sensors or images from cameras; they then inspect them using digital signal and image-processing techniques to monitor temperature or humidity, detect unauthorized access, or optimize resource usage.

Embedded devices are becoming an important component of the personal healthcare systems. Wearable embedded devices containing heart-rate and blood pressure sensors or even noninvasive glucose monitoring systems can continuously read health parameters, process them in real-time, and transfer this data to the wearer's doctor. These wearable EDs may significantly improve diagnosis and treatment by providing detailed information about the wearer's health status.

Embedded devices simplify energy usage monitoring through remote reading from power meters. Information coming from embedded devices controlling regulatory drivers can optimize power distribution.

Small, form-factor, smart devices are not limited to serious applications. They can also provide a lot of joy. Kinect and HoloLens are prominent examples of devices that have business and fun applications. Kinect is a motion controller, equipped with movement sensors and cameras to recognize complex gestures and track people; you might know it from Xbox computer games. HoloLens further advances these developments by providing augmented reality to significantly enhance perception. The core elements of both Kinect and HoloLens are multiple sensors and cameras that analyze the environment and process input from voice, gesture, or gaze.

Primarily, embedded devices act as artificial intelligence systems that automate everyday actions performed by people to make our lives easier and better. An ED takes data as an input, processes it, makes decisions, and implements control algorithms and corrective procedures. They not only automate specific processes but can predict manufacturing failures, diseases, accidents, or weather.

An ED can easily detect sensor readings that exceed thresholds set by the programmer and perform predictive analysis and even preventive maintenance ranging from saving food from being overheated to preventing a car engine from being damaged. Because microchips can now efficiently run very advanced software that implements sophisticated control and diagnostic algorithms, an ED can perform process automation and predictive analysis that significantly reduces the risk of using a particular system, diminish process cost and time, and improve efficiency.

A lot of applications for embedded devices already exist, but many more unexplored possibilities are easy to imagine, as are the advantages of building new devices. Many of these arise from the possibility of connecting smart devices into the advanced networks of hardware units.

Connecting embedded devices: the Internet of Things

ARPANET, the Internet predecessor, was created to enhance the potential of isolated general-purpose computer systems. Connecting workstations accelerated communication and data sharing. Moreover, new software versions could be quickly distributed among connected computers, and computations could be run in parallel on multiple systems. These advantages quickly proved very useful and were translated to public networks, which later became the one global system of interconnected computer networks—the Internet. Nowadays, the Internet is one of the fundamental elements helping people to communicate, share files, distribute information, and automate and simplify many everyday processes. In short, the power of general-purpose computers was enormously amplified when they became interconnected via the Internet.

A similar idea produced the Internet of Things (IoT), the network of distributed embedded devices. EDs are very useful in isolated systems, but their power is enhanced tremendously when they're connected into a global detection or monitoring system that includes many hardware units. This connectivity yields a lot of advantages, because the large amount of data can provide invaluable information about the status of a given business process or monitored system. Data analysis can then lead to completely new conclusions unavailable by using a single smart device or sensor or by monitoring the given process manually.

In a sense, IoT is the world of various connected devices, which acquire data from sensors and then distribute this information among other computer systems, either desktop or mobile, by using local or global communication networks. Depending on the nature of the application, you can benefit from IoT with just one ED—or billions. The number, type, and capabilities of devices in the IoT grid can be tailored to particular requirements, processes, or systems. But new devices aren't always necessary; IoT can be composed of existing devices and sensors, as in the case of the MyDriving app (<http://aka.ms/iotsampleapp>, <https://channel9.msdn.com/Shows/Visual-Studio-Toolbox/MyDriving-Sample-Application>).

IoT devices are becoming the crucial part of automation and robotics because of rapid technological advances in data transfer rates, sensor and device miniaturization, and microcontrollers that can process large amounts of data using advanced control and diagnostic algorithms. Although a single IoT device can process readings from connected sensors and perform appropriate actions, that device can't always store large amounts of data. Moreover, analysis of the information coming from many IoT devices becomes challenging, especially in the case of large IoT grids.

Current and future IoT applications rely not only on the embedded device itself but also on the ability to extract invaluable insights from data acquired using that device. Connecting smart devices yields new possibilities and brings new challenges in terms of processing and analyzing large amounts of data. Every device may be integrated with different sensors and thus use distinct communication protocols. Combining smart devices requires sophisticated acquisition, storage, and processing approaches in which data coming from different devices is unified and processed using statistical models on the shared system.

Such a centralized processing unit performs advanced analysis and turns untapped data into clear, readable reports by exposing the uniform interface for presenting, accumulating, and filtering acquired and processed data. Therefore, IoT is usually composed of a central storage and processing system, which gives users the ability to connect their devices and easily process, and more importantly, understand data coming from those smart units. This functionality is delivered by the Microsoft Azure IoT Suite, which I describe in detail in Part III, “Azure IoT Suite.”

Figure 1-2 shows an example of the Microsoft Azure IoT Suite as the central management system for IoT devices integrated with different sensors.

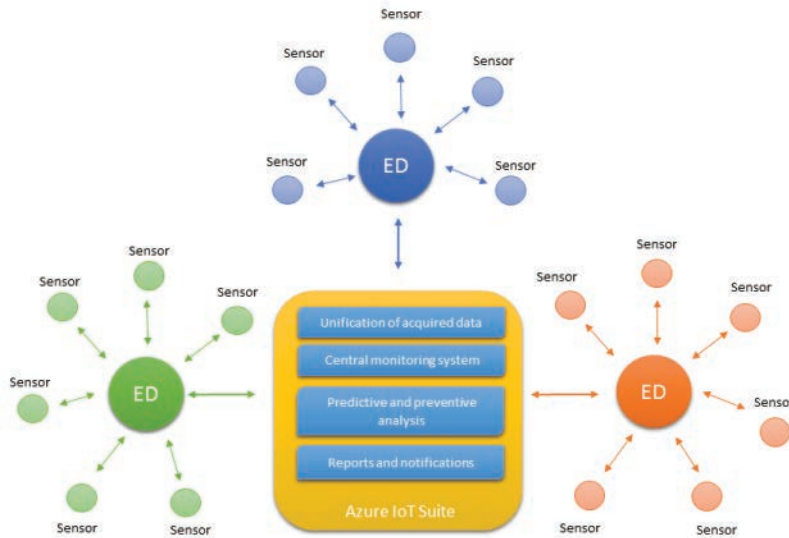


FIGURE 1-2 In an IoT, data coming from various sensors connected to distinct embedded devices (ED) is transferred to the central, cloud-based system.

To put it in its most basic terms, the *things* in IoT means devices and sensors of any kind, while the *Internet* refers to the centralized system that connects and manages those devices. That system further processes untapped data coming from sensors using business intelligence techniques to generate clean and readable information, which in turn simplifies decision-making, enables predictive and preventive analysis, and automates many business processes.

Electric energy usage provides a practical example of how IoT simplified business process. First, the electromechanical meters were replaced by electronic meters (embedded devices) that not only measure electric energy usage and provide clearer display but can also record other parameters to support time-of-day billing or prepayment meters. Electronic meters greatly enhanced measurements of electric energy usage; however, manual sensor readings were still required—until electric energy meters were connected to obtain readings remotely and store them in the central processing system. As a result, power stations automatically acquire and process data not only for billing purposes but also to optimize electric energy distribution, maintain the network, or predict malfunctions.

Fundamentals of embedded devices

How does software executed by the CPU interact with peripherals to acquire data from sensors and communicate with other devices? It requires several hardware and software concepts.

From the hardware point of view, the microcontroller is connected to peripherals through physical connectors, exposed as pins on the external casing. (See Figure 1-3.) The process is independent of the particular transmission protocol and medium used to transfer data (e.g., wires, fibers, or free-space communication channels) and works like this:

1. Wired connections between the microcontroller and peripherals carry electrical signals that encode bits of information as physical quantities such as the voltage or current.
2. These physical observables are converted to digital values using appropriate analog-to-digital converters.
3. Digital values are converted back to physical quantities before being sent to peripherals using digital-to-analog converters.

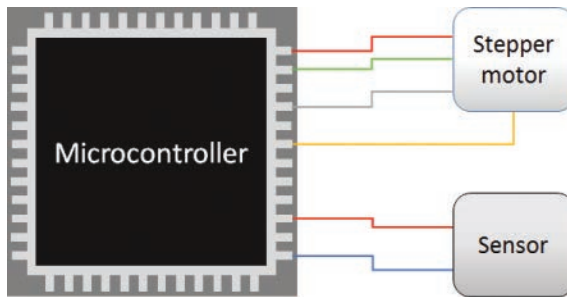


FIGURE 1-3 Peripherals are connected to the microcontroller pins.

Software accesses received binary data (read operation) and knows how to send data to peripherals (write operation). In general, the digital representation of signals received from peripherals is distributed among the devices using the data bus. Proper data distribution requires an address bus, which carries information about physical locations of binary data in physical memory. In general, there are two ways of reading and writing data to peripherals using data and address buses. These are defined as the port-mapped and memory-mapped I/O.

- In the port-mapped I/O, the CPU uses separate address buses for addressing data in local memory and in peripherals. Special read/write instructions transfer data between the microcontroller and peripherals. (See Figure 1-4.) From the hardware point of view, having separate address buses simplifies addressing. However, from the software point of view, accessing the peripherals using port-mapped I/O is quite complicated because it not only requires reading or writing data to memory registers but also requires appropriate I/O instructions for receiving and sending binary data from and to a peripheral.

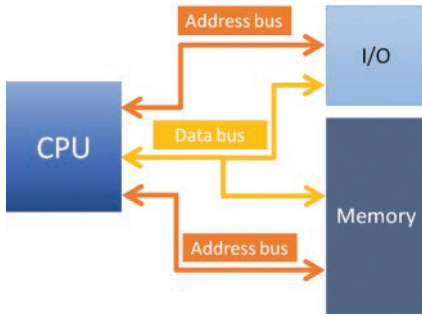


FIGURE 1-4 In port-mapped I/O, two separate address buses point to physical locations in memory and peripherals; memory and I/O devices have to be accessed separately.

- Memory-mapped I/O reserves some part of the RAM memory for communication, so firmware virtually accesses I/O devices in the same way it accesses memory registers—with a single address bus. (See Figure 1-5.) This approach naturally simplifies software. However, using a single address bus to control memory and peripherals transfers software complications to a lower level. More sophisticated schemas are therefore required for address decoding and encoding. In addition, the amount of memory available to the user is slightly decreased. On the Windows desktop platform, you can check the amount of such hardware-reserved memory using Task Manager. (See Figure 1-6.)

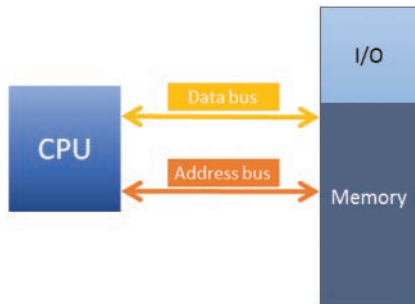


FIGURE 1-5 In memory-mapped I/O, a single address bus points to physical locations in memory and peripherals; additional I/O instructions become unnecessary.

Registers are the building blocks of the RAM memory, and—depending on the microcontroller type—may contain 8, 16, 24, 32, or 64 bits. Some of these registers in the amount defined by the microcontroller’s manufacturer are designed to exchange data between a processor and peripherals and thus decrease the amount of RAM memory for the user. (Refer to Figure 1-6.) Each bit of such specially designed register is mapped to the physical I/O ports, which constitute the physical pins of the microcontroller.

The logical bit values assigned to the particular pin are controlled by the voltage level or current intensity, which define their off (0) and on (1) states. Because these pins are mapped to memory registers, any voltage level or current changes are automatically reflected into the memory registers.

Thus, firmware accesses the data received from the peripheral by reading an appropriate memory register, identified by its address, pointing to its location in the memory. Data is sent back to the peripheral in the same way—that is, by modifying values stored in registers. This process requires conversion between physical quantity such as voltage or current and binary representation.

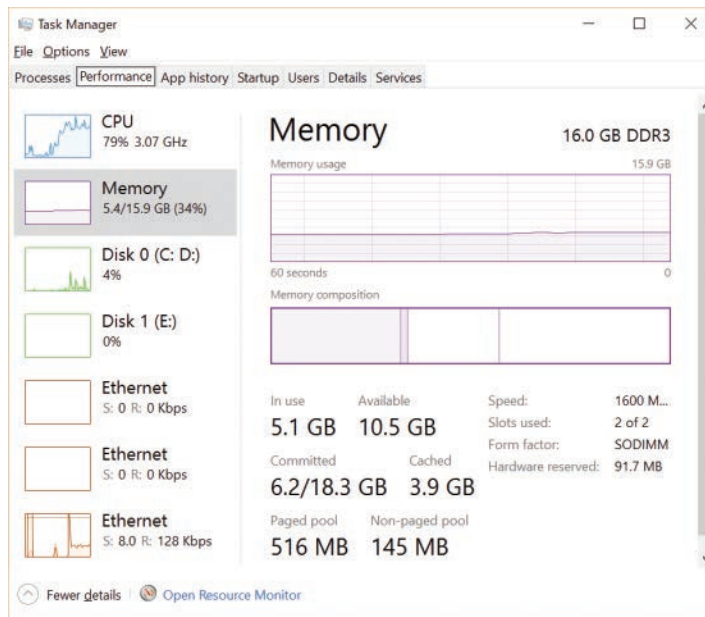


FIGURE 1-6 In computer systems, which use memory-mapped I/O, hardware-reserved memory decreases total memory available for applications; in this example, hardware uses approximately 92 MB.

In some circumstances, the approach may slow down overall memory access, because converting and transferring physical signals between CPU and peripherals can be slower than the intrinsic mechanism used by RAM. Moreover, some memory is reserved for communication and is not accessible to the user. Using port-mapped I/O, in which physical pins are not mapped to the memory registers, can sometimes be preferable. CPU uses additional commands to send requests and receive answers from the peripherals. The data transferred is stored in a separated address space but requires additional physical pins for initiating communication. The particular communication approach depends on the microcontroller manufacturer. Programmers need to know the address of memory registers associated with the particular peripheral and communication protocol. (See Chapter 6, “Input and output.”)

The software of the embedded device does not need to constantly read values from registers to get an updated state of the sensor. Instead, the firmware can be automatically informed whenever an appropriate event occurs. For this purpose, microcontrollers use interrupts, which are the signals generated whenever pins change their physical state. The CPU executes an interrupt handler, which is a software function associated with a given interrupt. This allows firmware to react to external events without endlessly reading register values. Such reactive programming is similar to the event-based approach known in high-level application programming. In such a case, every user request or action, like pressing a button, generates an event that in turn runs the associated event handler. The logic implemented within this procedure responds to the user request.

Embedded device programming vs. desktop, web, and mobile programming

Although embedded systems are programmed using the same languages and similar tools as desktop, web, and mobile apps are, ED coding requires direct interaction with hardware elements. Hence, ED programming differs from desktop, web, and mobile coding. There are also many similarities, however. Specific comparisons in terms of user interface, hardware abstraction layer, robustness, resources, and security are discussed below.

