



2ND EDITION

Internet of Things Programming Projects

Build exciting IoT projects using Raspberry Pi 5,
Raspberry Pi Pico, and Python

COLIN DOW

Internet of Things Programming Projects

Build exciting IoT projects using Raspberry Pi 5,
Raspberry Pi Pico, and Python

Colin Dow



Internet of Things Programming Projects

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capital. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Preet Ahuja

Publishing Product Manager: Vidhi Vashisth

Book Project Manager: Uma Devi

Senior Editor: Mohd Hammad

Technical Editor: Rajat Sharma

Copy Editor: Safis Editing

Proofreader: Mohd Hammad

Indexer: Tejal Soni

Production Designer: Aparna Bhagat

DevRel Marketing Coordinator: Sharon Sandhya

Senior DevRel Marketing Coordinator: Rohan Dobhal

First published: October 2018

Second Edition: June 2024

Production reference: 1070624

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK

ISBN 978-1-83508-295-9

www.packtpub.com

I would like to thank my wife, Constance, for her encouragement, support, and assistance, without which this book would not be possible. I would also like to thank my sons, Maximillian and Jackson, for making me the proudest father there could be.

– Colin Dow

Contributors

About the author

Colin Dow has been involved with technology since the early home computers caught his eye. He has worked as a programmer/analyst at some of Canada's biggest companies. He is the author of the Packt books *Internet of Things Programming Projects* (2018), *Hands-On Edge Analytics with Azure IoT*, and *Simplifying 3D Printing with OpenSCAD*. When he's not nerding out on programming or 3D printing, he likes to compose electronic music, which can be found on Spotify under the name *Project Josephine*.

About the reviewers

Muhammad Afzal is a senior software engineer, with more than 14 years of experience working on web-based and IoT systems in multinational organizations. He always enjoys working and solving real-world business problems with technology. He is also a book author and has published the book *Arduino IoT Cloud for Developers* with Packt Publishing, as well as working with EC-Council for the CodeRed project, where he develops courses regarding IoT.

He provides freelance services to IoT-based product companies, writing technical reviews and projects, as well as providing consultancy to organizations.

I would like to express my heartfelt gratitude to my parents (Mr. and Mrs. Muhammad Aslam), my wife, and my children; they have all always supported me in my decisions, especially my brothers and sisters. And to the memory of my grandfather, Muhammad Ameer, for his sacrifices and exemplifying the power of determination.

Jackson Dow is currently working toward a Bachelor of Science at Wilfrid Laurier University in Waterloo, Ontario, Canada. A computer science major, Jackson has a strong interest in technology and has developed phone apps, soon to be available in the App Store. He grew up in Brampton, Ontario, playing both rep hockey and baseball, and he achieved his French immersion certificate. He is grateful for the opportunity to participate in the review of this exciting technical publication.

I would like to thank Colin Dow for writing a book that was not only easy to follow but also interesting and informative. I would also like to thank the Book Project Manager and Senior Editor for making the reviewing process easier for me.

Table of Contents

Preface	xv

Part 1: Setting Up the Raspberry Pi for IoT Development

1

Understanding the Raspberry Pi	3
Technical requirements	5
Exploring Raspberry Pi models	5
Exploring alternatives to the Raspberry Pi	10
Looking at the power of HATs	12
Pibrella HAT	12
The Raspberry Pi Sense HAT	13
Investigating operating systems for the Raspberry Pi	14
Using the Raspberry Pi for IoT	16
Utilizing web services for IoT applications	16
Re-introducing T.A.R.A.S. – an IoT-based robotics project	17
Getting started with Raspberry Pi development	18
Raspberry Pi development tools	19
Raspberry Pi and Sense HAT development	20
Summary	32

2

Harnessing Web Services with the Raspberry Pi	33
Technical requirements	34
Exploring web services	34
Understanding approaches for web services	35
Connecting to a web service with our Raspberry Pi and Sense HAT	38
Creating a scrolling stock ticker application	44
Getting an API key	45
Writing web services client code	46
Enhancing our application	48

Developing weather display applications	51	Developing a GO-NO-GO application for decision-making	59
Getting an API key	52	Building other GO-NO-GO applications	70
Creating a scrolling weather information ticker	53	Summary	70
Scrolling weather information on Sense HAT	55		

3

Building an IoT Weather Indicator **73**

Technical requirements	74	Using GPIO Zero to control an LED	84
Looking into servo motors	75	Building the weather indicator stand	85
Connecting the SG90 servo motor to our Raspberry Pi	75	Assembling the weather indicator stand	86
Understanding servo motors	77	Developing code for our application	94
Exploring LEDs	78	Calibrating the needle	95
Connecting an LED to our Raspberry Pi	79	Creating the WeatherData class	96
Controlling servo motors and LEDs using Python	81	Creating the WeatherDashboard class	102
Setting up our development environment	81	Adding the updateDashboard() function and main methods	105
Using GPIO Zero to control a servo	82	Summary	107

4

Building an IoT Information Display **109**

Technical requirements	109	Creating a WeatherData class	116
Investigating displays compatible with our Raspberry Pi and exploring screen types	110	Creating a TrafficMap class	119
Creating an IoT information display	112	Adding Dashboard and MyApp classes	122
Setting up our development environment	113	Running the IoT information display application	128
		Summary	129

Part 2: Building an IoT Home Security Dashboard

5

Exploring the GPIO 133

Technical requirements	133	Understanding sensors, actuators, and indicators	139
Introducing the GPIO on Raspberry Pi	134	Setting up our development environment	140
Exploring the Raspberry Pi GPIO pinout diagram	134	Exploring the PIR sensor	142
Understanding GPIO pin communication protocols	135	Building a simple alarm system	144
		Summary	148

6

Building an IoT Alarm Module 149

Technical requirements	150	Introducing the RP2040 chip	155
Investigating MQTT	150	Configuring our alarm circuit	156
Understanding the publish-subscribe model in MQTT	151	Setting up our development environment	157
Understanding QoS in MQTT	152	Writing the alarm module client code	160
Exploring MQTT fundamentals with the MQTTHQ web client	152	Building an IoT alarm module case	171
Using a Raspberry Pi Pico W with MQTT	154	Identifying the parts of the custom case	172
		Building the alarm module case	174
		Summary	176

7

Building an IoT Button 177

Technical requirements	178	Creating our IoT button using the M5Stack ATOM Matrix	181
Introducing IoT buttons	179	Exploring M5Stack devices	181
Utilizing IoT buttons	179	Flashing the firmware to our ATOM Matrix	183
Exploring various technologies in IoT button development	180	Configuring the ATOM Matrix for programming	184

Turning our ATOM Matrix into an IoT button	186	Modifying our alarm module code	192
Testing our IoT button	189	Building our Raspberry Pi Pico W IoT button	193
Improving on our IoT button with the Raspberry Pi Pico W	190	Installing the components in a custom case	206
Setting up a CloudAMQP instance	191	Summary	208

8

Creating an IoT Alarm Dashboard 209

Technical requirements	210	Writing the dashboard code	216
Exploring IoT alarm dashboards	211	Building the external alarm buzzer stand	230
Using IoT alarm dashboards for industrial processes	211	Identifying the parts	230
Exploring the IoT security alarm dashboard	212	Building the stand	231
Creating a Raspberry Pi 5 alarm dashboard	213	Running our application	232
Modifying the IoT alarm module code	213	Summary	234

Part 3: Creating a LoRa-Enabled IoT Monitoring Station

9

Understanding LoRa 237

Technical requirements	238	Developing the code	248
Exploring LoRa	239	Installing the components in a custom case	255
Practical uses for LoRa technology	239	Building a LoRa receiver	258
Investigating the radio frequency spectrum	240	Wiring an LED to the Raspberry Pi Pico W	259
Understanding the LoRa SF	242	Creating code to receive LoRa messages	260
Using LoRa with the Raspberry Pi Pico and Pico W	243	Testing our application	262
Building a LoRa sensory transmitter	243	Summary	263
Constructing our circuit	244		

10

Integrating LoRa with the Internet 265

Technical requirements	266	Creating a new weather indicator	274
Connecting our LoRa receiver to the internet	266	Building the split stand	274
Installing the CircuitPython library for MQTT	267	Building the faceplate	276
Creating a CloudAMQP instance for our application	269	Configuring the RGB LED indicator	279
Adding MQTT functionality to the LoRa receiver	269	Configuring the servo motor	284
		Programming our weather indicator	288
		Exploring other IoT communication protocols	296
		Summary	298

Part 4: Building an IoT Robot Car

11

Introducing ROS 301

Technical requirements	301	Adding ROS to our Ubuntu installation	308
Exploring ROS	302	Testing our ROS installation	311
Reviewing our TurtleSim controller ROS application	302	Running and controlling a simulated robot	313
Understanding ROS node communication	303	Launching and testing TurtleSim	313
Investigating ROS project structure and organization	304	Creating an ROS workspace and package	316
Aligning ROS distributions with Ubuntu LTS versions	305	Modifying the generated Python code	318
Installing Ubuntu and ROS onto our Raspberry Pi	305	Updating package.xml	321
Installing Ubuntu on our Raspberry Pi 4	306	Compiling and running our code	322
		Controlling our robot with an MQTT message	323
		Summary	324

12

Creating an IoT Joystick 325

Technical requirements	325	Sending MQTT messages from our IoT joystick	334
Understanding our IoT joystick application	326	Creating a custom ROS node for our application	338
Wiring up our circuit	327	Creating our custom robot_control node	339
Developing the code for our IoT joystick	329	Controlling a ROS TurtleSim robot using our IoT joystick	343
Setting up our Raspberry Pi Pico WH	329	Constructing the IoT joystick case	345
Creating a Joystick class	331	Summary	349

13

Introducing Advanced Robotic Eyes for Security (A.R.E.S.) 351

Technical requirements	351	Running the installation script	365
Exploring our A.R.E.S. application	352	Creating alarm code for the Pico H	368
Constructing A.R.E.S.	353	Testing and controlling the motors	371
Identifying the 3D-printed frame parts	353	Testing communication between Pi and Pico	374
Identifying the components used to create A.R.E.S.	354	Testing the ToF sensor	379
Building A.R.E.S.	355	Streaming video from A.R.E.S.	380
Wiring up A.R.E.S.	357	Programming A.R.E.S. with ROS	385
Software setup and configuration	360	Summary	388
Installing Ubuntu onto our Raspberry Pi 3B+	361		

14

Adding Computer Vision to A.R.E.S. 389

Technical requirements	389	Understanding YOLO and neural networks	397
Exploring computer vision	390	Exploring object detection	397
Introducing OpenCV	391		

Adding computer vision to A.R.E.S.	403	Sending out a text alert	408
Creating the DogTracker class	403	Setting up our Twilio account	408
Building a smart video streamer	405	Adding text message functionality to A.R.E.S.	414
		Summary	418
Index			421
Other Books You May Enjoy			432

Preface

Internet of Things Programming Projects is a comprehensive hands-on guide, designed to walk you through a series of progressively advanced **Internet of Things (IoT)** projects using Raspberry Pi and associated peripherals.

In the initial stages, you will be introduced to the foundational concepts and components surrounding the Raspberry Pi, including a detailed exploration of various models, accessories such as hardware attached on top (HAT), and compatible operating systems. You will begin your journey into IoT by engaging in projects that demonstrate the Raspberry Pi's capacity to interface with real-time data and control physical devices, setting the stage for more advanced undertakings.

As you proceed, you will immerse yourself in the development of web services and IoT applications, creating real-time data displays and innovative solutions such as a weather indicator application. These applications not only provide Python programming and data acquisition skills but also venture into the physical world, utilizing hardware components to create tangible outcomes.

In the middle portion of the book, emphasis is laid on building a home security system from scratch. You will learn about the general-purpose input/output (GPIO) pins of the Raspberry Pi, sensor integrations, and hands-on alarm system development, using protocols such as message queuing telemetry transport (MQTT). A pivotal point will be reached when we create an MQTT-based standalone IoT alarm module and its associated peripherals.

Toward the latter part of the journey, the narrative progresses to more advanced and autonomous IoT projects. You will construct a LoRa-enabled IoT monitoring station capable of measuring various environmental factors, including air quality. This station, powered by a battery, communicates data through LoRa to a LoRaWAN network, illustrating the integration of IoT devices with wider network infrastructures and emphasizing sustainable, battery-operated solutions.

Culminating the series of projects is the creation of an IoT robot car, leveraging the Robot Operating System (ROS). This advanced endeavor guides you in building a robot car equipped to send sensory information over the internet through MQTT, facilitating remote control via a web browser or other applications. This project represents a pinnacle of IoT mastery, incorporating robotics, network communications, and remote control functionalities into a unified system.

Throughout the book, you will be empowered with the knowledge and skills to build practical, real-world IoT solutions, nurturing creativity and innovation through continuous project enhancement suggestions. From Python programming to hardware interfacing, this book promotes a rich, layered understanding of IoT principles, encouraging a readiness to tackle a myriad of complex problems in the IoT landscape, and stands as a beacon for aspiring IoT enthusiasts to develop robust, versatile, and innovative IoT solutions.

Let's get into it!

Who this book is for

Internet of Things Programming Projects is geared toward tech enthusiasts, hobbyists, and professionals who are eager to dive into the world of IoT. The book covers a range of topics, including web services, LoRa communication, Raspberry Pi, Raspberry Pi Pico, and interacting with GPIO. You will also learn about ROS, building a robot car, and implementing vision recognition. To get the most out of this book, you should have a basic understanding of programming, electronics, and networking. This comprehensive guide is ideal for those looking to expand their knowledge and skills in IoT by engaging in practical, hands-on projects.

What this book covers

Chapter 1, Understanding the Raspberry Pi, explores the Raspberry Pi before we embark on creating IoT projects using it.

Chapter 2, Harnessing Web Services with the Raspberry Pi, delves into writing Python code to transform our Raspberry Pi into an IoT device, by leveraging web services to pull data and create visual displays on the Sense HAT. This lays the groundwork for more advanced IoT web services development, through practical projects such as a stock ticker, weather display, and decision-maker applications.

Chapter 3, Building an IoT Weather Indicator, explores the integration of servo motors and LEDs with the Raspberry Pi to create a practical IoT weather indicator, leveraging its precision, control, and real-time feedback capabilities for enhanced system functionality.

Chapter 4, Building an IoT Information Display, shows you how to build an IoT information display, using the Raspberry Pi-branded 7-inch touchscreen to show real-time weather and local traffic information. The chapter starts with an exploration of compatible screens and culminating in a comprehensive *dashboard* project.

Chapter 5, Exploring the GPIO, dives deeper into the GPIO port functionality on both the Raspberry Pi and Raspberry Pi Pico by constructing an IoT home security application, including a basic alarm system with a passive infrared sensor (PIR) motion sensor, pushbutton, and buzzer.

Chapter 6, Building an IoT Alarm Module, looks at enhancing our basic alarm system by using a Raspberry Pi Pico W, a public MQTT server, and the MQTTHQ web client to build an IoT alarm module, where motion detection triggers messages and remote buzzer activation. This will form the foundation of our IoT home security system.

Chapter 7, Building an IoT Button, shows you how to build an essential component of our IoT home security system, the IoT button, using both the compact M5Stack ATOM Matrix and the versatile Raspberry Pi Pico W for different versions of the button.

Chapter 8, Creating an IoT Alarm Dashboard, covers using a Raspberry Pi 5 with a 7-inch touchscreen to create an IoT alarm dashboard, allowing us to arm and disarm the alarm module, review MQTT notifications, and display a map of alarm locations, thereby completing our advanced IoT alarm system with global application capabilities.

Chapter 9, Understanding LoRa, explores LoRa (short for Long Range) technology for IoT communication, its applications in agriculture and smart cities, and how to build a LoRa sensory transmitter and receiver using Raspberry Pi Pico and Pico W, respectively, demonstrating LoRa's extensive range capabilities and efficient low-power data transmission.

Chapter 10, Integrating LoRa with the Internet, shows you how to use our Raspberry Pi Pico W-equipped LoRa receiver to publish sensory data from a remote LoRa transmitter to an MQTT server, modify our analog-metered weather indicator to utilize this data, and explore various IoT communication technologies such as LoRaWAN and cellular.

Chapter 11, Introducing ROS, introduces the ROS, detailing its significance in robotics, setting it up on a Raspberry Pi 4 with Ubuntu, and using turtlesim to learn basic ROS concepts and operations, ultimately preparing us to build the advanced IoT robot A.R.E.S. (Advanced Robot Eyes for Security).

Chapter 12, Creating an IoT Joystick, shows you how to create an IoT joystick using a Raspberry Pi Pico WH. You will use it to remotely control a ROS TurtleSim robot, building on previous projects and demonstrating IoT's practical application in robotics.

Chapter 13, Introducing Advanced Robotic Eyes for Security (A.R.E.S.), focuses on converting our TurtleSim virtual robot into a real-life robot called ARES, which features a video feed accessible via the VLC media player and is controlled by the IoT Joystick from *Chapter 12*. We will use a Raspberry Pi for sensory input and a Raspberry Pi Pico for motor, LED, and buzzer control, with a 3D-printed frame.

Chapter 14, Adding Computer Vision to A.R.E.S., finally sees us add computer vision to ARES, enabling it to recognize objects and send text alerts. We will also use OpenCV and the You Only Look Once (YOLO) object detection system to build a smart video streaming application.

To get the most out of this book

You should have some experience with Python and JavaScript programming. Important skills include being able to work with Raspberry Pi, Python, and web services to create IoT applications, as well as engaging in Raspberry Pi-controlled robotics.

Software/hardware covered in the book	Operating system requirements
Python	Raspberry Pi OS and Ubuntu Linux
CircuitPython	
MQTT	
OpenCV	
LoRa	
ROS	
Raspberry Pi	
Raspberry Pi Pico	
Raspberry Pi Sense HAT	
Various sensors (PIR and temperature)	
M5Stack ATOM Matrix	
Pico robotics board	

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Internet-of-Things-Programming-Projects-2nd-Edition>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “We then define an `activate_buzzer()` function.”

A block of code is set as follows:

```
def timer_callback(self):
    if self.mqtt_message.should_draw_circle:
        self.vel_msg.linear.x = 1.0
        self.vel_msg.angular.z = 1.0
    else:
        self.vel_msg.linear.x = 0.0
        self.vel_msg.angular.z = 0.0
    self.publisher.publish(self.vel_msg)
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “We do so by clicking on the **Menu** icon in the Raspberry Pi taskbar, navigating to the **Programming** category, and selecting **Thonny**.”

Tips or important notes

Appear like this.

Get in touch

Feedback from you is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Internet of Things Programming Projects*, we'd love to hear your thoughts! Please click [here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781835082959>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1:

Setting Up the Raspberry Pi for IoT Development

In *Part 1*, we begin by understanding the Raspberry Pi, and then we will harness web services for IoT applications, build an IoT weather indicator using sensors and web data, and create an IoT information display with a Raspberry Pi and touchscreen for real-time weather and traffic updates.

This part has the following chapters:

- *Chapter 1, Understanding the Raspberry Pi*
- *Chapter 2, Harnessing Web Services with the Raspberry Pi*
- *Chapter 3, Building an IoT Weather Indicator*
- *Chapter 4, Building an IoT Information Display*

1

Understanding the Raspberry Pi

Welcome to the wonderful world of **Internet of Things (IoT)** projects and the Raspberry Pi in this, the second edition of *Internet of Things Programming Projects*.

In this book, we will take a journey into IoT projects using the Raspberry Pi. In the first part of this book, we'll explore IoT projects on the Raspberry Pi, initially transforming it into a weather station with the Sense HAT to capture real-time data. Later, we'll use motors and **general-purpose input/output (GPIO)** pins to repurpose the Pi as an analog metering device.

We will then create an IoT home security system, using the Raspberry Pi as the central hub for an alarm system. We'll also develop a LoRa-enabled IoT monitoring station for remote sensing. The book culminates with our most ambitious project: building an IoT robot car powered by the Raspberry Pi.

We will start this chapter by exploring the various Raspberry Pi models and their significance, observing the evolution and advancements in processing power, memory, and capabilities over time.

We will also look at alternatives to the Raspberry Pi available in the IoT landscape, enabling us to make informed decisions based on our project requirements. We compare these alternatives with the Raspberry Pi, highlighting their unique specifications and capabilities.

Furthermore, we dive into Raspberry Pi's **Hardware Attached on Top (HAT)**, which expands the Raspberry Pi's capabilities through add-on boards. Specifically, we explore the Pibrella HAT and the Raspberry Pi Sense HAT, highlighting their ability to enhance the Raspberry Pi with additional features and functionalities.

Our Sense HAT projects will make use of an optional custom Raspberry Pi case we see in *Figure 1.1*. This case allows us to display our Raspberry Pi in a vertical format and make use of the dot matrix display on the Sense HAT. The build files for the Raspberry Pi 4B and Raspberry Pi 5 versions' case are available in the GitHub repository for the book.

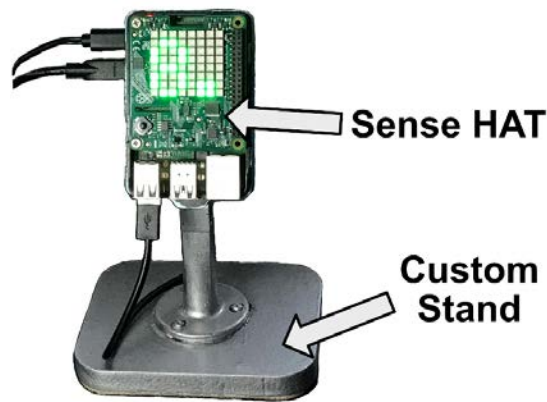


Figure 1.1 – Raspberry Pi 4B and Sense HAT in a custom case

We will also discuss a range of operating systems compatible with the Raspberry Pi beyond the official Raspberry Pi OS. These options cater to various applications, including networked audio systems, aviation-related projects, retro gaming, and 3D printing.

In the context of IoT applications, we will highlight the Raspberry Pi's versatility and power as it serves as an optimal platform for processing real-time data and controlling physical devices, pivotal in the development and deployment of versatile IoT projects.

Finally, we will explore the Raspberry Pi as a robust development platform, equipped with pre-installed tools for software development. We conclude the chapter with a series of programming projects using the Raspberry Pi with the Sense HAT to extract sensory information as we build a scrolling environmental data display.

Our hands-on dive into programming in this chapter will fine-tune our programming abilities and ready us for exciting IoT project development throughout this book. Although there is a lot of information that is covered in this chapter, we should not worry if we feel overwhelmed or if we can't digest all the information from the first chapter right away. As we progress through the book, we will gain more experience and understanding, making it easier to grasp the concepts introduced early on.

We will cover the following:

- Exploring Raspberry Pi models
- Exploring alternatives to the Raspberry Pi
- Looking at the power of HATs
- Investigating operating systems for the Raspberry Pi
- Using the Raspberry Pi for IoT
- Getting started with Raspberry Pi development

Technical requirements

The following are required to complete this chapter:

- Late-model Raspberry Pi, such as the Raspberry Pi 5 4/8 GB model or Raspberry Pi 4B 4/8 GB model
- Keyboard, mouse, and monitor
- Raspberry Pi Sense HAT is optional but encouraged; we will be making use of the Sense HAT software emulator

The GitHub repository for the chapter is located at <https://github.com/PacktPublishing/Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter1>.

For those that have access to a 3D printer or 3D printing service, there are .stl files available in the `Build Files` directory of this chapter's GitHub repository for the construction of the optional custom case. Files are available for the Raspberry Pi 5 and Raspberry Pi 4B versions of the case.

A general knowledge of programming is also required. We will be using the Python programming language in this book. We will start with basic code and work our way toward more advanced coding as we go through the projects in the book.

Exploring Raspberry Pi models

Every Raspberry Pi model, from the initial Raspberry Pi 1 to the current Raspberry Pi 5 and the compact Pi Zero, possesses distinct characteristics and functionalities that make it an exciting choice for IoT developers. Each model excels in different areas, such as processing power, size, and energy efficiency.

The Raspberry Pi models have evolved significantly over time, with each iteration bringing notable advancements in processing power, memory, and capabilities.

In the following list, we compare the various models of the Raspberry Pi, starting with the first one:

- **Raspberry Pi 1:** Launched in February 2012, the Raspberry Pi 1 changed the world of computing with its affordability and accessibility. Equipped with a 700 MHz processor, 512 MB RAM, and a \$35 price point, the first model of the Raspberry Pi spurred digital innovation and highlighted the potential of single-board computers.
- **Raspberry Pi 2:** Released in February 2015, the Raspberry Pi 2 improved on the first model with a 900 MHz quad-core processor and doubled RAM at 1 GB. The Raspberry Pi 2 also expanded the GPIO from 26 to 40 pins, allowing for a new wave of 40-pin HATs. These advancements allowed the Raspberry Pi 2 to become a hub for complex projects, from robotics to IoT applications.
- **Raspberry Pi 3:** Released in February 2016, the Raspberry Pi 3 sported a 1.2 GHz quad-core processor. This improved performance by 50-60% and enabled more resource-intensive applications. Like the Raspberry Pi 2, it maintained 1 GB RAM. Wi-Fi and Bluetooth 4.1 were

integrated, simplifying connectivity and freeing USB ports (a USB Wi-Fi dongle was required on the Raspberry Pi 2). A new dual-core VideoCore IV GPU enhanced multimedia projects with improved video capabilities. *Figure 1.2* provides a layout of a Raspberry Pi 3, highlighting several of its key components:

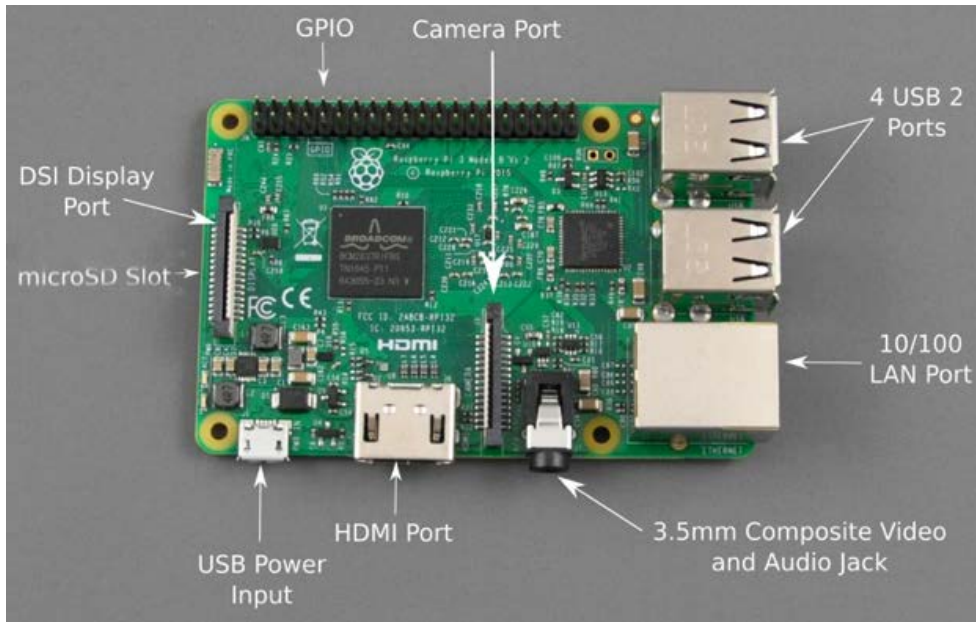


Figure 1.2 – Raspberry Pi 3B

- Raspberry Pi 4:** Unveiled in June 2019, the Raspberry Pi 4 marked a significant evolution in the series, pushing the boundaries of single-board computers closer to conventional desktop PCs in terms of capabilities, all while preserving its compact size and affordability. What set the Pi 4 apart was the variety of memory options it offered, 2 GB, 4 GB, and 8 GB LPDDR4-3200 SDRAM, a substantial improvement over the previous 1 GB LPDDR2 RAM, enabling smoother multitasking and handling of data-intensive tasks. Improved connectivity featured Gigabit Ethernet, dual-band 802.11ac Wi-Fi, and Bluetooth 5.0. Its multimedia capabilities saw a boost with two micro-HDMI ports supporting 4K resolution, allowing the operation of two monitors at once. The Raspberry Pi 4 introduced two USB 3.0 ports for quicker data transfer and replaced the micro-USB power connector with a USB-C, supporting its enhanced features. In *Figure 1.3*, we see a layout of a Raspberry Pi 4 with several of its key components highlighted:

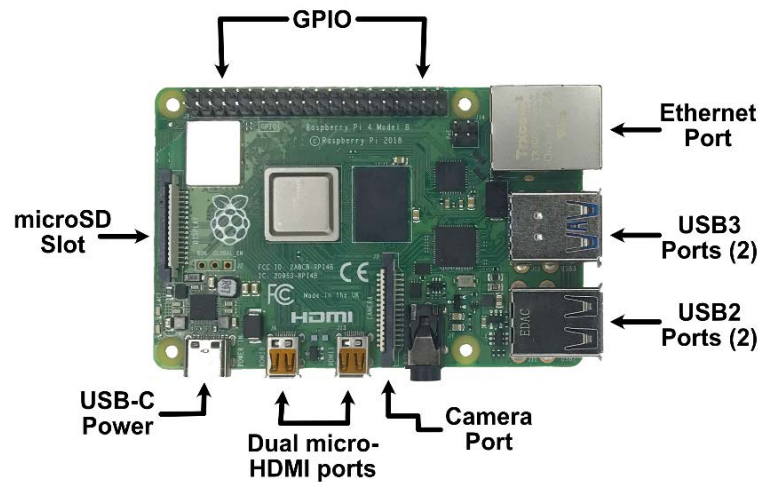


Figure 1.3 – Raspberry 4B

The Raspberry Pi 4 includes two micro-HDMI ports (for dual displays), four USB ports (two 3.0, two 2.0), a Gigabit Ethernet port, a USB-C power port, a micro-SD slot, a camera port, and a 3.5mm audio-composite video jack.

- **Raspberry Pi 5:** The Raspberry Pi 5, released in October 2023, marks a significant advancement in the series, enhancing computational and multimedia capabilities for educational and DIY applications with its upgraded CPU and GPU.

Priced at \$60 for 4 GB and \$80 for 8 GB, the Raspberry Pi 5 features a 2.4GHz quad-core Arm Cortex-A76 CPU, VideoCore VII GPU, dual 4Kp60 HDMI outputs, and various connectivity options including Wi-Fi and Bluetooth. It also introduces a power button, enhanced memory, and I/O capabilities, including two four-lane Mobile Industry Processor Interface (MIPI) camera/display transceivers. These transceivers offer the flexibility to connect any combination of two cameras or displays, making them ideal for advanced multimedia projects.

The Raspberry Pi 5 also features a PCIe 2.0 x1 interface, allowing the connection of fast peripherals to expand its capabilities for advanced applications, such as high-speed networking or storage solutions.

The accompanying Raspberry Pi Active Cooler, with its efficient heatsink and fan design, reduces CPU temperatures by approximately 20 degrees Celsius, vital for intensive tasks. It offers easy installation and up to 8000 RPM fan speed, enhancing performance and longevity. In *Figure 1.4*, the Raspberry Pi 5 is displayed alongside its Active Cooler, with key components such as the PCIe 2.0 Interface, GPIO Header, USB Ports, Ethernet Port, dual micro-HDMI Ports, and the two MIPI Transceivers labeled:

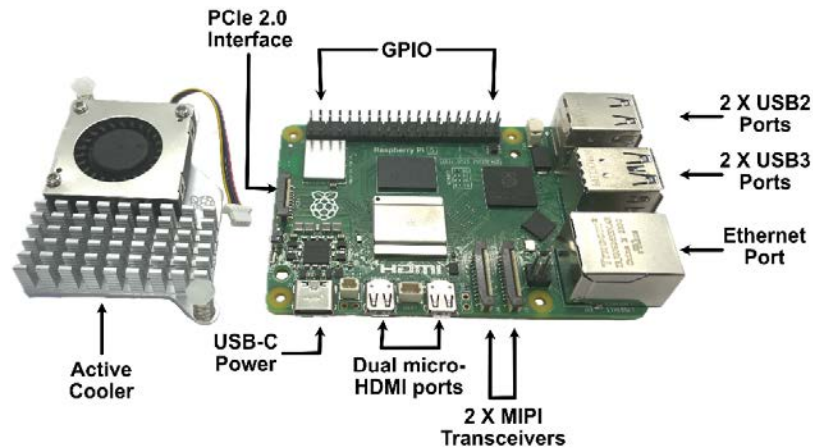


Figure 1.4 – Raspberry Pi 5 (right) and Raspberry Pi Active Cooler (left)

We will feature the Raspberry Pi 5 for our projects that involve a single-board computer, although the late-model Raspberry Pi 4B should suffice too.

- Raspberry Pi Zero and Zero W:** Launched in November 2015, the Raspberry Pi Zero shrank the Raspberry Pi to a size of just 65mm by 30mm. It packed a 1 GHz single-core CPU and 512 MB of RAM at an affordable price. It came with a mini-HDMI Port, a micro-USB OTG Port, a micro-USB Power Port, and a HAT-compatible 40-pin header, making it ideal for compact applications such as IoT projects, wearables, as well as embedded systems. The Raspberry Pi Zero 2W was built on the same form factor and was introduced in 2021. It brought an enhanced 1 GHz quad-core ARM Cortex-A53 CPU, boosting performance and handling more demanding tasks. Its wireless capabilities with onboard Wi-Fi and Bluetooth further expanded its versatility, making it an excellent choice for compact projects. *Figure 1.5* provides a layout of a Raspberry Pi Zero 2W:

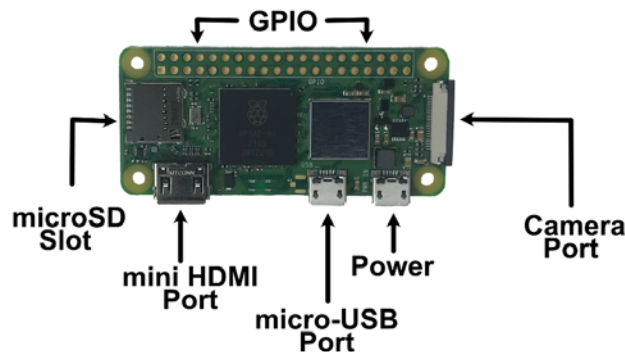


Figure 1.5 – Raspberry Pi Zero 2W

- Raspberry Pi Pico and Pico W:** Introduced in January 2021, the Raspberry Pi Pico is a compact microcontroller board designed for embedded projects and low-level programming. It's only 51mm x 21mm in size, but highly flexible. It supports **Serial Peripheral Interface (SPI)** for high-speed data exchange, **Inter-Integrated Circuit (I2C)** for communication between peripherals, and **Universal Asynchronous Receiver/Transmitter (UART)** for serial communication. The Raspberry Pi Pico W, launched in February 2022, extended this flexibility with onboard Wi-Fi and Bluetooth. Unlike the traditional Raspberry Pi boards, both Pico models offer a unique set of **General Purpose (GP)** pins. Importantly, the Pico series, as microcontrollers rather than single-board computers, do not have an operating system. This enables bare metal programming, an approach where code runs directly on the hardware, resulting in faster, more efficient code execution and precise control. This makes devices such as the Raspberry Pi Pico ideal for applications requiring immediate reactions and close interactions with hardware, making it perfect for time-sensitive tasks. In *Figure 1.6*, we see a Raspberry Pi Pico W and a mapping of its GP pins:

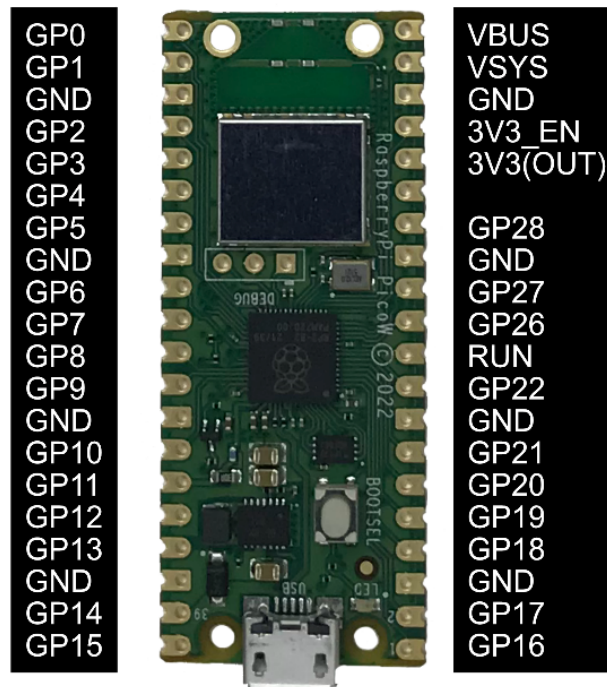


Figure 1.6 – Raspberry Pi Pico W

We will be using the variations of the Raspberry Pi Pico for projects where microcontrollers are more suitable than single-board computers.

The following is a table outlining some of the differences between Raspberry Pi models:

Model	Launch Date	CPU	RAM	Unique Features	Connectivity Options	Remarks
Raspberry Pi 1	Feb 2012	700 MHz Single-core	512 MB	Affordable and accessible, spurred digital innovation	-	Highlighted the potential of single-board computers
Raspberry Pi 2	Feb 2015	900 MHz Quad-core	1 GB	Expanded GPIO from 26 to 40 pins	-	Suited for complex projects like robotics and IoT
Raspberry Pi 3	Feb 2016	1.2 GHz Quad-core	1 GB	50-60% performance improvement, integrated Wi-Fi and Bluetooth 4.1, dual-core VideoCore IV GPU	Wi-Fi, Bluetooth 4.1	Enhanced multimedia projects capability
Raspberry Pi 4	June 2019	64-bit processor running at 1.5 GHz	2GB, 4GB, 8GB	Memory options, Gigabit Ethernet, dual-band 802.11ac Wi-Fi, Bluetooth 5.0, two micro-HDMI (4K) ports	Gigabit Ethernet, dual-band 802.11ac Wi-Fi, Bluetooth 5.0	Pushed closer to desktop PC capabilities
Raspberry Pi 5	Oct 2023	2.4 GHz Quad-core Arm Cortex-A76	4GB, 8GB	PCIe 2.0 x1 interface, dual 4Kp60 HDMI outputs, improved CPU and GPU, power button	Wi-Fi, Bluetooth, dual 4Kp60 HDMI outputs	Significant advancements in computational and multimedia capabilities
Raspberry Pi Zero / Zero W / Zero 2W	Nov 2015 / 2021	1 GHz Single-core / Quad-core ARM Cortex-A53	512 MB	Mini-HDMI port, micro-USB OTG port, 40-pin header	Wi-Fi and Bluetooth (Zero 2W)	Ideal for compact applications
Raspberry Pi Pico / Pico W	Jan 2021 / Feb 2022	A dual-core Arm Cortex-M0+ processor, running at up to 133 MHz	-	Microcontroller board, SPI, I2C, UART, GP pins	Wi-Fi and Bluetooth (Pico W)	Suitable for embedded projects and low-level programming

Figure 1.7 – Table of Raspberry Pi models

Now that we have a better understanding of the ecosystem of Raspberry Pi devices, let's look at some alternatives.

Exploring alternatives to the Raspberry Pi

The IoT landscape is filled with a variety of single-board computers and microcontrollers, each offering unique specifications that cater to specific project needs. By comparing these alternatives with the Raspberry Pi, we can make informed decisions about the ideal platform for our unique requirements. The following is a breakdown of some of the alternatives to the Raspberry Pi that we may consider

for our IoT projects. We will only focus on the Raspberry Pi and not the Raspberry Pi Pico as we will be using the former for this chapter's programming projects:

- **BeagleBone Black:** The BeagleBone Black features a 1GHz ARM Cortex-A8 processor, 512MB DDR3 RAM, 4 GB flash storage, and many connectivity options. Its distinct feature, the **Programmable Real-Time Units (PRUs)**, allow for precise real-time processing, ideal for applications such as robotics or industrial automation. Despite this, the Raspberry Pi remains popular due to its faster 1.5GHz 64-bit quad-core processor, beneficial for resource-intensive applications. The Raspberry Pi's main advantage, however, is its large community software ecosystem, which greatly helps in easing development.
- **Arduino Uno:** Like the Raspberry Pi Pico, the Arduino Uno is a microcontroller rather than a single-board computer. It uses the ATmega328P microcontroller, operates at 16 MHz, and has 2 KB of RAM. The Arduino Uno shines with its easy-to-use shields for functionality expansion, including Wi-Fi, Bluetooth, and sensor capabilities, making it ideal for simpler IoT projects. It benefits from an extensive ecosystem and easy-to-use **Integrated Development Environment (IDE)**. However, for projects requiring heavy computations, multitasking, or extensive data processing, the more powerful Raspberry Pi is a preferable option.
- **ESP32:** The ESP32 by Espressif Systems is a microcontroller like the Arduino Uno but with distinctive features. Its dual-core Xtensa LX6 microprocessor and 520 KB of SRAM, combined with in-built Wi-Fi and Bluetooth, enable robust wireless connectivity for IoT applications. Certain ESP32 models further expand their potential with additional sensors and communication protocols, such as LoRa for long-range, low-power communications. Although it doesn't match the Raspberry Pi's processing and RAM capabilities, it shines in areas prioritizing wireless connectivity and low power use. Its compactness and cost-effectiveness make it suitable for diverse IoT projects, from remote monitoring to home automation and wearables. In *Figure 1.8*, we see an ESP32 with built-in LoRa communication and **Organic Light-Emitting Diode (OLED)** screen (the LoRa board is covered by the screen):

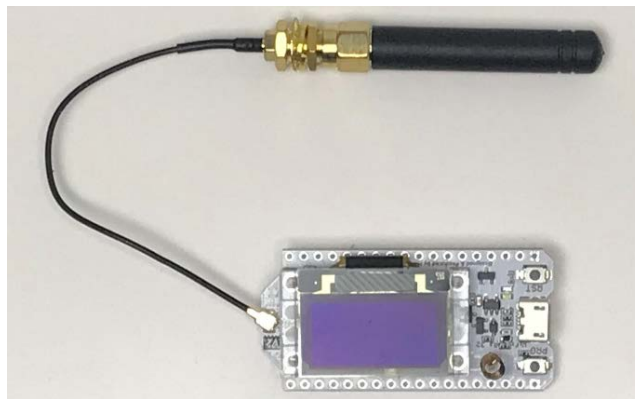


Figure 1.8 – ESP32 with the OLED screen and LoRa

- **Arduino Nano RP2040 Connect:** The Arduino Nano RP2040 Connect was developed to integrate Raspberry Pi's RP2040 microcontroller into a compact, feature-rich Arduino board, offering a unique blend of performance and connectivity for IoT projects and embedded AI solutions. It combines the dual-core processing power of the RP2040 with onboard Wi-Fi and Bluetooth, a six-axis IMU, microphone, and RGB LED, targeting a seamless IoT development experience. Its compatibility with the Arduino Cloud platform simplifies project management with capabilities such as over-the-air updates.

Now that we have explored some of the alternatives to the Raspberry Pi, we will now turn our focus to expanding the functionality of the Raspberry Pi using HATs.

Looking at the power of HATs

Raspberry Pi HATs are add-on boards that extend the capabilities of the Raspberry Pi, offering a wide range of functionality for various applications. These HATs provide an easy and convenient way to enhance the Raspberry Pi's capabilities by adding features such as sensors, actuators, displays, communication interfaces, and more. In this section, we will explore two notable HATs: the Pibrella HAT and the Raspberry Pi Sense HAT.

Pibrella HAT

The Pibrella HAT is an excellent beginner-friendly board designed to introduce electronics and programming concepts to users of all ages. It features buttons, LEDs, and a buzzer, providing a hands-on experience for physical computing projects with the Raspberry Pi. With its intuitive interface and Python library, the Pibrella HAT offers a great starting point for learning and prototyping.

Despite being designed for the first version of the Raspberry Pi and thus having only a 26-pin GPIO connection, the Pibrella HAT can still be used with the current 40-pin versions of the Raspberry Pi. We can see the Pibrella HAT being attached to a Raspberry Pi 3B in *Figure 1.9*:

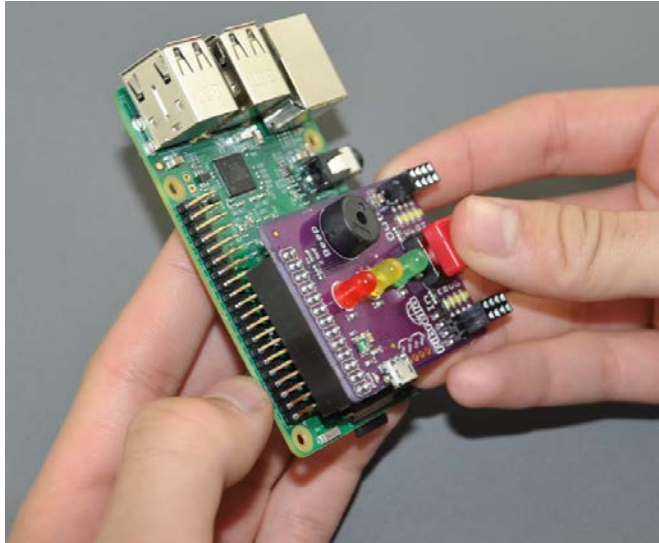


Figure 1.9 – Attaching the Pibrella to a Raspberry Pi 3B

With the Pibrella HAT, users can explore the fundamentals of physical computing, learn about input and output interactions, and gain hands-on experience in programming with the Raspberry Pi.

The Raspberry Pi Sense HAT

The Raspberry Pi Sense HAT (as shown in *Figure 1.10*) is an impressive add-on board designed to enhance the capabilities of the Raspberry Pi for sensing and environmental monitoring applications. Equipped with a variety of sensors, including temperature, humidity, pressure, gyroscope, accelerometer, and magnetometer, the Sense HAT allows users to gather data from the surrounding environment with ease. Its LED matrix display offers a visual output, enabling users to display real-time information, including animations:

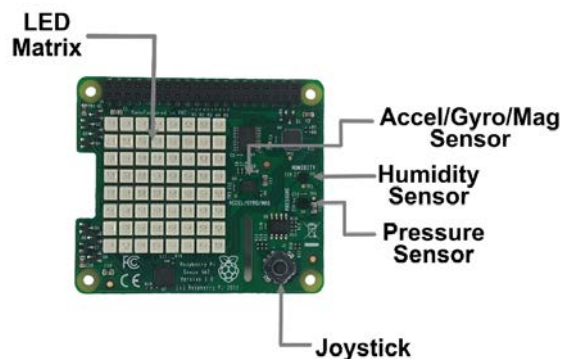


Figure 1.10 – Raspberry Pi Sense HAT

In addition to its wide range of applications, the Sense HAT also plays a vital role in the AstroPi program, which enables students to run their own experiments on the International Space Station. With its sensors and compact form factor, the Sense HAT is an ideal choice for collecting valuable data in a space station.

In this chapter, we begin writing code using a Raspberry Pi 5 and Sense HAT. We will write code to fetch the current temperature from Sense HAT's built-in temperature sensor and display it as a scrolling message on the Sense HAT's dot matrix display.

Before we do this, however, we will continue our exploration of the Raspberry Pi by looking at the various operating systems available for the Raspberry Pi.

Investigating operating systems for the Raspberry Pi

While the official Raspberry Pi OS (formerly Raspbian) is the most widely used operating system for the Raspberry Pi, there is also support for various other operating systems. These options range from specialized systems tailored for specific applications, such as audio players including Volumio, and more general-purpose systems, such as Ubuntu and Raspberry Pi OS itself. In this section, we explore a selection of these operating systems and their unique features. They are as follows:

- **Volumio:** For those of us aiming to build a networked audio system accessible via a computer or smartphone, Volumio is a fitting choice. It transforms the Raspberry Pi into a headless audio player, rendering a keyboard or mouse unnecessary. This system connects to our audio files via a USB or network, with the option to enhance the audio output quality through an add-on HAT. Notably, Volumio includes a Spotify plugin, letting us stream music over a sound system.
- **PiFM radio transmitter:** This system image turns the Raspberry Pi into an FM transmitter, broadcasting audio files to any standard FM radio receiver. We simply need to attach a wire to one of the GPIO pins to function as an antenna, creating a surprisingly robust FM signal.
- **Stratux:** Stratux is open-source aviation software that can transform a Raspberry Pi into a powerful **Automatic Dependent Surveillance-Broadcast (ADS-B)** receiver. ADS-B is a modern aviation standard that allows aircraft to share their position, velocity, and other flight data with air traffic control and other aircraft. By installing Stratux on a Raspberry Pi and pairing it with additional hardware, we can create our own ADS-B ground station. This enables us to receive real-time data from aircraft in our vicinity, including flight trajectories, altitude, and speed.
- **RetroPie:** For gaming buffs, RetroPie converts our Raspberry Pi into a retro gaming console. It effectively emulates several vintage gaming consoles and computers such as Amiga, Apple II, Atari 2600, and the Nintendo Entertainment System from the early 1980s.

- **OctoPi:** For those of us involved in 3D printing, OctoPi transforms a Raspberry Pi into a server for a 3D printer, offering control and monitoring remotely through a network connection.



Figure 1.11 – OctoDash main screen

A companion to OctoPi is **OctoDash** (as shown in *Figure 1.11*), a touchscreen interface that provides an easy-to-use, visually appealing control panel for managing and monitoring 3D printing tasks. Using OctoPi with OctoDash can make the interaction with a 3D printer more flexible and efficient.

- **Ubuntu for Raspberry Pi:** Ubuntu, a leading open-source Linux distribution company, offers a platform for Raspberry Pi. Among Ubuntu's key strengths is its compatibility with the **Robot Operating System (ROS)**, an adaptable framework for writing robot software. With ROS on Ubuntu installed on a Raspberry Pi, we can engage in robotics projects, from simple hobbyist endeavors to sophisticated industrial automation systems. We will begin our exploration of ROS in *Chapter 11*.
- **Raspberry Pi OS:** Formerly known as Raspbian, Raspberry Pi OS is the most widely used operating system for Raspberry Pi due to its direct compatibility, lightweight design, and ease of use. Tailored specifically for Raspberry Pi, this operating system is abundant in educational software and programming tools, thereby aligning with Raspberry Pi's mission of promoting learning in computer science and related fields. Installing Raspberry Pi OS is straightforward with the Raspberry Pi Imager, a tool that simplifies the process of burning the system image to a microSD card. Once installed, Raspberry Pi OS provides a graphical user interface along with a comprehensive suite of programming, internet, and multimedia apps, thereby making it a versatile choice for a wide range of Raspberry Pi projects.

Now that we have a general understanding of the operating systems available for the Raspberry Pi, let's explore using the Raspberry Pi in the field that this book is based on – IoT.

Using the Raspberry Pi for IoT

IoT has revolutionized the way we interact with technology, giving everyday objects the ability to communicate, automate tasks, and generate invaluable data for use over the internet. The Raspberry Pi is central to many of these systems. Its versatility and robust processing capabilities enable this powerful single-board computer to function both as a data processor and a controller for physical devices.

Uniquely equipped with features such as GPIO pins, USB ports, and Wi-Fi capabilities, the Raspberry Pi is a cost-effective and instrumental tool in implementing IoT solutions. The Raspberry Pi is especially valuable for processing real-time data and managing hardware. The following is a look at examples of Raspberry Pi-based IoT systems.

Utilizing web services for IoT applications

One of the distinct advantages of using the Raspberry Pi in IoT applications is its capacity to process real-time data and control other hardware elements based on this data. This capability becomes particularly useful when the data is sourced from the internet. The applications for such a setup are vast, from environmental monitoring to health tracking, traffic management, and more.

The following example introduces a specific instance of such an application – a Raspberry Pi-driven IoT system that suggests appropriate clothing based on the current weather conditions (*Figure 1.12*):

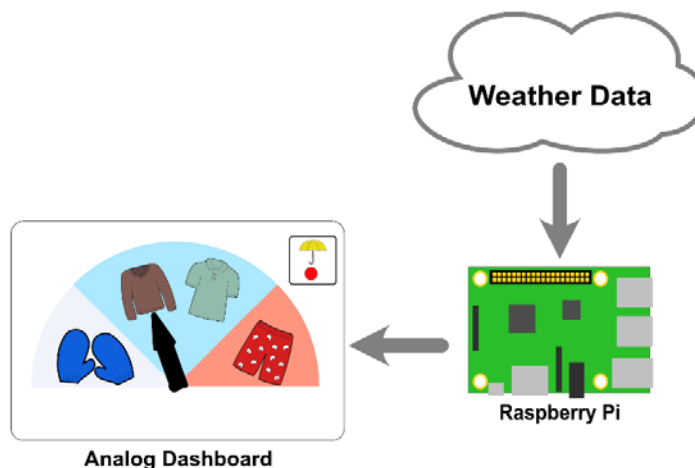


Figure 1.12 – Analog dashboard for weather information

Built in the first edition of *Internet of Things Programming Projects*, this application utilizes an intuitive analog dashboard to suggest appropriate clothing based on the weather conditions. It includes an LED, which indicates if an umbrella will be necessary.

In addition to being a weather-based application, we can modify this Raspberry Pi-powered IoT solution for various applications that involve collecting real-time data from a web service and representing it as analog data and an LED.

Here are a few examples:

- **Traffic density monitor:** Leveraging similar concepts as described in *Figure 1.12*, this application would gather real-time traffic data from a city's traffic monitoring API. The analog meter would show the current traffic density, and the LED could flash to indicate heavy traffic or congestion on a specific route. This allows commuters to choose the best routes and times to travel.
- **Health monitoring system:** With data sourced from a health API or smart health devices, the analog meter could display heart rate, blood pressure, or any other vital statistic. The LED would act as an immediate visual alert for abnormal values, prompting immediate medical attention if necessary.
- **Water quality monitor:** The IoT device could connect to a web service that receives data from water quality sensors in a river, lake, or ocean. The analog meter could display metrics such as pH level and the LED could flash when the readings indicate potentially hazardous conditions.
- **Agricultural monitor:** Connected to a web service that pulls data from sensors in a farm field (such as soil moisture, temperature, etc.), the analog meter could display current conditions, while the LED could indicate when conditions are ripe for irrigation or if there's a risk of frost.

Moving beyond metered dashboards, the powerful features of Raspberry Pi are particularly effective in robotics. The Raspberry Pi acts as the *brain* for these systems, managing tasks such as sensor data analysis, decision-making, and motor control. The integration of IoT and robotics has resulted in major advancements in various fields including security, automation, and surveillance. An example of this is **T.A.R.A.S.** (short for **This Amazing Raspberry-Pi Automated Security agent**), an automated security agent powered by Raspberry Pi and IoT principles, as presented in the first edition of *Internet of Things Programming Projects*.

Re-introducing T.A.R.A.S. – an IoT-based robotics project

T.A.R.A.S., a backronym named in honor of one of the author's business mentors, Taras, serves as an automated security guard.

This application of IoT in robotics exhibits how a Raspberry Pi can manage sensory and motor functions. It uses **Message Queuing Telemetry Transport (MQTT)** messages, a lightweight, publish-subscribe network protocol, enabling seamless communication between devices. We can see a graphic of T.A.R.A.S. in *Figure 1.13*:

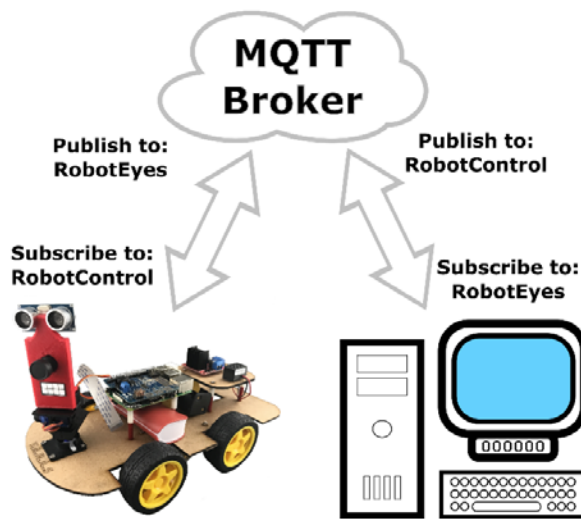


Figure 1.13 – T.A.R.A.S. communicates over the internet by sending and receiving MQTT messages

Alongside sensory inputs, LED, and buzzer outputs, T.A.R.A.S. encapsulates the dynamic range of possibilities within IoT. In the previous edition, we built a web-based controller for T.A.R.A.S., integrating a host of skills acquired throughout the book.

In this edition of *Internet of Things Programming Projects*, we say goodbye to T.A.R.A.S. as he has taken a package and retired from the security field. We will say hello to **A.R.E.S.**, or **Advanced Robotic Eyes for Security**. Named after the author's beloved late dog, A.R.E.S. will take security and mobility to a new level by incorporating vision recognition and ROS.

Getting started with Raspberry Pi development

Our Raspberry Pi can serve as a robust development platform ideal for various programming environments. The default Linux-based Raspberry Pi OS is equipped with Python, Scratch, and many other tools that cater to both beginners and seasoned programmers. It's particularly effective for IoT projects and edge computing, thanks to its small form factor. As developers, we can use it for software development, web hosting, home automation projects, and prototyping, making it a versatile and accessible tool for innovation.

In this book, we will develop IoT projects using the Raspberry Pi, specifically utilizing the Raspberry Pi 5 with 8 GB of RAM for optimal performance. While other versions of the Raspberry Pi may be sufficient, the Raspberry Pi 5 with 8 GB of RAM is currently the most powerful model available. Our projects will encompass an array of exciting applications, including an analog metered weather dashboard, an internet-connected home security system, an IoT remote monitoring station, and A.R.E.S. our advanced IoT-enabled robot car. In the remainder of this chapter, we will familiarize ourselves with reading sensory data from a Raspberry Pi Sense HAT.

Before diving into development though, it's crucial to familiarize ourselves with the development tools that are available for the Raspberry Pi.

Raspberry Pi development tools

The following is a range of development tools available for our Raspberry Pi projects. It's important to note that not all these programs come pre-installed with Raspberry Pi OS and may require manual installation:

- **Python:** Python is a high-level interpreted programming language for general-purpose programming. Its simple syntax and readability make it excellent for beginners, and its powerful libraries and versatility also suit advanced users for complex projects. Python comes pre-installed on Raspberry Pi OS.
- **Thonny:** Thonny is an IDE for Python that comes pre-installed with Raspberry Pi OS. It's easy to use for beginners and includes features such as **step-by-step debugging** and **error highlighting**. It's also robust enough for advanced users, offering comprehensive tools for more complex coding. We will utilize Thonny for our projects in this book.
- **Scratch:** Scratch is a block-based visual programming language aimed at children. With Scratch, children can create animation and games all while learning the basics of programming.
- **Greenfoot and BlueJ:** These are two IDEs for Java. They are primarily used in education and are designed to help beginners get a grip on object-oriented programming.
- **Mu** – This is a Python editor for beginner programmers. It's designed to be simple and easy to understand.
- **Geany:** It is a lightweight and highly customizable IDE that supports a wide range of programming languages including C, Java, PHP, HTML, Python, Perl, Pascal.
- **Wolfram Mathematica and Wolfram Language:** Wolfram provides a high-level language and interactive environment for programming, mathematical visualization, and general computation.
- **Node-RED:** This is a flow-based open source tool used for visual programming. Node-RED allows access to APIs and online services.
- **GCC and GDB:** The **GNU Compiler Collection (GCC)** (including gcc and g++) and the **GNU Debugger (GDB)** allow us to compile and debug code written in languages such as C and C++.

Apart from these, we can install other software development tools from the Raspberry Pi OS repositories using the apt package manager or download and install them manually. For example, we might want to install Git for version control, Docker for containerization, or Visual Studio Code for a more advanced development environment.

The Raspberry Pi supports a wide array of development tools, catering to various programming needs. By looking at these tools, we've underscored the Raspberry Pi's adaptability and capacity for innovative projects.

Raspberry Pi and Sense HAT development

It is a great idea to start our journey into IoT development by learning how to write simple code for the Raspberry Pi Sense HAT. The Sense HAT serves as an excellent IoT device, equipped with a range of sensors and an LED matrix display that can be harnessed to create innovative applications.

For our coding examples, we will securely mount our Raspberry Pi 5 into a case specifically designed for this book. In *Figure 1.14*, we can see a CAD render of the case that we will be using. The case features a front cover that exposes the dot matrix LED screen of the Sense HAT. It also includes a circular vent to ensure proper heat dissipation inside the case. On the back part of the case, we have incorporated a GoPro-style hook, providing us with the flexibility to mount the Raspberry Pi and Sense HAT using any GoPro stand. Files to build the case are available from the book's GitHub repository. Using a standard FDM 3D printer to make the case should be sufficient; however, better quality and durability are obtained using a liquid resin printer and engineering resins such as Siraya Tech Blu. *Figure 1.14* shows what the assembled case will look like with the printed stem and base plate in place of a GoPro stand:



Figure 1.14 – Custom-designed Sense HAT case for Raspberry Pi 5

We can find all the necessary files for 3D printing (.stl files) in the `Build Files` folder within the book's GitHub repository. It is worth noting that most Raspberry Pi cases do not account for the accommodation of HATs, making our Raspberry Pi/Sense HAT case an excellent accessory for this book. However, it is important to mention that the case is not mandatory to complete the exercises in this book. Furthermore, if we decide not to purchase a Sense HAT, we can still run the examples by utilizing the Sense HAT simulator available for the Raspberry Pi OS.

Before we start writing code, we'll set up the development environment on our Raspberry Pi. Once our environment is ready, we'll dive right into coding.

Let's get started!

Setting up our development environment

In this section, we will go through the process of setting up Thonny for development with the Raspberry Pi and Sense HAT. Thonny is a beginner-friendly IDE that offers a user-friendly interface. With Thonny, we can easily write, test, and debug our code, allowing us to focus on the development process without unnecessary complexity.

We will use a Python virtual environment for our development. As there are libraries that only work with the root installation of Python, we will use system packages in our Python virtual environment. To do so, we do the following:

1. On our Raspberry Pi 5, we open a Terminal application by clicking on the fourth icon on the top-left menu (*Figure 1.15*):

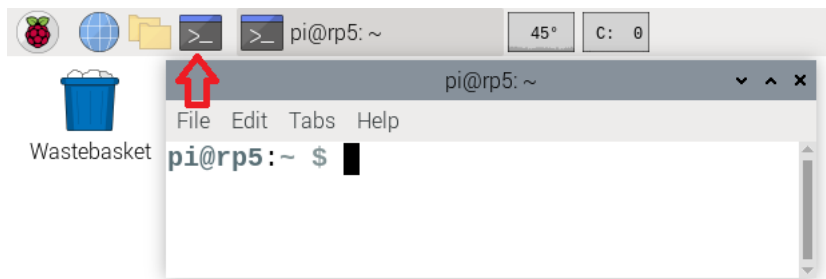


Figure 1.15 – Opening the Terminal application (indicated by the red arrow)

2. To store our project files, we create a new directory with the following:

```
mkdir Chapter1
```

3. We then navigate to the new directory with the following:

```
cd Chapter1
```

4. We create a new Python virtual environment for our project with the following:

```
python -m venv ch1-env --system-site-packages
```

5. With this command, we create a new Python virtual environment called `ch1-env` and enable access to the system site packages. This allows the virtual environment to inherit packages from the global Python environment, which can be useful when certain libraries are installed system wide. Once the environment is set up, we can activate it and begin installing project-specific packages without affecting the global Python environment.
6. With our new Python virtual environment created, we source into it with the following command:

```
source ch1-env/bin/activate
```


7. Sourcing into the new Python virtual environment activates the environment, allowing us to work with its specific settings and installed packages isolated from the global Python installation. Our Terminal application should now show that we are using the `ch1-env` Python virtual environment:



```
pi@rp5:~ $ mkdir Chapter1
pi@rp5:~ $ cd Chapter1
pi@rp5:~/Chapter1 $ python -m venv ch1-env --system-site-packages
pi@rp5:~/Chapter1 $ source ch1-env/bin/activate
(ch1-env) pi@rp5:~/Chapter1 $
```

Figure 1.16 – Terminal using `ch1-env` environment

8. For our project, we need the Sense HAT and the Sense HAT emulator libraries. The Sense HAT library is pre-installed in our Python virtual environment; however, the Sense HAT emulator requires manual installation via a Terminal command:

```
pip install sense-emu
```

9. Installing the Sense HAT emulator library lets us run code with either the actual Sense HAT or the Sense HAT emulator. We can close the Terminal with a specific command:

```
exit
```

10. We are now ready to load up Thonny. We do so by clicking on the **Menu** icon in the Raspberry Pi taskbar, navigating to the **Programming** category, and selecting **Thonny** (Figure 1.17):

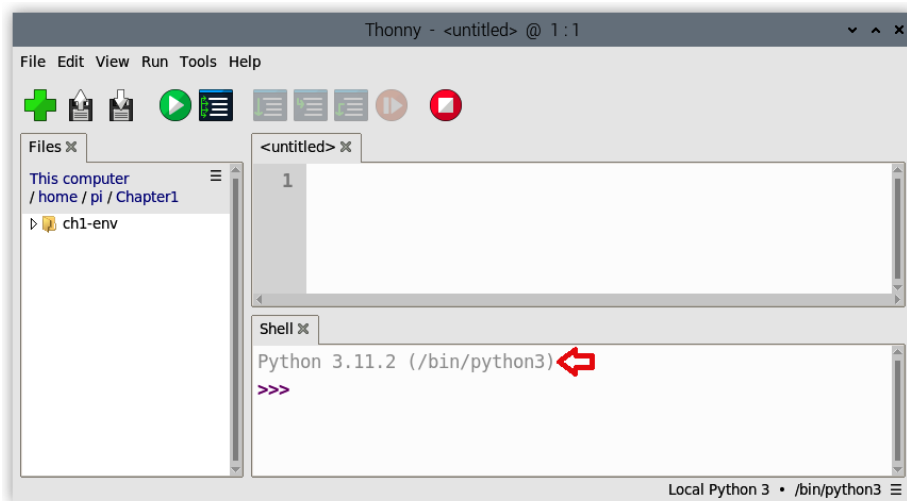


Figure 1.17 – Thonny IDE interface

11. By default, Thonny uses the Raspberry Pi's built-in version of Python (red arrow in *Figure 1.17*). For our project, we will use the Python virtual environment we just created. To start, we need to view the project files by clicking on **View** and selecting **Files** if it is not already selected (we may have to switch to regular mode first by clicking on the **Switch to regular mode** tab at the top-right side of the screen).
12. In the **Files** section, we locate the `ch1-env` directory.
13. We then right-click on the folder and select the **Activate virtual environment** option:

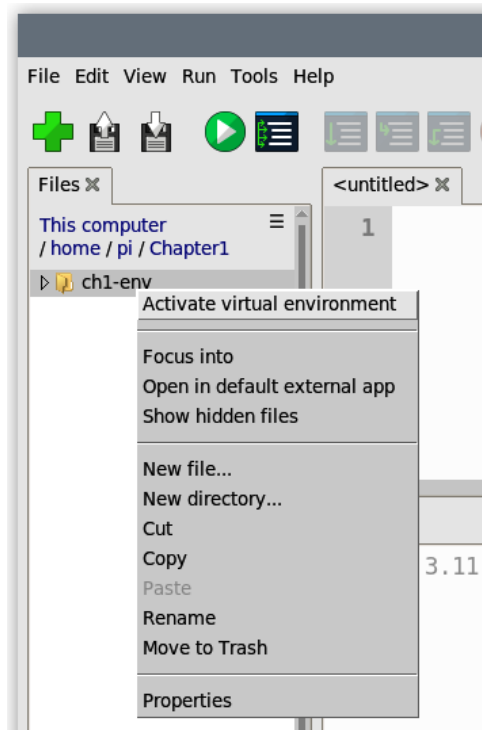


Figure 1.18 – Activating `ch1-env` virtual environment in Thonny

With our project folder created, our Python virtual environment set up and activated, and the Sense HAT emulator package installed, we may now start writing code.

Sense HAT development – reading sensor data

The goal for our first project is to create a program that reads sensor information from the Sense HAT and displays it in the Thonny Shell. Before we can read sensory information, however, we must ensure that we have the Sense HAT properly connected to our Raspberry Pi.

To start, follow these steps:

1. If not already running, we launch Thonny by clicking on the **Menu** icon in the Raspberry Pi taskbar, navigating to the **Programming** category, and selecting **Thonny**. We activate the `ch1-env` virtual environment if not already activated.
2. We then create a new tab by selecting **File** and then **New** or by hitting `Ctrl+N` on the keyboard:

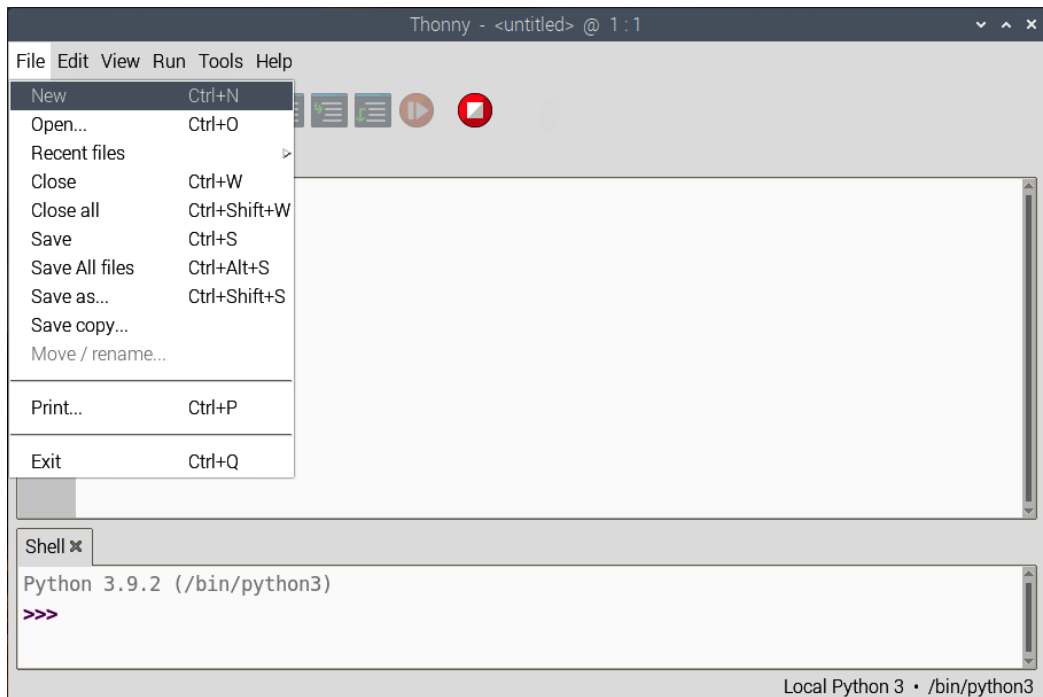


Figure 1.19 – Creating a new file in Thonny

3. Inside the new tab, we enter the following code:

```
from sense_hat import SenseHat

sense_hat = SenseHat()
temp = sense_hat.get_temperature()
humidity = sense_hat.get_humidity()
press = sense_hat.get_pressure()
accel = sense_hat.get_accelerometer_raw()
gyroscope = sense_hat.get_gyroscope_raw()

print("Temperature: {:.2f}°C".format(temp))
print("Humidity: {:.2f}%".format(humidity))
```

```
print("Pressure: {:.2f} millibars".format(press))

print("Accelerometer Data: x={:.2f}, y={:.2f}, z={:.2f}".
      format(accel['x'], accel['y'], accel['z']))
print("Gyroscope Data: x={:.2f}, y={:.2f}, z={:.2f}".
      format(gyroscope['x'], gyroscope['y'], gyroscope['z']))
```

4. For those of us using the Sense HAT simulator, we simply need to change the first line of code to the following:

```
from sense_emu import SenseHat
```

We save the code with a descriptive name such as `sensor-test.py`. Before we run the code, let's break it down to understand it:

- I. Our code starts by importing the `SenseHat` class from the `sense_hat` library.
 - II. An instance of the `SenseHat` class is created and assigned to the variable `sense_hat`.
 - III. The `get_temperature()` method is called on the `sense_hat` object to retrieve the temperature data and assigned to the `temp` variable.
 - IV. The `get_humidity()` method is called on the `sense_hat` object to retrieve the humidity data and assigned to the `humidity` variable.
 - V. The `get_pressure()` method is called on the `sense_hat` object to retrieve the air pressure data and assigned to the `press` variable.
 - VI. The `get_accelerometer_raw()` method is called on the `sense_hat` object to retrieve the raw accelerometer data, which includes values for the *x*, *y*, and *z* axes. The data is assigned to the `accel` variable.
 - VII. The `get_gyroscope_raw()` method is called on the `sense_hat` object to retrieve the raw gyroscope data, which also includes values for the *x*, *y*, and *z* axes. The data is assigned to the `gyroscope` variable.
 - VIII. The obtained data is then printed out with appropriate formatting using the `print()` function and formatted string placeholders (`{:.2f}`) to display the values with two decimal places.
5. We run the code by either clicking on the green **Run** button, hitting *F5* on the keyboard, or clicking on the **Run** menu option at the top and then **Run current script**:

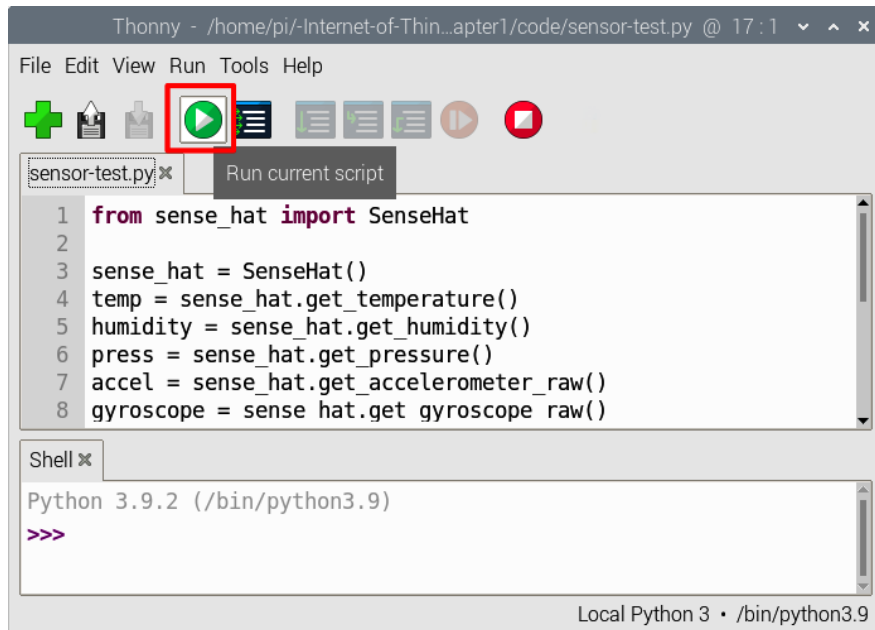


Figure 1.20 – Running a program in Thonny

After running the code, we should observe a message like the following in the Shell:

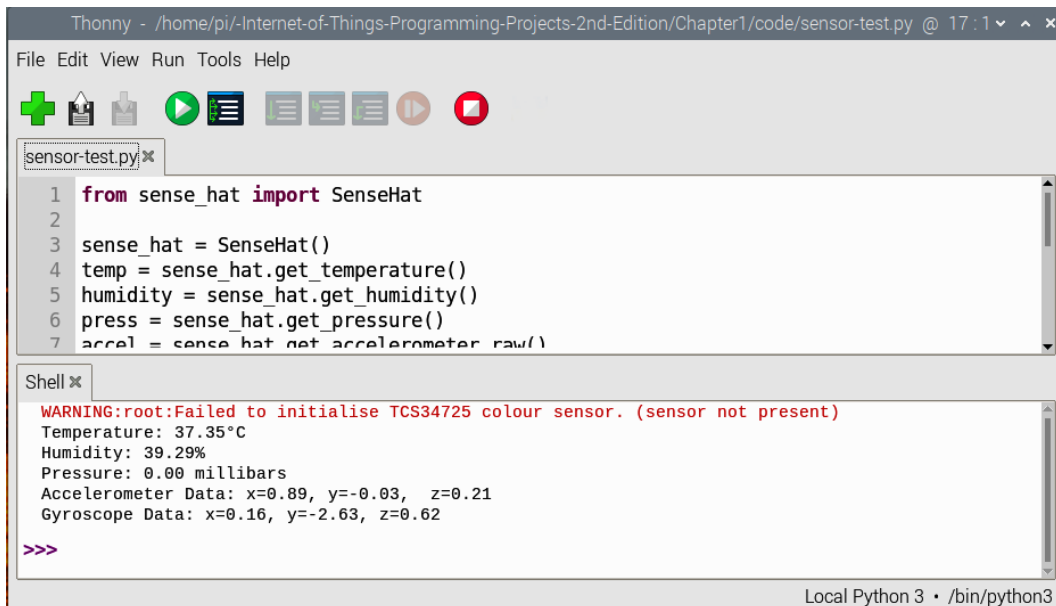


Figure 1.21 – Results after running sensor-test.py

For those of us using the Sense HAT emulator, the values displayed will be determined by the slider values set in the emulator.

Important notes

We can disregard the warning since the Sense HAT does not include a TCS34725 color sensor. It is likely that the warning is caused by an internal issue in the Sense HAT Python library. Our code is not affected by this warning and can continue to run as intended.

It's important to note that the temperature readings are influenced by the heat generated by the Raspberry Pi, resulting in higher values compared to the actual room temperature. The humidity readings are also affected. However, the accelerometer and gyroscope values accurately reflect the position and orientation of the Sense HAT and can be observed to change when the case is swiveled in different directions. To see updated values from the accelerometer and gyroscope, the code must be run again after swiveling the case.

Now that we have learned how to read sensory data from Sense HAT, let's shift our focus to the dot matrix screen. In the next section, we will explore creating a simple animation using the dot matrix display.

Sense HAT development – creating animations

In this section, we will create an explosion animation using Sense HAT and the Raspberry Pi. We will use Thonny as our development environment to write and execute the code.

To start, follow these steps:

1. If not already running, we launch Thonny by clicking on the **Menu** icon in the Raspberry Pi taskbar, navigating to the **Programming** category, and selecting **Thonny**. We activate the `ch1-env` virtual environment if not already activated.
2. We then create a new tab by selecting **File** and then **New** or by hitting `Ctrl + N` on the keyboard.
3. We start our code by entering our imports:

```
from sense_hat import SenseHat
import time
```

- I. We import the `SenseHat` module from the `sense_hat` library, allowing us to interact with the Sense HAT board.
- II. We also import the `time` module, which will be used for adding delays in our program.

4. We then set our variable declarations:

```
sense_hat = SenseHat()
R = [255, 0, 0] # Red
O = [255, 165, 0] # Orange
Y = [255, 255, 0] # Yellow
B = [0, 0, 0] # Black
frame1 = [
    B, B, B, B, B, B, B, B,
    B, B, B, B, B, B, B, B,
    B, B, B, R, R, B, B, B,
    B, B, R, R, R, R, B, B,
    B, B, R, R, R, R, B, B,
    B, B, B, R, R, B, B, B,
    B, B, B, B, B, B, B, B,
    B, B, B, B, B, B, B, B
]
frame2 = [
    B, B, Y, O, O, Y, B, B,
    B, Y, O, R, R, O, Y, B,
    Y, O, R, Y, Y, R, O, Y,
    O, R, Y, B, B, Y, R, O,
    O, R, Y, B, B, Y, R, O,
    Y, O, R, Y, Y, R, O, Y,
    B, Y, O, R, R, O, Y, B,
    B, B, Y, O, O, Y, B, B
]
frame3 = [
    O, R, R, Y, Y, R, R, O,
    R, Y, O, O, O, O, Y, R,
    Y, O, B, B, B, B, O, Y,
    O, B, B, B, B, B, B, O,
    O, B, B, B, B, B, B, O,
    Y, O, B, B, B, B, O, Y,
    R, Y, O, O, O, O, Y, R,
    O, R, R, Y, Y, R, R, O
]
frames = [frame1, frame2, frame3]
```

- I. We create an instance of the SenseHAT class called `sense_hat`, enabling us to access the Sense HAT's functionality.

- II. We define color values for red, orange, yellow, and black, which will be used to create our animation.
 - III. We create `frame1` as a list of color values, representing the desired image on the LED matrix.
 - IV. We define `frame2` and `frame3` as additional frames.
 - V. We create a list called `frames`, which holds the defined frames in a specific order.
5. To have our animation run continuously, we set up an infinite loop:

```
while True:
    for frame in frames:
        sense_hat.set_pixels(frame)
        time.sleep(0.5)
    sense_hat.clear()
    time.sleep(0.2)
```

- I. Within the loop, we iterate through each frame in the `frames` list.
 - II. We set the pixels of the LED matrix to the current frame using `sense_hat.set_pixels(frame)`, displaying the corresponding image.
 - III. We introduce a brief pause of 0.5 seconds using `time.sleep(0.5)` to control the animation speed.
 - IV. After displaying all frames, we clear the LED matrix using `sense_hat.clear()`, preparing for the next iteration.
 - V. We add a slight delay of 0.2 seconds using `time.sleep(0.2)` before starting the animation loop again, creating the explosion pattern.
6. For those of us using the Sense HAT simulator we simply need to change the first line of code to the following and open the Sense HAT simulator on our Raspberry Pi:

```
from sense_emu import SenseHat
```

7. We save the code with a descriptive name such as `animation-test.py`.

We run the code by clicking on the green **Run** button, hitting *F5* on the keyboard, or clicking on the **Run** menu option at the top and then **Run current script**. We should observe an explosion animation on our Sense HAT or in the Sense HAT emulator:

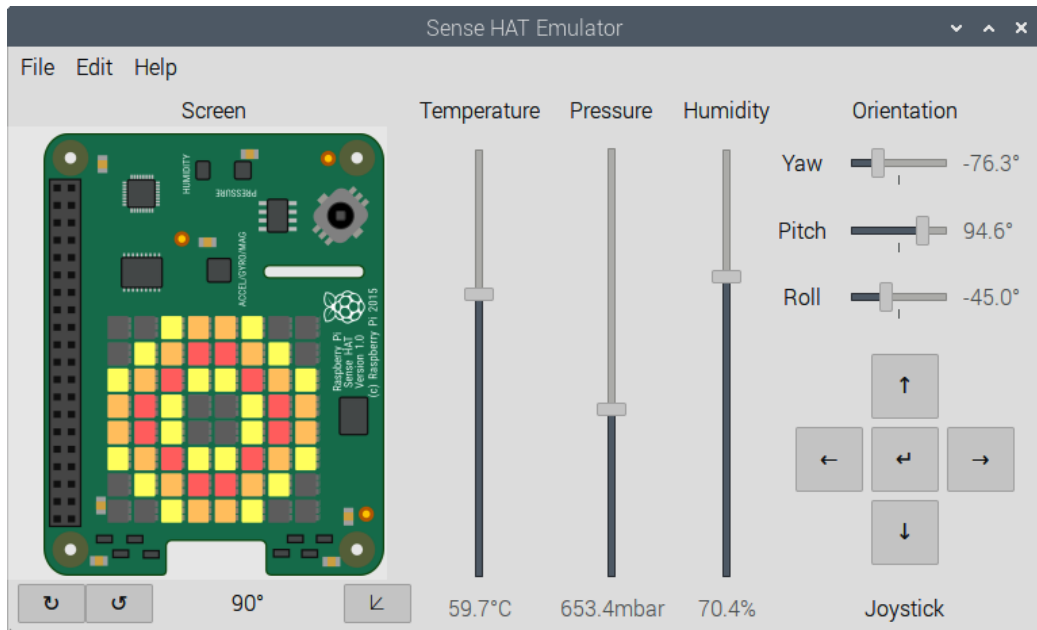


Figure 1.22 – Frame from explosion animation on the Sense HAT emulator

In this section, we explored creating animations with Sense HAT and Raspberry Pi. We learned how to write code that displays a sequence of frames on the LED matrix, resulting in an explosion animation. For our final programming project in this chapter, we will create a scrolling message on the Sense HAT's dot matrix screen. This message will dynamically show temperature, humidity, and air pressure data, providing us with valuable environmental insights in a visually appealing way.

Sense HAT development – creating a scrolling environmental data display

Building upon our previous projects, where we gained experience in reading sensory data from Sense HAT and creating animations on its LED matrix, we will now dive into the creation of a scrolling text application using the Sense HAT's dot matrix display.

To start, follow these steps:

1. If not already running, we launch Thonny by clicking on the **Menu** icon in the Raspberry Pi taskbar, navigating to the **Programming** category, and selecting **Thonny**. We activate the `ch1-env` virtual environment if not already activated.
2. We create a new tab by selecting **File** and then **New** or by hitting `Ctrl + N` on the keyboard.
3. We start our code by importing the libraries we need:

```
from sense_hat import SenseHat
import time
```

- I. We import the `SenseHat` module from the `sense_hat` library, allowing us to interact with the Sense HAT board. For those of us using the Sense HAT emulator, we would use the `sense_emu` library instead.
 - II. We also import the `time` module, which will be used for adding delays in our program.
4. We then set our variable declarations in our code:

```
sense_hat = SenseHat()
speed = 0.05
sense_hat.set_rotation(270)
red = [255, 0, 0]
green = [0, 255, 0]
```

- I. We create an instance of the `SenseHAT` class we call `sense_hat`.
 - II. We set the scrolling speed of the messages on the LED matrix using the variable `speed`.
 - III. The line `sense_hat.set_rotation(270)` adjusts the orientation of the Sense HAT's LED matrix so that it matches the orientation of the Raspberry Pi in the custom case.
 - IV. We define two color variables, `red` and `green`, for the text color.
5. To have our scrolling message continuously play, we create an infinite loop:

```
while True:
    sense_hat.show_message("Temperature: %.1fC, " %
                           sense_hat.get_temperature(),
                           scroll_speed=speed,
                           text_colour=green)
    sense_hat.show_message("Humidity: %.1f%%, " %
                           sense.get_humidity(),
                           scroll_speed=speed,
                           text_colour=green)

    sense_hat.show_message("Pressure: %.1fhPa" %
                           sense.get_pressure(),
                           scroll_speed=speed,
                           text_colour=red)
    time.sleep(1)
```

In the infinite loop, we do the following:

- I. We display the temperature reading on the LED matrix, formatted with one decimal place.
- II. We display the humidity reading on the LED matrix, formatted with one decimal place.

- III. We display the pressure reading on the LED matrix, formatted with one decimal place.
 - IV. We pause for one second before the next iteration of the loop.
- 6. We save the code with a descriptive name such as `sensor-scroll.py`.
 - 7. We run the code by clicking on the green **Run** button, hitting *F5* on the keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.

We should observe a scrolling message of sensory data displayed on the dot matrix screen of our Sense HAT (or Sense HAT emulator).

In this exercise, we created a scrolling text application on the Sense HAT, displaying real-time environmental data. This project enhanced our skills in data acquisition from sensors and visual presentation, using the Sense HAT's LED matrix for effective data visualization. Completing this final chapter project, we're now equipped to tackle more advanced IoT applications with the Sense HAT and Raspberry Pi.

Summary

In this chapter, we began our journey into the world of IoT projects with the Raspberry Pi. We explored the various Raspberry Pi models, their distinct characteristics, and their significance in IoT development. We took a brief look at the alternatives to the Raspberry Pi for IoT applications and explored Raspberry Pi HATs, such as the Pibrella HAT and the Raspberry Pi Sense HAT.

Additionally, we looked at operating systems compatible with the Raspberry Pi, outside of the official Raspberry Pi OS. We highlighted the versatility and power of the Raspberry Pi in IoT applications, emphasizing its ability to process real-time data and control physical devices. The chapter also introduced the Raspberry Pi as a robust development platform, equipped with pre-installed tools for software development.

At the end of the chapter, we engaged in practical programming projects that utilized Sense HAT and explored its potential in IoT applications. These projects included a scrolling environmental data display, providing hands-on experience in extracting sensory information, and creating dynamic visual displays.

This chapter equipped us with the knowledge we may use to make informed choices when developing IoT projects. Understanding the strengths of different Raspberry Pi models and alternatives prepares us to choose the right device for specific applications. Familiarity with Raspberry Pi HATs and various operating systems expands our toolkit. In this chapter, we unlocked the potential of the Raspberry Pi by understanding its data and control capabilities. Our Sense HAT practice prepared us for more complex projects in the future.

Looking ahead, in the next chapter, we will harness the power of web services as we continue to build more sophisticated IoT applications.

Harnessing Web Services with the Raspberry Pi

In this chapter, we will begin to write code for web services to turn our Raspberry Pi into an **Internet of Things (IoT)** device. Using Python, we will design programs that pull data from online resources and use the data to create visuals on Sense HAT's dot-matrix display. The practical examples we will cover in this chapter will serve as a building block for more advanced IoT web services development.

We begin by exploring the world of web services – understanding their role and how we can exploit them to our advantage. We may think of web services as the lifeblood of the internet, circulating vital data across the digital world. Understanding web services isn't just about adding another tool to our toolkit; it's about unlocking a world filled with limitless potential.

As we advance, we'll transform theoretical knowledge into practical application through a series of programming projects. These projects are specifically designed to utilize the advanced web services offered by providers such as Alpha Vantage and OpenWeather. Using the versatile Raspberry Pi Sense HAT, or its emulator, we'll construct applications including a scrolling stock ticker, a weather information display, and even a GO-NO-GO decision-maker for youth baseball games. These projects don't just teach us how to use technology; they immerse us in the fundamental principles of IoT.

Consider the vast array of opportunities that lie ahead with web services and the Raspberry Pi. Today, we're starting with a simple stock ticker. Tomorrow, we might be developing a device that notifies us whenever our favorite team scores or even assists us in navigating through traffic. This chapter is our first step toward exploring a broader and more thrilling universe of IoT innovation.

In this chapter, we will cover the following topics:

- Exploring web services
- Creating a scrolling stock ticker application
- Developing weather display applications

Let's begin!

Technical requirements

The following is required to complete this chapter:

- A Raspberry Pi 5 with either 4 GB or 8 GB of RAM (preferred); however, a late-model Raspberry Pi such as the Raspberry Pi 4 may be used.
- Raspberry Pi Sense HAT (Raspberry OS emulator may be used instead).
- Keyboard, mouse, and monitor.
- Access to a 3D printer or 3D printing service for the custom stand.
- The GitHub repository for the chapter, located at <https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter2>.
- A general knowledge of programming. We will be using the Python programming language in this book.

Exploring web services

Imagine remotely controlling our home's devices from our smartphone – this amazing convenience is powered by web services, invisible messengers seamlessly connecting our digital world.

Web services form an integral part of today's internet infrastructure. They allow for the seamless exchange of data between different software applications over the internet. As a result, they are an essential tool for IoT applications, including our Raspberry Pi IoT projects. With web services, we can draw on the wealth of data available online and bring it into the physical world using our Raspberry Pi.

At their core, web services are software interfaces that enable one software system to interact with another over a network. This interaction is typically done via specific protocols, such as **Representational State Transfer (REST)** or **Simple Object Access Protocol (SOAP)**. To get a sense of the power of web services, consider the example shown in *Figure 2.1* of a service that provides critical sensory data on a patient in a hospital room:



Figure 2.1 – Client application receiving data from the FHIR web service

Here, we see a diagram representing the **Fast Healthcare Interoperability Resources** web service. **FHIR** (pronounced “fire”) is a standard developed by **Health Level Seven International (HL7)** for the electronic exchange of healthcare information. Web services such as FHIR offer universal accessibility, enabling healthcare professionals to access and share patient data from anywhere at any time. They provide superior interoperability, making them more efficient and practical than local network **Application Programming Interface (API)** calls, especially in the healthcare sector where data needs to be shared across different systems and devices.

In this section, we will look at the protocols used by web services. We will also explore some of the more popular web services available before we write code to call a simple web service using our Raspberry Pi.

Understanding approaches for web services

In the field of web services, two prominent approaches are the REST and SOAP protocols. RESTful web services use HTTP methods explicitly and are more straightforward and efficient, making them a popular choice for many developers. SOAP, on the other hand, is a protocol that permits programs running on siloed systems to communicate by using HTTP and its XML-based messaging system. SOAP is highly extensible and has strong support for security and reliability, making it suitable for complex applications.

In our upcoming projects, we will primarily utilize REST for interacting with web services. However, understanding SOAP provides a broader view of the landscape of web service interactions. We will start by exploring SOAP, its functionality, and its applicable scenarios before diving into REST-based or RESTful web services.

Using SOAP web services

To understand SOAP, we turn our attention to the custom ordering system illustrated in *Figure 2.2*. As this system was built in-house for a particular company’s business operations, it is a complex, enterprise-level application with significant requirements in terms of security and reliability. This ordering system represents an ideal use case for SOAP-based web services:



Figure 2.2 – Custom ordering application connecting to the server using SOAP

With SOAP's stateful nature, the application can manage complex transaction management that involves multiple steps, from inventory checks and payment processing to order confirmation. The SOAP protocol, with its built-in security features and error handling, enables the application to manage orders efficiently and securely.

As we can see in *Figure 2.2*, an XML file is passed through HTTPS in a SOAP transaction. This XML file, nestled within the SOAP message, plays a crucial role in communication. It carries detailed instructions that guide the server's actions. The structured format of XML enables complex data exchanges, allowing the custom ordering system to seamlessly interact with various other systems despite their possible diverse data formats.

If SOAP is the desired approach for enterprise-level applications, REST is preferred for public web services. REST's simplicity, lightweight nature, scalability, and resource-oriented approach make it ideal for creating user-friendly and scalable web services that interact with resources through standard HTTP protocols.

Investigating RESTful web services

RESTful web services are a key aspect of modern web applications, allowing for efficient communication between clients and servers using the REST architecture. RESTful web services employ HTTP methods such as GET, POST, PUT, and DELETE to interact with resources and are stateless. Unlike SOAP, which typically uses XML, RESTful services can support multiple data formats, including JSON and XML. This flexibility often leads to RESTful services being perceived as simpler to use and more adaptable than SOAP.

Several widely recognized public RESTful web services have gained popularity due to their functionality and ease of use. Among them is the Twitter API, which allows developers to access and interact with core Twitter data, including timelines, status updates, and other information. Another notable example is the Google Maps API, which provides developers with the ability to embed Google Maps on web pages using JavaScript.

In *Figure 2.3*, we see a simplified diagram of a RESTful web service communicating with a web page using the GET, POST, PUT, and DELETE HTTP methods. Each of these HTTP methods corresponds to a specific type of interaction with the server:

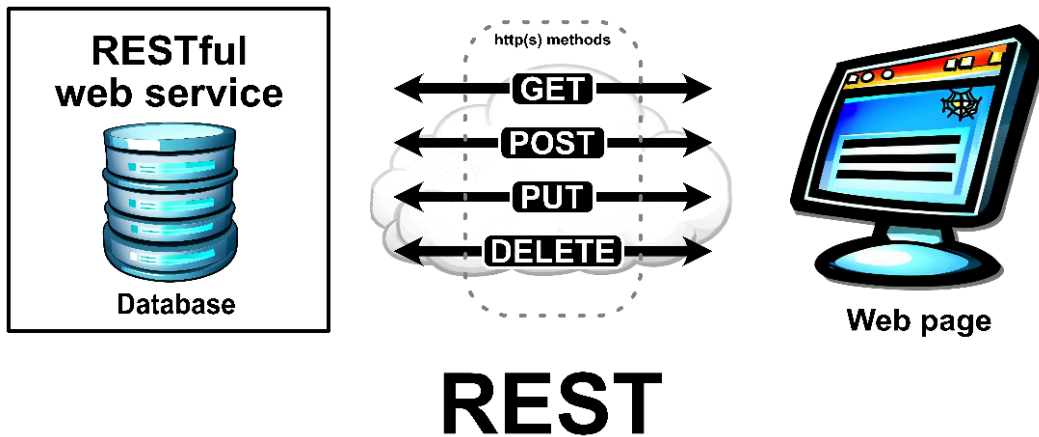


Figure 2.3 – Simplified diagram of a RESTful web service

The GET method retrieves existing data, POST is typically used to send new data for the creation of a resource, PUT is used to update existing resources, and DELETE removes data.

Difference between an API and a web service

We may find ourselves using the terms *API* and *web service* interchangeably; however, there is a difference. An **API** is a broad term defining rules and conventions for building and interacting with software applications. APIs can operate over various channels and not just the web. A **web service**, however, is a specific type of API that operates over the internet, typically using protocols such as HTTP. In essence, all web services are APIs, but not all APIs are web services.

Considering the Twitter API as an example of a web service that we could develop a client for, the application of these HTTP methods would be as follows:

- **GET:** We would use this method to retrieve data from Twitter. For example, we would use a GET request to fetch a specific user's tweets or search for tweets that contain a specific hashtag.
- **POST:** We would use this method when we want to create new data on Twitter, such as a new tweet.
- **PUT:** We wouldn't use this method as the Twitter API doesn't natively support the PUT HTTP method.
- **DELETE:** We would use the DELETE method to remove existing data on Twitter. However, this method is not widely used in the Twitter API as deletion capabilities are limited due to Twitter's policies.

We can see REST methods outlined in the following table:

Method	Description	Use Case	CRUD (Create, Read, Update, Delete) Operation
GET	Retrieves data from a server at the specified resource.	Fetching information without altering the server's state.	Read
POST	Sends data to the server to create a new resource.	Creating new data or performing actions that result in a change on the server.	Create
PUT	Replaces all current representations of the target resource with the request payload.	Updating existing resources entirely.	Update/Replace
DELETE	Removes the specified resource.	Deleting resources from the server.	Delete
PATCH	Applies partial modifications to a resource.	Making partial updates to an existing resource.	Update/Modify

Figure 2.4 – REST methods summarized

To summarize, REST is a straightforward approach that leverages standard HTTP or HTTPS methods such as PUT, POST, GET, and DELETE while also offering flexibility in data formats such as **JavaScript Object Notation (JSON)** or XML, whereas SOAP is a protocol that typically uses XML to transmit structured messages and can operate over various **Internet Protocol (IP)** suite networks such as HTTP or HTTPS.

Now that we have a basic understanding of web services and the way we can implement them, let's create a real-world example using our Raspberry Pi and Sense HAT.

Connecting to a web service with our Raspberry Pi and Sense HAT

In this section, we will connect our Raspberry Pi to a web service and display the result on the dot-matrix screen of our Sense HAT (or emulator). The service we will connect to will be a dummy web service, designed to evaluate RESTful web service calls.

In *Figure 2.5*, we see an example of a Raspberry Pi pulling weather information from a web service and using the dot-matrix display of the Sense HAT to display a cloud:

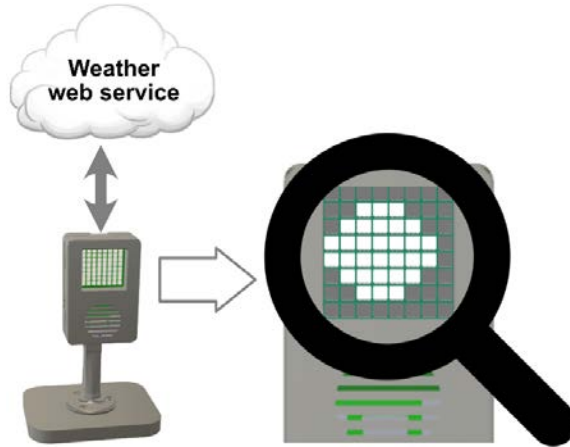


Figure 2.5 – Sense HAT displaying a cloud indicating current weather conditions

The cloud represents current weather conditions (not to be confused with the cloud representing the web service). Such an application could show an animation on the dot-matrix screen. We could even swap the weather web service for another web service and create an entirely new application, leveraging the existing code easily.

Before creating a web service client, we need to set up our development environment and install the necessary packages for our code to work. We will incorporate a Python virtual environment to do so.

Setting up our development environment

We will use a Python virtual environment for our development. As there are libraries that only work with the root installation of Python, we will use system packages in our Python virtual environment. To do so, we do the following:

1. On our Raspberry Pi 5, we open a Terminal application.
2. To store our project files, we create a new directory with the following command:

```
mkdir Chapter2
```

3. We then navigate to the new directory with the following command:

```
cd Chapter2
```

4. We create a new Python virtual environment for our project with the following command:

```
python -m venv ch2-env --system-site-packages
```

5. With this command, we create a new Python virtual environment called `ch2-env` and enable access to the system site packages. This allows the virtual environment to inherit packages from the global Python environment, which can be useful when certain libraries are installed system wide. Once the environment is set up, we can activate it and begin installing project-specific packages without affecting the global Python environment.
6. With our new Python virtual environment created, we source into it (set the Python virtual environment) with the following command:

```
source ch2-env/bin/activate
```

7. Our Terminal application should now show that we are using the `ch2-env` Python virtual environment:



Figure 2.6 – Terminal using `ch2-env` environment

8. We install the extra packages required for our code with the following command:

```
pip install requests sense-emu
```

9. The `requests` library in Python simplifies making HTTP requests to web servers, and the `sense-emu` library will give us the Sense HAT emulator to work with for our code. With the libraries installed, we may close the Terminal with the following command:

```
exit
```

10. We are now ready to load up Thonny. We do so by clicking on the **Menu** icon in the Raspberry Pi taskbar, navigating to the **Programming** category, and selecting **Thonny**.
11. By default, Thonny uses the Raspberry Pi's built-in version of Python. For our project, we will use the Python virtual environment we just created. To start, we need to view the project files by clicking on **View** and selecting **Files** if it is not already selected.
12. In the **Files** section, we locate the `ch2-env` directory.

13. We then right-click on the folder and select the **Activate virtual environment** option:

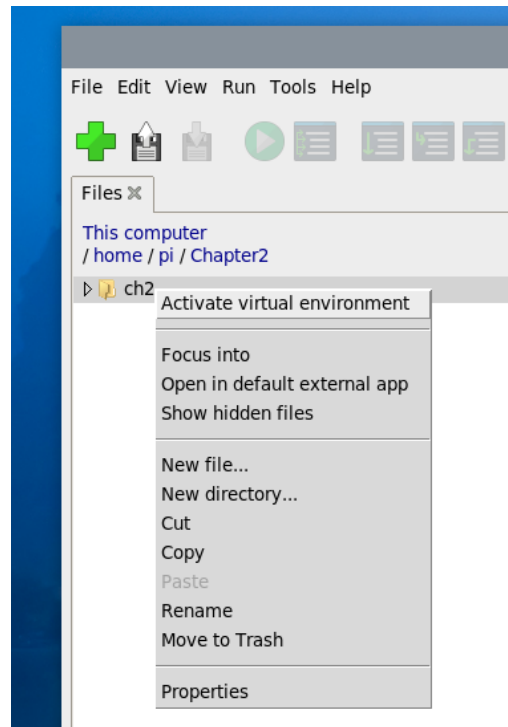


Figure 2.7 – Activating a Python virtual environment in Thonny

With our project folder created, our Python virtual environment set up and activated, and the `requests` package installed, we may now start writing code.

Writing our first web service code

We are now ready to write our first web service code. This code will make a call to a dummy web service used for testing. Upon successful transmission, a success message will scroll across the dot-matrix screen of our Sense HAT:

1. To create our web service application, inside Thonny we create a new tab. Inside the tab, we write the following code:

```
import requests
from sense_hat import SenseHat
response = requests.get(
    'https://jsonplaceholder.typicode.com/posts'
)
```

```
sense = SenseHat()
sense.set_rotation(270)

if response.status_code == 200:
    data = response.json()
    print(data[0]['title'])

    success_msg = 'Success with code: '
    success_msg += str(response.status_code)
    sense.show_message(success_msg)
else:
    error_msg = 'Failed with code: '
    error_msg += str(response.status_code)
    print(error_msg)
    sense.show_message(error_msg)
```

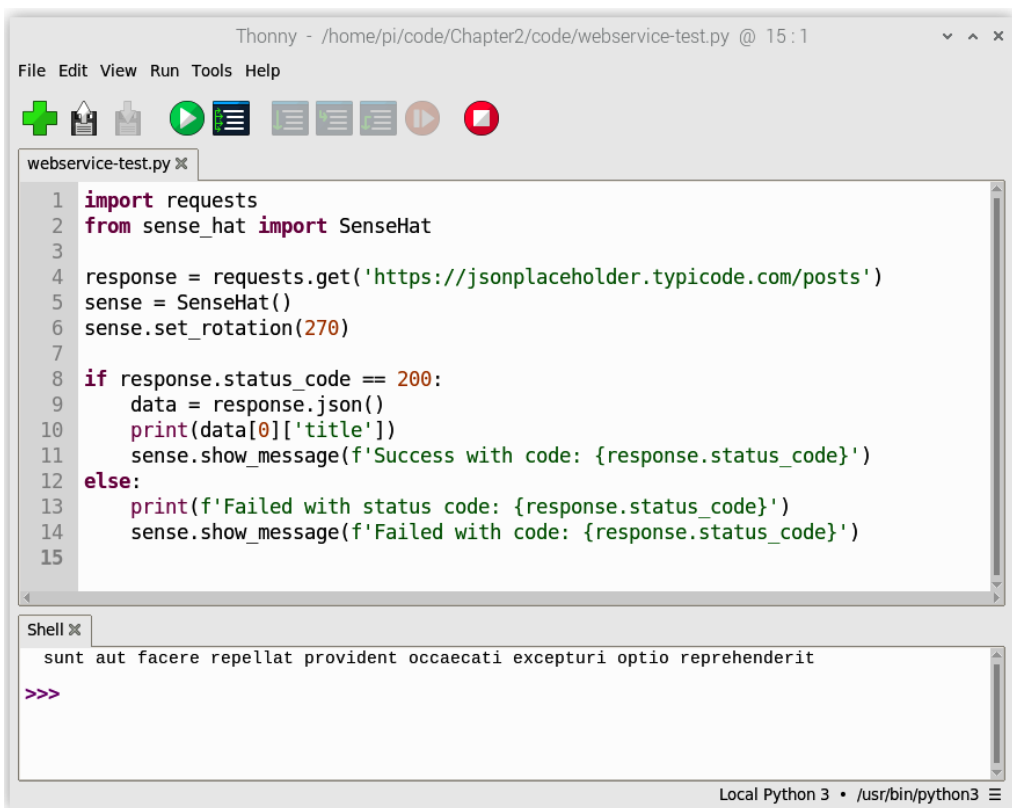
2. Before we run our code, let's break it down:

- I. We start by importing the `requests` library, a popular library in Python for making HTTP requests.
- II. We then import the `SenseHat` class from the `sense_hat` library, allowing us to interact with the Sense HAT board. For those of us using the Sense HAT emulator, we would use the `sense_emu` library instead.
- III. We send a GET request to the specified URL (`'https://jsonplaceholder.typicode.com/posts'`), which is an endpoint of a dummy API that provides placeholder data in the JSON format. The response from the server is stored in the `response` variable.
- IV. We create an instance of the `SenseHat` class. We use this object to control the Sense HAT's dot-matrix display.
- V. The `sense.set_rotation(270)` line adjusts the orientation of the Sense HAT's LED matrix so that it matches the orientation of the Raspberry Pi in our custom case (refer to *Chapter 1* for information on the custom Raspberry Pi case).

Our code then checks the HTTP response's status code:

- If the status code is 200, which indicates a successful HTTP request, the following then occurs:
 - `data = response.json()`: Our code converts the JSON response body into a Python data structure.
 - `print(data[0]['title'])`: Our code prints the title of the first post from the response data to the shell in Thonny.

- Our code displays a success message on the Sense HAT's LED matrix, indicating the successful status code.
 - If the status code is not 200, which indicates an unsuccessful HTTP request, the following happens:
 - Our code prints an error message to the Shell, indicating the unsuccessful status code.
 - Our code displays a failure message on the Sense HAT's LED matrix, indicating the unsuccessful status code.
3. We save the code as `webservice-test.py` and then run it by clicking on the green run button at the top, hitting *F5* on the keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.
 4. Upon a successful web service call (which is to be expected with this dummy service), we should see a title print to the Shell looking like the following:



The screenshot shows the Thonny IDE interface. The top bar indicates the file path: `Thonny - /home/pi/code/Chapter2/code/webservice-test.py @ 15:1`. Below the menu bar (File, Edit, View, Run, Tools, Help) is a toolbar with icons for opening files, saving, running, and other functions. The main editor window displays the following Python code:

```
1 import requests
2 from sense_hat import SenseHat
3
4 response = requests.get('https://jsonplaceholder.typicode.com/posts')
5 sense = SenseHat()
6 sense.set_rotation(270)
7
8 if response.status_code == 200:
9     data = response.json()
10    print(data[0]['title'])
11    sense.show_message(f'Success with code: {response.status_code}')
12 else:
13    print(f'Failed with status code: {response.status_code}')
14    sense.show_message(f'Failed with code: {response.status_code}')
15
```

Below the editor is a Shell window. It contains the output of the script execution:

```
sunt aut facere repellat provident occaecati excepturi optio reprehenderit
>>>
```

The status bar at the bottom indicates the Python version and path: `Local Python 3 • /usr/bin/python3`.

Figure 2.8 – Testing a web service call using Thonny

5. We shouldn't be too concerned about the contents of the title as it is just dummy data. Subsequently, upon successful completion of the web service call, we should observe a scrolling message on the Sense HAT display (or the emulator). This message signifies the successful status of our call, which should be denoted by the 200 HTTP status code.

Although our code lacks error checking, we have successfully constructed our first IoT device powered by the Raspberry Pi. It's important to note that considerations such as internet connectivity are not incorporated into our simple code.

In the remaining sections of this chapter, we will elevate our IoT device from being a simple web service testing tool to something more engaging. We will take on two exciting projects: the construction of a stock ticker application and a weather-dependent GO-NO-GO decision-making application.

Creating a scrolling stock ticker application

It's now time to build our first practical IoT device. For this project, we will create a stock ticker application with our Raspberry Pi and Sense HAT. A stock ticker is a device, physical or digital, that presents stock prices in real time. Our application will fetch real-time stock prices from Alpha Vantage, an online service providing free APIs for data on stocks. In our application, we will be retrieving the current stock price for Apple, listed on the Nasdaq stock exchange as *AAPL*.

We can see our stock ticker application illustrated in *Figure 2.9*. We will use the HTTP `GET` method to retrieve information with the response in JSON format, a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate:



Figure 2.9 – Diagram of our stock ticker application; the double arrows symbolize the call and subsequent response from the Alpha Vantage web service

In our case, when we request data for the AAPL stock symbol, the **Alpha Vantage API** sends back a JSON object response, as illustrated in *Figure 2.10*:

```
{
  "Global Quote": {
    "01. symbol": "AAPL",
    "02. open": "193.7800",
    "03. high": "193.8800",
    "04. low": "191.7600",
    "05. price": "192.4600",
    "06. volume": "31393616",
    "07. latest trading day": "2023-07-03",
    "08. previous close": "193.9700",
    "09. change": "-1.5100",
    "10. change percent": "-0.7785%"
  }
}
```

Figure 2.10 – JSON object response from Alpha Vantage web service call

The response encompasses 10 parameters for every successful API request. Out of these parameters, our stock ticker application will focus on `symbol`, `volume`, `price`, and `change`. These specific data points will be used to create a message that we will scroll across the dot-matrix screen of the Sense HAT.

However, before we can write our web service code, we must first acquire an API key from Alpha Vantage. This key grants us permission to make the necessary web service calls.

Getting an API key

Securing an API key from Alpha Vantage is a straightforward process that can be accomplished in just a few steps. We start by navigating to the Alpha Vantage website at <https://www.alphavantage.co>.

From there, we click on the **GET FREE API KEY** button – this should be easy to locate on the home page. Clicking this button will lead us to a sign-up form. We fill out the required details on this form, making sure to provide a valid email address. We should be given an API key once the form has been filled out, and we click on the **GET FREE API KEY** button.

Important note

The preceding instructions are valid as of the time of this writing. Please follow any changes to the process to get the API key.

Once the API key has been issued, we must copy and paste it into a text editor as we require this key every time we make a call to the Alpha Vantage web service. As a free user, we are limited to 5 API requests per minute and a total of 500 API requests per day.

Armed with our API key, we can now start to write code for our application.

Writing web services client code

In this section, we will start developing web service code to fetch current stock information for the company Apple (AAPL). Our objective is to retrieve the JSON object response from the Alpha Vantage web service, which will contain the relevant stock data we need for our scrolling stock ticker application.

To create our web service code, we do the following:

1. We launch Thonny on our Raspberry Pi and activate the `ch2-env` Python virtual environment using the steps in the previous section.
2. We then open a new tab in Thonny and enter the following code:

```
import requests
import json

api_key = 'xxxxxxxxxxxxxxxxxxxx'
symbol = 'AAPL'

base_url = 'https://www.alphavantage.co/query?'
function = 'GLOBAL_QUOTE'

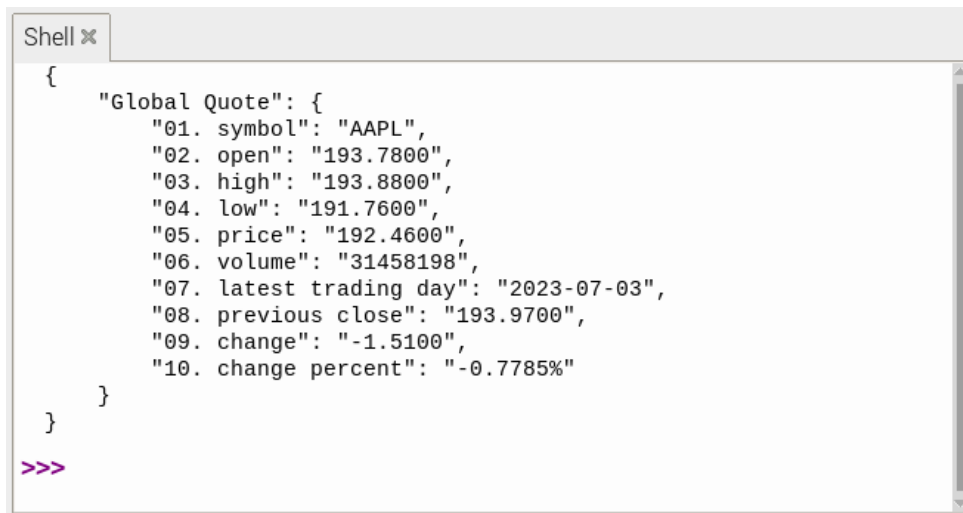
complete_url = f'{base_url}
function={function}&symbol={symbol}&apikey={api_key}'

response = requests.get(complete_url)
data = response.json()

print(json.dumps(data, indent=4))
```

3. Before we run our code, let's break it down:
 - I. We start by importing the `requests` module, which is a widely used Python library for making HTTP requests. It provides convenient methods to send HTTP requests such as GET, POST, and so on and handles underlying network communication.
 - II. We then import the built-in `json` module in Python. The `json` module provides methods to work with JSON data. It allows encoding Python objects into JSON strings (`json.dumps()`) and decoding JSON strings into Python objects (`json.loads()`).
 - III. We store our personal API key from Alpha Vantage in a variable called `api_key`.
 - IV. We set the `symbol` variable to `'AAPL'`, representing the stock symbol for Apple.

- V. The `base_url` variable stores the base URL for the Alpha Vantage API.
 - VI. We set the `function` variable to `'GLOBAL_QUOTE'`, indicating the specific function to retrieve global stock quotes.
 - VII. We construct the `complete_url` variable by combining the base URL, function, symbol, and API key to form a complete URL for the API request.
 - VIII. Our code then sends a GET request to the Alpha Vantage API using `requests.get()` and stores the response in the `response` variable.
 - IX. We extract the JSON response from the `response` object using `.json()`, and the resulting data is stored in the `data` variable.
 - X. Finally, the code prints data in a formatted JSON representation using `json.dumps()` with the `indent` parameter set to 4.
- 4. We save the code as `alphavantage-test.py` and then run it by clicking on the green run button, hitting `F5` on the keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.
 - 5. We should see a JSON object like what we see in *Figure 2.11* displayed in the console:



```
Shell x
{
  "Global Quote": {
    "01. symbol": "AAPL",
    "02. open": "193.7800",
    "03. high": "193.8800",
    "04. low": "191.7600",
    "05. price": "192.4600",
    "06. volume": "31458198",
    "07. latest trading day": "2023-07-03",
    "08. previous close": "193.9700",
    "09. change": "-1.5100",
    "10. change percent": "-0.7785%"
  }
}
>>>
```

Figure 2.11 – JSON response displayed in Thonny's Shell

Important note – code not for production

Please note that in the provided code, error checking has been omitted for the sake of simplicity. If this application were to be deployed to a production environment (for use by customers, for example), we would be sure to include appropriate error-handling and error-checking mechanisms to ensure the reliability and stability of the application.

With our code to pull stock information from the internet done, it is now time to utilize the dot-matrix screen of the Sense HAT to create a scrolling stock ticker.

Enhancing our application

With our understanding of how to use the Alpha Vantage web service, we are now able to create an application that fetches stock data and transforms it into a real-life scrolling stock ticker. Our application takes advantage of the Sense HAT's dot-matrix display, turning it into a dynamic canvas for the stock ticker. Instead of printing the JSON response to the console, the stock information will elegantly scroll across the Sense HAT display, providing a visually captivating representation of the data.

To create our web service code, we launch Thonny on our Raspberry Pi and create a new tab:

1. We launch Thonny on our Raspberry Pi and activate the `ch2-env` Python virtual environment using the steps in the previous section.
2. In a new tab in Thonny, we start our code by importing the necessary libraries:

```
import requests
from sense_hat import SenseHat
import time
```

In our code, we import the `requests` module, as well as the `SenseHat` class from the `sense_hat` module. For those of us using the Sense HAT emulator, we would change this to `from sense_emu import SenseHat`. We then import the `time` module.

3. With our libraries in place, we create and set the variables we use in our code:

```
api_key = 'xxxxxxxxxxxxxxxxxxxx'
symbol = 'AAPL'
base_url = 'https://www.alphavantage.co/query?'
function = 'GLOBAL_QUOTE'
sense = SenseHat()
sense.set_rotation(270)
last_call_time = time.time() - 180
last_ticker_info = ""
```

In these code lines, we use the `api_key` variable to store our unique Alpha Vantage API key for accessing a web service. We use the `symbol` variable to store the stock symbol (for example, 'AAPL') for which data will be fetched. The `base_url` variable is used to store the base URL of the web service API. The `function` variable is used to define the specific function to be called from the web service API (for example, 'GLOBAL_QUOTE'). We then create an instance of the `SenseHat` class and assign it to the `sense` variable to interact with the Sense HAT (or emulator). We set the rotation of the Sense HAT display to 270 degrees using `sense.set_rotation(270)`. This is so that it matches the orientation of the Raspberry Pi in our custom case. We could comment this line out for the emulator. We then initialize

the `last_call_time` variable with the current time minus 180 seconds, which allows an immediate first call to the web service. We initialize the `last_ticker_info` variable as an empty string to store the previous ticker information.

4. Below our variable declarations, we implement an infinite loop to continuously display the ticker information; however, to comply with API rate limits of 5 requests per minute and 500 requests per day, we introduce a time delay of 3 minutes between each web service call. We type the following code below our variable declarations:

```
while True:
    current_time = time.time()

    if current_time - last_call_time >= 180:
        complete_url = f'{base_url}function={
            function}&symbol={
            symbol}&apikey={api_key}'

        response = requests.get(complete_url)
        data = response.json()

        quote = data['Global Quote']
        ticker_info = (
            f"{quote['01. symbol']} "
            f"Price: {quote['05. price']} "
        )
        ticker_info += (
            f"Volume: {quote['06. volume']} "
            f"Change: {quote['09. change']} "
        )
        last_ticker_info = ticker_info

        sense.show_message(ticker_info,
                           scroll_speed=0.05,
                           text_colour=[255,
                                         255,
                                         255])

        last_call_time = current_time

    else:
        sense.show_message(last_ticker_info,
                           scroll_speed=0.05,
                           text_colour=[255,
                                         255,
```

```

                                255])
time.sleep(1)

```

Our code is wrapped in a `while True` loop, which ensures continuous execution of the following code block:

- I. We set the `current_time` variable to the current time using `time.time()`.
 - II. Our code then checks whether the difference between `current_time` and `last_call_time` is greater than or equal to 180 seconds.
 - III. If `True`, the following happens:
 - i. A `complete_url` variable is created using an f-string to form the URL for the API call.
 - ii. An HTTP GET request is sent to the API using `requests.get(complete_url)`, and the response is stored in the `response` variable.
 - iii. The response is parsed as JSON using `response.json()` and assigned to the `data` variable.
 - iv. The relevant stock information is extracted from the `data` dictionary and formatted into a `ticker_info` string.
 - v. The `last_ticker_info` variable is updated to store the current `ticker_info` value.
 - vi. The `ticker_info` string is displayed on the Sense HAT's dot-matrix display using `sense.show_message()`, with a scrolling speed of 0.05 seconds and white text color (255, 255, 255).
 - vii. The `last_call_time` variable is updated to the current time (`current_time`) to mark the timestamp of the last API call.
 - IV. If `False`, the previous `last_ticker_info` variable is displayed on the Sense HAT display with a scrolling speed of 0.05 seconds and white text color (255, 255, 255).
 - V. Our program then sleeps for 1 second using `time.sleep(1)` before the next iteration of the loop. This is done to regulate resource consumption and control the update frequency of the Sense HAT's dot-matrix display.
5. We save our code as `aapl-stock-ticker.py` and then run it by clicking on the green run button, hitting `F5` on the keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.
 6. After we execute the code, we should observe a stock message scrolling across the dot-matrix screen of the Sense HAT. If we are utilizing the emulator, the message will scroll across the simulated dot-matrix display, considering the 270-degree orientation set for the emulator. *Figure 2.12* provides a visual representation of how this appears when using the emulator:

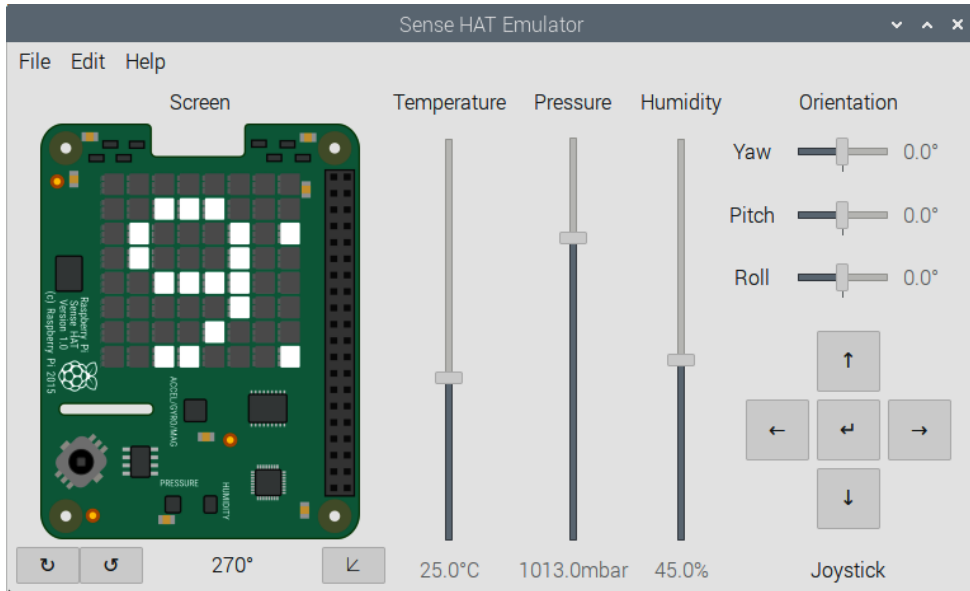


Figure 2.12 – Stock ticker information on Sense HAT emulator

Congratulations are in order as we have successfully built our first real-world IoT device, a stock ticker using Raspberry Pi and Sense HAT! This device opens a world of possibilities beyond just displaying stock information. In the next section, we will start developing applications to display weather conditions.

Developing weather display applications

Now that we are experienced IoT application developers, we are ready to take our skills to the next level and create more intricate projects. In this section, we will leverage the capabilities of Raspberry Pi and Sense HAT to create a weather display application and a weather-dependent GO-NO-GO decision-making application.

In *Figure 2.13*, we see a diagram depicting a call to the OpenWeather API from our Raspberry Pi and Sense HAT, enclosed within its custom case. For our weather display application, we will follow a similar approach to the scrolling stock ticker:

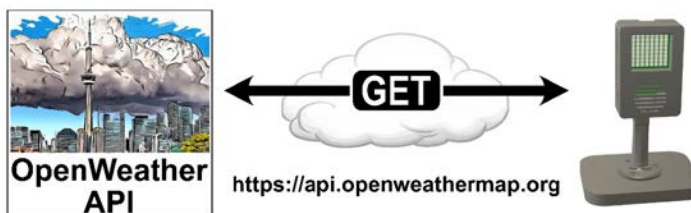


Figure 2.13 – Using the OpenWeather API to get the current weather conditions

We will first acquire an API key from OpenWeather and verify the API call by printing the response to the Shell for testing purposes. We will then utilize the Sense HAT to create a ticker-style display that displays the current weather conditions.

Finally, we will replace the scrolling display with visuals as we build a weather-dependent GO-NO-GO decision-making application.

We will start by obtaining an API key.

Getting an API key

To utilize the OpenWeather web service, it is necessary to obtain an API key. This API key serves as a unique identifier that grants access to the OpenWeather web service. The key is acquired by creating an account on the OpenWeather website and generating an API key by subscribing to the appropriate service. The API key acts as a credential to authenticate and authorize our requests to the OpenWeather web service, enabling us to retrieve weather data for various locations around the world. It's important to keep the API key confidential and securely store it as it grants access to the OpenWeather API on our behalf.

To obtain a free API key from OpenWeather, we start by navigating to the OpenWeather price page located at <https://openweathermap.org/price>. We then scroll down to the **Current weather and forecasts collection** section and click on the **Get API key** button:

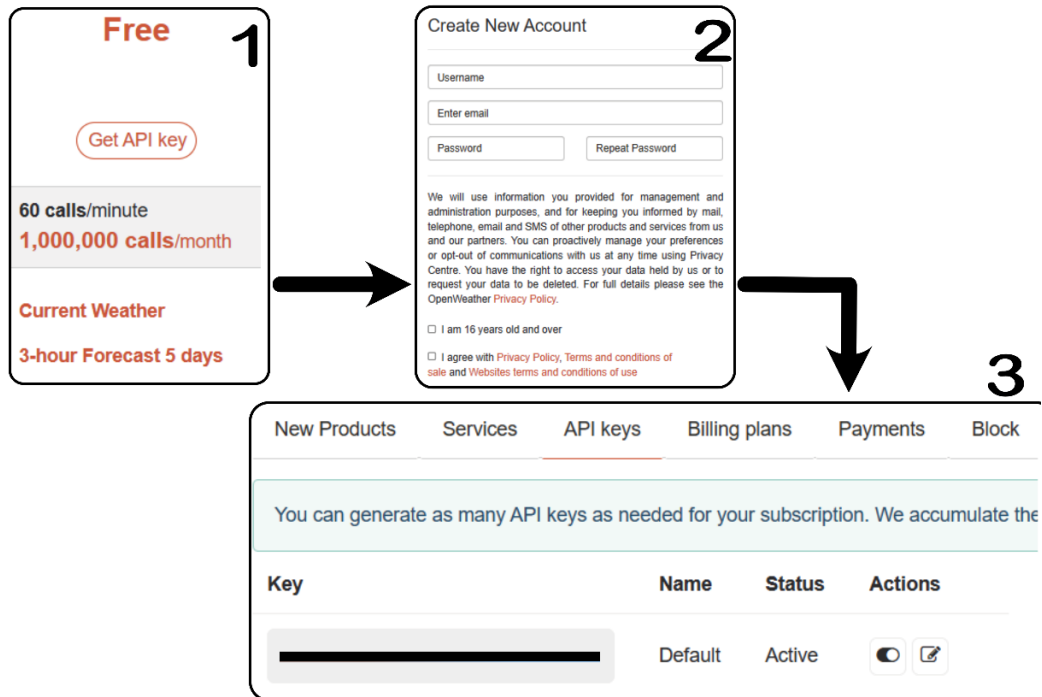


Figure 2.14 – Obtaining an API key from OpenWeather

We copy and paste our key into a text editor as we require this key every time we make a call to the OpenWeather web service. As a free user, we are limited to 60 API requests per minute and a total of 1,000,000 API requests per month. This should be more than enough for our application.

Creating a scrolling weather information ticker

After we are satisfied that our API key and web service work, we will integrate the web service data with the Sense HAT, displaying scrolling text that displays the temperature and weather conditions.

Before integrating the OpenWeather API into our Raspberry Pi and Sense HAT, we will ensure its functionality with a simple program. To create the test code, we do the following:

- ```
import requests

url = "https://api.openweathermap.org/data/2.5/weather"
api_key = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
location = "Toronto"
params = {
 "q": location,
 "appid": api_key,
 "units": "metric"
}
response = requests.get(url, params=params)

if response.status_code == 200:
 data = response.json()
 temperature = data["main"]["temp"]
 description = data["weather"][0]["description"]
 print(f"The current temperature in {location} is {temperature}°C.")
```

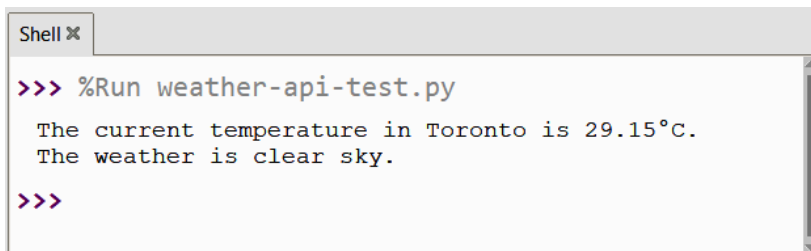


```
print(f"The weather is {description}.")
else:
 print("Error: Failed to retrieve weather information.")
```

Before we run our code, let's break it down:

- I. We start by importing the `requests` module to make HTTP requests.
  - II. We then set the `url` variable to the OpenWeather API endpoint.
  - III. We set the `api_key` variable with our OpenWeather API key.
  - IV. We set the `location` variable to the desired location for which we want to retrieve weather information. For our example, this is "Toronto".
  - V. We then create a dictionary called `params` with the parameters for the API request, including the location, API key, and desired units.
  - VI. A GET request is sent to the OpenWeather API using `requests.get()`, with `url` and `params` as arguments.
  - VII. We then check whether the response status code is 200 (indicating a successful request).
  - VIII. If the response is successful, we parse the JSON data from the response using `response.json()` and then do the following:
    - i. We extract the temperature and weather description from the parsed data.
    - ii. We then print the current temperature and weather information for the specified location.
  - IX. If there is an error (response status code other than 200), we print an error message indicating the failure to retrieve weather information.
2. We save our code as `weather-api-test.py` and then run it by clicking on the green run button, hitting *F5* on the keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.

After we execute the code, we should observe a message in the Shell:



```
Shell x
>>> %Run weather-api-test.py
The current temperature in Toronto is 29.15°C.
The weather is clear sky.
>>>
```

Figure 2.15 – OpenWeather API information on the weather in Toronto

As we can see, it is 29.15 °C and clear in Toronto at the time of this writing. If the web service call did not work, we would've seen an `Error: Failed to retrieve weather information` error in the console.

With our understanding of how to use the OpenWeather API, we are now ready to use the Sense HAT to create our scrolling weather information ticker. For this, we may reuse much of the code we wrote for our scrolling stock ticker application.

## Scrolling weather information on Sense HAT

As we highlighted in our previous project, the versatility of our scrolling stock ticker application allows us to adapt it to display various types of information beyond stocks. In this section, we will leverage this adaptability by integrating the OpenWeather API and our API key to transform our ticker into a dynamic weather display, scrolling real-time weather data such as temperature and current conditions. We will be able to reuse a lot of the code from the scrolling stock ticker. To create the scrolling ticker code, we do the following:

1. We launch Thonny on our Raspberry Pi, activate the `ch2-env` Python virtual environment, and create a new tab. We will start with our imports:

```
import requests
from sense_hat import SenseHat
import time
```

### Important note

As we have already covered these imports with our scrolling stock ticker application, we do not need to cover them again.

2. After our imports, we set our variables:

```
api_key = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
location = 'Toronto'
base_url = 'https://api.openweathermap.org/data/2.5/weather'
params = {
 'q': location,
 'appid': api_key,
 'units': 'metric'
}
sense = SenseHat()
sense.set_rotation(270)
last_call_time = time.time() - 30
last_weather_info = ""
```

In this code block, we do the following:

- I. We start by assigning the `api_key` variable to our OpenWeather API key.
  - II. We set the `location` variable to the desired location for which we want to retrieve weather information. For our example, this is 'Toronto'.
  - III. We then set the `base_url` variable to the OpenWeather API endpoint.
  - IV. We create a dictionary called `params` with the parameters for the API request, including the location, API key, and desired units.
  - V. A GET request is sent to the OpenWeather API using `requests.get()`, with `url` and `params` as arguments.
  - VI. We then create an instance of the `SenseHat` class and assign it to the `sense` variable to interact with the Sense HAT (or emulator).
  - VII. We set the rotation of the Sense HAT display to 270 degrees using `sense.set_rotation(270)`. This is so that it matches the orientation of the Raspberry Pi in our custom case. We could comment this line out for the emulator.
  - VIII. We set `last_call_time` to the current time minus 30 seconds.
  - IX. We then add `last_weather_info`, which is a variable that stores the previous weather information.
3. Below our variable declarations, we implement an infinite loop to continuously display the weather ticker information; however, to comply with API rate limits of 60 requests per minute and 1,000,000 requests per month, we introduce a time delay of 30 seconds between each web service call. We type the following code below our variable declarations:

```
while True:
 current_time = time.time()

 if current_time - last_call_time >= 30:
 response = requests.get(base_url,
 params=params)
 data = response.json()

 temperature = data['main']['temp']
 description = data['weather'][0]['description']
 weather_info = f"{location}: {temperature}°C,
{description}"

 last_weather_info = weather_info
```

```
sense.show_message(weather_info, scroll_speed=0.05,
text_colour=[255, 255, 255])

last_call_time = current_time

else:
 sense.show_message(last_weather_info, scroll_speed=0.05,
text_colour=[255, 255, 255])

time.sleep(1)
```

4. As in our scrolling stock ticker application, the heart of our code is wrapped in a while True loop, which ensures continuous execution of the main code:
  - I. We set the `current_time` variable to the current time using `time.time()`.
  - II. Our code then checks whether the difference between `current_time` and `last_call_time` is greater than or equal to 30 seconds.
  - III. If True, the following happens:
    - i. A GET request is sent to the OpenWeather API using `requests.get()`, with `base_url` and `params` as arguments.
    - ii. The response is parsed as JSON using `response.json()` and assigned to the `data` variable.
    - iii. We extract the temperature and weather description from the parsed data and store it as `weather_info`.
    - iv. The `last_weather_info` variable is updated to store the current `weather_info` value.
    - v. `weather_info` is displayed on the Sense HAT's dot-matrix display using `sense.show_message()`, with a scrolling speed of 0.05 seconds and white text color (255, 255, 255).
    - vi. The `last_call_time` variable is updated to the current time (`current_time`) to mark the timestamp of the last API call.
  - IV. If False, the previous `last_weather_info` variable is displayed on the Sense HAT display with a scrolling speed of 0.05 seconds and white text color (255, 255, 255).
  - V. Our program then sleeps for 1 second using `time.sleep(1)` before the next iteration of the loop. This is done to regulate resource consumption and control the update frequency of the Sense HAT's dot-matrix display.

5. We save our code as `weather-scroll.py` and then run it by clicking on the green run button, hitting `F5` on the keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.

After we execute the code, we should observe weather information scrolling across the dot-matrix screen of the Sense HAT. If we are utilizing the emulator, a message will scroll across the simulated dot-matrix display. *Figure 2.16* provides a visual representation of how this appears when using the emulator:

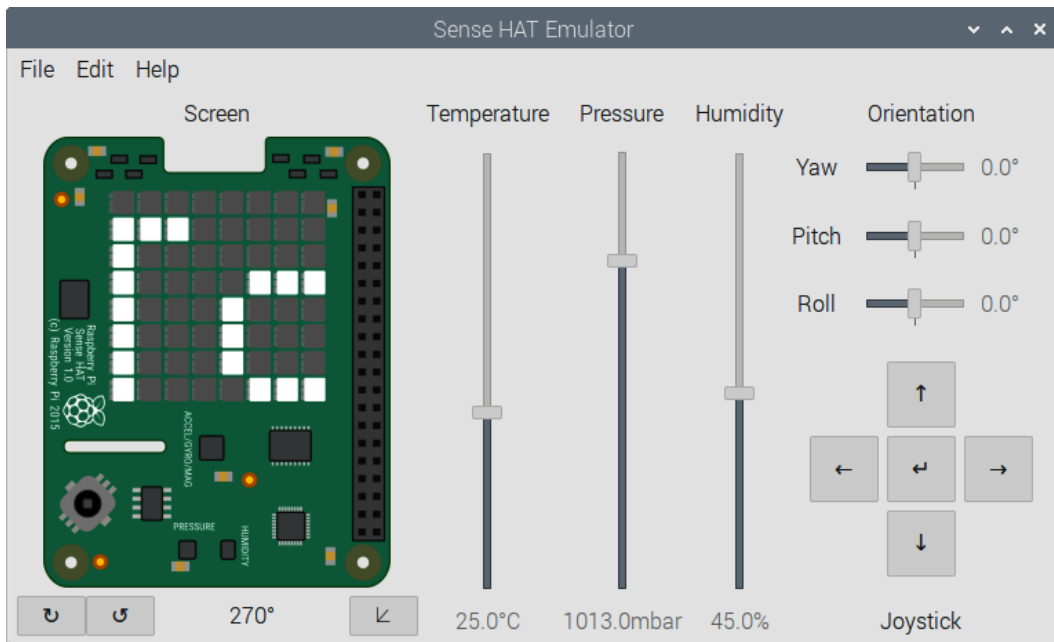


Figure 2.16 – Weather information scrolling across the simulated dot-matrix display

One key takeaway is the power of leveraging existing code to create new applications. Despite inherent differences between stock information and weather data, the process of obtaining information for both fields remain remarkably similar. With this realization, we unlock the potential to create a wide range of dynamic and engaging displays using the same underlying code structure.

Here are a few examples of other applications we could build:

- **News updates:** By modifying the code, we can integrate our device with news APIs to display real-time headlines or updates from popular news sources.
- **Social media notifications:** By connecting our application to social media APIs, we can configure it to display notifications from popular platforms such as Twitter or Facebook.

- **Sports scores:** With the integration of sports data APIs, our stock ticker application can be transformed into a real-time sports scoreboard. It can display live scores, game updates, or upcoming game schedules.
- **Personalized reminders:** By extending the functionality of the code, we can program the stock ticker application to display personalized reminders or to-do lists.

In our next and final project for the chapter, we will replace our scrolling text displays with dot-matrix images and animations. This shift from scrolling text elevates the user experience and will make our projects more visually appealing.

## Developing a GO-NO-GO application for decision-making

Consider the role of a youth baseball league convener, responsible for ensuring the safety of playing fields. Critical to this responsibility is making weather-based decisions. If the field is excessively wet, it can impact gameplay, potentially leading to game postponements or cancellations:

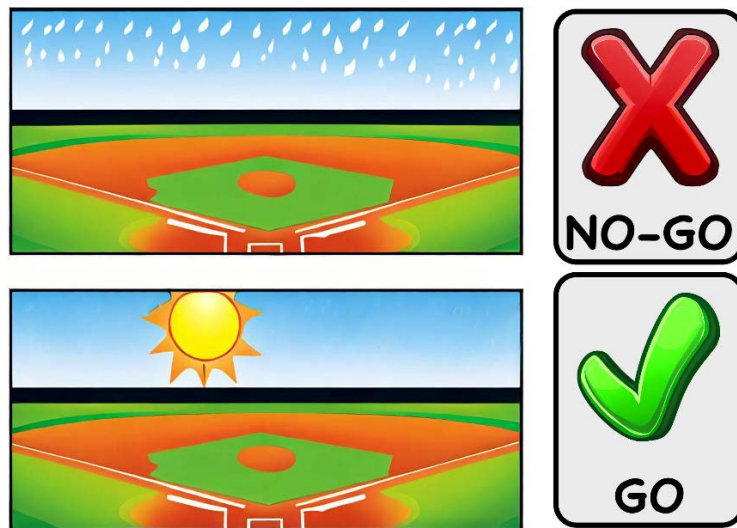


Figure 2.17 – Should the game go on?

Another factor to consider is the age of the players. For younger players, playing in the rain raises concerns, as parents are often present and may express dissatisfaction with unfavorable weather conditions. On the other hand, older players, who typically travel to games independently, may be less affected by wet conditions.

These decision-making scenarios represent an opportunity to develop an IoT application that displays a visual indicator, such as a **GO** or **NO-GO** graphic, based on weather conditions and player age (Figure 2.17). Imagine a setup with Raspberry Pi and Sense HATs for each baseball diamond, where

the Sense HAT display provides real-time guidance on whether the game should proceed as scheduled, be postponed, or be canceled altogether. This IoT application enables efficient decision-making and enhances the overall experience and safety of the youth baseball league.

In our simplified example, we will focus on incorporating basic decision-making into our IoT application. Based on the age of the players and the presence of rain, the Sense HAT will show either a green checkmark or an animated red X sign. While we could introduce additional complexity, the primary objective of this exercise is to demonstrate how decision-making can be integrated into an IoT application. By incorporating these visual indicators, we empower real-time decision-making. Instead of relying solely on the convener, our IoT application takes charge by providing immediate guidance on whether games should proceed or be postponed.

We will start by writing Sense HAT code for indication. For GO, we will show a simple green checkmark against a black background. For NO-GO, we will display a flashing red X. We will run our application using the Sense HAT emulator as it is easier to display screenshots for this book; however, it is strongly encouraged to use the Sense HAT as this makes our application a true IoT device.

We will start by writing code to display a green checkmark.

### ***Creating a checkmark on our Sense HAT***

In this section, we will create code that displays a green checkmark against a black background on the Sense HAT emulator. To enhance code implementation and organization, we will encapsulate the functionality within a Python class. This approach simplifies the integration process and promotes code reusability, allowing us to easily incorporate the green checkmark display into our IoT application project.

Prior to writing the code for the GO-NO-GO application, we will create a project directory named GO-NO-GO on our Raspberry Pi. This dedicated folder will serve as a centralized location for organizing and managing files and resources associated with our project. To create the checkmark code, we do the following:

1. We launch Thonny on our Raspberry Pi, activate the `ch2-env` virtual environment, and create a new tab. Inside the tab, we write the following code:

```
from sense_emu import SenseHat
class GreenCheck:
 black = (0, 0, 0)
 green = (0, 255, 0)
 check_mark_pixels = [
 black, black, black, black,
 black, black, black, green,
 black, black, black, black,
 black, black, green, green,
 black, black, black, black,
 black, black, green, green,
 black, black, black, black,
 black, green, green, black,
```

```

 green, black, black, black,
 green, green, black, black,
 black, green, black, black,
 green, green, black, black,
 black, green, green, green,
 green, black, black, black,
 black, black, black, green,
 black, black, black, black
]

 def __init__(self, rotation=0):

 self.sense = SenseHat()
 self.sense.set_rotation(rotation)

 def display(self):
 self.sense.set_pixels(self.check_mark_pixels)

if __name__ == "__main__":
 greenCheck = GreenCheck(rotation = 270)
 greenCheck.display()

```

In our code, we do the following:

- I. We start by importing the `SenseHat` class from the `sense_hat` module (use `sense_emu` for the emulator)
- II. We then define a `GreenCheck` class for displaying a green checkmark on the Sense HAT
- III. We set color values for black and green as RGB tuples.
- IV. We then define a list of pixel values representing the checkmark shape.
- V. The `GreenCheck` class is initialized with an optional rotation parameter, which defaults to 0.
- VI. Inside the `__init__` method, we create a Sense HAT instance and set the rotation to the value of `rotation`.
- VII. We define a `display` method that sets the Sense HAT's pixels to the checkmark pixel values.
- VIII. We use `if __name__ == "__main__":` to check whether the code is being run directly (not imported).
- IX. If True, we do the following:
  - i. We create an instance of the `GreenCheck` class named `greenCheck` with a rotation value of 270.
  - ii. We call the `display()` method to show a green checkmark on the Sense HAT.



2. We save our code as `green_checkmark.py` in the GO-NO-GO folder and then run it by clicking on the green run button, hitting `F5` on the keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.
3. After we execute the code, we should see a green checkmark against a black background on our Sense HAT emulator:

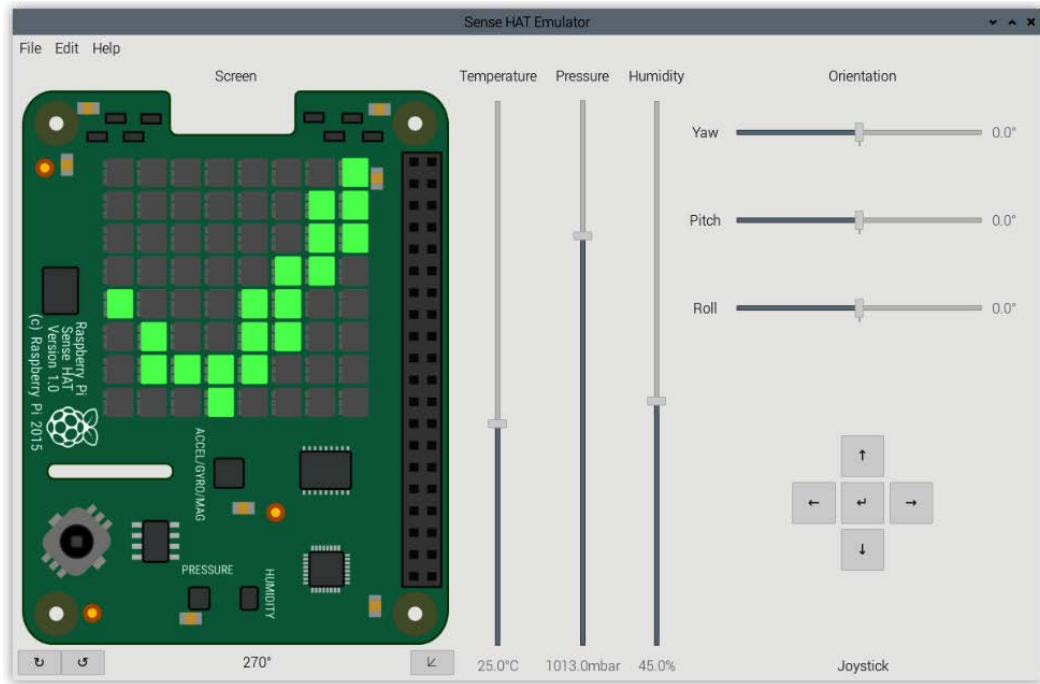


Figure 2.18 – Green checkmark against black background on Sense HAT’s dot-matrix display

With the completion of the green checkmark code, we will now shift our focus toward creating a NO-GO animation (flashing red X) for our application.

### ***Creating a NO-GO animation on our Sense HAT***

The NO-GO animation we have designed consists of a flashing effect on the Sense HAT emulator display, alternating between a red X sign on a black background and a full red display. To create code for the flashing X sign, we do the following:

1. We launch Thonny on our Raspberry Pi, activate the `ch2-env` Python virtual environment, and create a new tab. Inside the tab, we start by importing the packages we need:

```
from sense_emu import SenseHat
import time
```

2. Once we have our packages defined, we then start to wrap our code up in a Python class:

```
class RedXAnimation:
 black = (0, 0, 0)
 red = (255, 0, 0)

 frame1 = [
 red, black, black, black,
 black, black, black, red,
 black, red, black, black,
 black, black, red, black,
 black, black, red, black,
 black, red, black, black,
 black, black, black, red,
 red, black, black, black,
 black, black, black, red,
 red, black, black, black,
 black, black, red, black,
 black, red, black, black,
 black, red, black, black,
 black, black, red, black,
 red, black, black, black,
 black, black, black, red
]

 frame2 = [
 red, red, red, red, red, red, red, red,
 red, red, red, red, red, red, red, red,
 red, red, red, red, red, red, red, red,
 red, red, red, red, red, red, red, red,
 red, red, red, red, red, red, red, red,
 red, red, red, red, red, red, red, red,
 red, red, red, red, red, red, red, red,
 red, red, red, red, red, red, red, red
]
```

In our code, we do the following:

- I. We start by defining a `RedXAnimation` class.
- II. We then set color values for black and red as RGB tuples.
- III. We define `frame1` as a list of pixel values representing a red X sign on a black background.
- IV. We define `frame2` as a list of pixel values representing a full red display.

3. From here, we write code for the initialize method in the class:

```
def __init__(self, rotation=0):
 self.sense = SenseHat()
 self.sense.set_rotation(rotation)
```

In our code, we do the following:

- I. We use the `__init__` method to initialize the `RedXAnimation` object with an optional rotation parameter (defaulted to 0).
  - II. Inside `__init__`, a `SenseHat` instance is created, and the rotation is set based on the provided rotation value.
4. The `display_animation()` method will cycle through the 2 frames for 59 seconds. We do this to align with future client code:

```
def display_animation(self, duration):
 num_frames = 2
 frame_duration = duration / num_frames

 start_time = time.time()
 end_time = start_time + 59

 while time.time() < end_time:
 for frame in [self.frame1, self.frame2]:
 self.sense.set_pixels(frame)
 time.sleep(frame_duration)
```

In our code, the following happens:

- I. Our `display_animation()` method takes a duration parameter.
- II. We set the number of frames to 2.
- III. We calculate the duration for each frame by dividing the total duration by the number of frames.
- IV. We set the `start_time` variable to the current time using `time.time()`.
- V. We calculate the `end_time` value by adding 59 seconds to the `start_time` variable.
- VI. We create a loop that runs until the current time exceeds the `end_time` value:
  - i. Our code iterates over each frame in the list `[self.frame1, self.frame2]`.
  - ii. We set the Sense HAT display pixels to the current frame using `self.sense.set_pixels(frame)`.
  - iii. We then pause the execution for the frame duration using `time.sleep(frame_duration)`.

5. We use the `if __name__ == "__main__":` block to ensure that the test code is executed only when the script is run directly (not imported as a module):

```
if __name__ == "__main__":
 animation = RedXAnimation(rotation=270)
 animation.display_animation(duration=1)
```

In our code, the following happens:

- I. An instance of the `RedXAnimation` class is created with a rotation value of 270 degrees, assigned to the `animation` variable.
  - II. The `display_animation()` method of the `animation` object is called, specifying a duration of 1 second.
6. We save our code as `flashing_x.py` in the GO-NO-GO folder and then run it by clicking on the green run button, hitting `F5` on the keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.

After executing the code, we should observe an animation of a red X sign against a black background turn into a full screen of red and back again. In *Figure 2.19*, we can see what this would look like on the emulator:

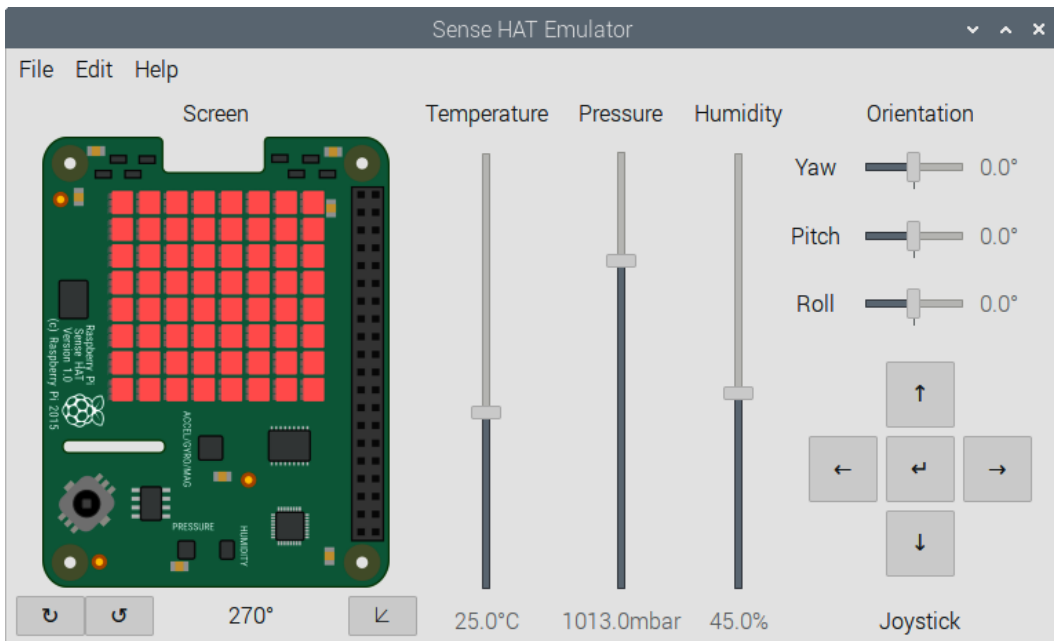


Figure 2.19 – NO-GO animation in red screen mode

The NO-GO animation we have created in this section provides a highly effective visual indicator on the Sense HAT display. By alternating between a red X sign on a black background and a full red display, this animation conveys unfavorable conditions that would necessitate the cancellation of a game.

#### Setting the geolocation for other cities

For finding a city's geolocation information such as latitude and longitude, websites such as *GPS Coordinates* (<https://gps-coordinates.org/>) and *Latitude and Longitude Finder* (<https://www.latlong.net/>) are useful. They allow us to input an address or place and receive its precise coordinates.

To finish off our application, we will now write the web service and logic layer, and we will incorporate our green checkmark and red X sign animation.

### Writing GO-NO-GO client code

Now, it's time to dive into the exciting phase (subjective, of course) of writing code to determine whether a game should be a GO or NO-GO based on weather conditions and the age of the players. Our approach will be straightforward: if it's raining and the players are under 16 years of age, it's a NO-GO; otherwise, it's a GO. While we can certainly implement more complex logic, including **Machine Learning (ML)** if there are multiple parameters to consider, for simplicity, we'll focus on this basic decision-making process. We do so with the following:

1. To create the client code, we launch Thonny on our Raspberry Pi, activate our `ch2-env` Python virtual environment, and create a new tab. Inside the tab, we start by importing the packages we need:

```
import requests
import time
from green_checkmark import GreenCheck
from flashing_x import RedXAnimation
```

We've covered the first three packages already. For the two modules, we do the following:

- We import the `GreenCheck` class from the `green_checkmark` module to display a green checkmark for the **GO** decision.
  - We import the `RedXAnimation` class from the `flashing_x` module to display a flashing red X sign animation when the decision is **NO-GO**.
2. With our packages and modules in place, we now set our variables:

```
latitude = '42.346268'
longitude = '-71.095764'

go = GreenCheck(rotation=270)
```

```

no_go = RedXAnimation(rotation=270)
timer = 1
age = 12

base_url = "https://api.openweathermap.org/data/2.5/weather"
api_key = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
params = {
 'lat': latitude,
 'lon': longitude,
 'appid': api_key,
 'units': 'metric'
}

```

In our code, we do the following:

- I. We set latitude to '42.346268' and longitude to '-71.095764' for our baseball diamond. For example, this is the GPS coordinates for Fenway Park in Boston, Massachusetts, US.
  - II. We create a GreenCheck object named go with a rotation value of 270 degrees.
  - III. We create a RedXAnimation object named no\_go with a rotation value of 270 degrees.
  - IV. We set our timer value to 1 second.
  - V. We set the age of our players to 12.
  - VI. Our code sets the base\_url value to "https://api.openweathermap.org/data/2.5/weather".
  - VII. Next, we add our OpenWeather api\_key value.
  - VIII. We then define a params dictionary we will use with our web service call (latitude, longitude, api\_key, and units).
3. We use an infinite loop to check the weather conditions every 60 seconds and update the display on our Sense HAT accordingly:

```

while True:
 response = requests.get(base_url, params=params)
 if response.status_code == 200:
 data = response.json()

 temperature = data['main']['temp']
 description = data['weather'][0]['main']

 print(f"The current temperature is {temperature}°C.")

```

```
 print(f"The weather is {description}.")

 if description == 'Thunderstorm' or description ==
'Rain' and age < 16:
 print("NO-GO!")
 no_go.display_animation(duration=1)
 timer = 1
 else:
 print("GO!")
 go.display()
 timer = 60

 else:
 print("Error: Failed to retrieve weather information.")

 time.sleep(timer)
```

In our code, we set up an infinite loop using `while True`:

- I. We make a GET request to the OpenWeather API using `requests.get()` and store the response in `response`.
- II. If the response status code is 200, we do the following:
  - i. We parse the JSON response into a Python dictionary using `response.json()` and assign it to `data`.
  - ii. We then retrieve the current temperature from `data['main']['temp']` and store it in `temperature`.
  - iii. We retrieve a weather description from `data['weather'][0]['main']` and store it in `description`.
  - iv. We then print the current temperature and weather description. If the weather description is 'Thunderstorm' or ('Rain' and `age < 16`), we print "NO-GO!" to the Shell, display the NO-GO animation using `no_go.display_animation(duration=1)`, and set the `timer` variable to 1 second. This is to make the total time before calling the web service 60 seconds, as the animation will go on for 59 seconds. Otherwise, we print "GO!" to the Shell and display the green checkmark animation using `go.display()` and then set the `timer` variable to 60 seconds.
- III. If the response status code is not 200, we print an error message.
- IV. We pause the execution for the value of `timer` seconds using `time.sleep(timer)`. This will result in a 60-second delay between calls to the OpenWeather web service.

4. We save our code as `go-no-go.py` in the GO-NO-GO folder and then run it by clicking on the green run button, hitting *F5* on the keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.

Upon running the code, we will observe the dot-matrix screen of our Sense HAT (or emulator) displaying either a green checkmark or a flashing red X sign, indicating a GO or NO-GO condition for a game at Fenway Park in Boston. As illustrated in *Figure 2.20*, the current status is a NO-GO for the game involving our players (under 16 years of age) due to the presence of thunderstorms:

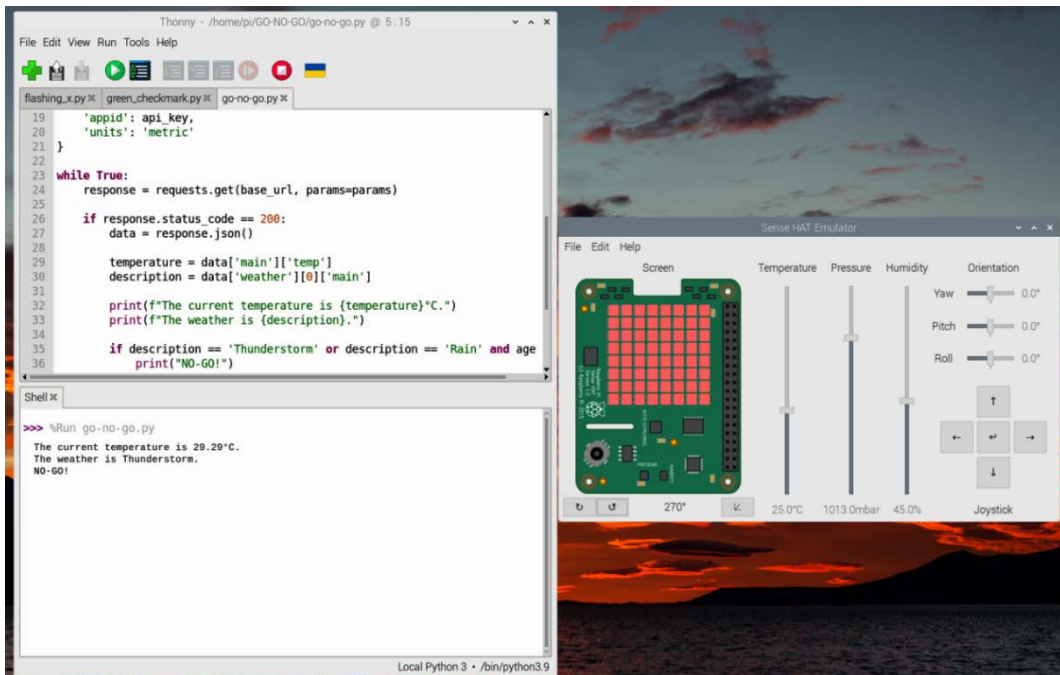


Figure 2.20 – Screenshot of the GO-NO-GO application using the Sense HAT emulator

As mentioned earlier, the flexibility of our code allows for easy expansion of the decision-making logic. In addition to weather data, we can extend our application to consider other factors such as wind speed, humidity, or any on-site sensor readings. By integrating sensors placed directly at the baseball diamond, we can gather real-time data on soil moisture levels or other measurements of interest. This sensor information can then be broadcasted to the internet, enabling us to seamlessly integrate it into our application.

To make our application more dynamic, we can incorporate scheduling information to determine the age of players scheduled to play at a specific baseball diamond at any given time. By extracting this information from a spreadsheet or an online repository, we can automate the process of obtaining



player age data and other game-related information such as whether the game is a playoff game. This allows our application to dynamically adjust its decision-making process, ensuring a more accurate GO or NO-GO decision.

## Building other GO-NO-GO applications

The GO-NO-GO application marks the last project in the book we will build using the Sense HAT. As we have demonstrated, the combination of the Raspberry Pi and Sense HAT makes for a powerful IoT device. It's not hard to imagine how we could easily change our baseball GO-NO-GO application for other scenarios. The following are a few examples of other GO-NO-GO applications we could build:

- **Flight status checker:** By integrating with a flight tracking API, we could build an application that can display a GO or NO-GO status for a specific flight.
- **Traffic condition monitor:** Utilizing a traffic data API, we could build an application that can assess current traffic conditions on a specific route or at a particular location.
- **Event availability indicator:** Integrating with an event ticketing API, we could build an application that can determine the availability of tickets for a desired event.
- **Public transportation tracker:** By connecting to a public transportation API, we could build an application that can provide real-time updates on the status of buses, trains, or other forms of public transportation.

The GO-NO-GO IoT application is but a glimpse into our vast potential in utilizing web services with IoT. With the Raspberry Pi and Sense HAT, our potential expands to diverse IoT applications, monitoring various data and fostering innovation beyond weather-related scenarios.

## Summary

In this chapter, we explored the world of web services development using Raspberry Pi and Sense HAT. We began by learning about web services and wrote web services code. With our newfound knowledge, we created our first IoT application: a scrolling stock ticker. By connecting to the Alpha Vantage web service, we retrieved real-time stock information and displayed it in a continuous scrolling format on the Sense HAT's dot-matrix display. This project demonstrated the ease of connecting to web services to obtain useful information.

Integrating web services with devices such as the Raspberry Pi is a skill crucial in today's tech industry. By handling data from sources such as Alpha Vantage and OpenWeather and displaying it on the Sense HAT, we've bridged theory with practical application. This knowledge enhances our project capabilities and professional skills, positioning us well in the IoT and data-driven domains.

---

We then ventured into building a weather display application. By leveraging the OpenWeather API, we obtained live weather information and transformed it into a scrolling message on the Sense HAT. We then took our development to the next step and used it to create a decision-making GO-NO-GO IoT application. In the GO-NO-GO application, we used weather conditions and player age as criteria to determine whether a baseball game should proceed (GO) or be canceled (NO-GO). We did so by displaying visual indicators such as a green checkmark or a flashing red X sign on the Sense HAT.

In the next chapter, we will explore IoT applications that involve physical interactions – specifically, the integration of motors. By incorporating motor control into our projects, we can create dynamic and interactive experiences that bridge the gap between the digital and physical worlds.



# 3

## Building an IoT Weather Indicator

In this chapter, we will learn about servo motors and LEDs and then use this knowledge, along with our understanding of the Raspberry Pi and web services, to create a practical project: an IoT weather indicator.

The weather indicator will draw weather information from a web service and then use a pointer attached to the servo motor to indicate suitable attire based on the current outdoor conditions. It will include an LED that turns on when it's raining and blinks on and off if there is a thunderstorm. The configuration that you'll use to create this application may be used for other applications, such as a water quality monitor or a traffic density monitor.

We'll start by looking at servo motors, which will help us understand what they are and how we may use them in our IoT applications. Then, we'll focus on LEDs.

Following that, we'll begin constructing a physical stand to house the components for our application. This will be the second specially designed stand for the Raspberry Pi that we will build for this book and the first one that features a motor.

While the usage of the SenseHAT case, as discussed in *Chapters 1* and *2*, was optional, it is recommended to build this stand for the weather indicator. This construction will represent our first foray into creating a tangible, physical entity, or more specifically, an Internet of Things (IoT) *thing*.

Once our stand has been assembled, we'll dive into coding. Our goal will be to extract information from the **OpenWeatherMap** web service and utilize it to dictate the position of the needle affixed to our servo motor based on the temperature and wind speed. We will also adjust the LED's behavior according to certain weather conditions. By implementing these processes, we will illustrate how real-world data can be transformed into physical movements, bridging the gap between the digital and mechanical worlds:

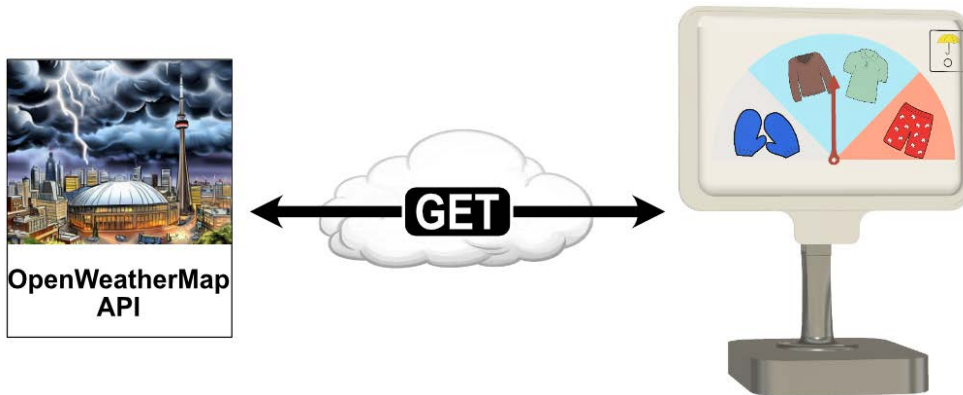


Figure 3.1 – Connecting our weather indicator to a web service

In this chapter, we will explore the following topics:

- Looking into servo motors
- Exploring LEDs
- Controlling servo motors and LEDs using Python
- Building the weather indicator stand
- Developing code for our application

Let's begin!

## Technical requirements

You'll need the following to complete this chapter:

- The Raspberry Pi 5 4/8 GB model is preferred, but the Raspberry Pi 4B with either 4 GB or 8 GB may be used (the figures in this book show the Raspberry Pi 4B model).
- The latest Raspberry Pi operating system with Thonny pre-installed.
- A keyboard, mouse, and monitor.
- 1x SG90 servo motor.
- 1x LED (single color).
- 1x 220 Ohm resistor.
- ½ inch PVC pipe.
- Jumper wires with connectors for the Raspberry Pi's **general-purpose input/output (GPIO)** port.

- Access to a 3D printer or 3D printing service for the custom stand.
- A general knowledge of programming. We will be using the Python programming language in this book.

The GitHub repository for this chapter is located at <https://github.com/PacktPublishing/Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter3>.

## Looking into servo motors

**Servo motors** are widely used in robotics, automation, and other applications where precise control of angular movement is required. Hooking up servo motors to the Raspberry Pi is a straightforward process that offers exciting possibilities for various projects. The Raspberry Pi provides a suitable platform to interface and control servo motors. In this section, we will investigate servo motors in more detail.

### Exploring the GPIO port

The GPIO port on the Raspberry Pi allows for direct hardware interaction, making it a key tool for hands-on projects. It allows us to connect devices, sensors, and circuits, allowing for real-world applications such as robotics or alarm systems to be created. By utilizing these pins, we can design and build projects that interact with the physical world. We will cover the GPIO port in more depth in *Chapter 5*. For now, it is enough to know that we can simply connect devices such as servo motors and LEDs to the GPIO port using female jumper connectors.

We will start by hooking up an SG90 servo motor to our Raspberry Pi 5 using the GPIO port.

## Connecting the SG90 servo motor to our Raspberry Pi

The SG90 servo motor is a popular and widely used micro-servo motor due to its compact size, versatility, precise control, and ease of use. When connecting the SG90 servo motor to our Raspberry Pi, it is necessary to modify the wire alignment within the female 3-prong connector. This connector is commonly known as the **JR-style servo connector** or simply **servo connector** and is initially wired with the power (+5V) and GND wires in reverse order from what is required for our Raspberry Pi.

Referring to *Figure 3.2*, we can adjust the wiring by performing the following steps:

1. First, identify the three wires in the connector – red (power), brown (ground), and orange (signal).
2. Next, gently pull the wires from the connector while holding the plastic housing up using a sharp object such as an Xacto knife.

- Then, rearrange them so that the red (power) wire is at one end, the brown (GND) wire is in the middle, and the orange (signal) wire is at the opposite end of the red wire:

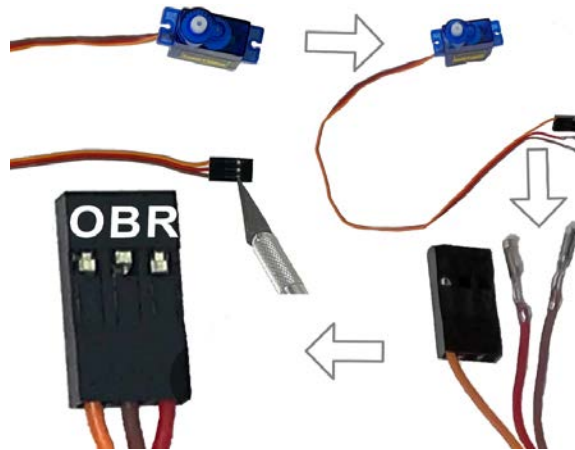


Figure 3.2 – Changing the wiring order for the SG90 servo

- With the wiring in the correct order, we can connect the servo motor directly to our Raspberry Pi. Using the diagram in *Figure 3.3* as a reference, connect the female 3-prong connector to the +5V, GND, and GPIO 14 pins of the Raspberry Pi:

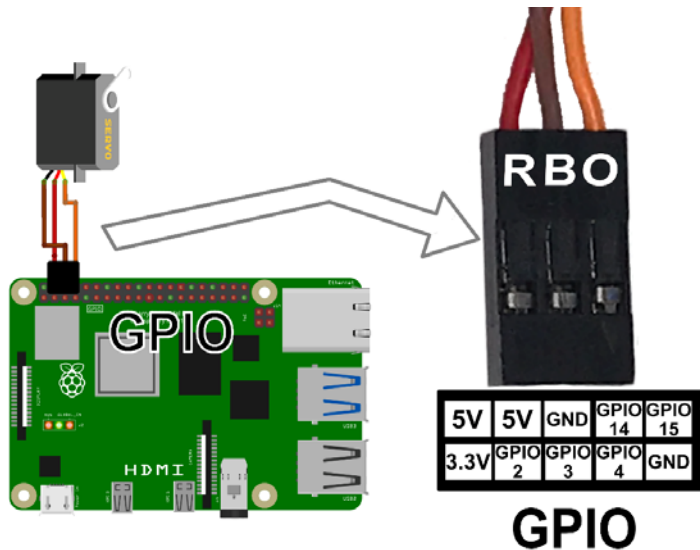


Figure 3.3 – Connecting our servo to the Raspberry Pi

### An alternate method for connecting our servo

For those of us who would rather not modify the existing connector of the servo, we may use three male-to-female jumper wires instead, with the male end inserted into the connector and the female end inserted into the GPIO port on the Raspberry Pi.

With our servo motor connected to our Raspberry Pi, let's investigate what servo motors are and how we may use them.

## Understanding servo motors

Servo motors consist of a **direct current (DC)** motor, a control circuit, and a feedback mechanism for maintaining the angular position of the output shaft. Robotics, toys, and radio-controlled cars are some of the applications where servos are predominantly used.

Servo motors stand out for their exceptional precision in controlling the position of the motor thanks to their closed-loop feedback mechanism. This mechanism constantly monitors the motor's actual position and adjusts it so that it matches the desired position, ensuring accurate and reliable performance. *Figure 3.4* shows the popular SG90 servo motor:



Figure 3.4 – SG90 servo motor

The range of movement for servo motors can vary, depending on the model. Some servos are designed for 180-degree movement, making them ideal for limited-range applications such as controlling robot joints or stabilizing camera gimbals. Other servos are capable of full 360-degree rotation, which makes them suitable for continuous motion applications, such as steering mechanisms or pan-tilt camera systems.



The angle control of servos is achieved through **pulse-width modulation (PWM)**. PWM involves sending varying pulse widths to the servo, akin to adjusting a volume knob to control sound level; in servos, these pulses dictate the arm's position. Different brands of servo have different maximum and minimum values to determine the angle of the servo needle. *Figure 3.5* demonstrates the relationship between PWM and the position of a 180-degree servo:

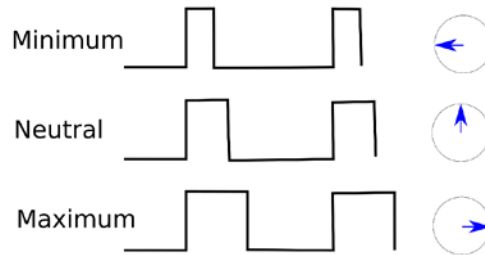


Figure 3.5 – PWM and servo position

Armed with our knowledge of the essentials of servo motors and their connection to the Raspberry Pi 5, we'll turn our attention to LEDs. Our exploration will cover how they function and the steps to incorporate them into our project, providing an additional layer of visual feedback.

## Exploring LEDs

LEDs were first developed in the early 1960s. The first LEDs were red and were first used as indicators in seven-segment displays. Today, LEDs are virtually everywhere – from indicator lights on our electronic devices and home appliances to the screens of our televisions and smartphones.

An LED is a simple semiconductor device. It has two leads – an anode (positive) and a cathode (negative). When a forward current passes through the diode from the anode to the cathode, it emits light. The color of the light depends on the materials that are used to make the diode and can range from infrared to ultraviolet, including all the colors of the visible spectrum.

LEDs come in various types, including single-color LEDs, RGB LEDs, which are capable of producing a multitude of colors, infrared LEDs, which are used in remote controls and night-vision systems, and bi-color LEDs, which can emit two different colors. *Figure 3.6* shows an array of LEDs ranging from a single-colored red to a seven-colored flash LED (second from left) to an RGB (second from right) LED capable of displaying any color. For our weather indicator, we will be using a single-colored LED. The color doesn't matter:

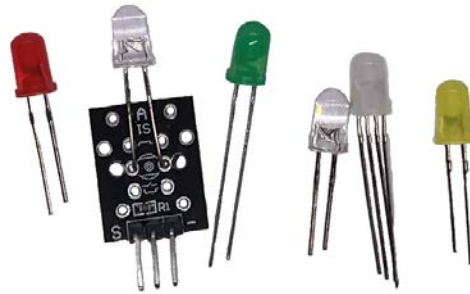


Figure 3.6 – LEDs in various formats

Now that we have a basic understanding of LEDs, it's time to connect an LED to our Raspberry Pi.

## Connecting an LED to our Raspberry Pi

Connecting an LED to a Raspberry Pi requires us to carefully consider voltage requirements – exceeding the voltage can cause the LED to burn out. To do this, we must solder a resistor to one end of the LED before connecting it to the GPIO port with jumper wires. Since the Raspberry Pi's GPIO pins output at a higher voltage than most LEDs can handle directly, the inclusion of the resistor is essential to regulate the voltage, ensuring that the LED operates correctly.

Figure 3.7 shows the materials that we need to connect our LED to the GPIO port on the Raspberry Pi – an LED, two jumper wires with female ends (brown and red), shrink tubing, and a 220 Ohm resistor:

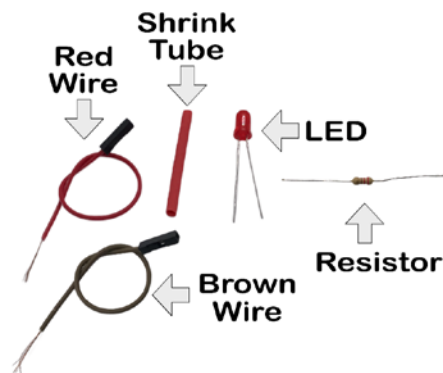


Figure 3.7 – The parts that are required to connect our LED to the Raspberry Pi

To connect our LED to our Raspberry Pi, we must do the following:

1. Start by soldering the resistor to the anode (positive) or longer leg of the LED (see *A* in *Figure 3.8*):

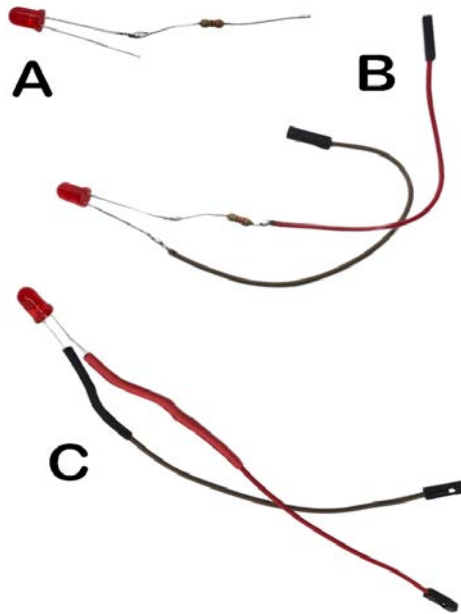


Figure 3.8 – Soldering a resistor and jumper wires to our LED

2. Then, splice and solder the brown jumper wire onto the cathode (negative) leg of the LED (see *B* in *Figure 3.8*).
3. Next, splice and solder the red jumper wire onto the other end of the resistor. We may consider the extra length created by the resistor and shorten the red wire accordingly (see *B* in *Figure 3.8*).
4. To strengthen the new connections and provide additional electrical insulation, apply heat shrink tubing over the soldered wires and resistor (see *C* in *Figure 3.8*).
5. Using the jumper wires, attach the brown wire to GPIO GND on the Raspberry Pi and the red wire to GPIO 25 (see *Figure 3.9*):

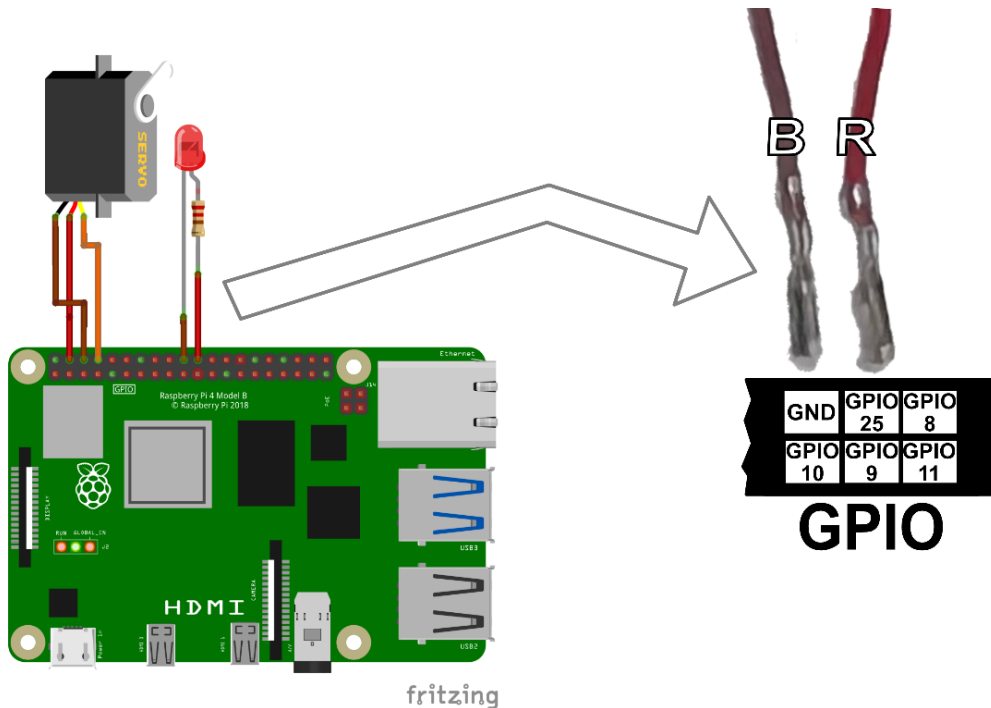


Figure 3.9 – Wiring up our servo and LED to the Raspberry Pi

Now that we've wired up our servo and LED components, let's write some code so that we can control our components through Python.

## Controlling servo motors and LEDs using Python

Having successfully connected the servo motor and LED to our Raspberry Pi, we'll start writing the Python control code. To facilitate this, we will be using the **GPIO Zero Python library**, a powerful tool for Raspberry Pi GPIO programming. Our first step in this process will be to set up a Python virtual environment so that we can develop our code.

### Setting up our development environment

Just like we did in *Chapter 2*, we will use a Python virtual environment for our development. As there are libraries that only work with the root installation of Python, we will use system packages in our Python virtual environment. To do this, follow these steps:

1. On our Raspberry Pi 5, open a Terminal application.
2. To store our project files, create a new directory by running the following command:

```
mkdir Chapter3
```

3. Then, navigate to the new directory:

```
cd Chapter3
```

4. Create a new Python virtual environment for our project:

```
python -m venv ch3-env --system-site-packages
```

5. With this command, we've created a new Python virtual environment called `ch3-env` and enabled access to the system site packages. With our new Python virtual environment created, we can source into it with the following command:

```
source ch3-env/bin/activate
```

6. Install the extra packages that are required for our code with the following command:

```
pip install requests
```

7. The `requests` library in Python simplifies making HTTP requests to web servers. We will use the `requests` library when we pull weather data from the web. With the `requests` library installed, we may close the Terminal by running the following command:

```
exit
```

With our project folder created, our Python virtual environment set up and activated, and the `requests` package installed, we may now start writing code. We will start by controlling the servo motor through Python code using the Terminal.

## Using GPIO Zero to control a servo

GPIO Zero is a Python library for controlling the GPIO pins on the Raspberry Pi. It was created in 2016 by Ben Nuttall and Dave Jones of the Raspberry Pi Foundation. GPIO Zero provides a user-friendly and high-level interface, making it easier to work with GPIO, including controlling LEDs, buttons, servos, and more. It comes pre-installed with the latest Raspberry Pi operating system.

The `Servo` class is a part of GPIO Zero and provides a way to control a servo motor. To test our servo motor's connection to our Raspberry Pi, follow these steps:

1. Open a Terminal window and navigate to our project folder by running the following command:

```
cd Chapter3
```

2. Source our `ch3-env` virtual environment with the following command:

```
source ch3-env/bin/activate
```

3. Open a Terminal window in our `ch3-env` virtual environment and launch Python with the following command:

```
python
```

4. Then, enter the following code to import the `Servo` class and create an object called `servo`. After that, initialize it with the PIN we connected our Servo to in *Figure 3.3*:

```
from gpiozero import Servo
servo = Servo(14)
```

5. The `Servo` class provides several useful methods to control a servo motor. To set the servo to the minimum position, run the following command:

```
servo.min()
```

6. After executing this command, we should notice that our servo motor has pivoted completely to one side. To move our servo motor to the middle position, we can use the following Python command:

```
servo.mid()
```

7. After executing this command, we should observe that our servo motor has moved to the mid position. For the final test, we'll move our servo motor to the maximum position:

```
servo.max()
```

8. We should observe that our servo motor moves to the maximum position (we shouldn't be alarmed if the motor doesn't move as far to the maximum position as it does to the minimum position as we will calibrate the motor later in this chapter). To close our servo connection, run the following command:

```
servo.close()
```

#### Why is the servo motor jittering?

We may observe jittering from our SG90 servo motor when it's controlled through GPIO Zero. A multitude of factors may contribute to this issue, ranging from the power provided to the servo via the GPIO port, to potential mechanical problems with the servo, or even software-related issues within the library. Although an investigation into these causes falls outside the scope of this project, we must acknowledge the potential for motor jittering. A straightforward solution involves closing the connection to the servo using the `servo.close()` command after setting its position.

With our servo motor tested, we can focus on the LED. In the next section, we will write some code to control the status of our LED using the GPIO library.

## Using GPIO Zero to control an LED

The `LED` class is a part of the GPIO Zero library and provides a simple interface to control an LED. We can use this class to control our LED. To test the connection of our LED, follow these steps:

1. First, open a new Terminal window and navigate to our project folder with the following command:

```
cd Chapter3
```

2. Then, source our `ch3-env` virtual environment with the following command:

```
source ch3-env/bin/activate
```

3. Launch Python by running the following command:

```
python
```

4. Enter the following code to import the `LED` class and create an object called `led`. Once you've done this, initialize it with the PIN we connected our LED to in *Figure 3.9*:

```
from gpiozero import LED
led = LED(25)
```

5. The `LED` class provides several useful methods to control an LED. To turn on the LED, type the following:

```
led.on()
```

6. After executing this command, we should notice that our LED has turned on. For the next test, we'll turn our LED off by running the following Python command:

```
led.off()
```

7. After executing this command, we should observe that our LED has turned off. For the final test, we'll blink our LED:

```
led.blink()
```

8. We should observe that our LED starts blinking. To stop this and turn the LED off, run the following command:

```
led.off()
```

Should we encounter issues during testing, there could be several potential causes:

- **Incorrect wiring:** Incorrect wiring is among the most common problems. It's crucial to verify our connections and ensure we've wired the correct GPIO pin according to our Python script.

- **Issues with the power supply:** Power supply inadequacies might also lead to issues. While the Raspberry Pi's GPIO pins may not supply adequate power for certain servos, especially under load, causing the servo to act unpredictably, our SG90 servo and LED shouldn't face this issue, given they have a lower power demand.
- **Software:** Software-related issues could also pose problems. Keeping the Raspberry Pi OS and GPIO Zero library up to date is an essential preventative measure.
- **Components:** Issues could lie within the components themselves. By testing them with a known, functioning device, we can either confirm or rule out this possibility.

Now that we've tested our servo and LED components alongside the corresponding code, we can construct the stand that will house our project.

## Building the weather indicator stand

In this section, our focus will shift to assembling the stand for our weather indicator. While it's not imperative for running the code, building the stand is highly recommended as it adds a tangible, real-world aspect to our project, bringing the *thing* in IoT to life.

While this book does not provide a comprehensive guide on 3D printing, we will briefly outline the key steps involved in fabricating the stand. We'll be using a standard **fused deposition modeling (FDM)** printer, ideally one with a print size such as the "Ender-3" (220mm x 220mm x 250mm). Our material of choice for this exercise is **polylactic acid (PLA)** since it's known for its ease of use and suitability for beginners. Alternatively, **polyethylene terephthalate glycol-modified (PETG)** is also a great option, offering enhanced strength and flexibility. We'll go through the basics of printing and assembling the stand components here.

### Should I print in PLA or PETG?

When deciding whether to print in PLA or PETG, there are several factors for us to consider. PLA is known for its ease of use, making it a popular choice for beginners. It prints at lower temperatures, doesn't warp as easily as other materials, and generally provides more precise details. PETG is known for its strength and flexibility, which surpasses that of PLA. PETG prints at a higher temperature than PLA and has excellent layer adhesion, resulting in more robust prints. However, PETG can be more challenging to print with due to its tendency to string or ooze. For the weather indicator stand we built for this chapter, PLA was used. This was because printing was done directly on a glass plate and PETG tends to stick too well to glass, potentially resulting in damage being done to the build plate upon removing the print.

For those of us who do not have access to a 3D printer or prefer not to print the parts ourselves, using a 3D printing service such as Shapeways (<https://www.shapeways.com>) is a convenient alternative.



Now that we've covered the options for obtaining our 3D-printed parts, either by printing them ourselves or using a service, let's move on to the next crucial phase of our project – assembling the weather indicator stand.

## Assembling the weather indicator stand

As mentioned previously, the weather indicator stand has been engineered to be composed of components produced via an FDM 3D printer. Unlike the SenseHAT case and stand from *Chapters 1* and *2*, the design approach for this weather indicator stand is built with a *plate-like* configuration. This method effectively eliminates thin horizontal walls, a structural vulnerability that's frequently found in parts created with FDM technology.

### Finding the files for 3D printing

The 3D model files for the parts, which have been specifically designed for 3D printing, can be downloaded from the `Build Files` directory in this chapter's GitHub repository. Please note that these files are provided in the `.stl` format only, not in other 3D model file formats.

Before we start constructing our stand, let's look at the parts.

### Identifying the parts

*Figure 3.10* shows the parts that make up the weather indicator stand. All the parts, except for the base, were printed on an FDM 3D printer. Two additional parts – a half-inch PVC pipe and the front sticker – aren't shown:

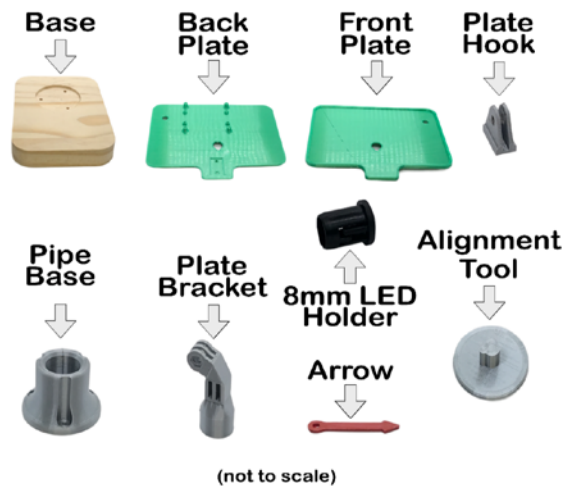


Figure 3.10 – Parts of the weather indicator stand

Let's take a closer look at each part:

- **Base:** The base serves as the foundation for our stand. It's the component where the **pipe base** is fastened. While the base is an integral part of the design, it can be considered optional, depending on the specific installation. For instance, the stand could be attached directly to a table, wall, or even a ceiling, offering flexible positioning based on individual preferences and requirements. The base displayed in *Figure 3.10* was created out of 20mm pine wood using a CNC router (a computer-controlled cutting machine that's used for cutting various hard materials, such as wood), although a 3D printer may be used to make this part as well.
- **Back plate and front plate:** These halves, when glued together, form the stand's plate. The division into halves aligns with the need for a flat surface in 3D printing, given the plate's lack of an inherent flat side. The plate is the main component of our stand as it holds the servo motor, arrow, and Raspberry Pi, which is attached to the back. The plate has been designed to hold either a Raspberry Pi, a Raspberry Pi Zero, or a Raspberry Pi Pico W.
- **Plate hook:** Designed to be compatible with GoPro series stands, the hook provides the main connection of the plate on the stand. Of note here is the deliberate printing orientation and built-in support of the plate hook to eliminate line adhesion issues that would cause the plate hook to break apart if printed in the traditional flat side down (see *Figure 3.11*):



Figure 3.11 – Plate hook print orientation

- **Pipe base and plate bracket:** These two components hold the ½ inch PVC pipe (not shown). The pipe base attaches the PVC pipe to the base and the plate bracket attaches the PVC pipe to the plate hook.
- **8mm LED holder:** We utilize the LED holder to fasten the LED onto the front of the main plate face.

- **Arrow:** The arrow is connected to the servo motor and acts as the analog needle for our weather indicator application.
- **Alignment tool:** This tool acts as a guide that aligns the back plate and front plate correctly during gluing, ensuring they are properly aligned to avoid any issues when installing the servo motor.
- **PVC pipe (not shown):** The stand features a half-inch PVC pipe serving as its stem or central support. This design offers the flexibility to adjust the stand's height according to our needs. In situations where the base is unnecessary, such as when securing it to a table, wall, or ceiling, we can easily set the stand's height to the desired level.
- **Face sticker (not shown):** The weather indicator's face is created using a sticker printed on sticker paper with an inkjet printer. To ensure precise placement on the main plate, a vinyl cutter such as the **Silhouette Cameo** can be utilized to cut out the sticker. For convenience, we have included both a *Silhouette* file and a *.svg* file with the cut lines in the *Build Files* folder of our GitHub repository.
- **M5 20mm bolt and M5 nut (not shown):** To fasten the plate to the pedestal portion of the stand, we use a 20mm M5 bolt and corresponding M5 nut.

With the parts identified, let's start by creating the main plate for our stand.

### *Assembling the plate*

The plate or face of the weather indicator stand is created by gluing the front plate and back plate together. To assemble the plate, follow these steps:

1. First, apply epoxy glue to the flat side of one of the plates (see *A* in *Figure 3.12*). When applying glue, it's crucial to avoid spreading glue near the servo or LED openings. It's recommended to apply glue right to the boundaries of the plate:



Figure 3.12 – Assembling the plate

2. After applying glue, put the flat sides of the plates together. Using the alignment tool, ensure that the plates are aligned with each other (see *B* in *Figure 3.12*).
3. Once the plates have been aligned, remove the alignment tool.
4. Should it be needed, use clamps on the plates to prevent any space from forming between them. Let the glue dry before proceeding (see *C* in *Figure 3.12*).
5. After the plates have dried, apply epoxy glue to the flat part of the hook and place it in the groove on the back of the plate (see *D* in *Figure 3.12*).
6. After the glue has dried, paint the plate if you wish.
7. To finish off the assembly, print and cut out the face sticker and apply it to the front of the plate inside the ridges (see *A* and *B* in *Figure 3.13*, respectively):



Figure 3.13 – Plate face sticker

With the main plate constructed, it is now time to build the pedestal, which will hold the main plate.

### ***Assembling the pedestal***

The pedestal portion of our stand includes the base, the pipe base, the PVC pipe, and the plate bracket. To assemble the pedestal, follow these steps:

1. Start by inserting the PVC pipe (cut to a desired length) into the pipe base (see *A* in *Figure 3.14*):

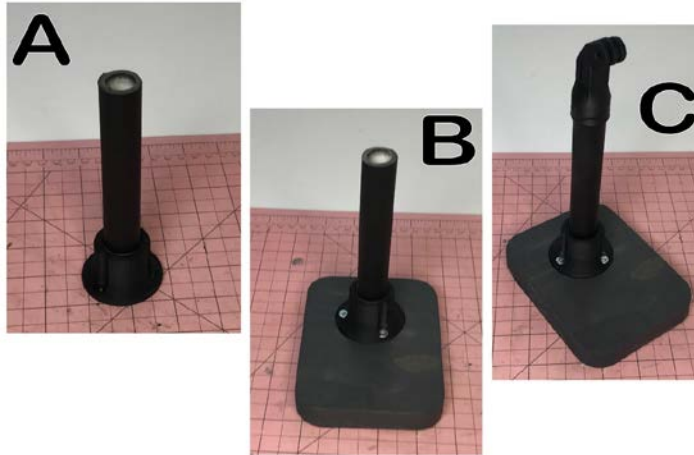


Figure 3.14 – Assembling the pedestal

2. Then, secure the pipe base to the base, table, wall, or ceiling as desired (see *B* in *Figure 3.14*).
3. To finish assembling the pedestal, insert the plate bracket into the top of the PVC pipe (see *C* in *Figure 3.14*).

Pay attention when assembling the pedestal as variations in the manufacturer's PVC pipe could result in an improper fit or difficulty when inserting it into the pedestal parts. If the pipe fits too loosely, it is recommended to apply epoxy glue to securely attach the pipe. Conversely, if the pipe is too thick to fit properly, we have two options: either sand down the insides of the pipe base or plate bracket or sand the outside of the pipe to achieve a proper and snug fit.

It's important to be aware that in our example, the components of the pedestal were coated with flat black spray paint, so any additional thickness resulting from the paint should be considered.

With the pedestal components constructed, we're ready to integrate our electronic components into our weather indicator stand.

### ***Installing the Raspberry Pi, servo, and LED***

With our plate and pedestal constructed, the last thing we must do before putting the stand together is install our Raspberry Pi, servo motor, and LED onto the plate and wire all these components together.

To do this, follow these steps:

1. Utilizing 10mm M2.5 bolts, fasten our Raspberry Pi to the base of the plate, ensuring it aligns correctly with the appropriate stand-offs (see *A* in *Figure 3.15*).
2. With the aid of a hot glue gun (preferred) or epoxy glue, secure our SG90 servo to the rear side of the plate. To do this, align the front structure of the SG90 with the hole present in the plate (see *B* in *Figure 3.15*):

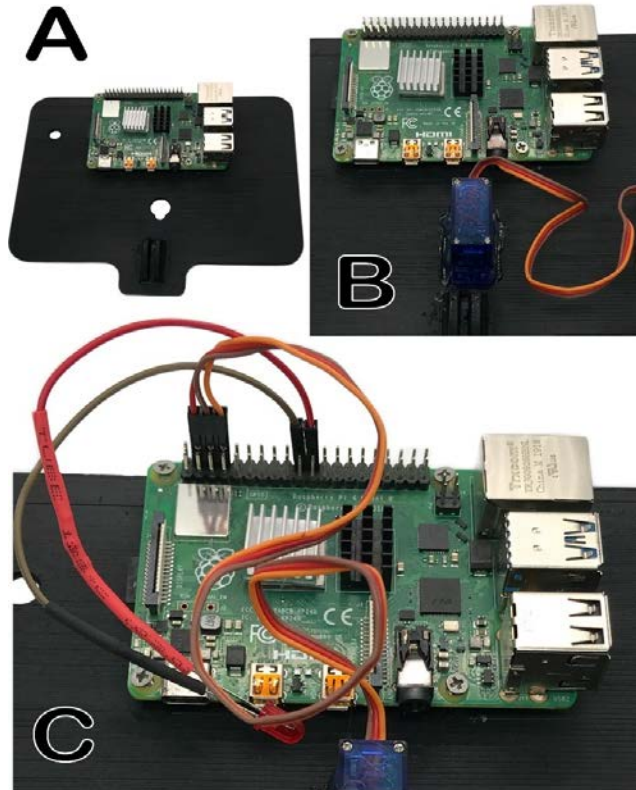


Figure 3.15 – Installing the Raspberry Pi, servo, and LED

3. Referring to the wiring diagrams in *Figure 3.3* and *Figure 3.9*, wire up our servo motor and LED to the GPIO port of the Raspberry Pi (see *C* in *Figure 3.15*).
4. With our Raspberry Pi wired up, it's time to put the LED in place. Start by threading the LED through the LED hole in the plate (see *A* in *Figure 3.16*):

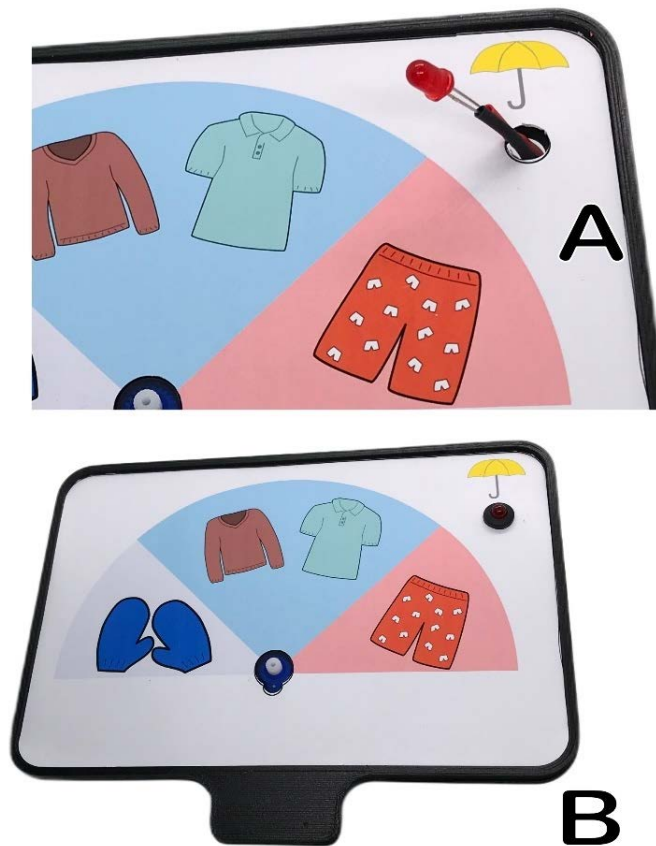


Figure 3.16 – Putting our LED in place

5. Then, thread the LED through the LED holder and push the holder into the LED hole in the front of the plate (see B in Figure 3.16).

With our plate fully wired and all the components securely fastened, we can attach the plate to the pedestal.

### ***Connecting the plate to the pedestal***

By attaching the plate to the pedestal, we will have finished constructing our weather indicator stand.

To complete our stand, follow these steps:

1. Start by aligning the hook on the back of the plate with the plate bracket on the pedestal (A in Figure 3.17):



Figure 3.17 – Attaching the plate to the pedestal

2. Then, insert a 20mm M5 bolt and fasten it with an M5 nut (see the red rectangle in A in *Figure 3.17*).
3. Finally, make any final adjustments to the angle of the plate before tightening the bolt fully (see B in *Figure 3.17*).

With that, we've finished assembling our weather indicator stand. This setup could be repurposed for many other IoT applications, including monitoring traffic conditions or tracking fluid levels in industrial environments.

Our next step is to write the code that will retrieve weather data from a web service and use it to control the servo motor and LED. Now is also the best time to introduce the arrow into our setup as it needs to be calibrated before it can be used in our code.



## Developing code for our application

In this section, we will connect our Raspberry Pi to the `OpenWeatherMap.org` web service and use the data to control the position of the needle on our weather indicator. For this, we need to use the API key we received from `OpenWeatherMap.org` when we set up an account in *Chapter 2*.

Throughout this section, our Raspberry Pi, once mounted, will remain connected to a monitor, keyboard, and mouse. When the moment arrives to set up our weather indicator independently, all we'll need is a power source and a means to remotely access the Raspberry Pi via SSH. Discussing how to configure a Raspberry Pi so that it can operate in *headless* mode falls outside the scope of this chapter. However, detailed instructions for this setup can easily be found through various online resources. Helpful starting points include the official Raspberry Pi documentation, technology-focused forums such as Stack Overflow, and dedicated Raspberry Pi community websites and blogs.

We will follow the software architecture laid out in *Figure 3.18* for our code. At the heart of our architecture is the `WeatherData` class. We will use this class to connect to the `OpenWeatherMap` web service:

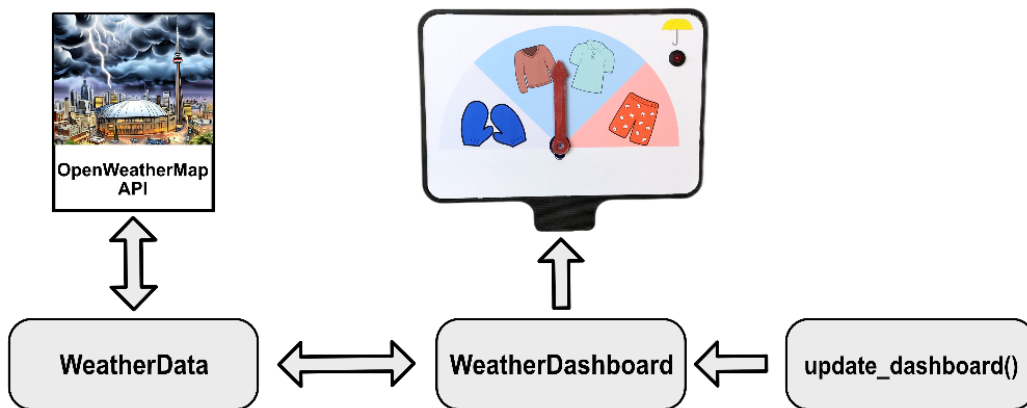


Figure 3.18 – Software architecture for our weather indicator

The `WeatherData` class fetches and processes weather data from the `OpenWeatherMap` API, while the `WeatherDashboard` class controls the display mechanisms: a needle indicator controlled by a servo motor and an LED that changes state based on the weather conditions.

These two classes work together, with `WeatherData` fetching and processing data, which is then used by `WeatherDashboard` to adjust the display. To keep the data current, the `update_dashboard()` function is set to run in an infinite loop, refreshing weather data and updating the display every 30 minutes.

To address the unique hardware constraint of the servo motor – that is, its tendency to jitter if the connection is not closed after a move – we'll create a fresh instance of `WeatherDashboard` (and consequently, a new `Servo` instance) with every cycle of the loop. While this approach may not be the most common in software design, it's a practical solution that successfully solves the servo jitter issue by *reopening* the connection with each new instance.

We'll begin by calibrating the arrow's (or needle's) position for optimal application performance before we create our `WeatherData` and `WeatherDashboard` classes.

## Calibrating the needle

Before we can use our weather indicator application, we must calibrate the needle (or the position of the arrow). To do so, follow these steps:

1. Close any open Terminals, open a new Terminal window, and navigate to our project folder by running the following command:

```
cd Chapter3
```

2. Then, source our `ch3-env` virtual environment with the following command:

```
source ch3-env/bin/activate
```

3. Launch Python with the following command:

```
python
```

4. Now, enter the following code to import the `Servo` class and create an object called `servo`:

```
from gpiozero import Servo
servo = Servo(14)
```

5. Next, set the servo to the mid position with the following command:

```
servo.mid()
```

6. Using the picture of the face of our weather indicator, install the arrow into the servo in the mid position.

7. With the needle (arrow) in place, let's do some testing. We will start by moving the needle to the minimum position with the following command:

```
servo.min()
```

8. We should notice that the needle swings to the right. This is not the behavior we want as the right-hand side should represent the maximum value, not the minimum value. We should also note that the needle did not swing 90 degrees as we were expecting. We will address both issues in the `WeatherData` and `WeatherDashboard` classes, respectively.

We are now ready to write code to control the needle and LED on our weather indicator. We will start with the `WeatherData` class.

## Creating the `WeatherData` class

The `WeatherData` class is designed to pull weather information for a given city using the OpenWeatherMap API, after which it will process this data to calculate servo and LED values based on the temperature, wind speed, and weather conditions. When initialized with a city name, the class fetches the weather data for that city and stores the temperature, weather conditions, and wind speed. It also provides methods, `getServoValue()` and `getLEDValue()`, to determine the output for the servo motor and LED, respectively, based on the retrieved weather data.

Let's get started:

1. Launch Thonny by clicking on the **Menu** icon in the Raspberry Pi taskbar, navigating to the **Programming** category, and selecting **Thonny** (Figure 3.19):

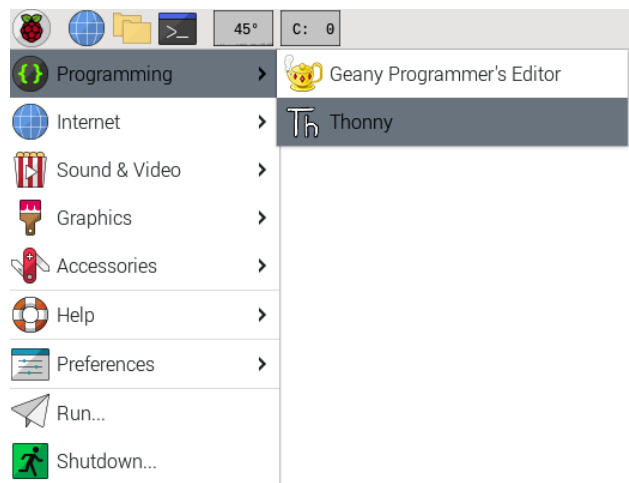


Figure 3.19 – Opening Thonny from the main menu

2. By default, Thonny uses the Raspberry Pi's built-in version of Python. For our project, we will use the Python virtual environment we just created. To start, we need to view the project files by clicking on **View** and selecting **Files** if it isn't already selected.
3. In the **Files** section, locate the `ch3-env` directory.
4. Then, right-click on the folder and select **Activate virtual environment**.

5. Once inside Thonny, create a new tab by selecting **File** and then **New** or by hitting *Ctrl* + *N* on your keyboard.
6. Let's start our code by entering our imports:

```
import requests
```

Here, `import requests` imports the `requests` library from Python, which is used to send HTTP requests, such as GET and POST requests, to interact with APIs or web services (refer to *Chapter 2*, for clarification on HTTP requests).

7. Then, define our class name, `WeatherData`, and our class variables:

```
class WeatherData:
 temperature = 0
 weather_conditions = ''
 wind_speed = 0
 city = ''
```

8. Our constructor takes one parameter, `city`. Make the call to the OpenWeatherMap web service in the constructor (when an instance of the `WeatherData` class is created):

```
def __init__(self, city):
 self.city = city
 api_key = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
 base_url = "http://api.openweathermap.org/data/2.5/
weather"

 complete_url = f"{base_url}?q={self.city}&appid={api_
key}&units=metric"

 response = requests.get(complete_url)
 data = response.json()

 if data["cod"] != "404":
 main = data["main"]
 wind = data["wind"]
 weather = data["weather"]
 self.temperature = main["temp"]
 self.weather_conditions = weather[0]["main"]
 self.wind_speed = wind["speed"]
```

Let's take a closer look at our code:

- `self.city = city`: saves the city name to the object's property
  - `api_key = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'`: Sets the OpenWeatherMap API key
  - `base_url = "http://api.openweathermap.org/data/2.5/weather"`: Sets the base URL for the OpenWeatherMap API
  - `complete_url = f"{base_url}?q={self.city}&appid={api_key}&units=metric"`: Formats the full API URL with the city and API key
  - `response = requests.get(complete_url)`: Sends a GET request to the API and saves the response
  - `data = response.json()`: Converts the API response from JSON format into a Python dictionary
  - `if data["cod"] != "404"`: Checks whether the API response is not a 404 error, which would indicate that the city was not found
  - `self.temperature = main["temp"]`: Saves the current temperature in the city to the object's property
  - `self.weather_conditions = weather[0]["main"]`: Saves the current weather conditions in the city to the object's property
  - `self.wind_speed = wind["speed"]`: Saves the current wind speed in the city to the object's property
9. By making the call to the OpenWeatherMap web service and setting the instance variables, all that's left for our code to do is convert weather data into values for our servo motor and LED. We'll start with the servo motor value and the `getServoValue()` method:

```
def getServoValue(self):

 if self.temperature < 0:
 temp_factor = 0
 elif self.temperature > 30:
 temp_factor = 1
 else:
 temp_factor = self.temperature / 30

 wind_factor = self.wind_speed / 20
 servo_value = -((temp_factor - (wind_factor/20)) * 2-1)

 return servo_value
```

---

Here's a breakdown of our code:

- I. First, the method checks whether the temperature (`self.temperature`) is less than 0. If the temperature is below zero, `temp_factor` is set to 0. This means that if the temperature is negative, there will be no contribution to the final servo value.
- II. Then, we check whether the temperature is greater than 30. If this is true, then `temp_factor` is set to 1.
- III. If the temperature falls between 0 and 30 degrees Celsius, then we normalize the temperature data. Normalization is the process of bringing or converting values into a common range. In this case, the temperature data is being scaled to a range of 0 to 1, assuming the temperature is within the range of 0 to 30 degrees Celsius. By dividing the temperature by 30, any temperature value within this range is proportionally scaled down to a fraction between 0 and 1. For example, if the temperature is 15 degrees Celsius, `temp_factor` will be 0.5. This normalization allows the method to process temperatures consistently, irrespective of their actual value.
- IV. `wind_factor = self.wind_speed / 20` calculates the wind factor by normalizing the wind speed. Like the temperature normalization process, this line of code is used to convert the wind speed into a value within a range of 0 to 1. The wind speed is assumed to be within a range of 0 to 20 kilometers per hour. For example, if the wind speed is 10 kilometers per hour, `wind_factor` will be 0.5. Normalization is particularly useful when combining or comparing different types of data, such as wind speed and temperature in this case.
- V. The next step is to calculate `servo_value` using both the temperature and wind factors. Specifically, it creates a dampening effect on temperature due to wind speed, representing a wind chill factor. The formula reduces the temperature factor (`temp_factor`) by one-twentieth of the wind factor (`wind_factor`), thereby signifying that for every 5% increase in wind speed, the temperature's impact decreases by 1%. This is based on our discretionary assumption that wind speed has a cooling effect. After this adjustment for wind chill, the resulting value is transformed to fit into the range of -1 to 1, which is suitable for the servo's operation. It multiplies the value by 2, shifting the range from 0 to 1 to 0 to 2, and subtracts one to adjust the range to -1 to 1. Finally, the result is negated to account for the servo's directionality (as we discovered when calibrating the needle). The calculated `servo_value` represents the position of the servo as per the adjusted temperature and wind speed readings.
- VI. Finally, our method returns `servo_value`, which is a value between -1 and 1 based on temperature, and then adjusted for wind speed. This value is ready to be used as input to the servo's value property in the GPIO Zero library.

10. The second method within the `WeatherData` class is utilized to determine the operational state of the LED, which varies based on the current weather conditions. More specifically, the LED is programmed to flash during a thunderstorm, remain solidly illuminated in the event of rain, and be switched off when the weather conditions are neither a thunderstorm nor rain. We call this method `getLEDValue()`:

```
def getLEDValue(self):
 if (self.weather_conditions=='Thunderstorm'):
 return 2;
 elif (self.weather_conditions=='Rain'):
 return 1
 else:
 return 0
```

Here, the `getLEDValue()` method designates specific values to different weather conditions: it returns 2 for thunderstorms, 1 for rain, and 0 for other conditions. These values are then utilized to control the LED's behavior – blinking for thunderstorms (2), a steady one for rain (1), and off for clear weather (0).

11. To test out our program directly, include the following code:

```
if __name__=="__main__":

 weather = WeatherData('Toronto')
 print(weather.getServoValue())
 print(weather.getLEDValue())
```

Let's take a closer look:

- `if __name__=="__main__":` This is the entry point for the program. It is only executed when the script is run directly, not when it is imported as a module in another script:
- `weather = WeatherData('Toronto')`: This creates an object of the `WeatherData` class, initializing it with the 'Toronto' string. We may enter a city of our choice. To verify whether the city we are interested in has data available, we can enter it into the search box on the OpenWeatherMap website (<https://openweathermap.org/>).
- `print(weather.getServoValue())`: This is a function call to the `getServoValue()` method of the `weather` object. This method calculates a value based on the temperature and wind speed data for Toronto and then prints this value to the console.
- `print(weather.getLEDValue())`: This calls the `getLEDValue()` method of the `weather` object, which sets a flag based on the weather conditions (thunderstorm, rain, or other) for Toronto. This flag determines the state of an LED (flashing, on, or off or 2, 1, or 0, respectively) and the method returns this state. The state is then printed to the console.

12. Save the code and name the file `WeatherData.py`. Run the code by clicking the green run button, hitting `F5` on your keyboard, or clicking on the **Run** menu option at the top and then **Run current script** (Figure 3.20):

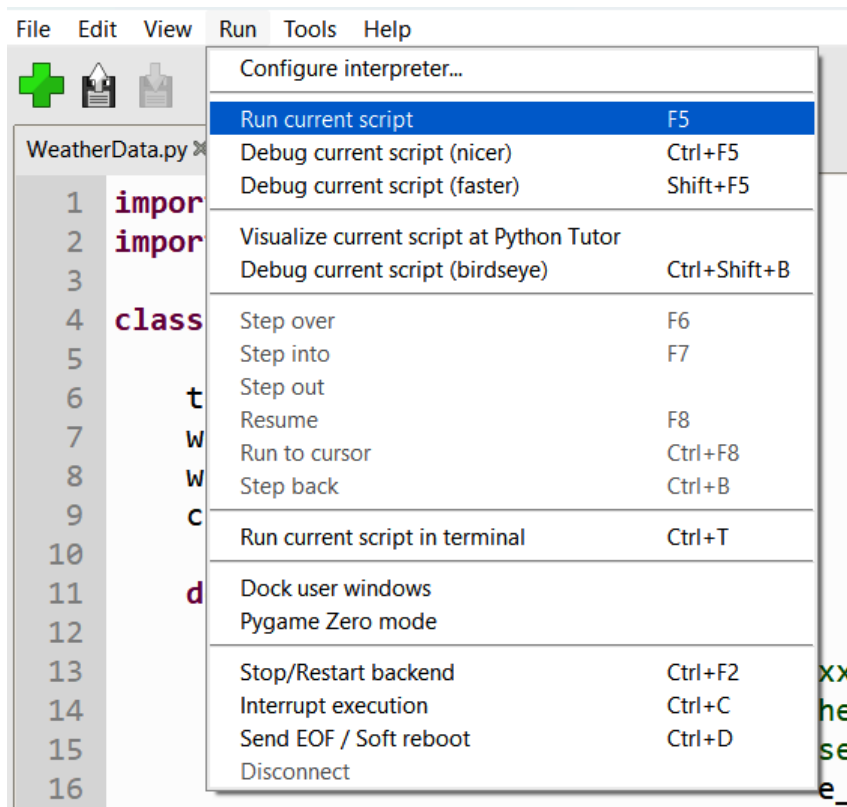


Figure 3.20 – Running a script in Thonny

13. You should see a response similar to the following:



Figure 3.21 – Output from running `WeatherData.py`



With the `WeatherData` class written and tested, it's time to move on to the code that controls the weather indicator: the `WeatherDashboard` class.

## Creating the `WeatherDashboard` class

The `WeatherDashboard` class is integral to the operation of the weather indicator as it's responsible for controlling the position of the needle and the state of the LED. This class utilizes the `WeatherData` class to gather and interpret weather data, which is then utilized to guide the operation of the physical display. The position of the servo, which dictates the position of the needle, is determined by the temperature and wind speed, while the LED state is indicative of specific weather conditions, such as rain or thunderstorms.

To create this class, follow these steps:

1. Inside Thonny, activate the `ch3-env` virtual environment if it hasn't been already.
2. Then, create a new tab by selecting **File** and then **New** or by hitting `Ctrl + N` on your keyboard (Figure 3.22):

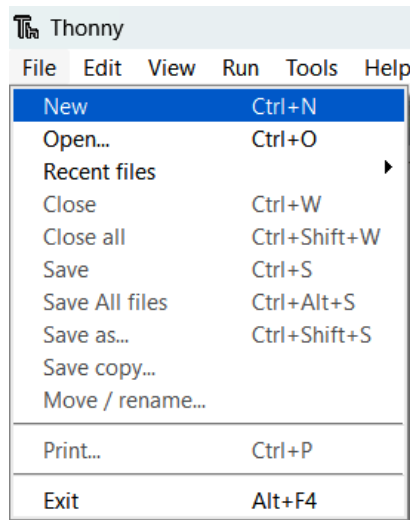


Figure 3.22 – Creating a new file in Thonny

3. We'll start with our imports:

```
from gpiozero import Servo
from gpiozero import LED
from time import sleep
from WeatherData import WeatherData
```

Here, we do the following:

- I. We start by importing the `Servo` and `LED` classes from the `gpiozero` library.
  - II. Then, we import the `sleep` function from the `time` module.
  - III. Finally, we import the `WeatherData` class to fetch weather condition data and derive the control values for the servo motor and the LED accordingly.
4. With the imports added, define the class and the class variables:

```
class WeatherDashboard:
 servoCorrection = 0.5
 maxPW = (2.0 + servoCorrection) / 1000
 minPW = (1.0 - servoCorrection) / 1000
```

Here, we have the following:

- `servoCorrection = 0.5`: This line declares a class-level variable called `servoCorrection` and assigns it a value of `0.5`. This is used to adjust the minimum and maximum pulse widths for the servo motor.
  - `maxPW = (2.0 + servoCorrection) / 1000`: This line calculates and sets the maximum pulse width (`maxPW`) that can be provided to the servo motor. It's computed by adding `2.0` to the `servoCorrection` variable and then dividing the result by `1000` to convert it into seconds – the servo motor expects pulse widths in seconds.
  - `minPW = (1.0 - servoCorrection) / 1000`: Like the previous line, this line calculates and sets the minimum pulse width (`minPW`) for the servo motor. It subtracts `servoCorrection` from `1.0` and then divides the result by `1000` for the same reason as before.
5. Create our `Servo` and `LED` instance objects while initializing the `WeatherDashboard` class:

```
def __init__(self, city, servo_pin, led_pin):
 self.city = city
 self.servo = Servo(servo_pin,
 min_pulse_width=self.minPW,
 max_pulse_width=self.maxPW)
 self.led = LED(led_pin)
```

Let's take a closer look at the code:

- `self.city = city`: This line sets an instance variable, `city`, equal to the `city` argument that's passed when the instance of the class is created. This variable represents the city for which weather data will be fetched.

- `self.servo = Servo(servo_pin, min_pulse_width=self.minPW, max_pulse_width=self.maxPW)`: This line creates a Servo object from the GPIO Zero library. The `servo_pin` argument specifies the GPIO pin that the servo is connected to. `min_pulse_width=self.minPW` and `max_pulse_width=self.maxPW` set the minimum and maximum pulse widths for the servo using class variables.
  - `self.led = LED(led_pin)`: This line creates an LED object from the GPIO Zero library. The `led_pin` argument indicates the GPIO pin the LED is connected to. This LED will be controlled based on the weather conditions that have been fetched for the specified city.
6. The `update_status()` method of the `WeatherDashboard` class grabs the latest weather information and updates the status of the needle and LED accordingly:

```
def update_status(self):
 weather_data = WeatherData(self.city)

 self.servo.value = weather_data.getServoValue()
 led_status = weather_data.getLEDValue()

 if led_status == 0:
 self.led.off()
 elif led_status == 1:
 self.led.on()
 else:
 self.led.blink()
```

Here's what's going on in our code:

- `weather_data = WeatherData(self.city)`: This creates an instance of the `WeatherData` class using the city that was specified when `WeatherDashboard` was instantiated. This object is used to fetch weather data for the city.
- `self.servo.value = weather_data.getServoValue()`: This calls the `getServoValue()` method on the `weather_data` object to get the value for the servo, based on the weather conditions. This value is then set as the new position for the servo motor.
- `led_status = weather_data.getLEDValue()`: This calls the `getLEDValue()` method on the `weather_data` object to get the value for the LED status, based on the weather conditions.
- The conditional block sets the LED's status based on the `led_status` value. If `led_status` is 0, it signifies calm weather conditions without any rainfall or thunderstorms, so the LED is turned off (`self.led.off()`). If `led_status` is 1, this indicates rain, resulting in the LED being turned on (`self.led.on()`). For an `led_status` value of 2, which is specific to thunderstorm conditions, the LED will blink (`self.led.blink()`), representing the intermittent nature of thunder and lightning. This value is returned by the `getLEDValue()` method in the `WeatherData` class when it detects thunderstorm conditions.

7. The final method in our `WeatherDashboard` class, `closeServo()`, simply closes the `Servo` instance that we created in our class. This is done to keep the servo motor from jittering:

```
def closeServo(self):
 self.servo.close()
```

8. Finally, save the code and call it `WeatherDashboard.py`.

We won't run our code just yet, as we need to add a new method to the file (but not the class). We will use this method to create a new `WeatherDashboard` object and update the position of the needle and the state of the LED.

## Adding the `updateDashboard()` function and main methods

The `update_dashboard()` function is a standalone function that is defined in the same Python file (`WeatherDashboard.py`) as the `WeatherDashboard` class, yet sits outside of it. It serves as an interface to encapsulate the process of updating the state of the weather dashboard. The reason for creating the `update_dashboard()` function is to help in overcoming issues related to servo motor jittering as it provides a controlled way to frequently create a fresh instance of the `WeatherDashboard` class, thereby *reopening* the `Servo` object connection each time an update to the dashboard is made.

Let's take a look:

1. In Thonny, open the `WeatherDashboard.py` file and add the following method to the bottom:

```
def update_dashboard(city, servo_pin, led_pin):
 weather_dashboard = WeatherDashboard(city,
 servo_pin,
 led_pin)

 weather_dashboard.update_status()
 sleep(2)
 weather_dashboard.closeServo()
```

Let's take a closer look:

- I. First, we define the `update_dashboard(city, servo_pin, led_pin)` function to take three arguments: the city for which the weather data is required, the PIN for the servo motor, and the PIN for the LED.
- II. Within this function, an instance of the `WeatherDashboard` class, named `weather_dashboard`, is created using the given city, servo PIN, and LED PIN as inputs.

- III. Next, the `update_status()` method of the `WeatherDashboard` instance is called. This method fetches the current weather data for the specified city, determines the servo and LED values based on this data, sets the servo's position, and sets the LED's state (off, on, or blinking) accordingly.
  - IV. The program then pauses for 2 seconds using the `sleep(2)` statement. This pause ensures that the servo has enough time to move to its new position and that the LED displays its new state before any further actions are taken.
  - V. Finally, the `closeServo()` method of the `WeatherDashboard` instance is called. This method is responsible for closing the connection to the servo motor, which is done to avoid potential issues with the servo, such as jittering.
2. Next, we will add code that only runs when the `WeatherDashboard.py` file is executed in Python. We will use this code to run our weather indicator continuously:

```
if __name__ == "__main__":
 city = 'Toronto'
 servo_pin = 14
 led_pin = 25

 while True:
 update_dashboard(city, servo_pin, led_pin)
 sleep(1800) # sleep for 30 minutes
```

Let's take a closer look:

- I. This block of code will only run when this script (`WeatherDashboard.py`) is executed directly based on `if __name__ == "__main__":`.
  - II. First, we set the `city`, `servo_pin`, and `led_pin` variables to `Toronto`, `14`, and `25`, respectively. Each reader may put in the city of their choice.
  - III. The `while True:` loop will continuously execute the code inside it, essentially making the script run forever, or until it is stopped manually.
  - IV. Inside the loop, the `update_dashboard()` function is called with the `city`, `servo_pin`, and `led_pin` values as arguments. This function updates the weather dashboard, effectively getting the new weather data and controlling the servo and LED accordingly.
  - V. After updating the dashboard, the script goes into sleep mode for 1,800 seconds, or 30 minutes (`sleep(1800)`). This means that the weather dashboard updates every 30 minutes.
  - VI. After the sleep period, the loop starts over, updating the dashboard again.
3. Resave the code and call it `WeatherData.py`.
  4. Run our code by clicking the green run button, hitting `F5` on your keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.

5. Once the script is up and running, we'll see that the position of the needle and the state of the LED reflect the current temperature, wind speed, and weather conditions accurately.

We may operate our weather indicator application independently, without the need for a keyboard, mouse, or monitor to be connected. *Figure 3.23* shows a practical example of how our application advises us on what to wear based on the weather in Toronto at the time the photo was taken, also indicating that there's no need for an umbrella:



Figure 3.23 – It looks like T-shirt weather in Toronto today

From the position of the needle on our weather indicator in *Figure 3.23*, we can conclude that it is T-shirt weather in Toronto today.

## Summary

In this chapter, we developed a weather indicator application, a tool that's designed to provide advice on clothing choices based on current weather conditions. This innovative project combined coding expertise with physical component design, manifesting not just on a computer screen, but in a real-world, tangible form. Central to this was the creation of a custom stand for our weather indicator, made from 3D-printed parts. This stand, which was designed to house the LED and the servo motor, contributed both practical functionality and aesthetic appeal to our project, successfully bridging the gap between the digital and physical worlds.

The lessons learned and the skills gained on this journey have wide-ranging applications. To illustrate, envision a new undertaking: a project that utilizes environmental sensors to track indoor climate conditions such as temperature, humidity, and air quality. The core principles we've mastered – from acquiring and interpreting data to interacting with hardware components – can be directly employed in such a scenario.

Also, it is easy to imagine adding more functionality to our weather indicator. For example, imagine replacing our LED with an RGB LED capable of a myriad of colors. We could use these different colors and various flashing patterns to indicate other types of weather conditions or warnings.

In this chapter, we added to our expertise in Python programming by focusing on interfacing with web services for data acquisition and analysis. In parallel, we ventured into the world of hardware interfaces, creating a tangible, physical component: our custom weather indicator stand.

With our newly acquired skills, we are poised to tackle a variety of complex problems. As demonstrated in our weather indicator project, being able to integrate web services, Python programming, and physical components can result in practical applications that significantly impact daily life. Whether advising on clothing choices based on weather conditions or envisaging new applications such as indoor climate monitoring, our abilities have proven versatile and invaluable.

In the next chapter, we'll construct an IoT display with a Raspberry Pi 7-inch touchscreen that presents real-time weather and traffic. This will involve exploring various screen types and creating a multifunctional dashboard, enhancing our skills in Raspberry Pi applications and IoT project development.

# 4

## Building an IoT Information Display

In this chapter, we will build an IoT information display using a Raspberry Pi-branded 7-inch touchscreen. We will use this information display to show real-time weather information and local traffic information.

We will start the chapter by exploring screens compatible for use with our Raspberry Pi. We will look at small **Organic Light-Emitting Diode (OLED)** screens, dot-matrix displays, seven-segment displays, and small LCD monitors for the Raspberry Pi.

For our project, we will build an IoT information display with Raspberry Pi's 7-inch touchscreen. This *dashboard* will not only show weather details but will also feature a map depicting local traffic conditions.

The insights gained in this chapter will equip us with a versatile toolkit, fostering creativity and innovation in future Raspberry Pi and IoT projects.

The following are the topics we will cover:

- Investigating displays compatible with our Raspberry Pi and exploring screen types
- Creating an IoT information display

Let's begin!

### Technical requirements

The following are the requirements for completing this chapter:

- Intermediate knowledge of Python programming
- A late-model Raspberry Pi, preferably a Raspberry Pi 5 with at least 4 GB of RAM
- A Raspberry Pi branded 7-inch touchscreen with a compatible case (optional)



The code for this chapter may be found here:

<https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter4>

## Investigating displays compatible with our Raspberry Pi and exploring screen types

Raspberry Pi offers the flexibility to interface with various external screens, catering to different applications and requirements. In *Figure 4.1*, we see examples of small screens that we may hook up to our Raspberry Pi:

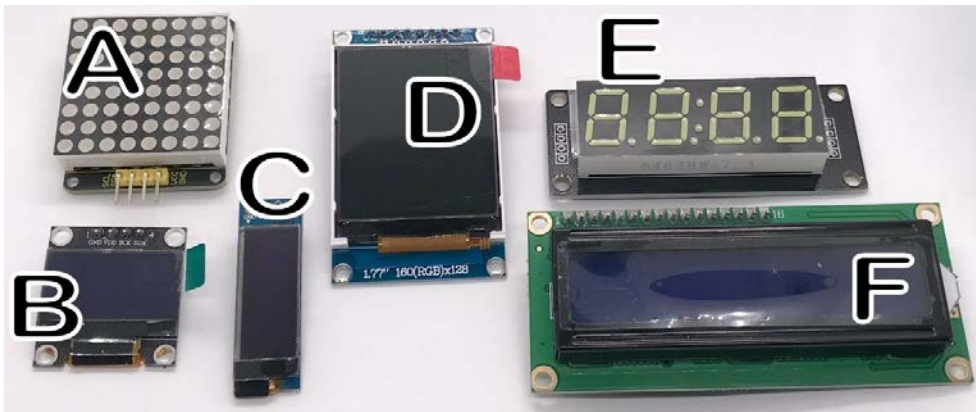


Figure 4.1 – Small displays that may be used with the Raspberry Pi

*Figure 4.1* shows different small screens tailored for various uses: OLED screens (*B*, *C*, *D*) for fine detail; a 16 x 2 LCD (*F*) for concise text; and dot-matrix (*A*) and segment-style displays (*E*) for clear numbers and characters. These options offer a versatile range of visual representations.

There are several larger screen options for the Raspberry Pi, including standard 4K computer monitors, as well as smaller monitor-style screens, as we see in *Figure 4.2*:

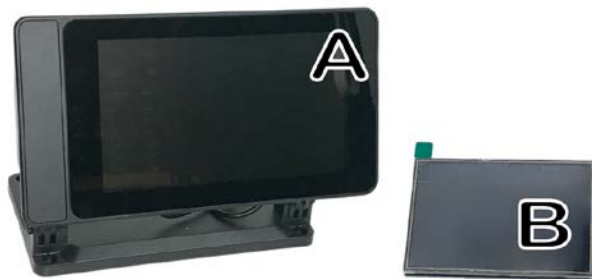


Figure 4.2 – Encased Raspberry Pi's 7-inch touchscreen (A) and 3.5-inch LCD screen (B)

The screen in *Figure 4.2 A* is the Raspberry Pi branded 7-inch touchscreen. It is shown here in a case specifically designed to house it. In *Figure 4.2 B*, we see a 3.5-inch LCD screen that connects to the GPIO port of the Raspberry Pi.

Let's take a closer look at each type of screen to understand their unique characteristics and applications.

Each screen shown in *Figure 4.1* and *Figure 4.2* offers unique advantages for specific applications, and understanding these can enhance our Raspberry Pi projects. The following are descriptions of each screen/display:

- **Small OLED screens:** OLED screens (see *B, C, and D* in *Figure 4.1*) are known for their crisp detail and energy efficiency. Small OLED screens (0. to 1.3 inches) are commonly used in projects that require minimal space and low power consumption.

OLED screens are used to display system status or in small gadgets such as fitness trackers. Ideal for showing limited information such as icons, temperature, or time, their compact size and energy efficiency make them a perfect choice for clear, concise displays without using much power or space.

- **16 x 2 LCD screens:** 16 x 2 LCD screens (see *F* in *Figure 4.1*) consist of 16 columns and 2 rows of characters. They offer a simple interface for text-based information and are ideal for applications where concise textual data needs to be displayed, such as in industrial settings where they may be used to present vital information such as error messages or production counts. Additionally, these screens are often found in 3D printers and are used to display control information to the user.
- **8 x 8 dot-matrix display:** An 8 x 8 dot-matrix display (see *A* in *Figure 4.1*), consisting of 64 individual LEDs arranged in an 8x8 grid. This grid enables the creation of simple images, characters, or animations by controlling each LED independently. Large versions of these displays are frequently found in public information boards for scrolling text messages. Of note, this type of display is featured on the Raspberry Pi Sense HAT, a device we used in *Chapters 1* and *2*.
- **7-segment display:** A 7-segment display (see *E* in *Figure 4.1*) consists of 7 or sometimes 8 individual segments (including a decimal point). This display can form the digit shapes for numbers and some letters by controlling which segments are illuminated. First invented in the early 20th century, 7-segment displays initially found use in devices such as calculators and digital clocks, and their application has since expanded with enhanced features such as improved brightness and energy efficiency, making them still relevant in modern digital clocks and industrial counters.
- **The Raspberry Pi 7-inch Touch Display** (see *A* in *Figure 4.2*) is a versatile accessory for Raspberry Pi projects. This touchscreen seamlessly connects through the **DSI** (short for **Digital Serial Interface**) port, requiring no extra drivers for touch functionality, and has an 800 x 480-pixel resolution.

- **3.5-inch** LCD screens (see *B* in *Figure 4.2*) connect to the Raspberry Pi via the GPIO port. With a resolution of 320 x 480 pixels, this compact display offers solutions for handheld devices and IoT projects. These screens also come in 5-inch sizes, often connecting to the HDMI and GPIO ports.

Now that we understand the various screens we may use with the Raspberry Pi, it is time to focus on this chapter's project: creating an IoT information display using the 7-inch Raspberry Pi touchscreen (see *A* in *Figure 4.2*).

## Creating an IoT information display

As outlined at the beginning of this chapter, our IoT information display will display real-time weather forecasts and a traffic map.

We will develop our IoT information display using the Raspberry Pi branded 7-inch touchscreen installed in a compatible case with a mouse and keyboard connected:



Figure 4.3 – Raspberry Pi development environment using a 7-inch touchscreen

The key benefit of the Raspberry Pi branded 7-inch screen is its connection to the **MIPI** (short for **Mobile Industry Processor Interface**) port on the Raspberry Pi 5 and not the GPIO or HDMI ports. This ensures seamless integration, eliminating the need to download and install additional drivers for touch functionality. We will not be using the touch feature for this project.

### Using our standard monitor

While the Raspberry Pi-branded 7-inch monitor offers impressive features, it is not essential for creating our IoT information display. We can utilize a standard monitor already connected to our Raspberry Pi. However, using a standard monitor may result in a large border around the display, as the positions of components in our code will be hardcoded.

Figure 4.4 outlines the software architecture we will follow to create our IoT information display:



Figure 4.4 – Software architecture for the IoT information display

We will use the `WeatherData` class to pull weather information from the `OpenWeatherMap` API.

The `Dashboard` class centralizes display information for our application, and we will leverage the **Kivy** library to create the GUI. By displaying our GUI in fullscreen mode, we create a kiosk-like effect.

Just as we use the `WeatherData` class to make calls to the `OpenWeatherMap` API, we use the `TrafficMap` class to call the `MapQuest Traffic` API for traffic data. Unlike the `WeatherData` class, the `TrafficMap` class generates an image file representing traffic conditions using GPS coordinates. This visual depiction of traffic not only adds value to the information being presented but also serves as the focal point of our IoT information display.

We will start our coding by setting up our development environment.

## Setting up our development environment

We will use a Python virtual environment for our development. As there are libraries that only work with the root installation of Python, we will use system packages in our Python virtual environment. To do so, we do the following:

1. On our Raspberry Pi 5, we open a Terminal application.

2. To store our project files, we create a new directory with the following command:

```
mkdir chapter4
```

3. We then navigate to the new directory with the following command:

```
cd chapter4
```

4. We require a subfolder for our project. We create this folder with the following command:

```
mkdir IoTInformationDisplay
```

5. We create a new Python virtual environment for our project with the following command:

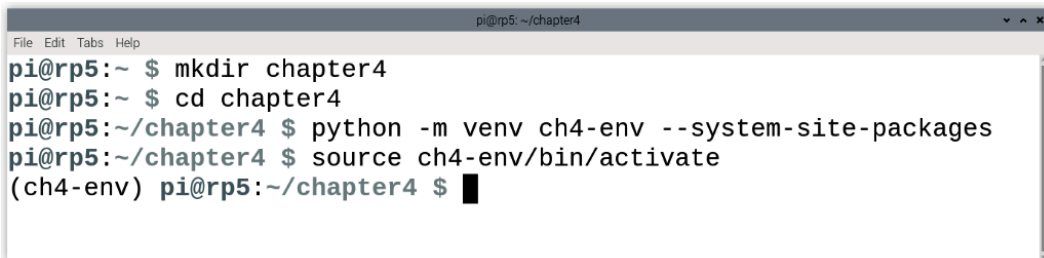
```
python -m venv ch4-env --system-site-packages
```

6. With this command, we create a new Python virtual environment called `ch4-env` and enable access to the system site packages. This allows the virtual environment to inherit packages from the global Python environment, which can be useful when certain libraries are installed system wide. Once the environment is set up, we can activate it and begin installing project-specific packages without affecting the global Python environment.

7. With our new Python virtual environment created, we source into it with the following command:

```
source ch4-env/bin/activate
```

8. Our Terminal application should now show that we are using the `ch4-env` Python virtual environment:



```
pi@rp5: ~/chapter4
File Edit Tabs Help
pi@rp5:~ $ mkdir chapter4
pi@rp5:~ $ cd chapter4
pi@rp5:~/chapter4 $ python -m venv ch4-env --system-site-packages
pi@rp5:~/chapter4 $ source ch4-env/bin/activate
(ch4-env) pi@rp5:~/chapter4 $
```

Figure 4.5 – Terminal using `dashboard-env` environment

9. We install the Python packages required for our code with the following command:

```
pip install requests kivy
```

10. `requests` is a Python library simplifying HTTP requests, ideal for web service interactions. Kivy enables the development of multitouch, cross-platform applications with a focus on rich user interfaces. With the Python packages installed, we may close the Terminal with the following command:

```
exit
```

11. We are now ready to load up Thonny. We do so by clicking on the **Menu** icon in the Raspberry Pi taskbar, navigating to the **Programming** category, and selecting **Thonny**.
12. By default, Thonny uses the Raspberry Pi's built-in version of Python. For our project, we will use the Python virtual environment we just created. To start, we need to view the project files by clicking on **View** and selecting **Files** if it is not already selected.
13. In the **Files** section, we locate the `ch4-env` directory.
14. We then right-click on the folder and select the **Activate virtual environment** option:

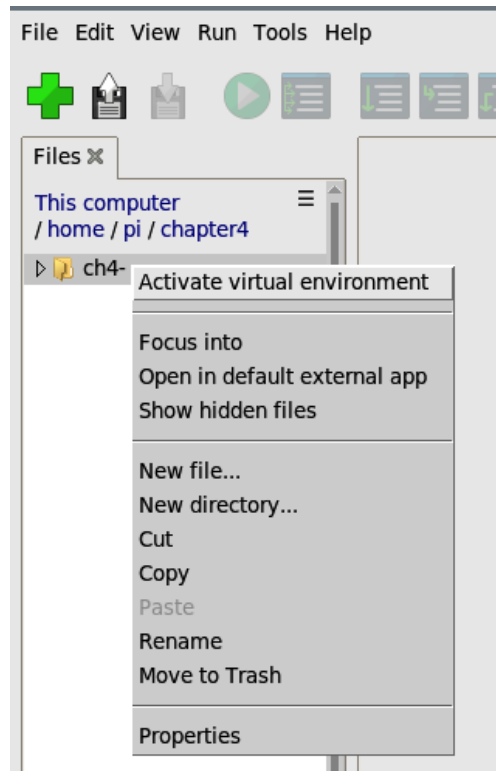


Figure 4.6 – Activating a Python virtual environment in Thonny

With our project folder created, our Python virtual environment set up and activated, and the packages we need for our project installed, we may now start writing code.

## Creating a WeatherData class

We use a `WeatherData` class to encapsulate calls to the OpenWeatherMap API. In addition to temperature and weather conditions, the weather portion of our IoT information display shows a weather conditions icon as well as an attire image. In *Figure 4.7*, we see images used for attire:



Figure 4.7 – Attire images used in our IoT information display

The attire images should look familiar to us as we used the same ones when we created our weather indicator in *Chapter 3*. With our IoT information display these graphics will be presented based on a factor determined by a calculation of temperature and wind speed.

In this section, we'll cover the `WeatherData` class code.

So, let's get started:

1. In Thonny, we create a new tab by selecting **File** and then **New** or by hitting *Ctrl + N* on the keyboard.
2. In our code, we start by entering our imports:

```
import requests
import json
```

3. We then define our class name, `WeatherData`, and our class variables:

```
class WeatherData:
 temperature = 0
 weather_conditions = ''
 wind_speed = 0
 city = ''
```

4. From here, we define our initialization method:

```
def __init__(self, city):
 self.city = city
 api_key = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
 base_url = "http://api.openweathermap.org/data/2.5/"
 weather"
```

```

 complete_url = f"{base_url}?q={self.city}&appid={api_
key}&units=metric"
 response = requests.get(complete_url)
 data = response.json()

 if data["cod"] != "404":
 main = data["main"]
 wind = data["wind"]
 weather = data["weather"]
 self.temperature = main["temp"]
 self.weather_conditions = weather[0]["main"]
 self.wind_speed = wind["speed"]
 self.icon = weather[0]["icon"]

```

5. We then define the `get_temperature()` and `get_conditions()` methods:

```

def get_conditions(self):
 return self.weather_conditions

def get_temperature(self):
 return str(int(self.temperature))

```

6. To accommodate the addition of a weather conditions icon, we add a method called `get_weather_conditions_icon()`:

```

def get_weather_conditions_icon(self):
 return f"http://openweathermap.org/img/wn/{self.icon}.
png"

```

This method constructs and returns a URL using an f-string, a feature in Python that allows for embedded expressions inside string literals. By appending the value of `self.icon` to the base URL from OpenWeatherMap, (`f"http://openweathermap.org/img/wn/{self.icon}.png"`), it forms a complete URL that leads to a PNG image representing the current weather conditions, such as sunny, cloudy, or rainy. This will allow us to embed an icon representing the current weather conditions into our IoT information display.

7. To determine the attire image to display, we require two additional methods. The first method uses wind speed and temperature to return a factor appropriately called `wind_temp_factor`:

```

def get_wind_temp_factor(self):
 if self.temperature < 0:
 temp_factor = 0
 elif self.temperature > 30:

```



```
 temp_factor = 30
 else:
 temp_factor = self.temperature
 wind_factor = self.wind_speed / 20
 wind_temp_factor = temp_factor - wind_factor
 return wind_temp_factor
```

This method calculates a wind temperature factor by first constraining the `self.temperature` value between 0 and 30 (assigning it to `temp_factor`), then dividing the wind speed by 20 (assigning it to `wind_factor`), and finally subtracting `wind_factor` from `temp_factor`, returning the resulting value as `wind_temp_factor`. These values are all arbitrary and may be changed.

8. The final method in our `WeatherData` class returns the image path for the attire image based on `wind_temp_factor`:

```
def get_attire_image(self):
 factor = self.get_wind_temp_factor()
 if factor < 8:
 return "images/gloves.png"
 elif factor < 18:
 return "images/long-shirt.png"
 elif factor < 25:
 return "images/short-shirt.png"
 else:
 return "images/shorts.png"
```

9. The final part of our code sits outside the `WeatherMap` class and allows us to test the class:

```
if __name__ == "__main__":
 weather = WeatherData('Toronto')
 print(weather.get_temperature())
 print(weather.get_attire_image())
 print(weather.get_conditions())
 print(weather.get_weather_conditions_icon())
```

This snippet creates an instance of the `WeatherData` class for Toronto and then prints various weather-related information to our console.

10. We save our code as `WeatherData.py` inside the `IoTInformationDisplay` project subfolder.

We are now prepared to construct a `TrafficMap` class, which will encapsulate the code used to create a map of local traffic.

## Creating a TrafficMap class

We utilize a `TrafficMap` class to interface with the MapQuest API, enabling the generation of a traffic map for our application. To make the connection to the MapQuest API, we must first create an account and generate an API key.

### *Generating an API key for our application*

MapQuest Developer offers various tools and services that enable developers to access maps, routing information, and more. For our project, we will need to obtain an API key from MapQuest to access their web services, particularly traffic map data. Here's how to set up a free account and get the API key:

1. We start by navigating to the MapQuest Developer site (<https://developer.mapquest.com/plans>) to access the developer's portal.
2. For our application, the **MapQuestGo** plan will be sufficient. This plan will give us 15,000 starter transactions. To create a plan, we click on the **Subscribe** button and follow the steps outlined.
3. Once our profile is created, we may generate a new API key by going to the following URL: <https://developer.mapquest.com/user/me/apps>.
4. We click on the **Create a New Key** button and enter an app name:

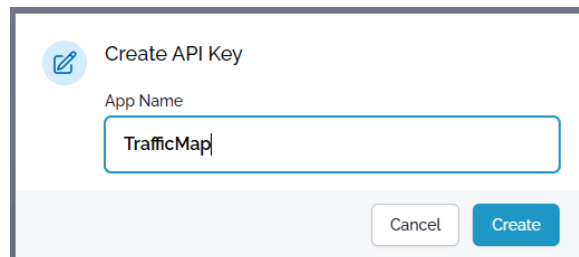


Figure 4.8 – Creating a new API key in MapQuest Developer

5. A new API key will be generated, and it can be viewed under the **Managed Keys** section:



Figure 4.9 – Viewing API keys in MapQuest Developer

We will be using this key to call the MapQuest Traffic API. It is a good idea to copy and paste the API key into a text document that can be accessed later. With the API key generated, we may now create our `TrafficMap` class.

### ***Coding the TrafficMap class***

We use Thonny to code our `TrafficMap` class:

1. We launch Thonny by clicking on the **Menu** icon in the Raspberry Pi taskbar. Then, we navigate to the **Programming** category and select **Thonny**.
2. Once inside Thonny, we activate the `ch4-env` virtual environment.
3. We create a new tab by selecting **File** and then **New** or by hitting `Ctrl + N` on the keyboard.
4. In our code, we start by entering our imports:

```
import requests
```

5. For our `TrafficMap` class, we only need to import the `requests` package. We define our class name, `TrafficMap`, and our initialization method:

```
class TrafficMap:
 def __init__(self, latitude, longitude, zoom):
 self.latitude = latitude
 self.longitude = longitude
 self.zoom = zoom
 self.size = "500,325"
 self.api_key = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
```

Let's look at what we just added:

- `self.latitude`: Specifies the latitude of the location on the map, used to center the map at that latitude.
- `self.longitude`: Specifies the longitude of the location on the map, used to center the map at that longitude.
- `self.zoom`: Sets the zoom level for the map, controlling how much of the map is visible (for example, higher values might show a closer, more detailed view).
- `self.size`: Sets a fixed size for the map, defining the width and height in pixels. The size is set as a string with dimensions of `"500,325"`.
- `self.api_key`: Stores a hardcoded API key, which is required to authenticate requests to the API.

6. At the heart of our `TrafficMap` class is the `get_traffic_map()` method. We use this method to make the call to the MapQuest Traffic web service and use the response to create our traffic map:

```
def get_traffic_map(self):
 base_url = "http://www.mapquestapi.com\
 /staticmap/v5/map"
 params = {
 'key': self.api_key,
 'center': f"{self.latitude},\
 {self.longitude}",
 'zoom': self.zoom,
 'size': self.size,
 'traffic': 'flow|cons|inc'
 }
 response = requests.get(base_url,
 params=params)
 if response.status_code == 200:
 with open('images/traffic_map.png', 'wb') as f:
 f.write(response.content)
 return "images/traffic_map.png"
 else:
 return "images/error.png"
```

Let's look at what we just added:

- `base_url`: The URL endpoint for the MapQuest static map API.
- `params`: We use this dictionary containing the necessary parameters for the API request:
  - `self.api_key`: Specifies the API key for authentication with the MapQuest static map API
  - `center - f"{self.latitude}, {self.longitude}"`: Defines the center of the map using the latitude and longitude attributes of the object
  - `self.zoom`: Specifies the zoom level for the map, controlling the scale or detail visible on the map
  - `self.size`: Sets the size of the map, using the previously defined size attribute of the object, likely defining the width and height in pixels
  - `traffic: 'flow|cons|inc'`: Specifies the traffic information to be included on the map, representing different types of traffic data such as flow, congestion, and incidents

- `response = requests.get(base_url, params=params)`: Sends a GET request to the URL stored in `base_url` with the parameters defined in `params` and stores the response in the `response` variable.
- If the request is successful (`response.status_code == 200`), the following happens:
  - An image file (`traffic_map.png`) is created or overwritten in the `images` directory.
  - The content of the response (which is image data) is written to the file.
  - The method returns the path to the saved image: `"images/traffic_map.png"`
- If the request is not successful, the method returns the path to a predefined `error.png` image in the `images` directory. This image contains an `Error loading traffic map` message.

**Important note**

The main takeaway from this method is that, unlike many API responses that require a JSON library to parse, the response from the GET request to the MapQuest Traffic API directly contains image data for the traffic map, so it can be saved as an image file without needing to parse JSON.

7. We save our code as `TrafficMap.py` inside the `IoTWeatherDisplay` project subfolder.

With the creation of the `WeatherData` and `TrafficMap` classes complete, we can now proceed to write a `Dashboard` class that will handle the display of information retrieved from these web services.

## Adding Dashboard and MyApp classes

For the `Dashboard` class, we use the Kivy library. Kivy is an open source Python framework designed for developing multitouch applications that can run on various platforms, including Windows, macOS, Linux, iOS, and Android.

In the same Python file where we define the `Dashboard` class, we will add a Kivy App class we call `MyApp`. The `MyApp` class in this code is a subclass of Kivy's `App` class, defining the main entry point for the application by creating an instance of the `Dashboard` class in its `build()` method.

To do this, we do the following:

1. We launch Thonny by clicking on the **Menu** icon in the Raspberry Pi taskbar. Then, we navigate to the **Programming** category and select **Thonny**.
2. Once inside Thonny, we activate the `ch4-env` virtual environment.
3. We create a new tab by selecting **File** and then **New** or by hitting `Ctrl + N` on the keyboard.

4. We start our code by configuring Kivy to run in fullscreen and borderless mode:

```
from kivy.config import Config
Config.set('graphics', 'fullscreen', 'auto')
Config.set('graphics', 'borderless', '1')
```

#### Why do we configure our Kivy application before adding other imports?

We configure our application before importing other packages in Kivy to ensure that the settings are applied at the beginning of the application's life cycle. If configuration were done after importing the packages, some settings might not be applied or could lead to unexpected behavior, as Kivy components might be initialized with the default configurations before the custom settings are set.

5. After configuration, we import the other packages we require for our application:

```
from kivy.app import App
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.label import Label
from kivy.uix.image import Image, AsyncImage
from kivy.clock import Clock
from WeatherData import WeatherData
from TrafficMap import TrafficMap
```

Let's look at what we just added:

- `from kivy.app import App`: Imports the main application class from the Kivy framework, which is necessary for building and running the app.
- `from kivy.uix.floatlayout import FloatLayout`: Imports the `FloatLayout` class from Kivy, which allows for positioning and resizing of widgets in a free-form manner. We do this so that we may position our GUI components in exact locations on our screen.
- `from kivy.uix.label import Label`: Imports the `Label` class from Kivy, which is used to display text on the screen.
- `from kivy.uix.image import Image, AsyncImage`: Imports the `Image` and `AsyncImage` classes from Kivy, allowing for the display of both static and asynchronous images within the application.
- `from kivy.clock import Clock`: Imports the `Clock` class from Kivy, enabling the scheduling of periodic functions within the app. We will be using this class to update our dashboard every 30 minutes.
- `from WeatherData import WeatherData`: Imports our custom `WeatherData` class, which is used to handle weather-related information.
- `from TrafficMap import TrafficMap`: Imports the custom `TrafficMap` class, used to manage traffic map data and images.

6. After our imports, we define our Dashboard class and its initialization method:

```
class Dashboard(FloatLayout):
 def __init__(self,
 city,
 latitude,
 longitude,
 zoom):
 super(Dashboard, self).__init__()
 self.city = city
 self.traffic_map = TrafficMap(latitude,
 longitude,
 zoom)

 self.init_widgets()
 Clock.schedule_interval(self.update_status, 1800)
 self.update_status(0)
```

Let's look at what we just added:

- I. Our Dashboard class inherits from Kivy's FloatLayout class, allowing positioning and layout features.
  - II. We initialize our class with specific parameters (city, latitude, longitude, and zoom).
  - III. We call the `__init__()` method of the FloatLayout superclass through `super(Dashboard, self).__init__()`, ensuring proper initialization and inheriting behaviors from the parent class.
  - IV. We then set up the traffic map with the given latitude, longitude, and zoom parameters.
  - V. We utilize Kivy's Clock class to schedule regular updates every 1800 seconds (30 minutes).
  - VI. We call the `self.init_widgets()` method to create and position widgets inside the layout.
  - VII. We then call `self.update_status(0)` once during initialization to set the initial state of the dashboard. If this method was not called, the initial state of the dashboard might not be set, possibly leading to a delay in displaying the initial content until the scheduled update occurs, which is set to happen every 1800 seconds or 30 minutes.
7. With the initialization of the class in place, we initialize the state of the widgets. We start with the Temperature, Conditions, and Attire widgets:

```
def init_widgets(self):
 self.add_widget(Label(text="Temperature",
 pos=(-275, 175),
```

```

 color=(1, 1, 1, 1),
 font_size=22,
 font_name='fonts/
 ArialBlack.ttf'))
self.add_widget(Label(
 text="Conditions",
 pos=(-275, 60),
 color=(1, 1, 1, 1),
 font_size=22,
 font_name='fonts/
 ArialBlack.ttf'))
self.add_widget(Label(text="Attire",
 pos=(-280, -80),
 color=(1, 1, 1, 1),
 font_size=22,
 font_name='fonts/
 ArialBlack.ttf'))
self.add_widget(Image(source='images/box.png',
 pos=(-275, 145)))
self.add_widget(Image(source='images/box.png',
 pos=(-275, 10)))
self.add_widget(Image(source='images/box.png',
 pos=(-275, -127)))

```

8. We then add a city field, weather conditions, and a traffic map:

```

self.city_label = Label(text=self.city,
 pos=(250, 185),
 color=(1, 1, 1, 1),
 font_size=30,
 font_name='fonts/
 ArialBlack.ttf')
self.add_widget(self.city_label)
self.temperature_label = Label(pos=(-275,
 125),
 color=(1, 1, 1, 1),
 font_size=40,
 font_name='fonts/
 ArialBlack.ttf')
self.add_widget(self.temperature_label)
self.conditions_image = AsyncImage(pos=(-278,
 20))
self.add_widget(self.conditions_image)
self.weather_conditions_label = Label(

```



```

pos=(-280,
 -25),
color=(1, 1,
 1, 1),
font_size=20,

font_name='fonts/
 ArialBlack.ttf')
self.add_widget(self.weather_conditions_label)
self.traffic_map_image = AsyncImage(pos=(120,
 -30))

self.add_widget(self.traffic_map_image)
self.attire_image = Image(pos=(-280, -140))
self.add_widget(self.attire_image)

```

The `init_widgets()` method is responsible for initializing and adding both static and dynamic widgets to the Dashboard layout, which is a subclass of Kivy's `FloatLayout` class.

#### Important note

Static widgets are components that remain the same throughout the life cycle of the application and don't need to be updated dynamically. This includes adding labels displaying Temperature, Conditions, and Attire text at specified positions with specific colors, font sizes, and fonts. It also includes three image widgets displaying an image located at 'images/box.png', placed at different positions on the layout.

**Dynamic widgets** are components that may need to be updated as the application runs, such as to display new data. This includes creating and adding a label that displays the city name (which could be made dynamic if you want to change the city later), a label for displaying the current temperature, an `AsyncImage` widget to display a weather conditions icon, a label for displaying weather conditions text, another `AsyncImage` widget to display a traffic map, and an image widget to display appropriate attire for the current weather conditions.

The `init_widgets()` method leverages the `add_widget()` method provided by Kivy's `FloatLayout` class to add each widget to the layout. The dynamic widgets are also stored as attributes of the Dashboard class so that they can be easily accessed and updated later.

9. We use the `update_status()` method of the Dashboard class to provide new values for our screen every 30 minutes (as set in the initialization method):

```

def update_status(self, *args):
 weather_data = WeatherData(self.city)
 self.traffic_map_image.source =
 self.traffic_map.
 get_traffic_map()

```

```

self.attire_image.source = weather_data.
 get_attire_image()
self.temperature_label.text = weather_data.
 get_temperature()
 + "\u00B0C"
self.weather_conditions_label.text =
 weather_data.get_conditions()
self.conditions_image.source =
 weather_data.
 get_weather_conditions_icon()

```

The `update_status()` method within the `Dashboard` class plays a major role in updating the dynamic widgets to reflect current data. Initially, it creates an instance of the `WeatherData` class using the current city. The method then proceeds to update the source of the `traffic_map_image` widget with a new traffic map image reflecting current conditions, obtained from the `get_traffic_map()` method of the `TrafficMap` class. It also changes the source of the `attire_image` widget to represent suitable clothing for the current weather, utilizing the `get_attire_image()` method of the `WeatherData` class.

The temperature label's text is updated with the current temperature from the `WeatherData` class, appending the degree symbol and the letter *C* to represent Celsius. The text of the `weather_conditions_label` widget is modified to provide a description of the present weather conditions, also obtained from the `WeatherData` class.

Finally, the source of the `conditions_image` widget is updated with an icon that symbolizes the current weather conditions, using the `get_weather_conditions_icon()` method from the `WeatherData` class.

10. Having completed our `Dashboard` class, we'll next develop the `MyApp` class. This class will serve as the entry point for our Kivy application, handling both the initialization and management of the dashboard interface. We'll create this class immediately following the `Dashboard` class in the same file:

```

class MyApp(App):
 def build(self):
 city = 'Toronto'
 latitude = 43.6426
 longitude = -79.3871
 zoom = 12
 return Dashboard(city,
 latitude,
 longitude,
 zoom)

if __name__ == '__main__':
 MyApp().run()

```

The `MyApp` class is inherited from Kivy's `App` class and is responsible for initializing and running the main application. Within the `build()` method of `MyApp`, the city of `Toronto`, along with its corresponding latitude and longitude coordinates (`43.6426` and `-79.3871`), and zoom level are specified. These GPS coordinates point to the location in Toronto (downtown Toronto near the CN Tower).

A `Dashboard` object is then created and returned with these parameters. The `if __name__ == '__main__':` line ensures that the code following it is only executed if the script is being run directly (not imported as a module in another script). When run directly, it creates an instance of the `MyApp` class and calls its `run()` method, starting the Kivy application and displaying the initialized dashboard with specified details related to Toronto.

11. We save our code as `Dashboard.py` inside the `IoTWeatherDisplay` project subfolder.

With all this code written, it is time to run it on our 7-inch screen.

## Running the IoT information display application

To execute our IoT information display through the `Dashboard.py` script, we use Thonny. We can either click on the green run button and hit `F5` on the keyboard or click on the **Run** menu option at the top and then **Run current script**.

With all the aforementioned code entered without errors (or cloned from the GitHub repository at <https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter4>), we should see our IoT information display run full screen on our 7-inch monitor:

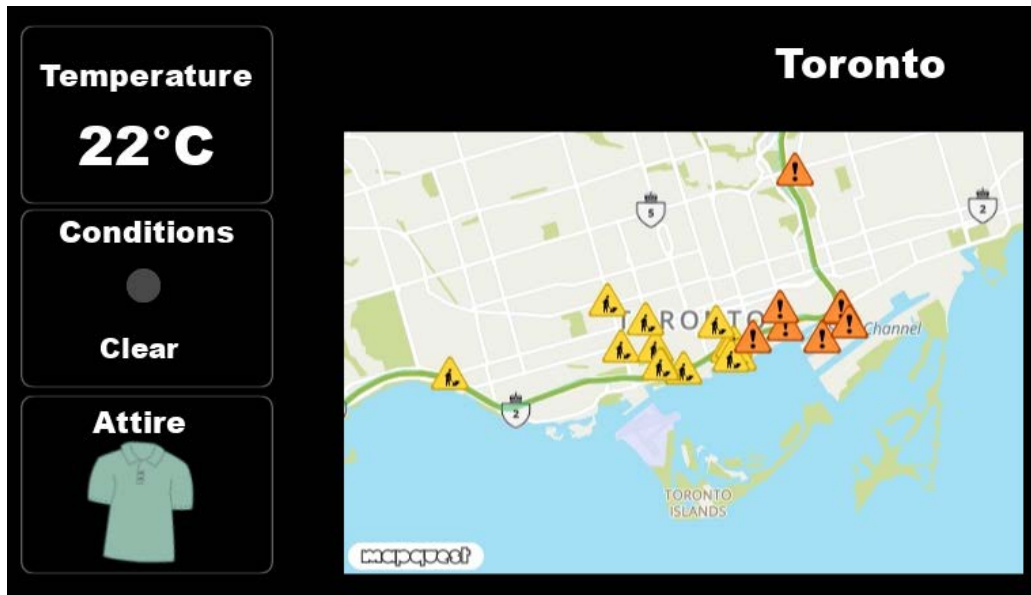


Figure 4.10 – IoT information display showing weather and traffic for Toronto, Canada

---

As we can see, it is a relatively warm day in Toronto. In terms of traffic flow, there is construction and other areas to avoid. The weather is clear, and it is suggested we wear short-sleeved shirts today. If we were to run this on a standard computer monitor, we may see a large black border around our display.

In a specific application such as creating an IoT information display, the mouse and keyboard can be removed from the Raspberry Pi, allowing the device to be used as a kiosk with a 7-inch touchscreen.

## Summary

In this chapter, we explored various small screens compatible with the Raspberry Pi. We then used Raspberry Pi's 7-inch touchscreen to create a versatile IoT information display that included weather forecasts and traffic maps for a specific city.

By utilizing and optimizing web service data, we demonstrated the flexibility of Raspberry Pi in handling complex information display tasks. The principles and code developed in this chapter can be adapted for other uses, such as home automation dashboards, public information kiosks, or industrial monitoring systems for manufacturing processes.

We have focused primarily on displaying information – in particular, weather information via web services – over the last few chapters. In the next chapter, we will start exploring sensory information with our Raspberry Pi as we work toward building an IoT security application.



# Part 2:

## Building an IoT Home Security Dashboard

In *Part 2*, we explore the general-purpose input/output (GPIO) ports on the Raspberry Pi and Raspberry Pi Pico, build an IoT alarm module with a Raspberry Pi Pico W and a PIR motion sensor, create an IoT button using the M5Stack ATOM Matrix and Raspberry Pi Pico W, and develop an IoT alarm dashboard on a Raspberry Pi 5 with a 7-inch touchscreen for control and monitoring.

This part has the following chapters:

- *Chapter 5, Exploring the GPIO*
- *Chapter 6, Building an IoT Alarm Module*
- *Chapter 7, Building an IoT Button*
- *Chapter 8, Creating an IoT Alarm Dashboard*



# 5

## Exploring the GPIO

Throughout the first four chapters of the book, we have touched on the **General-Purpose Input/Output (GPIO)** port on the Raspberry Pi. In *Chapter 3*, we used it extensively in the construction of our weather indicator. In this chapter, we will dive deeper into the functionality and applications of the GPIO port on the Raspberry Pi as we start to build our IoT home security application. We will also explore the GPIO port on the Raspberry Pi Pico, the microcontroller cousin of the Raspberry Pi.

In the hands-on tutorial section, we will construct a basic alarm system using a PIR motion sensor to detect human presence. This system will integrate a pushbutton for activation control and a buzzer as an alert mechanism. Through this practical exercise, we will demonstrate how the Raspberry Pi can interface with various components to create functional real-world applications.

In this chapter, we will cover the following:

- Introducing the GPIO on the Raspberry Pi and Raspberry Pi Pico
- Understanding sensors, actuators, and indicators
- Building a simple alarm system

Let's begin!

### Technical requirements

The following are the requirements for completing this chapter:

- Intermediate knowledge of Python programming
- A late model Raspberry Pi, preferably a Raspberry Pi 5 with at least 4 GB of RAM
- A PIR sensor
- An SFM-27 buzzer
- A pushbutton such as an arcade-style button



The code for this chapter may be found here:

<https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter5>

## Introducing the GPIO on Raspberry Pi

The GPIO port is a versatile set of 40 pins on the Raspberry Pi and Pico, enabling interaction with the external world. These pins can be configured as input or output and can connect to sensors, LEDs, and more. Included are ports to provide power, several ground connections, and GPIOs designated for specific protocols such as I2C, UART, SPI, and PCM. In the following table, we can see how we may match up pins with specific communication protocols:

| Model             | GPIO Pins | Power Ports         | I2C Pins             | UART Pins          | SPI Pins                                | PCM Pins                |
|-------------------|-----------|---------------------|----------------------|--------------------|-----------------------------------------|-------------------------|
| Raspberry Pi      | 40        | Two 5V and two 3.3V | GPIO 2 and 3         | GPIO 14 and 15     | GPIO 7, 8, 9, 10, 11                    | GPIO 12, 35, 38, and 40 |
| Raspberry Pi Pico | 40        | 3.3V, VBUS          | GP4 (SDA), GP5 (SCL) | GP0 (TX), GP1 (RX) | GP2 (CS), GP3 (SCK), GP4 (TX), GP5 (RX) | Not specified           |

Figure 5.1 – GPIO pins and communication protocols

## Exploring the Raspberry Pi GPIO pinout diagram

In *Figure 5.2*, we can see a pinout diagram of the GPIO ports on the Raspberry Pi and Raspberry Pi Pico. Outlined are the GPIO pin numbers as well as the pins that may be configured for special operations.

### Important note

Common to many purchases of a Raspberry Pi and Raspberry Pi Pico is a reference sheet with the GPIO pinout diagram. This tool is extremely helpful for development, as it acts as a guide to the GPIO (Raspberry Pi) and GP (Pico) pin numbers required for device connections to the Pi and Pico, respectively.

We may find various versions of these diagrams on the internet. For those of us interested, there is an interactive version of the Raspberry Pi GPIO pinout at <https://pinout.xyz>.

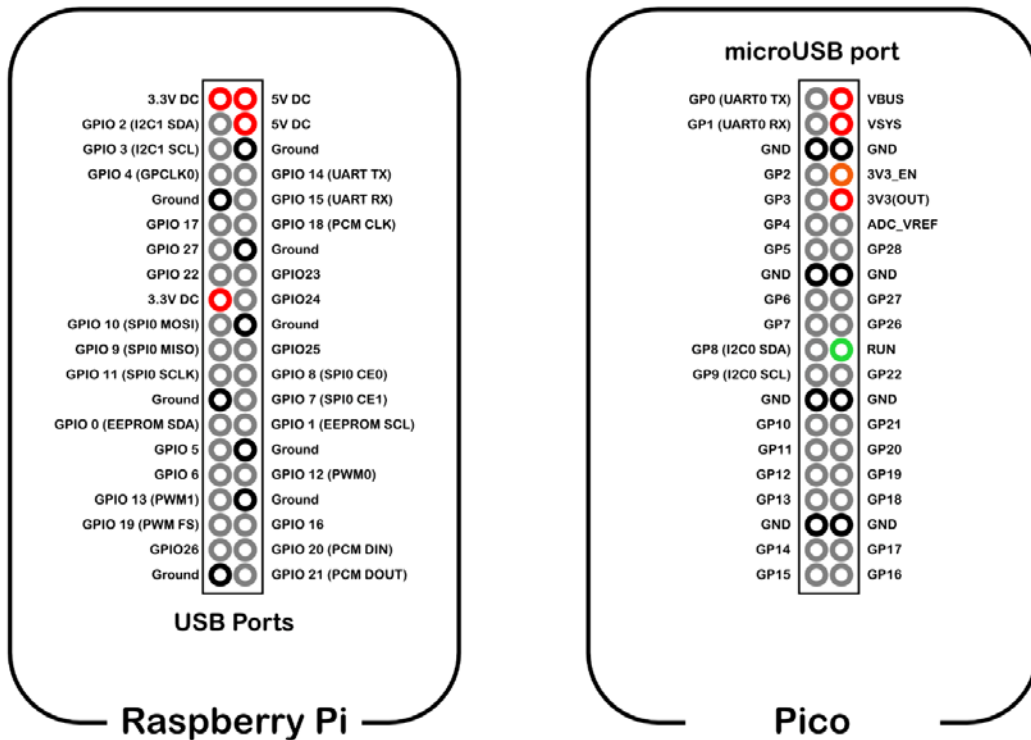


Figure 5.2 – GPIO pinout diagram

The Raspberry Pi's GPIO port includes pins GPIO 0 (EEPROM SDA) and GPIO 1 (EEPROM SCL), which enable automatic communication with attached HATs. Additionally, there are pins designated for various communication protocols.

To better understand the communication capabilities of the Raspberry Pi and Pico, let's examine the specific communication protocols that we may configure our devices to work with.

## Understanding GPIO pin communication protocols

The Raspberry Pi's and Pico's GPIO pins enable several key communication protocols such as I2C, SPI, UART, and PCM. These allow the Raspberry Pi and Pico to interact with various devices and sensors. Each protocol has its unique applications, from interfacing with sensors to digital audio transmission. In the following sections, we'll provide an overview of these communication methods that may be configured.

## I2C

I2C is a serial communication protocol developed by Philips Semiconductor, now NXP Semiconductors. I2C allows multiple devices to communicate with each other using a two-wire interface.

Here's a breakdown of the connections:

- **Serial Data Line (SDA):** This is the data line.
- **Serial Clock Line (SCL):** This is the clock line that synchronizes data transfer over the I2C bus.

The key features of the I2C protocol include the following:

- **Multi-master and multi-slave:** Multiple devices can be connected to the bus, with the capability for more than one master.
- **Address-based communication:** Each device on the bus has a unique address, allowing targeted communication.
- **Simple hardware connection:** Only two pull-up resistors are required for the SDA and SCL lines when setting up an I2C connection.
- **Speed variants:** I2C supports different speed modes, including standard (up to 100 Kbps), fast (up to 400 Kbps), high speed (up to 3.4 Mbps), and more.

On the Raspberry Pi and Pico, the I2C protocol is particularly valuable for connecting peripherals such as PIR sensors, mini-OLED screens, and other components using just two wires.

## SPI

The **Serial Peripheral Interface (SPI)** is a synchronous serial communication protocol used for short-distance communication, primarily in embedded systems, to connect microcontrollers and peripheral devices such as sensors, SD cards, and LCDs.

Here's a breakdown of the connections:

- **SDO—Serial Data Out:** This line is used for sending data from the controller to the peripherals.
- **SDI—Serial Data In:** This line allows peripherals to send data back to the controller.
- **SCLK—Serial Clock:** Like the SCL line in I2C, this provides the clock signal, synchronizing data transfer between devices.
- **CS0—Chip Select 0:** This line is crucial for selecting specific devices when there are multiple peripherals. By toggling CS0, the controller can determine which device it's communicating with.
- **CS1—Chip Select 1:** Like CS0, CS1 provides an additional selection line, enabling the addressing of more devices on a single SPI bus.

---

The following are the features and advantages of SPI:

- **Full-duplex communication:** SPI supports simultaneous bidirectional communication, allowing data to be sent and received at the same time. This means that during an SPI communication, data can flow from the master to the slave and from the slave to the master concurrently. This is unlike I2C, which operates in a half-duplex mode (can either send or receive data at any given moment, but not at the same time).
- **Speed:** SPI often outpaces I2C in terms of data transfer rates, making it suitable for applications that demand higher-speed communications.
- **Direct hardware control:** Without a specific addressing scheme as in I2C, SPI offers more straightforward device control via the CE pins.

SPI communication is useful when connecting components such as SD cards, displays, **ADCs** (short for **analog-to-digital converters**), and more. An example is an application where we connect an SD card via SPI for data logging.

## UART

UART, which stands for **Universal Asynchronous Receiver–Transmitter**, is a prevalent communication protocol in the electronics domain, especially known for its simplicity and effectiveness in point-to-point communications between devices. UART stands out with its full-duplex communication capability. This allows two devices to exchange data concurrently.

Here’s a breakdown of the connections:

- **TX—Transmit:** This connection is used to send data out to another device.
- **RX—Receive:** Conversely, this connection is used to receive data from another device.

These connections allow for simultaneous bidirectional communication; while the Raspberry Pi and Pico transmit data via their TX pins, they can also receive data through their RX pins.

Some features of UART compared to other communication methods such as I2C and SPI include:

- **Peer-to-peer communication:** Unlike I2C and SPI, which have defined master-slave relationships, UART devices communicate as equals, with no designated master or slave.
- **Asynchronous mechanism:** Unlike SPI, UART communication doesn’t rely on a shared clock signal. Instead, both devices must agree on a baud rate before initiating communication.

Given its simplicity, UART is commonly used for serial console access and interfacing with peripherals that demand a straightforward communication path. One common use of UART and the Raspberry Pi is the connection of flight controllers for drones.

## PCM

We can set GPIO pins 12, 35, 38, and 40 on the Raspberry Pi for **PCM (Pulse-Code Modulation)**, which digitally represents analog signals by sampling their magnitude at regular intervals and quantizing it into a digital code. These pins are specifically designated for PCM communication on the device:

- **GPIO 12 (PCM\_CLK):** This is the clock pin, ensuring synchronization during data transfer.
- **GPIO 35 (PCM\_FS):** This is frame sync. It helps in defining the start and end of the data frame.
- **GPIO 38 (PCM\_DIN):** This is the data input. This is where the Raspberry Pi receives PCM data from external devices.
- **GPIO 40 (PCM\_DOUT):** This is the data output. The Raspberry Pi uses this pin to send PCM data to other devices.

Some notable aspects of PCM include the following:

- **Digital representation:** PCM converts analog signals into a digital format, making it ideal for preserving the nuance of the original signal in a format that's resistant to noise and interference.
- **Common in audio:** Many audio formats, such as WAV, use PCM to represent sound digitally, ensuring high fidelity.
- **Flexibility:** PCM can be used for a range of applications beyond audio, including telecommunications and data storage.

On the Raspberry Pi, PCM's utility is evident in various tasks, from digital audio playback and recording to interfacing with devices that require precise analog-to-digital representations. A notable application of PCM with the Raspberry Pi is in the field of digital audio systems and sound interfaces.

The following is a table summarizing the communication protocols and the Raspberry Pi and Raspberry Pi Pico:

| Protocol           | Key Features                                                                                                           | Application                                                                                                 | Connections                                                                                                | Range                                                                                           |
|--------------------|------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| I2C                | Multi-master and multi-slave, address-based communication, simple hardware connection, supports different speed modes. | Connecting peripherals like PIR sensors, mini-OLED screens                                                  | Serial Data Line (SDA), Serial Clock Line (SLC)                                                            | Short (a few meters within the same board or device, extendable with care)                      |
| SPI                | Full-Duplex Communication, higher speed compared to I2C, direct hardware control.                                      | Connecting components like SD cards, displays, ADCs using SPI for data logging.                             | SDO (Serial Data Out), SDI (Serial Data In), SCLK (Serial Clock), CS0 (Chip Select 0), CS1 (Chip Select 1) | Short to moderate (depends on clock speed and signal integrity, extendable with careful design) |
| UART               | Full-Duplex Communication, Peer-to-Peer Communication, Asynchronous Mechanism.                                         | Used for serial console access and interfacing with peripherals like flight controllers for drones.         | TX - Transmit, RX - Receive                                                                                | Moderate (several meters, extendable with RS-232 or RS-485 for longer distances)                |
| PCM (Raspberry Pi) | Digital Representation of analog signals, common in audio, flexibility for various applications.                       | Digital audio playback and recording, interfacing with devices requiring analog-to-digital representations. | GPIO 12 (PCM_CLK), GPIO 35 (PCM_FS), GPIO 38 (PCM_DIN), GPIO 40 (PCM_DOUT)                                 | Varies (depends on specific application and implementation)                                     |

Figure 5.3 – Communication protocols summarized

Now that we understand the communication protocols we may use with the Raspberry Pi and Raspberry Pi Pico, let's look at sensors, actuators, and indicators and how we may hook them up to our Raspberry Pi and Raspberry Pi Pico. This is the main purpose of the GPIO port.

## Understanding sensors, actuators, and indicators

The GPIO functionality of the Raspberry Pi and Pico offers a foundation for connecting various sensors and driving indicators like LEDs and controlling actuators like servo motors. By integrating these devices, our devices can both collect data and execute responsive actions based on this information.

For instance, a PIR sensor (*A* in *Figure 5.4*) can detect motion in a room, prompting an LED or triggering an alarm. Using temperature and humidity sensors, such as DHT11 (*B* in *Figure 5.4*), we can assess environmental conditions and, in response, engage a fan or a heating element.

With distance sensors (C in *Figure 5.4*), we can measure the closeness of objects and instruct a servo motor to halt a robot, preventing collisions:

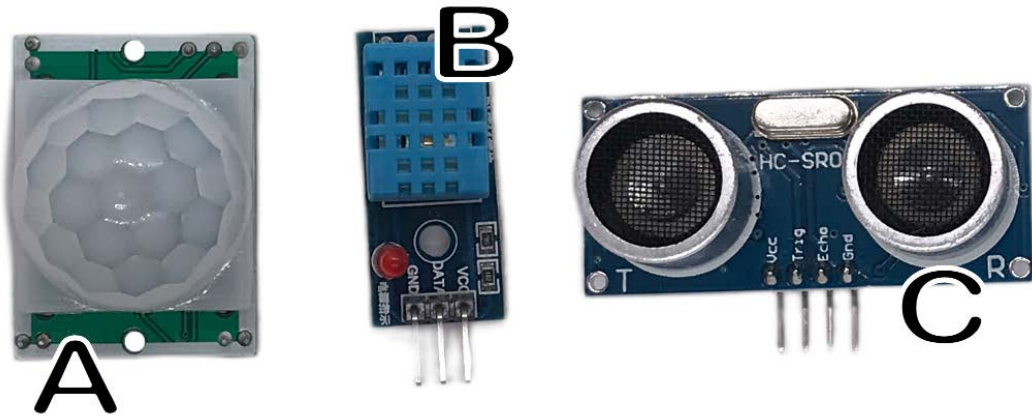


Figure 5.4 – Sensors for the Raspberry Pi

Numerous devices can be integrated with our Raspberry Pi and Pico. Robotics alone offers line tracking sensors for directing robots along set paths and **Time of Flight (TOF)** sensors, which are compact and precise distance sensors. Beyond robotics, there are sensors for soil moisture, rain detection, light, and temperature/humidity, creating the potential for a self-regulating greenhouse.

In the next section, we'll explore connecting and reading data from a PIR sensor (A in *Figure 5.4*) and a Raspberry Pi using Python and the GPIO Zero library. Taking what we learn, we will use our knowledge to create a basic alarm system with our Raspberry Pi.

We will start by setting up our development environment before we write code to interact with the PIR sensor.

## Setting up our development environment

We will use a Python virtual environment for our development. As there are libraries that only work with the root installation of Python, we will use system packages in our Python virtual environment. To do so, we do the following:

1. On our Raspberry Pi 5, we open a Terminal.
2. To store our project files, we create a new directory with the following:

```
mkdir chapter5
```

3. We then navigate to the new directory with the following:

```
cd chapter5
```

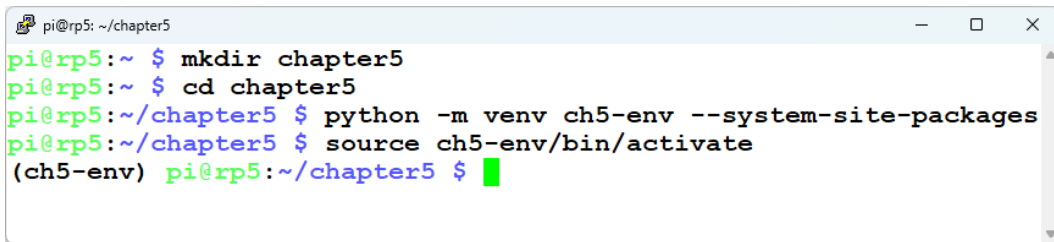
4. We create a new Python virtual environment for our project with the following:

```
python -m venv ch5-env --system-site-packages
```

5. With this command, we create a new Python virtual environment called `ch5-env` and enable access to the system site packages. This allows the virtual environment to inherit packages from the global Python environment, which can be useful when certain libraries are installed system wide. Once the environment is set up, we can activate it and begin installing project-specific packages without affecting the global Python environment.
6. With our new Python virtual environment created, we source into it with the following command:

```
source ch5-env/bin/activate
```

7. Our Terminal should now show that we are using the `ch5-env` Python virtual environment:



```
pi@rp5: ~/chapter5
pi@rp5:~$ mkdir chapter5
pi@rp5:~$ cd chapter5
pi@rp5:~/chapter5$ python -m venv ch5-env --system-site-packages
pi@rp5:~/chapter5$ source ch5-env/bin/activate
(ch5-env) pi@rp5:~/chapter5$
```

Figure 5.5 – Terminal using dashboard-env environment

8. We close the Terminal with the following command:

```
exit
```

9. We are now ready to load up Thonny. We do so by clicking on the **Menu** icon in the Raspberry Pi taskbar, navigating to the **Programming** category, and selecting **Thonny**.
10. By default, Thonny uses the Raspberry Pi's built-in version of Python. For our project, we will use the Python virtual environment we just created. To start we need to view the project files by clicking on **View** and selecting **Files** if it is not already selected.
11. In the **Files** section, we locate the `ch5-env` directory.



12. We then right-click on the folder and select the option **Activate virtual environment** :

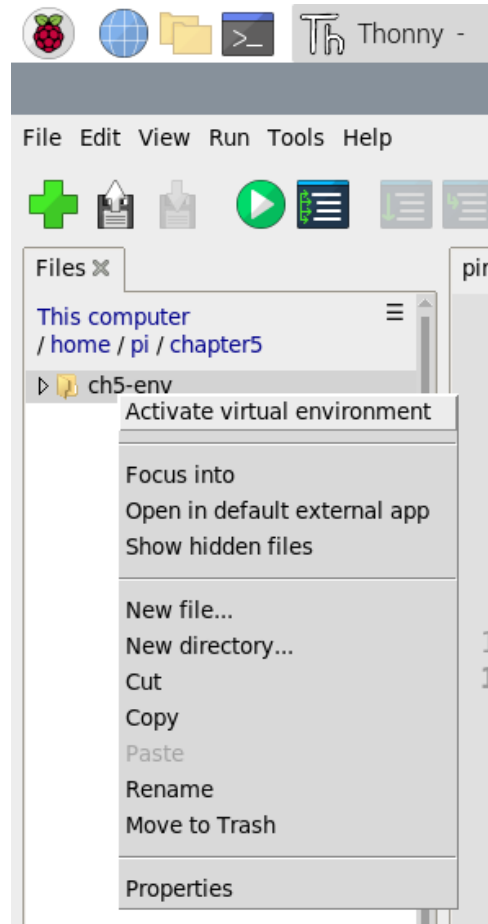


Figure 5.6 – Activating a Python virtual environment in Thonny

With our project folder created and our Python virtual environment set up and activated, we may now start writing code to access sensors connected to the GPIO port. We will start by exploring PIR sensors a little more.

## Exploring the PIR sensor

**Passive infrared (PIR)** sensors are devices specialized in detecting infrared radiation, typically emitted by living beings due to their body heat. These sensors work by monitoring changes in infrared levels, which occur when an infrared source, such as a human, moves across its field of view. PIR sensors can be recognized by their distinctive sphere shell that dominates the sensor (*A* in *Figure 5.4*).

Connecting a PIR sensor to a Raspberry Pi is straightforward. Three pins are needed:

- VCC is connected to 5V from the Raspberry Pi.
- GND is connected to a GND pin.
- Signal is connected to GPIO 23.

In *Figure 5.7*, we see a PIR sensor connected to a Raspberry Pi using a breadboard:

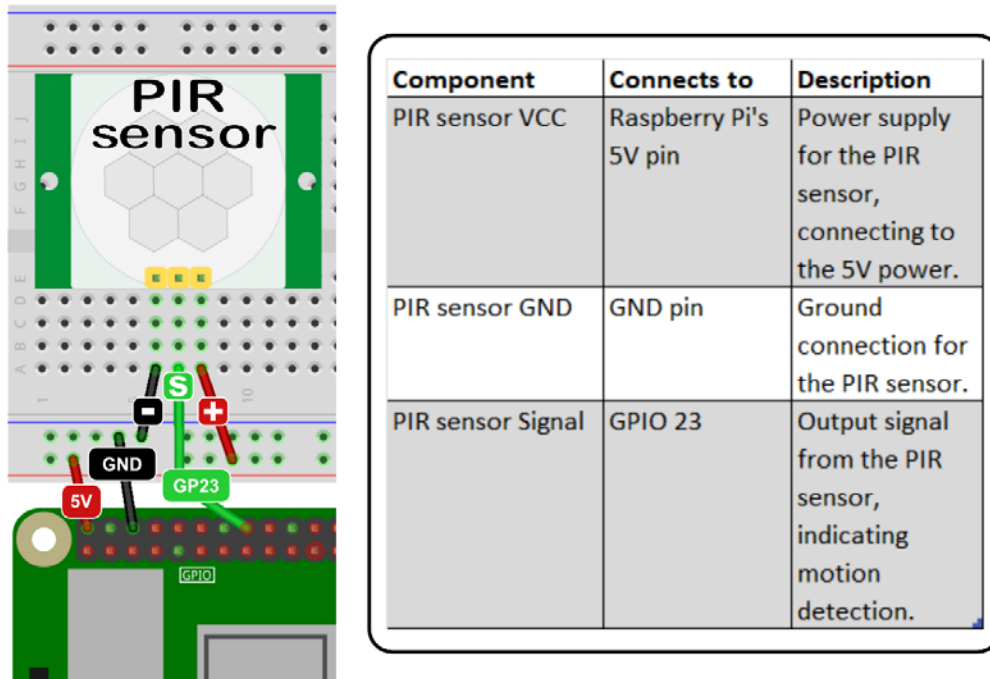


Figure 5.7 – Connecting a PIR sensor to a Raspberry Pi

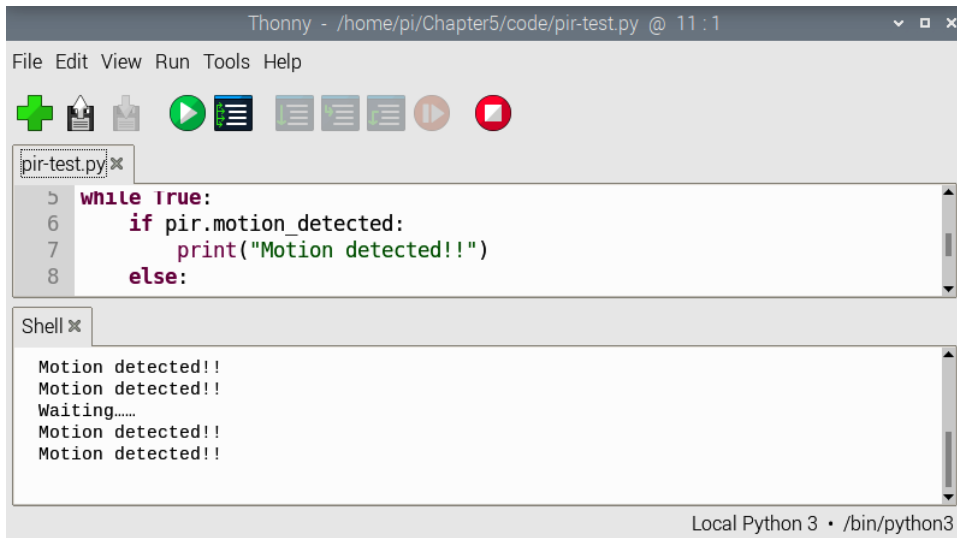
To acquire sensory data from our PIR sensor, we use the GPIO Zero library. To do so, we do the following:

1. In Thonny, we create a new tab by selecting **File** and then **New** or by hitting *Ctrl + N* on the keyboard.
2. Inside the tab, we enter the following code:

```
from gpiozero import MotionSensor
from time import sleep
pir = MotionSensor(23)
while True:
 if pir.motion_detected:
 print("Motion detected!!")
```

```
else:
 print("Waiting.....")
 sleep(5)
```

3. We save the code as `pir-test.py`.
4. To run the code in Thonny, we click on the green run button and hit *F5* on the keyboard or click on the **Run** menu option at the top and then **Run current script**.
5. We should see the message "Motion detected!!" when we move our hand near the PIR sensor:



```
Thonny - /home/pi/Chapter5/code/pir-test.py @ 11:1
File Edit View Run Tools Help
+ [Icons]
pir-test.py x
5 while True:
6 if pir.motion_detected:
7 print("Motion detected!!")
8 else:
Shell x
Motion detected!!
Motion detected!!
Waiting.....
Motion detected!!
Motion detected!!
Local Python 3 • /bin/python3
```

Figure 5.8 – Results from testing a PIR sensor

With the PIR sensor tested, we are now ready to build a basic alarm system using the Raspberry Pi.

## Building a simple alarm system

The Raspberry Pi's GPIO pins can be configured for specific communication protocols such as I2C or set as standard input/output pins to gauge conditions or levels.

In this final section of the chapter, we will use our knowledge to build a simple alarm system. Our alarm system will consist of a pushbutton, a PIR sensor, and a buzzer. All components are connected to the GPIO port (Figure 5.9).

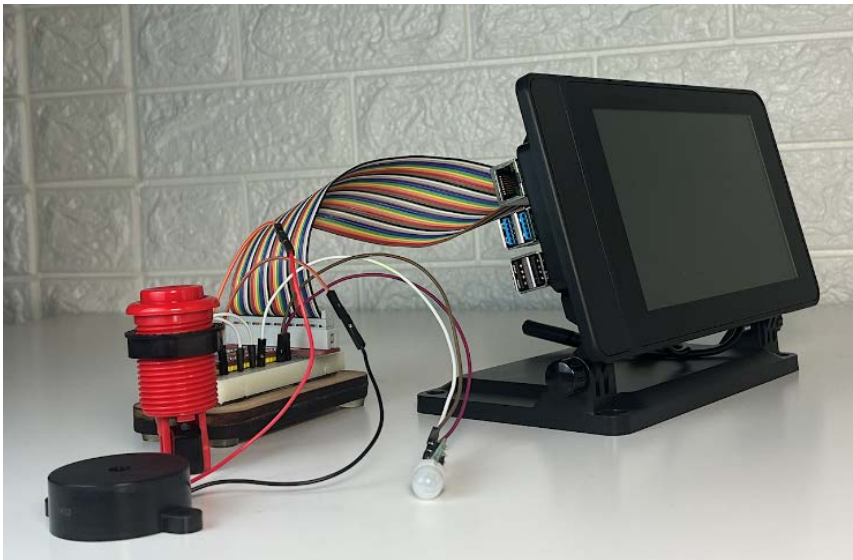


Figure 5.9 – Buzzer, pushbutton, and PIR sensor connected to the GPIO port through extension ribbon

In *Figure 5.9*, we are using a GPIO extension ribbon to connect the GPIO port to a breadboard so that we can easily prototype and reconfigure connections. The ribbon simplifies the process and keeps the wiring organized. Using the GPIO extension cable is entirely optional. Just as we did in *Chapter 4*, our Raspberry Pi is installed with the Raspberry Pi seven-inch touchscreen with the associated case.

We connect the components to the GPIO port using the following diagram:

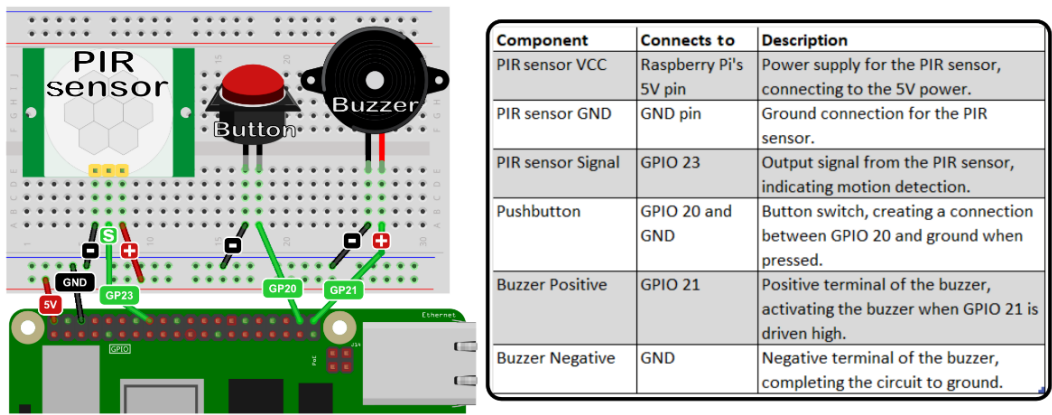


Figure 5.10 – Basic alarm circuit

We write the code for our alarm system in Thonny:

1. Inside Thonny, we create a new tab by selecting **File** and then **New** or by hitting *Ctrl + N* on the keyboard.
2. Inside the tab, we enter the following code:

```
from gpiozero import MotionSensor, Button, Buzzer
from time import sleep
pir = MotionSensor(23)
button = Button(20)
buzzer = Buzzer(21)
active = False
def toggle_alarm():
 global active
 if active:
 active = False
 buzzer.off()
 print("Alarm deactivated!")
 else:
 active = True
 print("Alarm activated!")
def monitor():
 while True:
 if active:
 pir.wait_for_motion()
 print("Motion detected!")
 sleep(5)
 if active:
 buzzer.on()
 print("Alarm triggered!")
 else:
 buzzer.off()
button.when_pressed = toggle_alarm
try:
 monitor()
except KeyboardInterrupt:
 print("Exiting...")
 buzzer.off()
```

Before executing our code, let's go through it.

- We start by importing our libraries:
  - `gpiozero` for Raspberry Pi GPIO operations
  - `time` for creating sleep intervals

- We then define our GPIO connections:
    - PIR sensor connected to GPIO pin 23
    - Button connected to GPIO pin 20
    - Buzzer connected to GPIO pin 21
    - `active`: The variable to keep track of alarm state (on/off)
  - We define our `toggle_alarm()` function:
    - Switches the alarm state between active and inactive
    - If the alarm is deactivated while it's sounding, the buzzer is turned off
  - Our `monitor()` function is defined next:
    - Continuously checks for motion if the alarm system is active
    - If motion is detected, the program waits for five seconds to allow for alarm deactivation
    - If the alarm is still active after the five-second delay, the buzzer sounds
  - We then set the `button.when_pressed` `gpiozero` property to bind it to the `toggle_alarm()` function. This allows us to assign a function to be executed immediately when a button is pressed. This event-driven approach eliminates the need for continuous polling.
  - Our main execution block runs the `monitor()` function.
  - The `try / catch` allows for a clean exit with a keyboard interrupt (`Ctrl + C`), ensuring the buzzer is turned off upon exit.
3. We save the program as `basic-alarm.py`.
  4. To run the code in Thonny, we click on the green run button, hit `F5` on the keyboard, or click on the **Run** menu option at the top, and then **Run current script**.
  5. We activate the alarm by pressing on a pushbutton. Our alarm's single sensor is a PIR sensor, which monitors motion.
  6. Once motion is detected, there is a five-second delay before the buzzer is activated. This allows time for the person who is aware of the alarm to turn it off before the buzzer goes off.
  7. To turn off the alarm, we simply push the button again.

In upcoming chapters, we'll transform our basic alarm system into an IoT-enabled alarm using a Raspberry Pi Pico W. This will allow us to monitor sensor data from any location worldwide, offering the convenience and versatility of managing our alarm system as though we were right beside the installed sensors.

## Summary

In *Chapter 5*, we explored the Raspberry Pi's GPIO pins and their communication capabilities, including protocols such as I2C, SPI, UART, and PCM. We highlighted the significance of the GPIO pinout diagram when working with the Raspberry Pi and Raspberry Pi Pico.

We focused on the PIR sensor for motion detection and connected it to our Raspberry Pi. We then used this knowledge to build a basic alarm system using the PIR sensor, a pushbutton, and a buzzer. Even though we didn't engage in a hands-on exercise with the Pico, the principles, and techniques we learned apply to both the Raspberry Pi and Raspberry Pi Pico.

This chapter commenced the construction of our IoT home security system. In the upcoming chapters, we will add to our basic alarm as we turn it into an impressive IoT home security system by creating an IoT alarm module and an IoT button to arm it.

# Building an IoT Alarm Module

In the previous chapter, we explored the Raspberry Pi's GPIO port and built a basic alarm system. We learned about different communication protocols and worked with a set of sensors, which we accessed using the GPIO port. In this chapter, we will enhance our basic alarm system using a Raspberry Pi Pico W, a public **Message Queuing Telemetry Transport (MQTT)** server, and the MQTTTHQ web client (*Figure 6.1*).

We will use a Raspberry Pi Pico W to host a **passive infrared (PIR)** sensor and buzzer as we build our IoT alarm module. In our setup, when motion is detected, a `motion` message is sent to the MQTT server and viewed using the MQTTTHQ web client:

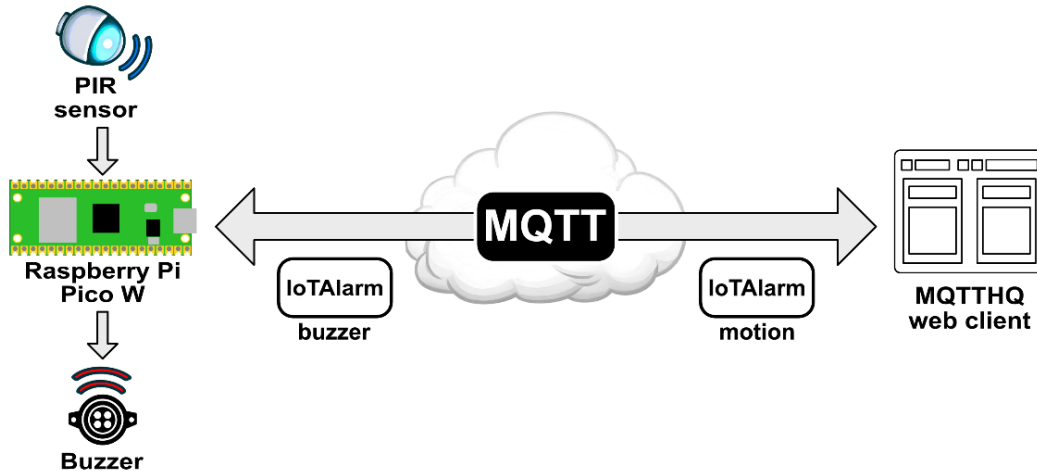


Figure 6.1 – IoT-powered alarm module using MQTT

We will send a `buzzer` message to the Raspberry Pi Pico W from the MQTTTHQ web client to activate the buzzer. This Raspberry Pi Pico W and MQTTTHQ web client setup forms the foundation of our IoT home security system.



We'll begin this chapter with an exploration of MQTT, using a public server as our development platform. Then, we'll familiarize ourselves with the Raspberry Pi Pico W, highlighting its strengths in IoT applications. Finally, we'll conclude by installing the components of our IoT alarm module into a custom 3D-printed case.

As such, we will cover the following main topics in the chapter:

- Investigating MQTT
- Using a Raspberry Pi Pico W with MQTT
- Building an IoT alarm module case

Let's begin!

## Technical requirements

The following are the requirements for completing this chapter:

- Intermediate knowledge of Python programming.
- 1 x Raspberry Pi Pico WH (with headers) to use with breadboard or Raspberry Pi Pico GPIO expander.
- 1 x Raspberry Pi Pico W (no headers) to be installed in an optional 3D-printed case.
- 1 x HC-SR501 PIR sensor.
- 1 x LED connected with a 220 Ohm resistor (refer to *Chapter 3* for construction).
- 1 x SFM-27 active buzzer.
- Access to a 3D printer or 3D printing service to print an optional case.

The code for this chapter can be found here:

<https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter6>

## Investigating MQTT

**MQTT** is a pivotal component in IoT, enabling lightweight and efficient communication among connected devices. In *Figure 6.1*, we illustrate MQTT within the ubiquitous cloud symbolizing the internet. MQTT operates on a **publish-subscribe** model, allowing devices to publish messages to specific topics while subscribing to relevant topics. This framework ensures efficient and selective communication, enabling devices to receive only messages relevant to their functions. MQTT's lightweight design minimizes resource overhead, making it an ideal choice for devices with constrained capabilities.

We start our investigation into MQTT by looking at how the publish-subscribe model in MQTT works.

## Understanding the publish-subscribe model in MQTT

The effectiveness of MQTT in enabling communication among IoT devices is due to its publish-subscribe model. This model provides a flexible and scalable way for devices to communicate with each other.

In MQTT, devices are categorized into two roles – publishers and subscribers:

- **Publishers:** Publishers create data or messages for sharing with other devices. They send these messages to designated **topics** to organize information for distribution. In *Figure 6.2*, both the PIR sensor and the temperature sensor act as publishers, utilizing a microcontroller (such as the Raspberry Pi Pico W, which is not depicted) for this purpose. The PIR sensor sends a `motion` message under the `IoTAlarm` topic, while the temperature sensor communicates a `25 C` message under the `temp` topic.
- **Subscribers:** Subscribers subscribe to one or more topics and receive any messages published under those topics. In *Figure 6.2*, the PC is subscribed to both the `IoTAlarm` and `temp` topics, while the phone is only subscribed to the `IoTAlarm` topic.

Topics in MQTT serve as channels or communication pathways. We may think of topics as labels or categories that messages fall under, such as the `IoTAlarm` and `temp` topics marked with black boxes with white lettering in *Figure 6.2*. When a publisher sends a message to a specific topic, the MQTT broker (server) manages the message.

The broker maintains a list of all subscribers to that topic, guaranteeing message delivery to each. This mechanism allows for efficient and selective communication because devices only receive messages from topics they have subscribed to. In *Figure 6.2*, we see our PC subscribing to the `IoTAlarm` and `temp` topics and our phone subscribing to only the `IoTAlarm` topic:

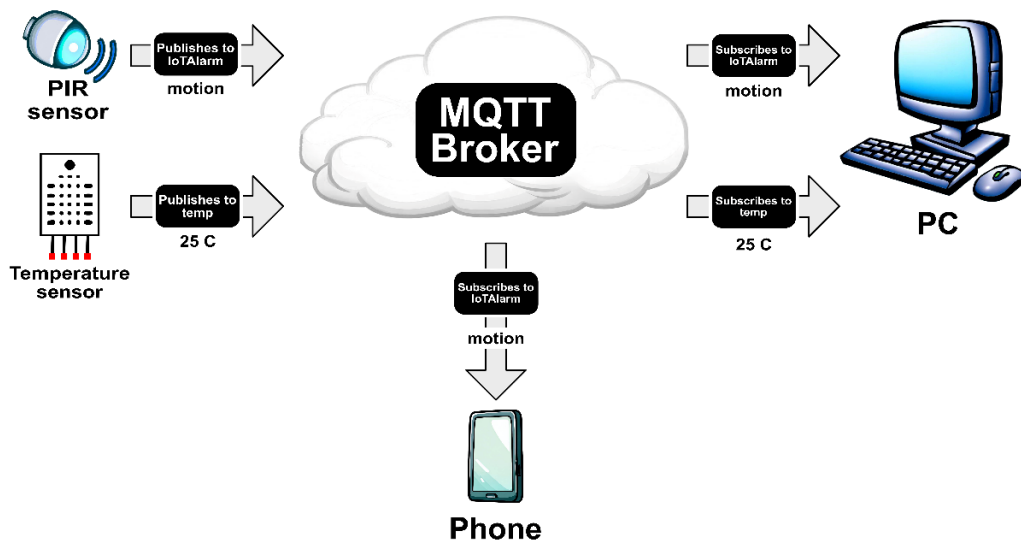


Figure 6.2 – MQTT communications illustrated

Before we try some hands-on experience with MQTT, we will look at **Quality-of-Service (QoS)** levels in MQTT. Understanding QoS levels is crucial as they determine the reliability and delivery assurance of messages in MQTT.

## Understanding QoS in MQTT

**QoS** is an important aspect of MQTT that determines the level of guarantee for message delivery between the MQTT broker (server) and MQTT clients (devices or applications).

MQTT offers three levels of QoS:

- **QoS 0 (At Most Once):** In this mode, the message is delivered at most once, meaning that the message may not be delivered to the recipient, and it may get lost without any acknowledgment or guarantee of receipt. This level of QoS is suitable for scenarios where message loss is acceptable and message delivery is not critical.
- **QoS 1 (At Least Once):** QoS 1 ensures that the message is delivered at least once to the recipient. If the broker or client doesn't receive an acknowledgment (ACK) that the message was received, it will resend the message. This level of QoS guarantees that the message is received by the recipient but may result in duplicate messages.
- **QoS 2 (Exactly Once):** QoS 2 provides the highest level of guarantee. It ensures that the message is delivered exactly once to the recipient. This level of QoS involves a more complex handshake between the sender and receiver to ensure no duplicates or message loss occur.

For our development purposes, QoS 0 is adequate, as it offers reasonable message delivery without the need for the more intricate message tracking and acknowledgment mechanisms required by QoS 1 and QoS 2. QoS 0 simplifies message handling in code, making it a practical choice for development scenarios.

## Exploring MQTT fundamentals with the MQTTTHQ web client

To acquire practical knowledge of MQTT, we'll utilize the **MQTTTHQ web client**. This web-based service streamlines the process of learning MQTT, eliminating the need for complex installations or extensive programming. As a public resource aimed at development and testing, it provides an accessible environment for us to explore and understand the features of MQTT.

We begin by opening the web client in a web browser:

1. In our browser, we navigate to the client using the following URL: `https://mqttthq.com/client`.
2. To ensure that we can use the client for our testing, we verify that we are connected to `public.mqttthq.com` from the message at the top right of the screen:



Figure 6.3 – Connected to the mqtthq.com client

If the message indicating *connected* does not appear, we continue refreshing the page until it does.

3. In the **Subscribe to topic** section, we change the topic to `IoTAlarm`, keep the **QoS** level at 0, and click on the **Subscribe** button.
4. We should notice that the text under **Received payloads** updates to display **Waiting for data...** and the **Subscribe** button has turned into an **Unsubscribe** button:

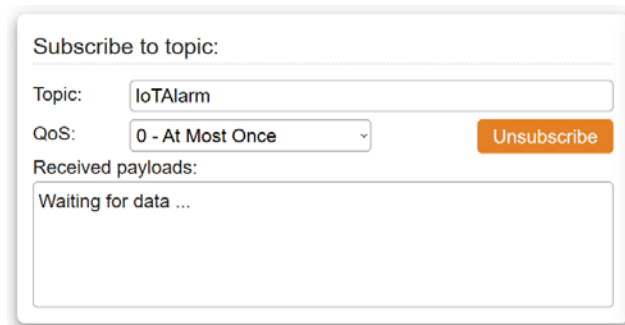


Figure 6.4 – Subscribing to the IoTAlarm topic

5. In the **Publish to topic** section, we change the topic to `IoTAlarm`, keep the **QoS** level at 0, replace the `Hello World!` message with `motion`, and then click on the **Publish** button:

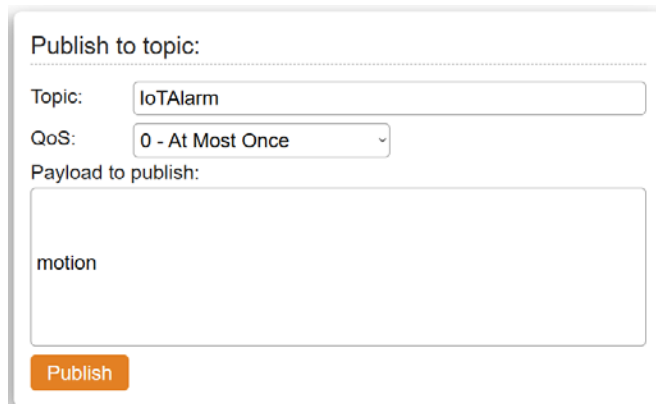
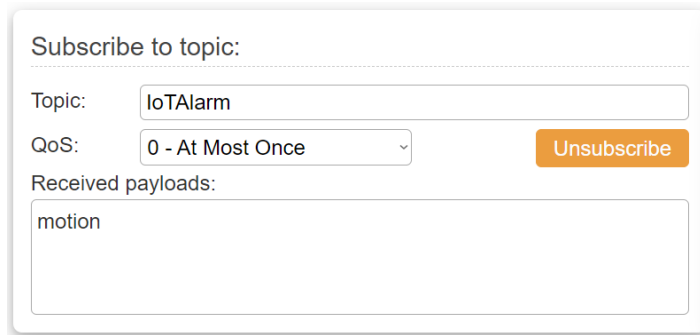


Figure 6.5 – Publishing a message to the IoTAlarm topic

6. We should notice that our `motion` message has now appeared under the **Received payloads** box in the **Subscribe to topic** section:



The screenshot shows the MQTTTHQ web client interface. At the top, there's a section titled "Subscribe to topic:". Below this, there are two input fields: "Topic:" with the value "IoTAlarm" and "QoS:" with a dropdown menu showing "0 - At Most Once". To the right of these fields is an orange button labeled "Unsubscribe". Below the input fields is a section titled "Received payloads:". Inside this section is a text box containing the word "motion".

Figure 6.6 – MQTT message received

7. To confirm that we are successfully sending MQTT messages from a publisher to a subscriber, we can use two separate computers: one to publish messages and the other to subscribe and receive them.

Through this exercise, we have effectively demonstrated the process of publishing and subscribing to MQTT messages using the MQTTTHQ web client. In the next section, we will start building our IoT alarm module by leveraging the capabilities of the Raspberry Pi Pico W.

## Using a Raspberry Pi Pico W with MQTT

In this section, we will build the physical alarm part of our application using a **Raspberry Pi Pico W**. This microcontroller not only is affordable but also brings a range of capabilities to our project, allowing us to execute tasks efficiently without the full power of a **Single-Board Computer (SBC)** such as the Raspberry Pi.

The Raspberry Pi Pico W doesn't replace our Raspberry Pi; it complements it, adding unique strengths to our toolkit. Being a microcontroller, the Pico W is substantially more cost-effective compared to the Raspberry Pi, and it typically doesn't heat up as much due to its simpler architecture and lower power consumption. This distinction is pivotal for projects such as our IoT alarm module, where the primary task is to capture sensory data – a function that does not require the computational power of an SBC. This allows us to reserve our Raspberry Pi for tasks that demand more computational resources.

Being a microcontroller, our Raspberry Pi Pico W boots up quickly, providing a rapid start for our program. We do not need to load a heavy operating system.

## Introducing the RP2040 chip

The Raspberry Pi Pico W utilizes the dual-core ARM Cortex-M0+ processor RP2040 chip created by the Raspberry Pi Foundation. This chip was designed as a bridge between microcontrollers and microcomputers by merging the streamlined operation typical of microcontrollers with the capacity to undertake more demanding microcomputer-type tasks.

The *W* in our Raspberry Pi Pico W denotes that our microcontroller supports Wi-Fi. Besides the Raspberry Pi Pico W, there is the standard Pico (without Wi-Fi), the Pico H (without Wi-Fi and with soldered headers), and the Pico WH (with Wi-Fi and soldered headers).

The RP2040 chip our Raspberry Pi Pico W is based on may also be found on other microcontrollers such as the *Arduino Nano RP2040 Connect*, *Pimoroni Tiny 2040*, and *Adafruit Feather RP2040*. In Figure 6.7, we see a Waveshare RP2040-Zero-M (A) a Raspberry Pi Pico (B), and a Raspberry Pi Pico W (C):

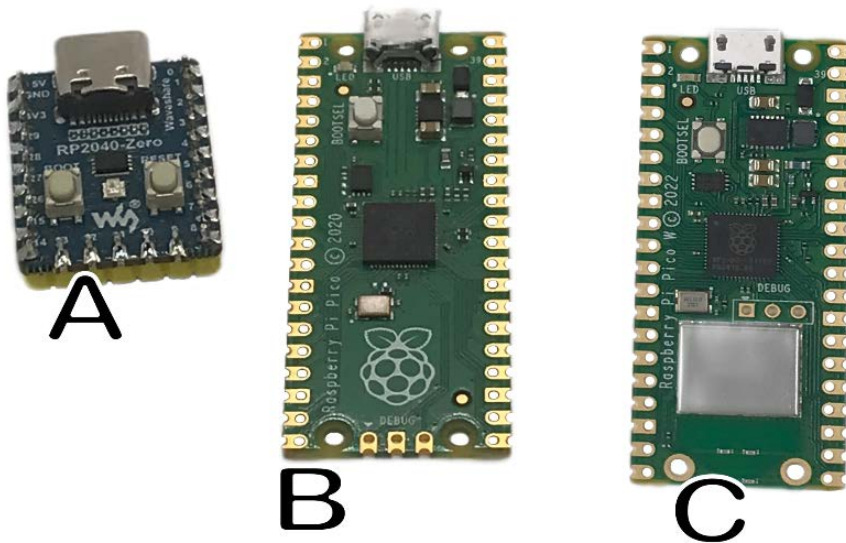


Figure 6.7 – RP2040-based microcontrollers

To construct our alarm circuit on a breadboard, we need headers on our Raspberry Pi Pico W. We can either solder them on ourselves or choose to purchase the Pico WH version.

We will start the construction of our IoT alarm module by building out the circuit on a breadboard.

## Configuring our alarm circuit

In this section, we'll build our alarm circuit using an SFM-27 active buzzer, an LED with a resistor, and an HC-SR501 PIR sensor. We will configure our circuit on a breadboard before moving our components to a 3D-printed case. We may use a Raspberry Pi Pico GPIO expander in place of a breadboard. For the breadboard, we use male jumpers soldered to our components for connection to the breadboard. For the GPIO expander, we use female jumper wires soldered to our components for connections.

In *Figure 6.8*, we can see our circuit illustrated in a Fritzing diagram. To create our circuit, we wire our components to our Raspberry Pi Pico W with the connections outlined in *Figure 6.8*:

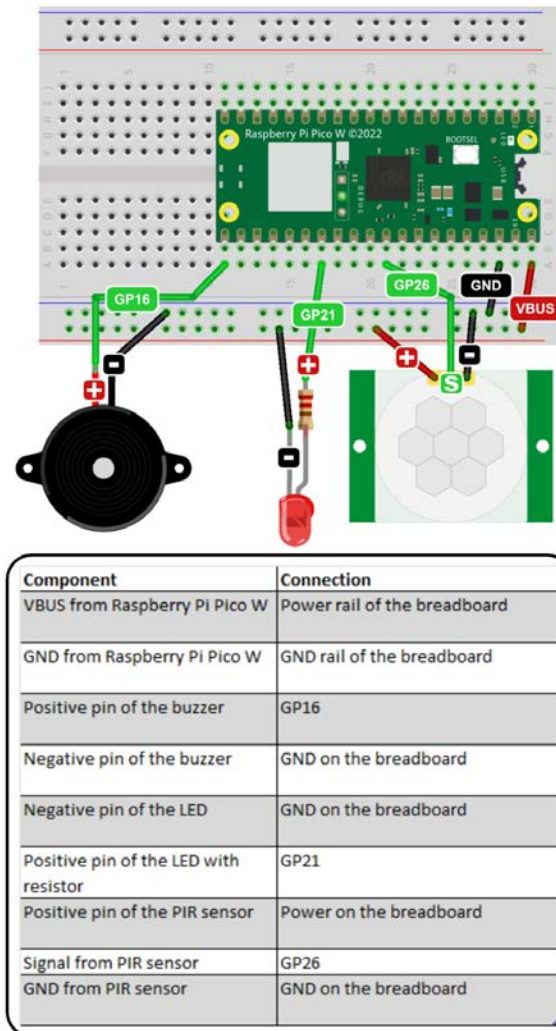


Figure 6.8 – Raspberry Pi Pico W alarm circuit

It's important to note our connection to the VBUS port on the Raspberry Pi Pico W. When the Pico is powered via USB, components connected to the VBUS port will receive approximately 5V, which is the standard USB voltage. We will be powering our IoT alarm module through the USB port.

The following table outlines the power ports on the Raspberry Pi Pico W and provides insights into how we might utilize them in future projects:

| Port     | Input Voltage Use                                                             | Output Voltage Use                                                            |
|----------|-------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| VBUS     | Used to supply power from a 5V USB source.                                    | Can be used to supply 5V to external components when the Pico is USB-powered. |
| VSYS     | Accepts an external power supply ranging from 1.8V to 5.5V.                   | Not typically used to supply power to external components.                    |
| 3V3(OUT) | Not commonly used for input.                                                  | Provides a regulated 3.3V supply to power external 3.3V components.           |
| 3V3_EN   | Not a power supply port, but a control pin to enable/disable the 3.3V supply. | Not applicable.                                                               |

Table 6.1 – Power ports on the Raspberry Pi Pico (W)

With our circuit wired up, we are ready to start coding. We will start by setting up Thonny for microcontroller development.

## Setting up our development environment

As with all the software development we have done so far, we will use Thonny as our **Integrated Development Environment (IDE)**. The choice of Thonny's OS version (Windows, macOS, Linux, Raspberry Pi OS, and so on) is flexible since our focus is on writing code for a connected microcontroller rather than the computer used for coding.

### Important note

It's important to note that different OS versions of Thonny may exhibit varying levels of functionality necessary for this chapter. The content in this section is based on the Windows version of Thonny, and the screenshots provided reflect this.

We'll develop our code using **MicroPython**, a lightweight version of Python optimized for microcontrollers. MicroPython shares core syntax and functionality with Python, but it's important to note that due to its focus on resource-constrained environments, it may lack some of the extensive libraries and features available in standard Python. These differences, however, are a trade-off for the efficiencies of MicroPython when used for microcontroller programming.



To install MicroPython on our Raspberry Pi Pico W, we do the following:

1. If Thonny is not available on our operating system, we visit the Thonny website and download an appropriate version (<https://thonny.org>).
2. We then launch Thonny using the appropriate method for our operating system.
3. While holding the *BOOTSEL* button on the Pico W (the small white button near the USB port), we insert it into an available USB port and disregard any pop-up windows that may appear.
4. We then click on the interpreter information at the bottom right-hand side of the screen and select **Install MicroPython...**:

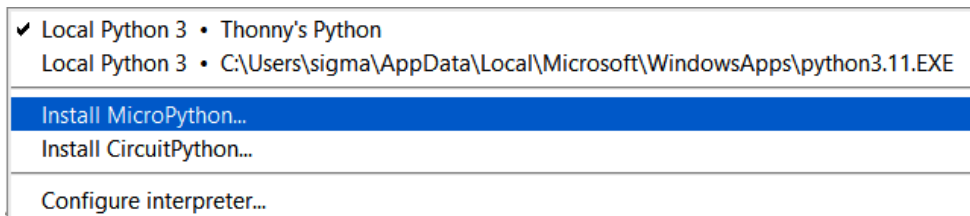


Figure 6.9 – Install MicroPython... option

5. For the **MicroPython variant**, we select **Raspberry Pi • Pico W / Pico WH**:

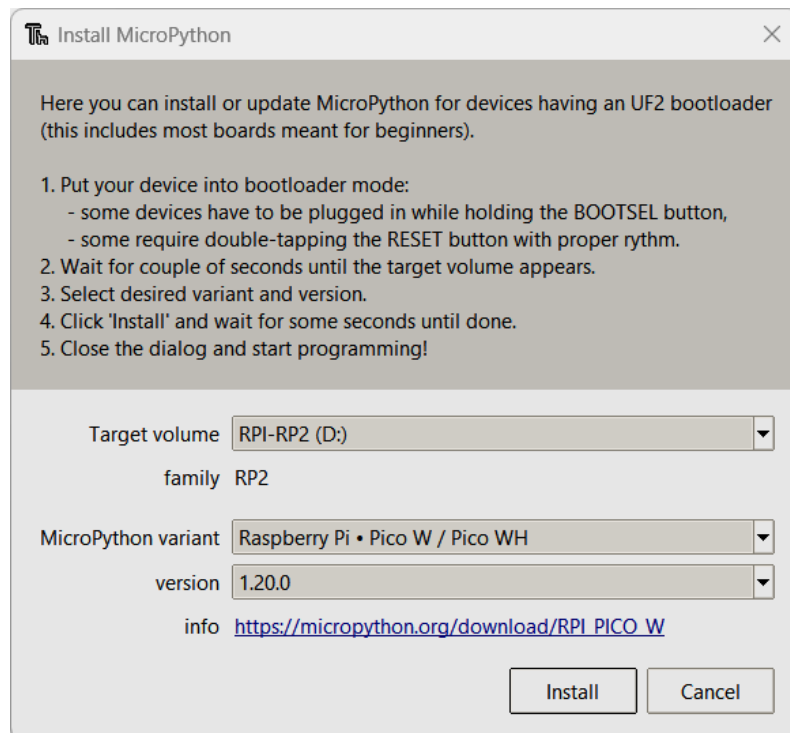


Figure 6.10 – Installing MicroPython on the Raspberry Pi Pico W

6. We click on the **Install** button and then the **Close** button once the installation has been completed.
7. To have Thonny configured to run the MicroPython interpreter on our Pico W, we select it from the bottom right-hand side of the screen:

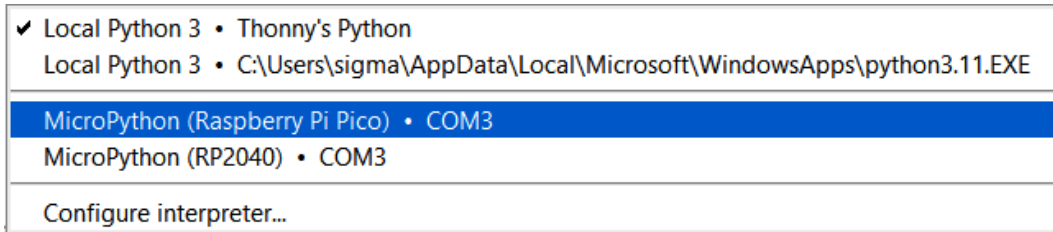


Figure 6.11 – Selecting the MicroPython interpreter from our Pico W

8. We confirm that Thonny is using the MicroPython interpreter on our Raspberry Pi Pico W by checking the **Shell**:

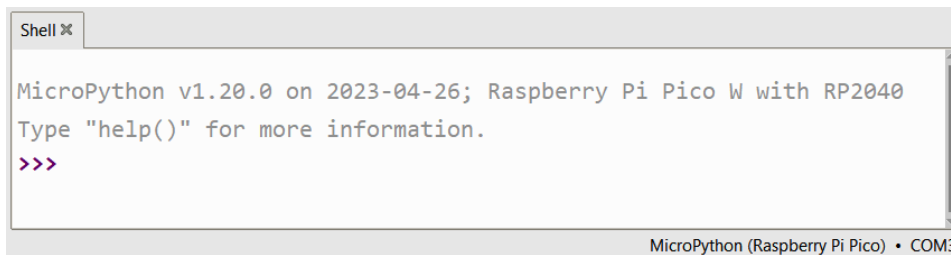


Figure 6.12 – MicroPython prompt in Thonny

9. To run the MQTT code, we will require an MQTT library to be installed. To do so, we select **Tools | Manage packages....**, enter `umqtt` in the search box, and click on **Search on PyPI**:

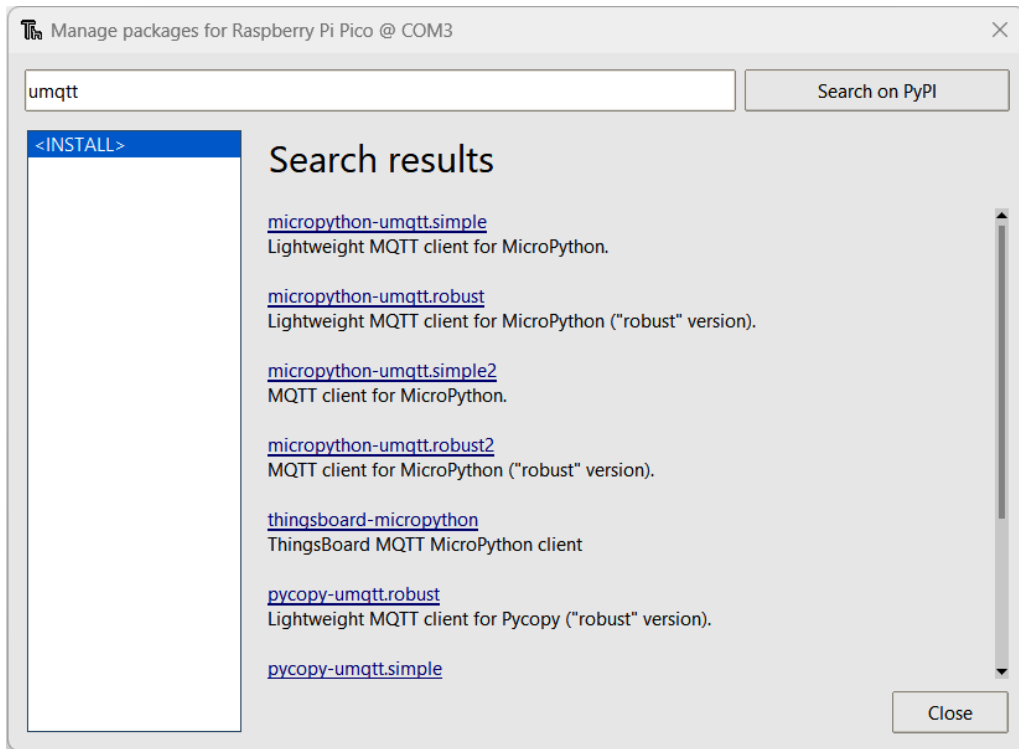


Figure 6.13 – Installing an MQTT library onto our Pico W

10. We select the `micropython-umqtt.simple` package and click on the **Install** button.
11. We then close the dialog by clicking on the **Close** button.

Now that we have MicroPython and an MQTT library installed on our Raspberry Pi Pico W, we're prepared to begin coding. Our initial focus will be on connecting the components, followed by the implementation of MQTT-related code.

## Writing the alarm module client code

By now, we should be well acquainted with the Thonny IDE. Connecting to the MicroPython interpreter on the Raspberry Pi Pico W doesn't significantly alter our interaction with Thonny.

Nonetheless, it's advantageous for our development process to have visibility into files stored on both the Pico W and our computer. This visibility allows us to easily verify file locations and manage our project effectively.

To open the **Files** view in Thonny, we click on the **View** menu at the top and select **Files**:

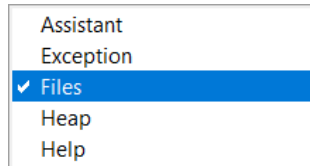


Figure 6.14 – Enabling Files view in Thonny

We should see a view of our project files on both the Raspberry Pi Pico W and our computer on the left-hand side of Thonny.

We are now ready to start writing code. We will start with the buzzer.

### ***Activating the buzzer through code***

Figure 6.1 illustrates an `IoTAlarm` type message sent from the MQTT broker to our Raspberry Pi Pico W. This message serves the purpose of activating the buzzer in our alarm module. To handle this task, we'll create a separate program. Activating the buzzer involves a slightly more complex process compared to monitoring the PIR sensor or LED components in our circuit, and thus a desire to separate its code.

To do this, we do the following:

1. We connect our Raspberry Pi Pico W to a USB port and launch Thonny. We may use our Raspberry Pi or another operating system for this.
2. We then activate the MicroPython environment on our Pico W by selecting it from the bottom right-hand side of the screen.
3. In a new editor tab, we enter the following code:

```
from machine import Pin, PWM
import utime

BUZZER_PIN = 16
buzzer = PWM(Pin(BUZZER_PIN))
BUZZER_FREQ = 4000

def activate_buzzer(duration=5):
 buzzer.freq(BUZZER_FREQ)
 buzzer.duty_u16(32768)
 utime.sleep(duration)
 buzzer.duty_u16(0)
```

4. To save the file, we click on **File | Save as...** from the drop-down menu. This will open the following dialog:

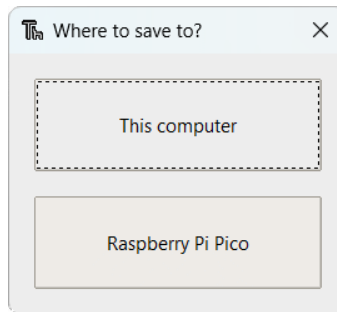


Figure 6.15 – Saving a file to our Raspberry Pi Pico W

5. In this dialog, we are given the option to choose where to store our file. To save it on our Raspberry Pi Pico W, we click on the corresponding button.
6. We then name the file `buzzer.py` and click **OK**.

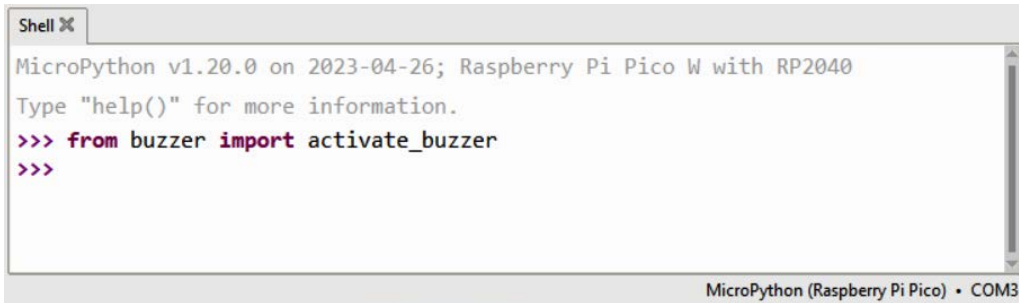
Before we test out our code, let's break it down:

- I. We start by importing the `Pin` and `PWM` (for **pulse-width modulation**, or **PWM**) classes from the `machine` module.
- II. We import the `utime` class for timer functionality.
- III. We set the `BUZZER_PIN` constant to 16. This corresponds to our wiring diagram for the buzzer.
- IV. We create a `buzzer` object by initializing the `PWM` class on the specified `BUZZER_PIN` constant. This PWM-based approach allows us to vary the voltage supplied to the buzzer rapidly, enabling control over the sound's tone and volume.
- V. We then set the `BUZZER_FREQ` constant to 4000, representing the frequency of the PWM signal used for the buzzer.
- VI. We then define an `activate_buzzer()` function. This function takes an optional `duration` parameter (default is 5 seconds).
- VII. Inside the `activate_buzzer()` function, we do the following:
  - i. We set the frequency of the `buzzer` object to the specified `BUZZER_FREQ` constant.
  - ii. We set the buzzer's duty cycle to 50% (32768 out of the full 16-bit range of 65536), creating a balanced tone, with the buzzer active for half of the signal's 16-bit cycle and inactive for the other half.

- iii. Our code then pauses the program for the specified duration in seconds using the `utime.sleep()` function.
- iv. After the specified duration, set the duty cycle of the `buzzer` object back to 0, turning off the buzzer.

We can test our code using the Shell in Thonny. To do so, we do the following:

1. In the Shell, we import the `activate_buzzer()` function from the `buzzer` file:

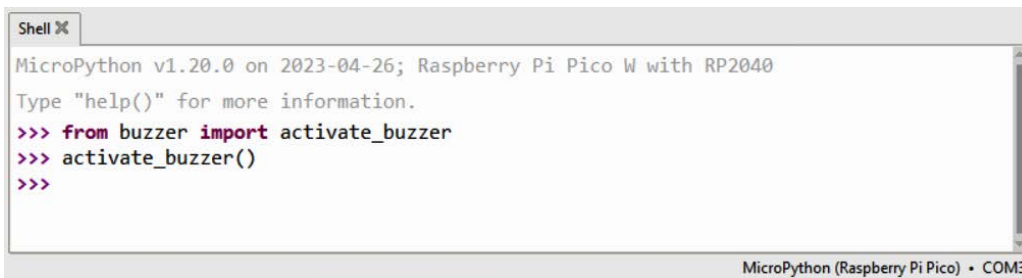


```
Shell X
MicroPython v1.20.0 on 2023-04-26; Raspberry Pi Pico W with RP2040
Type "help()" for more information.
>>> from buzzer import activate_buzzer
>>>
```

MicroPython (Raspberry Pi Pico) • COM3

Figure 6.16 – Importing the `activate_buzzer()` function

2. With the function imported, we can run it by simply calling it and hitting *Enter* on the keyboard:



```
Shell X
MicroPython v1.20.0 on 2023-04-26; Raspberry Pi Pico W with RP2040
Type "help()" for more information.
>>> from buzzer import activate_buzzer
>>> activate_buzzer()
>>>
```

MicroPython (Raspberry Pi Pico) • COM3

Figure 6.17 – Activating the buzzer

3. With our buzzer wired correctly, we should hear it sound for 5 seconds. To adjust the duration, we simply pass another value into the `activate_buzzer()` function.

We must congratulate ourselves as we have just written and executed our first MicroPython program! With the buzzer code done, it is time to create the main program for our alarm module.

### ***Creating the main code***

In this section, we will write the code to power our IoT alarm module. In this code, we will connect to the PIR sensor and LED, as well as the Wi-Fi and MQTT broker.

In MicroPython, two files control Pico W's startup and code execution: `boot.py` handles essential initialization during boot, while `main.py` contains the primary user code for custom logic and functions.

For our code, we won't concern ourselves with `boot.py`. We will, however, concentrate on `main.py`, the program responsible for launching our application on Pico W when it is first powered.

To write our IoT alarm module code, we do the following:

1. In a new tab in Thonny, we start by entering our code for imports:

```
import machine
import utime
import network
import _thread
from umqtt.simple import MQTTClient
from buzzer import activate_buzzer
```

In our preceding code, we have the following:

- `machine`: Provides access to hardware-related functions and classes for our microcontroller.
- `utime`: Offers time-related functions and timing control for managing delays and timestamps.
- `network`: Provides network-related functions for configuring and managing network connections.
- `_thread`: Allows creating and managing threads for concurrent execution of code.
- `MQTTClient` (from `umqtt.simple`): Provides MQTT client functionality for communication over MQTT.
- `activate_buzzer` (from `buzzer`): Our custom function for activating a buzzer.

2. We then set our variables:

```
SSID = "WiFiNetworkName"
PASSWORD = "xxxxxxxxxx"
MQTT_SERVER = "broker.mqtthq.com"
MQTT_PORT = 1883

pir = machine.Pin(26, machine.Pin.IN)
led = machine.Pin(21, machine.Pin.OUT)
```

```
wlan = network.WLAN(network.STA_IF)
mqtt_client = None
```

In the preceding code, we have the following variables:

- `SSID`: Variable storing the name (SSID) of the Wi-Fi network our program connects to
  - `PASSWORD`: Variable storing the password for the Wi-Fi network
  - `MQTT_SERVER`: Variable storing the MQTT broker's/server's address (we will be using `broker.mqtt.com`)
  - `MQTT_PORT`: Variable storing the MQTT port number
  - `pir`: Configures a GPIO pin 26 as an input pin for our PIR sensor
  - `led`: Configures a GPIO pin 21 as an output pin to control our LED
  - `wlan`: Initializes a WLAN (Wi-Fi) interface for connecting to a Wi-Fi network in the station (client) mode, allowing our MicroPython device to act as a client and join an existing wireless network. This initialization is essential, as it allows our Pico W to connect to an existing Wi-Fi network, enabling networked communication
3. After defining our variables, we create a function to connect our Raspberry Pi Pico W to our Wi-Fi network:

```
def connect_wifi():
 wlan.active(True)
 wlan.connect(SSID, PASSWORD)

 while not wlan.isconnected():
 print('Trying to connect to WiFi...')
 utime.sleep(5)

 print('WIFI connection established')
```

In our code, we have the following:

- `wlan.active(True)`: Activates the Wi-Fi interface
- `wlan.connect(SSID, PASSWORD)`: Initiates the connection to the Wi-Fi network using the specified SSID (network name) and password (network password)
- `while not wlan.isconnected() :` This loop continuously checks whether the device is connected to the Wi-Fi network:
  - `print('Trying to connect to WiFi...')`: If not connected, it prints a message indicating the ongoing connection attempt
  - `utime.sleep(5)`: It pauses for 5 seconds before checking the connection status again



- `print('WiFi connection established')`: Once connected, our code prints a message confirming the successful connection to the Wi-Fi network
4. With our Wi-Fi connection function in place, we then add a function responsible for handling the buzzer message received from our MQTT broker:

```
def sub_iotalarm(topic, msg):
 print((topic, msg))
 if topic == b'IoTAlarm' and msg == b'buzzer':
 print("buzzer is detected")
 activate_buzzer()
```

In our code, the following happens:

- I. Our `sub_iotalarm()` function handles incoming MQTT messages by first printing the topic and message received
  - II. If the topic is `IoTAlarm` and the message is `buzzer`, it calls the `activate_buzzer()` function to trigger the buzzer
5. The `motion_handler()` function is responsible for handling motion detection events, printing notifications, and, if the MQTT client is connected, publishing a `motion` message to the `IoTAlarm` topic:

```
def motion_handler(pin):
 print('Motion detected!!')
 if mqtt_client is not None:
 mqtt_client.publish(b"IoTAlarm", b"motion")
 else:
 print("MQTT client is not connected.")
```

In our code, the following happens:

- I. Our `motion_handler()` function takes in a parameter called `pin`. This parameter is needed as a placeholder for the interrupt handler's expected argument; even though we don't use it within the function, it is required to maintain compatibility with the interrupt system.
  - II. We use the `b` prefix to denote that the string (`IoTAlarm` and `motion`) should be treated as a byte object rather than a text (Unicode) string, which is required for sending binary data in protocols such as MQTT.
6. The `connect_mqtt()` function establishes a connection between our code and the MQTT broker:

```
def connect_mqtt(device_id, callback):
 global mqtt_client

 while mqtt_client is None:
 try:
```

```

 print("Trying to connect to
MQTT Server...")
 mqtt_client = MQTTClient(
 device_id,
 MQTT_SERVER,
 MQTT_PORT)

 mqtt_client.set_callback(callback)
 mqtt_client.connect()
 print('MQTT connection established')
 except:
 mqtt_client = None
 print('Failed to connect to MQTT Server,
retrying...')
 utime.sleep(5)

```

In our code, the following happens:

- I. The `connect_mqtt()` function establishes a connection between our code and the MQTT server, taking two parameters: `device_id` for device identification and `callback` for specifying a function that processes incoming messages (known as a callback function). The `device_id` parameter is a unique identifier assigned to each MQTT client, allowing our MQTT broker to distinguish specific devices on the network.
- II. Within the `while` loop, our code attempts to connect to the MQTT server using the given device ID, configures the callback function to handle messages, and successfully establishes an MQTT connection. If the connection encounters any issues, our function retries after a 5-second pause.

#### What is a callback function?

In the context of our IoT alarm system, a callback function is used as part of the MQTT communication process. In our code, we use the `sub_iotalarm()` function as the callback, which means that when relevant MQTT messages are received from the MQTT broker, the `sub_iotalarm()` function is automatically invoked. Inside our callback function, we have defined specific actions to be taken based on the received messages, such as activating the buzzer.

7. The final method controls the LED's blinking pattern, indicating the application's connection status, and enables troubleshooting when the Raspberry Pi Pico W is running independently from a computer:

```

def connection_status():
 while True:
 if wlan.isconnected():
 if mqtt_client is not None:
 led.on() # Steady on when both WiFi and MQTT
connected

```

```

 else:
 led.on() # Blink every half-second when only
WiFi is connected
 utime.sleep(0.5)
 led.off()
 utime.sleep(0.5)
 else:
 led.on() # Blink every second when WiFi is not
connected
 utime.sleep(1)
 led.off()
 utime.sleep(1)

```

In our code, we have the following:

- **Steady On:** The LED remains constantly on when both Wi-Fi and MQTT are connected. This occurs when Wi-Fi is connected (`wlan.isconnected()` is `True`) and there is a value for `mqtt_client`.
  - **Fast Blink:** When only Wi-Fi is connected (MQTT client is `None`), the LED blinks rapidly every half-second.
  - **Slow Blink:** When neither Wi-Fi nor MQTT is connected, the LED blinks more slowly, with a 1-second interval for on and off states.
8. To enable independent execution of the `connection_status()` function, our code starts a new thread. Threading enables concurrent execution of multiple tasks or functions, making efficient use of the RP2040's dual-core processor to run distinct operations simultaneously:

```
_thread.start_new_thread(connection_status, ())
```

9. Our code then calls functions to connect to the Wi-Fi and MQTT broker using a unique client ID, `IoTAlarmSystem`:

```

connect_wifi()
connect_mqtt("IoTAlarmSystem", sub_iotalarm)

```

10. We then subscribe to the `IoTAlarm` message:

```
mqtt_client.subscribe("IoTAlarm")
```

11. To enable our PIR sensor, we set its `irq()` method with the following:

```

pir.irq(trigger=machine.Pin.IRQ_RISING,
handler=motion_handler)

```

In our code, the following happens:

- I. An **interrupt request (IRQ)** triggers when the pin detects a rising edge, indicating a change from low to high voltage.
  - II. When an IRQ is triggered, the `motion_handler()` function is invoked to publish a `motion` message to the MQTT broker.
12. In an infinite loop, we wait for a message:

```
while True:
 mqtt_client.wait_msg()
```

Once a message is received it is processed by the callback function, which we defined as `sub_iotalarm()` earlier in the code.

13. We save the code as `main.py` onto our Raspberry Pi Pico W to ensure that when we power it on or reset it, our code runs automatically. This is standard with MicroPython.

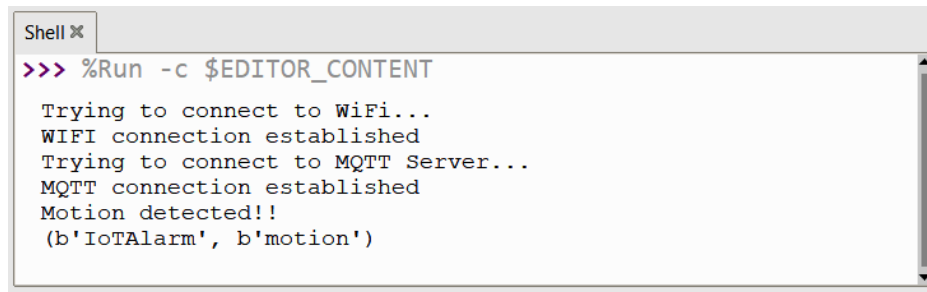
With the code written, it is time to test it out with the MQTTTHQ web client.

### ***Running our alarm module application***

We have two options for running the code on our Raspberry Pi Pico W. One is to power up the Pico W with a standard USB power cable, which is suitable for use after testing, but it won't provide access to debug print messages. The second option is to run our code in Thonny. This will allow us to troubleshoot any issues we encounter. To do this, we follow these steps:

1. We select the `main.py` tab in Thonny, ensuring that we pick the version installed on our Pico W rather than our operating system.
2. We click on the green run button, hit *F5* on the keyboard, or click on the **Run** menu option at the top and then **Run current script**.
3. We should observe messages in the Shell that our code is connecting first to the Wi-Fi network and then to the MQTT server.
4. We should also observe that our LED blinks accordingly, slowly before connecting to the Wi-Fi network, faster after connecting to the Wi-Fi network but before connecting to the MQTT server, and steady once both connections are made.

5. Moving our hand in front of the PIR sensor, we should observe a `Motion detected!!` message followed by a message coming back from the MQTT server:

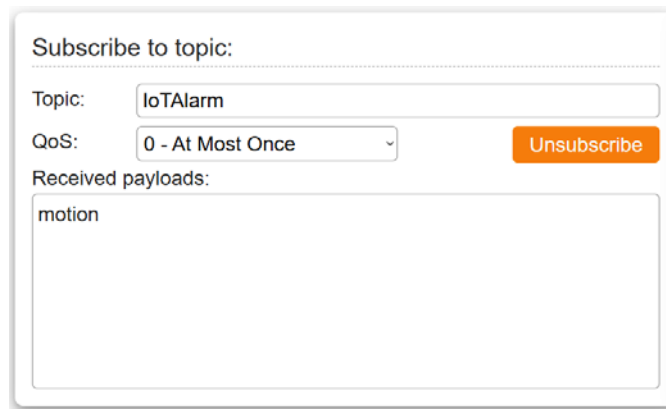