Similarities and user interaction

The idea behind interrupts and interrupt handling is quite similar to events and event handling in desktop and mobile programming. However, each technology uses distinct nomenclature, which closely matches internal aspects of the particular programming scope. In desktop and mobile applications, events are related to any user action, like pressing a button or scrolling lists. Any action of this kind generates an event, which can be processed by methods called event handlers. Events can be also triggered by hardware or system-related issues to indicate, for example, low battery level, loss of wireless connection, or connection/disconnection of an external device. Event handlers can be used to respond to any user action or occurrence triggered by the hardware or the operating system.

Similarly, in Model-View-Controller applications, developed for mobile (Android, iOS) or web platforms (ASP.NET), every user action or query incoming from other applications or services is defined as the request or action. Every request is processed by the request-handler module, which maps the particular action to the appropriate method of the class, implementing the controller. The latter interprets a request, updates the state (model) of the application, and produces the corresponding response by presenting a view.

Thus, in all cases, the software responds to user requests or external signals and processes them to take an appropriate action or produce a corresponding response. However, in desktop, web, and mobile programming, these requests are mostly generated by the user; IoT interrupts are usually generated by external signals related to sensors (electrical signals). Thus, embedded device programming differs from desktop, web, and mobile programming by the source that generates events. This doesn't mean that an ED doesn't respond to user requests at all. IoT devices can be equipped with input systems like touch screens, by which users configure an ED. Hence, IoT may also implement a user interface (UI).

In subsequent chapters, to distinguish two possible sources of events, I will refer to interrupts whenever I deal with sensors and to events whenever I discuss methods handling user requests generated through a UI.

Hardware abstraction layer

At first glance, some aspects of ED programming are similar to desktop, web, and mobile application-development techniques, but more key differences than similarities exist. These differences come mostly from the fact that typical high-level programming doesn't require low-level interaction with the hardware.

Because hardware-related aspects are implemented within the operating system or hardware drivers, they typically aren't accessed directly by the programmer.

The common computer or smartphone is an advanced version of an ED. Internal functions of mobile and desktop systems are based on the same concepts as an IoT device. Namely, the CPU communicates with peripherals using similar techniques. Software developers implicitly use them during software development tasks such as accessing files on the hard drive, sending serialized data over a network, or simply displaying messages on the screen.

Conventional computer systems, however, are standardized and use operating systems that implement a hardware abstraction layer. Programming frameworks and convenient application programming interfaces significantly simplify the process of software development by providing a large number of implemented algorithms, data structures, and functions for performing common operations.

ED programming is similar to writing device drivers, which map hardware operations to operating system functions. However, ED software not only provides such an intermediate layer but also controls hardware units. ED development means combining the roles of hardware, drivers, OS, and application into one piece of firmware, so the boundaries are more blurred.

In general-purpose computer systems, the hardware abstraction provides a unified layer, which allows customers to buy any kind of keyboard, mouse, display, storage, and so on, without worrying about their underlying differences. For ED development, you can virtually work with any kind of device without any intermediate layer.

Robustness

Computing devices embedded in car safety systems, electronic stability programs (ESPs) offer a good example of ED robustness. ESP controls a vehicle's stability by detecting loss of traction. It analyzes data coming from various hardware units sensing the speed and acceleration of the car's wheels. The data analysis required to predict understeer or oversteer can be very complex. This task requires constant data processing and noise-issue handling.

In real-world applications, sensor readings can be affected by noise arising from intrinsic electronic circuits due to some fundamental physical effects. Therefore, the sensor readings may vary in time. The programmer must take these noise effects into account, usually by accumulating readings over time and processing them using statistical measures, like mean or median. Depending on the application, such processing may require more advanced control algorithms to filter incorrect readings and to provide steady and predictable control of the hardware units. This is especially important for safety-critical applications. For this reason, the firmware should be robust and error-free in order to quickly respond to rapidly changing and noise-affected sensor readings.

For specific applications (like ESP), the IoT software needs to be optimized to act very fast, because sometimes a few milliseconds' delay may play a very crucial role, while in the case of typical desktop, web or mobile applications, such a delay will be probably unnoticed by the user.

Resources

IoT devices are frequently very small to fit into the body of the system they control or monitor. Embedded devices can be as small as a coin or credit card, which means they typically have limited storage and processing power in comparison to typical computers or smartphones. The software controlling the embedded device has to use hardware resources responsibly, without wasting memory and CPU time. This issue is also crucial for high-level programming, but it may not be as important for typical computer systems, which come equipped with large amounts of memory and huge processing capabilities.

Security

As Figure 1-3 shows, data transferred between peripherals and CPU could be accessed by monitoring physical pin signals. This way of reverse engineering your ED could utilize an oscilloscope, although that's unlikely since it requires physical access to your device. On the other hand, when you connect your ED to the network, especially a wireless network, the probability of your data being intercepted increases significantly.

Security is an important issue for any type of programming, but IoT devices connected to wireless networks are most prone to loss of data. For this reason, you need to secure data transferred over a network using cryptography algorithms.

Connected EDs process and collect sensitive data and control critical hardware. Often, they are directly exposed to cyber-attacks—for example, through the Internet connection. Therefore, the so-called *Security of Things* becomes a very important issue.

The data collected by connected EDs includes private information. For example, home-automation electronics track your daily habits—when you leave and come back from your house, TV channels you watch, and so on. It also tracks what images your security cameras capture. And of course, it has the access codes for various devices. Therefore, if home automation is not properly secured, all this data can be stolen. Worse, it could be used to take control of your house or to spy on you. There is also a danger of an attacker remotely taking control of IoT devices that run on your car. For instance, attacker could disable your braking or steering system while you are driving.

These two examples prove how important Security of Things is. You can secure your IoT system by adopting proper filtering and by validating and encrypting transferred data. Further, you must check the internal consistency of the filesystem and other components of the IoT system.

Benefits of the Windows 10 IoT Core and Universal Windows Platform

Several problems arise when programming IoT devices—and can quickly discourage you from developing software for smart devices. These problems are typically due to the following issues:

- Having to use native tools, compiler chains, and programming environments provided by microcontroller manufacturers

- Low-level programming languages and tools
- Debugging difficulties
- The lack of patterns and best practices
- Narrow community
- UI development using cross-platform libraries and tools
- Broad range of sensors and hardware units to control
- Having to secure communication protocols by writing cryptography algorithms from scratch

Windows 10 IoT Core and the Universal Windows Platform (UWP) solve these problems. The former is the most compact version of Windows 10 and is tailored for IoT needs, while the latter is the API for accessing Windows 10 functions and thus simplifies embedded programming.

The UWP provides you a unified API and set of programming tools, which are exactly the same as you probably already use for your web, mobile, or desktop programming. UWP programming tools follow the *write once run everywhere* paradigm. Such an approach provides programming tools and technologies that enable you to write the application by using the single programming language and environment, and then deploy your app to multiple devices, ranging from IoT, through smartphones, and up to desktop and enterprise servers. By using the same tool set you can target additional platforms.

The UWP also implements many comprehensive algorithms and functions that

- Simplify access to sensors
- Perform robust calculations
- Write advanced functionalities with minimal code
- Extensively query your data
- Secure data transfer using cryptography algorithms
- Support many other IoT applications, like signal and image processing, programming artificial intelligence, and interacting with central processing systems, which unify your untapped data and turn them into readable reports

Finally, the UWP can be accessed using several high-level and popular programming languages including C# and JavaScript, which significantly simplifies your software development process. The UWP provides a set of UI controls that can be seamlessly integrated into software for IoT devices. This turns the problematic native-based development of firmware into a cheerful experience of building software for smart, connected devices.

Windows 10 IoT Core and a broad range of UWP functionalities described in subsequent chapters of this book allow you to quickly develop applications for IoT, prepare proof-of-concept solutions, and yield a unique opportunity to build and program custom devices whose functionality is limited only by the maker's imagination.

Summary

This chapter provided theoretical information. It discussed the most important concepts behind IoT, which you typically do not think of when developing software for desktop, web, or mobile platforms. I pointed out several common challenges that embedded programmers must tackle. I also presented how Windows 10 IoT Core and Azure IoT Suite can help you develop IoT solutions.

Universal Windows Platform on devices

IoT devices typically do not have a full-size display, which could be used for the type of output that comes from a typical “Hello, world!” starter application. Instead, IoT devices have only a few LEDs or pixels—or, in the extreme case, just a single LED or pixel—to display information. So, in the world of embedded programming we say “Hello” by turning an LED on and off.

In this chapter, I show you how to use the UWP interfaced through C#, and C++ to trigger the LED. I use these two languages only, because in this book all apps will be implemented using C#. C++ will be used later to implement the Windows Runtime Component, interfacing native code.

I first define Windows 10 IoT Core and guide you through the installation and configuration processes of all the required software and hardware components. I then explain an electronic LED circuit assembly. After you’ve had your hands on the code, I show you useful tools and utilities for remote device management and accessing its contents.

What is Windows 10 IoT Core?

Windows 10 IoT Core is a compact version of Windows 10 designed and optimized for embedded devices. Windows 10 IoT Core implements the platform, hardware, and software abstraction layers, which simplifies the process of application development for IoT devices. Until recently this area was exclusively reserved for rare and native programming technologies. However, thanks to Windows 10 IoT Core, every high-level software developer can now code embedded devices by using the Universal Windows Platform programming interfaces available for all Windows 10 platforms.

Hardware, platform, and software abstraction layers implemented within Windows 10 IoT Core are composed of the native drivers, which can be accessed using any of the UWP programming languages, including C#, C++, Visual Basic, or JavaScript. Windows 10 IoT Core also supports the Python and Node.js runtimes. Therefore, you can easily access microcontroller interfaces, capabilities using high-level programming languages. Most of the low-level stuff—which typically forces you to use archaic or low-level programming constructs—is thankfully performed within Windows 10 IoT Core.

.NET Micro Framework

Windows 10 IoT Core advances from the .NET Micro Framework (NMF), which is the most compact version of the Microsoft .NET Framework. The ideas behind Windows 10 IoT Core and NMF are very similar. Each includes an execution system for IoT devices such that embedded software is executed within the environment, which exposes rich and friendly programming interfaces for quick, secure, robust, and reliable application development.

The power of Universal Windows Platform for devices

All Windows 10 platforms use a common base, implementing a unified kernel and a common application model. The Universal Windows Platform contains a unified application programming interface available for every UWP device. As a result, an application developed using this core part can run on any Windows 10 device, including desktop, mobile, tablet, HoloLens, Xbox, Surface Hub, and IoT devices. However, the core parts of Windows and its API do not contain some specific features designed exclusively for one particular platform or another. This is because some hardware platforms provide features not available on other devices. For example, IoT devices can control certain custom sensors that aren't available on desktop or mobile devices. The implementation of all programming interfaces in the core part of the UWP would be redundant, so, to target platform specificity, the UWP delivers software development kit (SDK) extensions designed for particular device families (enabling access to features available exclusively on the IoT platform, for example). Figure 2-1 shows the relationship between the core part of the UWP and SDK extensions.

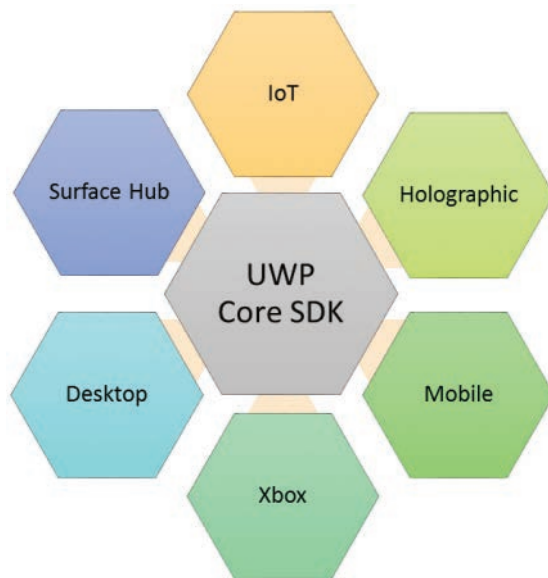


FIGURE 2-1 To access features specific for a particular device family you can reference an appropriate SDK extension: IoT, Holographic, Mobile, Xbox, Desktop, and Surface Hub.

The important advantage of the UWP is that even when an extension SDK is referenced by a UWP project, the application can still be deployed to the platforms that do not support the particular extension set. Conditional compilation is not required. However, a programmer still needs to ensure that an application does not access features that are unavailable. To check whether a particular API is available for the current platform, use static methods of the `ApiInformation` class, defined in the `Windows.Foundation.Metadata` namespace. The following code can be used to check if a particular type, `Windows.Phone.Devices.Power.Battery`, is present. It produces `false` when the code is run on the desktop platform and `true` if the app utilizing this code is run on Windows Phone.

```
var typeName = "Windows.Phone.Devices.Power.Battery";  
var canIReadBatteryLevelOfMyWindowsPhone = Windows.Foundation.Metadata.  
    ApiInformation.IsTypePresent(typeName);  
System.Diagnostics.Debug.WriteLine("Can I access a battery level of my Windows  
Phone: " + canIReadBatteryLevelOfMyWindowsPhone);
```

Because of the common, unified programming interfaces available on every Windows 10 device and the abstraction layer that exempts developers from writing their own native drivers (for example, for mapping memory registers), Windows 10 IoT Core also delivers software development kits available for other UWP devices. IoT developers can easily perform various programming tasks, such as creating rich and adaptive user interfaces, handling gesture and voice input, and connecting the device to web and cloud services, just to name a few.

Such an approach has several advantages over other solutions. First, Windows 10 IoT Core programmers can benefit from functionality already implemented within the core part of the Universal Windows Platform. This shortens development time and significantly increases software capabilities. Second, UWP apps can target novel prototype devices, running Windows 10 IoT Core. Finally, an app can be monetized through the same distribution channel (that is, Windows Store).

Windows 10 IoT Core is a great tool for rapid prototype development. However, native solutions can be preferable in some scenarios. That doesn't mean Windows 10 IoT Core and the UWP are not fully functional for IoT development, but that for exceptional cases, e.g., extremely time-critical applications, you'd want to avoid the additional processing time for transferring signals and data between additional layers provided by Windows 10 IoT Core. In such cases, you'd use native tools and reduce usability and flexibility for performance increases.

Tools installation and configuration

Let's make sure you've got all the software tools you'll need for this book. Here are the required elements.

- A development PC with Windows 10 installed and enabled developer mode
- Visual Studio 2015 (Update 1 at least) as the integrated development environment
- Windows IoT Core project templates
- Windows 10 IoT Core Dashboard
- IoT device

Windows 10

Application development for Windows IoT Core requires a PC controlled by Windows 10, version 10.0.10240 or higher. Windows 10 installation is straightforward, and for this reason I don't describe it in detail here.

If Windows 10 is already installed on your development PC, you should verify its version—and upgrade if necessary. To verify the system version run `winver` from a command prompt or by searching for it in the start menu. (See Figure 2-2.)

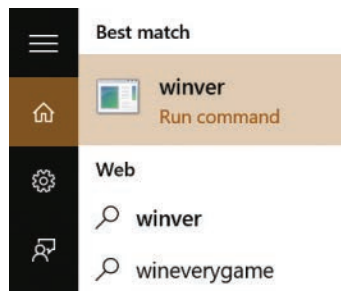


FIGURE 2-2 The `winver` application will tell you the build number of Windows 10 on your machine.

After verifying the Windows version, enable developer mode on the development PC. You can do so by using the Windows Settings application, which can be executed by searching for it in the Start menu (as `winver`). To enable developer mode, go to the **Update & Security** section of the settings and select **Developer Mode** on the **For Developers** tab, as you see in Figure 2-3.

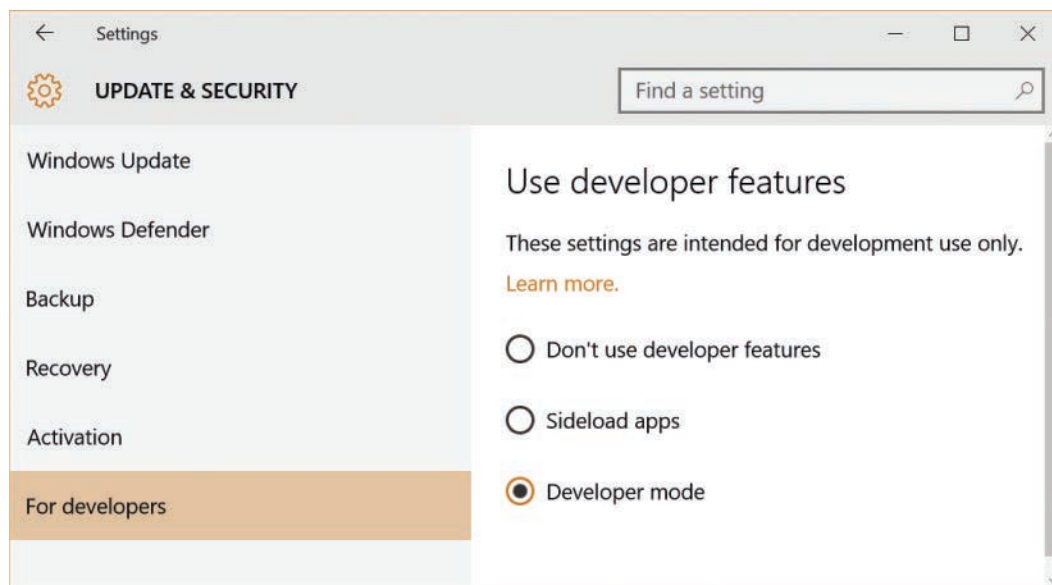


FIGURE 2-3 Enable developer mode in Windows 10 through Update & Security within Settings.

Visual Studio 2015 or later

Your Windows IoT Core development PC should also include Visual Studio 2015 Update 1 (or later) as the integrated development environment. It's available for download at <https://www.visualstudio.com/vs/> in three different versions: Community, Professional, and Enterprise. The first version is free, and the other two require licenses but can be used for free within an evaluation period. In this book, I am using Visual Studio 2015 Community. In Appendix F, "Setting up Visual Studio 2017 for IoT development," I also show how to setup Visual Studio 2017 RC for IoT (UWP) development. Aspects presented in this book are compatible with Visual Studio 2017 RC.

The installation process of Visual Studio 2015 is automatic. However, the SDK for the Universal Windows Platform might not be included in the default Visual Studio 2015 installation. Therefore, during Visual Studio 2015 installation, make sure that the **Tools and Windows 10 SDK** check box under the **Windows and Web Development** node is selected, as shown in Figure 2-4.

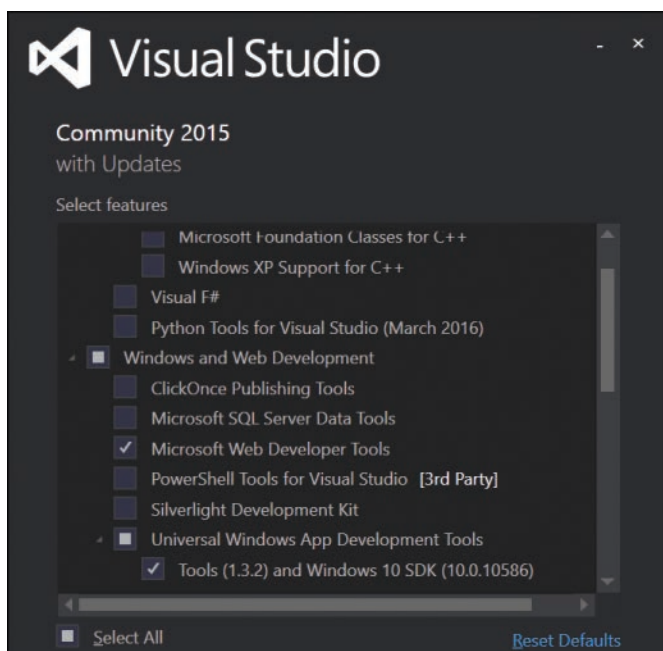


FIGURE 2-4 The Universal Windows App Development Tools are required for Windows IoT Core development. This figure shows the installer of the Visual Studio 2015 Community Update 2 with Universal Windows App Development Tools in version 1.3.2 and the Windows 10 SDK in version 10586.

Windows IoT Core project templates

Microsoft provides additional project templates designed for developing IoT applications without a user interface of any form. These are the so-called headless apps—in contrast to headed apps with the UI. I tell you more about the headless apps in Chapter 3, "Windows IoT programming essentials."

The IoT project templates for headless apps can be installed as an extension to Visual Studio 2015 by following these steps:

1. In Visual Studio 2015, go to **Tools > Extensions and Updates**.
2. In the Extensions and Updates dialog box, expand the **Online** node and in the search box at the top right of the dialog box, type **IoT**.
3. In the search results, find **Windows IoT Core Project Templates**, as shown in Figure 2-5, and click the **Download** button. This will start the download process for the templates.

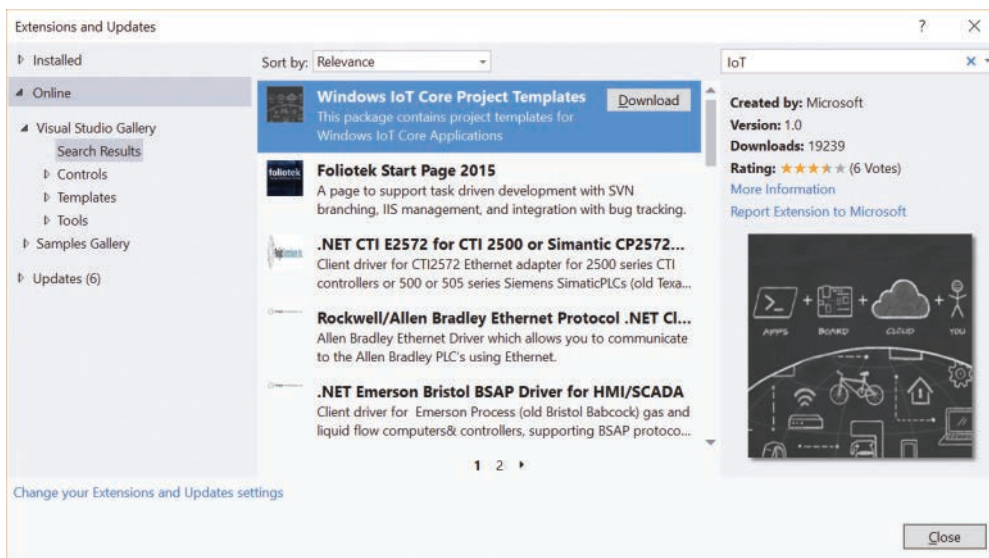


FIGURE 2-5 The Extensions and Updates dialog box in Visual Studio 2015.

4. After the project templates are downloaded, the Visual Studio Extension (VSIX) Installer will display a license terms screen. Click **Install**.
5. The Installer will confirm that the extension has been successfully installed. Click **Close**, and restart Visual Studio 2015.

Windows 10 IoT Core Dashboard

After installing Visual Studio 2015, you need to download and install the Windows 10 IoT Core Dashboard. This application helps set up new—and manage and configure existing—Windows 10 IoT Core devices connected to the local network, available both to the development PC and your IoT devices.

First, download the installer for the dashboard from http://bit.ly/iot_dashboard. Then run the installer file you just downloaded, and click **Install** on the first security warning dialog box that appears. This will start the download and installation process for the dashboard, and you'll see an Installing Windows 10 IoT Core Dashboard dialog box, after which another security warning might be displayed.

(Click **Run** if this second warning appears.) When the dashboard is installed and ready for use, you'll see its welcome screen, shown in Figure 2-6.

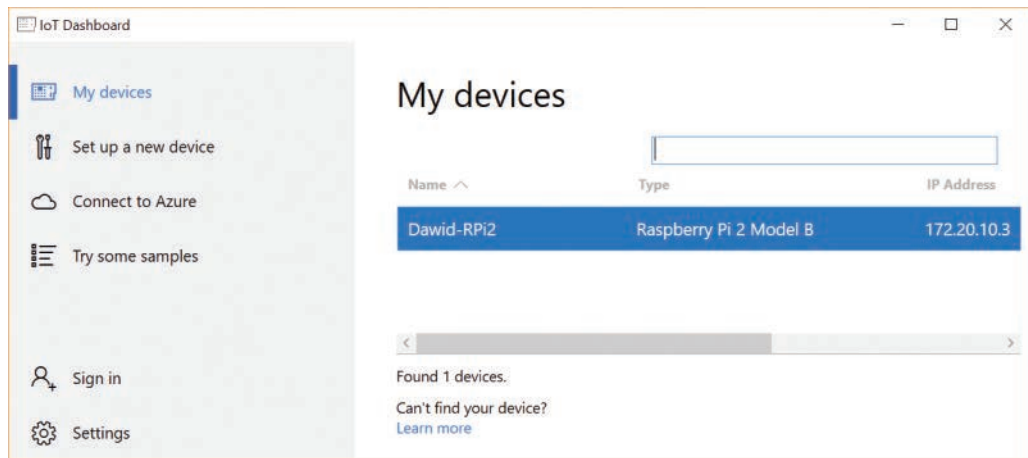


FIGURE 2-6 The Windows 10 IoT Core Dashboard showing the list of discovered IoT devices. Note that the My Devices list will be most likely empty at this stage.

Device setup

Now that you have all the necessary software components for your development environment, let's set up an IoT device.

Windows 10 IoT Core Starter Pack for Raspberry Pi 2 and Pi 3

While I was writing this chapter, Windows 10 IoT Core was available for four development boards (see Table 2-1 for a comparison):

- Raspberry Pi 2 (RPi2)
- Raspberry Pi 3 (RPi3)
- MinnowBoard MAX
- Qualcomm DragonBoard 410c

I decided to use the first of these because it was available within the Starter Pack for Windows 10 IoT, prepared by Adafruit (http://bit.ly/iot_pack). This pack offers a very convenient way to start IoT programming, since it contains all the necessary tools and verified, compatible components to assemble prototype circuits and develop software for Windows IoT devices. This is very important, especially in the early stage of development, because in case of any troubleshooting you can eliminate basic problems related to some of the hardware components like wires, LEDs, or sensors. Of course, you don't have to use the Starter Pack for RPi2. You can always get the RPi2 and other components separately.

However, you need to ensure that your hardware elements are compatible with the RPi2 and Windows 10 IoT Core. You can find the Microsoft-verified list of Windows 10 IoT Core compatible hardware components (like micro SD cards, sensors, and so on) at http://bit.ly/iot_compatibility_list. This remark is especially important when you want to use a different micro SD card. In that case you would need at least a class 10 micro SD card. Note that instead of the Starter Pack for RPi2, you can also use an updated version of this pack containing the RPi3. However, the latter is not equipped with an internal ACT LED. This requires you to use an external LED circuit to run a few sample apps, described in later chapters. Moreover, the Starter Pack for RPi3 does not contain an external Wi-Fi module.

TABLE 2-1 Selected features of development boards supporting Windows 10 IoT Core

Board	Raspberry Pi 2	Raspberry Pi 3	MinnowBoard MAX	Qualcomm DragonBoard 410c
Architecture	ARM	ARM	x64	ARM
CPU	Quad-core ARM® Cortex® A7	Quad-core ARM® Cortex® A8	64-bit Intel® Atom™ E38xx Series SoC	Quad-core ARM® Cortex® A53
RAM	1 GB	1 GB	1 or 2 GB	1 GB
On-board WiFi	-	+	-	+
On-board Bluetooth	-	+	-	+
On-board GPS	-	-	-	+
HDMI	+	+	-	+
USB ports	4	4	2	2
Ethernet port	+	+	+	+

Besides the RPi2 (or RPi3), the Microsoft IoT Pack for Raspberry Pi 2/3, as shown in Figure 2-7, contains the following components, wires and sensors:

- Raspberry Pi 2/3 Case—housing for the Raspberry; detailed instructions for inserting a board into the case can be found at http://bit.ly/rpi_case
- 5V 2A Power Supply with micro-USB cable
- Solderless Breadboard—required for circuit assembly
- Ethernet cable
- USB Wi-Fi module
- 8 GB micro SD card with Windows IoT Core (16 GB card in the case of RPi3)
- Male/Male jumper wires
- Female/Male jumper wires
- Two potentiometers

- Three tactile switches
- Ten resistors
- One capacitor
- Six LEDs
- One photocell
- Temperature and barometric sensor
- Color sensor



FIGURE 2-7 Contents of the Starter Pack for Windows 10 IoT Core on Raspberry Pi 2. A pack containing Raspberry Pi 3 is very similar. Source: <http://www.adafruit.com>.

You will use some of the above components to assemble the circuit, which is composed of the LED and resistor and will be controlled by the Windows universal app. However, before doing that, you will deploy Windows 10 IoT Core to the RPi2 (or RPi3) device and perform its basic configuration. Windows 10 IoT Core is already preloaded to the micro SD card, which comes with the Windows 10 IoT Starter Pack, but I prefer to describe the Windows 10 IoT Core installation and deployment because the IoT Starter Pack may not include the newest build of Windows 10 IoT Core. Also, you may need to deploy Windows 10 IoT Core if your micro SD card needs to be replaced.

Windows 10 IoT Core installation

The easiest way to install Windows 10 IoT Core on the RPi2 (or RPi3) is through the IoT Dashboard. The wizard available on the Set Up a New Device tab of the IoT Dashboard fully automates this process after you insert one of the compatible micro SD cards into the PC's card reader. Subsequently, select the device type, type in your device name (I set it to "Dawid RPi-2") and new administrator password, choose the Wi-Fi network connection (if available), accept the software license terms, and click **Download and Install**. (See Figure 2-8.) The IoT Dashboard will start flashing your SD card. The current progress will be displayed, as shown in Figure 2-9. The Deployment Image Servicing and Management tool will apply the Windows 10 IoT Core image to the SD card. (See Figure 2-10.) The IoT Dashboard will then display the confirmation screen that looks like Figure 2-11.

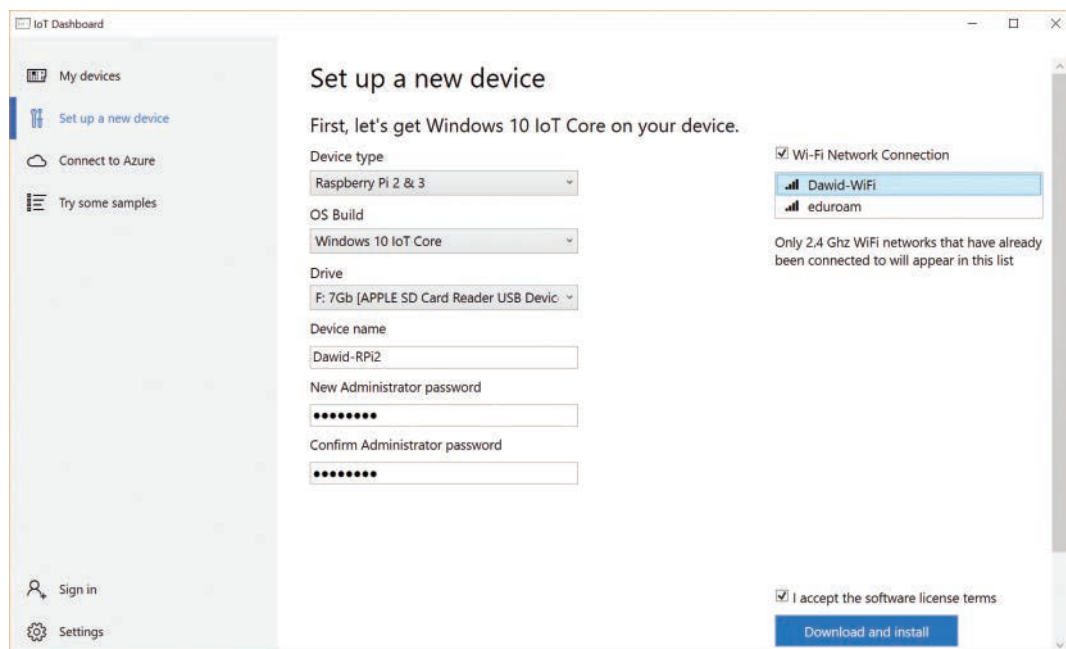


FIGURE 2-8 Setting up a new Windows 10 IoT Core device.

Downloading Windows 10 IoT Core

63 MB downloading - 9%

Cancel

Flashing your SD card

Pending

FIGURE 2-9 SD card preparation.

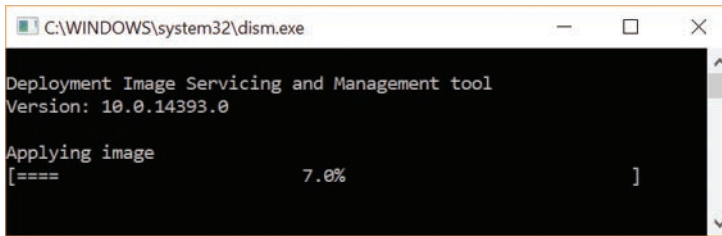


FIGURE 2-10 Deployment Image Servicing and Management tool is applying the Windows 10 IoT Core image to the SD card.

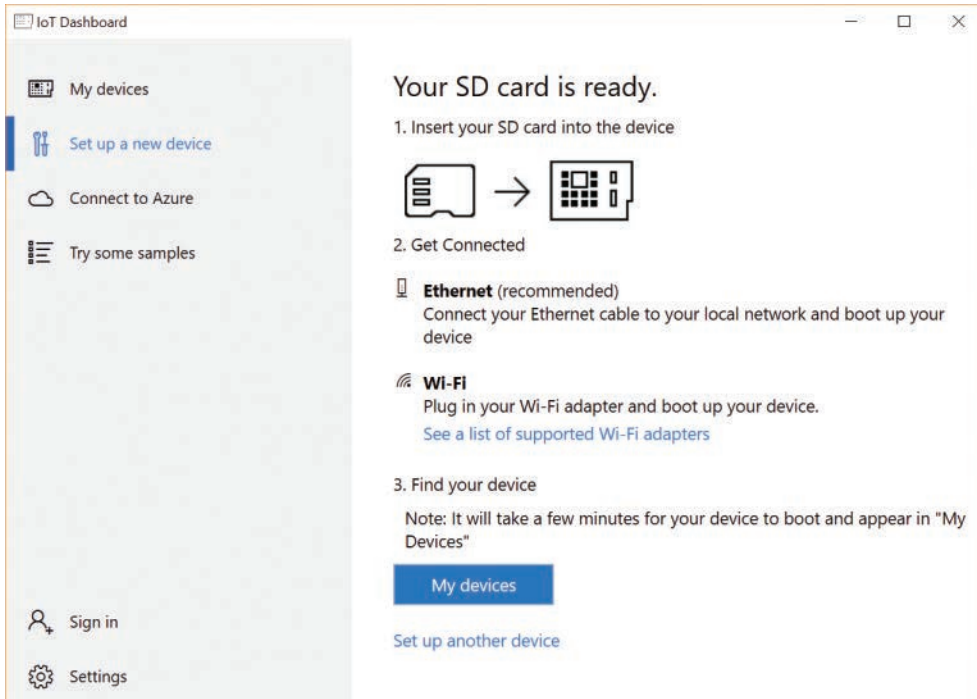


FIGURE 2-11 The IoT Dashboard shows successful image deployment by displaying the confirmation screen with subsequent instructions.

Configuring the development board

You see the top and the bottom views of the Raspberry Pi 2 Model B V 1.1, which are in Figure 2-12 and Figure 2-13, respectively. The main part of the board is the Broadcom BCM2836 chip, integrating the 900 MHz quad-core ARM Cortex-A7 CPU and VideoCore IV 3D graphics core with 1 GB of RAM memory. Raspberry Pi 2 is equipped with the following ports:

- 4 USB type A ports
- 1 micro-B USB port for power purposes

- 1 Local Area Network (LAN) adapter
- 1 HDMI port
- 1 3.5 mm audio jack and composite video (A/V)
- 40 GPIO pins
- 1 micro SD card slot, located at the back surface of the board (see Figure 2-13)

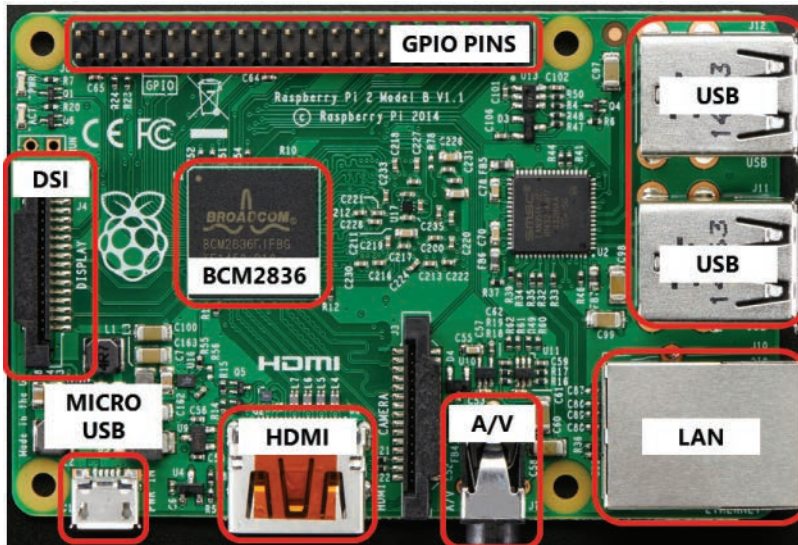


FIGURE 2-12 The top view of the Raspberry Pi 2. The CSI interface is located between HDMI and A/V ports.

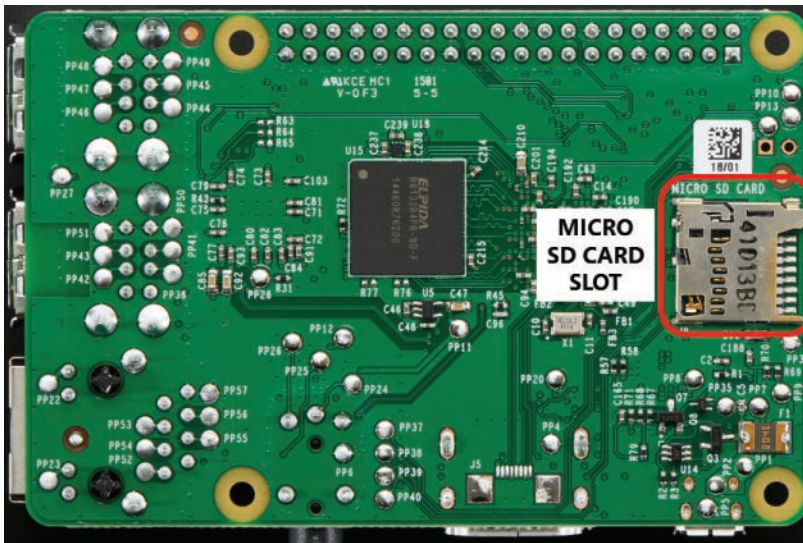


FIGURE 2-13 The bottom view of the Raspberry Pi 2.

The Raspberry Pi 2 and Pi 3 also include a Camera Serial Interface (CSI) and a Display Serial Interface (DSI). However, when I was writing this chapter, the DSI interface was not supported by Windows 10 IoT Core.

In order to prepare and run the RPi2 (or RPi3), complete the following procedure:

1. Insert the micro SD card with Windows 10 IoT Core into the Raspberry Pi 2 SD card slot.
2. Plug in the 5V 2A micro-B USB power supply to the board. The RPi2 will boot Windows 10 IoT Core automatically, which may take a minute or two.
3. Connect the Ethernet cable to the same local network as the development PC or plug the USB Wi-Fi module into one of the Raspberry's type A USB ports.

The RPi2 board is now ready. To proceed, you need to run the IoT Dashboard and go to the **My Devices** tab. The IoT Dashboard will automatically discover available IoT devices as shown in Figure 2-14. Note that if you choose the Wi-Fi connection during installation (refer to Figure 2-8), your IoT device will be automatically connected to this network. Also, be patient when booting your device for the first time. It may take longer. If your IoT device is still unavailable, you need to restart the RPi2 or RPi3 by reconnecting the power supply.

My devices

<div>Search</div>				
Name ^	Type	IP Address	Settings	OS
Dawid-RPi2	Raspberry Pi 2 Model B	172.20.10.3		10.0.14393.0

FIGURE 2-14 The list of discovered IoT devices.

Hello, world! Windows IoT

With all the software tools installed and configured, and the IoT device in place, you can now write your first embedded UWP application: a “Hello, world!” app that toggles the LED connected to the RPi2 (or RPi3) through the electronic circuit.

Circuit assembly

To control an LED, you first need to assemble an electrical circuit. You can do so by connecting one of the LEDs delivered with the Windows 10 IoT Core Starter Pack for RPi2 (or RPi3) to the appropriate GPIO pins of the IoT device. For development boards, physical pins of the microcontroller are typically available through the pin expansion header to simplify solderless pin connection. (Refer to Chapter 1, “Embedded devices programming,” and Figure 1-3 for more information.) The expansion header is physically connected to microcontroller pins by traces on the printed circuit board (PCB). You can visually inspect these traces by analyzing the RPi2 board or an appropriate PCB design.

LED, resistor, and electronic color codes

Every LED has a specified operating current required to power the LED, but you need to be careful to not exceed the maximum threshold value lest you destroy the LED. Input current to the LED circuit is regulated by a resistor. The recommended resistance is usually described in the LED manufacturer's datasheet.

The LEDs available within the Windows 10 IoT Core Starter Pack are low power, so you'll use a 560 Ohm (Ω) resistor to limit the current flow. The resistors contained in the Starter Pack for Windows 10 IoT Core are encoded using the 4-digit color code, consisting of four vertical color stripes, called bands. The first three bands encode the actual resistance, and the last one, which is usually spaced from the other bands, denotes the resistance tolerance, i.e., any deviation from the value declared by the manufacturer.

The resistance value is encoded using two significant figures (first and second band) and the multiplier (third band). Table 2-2 shows you the meaning of each band color. The resistor in your Starter Pack with color code green-blue-brown-gold has a resistance of $R = 56 \times 10^1 = 560 \Omega$ with a tolerance ΔR of $\pm 5\%$. Similarly, the color code brown-black-orange-gold represents the following values: $R = 10 \times 10^3 = 10 \text{ k}\Omega$ and $\Delta R = \pm 5\%$. For more details, see http://bit.ly/electronic_color_code.

TABLE 2-2 Electronic resistance color codes

Color	Significant figure value	Multiplier	Tolerance
Black	0	10^0	Does not apply
Brown	1	10^1	$\pm 1\%$
Red	2	10^2	$\pm 2\%$
Orange	3	10^3	Does not apply
Yellow	4	10^4	$\pm 5\%$
Green	5	10^5	$\pm 0.5\%$
Blue	6	10^6	$\pm 0.25\%$
Violet	7	10^7	$\pm 0.1\%$
Gray	8	10^8	$\pm 0.05\%$
White	9	10^9	Does not apply
Gold	Does not apply	10^{-1}	$\pm 5\%$
Silver	Does not apply	10^{-2}	$\pm 10\%$
None	Does not apply	Does not apply	$\pm 20\%$

The longer leg of an LED is called anode (positive charge), and the shorter leg is a cathode (negative charge). Because the electric charge flows from the positive to the negative leg, the resistor should be connected to the longer leg of the LED. You subsequently connect the resistor and the shorter LED leg to the GPIO pins of the RPi2. You can use either of two configurations for controlling the LED using a microcontroller: the logical active-low and active-high states.

Active-low and active-high states

The GPIO pins can possess the logical (digital) values of 0 (or low) or 1 (or high). The first one typically corresponds to a voltage of 0 V or less, while the second represents a voltage level above some threshold. In practice, an analog voltage signal is susceptible to noise. Accordingly, a signal randomly oscillates around low or high values. To ensure that an analog voltage signal represents valid low and valid high logic levels, pull-down (for low state) and pull-up (for high state) resistors are used. Thus, you have two options to induce the carrier flow through the LED:

- **Active-low state** Connect the longer LED leg through the resistor to the power supply pin with the voltage of 3.3 V and the second LED leg to the GPIO pin. When the GPIO pin is driven to a low state, current flows from the power pin through the resistor to the LED.
- **Active-high state** Connect the LED's cathode to the ground (GND), and the anode to a GPIO pin. A GPIO pin driven into a high state induces the carrier flow.

Raspberry Pi 2 pinout

Before you configure an LED circuit in the active-low or active-high state, you need to familiarize yourself with the RPi2 pinout, which exposes the peripherals through a 40-pin expansion header; refer to Figure 2-12 and see Figure 2-15. Each pin of this header is numbered, starting from 1. Pins with odd numbers are located in the top header row. This means that the pin number 1 is located in the bottom left corner of the header (first element in the second header row), while the pin number 2 is located right above pin number 1. Consequently, the 40th pin sits in the far right end of the first header row.

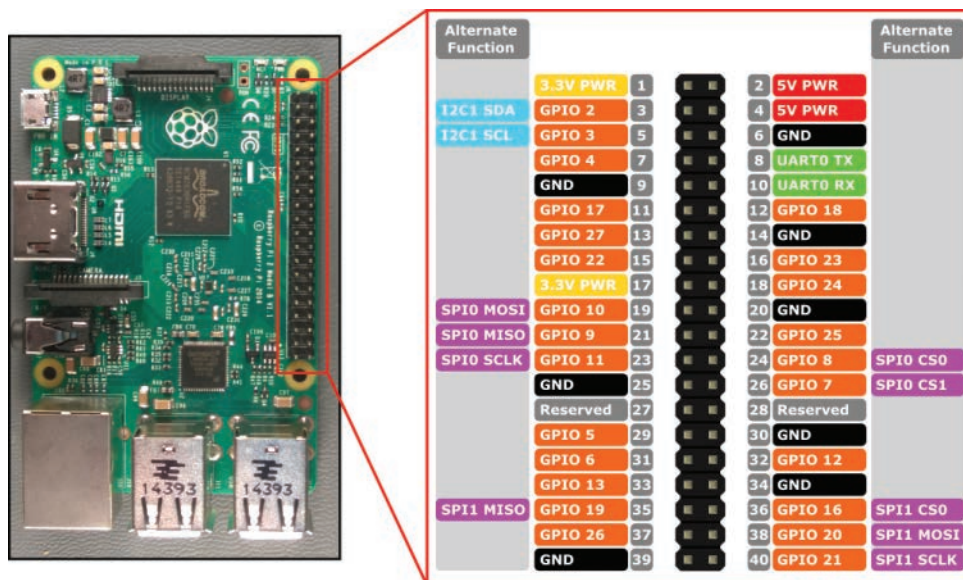


FIGURE 2-15 Pin mappings of the Raspberry Pi 2. Source: Windows Dev Center (<http://windowsondevices.com>). You can find an interactive version of this diagram at <http://pinout.xyz/>.

All the physical pins of the Raspberry Pi 2 can be divided into six groups, assigned to the following expansion header pins:

- 3.3V power pins: 1, 17
- 5V power pins: 2, 4
- I²C bus pins: 3, 5
- Ground (GND) pins: 6, 9, 14, 20, 25, 30, 34, and 39
- SPI bus pins: 19, 21, 23, 24, and 26
- Manufacturer-reserved pins: 8, 10, 27 and 28

Additionally, 17 GPIO pins are available to the user. Table 2-3 summarizes their numbers and initial states (right after the boot).

TABLE 2-3 GPIO ports assignment of the Raspberry Pi 2

Header pin	GPIO number	Initial state	Header pin	GPIO number	Initial state
7	4	Pull up	31	6	Pull up
11	17	Pull down	32	12	Pull down
12	18	Pull down	33	13	Pull down
13	27	Pull down	35	19	Pull down
15	22	Pull down	36	16	Pull down
16	23	Pull down	37	26	Pull down
18	24	Pull down	38	20	Pull down
12	25	Pull down	40	21	Pull down
29	5	Pull up			

Two additional GPIO ports, 35 and 47, control the status of two LEDs located on the RPi2 board. These LEDs are located above the DSI interface of the RPi2. (See Figure 2-12.) The GPIO port 35 controls the red power LED (PWR), while the second controls the green LED (ACT). In this chapter, I use an external LED only. Note that the ACT LED is unavailable on the RPi3.

As Figure 2-15 shows, several pins can have an alternate function. For example, header pins 2 and 5 (GPIO 2 and 3, respectively) can also provide access to the I²C interface. The section “Using C# and C++ to turn the LED on and off” delves into this issue.

According to the RPi2 pinout, the active-low state can be assembled by wiring the shorter leg of an external LED to pin 29 on the expansion header (GPIO 5) and connecting the longer LED through the resistor to pin 1 (3.3 V power supply). In the active-high state, you connect the cathode to a GPIO pin.

Note that the LED can be powered without writing the actual software. You can connect the second LED leg to one of the GND pins. The LED will be turned on immediately, and the RPi2 will simply act as a 3.3 V battery.

In the active-high configuration, the shorter leg of the LED can be connected to ground, e.g., header pin 6. The longer LED leg is then connected to the resistor and the GPIO pin, with an initial state of 0 (pull down). Then the LED is powered by driving the selected GPIO port to an active-high state.

Solderless breadboard connection

The Starter Pack for Windows 10 IoT Core contains the breadboard, which supports prototype wiring of electronic elements and does not require soldering. This breadboard is composed of two power rails on both sides and also contains 610 tie points, arranged in an array; see the bottom part of Figure 2-7. Each row of this array is labeled by a capital letter A through J, and the columns are numbered starting from 0. These labels help to localize tie points on the breadboard.

To assemble the LED circuit in an active-low state, you bend both legs of the resistor and use two female/male jumper cables. Figure 2-16 shows a connection diagram, which I made using the open-source tool Fritzing. (Download it at fritzing.org.) Table 2-4 shows you the breadboard-header map, and Figure 2-17 shows the actual connection.

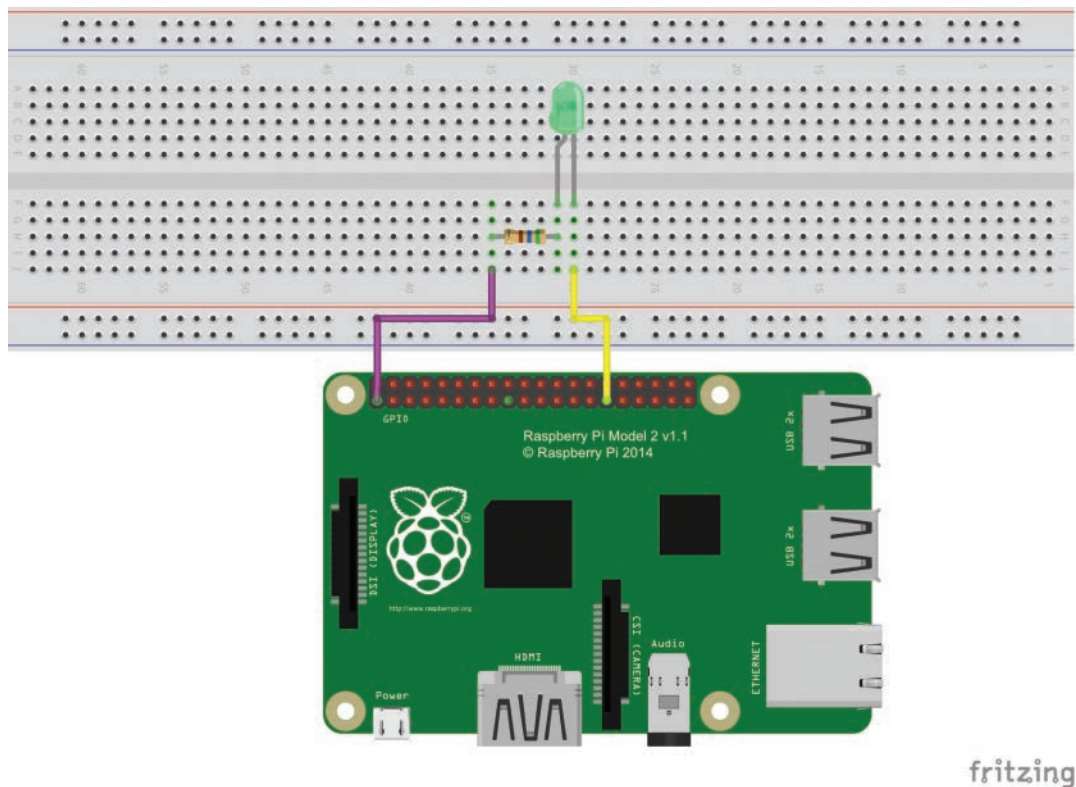


FIGURE 2-16 An active-low state LED circuit visualization using a fritzing diagram.

TABLE 2-4 Sample connection map for an active-low LED circuit

Component	Leg or connector	Breadboard tie point location	Header pin
LED	Shorter leg	Row: F, Column: 30	-
	Longer leg	Row: F, Column: 31	-
Resistor	First leg	Row: H, Column: 31	-
	Second leg	Row: H, Column: 35	-
First jumper cable (purple in Figure 2-16 and in Figure 2-17)	Male connector	Row: J, Column: 35	-
	Female connector	-	1 (or 17)
Second jumper cable (yellow in Figure 2-16 and in Figure 2-17)	Male connector	Row: J, Column: 30	-
	Female connector	-	29 (or pins 7, 31 for pull-up initial state)

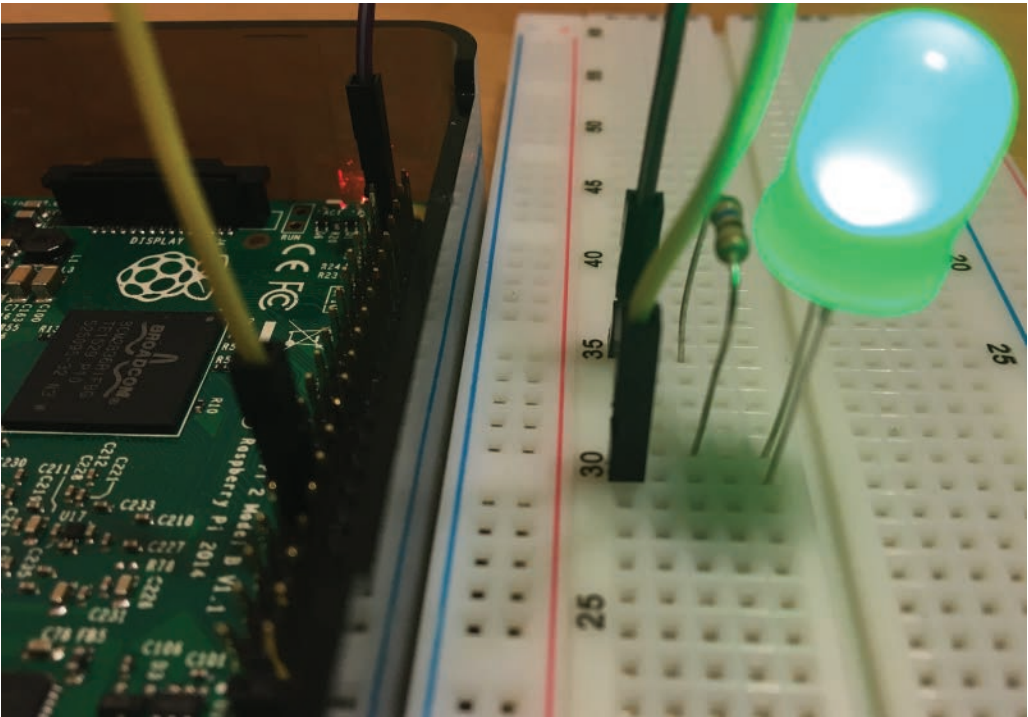


FIGURE 2-17 Real assembly of an active-low state LED circuit.

Typically, for the active-low state, you want to choose the GPIO pin, which is initially in a pull-up state. This ensures that the current flow is disabled when you access the port. Conversely, in the active-high state, you use the GPIO ports, which are in a pull-down state when the IoT device is powered up and the carrier’s flow is disabled. Such alternative (active-high) circuit assembly can be configured as shown in Figure 2-18, Table 2-5, and Figure 2-19. Compare this configuration with an active-low state.

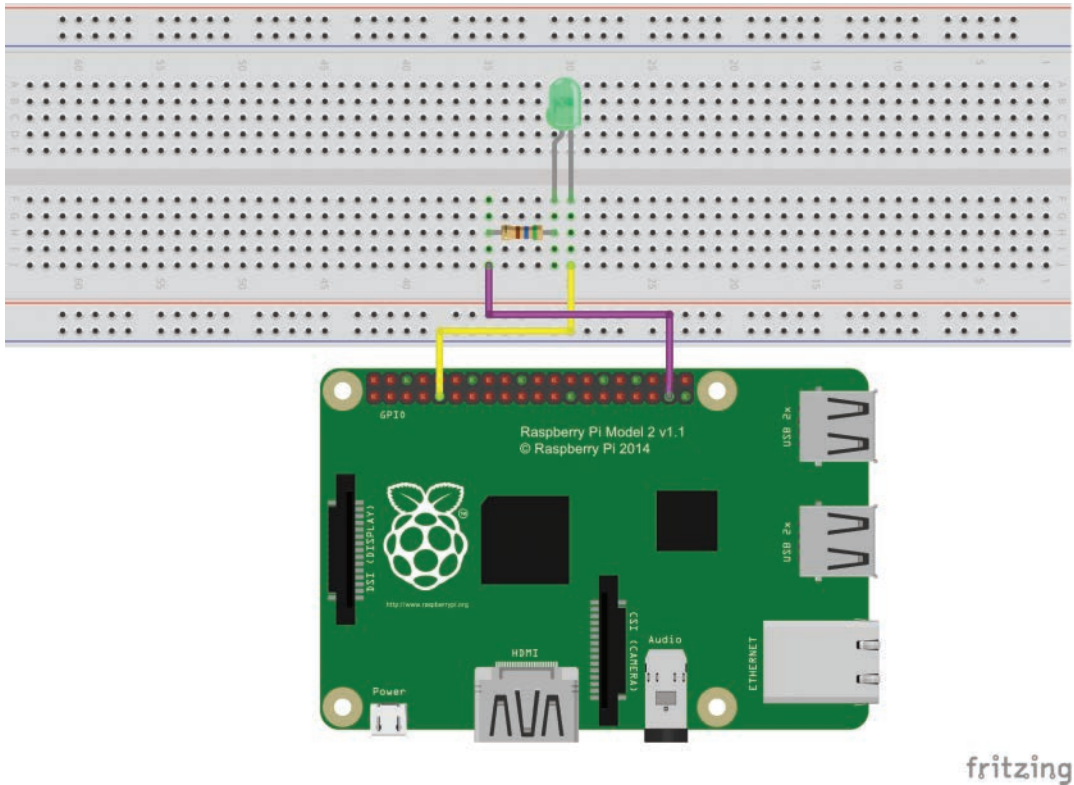


FIGURE 2-18 An active-high state LED circuit visualization using a fritzing diagram (compare with Figure 2-16).

TABLE 2-5 Sample connection map for an active-high LED circuit

Component	Leg or connector	Breadboard tie point location	Header pin
LED	Shorter leg	Row: F, Column: 30	-
	Longer leg	Row: F, Column: 31	-
Resistor	First leg	Row: H, Column: 31	-
	Second leg	Row: H, Column: 35	-
First jumper cable (purple in Figure 2-18 and in Figure 2-19)	Male connector	Row: J, Column: 35	-
	Female connector	-	37 (or any other GPIO port with pull-down initial state, e.g., pins 11, 12)
Second jumper cable (yellow in Figure 2-18 and in Figure 2-19)	Male connector	Row: J, Column: 30	-
	Female connector	-	9 (or any other GND, e.g., pins 6, 14)

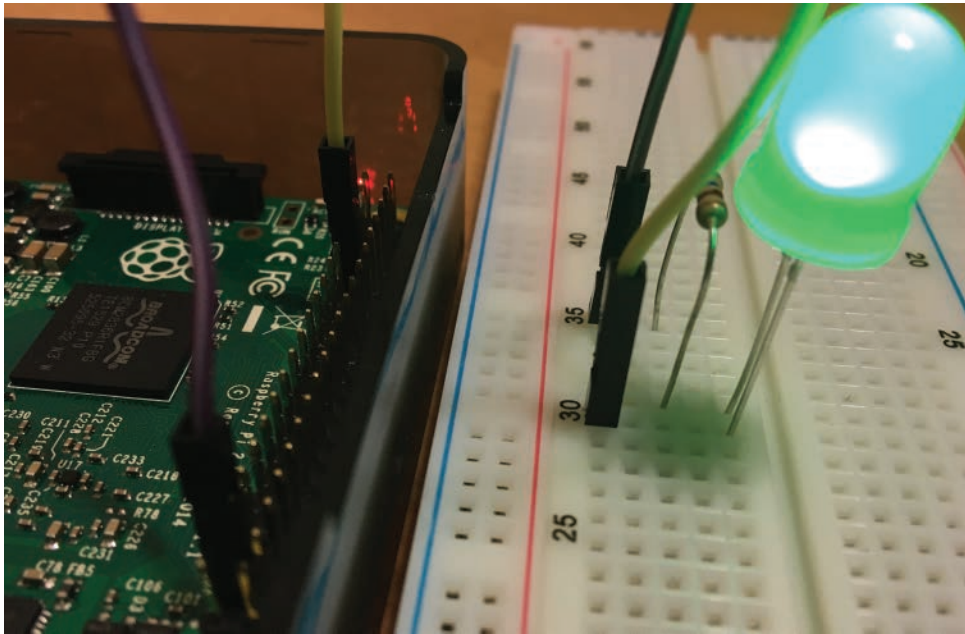


FIGURE 2-19 Real assembly of an active-high state LED circuit.

Using C# and C++ to turn the LED on and off

You are now ready to write the UWP app that will power an LED that's connected in the active-low configuration. You can accomplish this task with any of several programming models available on the UWP. You can implement the logic layer of the application by using any of the following:

- C#
- C++
- Visual Basic
- JavaScript

Depending on the language, the user interface can be declared using XAML for C#, C++, and Visual Basic or using HTML/CSS for JavaScript. In this section I use C# and C++. Examples for Visual Basic and JavaScript can be found in Appendix A, "Code examples for controlling LED using Visual Basic and JavaScript."

C#/XAML

Follow these steps to write the first UWP app for the Windows IoT device using C#/XAML programming languages:

1. Open VS 2015 (or later) and go to **File > New > Project**.

2. In the new New Project dialog box:
 - a. Type **Visual C#** in the search box, as shown in Figure 2-20.

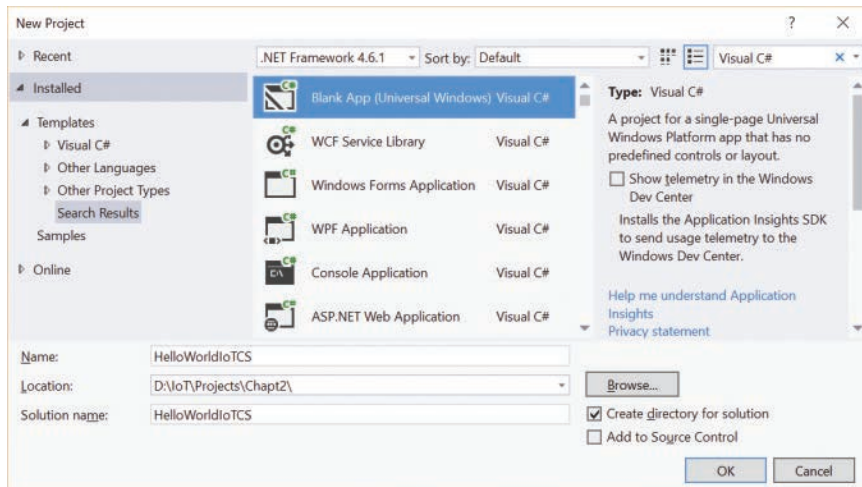


FIGURE 2-20 A New Project dialog box of Visual Studio 2015. The Blank App (Universal Windows) project template for Visual C# is selected.

- b. Select the **Visual C# Blank App (Universal Windows)** project template.
 - c. Change the project name to **HelloWorldIoTCS** and click the **OK** button.
 - d. In the New Universal Windows Project dialog box, set **Target Version** and **Minimum Version** to **Windows 10 (10.0; Build 10586)**. (See Figure 2-21.) The new blank project has been created.

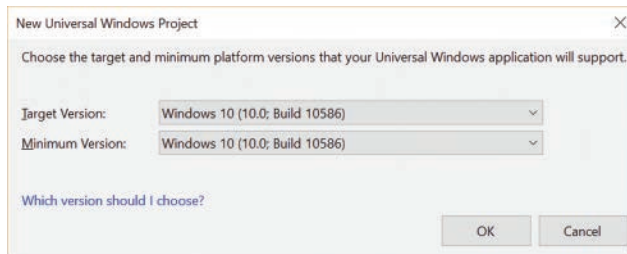


FIGURE 2-21 The New Universal Windows Project dialog box of Visual Studio 2015 lets you configure the target and minimum supported version of Windows 10.

Target and minimum platform versions

Target platform versions specify the UWP API available to your app. The higher the value, the more updated API you can use. Similarly, the minimum platform version specifies the minimum UWP version on which your app can run.



Note In subsequent chapters, if not stated otherwise, I will set the target version to Windows 10 (10.0; Build 10586).

3. Open the Solution Explorer by clicking **View > Solution Explorer**.
4. In the Solution Explorer, expand the **HelloWorldIoTCS** node, and then right-click the **References** option. From the context menu select **Add Reference**. A Reference Manager window appears.
5. In the Reference Manager window, go to the **Universal Windows** tab and then click the **Extensions** tab.
6. Select the **Windows IoT Extensions for the UWP** check box, as shown in Figure 2-22, and then close Reference Manager by clicking the **OK** button.

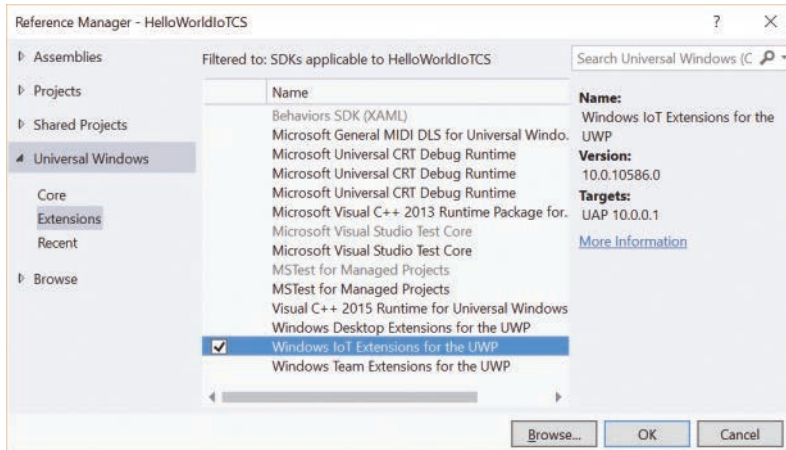


FIGURE 2-22 A Reference Manager of the HelloWorldIoTCS project. The Windows IoT Extensions for the UWP check box is selected.

7. Using the Solution Explorer, open the **MainPage.xaml.cs** file and modify its contents according to Listing 2-1.

LISTING 2-1 An LED is driven using GpioController

```
using System.Threading.Tasks;
using Windows.Devices.Gpio;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace HelloWorldIoTCS
{
```

```

public sealed partial class MainPage : Page
{
    private const int gpioPinNumber = 5;
    private const int msShineDuration = 5000;

    public MainPage()
    {
        InitializeComponent();
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        base.OnNavigatedTo(e);

        BlinkLed(gpioPinNumber, msShineDuration);
    }

    private GpioPin ConfigureGpioPin(int pinNumber)
    {
        var gpioController = GpioController.Default();

        GpioPin pin = null;
        if (gpioController != null)
        {
            pin = gpioController.OpenPin(pinNumber);
            if (pin != null)
            {
                pin.SetDriveMode(GpioPinDriveMode.Output);
            }
        }

        return pin;
    }

    private void BlinkLed(int gpioPinNumber, int msShineDuration)
    {
        GpioPin ledGpioPin = ConfigureGpioPin(gpioPinNumber);

        if (ledGpioPin != null)
        {
            ledGpioPin.Write(GpioPinValue.Low);

            Task.Delay(msShineDuration).Wait();

            ledGpioPin.Write(GpioPinValue.High);
        }
    }
}

```

8. Under the **Project** menu go to **HelloWorldIoTCS Properties**.

9. In the HelloWorldIoTCS Properties dialog box, go to the **Debug** tab (see Figure 2-23) and do the following:
 - a. From the **Platform** drop-down list, select **ARM**.
 - b. In the **Start Options** group, select **Remote Machine** from a **Target Device** drop-down list, and then click the **Find** button. Your IoT device will appear under the Auto Detected expander as shown in Figure 2-24. If it does not appear, you need to provide its name or IP address manually. You can obtain these values through the Windows 10 IoT Core Dashboard. (See Figure 2-14.)
 - c. Click the **Select** button and close the project properties window.

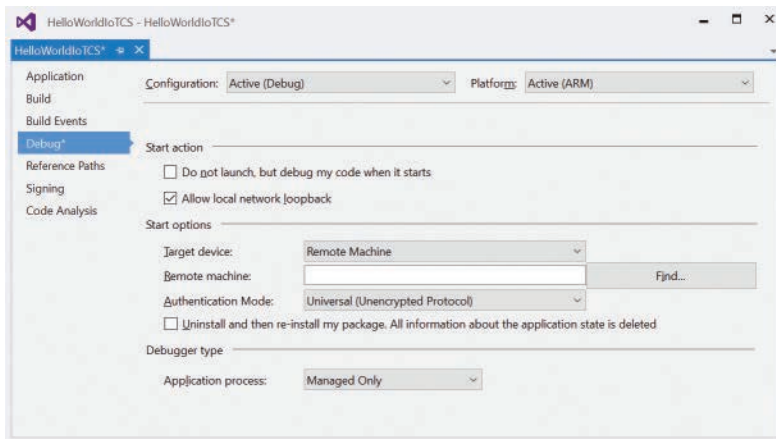


FIGURE 2-23 The Debug tab of the project properties window. Note that you need to select Active (ARM) from the Platform drop-down list.

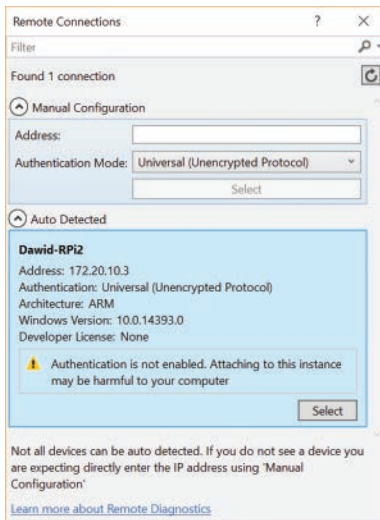


FIGURE 2-24 IoT device discovery.

10. Above Visual Studio 2015, locate the configuration toolbar, which is shown in Figure 2-25. Using the drop-down list, set the configuration to **Debug**, the platform to **ARM**, and the debugging target to **Remote Machine**.



FIGURE 2-25 Configuration toolbar.

11. Run the app. Use the **Start Debugging** option of the **Debug** menu or click the **Remote Machine** button in the configuration toolbar.

After you perform the above procedure, the UWP app will be automatically deployed to the RPi2 device and then executed. Subsequently, the LED will shine for 5 seconds. You can break the application execution at any time by clicking **Debug > Stop Debugging**.

Several aspects of the above solution require additional attention. Notice, first, that the `HelloWorldIoTCS` project is composed of the following elements:

- **project.json** This file specifies project dependencies, frameworks, and runtimes as JSON objects. Each entry of dependencies comprises the name-version pair of the NuGet package. By default, there is only one such package: `Microsoft.NETCore.UniversalWindowsPlatform`. Under the frameworks collection you specify the frameworks that your project targets. For the UWP, you use the `uap10.0` framework (the Universal App Platform). Runtimes contain the list of runtime identifiers (RIDs). In general, you can use any value specified here: <http://bit.ly/runtimes>. But for the UWP project template we used, there are only Windows 10 RIDs, and you do not need other RIDs to complete examples developed in this book. You will need other RIDs when developing cross-platform .NET Core apps—for example, ASP.NET Core MVC web apps or web services.
- **Package.appxmanifest** This is an application manifest file. This XML file contains the information necessary to publish, display, and update an application, and it defines the capabilities, functionality, and application requirements.
- **Assets folder** This contains the project assets.
- **App.xaml and App.xaml.cs files** These implement the App class.
- **MainPage.xaml and MainPage.xaml.cs** These implement the main (default) view of the application.

The default implementation of the App class, generated automatically, displays the view implemented in the `MainPage` class. The `MainPage` class derives from the `Page` class and implements the default application view using two files: `MainPage.xaml` and `MainPage.xaml.cs`. `MainPage.xaml` declares the user interface, while `MainPage.xaml.cs` implements the logic associated with the view. Hence, `MainPage.xaml.cs` is commonly referred to as the *code-behind*.

Chapter 3, “Windows IoT programming essentials,” explains a mechanism of navigation between views and an entry point of the UWP IoT apps. For now, I turn your attention to the `MainPage.xaml.cs` only, because the application logic is included in this file only. Moreover, the current application implements an empty UI, which does not contain any visual elements.

Within the `MainPage` class, given in Listing 2-1, I declared two constant fields: `gpioPinNumber` and `msShineDuration`. The first one defines the pin number of the GPIO port used to control the LED state, while the second determines how long an LED will shine. In this example I assume that the LED circuit is assembled according to Table 2-4. Therefore, the `gpioPinNumber` was assigned the value of 5.

The procedures responsible for shining the LED are implemented within the `BlinkLed` method. This is called under the overridden implementation of the `Page.OnNavigatedTo` event handler. In the `BlinkLed` method two logical parts can be distinguished. The first one invokes the `ConfigureGpioPin` method, which acquires the reference to the default GPIO controller of the IoT device. An abstract representation of this object is the `Windows.Devices.GpioController` class.

The `GpioController` class exposes several members, designed to simplify interfacing the GPIO peripherals. In particular, the static method `GetDefault` returns the default GPIO controller of the embedded device. I use this method in Listing 2-1 to get an access to the RPi2 GPIO controller. After obtaining an instance of the `GpioController` class representing a default GPIO controller, the selected GPIO port is opened by calling an `OpenPin` method. A successful call to this method returns an instance of the `GpioPin` class, being an abstract representation of the GPIO port.

The most general version of the `OpenPin` method accepts two input arguments. The first one (`pinNumber`) indicates the GPIO pin number, while the second (`sharingMode`) defines the sharing mode of the GPIO port. This sharing mode is defined by one of the values of the `Windows.Devices.Gpio.GpioSharingMode` enumeration. Namely, this type exposes two values: `Exclusive` and `SharedReadOnly`. In an exclusive mode, the programmer may either write to or read from the GPIO port, while in the second case, the write operations are not allowed. In the shared mode, the programmer may use several instances referencing the same GPIO port. This is impossible if the GPIO pin is accessed in an exclusive mode. In such a case, subsequent attempts to open the GPIO port will cause an exception.

The second version of the `OpenPin` method, used in Listing 2-1, expects the GPIO pin number only, and opens the GPIO pin in an exclusive mode.

The instance method `SetDriveMode` of the `GpioPin` class is subsequently used to switch the GPIO to the output mode. The available GPIO drive modes are represented by values implemented within the `GpioPinDriveMode` enumeration.

After configuring the GPIO drive mode, all you need to do is to set the GPIO port to the low state. This will induce the current flow through the LED circuit. The LED circuit will be powered as long as the control GPIO pin is set to the high state. This happens automatically after a delay, specified using the `msShineDuration` member of the `MainPage` class. The delay is implemented using the `Delay` static method of the `Task` class, which I tell you more about in Chapter 3.

When the LED is connected to the RPi2 in an active-high state, the above procedure progresses differently. Namely, to turn on an LED you drive the GPIO pin to the high state and subsequently write a low value to disable current flow. The `BlinkLed` method takes the form from Listing 2-2, and according to Table 2-3, you would need to update the value of `gpioPinNumber` to 26.

LISTING 2-2 Blinking an LED connected using an active-high configuration. Note that the `GpioPinValue.High` and `GpioPinValue.Low` states are reversed from Listing 2-1.

```
private void BlinkLed(int gpioPinNumber, int msShineDuration)
{
    GpioPin ledGpioPin = ConfigureGpioPin(gpioPinNumber);

    if(ledGpioPin != null)
    {
        ledGpioPin.Write(GpioPinValue.High);

        Task.Delay(msShineDuration).Wait();

        ledGpioPin.Write(GpioPinValue.Low);
    }
}
```

By default, the GPIO ports available to the user are either in the pull-up or pull-down input mode; see Table 2-3. These modes correspond to the logically active (pull-up) or inactive (pull-down) input GPIO ports, which are represented as `GpioPinDriveMode.InputPullUp` and `GpioPinDriveMode.InputPullDown`, respectively.

When you configure a physical pin with alternate functions as the GPIO, then you cannot access these alternate functions without releasing the instance of the `GpioPin` by calling a `Dispose` method.

A final note is devoted to the Windows IoT extensions for the UWP. By referencing them, you get access to a GPIO-specific—or more generally, an IoT-specific—API of the UWP. If you right-click Windows IoT Extensions for the UWP entry in the Solution Explorer, you will find the location of this SDK. In this case, the default Visual Studio installation, the Windows IoT extensions for the UWP 10.0.10586 reside in the following folder:

`%ProgramFiles(x86)%\Windows Kits\10\Extension SDKs\WindowsIoT\10.0.10586.0`

After opening this folder, you will find an `Include\winrt` subfolder. It contains the set of generated interface definition language (IDL) and C++ header files. For instance, `windows.devices.gpio.idl` and `windows.devices.gpio.h` are provided to access low-level OS features for interfacing GPIO, which we implicitly used in the preceding example. This analysis shows that you can use C++ to access low-level Windows APIs. The next section shows how C++ can be used to implement the LED blinking functionality, where a delay is implemented using the `Sleep` function of the Windows API. Moreover, Chapter 8, “Image processing,” shows how you can use C++ to implement Windows Runtime Components for native code interfacing. You can find a detailed discussion of C++ and Windows Runtime Components in articles by Kenny Kerr in the MSDN magazine at http://bit.ly/cpp_winrt.

C++/XAML

In this section I will show you how to implement the C++ application, which controls the LED circuit. Follow these steps:

1. Create a new project using VS 2015 by going to **File > New > Project**.
2. In the New Project dialog box (see Figure 2-26):
 - a. Type **Visual C++** in the search box.
 - b. Select the **Blank App (Universal Windows)** project template and set **Target** and **Minimum Versions** to **Windows 10 (10.0; Build 10586)**. (See Figure 2-21.)
 - c. Change the project name to **HelloWorldIoTCpp** and click the **OK** button.

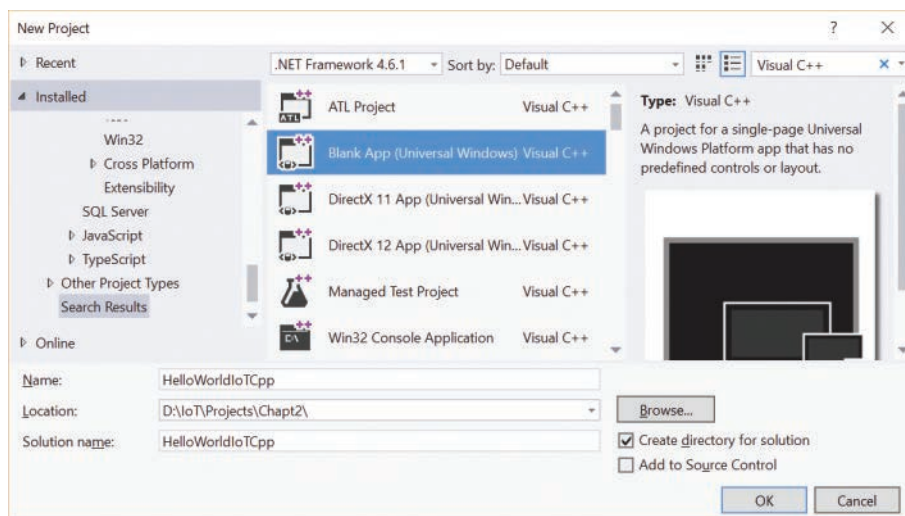


FIGURE 2-26 The New Project dialog box of Visual Studio 2015. The Blank App (Universal Windows) for Visual C++ project template is highlighted.

3. Add a reference to **Windows IoT Extensions for the UWP**. You can do that just as you did in the previous section. (See Figure 2-22.)
4. Modify **MainPage.xaml.h** according to the code snippet in Listing 2-3.

LISTING 2-3 MainPage class declaration

```
#pragma once

#include "MainPage.g.h"

using namespace Windows::UI::Xaml::Navigation;
using namespace Windows::Devices::Gpio;

namespace HelloWorldIoTCpp
```

```

{
    public ref class MainPage sealed
    {
    public:
        MainPage();

    protected:
        void OnNavigatedTo(NavigationEventArgs ^e) override;

    private:
        const int pinNumber = 5;
        const int msShineDuration = 2000;

        GpioPin ^ConfigureGpioPin(int pinNumber);
        void BlinkLed(int ledPinNumber, int msShineDuration);
    };
}

```

5. In the MainPage.xaml.cpp file, insert the code block from Listing 2-4.

LISTING 2-4 MainPage implementation

```

#include "pch.h"
#include "MainPage.xaml.h"

using namespace HelloWorldIoTCpp;
using namespace Platform;

MainPage::MainPage()
{
    InitializeComponent();
}

void MainPage::OnNavigatedTo(NavigationEventArgs ^e)
{
    __super::OnNavigatedTo(e);

    BlinkLed(pinNumber, msShineDuration);
}

GpioPin ^MainPage::ConfigureGpioPin(int pinNumber)
{
    auto gpioController = GpioController::GetDefault();

    GpioPin ^pin = nullptr;

    if (gpioController != nullptr)
    {
        pin = gpioController->OpenPin(pinNumber);
    }
}

```

```

        if (pin != nullptr)
        {
            pin->SetDriveMode(GpioPinDriveMode::Output);
        }
    }

    return pin;
}

void MainPage::BlinkLed(int ledPinNumber, int msShineDuration)
{
    GpioPin ^ledGpioPin = ConfigureGpioPin(ledPinNumber);

    if (ledGpioPin != nullptr)
    {
        ledGpioPin->Write(GpioPinValue::Low);

        Sleep(msShineDuration);

        ledGpioPin->Write(GpioPinValue::High);
    }
}

```

6. Compile and deploy an app to the IoT device:
 - a. Open the HelloWorldIoTcpp properties window and navigate to the **Debugging** tab under the **Configuration Properties** node.
 - b. Select **ARM** from the **Platform** drop-down list.
 - c. Choose **Remote Machine** from the **Debugger** tab to launch a drop-down list.
 - d. Change **Authentication Type** to **Universal (Unencrypted Protocol)**, and find your IoT device using the **<Locate...>** option under the **Machine Name** drop-down list. (See Figure 2-27.)
 - e. Click the **Apply** button and close the project properties window.
7. Run the app.

As in the previous section, the app will automatically deploy to the IoT device and execute. The application implements the same functionality—i.e., it shines the LED for a specified amount of time, determined by the value of the `msShineDuration` member; see Listing 2-3.

The main difference between C++ and C# implementations is that the default application view, i.e. `MainPage`, is now implemented within three (C++) instead of just two (C#) files. Namely, a C++ project includes `MainPage.xaml`, `MainPage.xaml.h`, and `MainPage.xaml.cpp`. The first file, `MainPage.xaml`, defines the UI, while the other two implement the logic (code-behind). The header file, `MainPage.xaml.h`, contains the declaration of the `MainPage` class, whose definition is stored in `MainPage.xaml.cpp`.

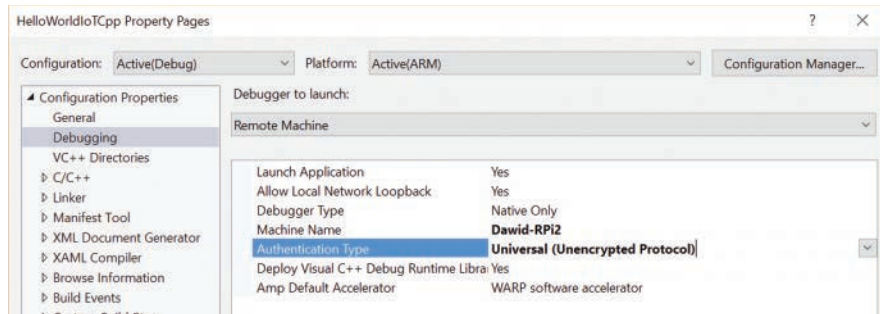


FIGURE 2-27 The Remote Machine configuration for a C++ Universal Windows project.

Additional symbols (e.g. ^), which are related to the Component Extensions (CX) of the C++ language, allow access to the objects from the Universal Windows Platform programming interface and are explained in Appendix E, “Visual C++ component extensions.”

By using C++ as the programming language, you get access not only to the UWP API but also to the low-level Windows API. In particular, in Listing 2-4, to implement a delay between subsequent calls to the method `Write` of the `GpioPin` class, I used the `Sleep` function, declared in the Windows API.

Useful tools and utilities

Very often an embedded device works in a remote location. To remotely manage such a device you can use several tools and utilities, including Device Portal and Windows IoT Remote Client. Moreover, you can associate the Secure Shell (SSH) connection and manage the device by using the command line. To access files stored on the device's SD card, you can use File Transport Protocol (FTP). In this section, I show you how to use Device Portal and Windows IoT Remote Client, and how to connect to the Windows 10 IoT Core device using the free FTP and SSH clients.

Device Portal

Device Portal is a web-based utility that enables you to configure an IoT device, install or uninstall its applications, display the active processes, and update Windows 10 IoT Core. Basically, Device Portal is the layer that exposes functionality, which you typically access in the desktop version of Windows 10 through the Task Manager or Control Panel. Naturally, not all functions of Task Manager and Control Panel are available in Device Portal—only those related to Windows 10 IoT Core. Simply, Device Portal lets you remotely manage your device. Interestingly, a very similar Device Portal is available for holographic platforms (HoloLens), and even for desktop Windows 10 (starting with its Anniversary Edition).

To access Device Portal you use the IoT Dashboard. Go to the **My Device** list and right-click your IoT devices. Then, as shown in Figure 2-28, select the **Open in Device Portal** option from the context menu. Device Portal will open in the default browser and ask you to provide credentials (see Figure 2-29). Type **administrator** for the login, and for the password, provide a value you previously configured during Windows 10 IoT Core installation through the IoT Core Dashboard.

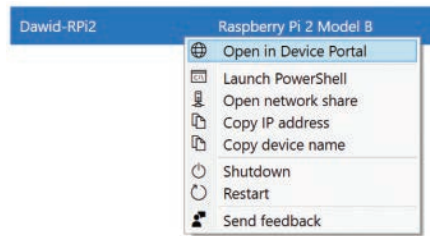


FIGURE 2-28 Context menu of the IoT device in the IoT Dashboard.

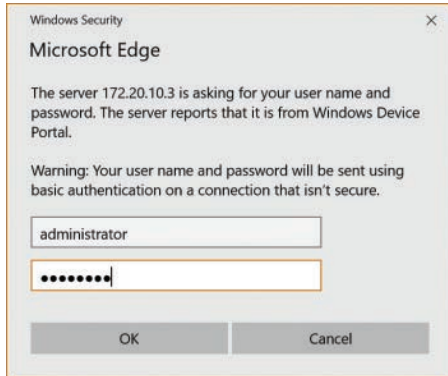


FIGURE 2-29 Windows Device Portal login screen.

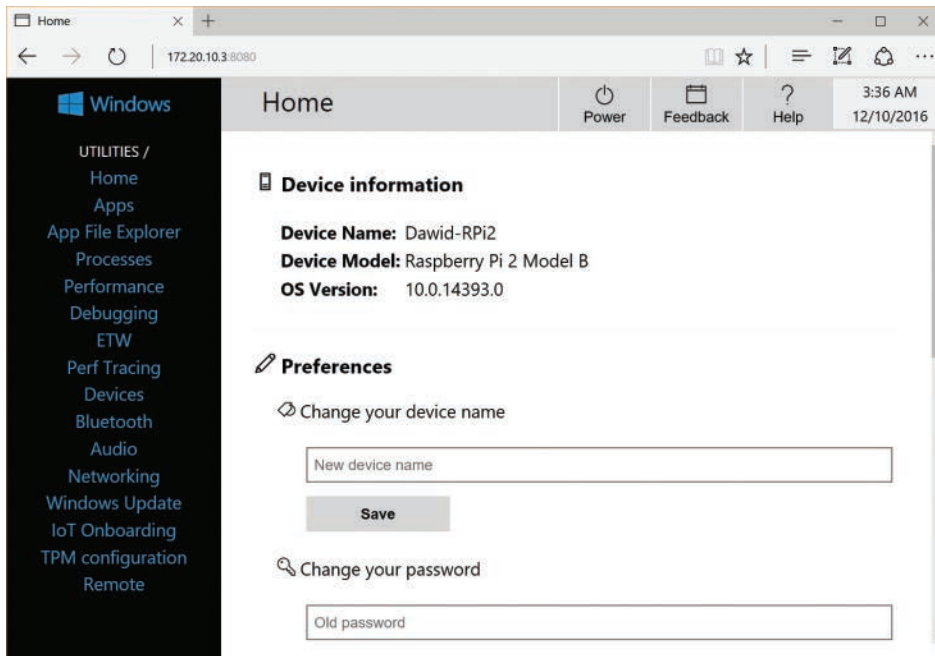


FIGURE 2-30 The Home tab of the Windows Device Portal.

After successful login to Device Portal, you will see the screen shown in Figure 2-30. By default, it displays the Home tab. It contains basic information about your device and enables you to configure device preferences, like name, password, and display settings. I encourage you to navigate among the tabs of the Device Portal to see what is available there. We will use specific functions of the Device Portal later.

Windows IoT Remote Client

Starting from the Windows 10 IoT Core Anniversary Edition build, you can remotely control your IoT device using the Windows IoT Remote Client. This is a tiny app that you install on your development PC, tablet, or phone from the Windows Store. When you set up the connection using Remote Client between a PC, tablet, or phone and the IoT device, the IoT device will transmit its current screen to the Windows IoT Remote Client app. In this way you can preview your UWP apps running on the remote IoT device from another UWP device (desktop or mobile).

To set up such a connection, you first need to enable Windows IoT Remote Server using Device Portal. As shown in Figure 2-31, all you need to do is to select the **Enable Windows IoT Remote Server** check box on the Remote tab. Then, you simply run the Windows IoT Remote Client, where you either choose your device from the drop-down list or type its IP address (see Figure 2-32). After clicking the **Connect** button, you will see an IoT device screen, as shown in Figure 2-33.

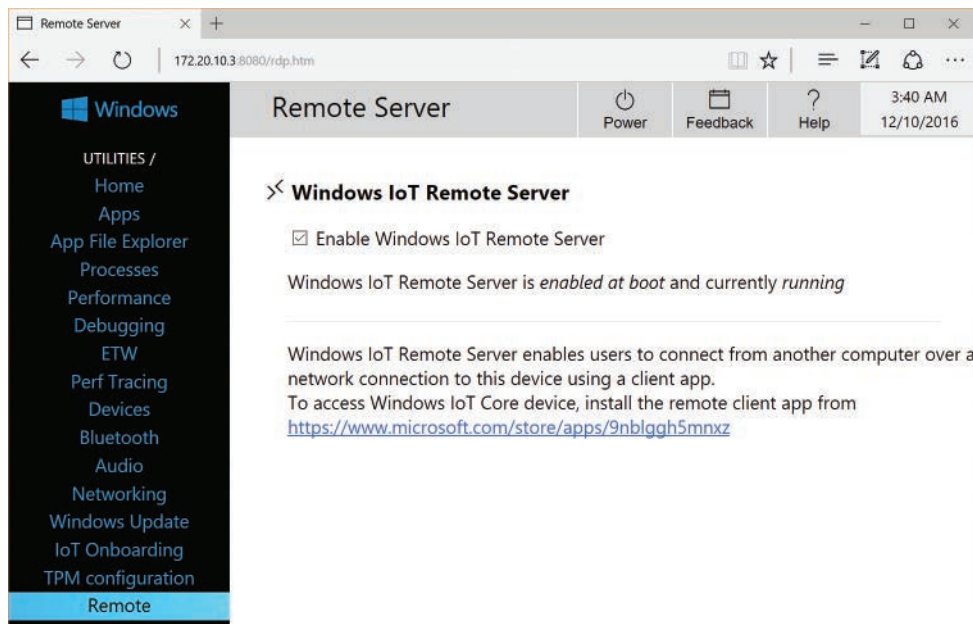


FIGURE 2-31 Enabling Windows IoT Remote Server using the Device Portal.

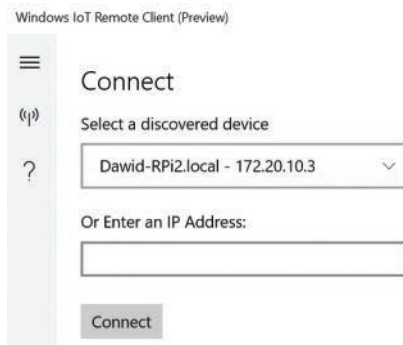


FIGURE 2-32 Connecting to the remote IoT device using Windows IoT Remote Client.

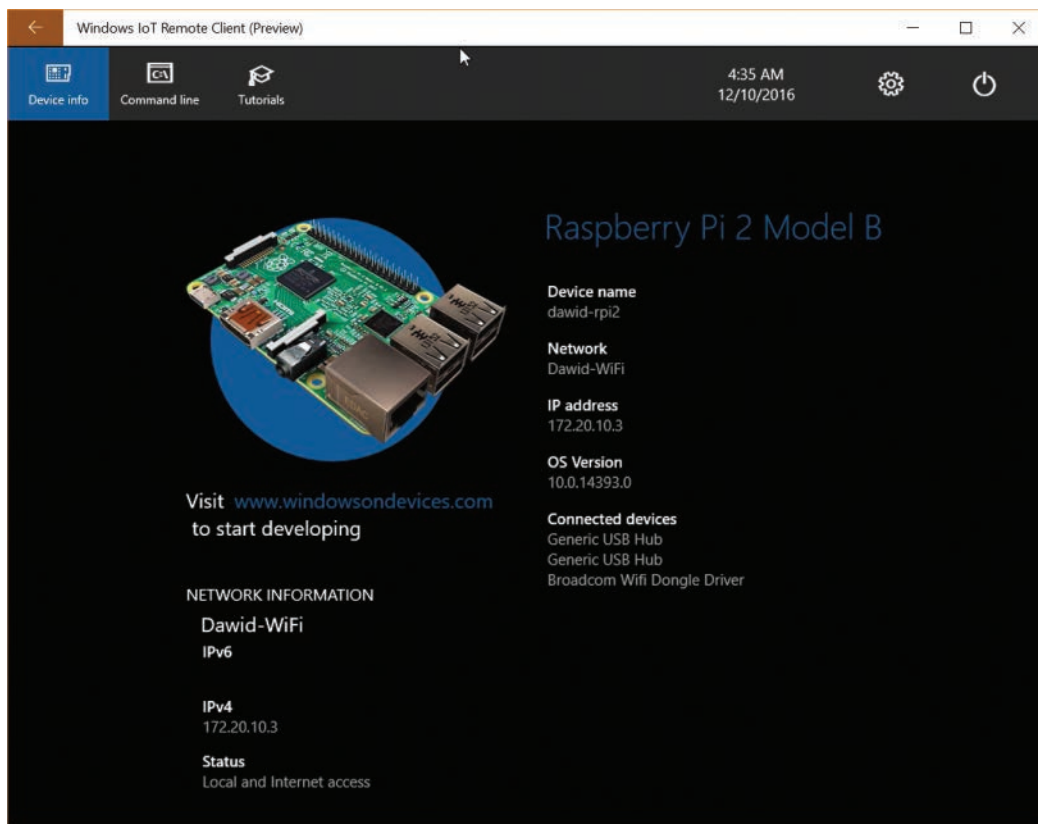


FIGURE 2-33 Windows IoT Remote Client showing the default Windows 10 IoT Core headed app running on the Raspberry Pi 2 Model B.

Note that Windows IoT Remote Client works similarly to remote desktop clients you're familiar with. So you can use the input devices of your desktop PC (keyboard and mouse) or mobile (touchscreen) to control remote IoT apps. This offers a very convenient way of testing your apps without the need of hooking up the physical input devices to the IoT device.

SSH

You can use the Putty application to associate an SSH connection. Putty is one of the most popular SSH Windows clients. You can download this lightweight application-executable tool from <http://www.putty.org/>.

Follow these steps to connect to your Windows 10 IoT Core using the Putty terminal:

1. Download and run the Putty SSH client.
2. In the main Putty window, shown in Figure 2-34, enter a hostname (or an IP address) of your IoT device and then click **Open**.

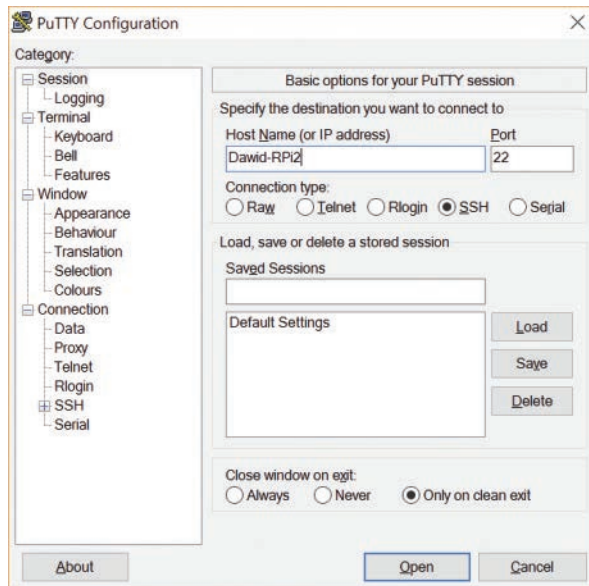
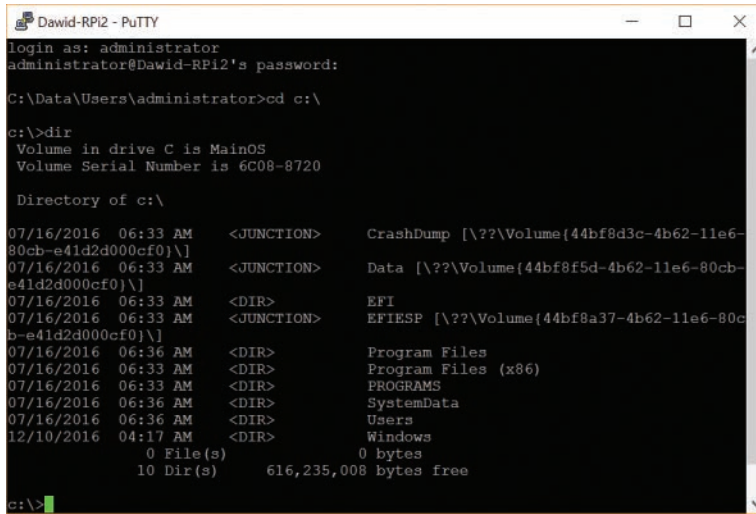


FIGURE 2-34 The Putty application.

3. A security warning appears. Click the **Yes** button.
4. Enter your credentials.
5. List the SD card content by typing the following commands (see Figure 2-35):

```
cd C:\
```

```
dir
```



```
Dawid-RPi2 - PuTTY
login as: administrator
administrator@Dawid-RPi2's password:

C:\Data\Users\administrator>cd c:\

c:\>dir
Volume in drive C is MainOS
Volume Serial Number is 6C08-8720

Directory of c:\

07/16/2016  06:33 AM  <JUNCTION>      CrashDump  [\??\Volume{44bf8d3c-4b62-11e6-80cb-e41d2d000cf0}\]
07/16/2016  06:33 AM  <JUNCTION>      Data  [\??\Volume{44bf8f5d-4b62-11e6-80cb-e41d2d000cf0}\]
07/16/2016  06:33 AM  <DIR>          EFI
07/16/2016  06:33 AM  <JUNCTION>      EFIESP  [\??\Volume{44bf8a37-4b62-11e6-80cb-e41d2d000cf0}\]
07/16/2016  06:36 AM  <DIR>          Program Files
07/16/2016  06:33 AM  <DIR>          Program Files (x86)
07/16/2016  06:33 AM  <DIR>          PROGRAMS
07/16/2016  06:36 AM  <DIR>          SystemData
07/16/2016  06:36 AM  <DIR>          Users
12/10/2016  04:17 AM  <DIR>          Windows
               0 File(s)                0 bytes
               10 Dir(s)              616,235,008 bytes free

c:\>
```

FIGURE 2-35 Folder structure of the Windows 10 IoT Core device obtained using the SSH client application.

While connected to Windows 10 IoT Core using the SSH protocol, you can use similar commands to those in the desktop command prompt. For example, to display a list of active processes, you can type **tlst**. To investigate network connections, you can use the **netstat** utility.

FTP

By default, the FTP server is disabled on Windows 10 IoT Core. To enable it, you can use the SSH connection, where you run the following command: **start c:\Windows\System32\ftpd.exe**. This will start the FTP server. You confirm that this server is running by typing **tlst | more**. This command displays the list of active processes. Press the spacebar to go to the next page of this list or **Enter** to show the next line.

You can stop the FTP server anytime by typing **kill <PID>**, where **<PID>** is the process identifier. It is displayed in the process list on the left of the process name.

To establish the FTP connection with your IoT device, you need an FTP client application. Here, I am using WinSCP, which is the free FTP/SFTP client for Windows. You can download it from: <https://winscp.net/eng/download.php>.

After you install and run this application, you see the configuration screen (see Figure 2-36). Using this dialog box, perform the following steps:

1. Select **FTP** from the **File Protocol** drop-down list.
2. Provide the IP address (hostname) and connection credentials of the embedded device running Windows 10 IoT Core. Use **administrator** as a login; for the password, use the value you configured previously using the IoT Dashboard.
3. Click the **Login** button to connect to the IoT device.

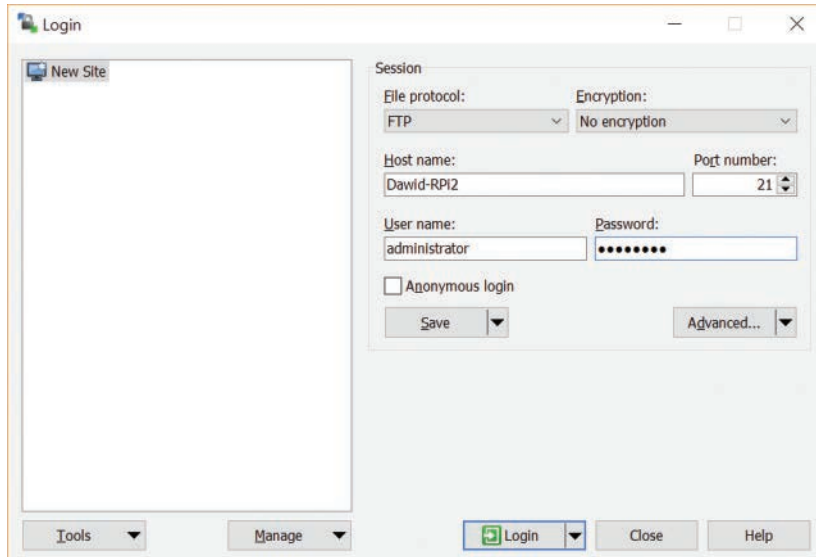


FIGURE 2-36 An FTP connection configuration.

4. During the connection process, a security warning appears. Confirm it by clicking the **Yes** button.
5. The contents of the IoT device SD card appear (See Figure 2-37.)

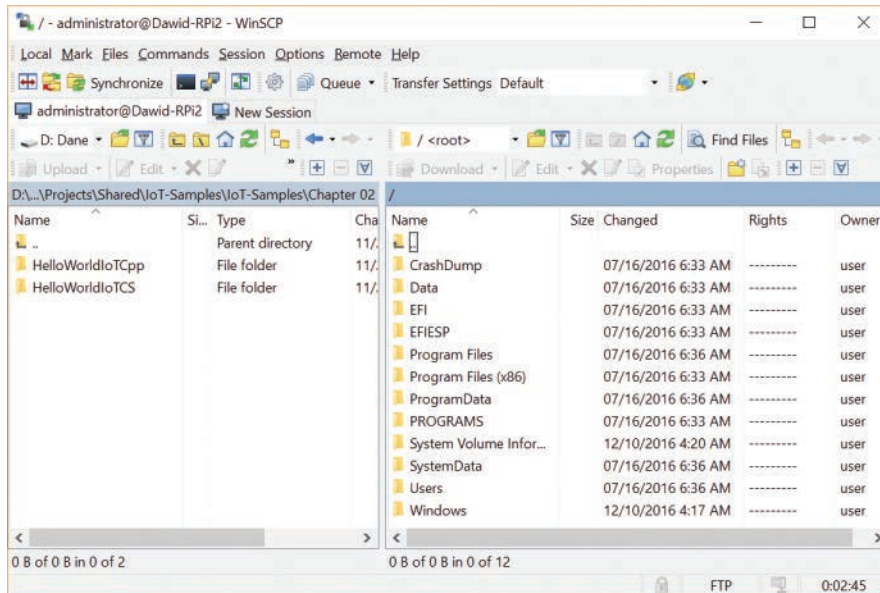


FIGURE 2-37 Contents of the SD card of a Windows 10 IoT Core device.