```
Shell x
>>> %Run -c $EDITOR_CONTENT

Trying to connect to WiFi...
WIFI connection established
Trying to connect to MQTT Server...
MQTT connection established
Motion detected!!
(b'IoTAlarm', b'motion')
```

Figure 6.18 – Detecting motion with the PIR sensor

6. If we only receive a `Motion detected!!` message but not a message coming back from the MQTT server (broker), then our application has lost connection to the server. This should also be indicated by our LED flashing slowly. To fix this, we stop and restart our program using the **Stop** and **Run** buttons respectively.
7. To verify that our code is sending MQTT messages, we follow the steps from the previous section, *Exploring MQTT fundamentals with the MQTTHQ web client*. After subscribing to the `IoTAlarm` topic, the web client should receive a `motion` message whenever our PIR sensor is triggered:



Subscribe to topic:

Topic:

QoS:

Received payloads:

Figure 6.19 – Receiving motion messages

8. To test out our buzzer, we publish a `buzzer` message using the MQTTTHQ web client:

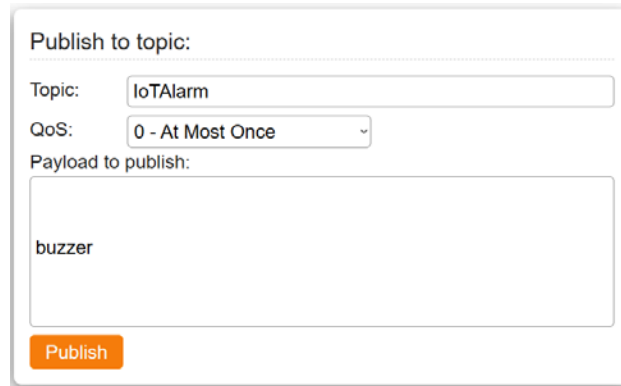
The image shows a web interface for publishing an MQTT message. It has a title "Publish to topic:" followed by a dashed line. Below this, there are three fields: "Topic:" with a text input containing "IoTAlarm", "QoS:" with a dropdown menu showing "0 - At Most Once", and "Payload to publish:" with a large text area containing the word "buzzer". At the bottom left of the form is an orange button labeled "Publish".

Figure 6.20 – Publishing a buzzer message

9. After clicking the **Publish** button, we should hear our alarm sound for 5 seconds.

We have now created our first MQTT application! We have sent MQTT messages from our Raspberry Pi Pico W to and from the internet. As we can imagine, the possibilities of our application are vast. For instance, we can extend our IoT alarm system by adding more sensors such as a door/window contact sensor for security, or a temperature and humidity sensor for home climate monitoring. In the next chapter, we will add monitoring and remote arming of our alarm module as we continue to build our IoT home security system.

To finish off our IoT alarm module, we will now install our components into a custom 3D-printed case.

## Building an IoT alarm module case

As we have done previously, we will install our components into a custom-designed 3D-printed case. *Figure 6.21* features a rendering of our alarm module case, designed to accommodate the PIR sensor, buzzer, LED with a resistor, and the Raspberry Pi Pico W.

In the interest of compactness, we've opted for the Raspberry Pi Pico W without headers, simplifying component fitting and wire soldering. It's important to note that this choice is optional, and we can use the header-equipped Raspberry Pi Pico WH:



Figure 6.21 – Alarm module custom case

We will start by identifying the parts before we move on to constructing and then testing our IoT alarm module.

### Identifying the parts of the custom case

In *Figure 6.22*, we can see the parts required to assemble the custom case for our IoT alarm module:



Figure 6.22 – Parts for the alarm module

Let's break down each part:

- **Raspberry Pi Pico W (A)**: Header version (shown here), or header-less version (preferred).
- **Backplate (B)**: 3D printed using **Polylactic Acid (PLA)**, **Acrylonitrile Butadiene Styrene (ABS)**, or **Polyethylene Terephthalate Glycol (PETG)**.
- **Hook (C)**: 3D printed using PLA, ABS, PETG, or engineering-grade resin using a liquid resin printer (as shown here). For **Fused Deposition Modeling (FDM)** printers, the part should be printed on its side with supports for strength.
- **SFM-27 active buzzer (D)**: Case designed to fit this buzzer.
- **Side mount stand (E)**: 3D-printed optional stand for mounting alarm module on a wall. May be printed in PLA, ABS, PETG, or engineering-grade resin using a liquid resin printer (as shown here).

#### Printing the split stand with FDM printers

The split stand in the SenseHAT case files (Build Files folder, *Chapter 1* repository) is ideal for FDM printing. By splitting and printing each half on its side, the stand gains significant strength. An accompanying base is also provided.

- **LED with resistor (F)**: Refer to *Chapter 3* for construction.
- **LED holder (G)**: To hold LED into the case.
- **Front shell (H)**: 3D printed using PLA, ABS, or PETG.
- **HC-SR501 PIR sensor (I)**: Case designed to fit this PIR sensor.
- 6 x M2 5 mm screws (not shown).
- 2 x M4 10 mm bolts (not shown).
- 2 x M4 nuts (not shown).
- 4 x M3 10 mm bolts (not shown).
- Glue gun with a glue stick (not shown).

#### What is an engineering-grade resin?

Liquid resin 3D printers function by creating shapes layer by layer using UV light to solidify liquid resin. Standard resins are typically used for small artistic prints, offering excellent detail but often resulting in brittle parts. Engineering resins such as Siraya Tech Blu, on the other hand, provide superior strength, making them suitable for functional components. Parts C and E in *Figure 6.22* were printed with an 80–20 mix of standard resin and Siraya Tech Tenacious, giving the parts more flexibility and reducing their brittleness.



Files for the 3D-printed parts are in the `Build Files` directory of this chapter's GitHub repository.

Now that we have identified the parts needed to construct the case for our alarm module, let's put it together.

## Building the alarm module case

Figure 6.23 illustrates the steps to build the IoT alarm module case:

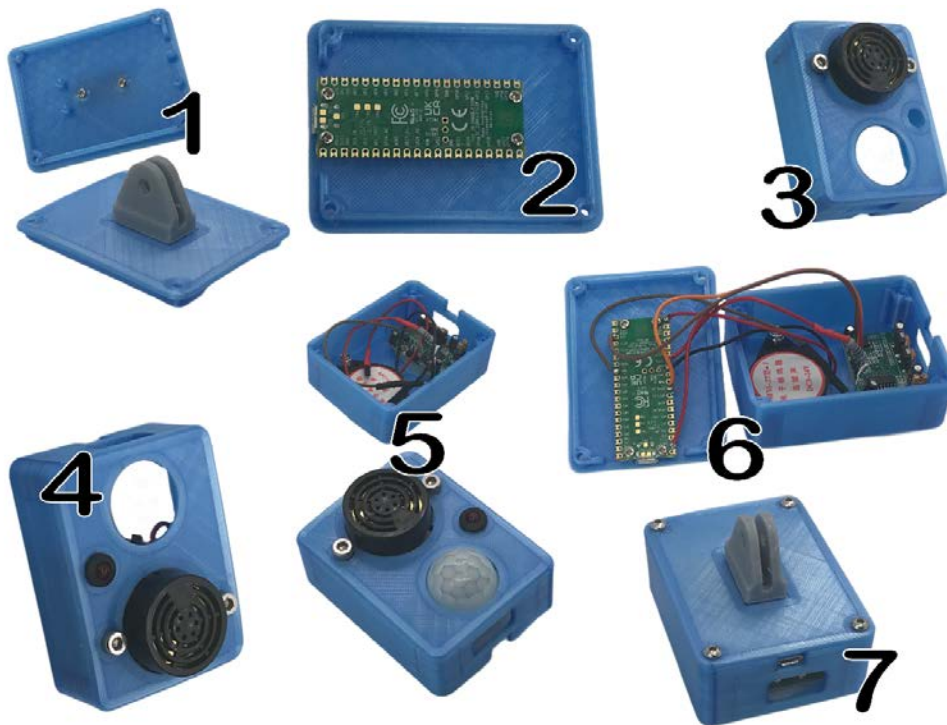


Figure 6.23 – Steps to build the alarm module case

To build the case, we do the following (the letters representing the different components in Figure 6.23 are mentioned beside the respective component's name in the following steps):

1. Using two M2 5 mm screws, we screw the hook (C) onto the backplate (B) (Figure 6.23, Step 1). We may also use epoxy glue for this.
2. Using four M2 5 mm screws, we secure the Raspberry Pi Pico W to the backplate (B) such that the USB port is facing down and toward the opening (Figure 6.23, Step 2).

3. We secure the buzzer (*D*) to the front shell (*H*) with two M4 10 mm bolts and M4 nuts (*Figure 6.23, Step 3*).
4. We then install the LED with the resistor (*F*) into the front shell (*H*) using the LED holder (*G*) (*Figure 6.23, Step 4*).
5. Using the glue gun, we secure the PIR sensor (*I*) to the front shell (*H*). Ensure that the PIR controls face the opening. We may also use the glue gun to reinforce the placement of the LED (*Figure 6.23, Step 5*).
6. Using a soldering iron, we solder the components to the Raspberry Pi Pico W using the wiring diagram from *Figure 6.8* as a reference (*Figure 6.23, Step 6*).
7. Using four M3 10 mm bolts, we secure the backplate (*B*) to the front shell (*H*) (*Figure 6.23, Step 7*).
8. If not already installed, we install MicroPython and our client code onto our Raspberry Pi Pico W using Thonny.

For our design, we use the micro-USB port for both power and access to the installed programs on our Raspberry Pi Pico W. This allows us to easily update our client software, such as changing the Wi-Fi network and password or the MQTT topic used.

Also, our case provides access to the controls on our PIR sensor so that we may control the sensitivity and off time.

**What are the controls on the HC-SR501 PIR sensor?**

The HC-SR501 PIR sensor is equipped with two adjustable controls: the sensitivity control, which fine-tunes the sensor's responsiveness to motion by increasing sensitivity when turned clockwise and decreasing it when turned counterclockwise, and the time delay control, which regulates the duration of the output signal after detecting motion, with clockwise rotation extending the signal duration and counterclockwise rotation shortening it. These controls sit beside each other and may be adjusted using a small screwdriver.

To operate our IoT alarm module, we simply connect a micro-USB cable from our Raspberry Pi Pico W to a standard USB charger. The LED should blink quickly at first as a Wi-Fi connection is established, followed by a slow blink while our program connects to the MQTT broker, and finally, a solid light indicating our module is ready for use. If we decide not to print the stand from *Figure 6.22 E*, we may mount our module onto a GoPro camera mount of our choice:



Figure 6.24 – Alarm module installed on a GoPro camera mount

To test our IoT alarm module, we connect to the `mqtt.hq.com` web client and subscribe to the `IoTAlarm` topic. Passing our hands over the PIR sensor, we should see `motion` messages appear in the client. Publishing the `IoTAlarm` topic and sending `buzzer` messages should activate our buzzer for 5 seconds.

We have just built our first MQTT-based IoT alarm module, enclosed in a physical case, capable of sensing motion and activating an alarm remotely through MQTT messages. With its built-in GoPro hook, we can easily install our IoT alarm module anywhere there is a Wi-Fi connection.

## Summary

In this chapter, we explored MQTT and used it to create an MQTT-based IoT alarm module. We introduced the amazing Raspberry Pi Pico W, a microcontroller that complements our Raspberry Pi. We began by understanding MQTT's publish-subscribe model, which enables efficient and selective communication among connected devices. Additionally, we examined the significance of threading in maximizing the utilization of the Raspberry Pi Pico's dual-core processor.

We created code for connecting to Wi-Fi and MQTT servers, handling motion detection, and activating alarm components. We learned how to use callback functions to process MQTT messages.

Furthermore, we covered saving and running our code on the Raspberry Pi Pico W, making it a standalone IoT alarm system. We also 3D printed a custom case to house the PIR sensor, LED, buzzer, and Raspberry Pi Pico W.

With our MQTT-based IoT alarm module now complete, we are ready to explore further enhancements as we expand the capabilities of our IoT home security system. In the next chapter, we will build an IoT button that we will use to control our alarm module.

## Building an IoT Button

In this chapter, we will build an essential component of our IoT home security system: an IoT button. We will build two versions of this button, using different hardware bases – the M5Stack ATOM Matrix and the Raspberry Pi Pico W.

The M5Stack ATOM Matrix is a compact ESP32-based microcontroller boasting a built-in dot-matrix screen that also serves as a touch button, a design choice that significantly reduces its size, making it a highly compact solution for IoT projects. Our now familiar Raspberry Pi Pico W stands as a favored microcontroller option, noted for its versatility and seamless integration with a wide array of external peripherals.

We will build a simple version of an IoT button, starting with the ATOM Matrix, before we create a more advanced version with our Raspberry Pi Pico W:

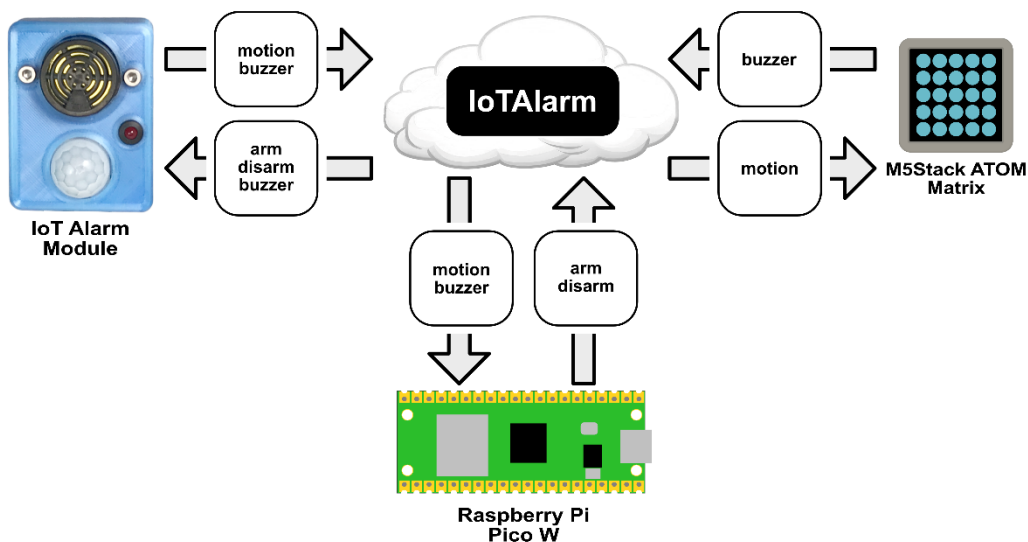


Figure 7.1 – IoT button architecture

In our design, the IoT alarm module (from *Chapter 6*), Raspberry Pi Pico W, and M5Stack ATOM Matrix all communicate using the `IoTAlarm` topic, as represented by the black box with white lettering in the cloud in *Figure 7.1*. In our graphic, we moved the topic to the cloud since we are now familiar with the MQTT protocol and no longer need to represent it visually. Messages are presented as white boxes with black lettering, and while every device can read and send any message through this topic, they are configured to filter relevant messages and transmit device-specific ones. In *Figure 7.1*, we see the specific messages each device handles as well as the difference in complexity between the implementation of our two IoT button microcontrollers.

We'll use CloudAMQP for MQTT communication with our second IoT button, ensuring efficient and reliable data transmission through its scalable messaging service. This enhances the button's performance and real-time data exchange reliability.

As we navigate the creation process of these buttons, we deepen our understanding of MQTT communication and Python programming while getting accustomed to different hardware platforms. By using two distinct bases for our projects, we equip ourselves with the insight needed to choose the best platform for our upcoming IoT applications.

We will cover the following main topics in this chapter:

- Introducing IoT buttons
- Creating our IoT button using the M5Stack ATOM Matrix
- Improving on our IoT button with the Raspberry Pi Pico W

Let's begin!

## Technical requirements

The following are the requirements for completing this chapter:

- Intermediate knowledge of Python programming
- 1 x M5Stack ATOM Matrix
- 1 x Raspberry Pi Pico WH (with headers) to use with breadboard
- 1 x Raspberry Pi Pico W (no headers) to be installed in an optional 3D-printed case
- 1 x 24 mm arcade-style pushbutton
- 1 x mini SPST switch
- 1 x 0.96-inch OLED screen
- 1 x LED connected with a 220 Ohm resistor (as previously used in *Chapter 3*)
- 1 x SFM-27 active buzzer

- 12 x M2 5 mm screws
- 2 x M4 20 mm bolts
- 2 x M4 nuts
- 4 x M3 10 mm bolts
- 1 x M5 25 mm bolt
- 1 x M5 nut
- 1 x LED holder
- Hot glue gun
- Access to a 3D printer to print optional case

The code for this chapter may be found here:

<https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter7>

## Introducing IoT buttons

Toward the end of the animated movie *WALL-E* (2008), Captain McCrea battles against the AI autopilot to regain control of the Axiom spaceship by pushing a big blue button. This results in the massive Axiom starting its hyperjump on a return to Earth. The single button in sci-fi often stands as a beacon of innovation, a dramatic tool that accentuates pivotal moments. In the case of *WALL-E*, this button surpasses a simple one-action result, such as sounding a buzzer, and instead carries the weight of humanity's future, triggering a series of actions that eventually lead to the redemption and rebirth of both mankind and Earth.

In this chapter, we will build our own powerful button, although not one with the power of the blue button from *WALL-E*. Our button will, however, introduce us to the world of IoT buttons where single actions can trigger a series of automated tasks – in our case, interacting with the IoT alarm module we built in the last chapter.

## Utilizing IoT buttons

IoT buttons are central in the IoT landscape, bridging distances with the aid of the internet to initiate actions globally. Consider the possibility of activating a machine in Mumbai with a button press from Toronto or remotely triggering an alarm to assist a dormitory-based student in waking up on time for classes. This global reach transforms simple buttons into powerful tools, making daily tasks more efficient and connected.

The following are a few examples of what we can do with IoT buttons:

- **Smart home control button:** This button facilitates control over various household appliances and systems through simple configured commands – a single or double press can control lights
- **Ordering button:** Especially useful in retail settings, this button facilitates quick orders or reorders of specific products from suppliers, enhancing business efficiency
- **Feedback button:** These buttons can be installed in corporate or service environments to accumulate immediate feedback from users or customers, helping to maintain a high standard of service
- **Conference room booking button:** In office spaces, these buttons can aid in the smooth booking of conference rooms, preventing booking conflicts and promoting efficiency
- **Smart agriculture:** These buttons can streamline processes in agriculture, with functionalities such as immediate watering of farm sections or automated feed release for livestock

In this chapter, we will use our IoT button to interact with our IoT alarm module. For this, we will build a simple button with the M5Stack ATOM Matrix and a more complex button with our Raspberry Pi Pico W.

## Exploring various technologies in IoT button development

In the fast-growing field of IoT, the choice of technology can greatly affect the functionality and adaptability of the devices we create. In this chapter, we adopt this approach as we use two different yet effective platforms – the M5Stack ATOM Matrix and the Raspberry Pi Pico W – to construct IoT buttons with varying levels of complexity.

We start with the M5Stack ATOM Matrix to build our initial button. This microcontroller is notable for its compactness, featuring an integrated dot-matrix screen that functions as a touch button. Its straightforward design not only allows for easy assembly but also supports a simple button solution, ideal for our first IoT button (see *B* in *Figure 7.2*):



Figure 7.2 – Raspberry Pi Pico W and M5Stack ATOM Matrix

Building on what we learned from the ATOM Matrix project, we then move on to leverage the capabilities of the Raspberry Pi Pico W (see *A* in *Figure 7.2*). We know this platform for its versatility and compatibility with various peripherals. This offers us more customization in button development. The Raspberry Pi Pico W enables us to create a button with a higher degree of functionality than the M5Stack ATOM Matrix.

Working with two different technological bases not only broadens our understanding but also encourages a flexible approach to IoT project development. This progression from a simple to a more advanced button is designed to help us steadily build on our knowledge, equipping us with the insight to choose the right platform for future IoT projects.

Let's get started.

## Creating our IoT button using the M5Stack ATOM Matrix

In *Figure 7.1*, we depict the ATOM Matrix on the right-hand side, receiving a `motion` input message and issuing a `buzzer` output message. These messages are associated with **passive infrared (PIR)** sensor detections and buzzer activation on the IoT alarm module, respectively. Utilizing its integrated dot-matrix screen, which also serves as a touch button, we will use the M5Stack ATOM Matrix to create our first IoT button.

Before we initiate the setup and programming of the ATOM Matrix, let's take a moment to familiarize ourselves with the range of products M5Stack offers.

### Exploring M5Stack devices

M5Stack is known for its stackable development kits, suitable for hobbyists and professionals alike. Based on the ESP32 microcontroller, these kits offer functionality and scalability, crucial for IoT, AI, and robotics projects. M5Stack's modules provide easy integration of various functionalities, coupled with a user-friendly development environment.



In *Figure 7.3*, we see a photo of various M5Stack devices:

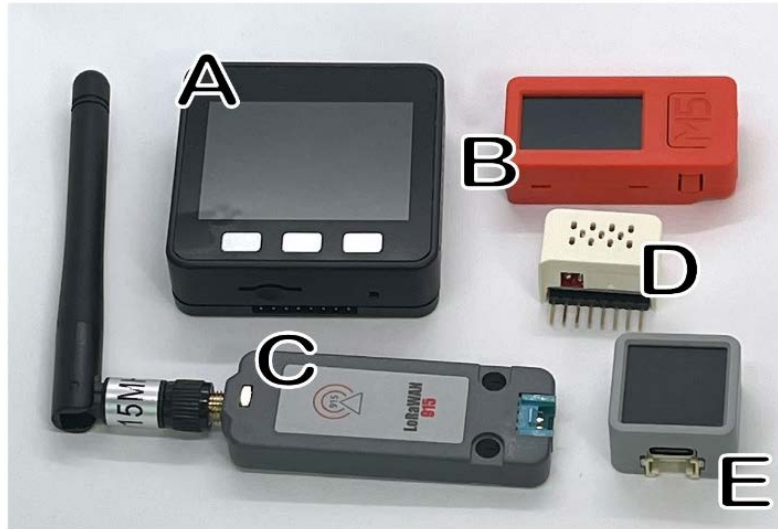


Figure 7.3 – M5Stack devices

Let's see what the devices do:

- **M5Stack Basic (A):** The M5Stack Basic is an all-encompassing central controller for IoT applications, powered by the Espressif ESP32 chipset housing two Xtensa® 32-bit LX6 microprocessors with a 240 MHz peak frequency. It provides a rich array of development interfaces, including **Analog-to-Digital Converter (ADC)**, **Digital-to-Analog Converter (DAC)**, and **Inter-Integrated Circuit (I2C)**, alongside 15 Input/Output (IO) pins. It features a 2.0-inch HD **in-plane switching (IPS)** display panel, accompanied by a speaker and microSD card slot.
- **M5StickC PLUS (B):** The M5StickC PLUS is powered by ESP32-PICO-D4 with Wi-Fi. It offers a large 1.14-inch screen with 135 x 240 px resolution. The board houses infrared, **Real-Time Clock (RTC)**, microphone, and LED, and features a robust 120 mAh battery. It supports HAT and Unit product families.
- **M5Stack Unit LoRaWAN915 (C):** The M5Stack Unit LoRaWAN915 is a **LoRaWAN** (short for **Long Range Wide Area Network**) module designed for 915 MHz frequency communications, leveraging the ASR6501 chipset to enable long-distance connections while maintaining low power usage and high sensitivity.
- **ENV III HAT (D):** The ENV III HAT is a versatile environmental sensor compatible with the M5StickC series, housing the SHT30 and QMP6988 to measure temperature, humidity, and atmospheric pressure.

- **ATOM Matrix (E):** The ATOM Matrix is M5Stack's most compact development board at 24\*24 mm, offering an extensive range of GPIO pins for compact embedded device projects. Powered by the ESP32-PICO-D4 chip, it integrates Wi-Fi technologies and 4 MB of **SPI** ( short for **Serial Peripheral Interface**) flash memory. The board features a 5\*5 RGB LED matrix, an infrared LED, a programmable button for added input support, and a built-in **Inertial Measurement Unit (IMU)** sensor (MPU6886).

For our first IoT button, we will be using the ATOM Matrix. While we could have opted for the Basic, M5StickC PLUS, or the newer ATOMS3 (not shown), as all these devices provide a pushbutton and a screen for feedback, we chose the ATOM Matrix because it offers a unique blend of compactness and simplicity, making it ideal for this introductory project. Furthermore, its integrated dot-matrix screen, doubling as a touch button, presents a more intuitive and interactive experience for users.

M5Stack provides intuitive tools to set up and program our ATOM Matrix. We'll begin with the burner tool to configure our Matrix.

## Flashing the firmware to our ATOM Matrix

The M5Burner allows us to flash firmware onto our M5Stack devices. This tool simplifies the process the process for us.

To use the tool, we do the following:

1. We download the installation file from the M5Stack website using this URL: <https://docs.m5stack.com/en/download>.
2. For our project, we'll download and install the Windows version of the burner. Once installed, we'll run the program and click on the **ATOM** tab on the left, and we should see the following screen:



Figure 7.4 – M5Burner ATOM screen

3. The **UIFlow\_MATRIX** firmware is designed for the ATOM Matrix, enabling drag-and-drop graphical programming that translates to MicroPython. It features built-in libraries and offers **over-the-air (OTA)** updates for wireless programming. We click on the **Download** button to download the firmware onto our local computer.

4. Once the firmware has finished downloading, we click on the **Burn** button (formally the **Download** button) to start flashing the **UIFlow\_MATRIX** firmware onto our ATOM Matrix. We should see a dialog requesting our Wi-Fi information.
5. By clicking the top-right blue button, we can auto-fill our computer's Wi-Fi details or enter them manually. Once entered, we click **Next** to continue. We should see the **Burn** screen next.
6. We choose the port where the ATOM Matrix is connected and click the **Start** button to initiate the firmware burn. A progress screen will then display the ongoing burn status.
7. Upon successful completion, we click on the green **Burn successfully, click here to return** button:

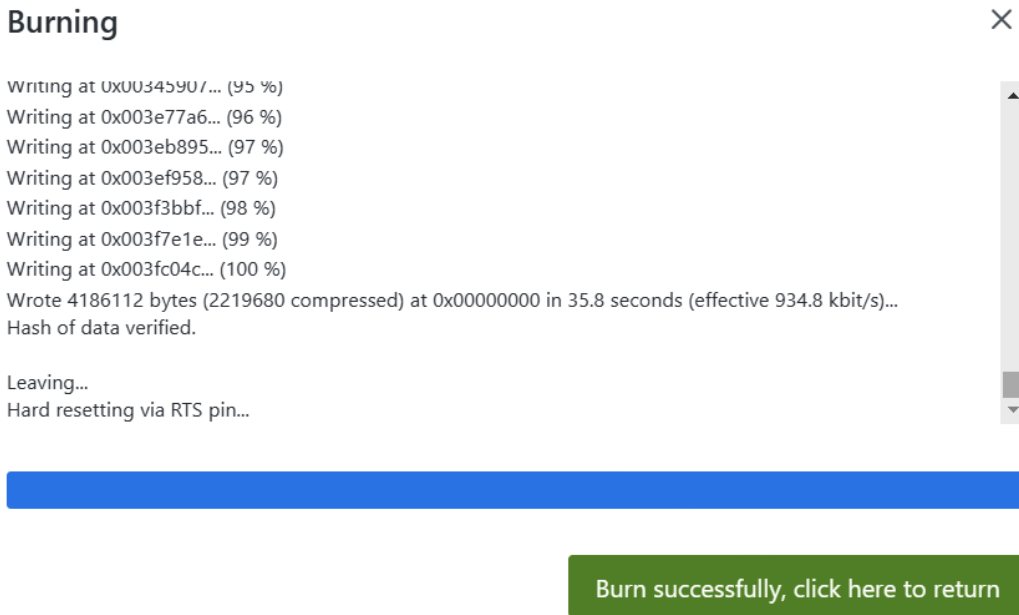


Figure 7.5 – Burn successful

We now have successfully installed the **UIFlow\_MATRIX** firmware onto our ATOM Matrix. We should observe that the dot-matrix screen on our device blinks green. This firmware comes with dedicated libraries that we will use to create our first IoT button.

With the firmware installed, it is now time to configure our ATOM Matrix so that we can start programming it.

## Configuring the ATOM Matrix for programming

To program our ATOM Matrix, we'll use M5Stack's UIFlow, specifically its Python environment. UIFlow serves as an online IDE, connecting to our devices through an API key. For successful development, it's essential to retrieve this key and properly configure our ATOM Matrix.

To configure our ATOM Matrix, we do the following:

1. In the **M5Burner** window, we click on the **ATOM** tab.
2. We click on the **Configure** button in the **UIFlow\_Matrix** section:



Figure 7.6 – UIFlow Matrix Configure button

3. This should open the **UIFlow Configuration** dialog, and the port to which our ATOM Matrix is connected should be displayed. We click on **Load** to proceed to the next screen.
4. This will bring us to the next screen, **UIFlow Configuration**:

UIFlow Configuration

×

|               |                  |   |
|---------------|------------------|---|
| COM           | COM8             | ▼ |
| MAC           | 24a1604600ac     |   |
| APIKEY        |                  |   |
| Start Mode    | Internet Mode    | ▼ |
| Boot Menu     | Show             | ▼ |
| Boot Beep     | On               | ▼ |
| Server        | flow.m5stack.com | ▼ |
| WIFI SSID     | MyNetwork        | 🖥 |
| WIFI Password | .....            | 👁 |
| COM.X         | True             | ▼ |
| APN           |                  |   |

< Cancel

💾 Save

Figure 7.7 – Main UIFlow configuration screen

5. Using *Figure 7.7* as a reference, we'll focus on the parameters highlighted by the boxes. Starting with **COM**, it should be set to the port where our ATOM Matrix is connected.
6. The **APIKEY** parameter serves as the connection bridge between the UIFlow IDE and our device. We take note of this key and keep it easily accessible.
7. **Start Mode** determines the device's startup behavior. We configure it to **Internet Mode** for connectivity with the UIFlow IDE. After programming our ATOM Matrix, this mode will switch to **App Mode** automatically. To re-connect with the IDE after that, we'll need to revisit **M5Burner** and reset this to **Internet Mode**.
8. We set the **WIFI SSID** and **WIFI Password** parameters if they are not already set to the correct values.
9. We click on the blue **Save** button to save the parameters to our ATOM Matrix.

With our ATOM Matrix configured, we're set to start programming and turn the device into our first functional IoT button.

## Turning our ATOM Matrix into an IoT button

We will use M5Stack's UIFlow online IDE for development as it provides a straightforward coding platform for M5Stack devices. We will make use of the API key that we recorded from the configuration of our device to connect to the IDE.

To add MicroPython code to our ATOM Matrix, we do the following:

1. In an internet-enabled browser, we navigate to the following URL: `https://flow.m5stack.com`.
2. We will be presented with a screen to choose either UIFlow1.0 or UIFlow2.0. Since we are using an ATOM Matrix, we choose UIFlow1.0 and click on **Confirm**.
3. In the IDE, we click on the `</>` **Python** tab so that we can program our ATOM Matrix in MicroPython:

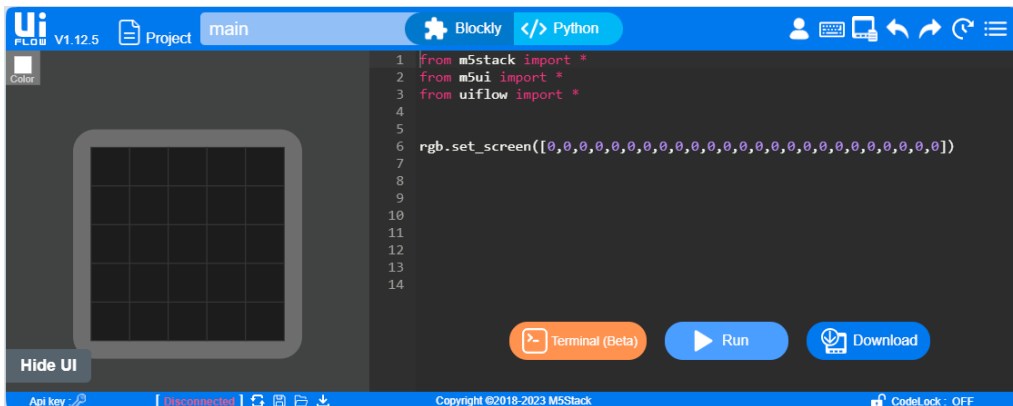


Figure 7.8 – UIFlow IDE

4. To link our ATOM Matrix to UIFlow, we click the **Api key** label at the bottom left of the screen to open the **Setting** screen:

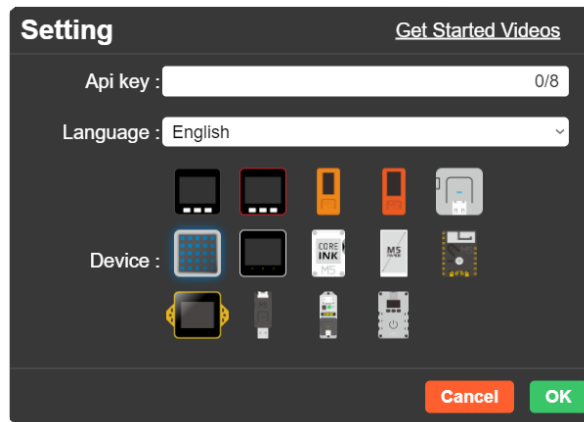


Figure 7.9 – Setting screen

5. We enter the API key for our device and click **OK** to connect our ATOM Matrix to the UIFlow IDE.
6. In the code editor, we overwrite the existing code with the following:

```
from m5stack import *
from m5ui import *
from uiflow import *
from m5mqtt import M5mqtt

import time

rgb.setColorAll(0x000000)

def cb_IoTAlarm(topic_data):
 if topic_data == 'motion':
 rgb.setColorAll(0xff0000)
 wait(5)
 rgb.setColorAll(0x00cccc)

def buttonA_pressFor():
 m5mqtt.publish(str('IoTAlarm'), str('buzzer'), 0)

btnA.pressFor(1, buttonA_pressFor)

m5mqtt = M5mqtt(
 'IoTMatrix',
```

```

 'broker.mqtthq.com',
 1883,
 '',
 '',
 300
)
 m5mqtt.subscribe(str('IoTAlarm'), cb_IoTAlarm)
 rgb.setColorAll(0x00cccc)
 m5mqtt.start()

```

Before downloading the code to our ATOM Matrix, let's break it down. We start with our imports:

- `from m5stack import *`: Imports all functions and classes from the M5Stack library.
- `from m5ui import *`: Imports all UI-related functions and classes for M5Stack.
- `from uiflow import *`: Imports all UIFlow-specific functions and classes.
- `from m5mqtt import M5mqtt`: Imports the M5mqtt class, which allows for MQTT communication.
- `import time`: Imports the standard Python time library.

7. We then set our screen to black with the following command:

```
rgb.setColorAll(0x000000)
```

8. We define our callback function as `cb_IoTAlarm(topic_data)`. This does the following:

- I. Checks if the incoming message under the `IoTAlarm` topic is `motion`.
- II. If `motion` is received, the RGB LED color is set to red (`0xff0000`) for 5 seconds.
- III. Afterward, the RGB LED color is reset to a cyan color (`0x00cccc`).

9. We then define a function to scan for a press of button A (the screen) and call it `buttonA_pressFor()`.

In this method, we publish a message with the `IoTAlarm` topic and `buzzer` payload when button A is pressed for 1 second.

10. We then create an instance of the M5mqtt class we call `m5mqtt` with the given parameters:

- The device name is set to `IoTMatrix`.
- The MQTT broker address is set to `broker.mqtthq.com`.
- The MQTT port is set to 1883.
- We provide empty strings for username and password as none are required.
- We set the MQTT keep-alive time set to 300 seconds.

11. Our code then subscribes to the `IoTAlarm` topic and sets the callback function with the following code:

```
m5mqtt.subscribe(str('IoTAlarm'), cb_IoTAlarm)
```

12. We then set the RGB LED matrix color to cyan as an initial state:

```
rgb.setColorAll(0x00cccc)
```

13. The final line starts the MQTT client, enabling it to send and receive messages:

```
m5mqtt.start()
```

14. With the code in place, we download it to our ATOM Matrix by clicking on the blue **Download** button located on the lower right-hand side of the screen.

With the code loaded onto our ATOM Matrix, we are now ready to test it.

## Testing our IoT button

For our initial tests, we'll use the MQTTTHQ web client, as previously covered in *Chapter 6*, before testing our IoT button on our IoT alarm module.

To do this, we do the following:

1. In a browser, we navigate to the following URL: `https://mqttthq.com/client`.
2. In the **Publish to topic:** section, we set the topic to `IoTAlarm`.
3. In the **Payload to publish:** section, we type in the `motion` message and press the orange **Publish** button.
4. On our ATOM Matrix, we should observe that our screen turns red for 5 seconds before returning to its initial color.
5. In the **Subscribe to topic:** section of the MQTTTHQ web client, we subscribe to the `IoTAlarm` topic.
6. On our ATOM Matrix, we press and hold down the main button (the screen) for 1 second before releasing.
7. In the **Subscribe to topic:** section, we should observe a `buzzer` message.
8. With the successful completion of our MQTTTHQ tests, we are now ready to test our IoT button on our IoT alarm module. Using a micro-USB cable, we plug the IoT alarm module we created in *Chapter 6* into a USB power brick.
9. After initializing, we wave our hand in front of the PIR sensor and observe the screen on our ATOM Matrix turn red for 5 seconds.
10. Pressing and holding the primary button on our ATOM Matrix for a second and then releasing should trigger the buzzer on our IoT alarm module.



Congratulations are in order as we have just created our first IoT button using the M5Stack ATOM Matrix! As the connection between our IoT alarm module and our IoT button is through the internet, we may place either device anywhere in the world and have them communicate.

Even though our ATOM Matrix comes in a convenient form factor, as do all M5Stack controllers, we do have a custom stand that we may mount it with (*Figure 7.10*). The 3D printer files are in the `Build Files` folder of this chapter's GitHub repository:



Figure 7.10 – M5Stack ATOM Matrix stand

To use the stand, we position the ATOM Matrix (see *B* in *Figure 7.10*) into the stand's cup section (see *A* in *Figure 7.10*) with the USB-C port facing the stand's base. The stand features a rear hole, accommodating an M2 5 mm screw (not shown) to anchor the ATOM Matrix, which has a 2 mm mounting hole at the back. The stand is designed to accommodate a USB-C cable or adapter at a 90-degree angle. Additionally, it can be mounted over an opening to conceal the USB-C cable, making it suitable for ceilings or shelves.

With our first IoT button completed, we are now ready to build the more complex second version of our button.

## Improving on our IoT button with the Raspberry Pi Pico W

As our IoT alarm system becomes more complex, the limitations of the public MQTTHQ server become increasingly evident. Given that it's a public platform, its reliability can be inconsistent. Transitioning to a reliable, private server would significantly enhance our development process and system dependability.

In this section, we will build an improved IoT button using a Raspberry Pi Pico W, a buzzer, an arcade-style pushbutton, a switch, an LED, and an OLED screen (*Figure 7.11*). We're enhancing our project's reliability and efficiency by moving to a private MQTT server using CloudAMQP.



Figure 7.11 – Enhanced IoT button

While using a private server is optional, it stands as a significant upgrade over continuing with the public MQTTHQ server. The code in this section will still work with the public MQTTHQ server (with configuration changes); however, by opting for CloudAMQP, we will improve the reliability and security of our IoT alarm system.

We will start by setting up an MQTT instance with CloudAMQP.

## Setting up a CloudAMQP instance

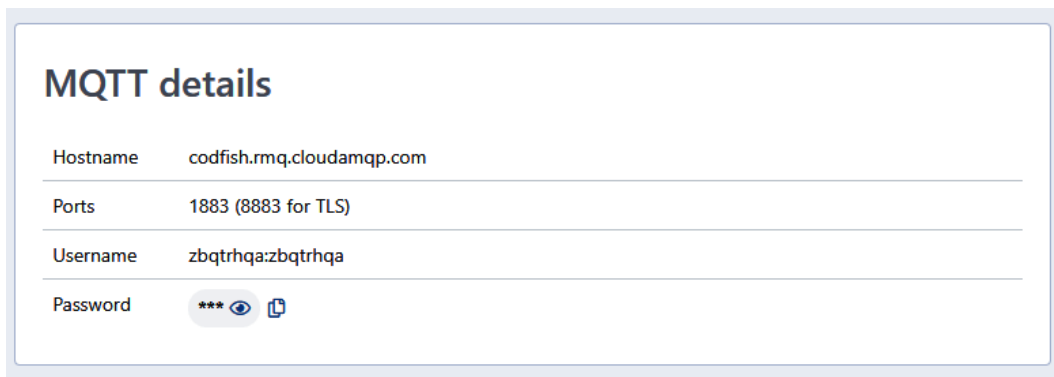
CloudAMQP is a managed MQTT service optimized for IoT devices and applications. It ensures reliable, real-time messaging with features such as WebSockets and retained messages. With its intuitive interface, CloudAMQP caters to both hobbyists and enterprises and makes an excellent choice for our IoT alarm system.

We may view the pricing of the services here – <https://www.cloudamqp.com/plans.html>

For our purposes, the free **Little Lemur** service will suffice. Upon setting up an account, we create an instance to use with our project. To do so, we do the following:

1. Logging in to our account will bring up the **Instances** page. To create a new instance, we click on the green **Create New Instance** button.
2. This will take us to the **Select a plan and name** page. In the appropriate boxes, we fill out the name of our instance, the plan we are using, and any tags we may wish to associate our instance with. For our example, we call our instance `IoTProgrammingProjects`, set our plan to `Little Lemur`, and leave the **Tags** field blank. We click on the green **Select Region** button to go to the next screen.

3. This takes us to the **Select a region and data center** screen where we select a nearby data center. For our example, we select `CA-Central Canada-1 (Canada)` under **AWS**. We click on the green **Review** button to go to the next screen.
4. In the **Confirm new instance** screen, we review our instance settings.
5. To create our instance, `IoTProgrammingProjects`, we click on the green **Create instance** button.
6. To get more details on our instances, we click on the link in the **Name** field. In our example, this is the `IoTAlarmSystem` link.
7. Clicking the link provides our instance details, which we'll use to connect our applications to the MQTT server:



The screenshot shows a web interface titled "MQTT details". It contains a table with the following information:

|          |                                                                                                                                                                         |
|----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Hostname | codfish.rmqs.cloudamqp.com                                                                                                                                              |
| Ports    | 1883 (8883 for TLS)                                                                                                                                                     |
| Username | zbqtrhqa:zbqtrhqa                                                                                                                                                       |
| Password | ***   |

Figure 7.12 – MQTT connection details

Having created our instance and obtained the details, we can now update the alarm module code from *Chapter 6* to integrate with the new MQTT server and enhance its functionality.

## Modifying our alarm module code

We will modify the code from *Chapter 6* for our alarm module so that it will work with our more advanced second IoT button. Our revised code shifts from a public MQTT server to the more dependable CloudAMQP private server.

To support this, our new code adds `USERNAME`, `MQTT_PASSWORD`, and `DEVICE_ID` authentication parameters to enhance security. Another significant update is the **ARMED** mode, which lets us arm or disarm the system via MQTT. The modified `motion_handler()` function, when detecting motion, considers the **ARMED** status, sounding the buzzer.

Finally, our revised code alters the LED feedback system. Beyond showing Wi-Fi and MQTT connection statuses, the LED also indicates if the alarm is armed by blinking very slowly.

The new version of our IoT alarm module code may be found under the `CloudAMQP` folder of this chapter's GitHub repository. To install the code onto the IoT alarm module, we follow the steps outlined in *Chapter 6*.

## Building our Raspberry Pi Pico W IoT button

The second IoT button features significant advancements in both the hardware and software as we make use of the Raspberry Pi Pico W. The Raspberry Pi Pico W and the M5Stack ATOM Matrix ESP32 both stand out as impressive microcontrollers. The Pico W is notable for its dual-core ARM Cortex-M0+ processor and Wi-Fi capabilities, whereas the ATOM Matrix brings both Wi-Fi and Bluetooth to the table with its ESP32 chip. However, considering our second IoT button project values computational strength and Wi-Fi above all, we will go with the Raspberry Pi Pico W.

The inclusion of an OLED screen provides us with the status of the alarm, as well as acting as a monitor for our MQTT messages. Software-wise, the shift from a public MQTTHQ server to CloudAMQP's private server improves reliability and security. This move reduces risks linked to public servers.

### *Looking at the components for our Pico W IoT button*

For the second version of our IoT button, we will be using an arcade-style button to send a message to arm the alarm module. An OLED screen will display MQTT messages sent back from the IoT alarm module. A buzzer message sent from the IoT alarm module will start a melody on the active buzzer that makes up the second version of the IoT button assembly. To disarm the IoT alarm module, we simply toggle a switch from its current position.

In *Figure 7.13*, we see the components that make up our Raspberry Pi Pico W IoT button:

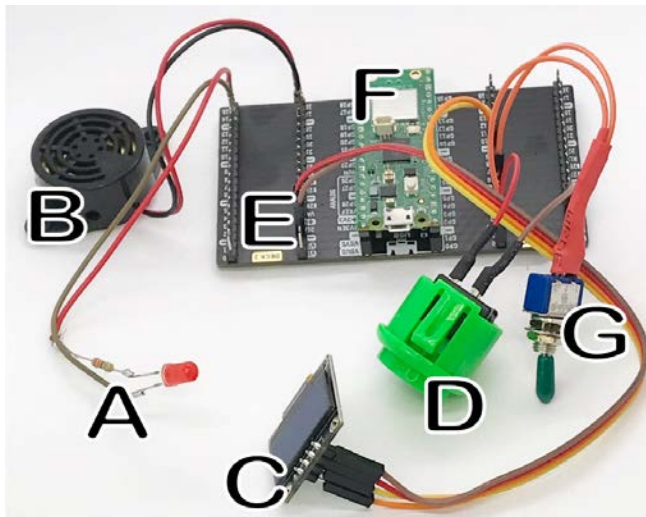


Figure 7.13 – IoT button V2 circuit arranged on a Pico GPIO expander

The components are the following:

- A: Single color LED with 220 Ohm resistor
- B: SM-127 active buzzer
- C: 0.96-inch OLED screen
- D: 24 mm arcade button
- E: GPIO expander board for the Raspberry Pi Pico (optional)
- F: Raspberry Pi Pico WH for development usage
- G: SPST switch

For development, we will use a Raspberry Pi Pico WH but switch to the non-header version for installation into the custom case.

#### **Using a GPIO expander**

The use of a GPIO expander (see *E* in *Figure 7.13*) is optional. To use the expander, female jumper connections are required. Using a GPIO expander offers the benefit of easily transitioning components to installations with the Pico WH in future projects, thanks to its female jumper connections.

We will start our construction of the Raspberry Pi Pico W IoT button by examining the wiring diagram on a standard breadboard.

### ***Wiring up our Raspberry Pi Pico W IoT button***

In *Figure 7.14*, we can see a wiring diagram for the Raspberry Pi Pico W IoT button. We will use a standard micro-USB cable connected to the USB port of the Pico WH to provide power. The 3.3 V pin from the Pico W is used to provide power to the rails of the breadboard:

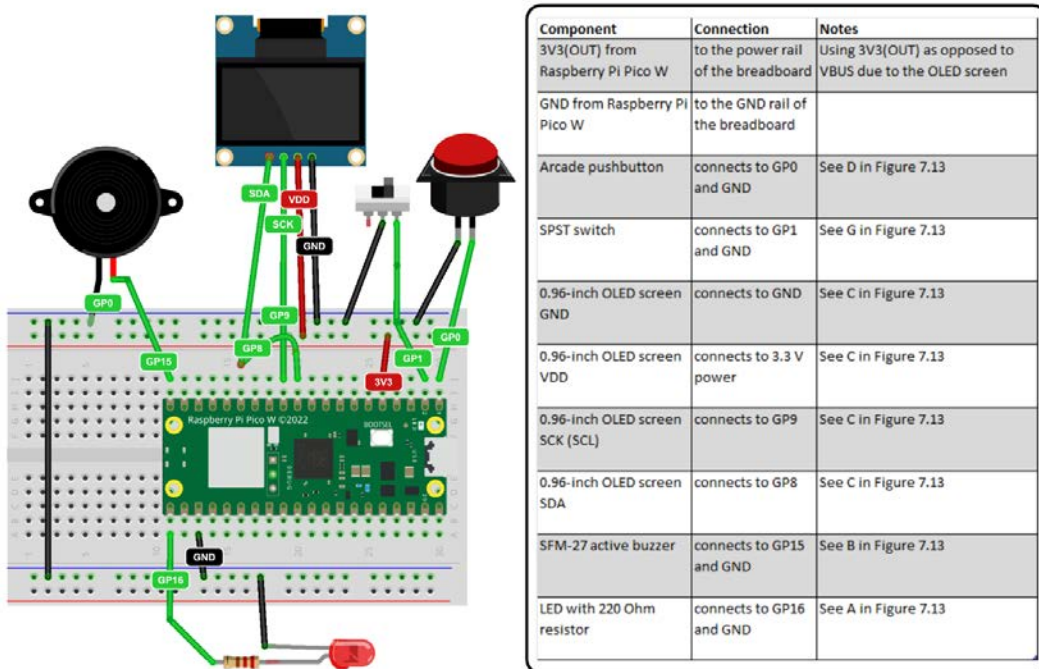


Figure 7.14 – Raspberry Pi Pico W IoT button wiring diagram

Now that our wiring is set, we'll move on to the coding phase. We will start by loading the necessary packages we require for our code.

### Setting up the required packages for our program

The packages required to run our code for our Raspberry Pi Pico W IoT button are the `micropython-umqtt.simple` package and the `micropython-ssd1306` package. To load the packages onto our Raspberry Pi Pico W, we do the following:

1. Using the Thonny IDE, we click on **Tools | Manage packages**.
2. In the search box, we type in the name of the package we would like to search for and then click on the **Search on PyPI** button.
3. We then proceed to install the package by clicking on the **Install** button.

**What to do if there are errors when loading packages?**

In situations where there is an error when loading a package, we may simply copy the library folder from our GitHub repository to our Pico W. The folder may be found under `Second IoT Button/library-files-from-pico-w` from this chapter's repository. See *Figure 7.15* for clarification on the file structure of the Raspberry Pi Pico W.

4. Before proceeding to write our code, we should verify that the file structure on our Pico W looks like the following:

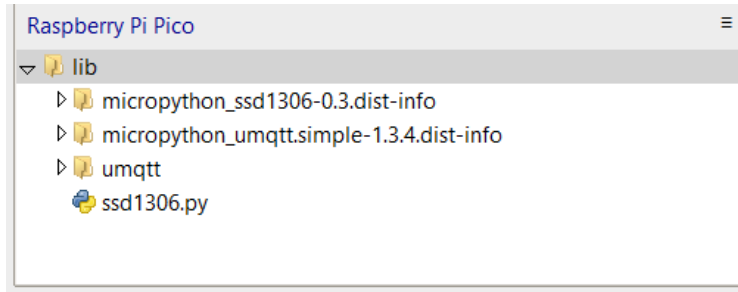


Figure 7.15 – Raspberry Pi Pico W library structure

With our packages installed, it is now time to write our code. We will start with code to control the buzzer as it will be contained in a separate file from the main code.

**Activating the buzzer**

In *Figure 7.1*, we see a buzzer message sent from the IoT alarm module. In our rewrite of the IoT alarm module code, a buzzer message is sent out when the alarm has been armed and motion is detected from the PIR sensor. We will use this message in our Raspberry Pi Pico W IoT button to activate its alarm, although we will use it to play a melody through its speaker. As we did with the IoT alarm module code, we will use a separate file for our code to activate the buzzer.

To do this, we do the following:

1. We connect our Raspberry Pi Pico W to a USB port and launch Thonny. We may use our Raspberry Pi or another operating system for this.
2. We then activate the MicroPython environment on our Pico W by selecting it from the bottom right-hand side of the screen.
3. In a new tab, we enter the following code:

```
from machine import Pin, PWM
import utime

BUZZER_PIN = 16
```

```

buzzer = PWM(Pin(BUZZER_PIN))

def play_notes(
 notes = [
 (330, 0.5), # E4 for 0.5 seconds
 (262, 0.5), # C4 for 0.5 seconds
 (330, 0.5), # E4 for 0.5 seconds
 (392, 0.5), # G4 for 0.5 seconds
 (349, 0.5), # F4 for 0.5 seconds
 (262, 1), # C4 for 0.5 seconds
]
):

 for freq, duration in notes:
 buzzer.freq(freq)
 buzzer.duty_u16(32768)
 utime.sleep(duration)

 buzzer.duty_u16(0)

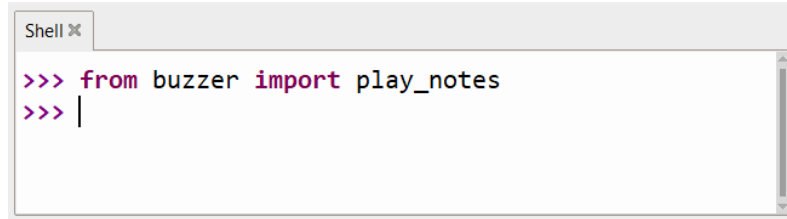
```

Before we test out our code, let's break it down:

- I. We start by importing `Pin` and `PWM` from the `machine` module, and the `utime` module.
  - II. We set the `BUZZER_PIN` constant to pin number 16, which corresponds to our wiring diagram (*Figure 7.14*).
  - III. We then initialize `buzzer` using the `PWM` class with the defined pin.
  - IV. Our `play_notes()` function takes a default argument, `notes`, which is a list of tuples. Each tuple represents a frequency in Hertz (such as E4, C4, and so on) and a duration in seconds.
  - V. For each frequency-duration pair in the `notes` list, we do the following:
    - i. We set the buzzer's frequency to the specified frequency.
    - ii. We activate the buzzer with a 50% duty cycle (`duty_u16(32768)`). This produces a square wave, defining the character of the sound emitted by the buzzer.
    - iii. We wait for the specified duration using `utime.sleep(duration)`.
  - VI. After playing all the notes, we turn off the buzzer (by setting its duty cycle to 0).
4. To save the file, we click on **File | Save as...** from the drop-down menu. We save our file as `buzzer.py` to our Raspberry Pi Pico W.



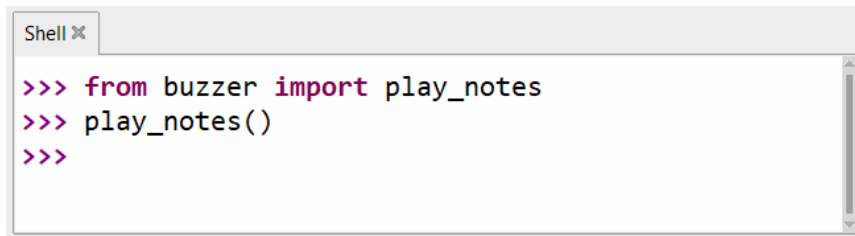
5. To test our code, we use the **Shell** in Thonny. We start by importing the `play_notes()` function from our new buzzer script:



```
Shell ✕
>>> from buzzer import play_notes
>>> |
```

Figure 7.16 – Importing the `play_notes()` function

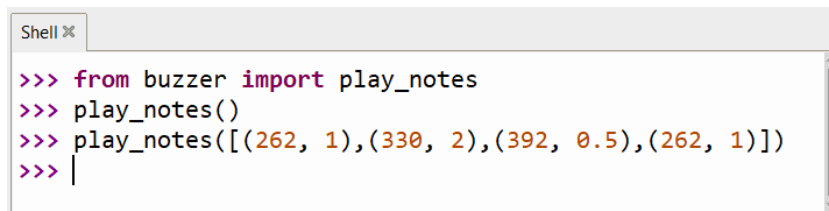
6. To activate our buzzer, we simply call the function and hit *Enter*:



```
Shell ✕
>>> from buzzer import play_notes
>>> play_notes()
>>>
```

Figure 7.17 – Running the `play_notes()` function

7. We should hear our buzzer play its default melody. To test our function further, let's send `[(262, 1), (330, 2), (392, 0.5), (262, 1)]` notes to it:



```
Shell ✕
>>> from buzzer import play_notes
>>> play_notes()
>>> play_notes([(262, 1), (330, 2), (392, 0.5), (262, 1)])
>>> |
```

Figure 7.18 – Playing a new melody with the `play_notes()` function

8. We should notice that a different melody plays from our buzzer.

With our buzzer code in place, it's now time to focus on the main code that will power our Raspberry Pi Pico W IoT button.

### ***Coding the primary functionality for our IoT button***

After finalizing the buzzer script, we're ready to develop the main code for our enhanced IoT button. This new iteration can arm the IoT alarm module and features a display screen. The screen indicates the alarm module's status (either armed or disarmed) and presents the latest MQTT message from our CloudAMQP server.

When the IoT alarm module triggers, our IoT button's buzzer provides audible feedback. It activates upon receiving a buzzer message. We use the switch on our IoT button assembly as a toggle to disarm the alarm on our IoT alarm module.

To create the code for our enhanced IoT button, we do the following:

1. We connect our Raspberry Pi Pico W to a USB port and launch Thonny. We may use our Raspberry Pi or another operating system for this.
2. We then activate the MicroPython environment on our Pico W by selecting it from the bottom right-hand side of the screen.
3. We will start with our imports. In a new tab, we enter the following code:

```
from machine import Pin, I2C
import utime
import network
from umqtt.simple import MQTTClient
from buzzer import play_notes
import ssd1306
import _thread
```

4. We then define variables with values obtained from our CloudAMQP account (*Figure 7.12*):

```
SSID = "MyWiFiNetwork"
WIFI_PASSWORD = "xxxxxxxxxxxxxx"

led = machine.Pin(15, machine.Pin.OUT)
button = Pin(0, Pin.IN, Pin.PULL_UP)
switch = Pin(1, Pin.IN, Pin.PULL_UP)
previous_switch_state = switch.value()

MQTT_SERVER = "codfish.rmq.cloudamqp.com"
MQTT_PORT = 1883
USERNAME = "xxxxxxx"
PASSWORD = "xxxxxxx"
DEVICE_ID = "IoTAlarmSystem"
last_message = ""
```

```
i2c = I2C(0, scl=Pin(9), sda=Pin(8))
display = ssd1306.SSD1306_I2C(128, 64, i2c)

mqtt_client = None
```

5. The `on_message_received()` method serves as our MQTT client's callback. By using the `global` keyword with `last_message`, we ensure updates to this variable are reflected globally throughout the code:

```
def on_message_received(topic, msg):
 global last_message
 print("Received:", topic, msg)
 if topic == b"IoTAlarm":
 last_message = msg.decode()
 if msg == b"buzzer":
 play_notes()
```

6. The `connect_wifi()` function initializes and activates Pico W's Wi-Fi. If not connected, it attempts to join the network using a predefined SSID and password. Meanwhile, an LED blinks to indicate the connection process. Upon successful connection, the LED stays on, and the device's IP address is displayed:

```
def connect_wifi():
 wlan = network.WLAN(network.STA_IF)
 wlan.active(True)

 if not wlan.isconnected():
 print('Connecting to WiFi...')
 wlan.connect(SSID, WIFI_PASSWORD)

 while not wlan.isconnected():
 led.on()
 utime.sleep(0.5)
 led.off()
 utime.sleep(0.5)

 led.on()
 print('WiFi connected, IP:', wlan.ifconfig()[0])
```

7. The `connect_mqtt()` function tries to establish an MQTT connection using predefined server details. If successful, it sets a message callback and subscribes to the `IoTAlarm` topic. If the connection fails, the function waits for 5 seconds and retries:

```
def connect_mqtt():
 global mqtt_client
```

```

while mqtt_client is None:
 try:
 print('Trying to connect to MQTT Server...')
 mqtt_client = MQTTClient(
 DEVICE_ID, MQTT_SERVER, MQTT_PORT,
 USERNAME, PASSWORD
)
 mqtt_client.set_callback(
 on_message_received
)
 mqtt_client.connect()
 mqtt_client.subscribe(b"IoTAlarm")
 print('MQTT connection established and subscribed to
IoTAlarm')
 except:
 mqtt_client = None
 print('Failed to connect, retrying...')
 utime.sleep(5)

```

8. The `display_status()` function continuously updates the display every 5 seconds. It shows the MQTT connection status at the top, the arm/disarm status in the middle, and the last received MQTT message at the bottom. If connected to MQTT, `MQTT Connected` is displayed; otherwise, `MQTT waiting` is shown. We run this method in a separate thread in our code:

```

def display_status():
 global last_message
 is_armed = False

 while True:
 display.fill(0)

 if mqtt_client:
 msg = "MQTT Connected"
 else:
 msg = "MQTT waiting"
 display.text(msg, 0, 0)

 if last_message == "arm":
 is_armed = True
 elif last_message == "disarm":
 is_armed = False
 if is_armed:

```

```
 display.text("Status: Armed", 0, 20)
 else:
 display.text("Status: Disarmed", 0, 20)

 display.text("Msg: " + last_message, 0, 40)

 display.show()
 utime.sleep(5)
```

9. The `main()` function initiates Wi-Fi and MQTT connections and continuously checks the button and switch states. If the button is pressed for over a second, an arm message is published via MQTT. If the switch state changes, a disarm message is sent. The function also checks for incoming MQTT messages. If an error occurs while checking for messages, the error is printed, and the system waits for 5 seconds before resuming checks. The system waits 0.1 seconds between loop iterations to optimize performance. We start by defining the `main()` function and variables:

```
def main():
 global last_message, previous_switch_state
 connect_wifi()
 connect_mqtt()
 button_start_time = None
```

10. Then, we define an infinite loop and check to see if the main button has been pressed for a second (1000 ms) or greater:

```
while True:
 if button.value() == 0:
 if button_start_time is None:
 button_start_time = utime.ticks_ms()
 else:
 if button_start_time is not None:
 button_elapsed_time = utime.ticks_diff(utime.
ticks_ms(), button_start_time)
 if button_elapsed_time >= 1000:
 mqtt_client.publish(
 b"IoTAlarm",
 b"arm"
)
 last_message = "arm"
```

```
button_start_time = None

current_switch_state = switch.value()
```

11. Our code then sends a disarm message if the current switch state is not equal to the previous switch state. This conditional check allows us to use our switch as a toggle and not have a defined on or off state:

```
if current_switch_state != previous_switch_state:
 mqtt_client.publish(
 b"IoTAlarm",
 b"disarm"
)
 last_message = "disarm"
 previous_switch_state = current_switch_state

try:
 mqtt_client.check_msg()
except Exception as e:
 print("Error checking MQTT message:", str(e))
 utime.sleep(5)

utime.sleep(0.1)
```

12. We then initialize a new thread to run the `display_status()` function concurrently. By using threading, it allows the `display_status()` function to operate independently and simultaneously with other parts of the program, ensuring continuous updates to the display status without hindering or waiting for other processes:

```
_thread.start_new_thread(display_status, ())
```

13. Finally, we call the `main()` function, which directs the program's core activities — handling connections, button inputs, and managing MQTT messages:

```
main()
```

14. To save the file, we click on **File | Save as...** from the drop-down menu. We save our file as `main.py` to our Raspberry Pi Pico W.

With our main code written, it is time to test it.

## Running our enhanced IoT button

Executing the code on our Raspberry Pi Pico W IoT button allows it to interact with the IoT alarm module. The button can arm or disarm the module. We will use the **MQTT Explorer** app in Windows to monitor real-time MQTT messages, as well as send messages to test our enhanced IoT button.

We will start by sending messages from the MQTT Explorer app. To do so, we do the following:

1. From the Microsoft Store in Windows, we search for the MQTT Explorer app:

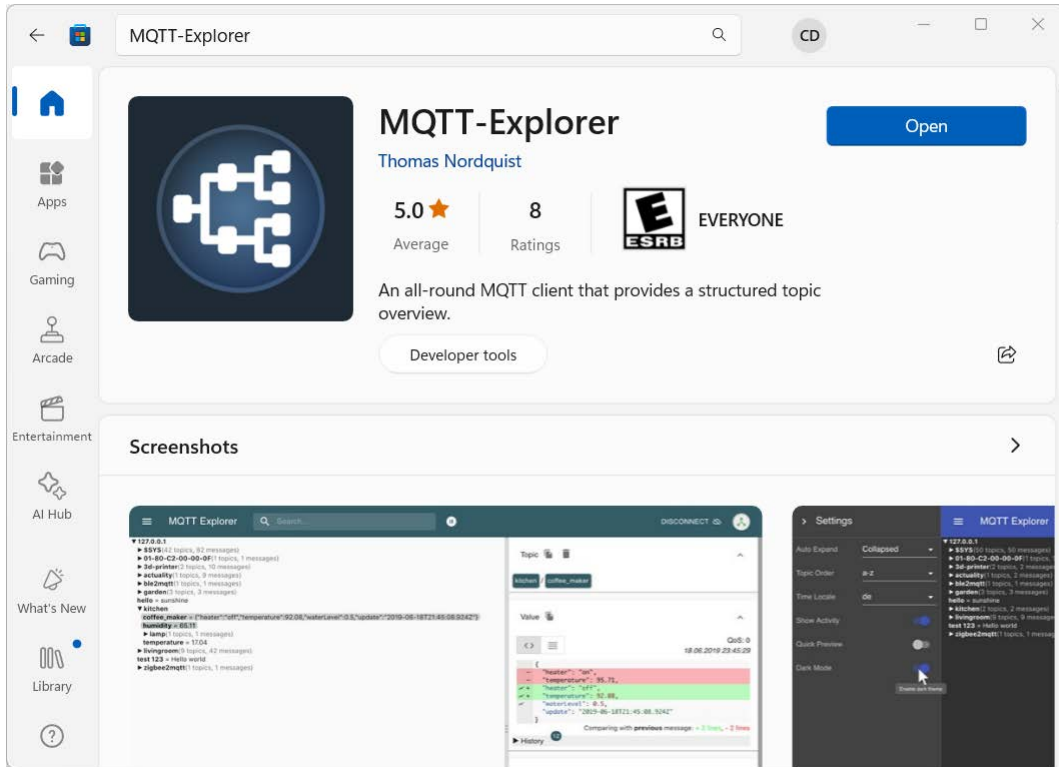


Figure 7.19 – MQTT Explorer in the Microsoft Store

2. We run `main.py` on our Raspberry Pi Pico W IoT button application by either clicking on the **Run** button in Thonny or by plugging our Pico W into a USB power supply.
3. Using MQTT Explorer, we create a connection called `IoTAlarmSystem` with the MQTT server credentials from Figure 7.12:

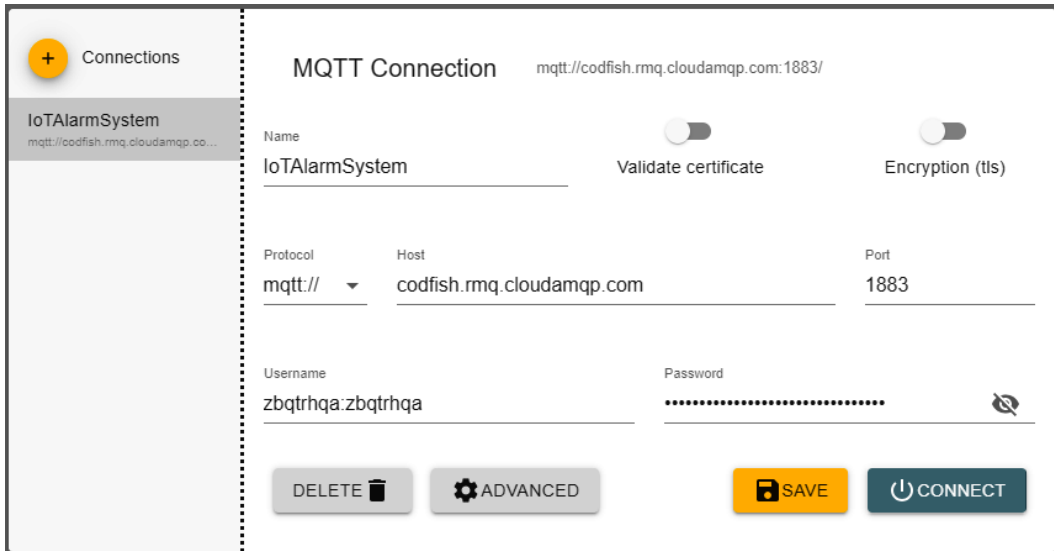


Figure 7.20 – Creating an MQTT connection using MQTT Explorer

4. To send a message using MQTT Explorer, we type in `IoTAlarm` for the topic and `test` for the message before clicking on the **PUBLISH** button:

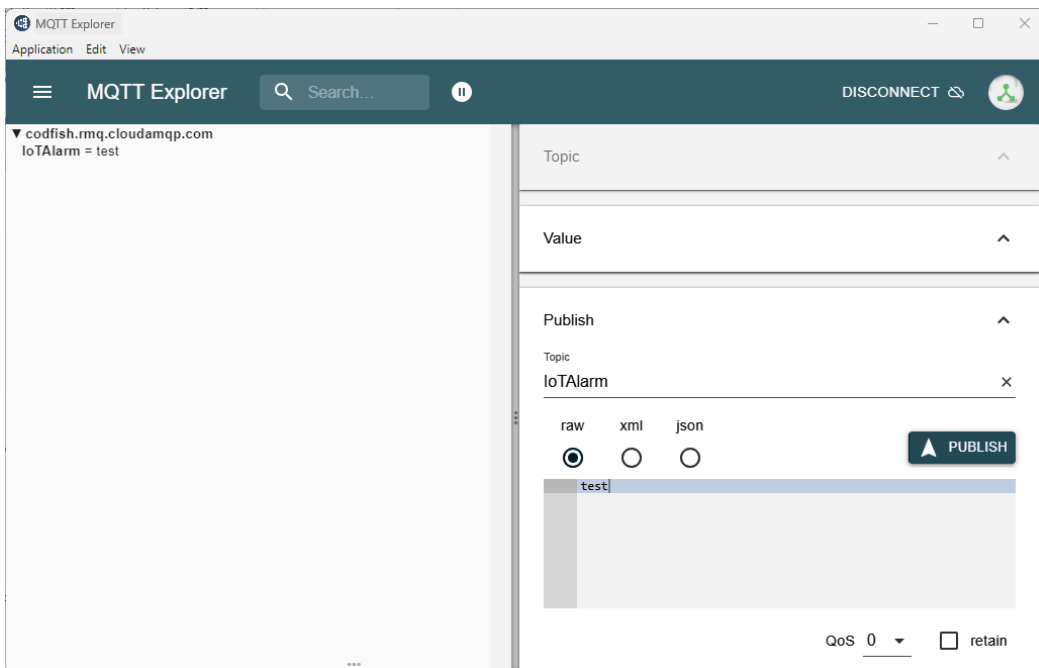


Figure 7.21 – Sending an MQTT message



5. We should observe that a test message appears on the OLED screen of our IoT button circuit.
6. We then press and hold the arcade pushbutton on the IoT button circuit for more than a second.
7. We should observe an arm message on the MQTT Explorer app.
8. We should also observe the LED on the IoT alarm module start blinking every 5 seconds. This indicates that the IoT alarm module is armed.
9. We test the IoT alarm module by waving our hands in front of the PIR sensor. We should observe that the alarm goes off.
10. We should observe that after a short time, the alarm on the enhanced IoT button circuit goes off.
11. To disarm the alarm, we toggle the switch from its current position. This should result in the buzzer turning off when motion is detected by the PIR sensor. This should also disable the buzzer on the enhanced IoT button.

We should congratulate ourselves as we have successfully built a basic IoT alarm system! For the final section of this chapter, we will install the components into the custom 3D-printed case.

## Installing the components in a custom case

Wiring our components and running our code is an exciting step. Yet, housing them in a custom case elevates our project and allows us to use our application for practical purposes. A custom case offers not only protection but also the versatility to install our advanced IoT button in any desired location.

In *Figure 7.22*, we view the components of our custom case, all 3D printed. Parts *A* and *B* were produced using a **Fused Deposition Modeling (FDM)** printer, while *C* and *D* utilized a liquid resin printer. While either printer type is suitable, FDM printing requires careful part orientation on the print bed to account for layer-line strength:

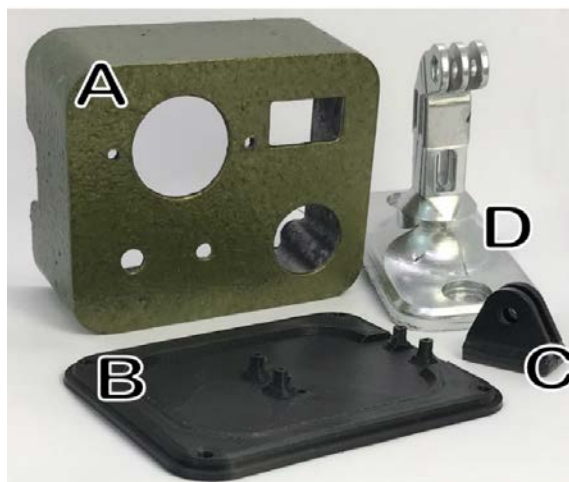


Figure 7.22 – Custom case for enhanced IoT button

The parts of our custom case are broken down as follows:

- A: Shell
- B: Backplate
- C: Hook
- D: Mounting bracket wall side mount

All parts may be found in the `Pico Button` subfolder of the `Build Files` folder in this chapter's GitHub repository. The mounting screws and LED holder are not shown.

#### Printing the split stand with FDM printers

The split stand in the SenseHAT case files (`Build Files` folder, *Chapter 1* repository) is ideal for FDM printing. By splitting and printing each half on its side, the stand gains significant strength. An accompanying base is also provided.

To install the components into our custom case, we follow the steps in *Figure 7.23*:

1. We start by installing the buzzer (see *B* in *Figure 7.13*) into the shell (see *A* in *Figure 7.22*) with two M5 10 mm bolts and two M5 nuts (*Figure 7.23, Step 1*).
2. Then (*Figure 7.23, Step 2*), we install the switch (*Figure 7.13, G*) and arcade pushbutton (see *D* in *Figure 7.13*) into their appropriate holes in the shell (see *A* in *Figure 7.22*):



Figure 7.23 – Constructing the case for the enhanced IoT button

3. Using an LED holder, we install the LED with 220 Ohm resistor (see A in *Figure 7.13*) into the shell (see A in *Figure 7.22*) using the left-side hole (*Figure 7.23, Step 3*).
4. We secure the OLED screen (see C in *Figure 7.13*) into the shell (see A in *Figure 7.22*) using 4 M2 5 mm screws or glue from a hot glue gun (*Figure 7.23, Step 4*).
5. Then (*Figure 7.23, Step 4*), we wire the components to a new Raspberry Pi Pico W.
6. We then install the hook (see C in *Figure 7.22*) onto the back plate (see B in *Figure 7.22*) using two M2 5 mm screws (*Figure 7.23, Step 5*).
7. We secure the Pico W onto the back plate (see B in *Figure 7.22*) with four M2 5 mm screws (*Figure 7.23, Step 6*).
8. Then (*Figure 7.23, Step 7*), we use four M3 10 mm bolts to secure the back plate (see B in *Figure 7.22*) to the shell (see A in *Figure 7.13*).
9. With the back plate (see B in *Figure 7.22*) secured to the shell (see A in *Figure 7.22*), we connect the assembly to the mounting bracket (see D in *Figure 7.22*) using an M5 20 mm bolt and M5 nut.
10. Using the steps from the section, *Coding the primary functionality for our IoT button*, we install the packages and client code for our enhanced IoT button.

With the components securely installed, we can now position our Raspberry Pi Pico W IoT button in any desired location, be it a home, office, or workshop setting. Our design not only serves our primary alarm system purpose but also makes it adaptable for a multitude of other applications, expanding its utility and offering potential for innovative integrations in many different scenarios.

## Summary

In this chapter, we explored IoT buttons, starting with a description of what they are and where they are used. We then explored various technologies that we could use to build an IoT button.

We then proceeded to build our first IoT button using a public MQTT service and an M5Stack ATOM Matrix. We were able to connect our IoT button to the IoT alarm module we built in *Chapter 6*.

From there, we upgraded our MQTT server to a private one using CloudAMQP. We did so for reliability and security reasons as we started to build out our IoT alarm system more. We upgraded the code sitting on our IoT alarm module before building our second IoT button using a Raspberry Pi Pico W and various components.

We finished the chapter by installing the components of our second (enhanced) IoT button into a 3D-printed case. Doing so transformed our circuit from an educational tool to a working device suitable for deployment in commercial settings.

In the next chapter, we will continue to build our IoT alarm system by going back to the Raspberry Pi, where we will build a security dashboard.

# Creating an IoT Alarm Dashboard

In today's digitalized age, IoT has revolutionized security, transforming basic alarm dashboards into comprehensive, real-time security monitors. In *Chapter 6*, we started constructing an IoT alarm system by building an IoT alarm module that could detect motion and relay MQTT messages. *Chapter 7*, introduced two versions of the IoT button: the first uses the M5Stack ATOM Matrix with an LCD matrix screen, and the second incorporates an OLED screen, buzzer, arcade-style button for arming our IoT alarm module, and a toggle switch for disarming:

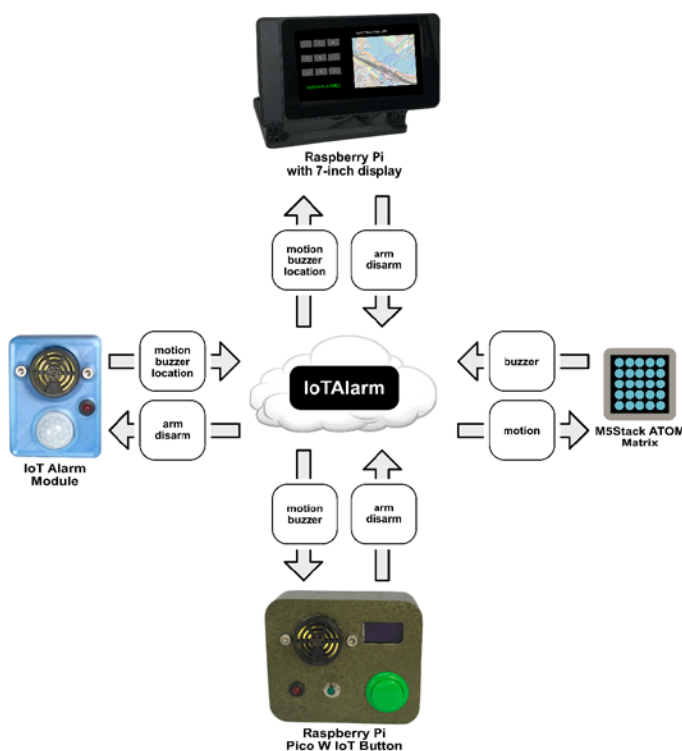


Figure 8.1 – IoT alarm system

In this chapter, we will use our Raspberry Pi 5 with a 7-inch touchscreen to serve as our IoT alarm dashboard (*Figure 8.1*). This dashboard will let us arm and disarm our IoT alarm module, review the latest MQTT notification, and observe a map pinpointing the location where our alarm was activated. It will subscribe and publish to the same MQTT topic, `IoTAlarm`, as the other IoT alarm system devices – that is, the IoT alarm module, the M5Stack ATOM Matrix IoT button, and the Raspberry Pi Pico W IoT button.

The IoT alarm dashboard completes our advanced IoT alarm system. Our system utilizes IoT technology and the internet's vast reach for global application deployment.

We will cover the following topics in this chapter:

- Exploring IoT alarm dashboards
- Creating a Raspberry Pi 5 IoT alarm dashboard
- Building the external alarm buzzer stand
- Running our application

Let's begin!

## Technical requirements

The following are the requirements for completing this chapter:

- Intermediate knowledge of Python programming
- A late model Raspberry Pi, such as the Raspberry Pi 5
- A Raspberry Pi branded 7-inch touchscreen with a compatible case
- 1x SFM-27 active buzzer
- 2x M2 5mm screws
- 2x M4 20mm bolts
- 2x M4 nuts
- 1x M5 20mm bolt
- 1x M5 nut
- Composite (multi-wire) cable with USB plug (a discarded USB charging cable works well)
- Hot glue gun
- Access to a 3D printer or 3D printer service to print an optional case

The code for this chapter can be found here:

<https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter8>

## Exploring IoT alarm dashboards

IoT's true strength is its limitless capabilities, as seen in IoT alarm dashboards. Integrating devices such as the Raspberry Pi 5 with the expansive network of IoT opens up new possibilities, especially in security systems. Using the Raspberry Pi in alarm systems, whether for homes or businesses, offers immense innovation potential. This versatile device opens up possibilities for advanced security solutions beyond traditional alarm systems.

### Using IoT alarm dashboards for industrial processes

In modern industrial settings, monitoring safety and efficiency is essential. The industrial IoT alarm dashboard, which is typically integrated into a control room, provides an overview of the facility's operations. It shows real-time metrics and system statuses and sends alarms if equipment deviates from standard parameters.

The dashboard's strength lies in its ability to detect and communicate issues quickly via the Internet. For example, in a petrochemical plant, sensors on a tank send data to the dashboard, as illustrated in *Figure 8.2*. Here, sensors on the tank publish “temp” and “level” MQTT messages to indicate the temperature of the liquid in the tank and the level, respectively. The Raspberry Pi has been set up to subscribe to these messages and passes this information onto the web interface and analog meter. If the tank's level drops or its temperature changes drastically, the system flags this discrepancy:

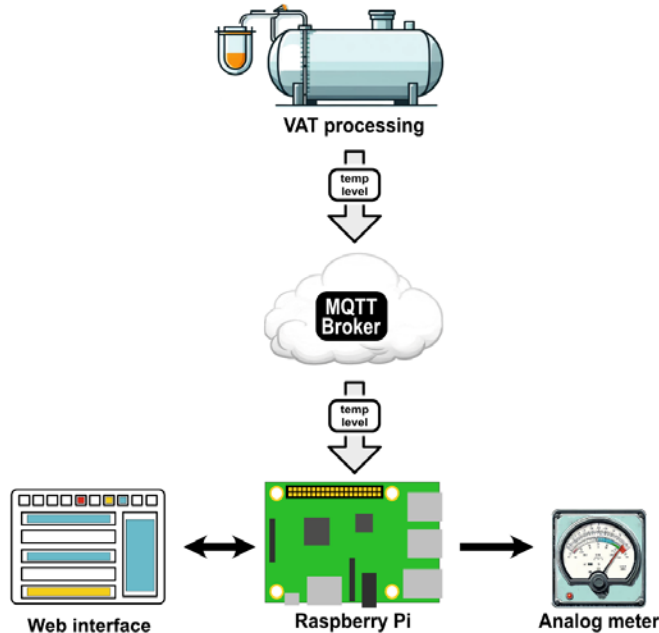


Figure 8.2 – Industrial IoT application

Alarms alert the team, allowing for prompt action to be taken to ensure worker safety and prevent potential hazards. The dashboard can also initiate automated responses, such as shutting down affected areas.

Beyond safety, the IoT dashboard enhances operational efficiency. Analyzing long-term data can help industries predict maintenance needs, reducing downtimes. The dashboard can also connect to supply systems, updating raw material levels and product counts. By using data from various sources, industries can improve safety and efficiency.

## Exploring the IoT security alarm dashboard

For modern security, the integration of IoT has redefined the capabilities of alarm dashboards. These aren't the traditional systems of the past; IoT-enhanced alarm dashboards are dynamic, offering remote access and responsive actions. For instance, with the rise of smart homes and businesses, a breach in security doesn't just trigger a loud siren but can instantly notify homeowners through their mobile devices, initiate real-time video footage capture, and even communicate with local law enforcement, all powered by IoT connectivity.

Armed with our Raspberry Pi 5 and its 7-inch touchscreen display, we will build an alarm dashboard (Figure 8.3) for our IoT alarm system. With this dashboard, we can arm and disarm our IoT alarm module using a 4-digit pin. Our dashboard will display the latest IOTALarm MQTT message and provide a map of the area where our alarm has been triggered:

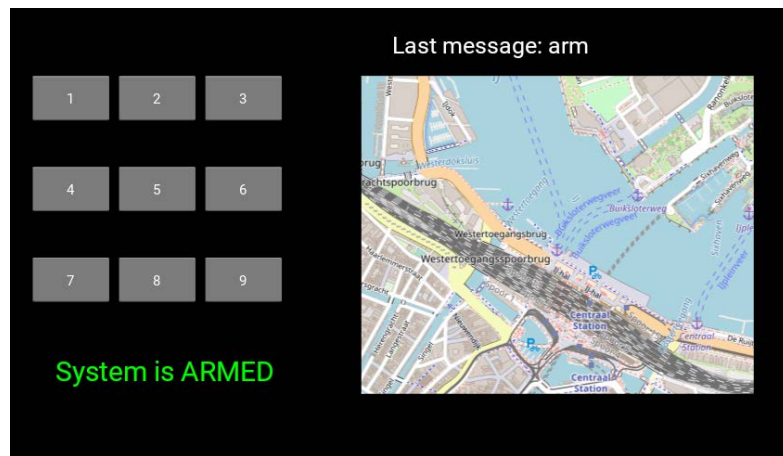


Figure 8.3 – IoT alarm dashboard

Since our Raspberry Pi 5 doesn't have a built-in buzzer, we will add an external one that we will connect via the GPIO port using a composite (multi-wire) cable. This external buzzer will sit in a custom-designed 3D-printed stand and play a melody whenever our armed IoT alarm module is triggered.

We will start our development by upgrading the code for our IoT alarm module so that it works with the map feature of the IoT alarm dashboard.

## Creating a Raspberry Pi 5 alarm dashboard

Since its release in late 2023, the Raspberry Pi 5 has set a new standard for compact computing, enabling developers to build more efficient and powerful applications across various domains.

The Raspberry Pi 5 boasts an advanced processor that enhances our IoT alarm dashboard's data processing speed and multitasking capabilities. Its robust software support and extensive Python support offer unparalleled programming flexibility, tailored to the IoT alarm system's needs. Paired with the 7-inch touchscreen, our Raspberry Pi 5 provides us with a user-friendly and efficient interface for our system.

We will begin developing our IoT alarm dashboard by modifying the IoT alarm module code from *Chapter 6* so that it publishes the `location` data on the `IoTAlarm` MQTT topic. This data will enable our dashboard to identify the geographic position of the IoT alarm module with precision once it's armed and activated.

### Modifying the IoT alarm module code

*Figure 8.1* shows a slightly modified version of our IoT alarm module with the addition of a `location` message that sends geolocation data to our IoT alarm dashboard. In this case, we could integrate a GPS module, like the shown one in *Figure 8.4*, into our IoT alarm module:



Figure 8.4 – The GPS module next to a Raspberry Pi Pico

#### GPS test code

This chapter's GitHub repository contains test code for a BN-180 GPS module and a Raspberry Pi Pico.



However, despite its compact size and ability to easily connect to the Raspberry Pi Pico W, its value is limited as our IoT alarm module is for indoor use, and the GPS would struggle to obtain a strong signal:

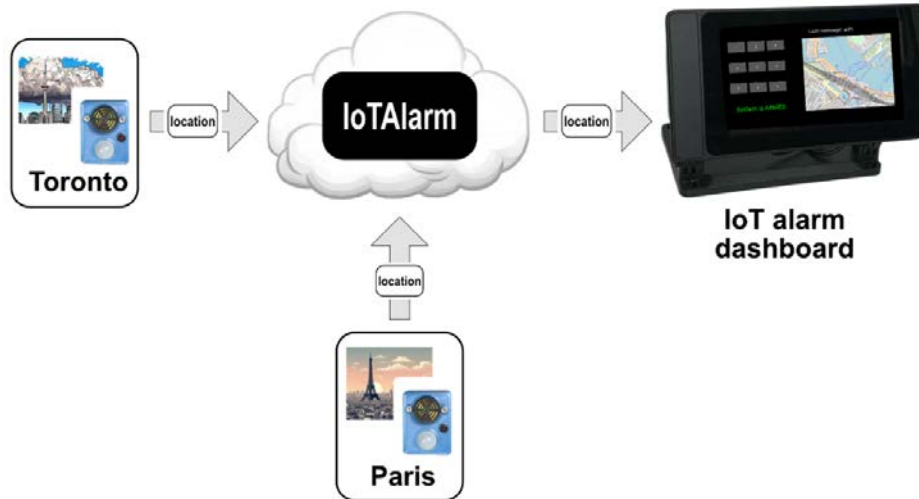


Figure 8.5 – Worldwide deployment of the IoT alarm module

Instead, we'll embed the GPS coordinates directly into our IoT alarm module's code, operating under the assumption that the module's location will remain static post-deployment. As the code requires Wi-Fi configuration updates, it's straightforward to adjust the GPS details when we deploy the IoT alarm module on-site. This geolocation data will be published within the `location` message as part of enhancing our module code. This approach allows us to deploy our IoT alarm modules globally, as illustrated in *Figure 8.5*.

To publish the geolocation information, we will modify the `motion_handler()` method from the `main.py` file, which is stored on our IoT alarm module.

#### The advantage of using MicroPython over C with microcontrollers

While MicroPython runs slower than C, its adaptability is evident when it comes to modifying the code for the IoT alarm module. With C, changes require recompilation and external code tracking. MicroPython, however, can be edited directly on the microcontroller, bypassing filesystem searches during alterations.

To modify the code, we must do the following:

1. First, we connect our Raspberry Pi Pico W to a USB port on your computer and launch Thonny.
2. Then, we activate the MicroPython environment on our Pico W by selecting it from the bottom right-hand side of the screen:

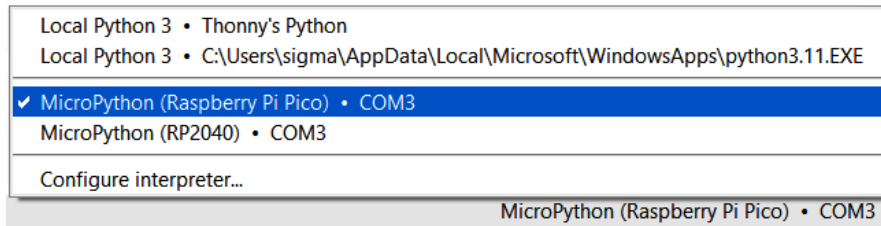


Figure 8.6 – Selecting MicroPython and Pico in Thonny (Windows version shown)

3. Under the **Raspberry Pi Pico** tab on the left-hand side of the screen, we double-click on the `main.py` file to open it:

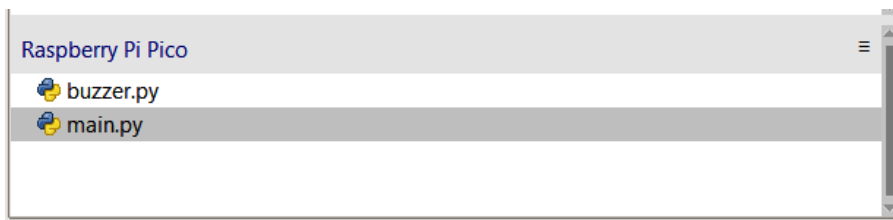


Figure 8.7 – Selecting the `main.py` file

4. Then, we rewrite the `motion_handler()` method so that it reads as follows:

```
def motion_handler(pin):
 print('Motion detected!')
 if mqtt_client:
 if ARMED:
 activate_buzzer()
 mqtt_client.publish(b"IoTAlarm", b"buzzer")
 mqtt_client.publish(b"IoTAlarm",
 b"location:43.6426,-79.3871")
 else:
 mqtt_client.publish(b"IoTAlarm", b"motion")
 else:
 print("MQTT client not connected.")
```

5. Our only change here is the addition of another publish method on the `mqtt_client` object. In the new `publish()` method, we create a message called `location` that provides the approximate GPS coordinates of the CN Tower in Toronto, Canada.

**IoT alarm module geolocation**

We've designed our IoT alarm module to be permanently installed at an indoor location – that is, the hard-coded geolocation. We are using the CN Tower in Toronto, Canada as an example. You're encouraged to provide your own unique GPS coordinates.

6. We then save our changes to our Raspberry Pi Pico W.

We'll see the impact of this adjustment when we set up and launch the IoT alarm dashboard on our Raspberry Pi 5 in the upcoming section.

## Writing the dashboard code

For the IoT alarm dashboard, we'll use the Raspberry Pi 5, the Raspberry Pi 7-inch touchscreen, and its compatible case. This configuration is like what we established in *Chapter 4*, with the addition of an external buzzer exclusive to the IoT alarm dashboard.

We'll begin by setting up our development environment and installing the packages required for our code.

### *Setting up our development environment*

We will use a Python virtual environment for our development. As there are libraries that only work with the root installation of Python, we will use system packages in our Python virtual environment. To do so, we must do the following:

1. On our Raspberry Pi 5, we open a Terminal application.
2. To store our project files, we create a new directory by running the following command:

```
mkdir dashboard
```

3. Then, we navigate to the new directory:

```
cd dashboard
```

4. Next, we create a new Python virtual environment for our project:

```
python -m venv dashboard-env --system-site-packages
```

With this command, we create a new Python virtual environment called `dashboard-env` and enable access to the `--system-site-packages`. This allows the virtual environment to inherit packages from the global Python environment without affecting the global Python environment.

5. With our new Python virtual environment created, we source into it with the following command:

```
source dashboard-env/bin/activate
```

6. Our Terminal application should now show that we are using the `dashboard-env` Python virtual environment:

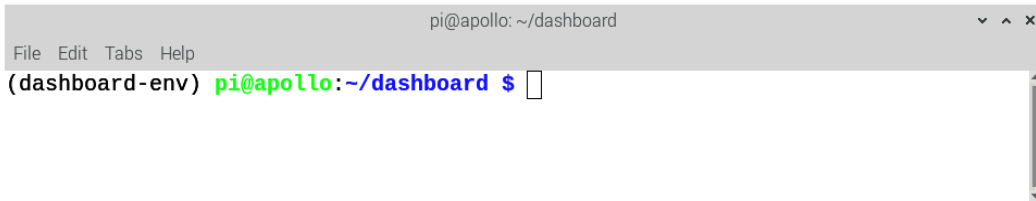


Figure 8.8 – Terminal showing the use of the `dashboard-env` environment

7. Next, we install the extra packages that are required for our code:

```
pip install kivy kivy-garden.mapview paho.mqtt==1.5.1
```

Kivy is an open source Python library for developing multitouch applications on various devices and platforms. The `kivy-garden.mapview` extension offers an interactive map widget, while `paho.mqtt` is a Python client library that enables MQTT communications; it's popular in IoT due to its efficiency. We're ensuring that we install the version of `paho.mqtt` that will work with our program by specifying that we want `1.5.1`.

8. With the extra packages installed, we close the Terminal:

```
exit
```

9. We are now ready to load up Thonny. To do so, we click on the **Menu** icon in the Raspberry Pi taskbar, navigate to the **Programming** category, and select **Thonny**.
10. By default, Thonny uses the Raspberry Pi's built-in version of Python. For our project, we will use the Python virtual environment we just created. To start, we need to view the project files by clicking on **View** and selecting **Files** if it isn't already selected.
11. In the **Files** section, we locate and open the `dashboard-env` directory.

12. Then, right-click on the `pyvenv.cfg` file and select the **Activate virtual environment** option:

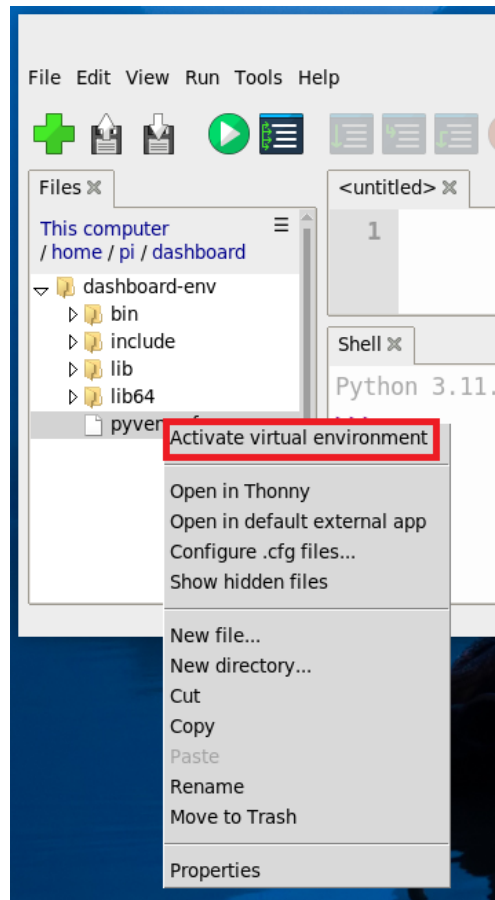


Figure 8.9 – Activating a Python virtual environment in Thonny

#### Activating the Python virtual environment

In the steps above we activated the Python virtual environment by clicking on the `pyvenv.cfg` file. The step was outlined this way to show an alternative way to activate the Python virtual environment than the way shown in previous chapters.

With our project folder created, our Python virtual environment set up, and the packages we need for our project installed, we can start writing the code for our IoT alarm dashboard. We will divide our code into two files – one for the GUI that creates the dashboard and the other to activate the buzzer. But before we do, we must wire up the buzzer to the GPIO port of our Raspberry Pi 5.

### *Wiring up the buzzer*

For our project, we will use an SFM-27 active buzzer. We wire the buzzer with the positive wire (red) connected to GPIO 4 and the negative wire (black) connected to GND on the Raspberry Pi 5. We have the option of installing the buzzer into a custom case, something we'll cover later in this chapter. For our code development and testing purposes, it is enough to wire the SFM-127 active buzzer directly to the Raspberry Pi 5:

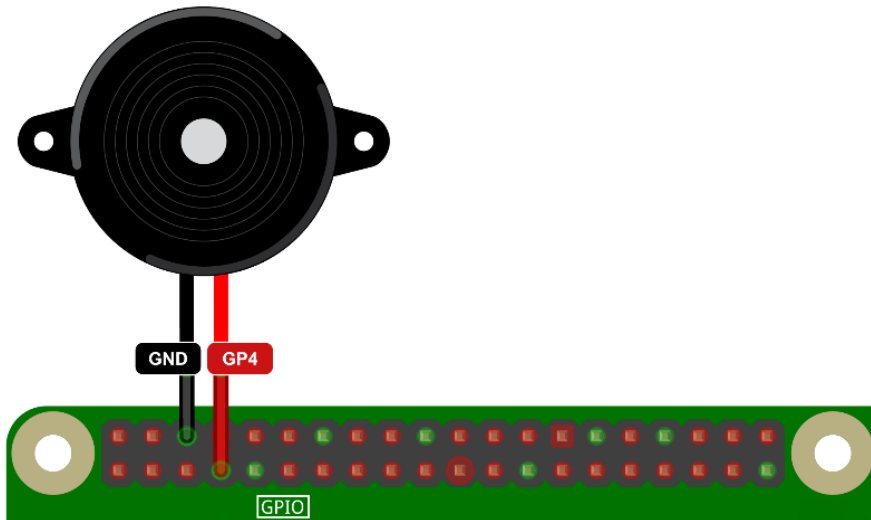


Figure 8.10 – Wiring the buzzer to the Raspberry Pi 5's GPIO port

With the buzzer connected to our Raspberry Pi 5, it is now time to write and test the buzzer code. We will use this code to activate the buzzer when our dashboard receives a `buzzer` message from the MQTT server.

### *Writing and testing the buzzer code*

We write our buzzer code using Thonny on our Raspberry Pi 5, which has a 7-inch screen. For more screen space to assist in coding, we can add another monitor through the Raspberry Pi 5's mini-HDMI port, creating a dual-monitor setup.

To write and test our buzzer code, do the following:

1. We launch Thonny by clicking on the **Menu** icon in the Raspberry Pi taskbar, navigating to the **Programming** category, and selecting **Thonny**.
2. We then activate the `dashboard-env` Python virtual environment.

3. Once inside Thonny, we create a new tab by selecting **File** and then **New** or by hitting *Ctrl + N* on your keyboard.
4. In our new file, we enter the following code:

```
from gpiozero import TonalBuzzer
from gpiozero.tones import Tone
from time import sleep
class BuzzerMelody:
 def __init__(self, pin, notes=[('E4', 1),
 ('E4', 0.5),
 ('F4', 0.5),
 ('G4', 1.5)]):
 self.buzzer = TonalBuzzer(pin)
 self.melody = notes

 def play_melody(self):
 for note, duration in self.melody:
 self.buzzer.play(Tone(note))
 sleep(duration)
 self.buzzer.stop()
 sleep(0.1) # pause between notes

if __name__ == "__main__":
 buzzer_melody = BuzzerMelody(4)
 buzzer_melody.play_melody()
```

5. We save the file as `buzzer.py` in the dashboard project folder on our Raspberry Pi 5.

Before we test our code, let's examine it:

- I. We start by importing the `TonalBuzzer` class from the `gpiozero` module.
- II. Then, we import the `Tone` class from `gpiozero.tones`.
- III. We finish our imports by importing the `sleep` function from the `time` module.
- IV. Next, we define a `BuzzerMelody` class: Initializer (`__init__()`) accepts a pin and a list of notes with their durations. The list has a default melody. Within the initializer, we do the following:
  - i. Initialize a `TonalBuzzer` object with the provided pin
  - ii. Set the melody

- V. Then, we define the `play_melody()` method. In this method, we do the following:
  - i. Iterate through the melody notes
  - ii. Play each note for its specified duration
  - iii. Ensure the buzzer stops after the notes are played
  - iv. Add a brief pause between notes
- VI. If the script is executed as the main program, we do the following:
  - i. Create an instance of the `BuzzerMelody` class with pin 4
  - ii. Play the melody using the `play_melody()` method
6. We run the code in Thonny by clicking on the green run button, hitting *F5* on your keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.
7. We should hear a melody play from our buzzer.

With our buzzer code in place, it's time to write our main dashboard code using Kivy.

### ***Creating our Kivy dashboard***

As mentioned previously, we will be using the Raspberry Pi 5's 7-inch touchscreen to display our dashboard. The use of this screen is optional as we may use a standard monitor with our Raspberry Pi.

To write and test our Kivy dashboard code, follow these steps:

1. Launch Thonny by clicking on the **Menu** icon in the Raspberry Pi taskbar, navigating to the **Programming** category, and selecting **Thonny**.
2. We then activate the `dashboard-env` Python virtual environment.
3. Once inside Thonny, create a new tab by selecting **File** and then **New** or by hitting *Ctrl + N* on your keyboard.
4. We will start our code with the necessary imports:

```
from kivy.config import Config

Config.set('graphics', 'fullscreen', 'auto')
Config.set('graphics', 'borderless', '1')

from kivy.app import App
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.label import Label
from kivy.uix.button import Button
from kivy.clock import Clock
from threading import Thread
```



```
import paho.mqtt.client as mqtt
from kivy_garden.mapview import MapView, MapMarkerPopup
from buzzer import BuzzerMelody
```

Let's examine this code:

- `kivy.config`: Our code imports the configuration settings from Kivy to tailor the application's behavior.
- `fullscreen`: We configure the application to run in fullscreen mode.
- `borderless`: Our settings eliminate the window border.
- `kivy.app`: Our code accesses the primary application class from Kivy that's used for initializing and operating Kivy apps.
- `kivy.uix.floatlayout`: We import `FloatLayout`, a flexible layout mechanism that places widgets based on relative coordinates.
- `kivy.uix.label`: Our code uses the `Label` widget, letting us display text within the application.
- `kivy.uix.button`: We integrate the `Button` widget so that it can be used with our keypad.
- `kivy.clock`: We utilize the `Clock` class to schedule specific functions to run at timed intervals.
- `Threading`: To ensure smooth multitasking, our code uses threading for parallel operations. We use threading to run the `start_mqtt()` method in a separate thread.
- `paho.mqtt.client`: We use the MQTT client library in our code to communicate with our MQTT server.
- `kivy_garden.mapview`: Our code pulls in classes for presenting maps (`MapView`) and interactive map markers (`MapMarkerPopup`). We use these classes to visually represent the exact location on the map where the alarm has been triggered.
- `buzzer.BuzzerMelody`: We import the `BuzzerMelody` class so that we can activate the external buzzer.

5. Then, set the variable declarations for our code:

```
MQTT_SERVER = "codfish.rmq.cloudamqp.com"
MQTT_PORT = 1883
USERNAME = "<<MQTT server username>>"
MQTT_PASSWORD = "<<MQTT server password>>"
DEVICE_ID = "IoTAlarmDashboard"
TOPIC = "IoTAlarm"
```

6. From here, we must define the `AlarmDashboard` class, which inherits from the `FloatLayout` class, and define the initialization method:

```
class AlarmDashboard(FloatLayout):

 def __init__(self, **kwargs):
 super(AlarmDashboard, self).__init__(**kwargs)
 self.password_toggle = "1234"
 self.entered_password = ''
 self.system_armed = False
 self.lockout = False
 self.last_message = ''

 self.client = mqtt.Client(client_id=DEVICE_ID)
 self.client.username_pw_set(USERNAME, MQTT_PASSWORD)
 self.client.on_message = self.on_message

 Thread(target=self.start_mqtt).start()

 self.init_widgets()
 self.buzzer = BuzzerMelody(4)
```

Let's examine this code:

- I. First, we initialize specific attributes, such as the default password (`password_toggle`), the currently entered password (`entered_password`), a system status flag (`system_armed`), a lockout status (`lockout`), and the last received message (`last_message`).
  - II. Then, we set up an MQTT client with a unique `client_id` and provide it with the necessary authentication details using the `username_pw_set` method.
  - III. The `on_message` attribute of the client is set to a method named `on_message()`, which our code will use to handle incoming MQTT messages.
  - IV. We start the MQTT connection on a separate thread using the `start_mqtt` method to avoid blocking the main application.
  - V. The `init_widgets` method, which we'll define later, is called to initialize and arrange the user interface elements.
  - VI. Lastly, we initialize the `buzzer` attribute with an instance of the `BuzzerMelody` class, setting it up on GPIO pin 4.
7. Now, use the `on_message()` method to handle messages coming from our MQTT server:

```
def on_message(self, client, userdata, msg):
 message = f"{str(msg.payload.decode('utf-8'))}"
```

```
self.last_message = "Last message: " + message

if message.startswith("location:"):
 parts = message.split(":")[1].split(",")
 lat = float(parts[0])
 lon = float(parts[1])

 Clock.schedule_once(
 lambda dt: self.update_map(lat, lon), 0
)

if message == "arm":
 Clock.schedule_once(
 lambda dt: self.update_system_status(True), 0
)
elif message == "disarm":
 Clock.schedule_once(
 lambda dt: self.update_system_status(False), 0
)

if message == "buzzer":
 self.buzzer.play_melody()

Clock.schedule_once(
 lambda dt: self.update_message_display(), 0
)
```

Let's examine this code:

- I. Upon receiving a message, our code decodes the message payload from bytes to a string
- II. The most recent message is stored with the `Last message:` prefix
- III. If the received message starts with `location:`, it extracts latitude and longitude values
- IV. These values are then used to update the displayed map, scheduling the `update_map()` method
- V. If the message that's received is `arm`, the system's status is updated to armed via the `update_system_status()` method
- VI. If the message that's received is `disarm`, the system's status is updated to disarmed via the `update_system_status()` method
- VII. If the message that's received is `buzzer`, a melody is played through the `buzzer` instance via the `play_melody()` method
- VIII. Regardless of the message's content, the message that's displayed is updated using the `update_message_display()` method

### Using the `lambda()` function in our code

The `lambda()` function is used as an anonymous, inline function in Python. It's typically used for brief operations that are expressed in a single statement. Specifically, in our code, the `lambda()` function pairs with Kivy's `Clock.schedule_once()` method to delay certain method executions. By using the `Clock.schedule_once()` method, a function can be scheduled to run after a specified delay. When 0 is given as its second argument, it indicates the function should be called immediately on the next frame.

Each instance of the `lambda()` function in our code follows the `lambda dt: some_method(arguments)` pattern. Here, `dt` represents the time that's elapsed since the last frame. It's an argument that `Clock.schedule_once()` automatically provides. For example, `Clock.schedule_once(lambda dt: self.update_map(lat, lon), 0)` schedules the `self.update_map(lat, lon)` method for execution on the upcoming frame, with the already parsed `lat` and `lon` values as its arguments. Essentially, these `lambda()` functions act as a conduit, passing arguments to methods and setting them up for execution on the next frame via Kivy's scheduling mechanism. Utilizing the `Clock()` function ensures that our UI updates remain smooth and synchronized with the display's refresh rate, and also prevents blocking of the main thread, thus enhancing the responsiveness of the application.

8. Our code defines a method called `update_map()`, whose purpose is to update the displayed map in two main ways:

```
def update_map(self, lat, lon):
 self.mapview.center_on(lat, lon)
 marker = MapMarkerPopup(lat=lat, lon=lon)
 self.mapview.add_widget(marker)
```

Let's examine this code:

- I. Our code adjusts the center of the map to the new latitude and longitude coordinates, which are provided as arguments (`lat` and `lon`).
  - II. Then, it places a marker (specifically an interactive marker that can show a popup) on the map at the specified coordinates. This marker indicates the exact location of the IoT alarm module that triggered the alarm.
9. The `update_system_status()` method updates the status message on our dashboard based on the value of `is_armed`:

```
def update_system_status(self, is_armed):
 if is_armed:
 self.system_armed = True
 self.system_status.text = "System is ARMED"
 self.system_status.color = (0, 1, 0, 1)
 else:
```

```

self.system_armed = False
self.system_status.text = "System is DISARMED"
self.system_status.color = (1, 0, 0, 1)

```

10. Then, our code defines a method named `start_mqtt()` that sets up and initiates the MQTT communication for the application:

```

def start_mqtt(self):
 self.client.connect(MQTT_SERVER, MQTT_PORT)
 self.client.subscribe(TOPIC)
 self.client.loop_forever()

```

Let's examine this code:

- I. The method connects the MQTT client to the specified MQTT server using the given server address (`MQTT_SERVER`) and port number (`MQTT_PORT`).
  - II. Once connected, the client subscribes to a specific topic (`TOPIC`), meaning it will start listening for messages that are published on that topic.
  - III. Finally, the `loop_forever()` method of the client is called, which keeps the MQTT client continuously checking for incoming messages and handling them for as long as the application runs.
11. Our code defines a method named `init_widgets()` for initializing and placing various user interface components on our dashboard. We will start with the **keypad initialization** process:

```

def init_widgets(self):
 # Keypad buttons
 positions = [
 (0.03, 0.75), (0.14, 0.75), (0.25, 0.75),
 (0.03, 0.55), (0.14, 0.55), (0.25, 0.55),
 (0.03, 0.35), (0.14, 0.35), (0.25, 0.35)
]

 for index, pos in enumerate(positions, 1):
 btn = Button(
 text=str(index), size_hint=(0.1, 0.1),
 pos_hint={'x': pos[0], 'y': pos[1]}
)
 btn.bind(on_press=self.handle_key_press)
 self.add_widget(btn)

```

Here, a keypad layout is set up with buttons arranged in a 3x3 grid. The positions of these buttons are specified using the `positions` list, where each tuple represents the relative x and y coordinates. As we loop through these positions, we create a button with a corresponding number and bind its `on_press` event to the `handle_key_press()` method, which will capture the button press actions. Each button, once initialized, is added to the dashboard using the `add_widget()` method.

12. Now, write the code for the **system status label**:

```
System status
self.system_status = Label(
 text="System is DISARMED",
 size_hint=(1, 0.2),
 pos_hint={'x': -0.3, 'y': 0.1},
 font_size=30,
 color=(1, 0, 0, 1)
)
self.add_widget(self.system_status)
```

This label displays the status of the system, indicating whether it is armed or disarmed. It's styled with a specific text size, color, and positioning. Once initialized, the label is added to the dashboard.

13. Next, set up a label for the **MQTT message display**:

```
MQTT Messages
self.message_display = Label(
 text="Waiting for message...",
 size_hint=(0.77, 0.6),
 pos_hint={'x': 0.23, 'y': 0.62},
 font_size=25,
 color=(1, 1, 1, 1)
)
self.add_widget(self.message_display)
```

Here, a label has been set up to display incoming MQTT messages. It has a default text of `Waiting for message...` and is styled similarly to the system status label. Once it's been created, it's added to the dashboard.

14. Finally, add a **MapView widget**:

```
self.mapview = MapView(
 zoom=15, lat=52.379189, lon=4.899431,
 size_hint=(0.5, 0.7), pos_hint={'x': 0.45, 'y': 0.15}
)
self.add_widget(self.mapview)
```

Here, we initialize a `MapView` widget to display geographical locations. It's preset to a specific zoom level and initial latitude and longitude coordinates. This map view allows us to visualize locations on a map. Once initialized, it's added to our dashboard.

15. The `update_message_display()` method is used to refresh or update the text that's displayed in the `message_display` widget to show the latest message that's been received by the system:

```
def update_message_display(self):
 self.message_display.text = self.last_message
```

16. The `handle_key_press()` method manages the user's interactions with the virtual keypad when inputting the alarm system's passcode, determines the validity of the entered code, and adjusts the alarm system's status based on the passcode input:

```
def handle_key_press(self, instance):
 if not self.lockout:
 self.entered_password += instance.text
 print("The key:" + instance.text + " was pressed")

 if len(self.entered_password) == 4:
 if self.entered_password == self.password_toggle:
 if self.system_armed:
 self.system_armed = False
 self.system_status.text = "System is\
DISARMED"

 self.system_status.color = (1, 0, 0, 1)
 self.client.publish(TOPIC, "disarm")
 else:
 self.system_armed = True
 self.system_status.text = "System is ARMED"
 self.system_status.color = (0, 1, 0, 1)
 self.client.publish(TOPIC, "arm")
 self.entered_password = ''
 else:
 self.lockout = True
 Clock.schedule_once(self.end_lockout, 5)
 self.entered_password = ''
```

Let's examine this code:

- **Lockout check:** The method begins by determining if the system is currently in a *lockout* state, preventing any further input if true.
- **Accumulating key presses:** Each button press on the keypad is appended to the `self.entered_password` string, representing the user's current passcode input.

- **Logging the keypress:** For debugging purposes, the specific key that was pressed is printed out.
  - **Password verification:** Once the user has entered a 4-digit code, the `handle_key_press()` method verifies if it matches the predefined alarm system code. If the code is correct, the following occurs:
    - i. The alarm system's status toggles between `armed` and `disarmed`.
    - ii. A corresponding message (`arm` or `disarm`) is sent to the `IoTAlarm` topic.
  - **Incorrect code handling:** If the code that's been entered is incorrect, the system is placed in `lockout` mode:
    - i. A timer is initiated to end the lockout state after a specified duration (5 seconds in this case).
    - ii. The code that was entered is reset in anticipation of another attempt.
17. The `end_lockout()` function is designed to terminate the lockout state of the system. Setting the `lockout` attribute to `False` ensures that subsequent operations or interactions that were restricted during the lockout are now permitted:

```
def end_lockout(self, dt):
 self.lockout = False
```

18. At this point, our code introduces the main application class, `MyApp`, which is built upon Kivy's App framework. Inside this class, the `build()` method constructs and returns an instance of `AlarmDashboard`. If we run this code directly (not imported into another script), the `if __name__ == '__main__':` check ensures that a new `MyApp` instance is created and launched, initiating the entire alarm application:

```
class MyApp(App):

 def build(self):
 return AlarmDashboard()

if __name__ == '__main__':
 MyApp().run()
```

19. Save our code as `dashboard.py` in the project folder.

With our dashboard code written and our buzzer tested, it is time to build the custom stand for our buzzer. As always, this is optional; we may run our application without encasing the buzzer into a stand. However, doing so makes our application more professional-looking, enhances user experience, and offers a more polished presentation, ensuring that the buzzer is positioned securely and easily accessible when needed.



## Building the external alarm buzzer stand

Encasing our buzzer in a custom stand enhances our IoT alarm dashboard. In this section, we will build the stand. We will start by identifying the parts that make up our custom buzzer stand before we assemble it.

### Identifying the parts

The parts that make up the custom buzzer stand can be made with either a 3D printer or a 3D printing service such as Shapeways (<https://www.shapeways.com>).

These parts are shown in *Figure 8.11*:

- *A*: This is a 3D-printed version of the `Stand.stl` file, which is located in the `Build Files/Buzzer Stand` folder of this chapter's GitHub repository. This version of the file was 3D printed using a liquid resin 3D printer and was subsequently painted. For those of us who prefer to use an FDM printer, the split stand located under `Build Files/Split Stand` would be a better option:



Figure 8.11 – Parts of the custom buzzer stand

- *B*: This is our standard hook. It's a liquid resin 3D-printed version of the `Hook.stl` file, which is located in the `Build Files` folder of this chapter's GitHub repository.
- *C*: This is the standard SFM-127 active buzzer that this stand was designed to hold.
- *D*: This is a painted FDM 3D-printed version of the `Front.stl` file, which is also located in the `Build Files` folder.
- *E*: An FDM 3D printer was used to create a version of the `Back.stl` file. This part connects the front (*D*) to the stand (*A*) using the hook (*B*).

- Not shown: A multi-conductor cable such as a discarded USB cable.
- Not shown: 2x M4 20mm bolts.
- Not shown: 2x M3 10mm bolts.
- Not shown: 2x M2 5mm bolts.
- Not shown: 1x M5 20mm bolt.
- Not shown: 1x M5 nut.
- Not shown: 2x female jumper terminals for connecting the buzzer to the GPIO port.

#### Which type of 3D printer should be used to create the parts?

In *Figure 8.9*, parts were created using both FDM and liquid resin 3D printers. The choice of printer depends on the user's experience and the part's design. Flat parts, such as *E* and *D* in *Figure 8.10*, are ideal for FDM printers. Ornamental designs, such as part *A*, benefit from liquid resin printers. While hook (*B*) can use either method, we chose a liquid resin printer with an engineering-grade resin (Siraya Tech Blu) for added strength.

With our parts identified, it is time to build our custom buzzer stand.

## Building the stand

To build the custom buzzer stand, we must follow the steps shown in *Figure 8.12*:



Figure 8.12 – Steps to build the custom buzzer stand

Let's take a closer look:

1. Using two M4 10mm bolts, secure the buzzer (see *C* in *Figure 8.11*) to the front casing (see *D* in *Figure 8.11*). The bolts should screw tightly into the buzzer. If they're loose, we may tighten them with two M4 nuts.
2. Using a multi-conductor cable, such as a discarded USB cable, solder two wires to the buzzer (see *C* in *Figure 8.11*).
3. Thread the multi-conductor cable through the hole on the front casing (see *D* in *Figure 8.11*) and apply glue via a hot glue gun to secure the cable to the casing.
4. Using the same two wires (red and black, in our example) of the multi-conductor cable, crimp two female jumper terminals to the ends of the wire. An option here is to solder pre-existing jumper cables to the end instead of crimping.
5. Then, secure the hook (see *B* in *Figure 8.11*) to the back plate (see *E* in *Figure 8.11*) by either using two M2 5mm screws or epoxy glue.
6. Using two M3 10mm bolts, secure the back plate (see *E* in *Figure 8.11*) to the front casing (see *D* in *Figure 8.11*).
7. Using an M5 20mm bolt and M5 nut, secure the assembled casing to the stand (see *A* in *Figure 8.11*).

With the custom stand built, we may re-connect our buzzer to the GPIO port of the Raspberry Pi 5 by following the wiring diagram shown in *Figure 8.10*.

We are now ready to test our application.

## Running our application

It's now time for the moment we have been building toward over the last few chapters: testing our entire IoT alarm system. With the IoT alarm dashboard built, our IoT alarm system is complete:

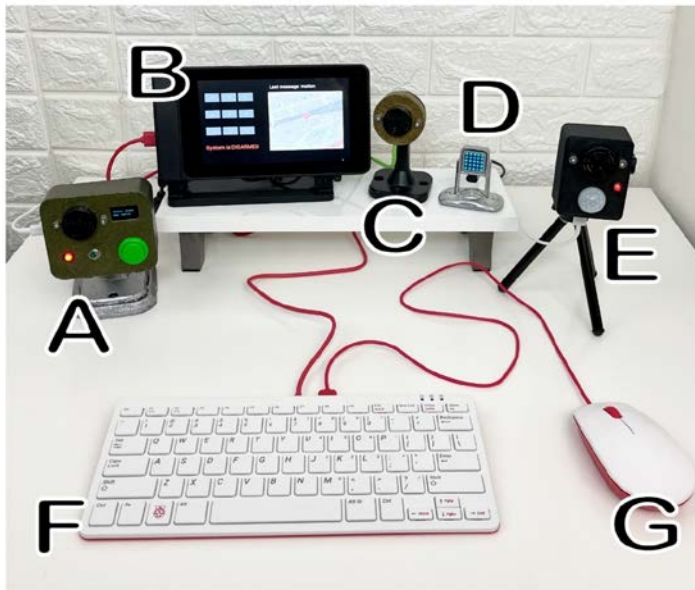


Figure 8.13 – Completed IoT alarm system

We started by building the IoT alarm module (see E in Figure 8.13) in Chapter 6, before proceeding to build IoT buttons (see A and D in Figure 8.13) in Chapter 7. In this chapter, we used the Raspberry Pi 5 with an active buzzer (see B and C in Figure 8.13), a Raspberry Pi 7-inch screen and case (see B in Figure 8.13), a keyboard (see F in Figure 8.13), and a mouse (see G in Figure 8.13) to build the IoT alarm dashboard.

To test the IoT alarm dashboard, we simply need to arm the IoT alarm module and activate it by moving an object in front of its PIR sensor. To do so, follow these steps:

1. To arm the IoT alarm module, using our mouse, we type the four-digit PIN code (1234) into the keypad of the IoT alarm dashboard.
2. We should observe that the IoT alarm module goes into armed mode by the long blinking of the LED. To activate the alarm, we wave our hands in front of the PIR sensor.
3. After a few seconds, we should hear the buzzer on the IoT alarm module, followed by the buzzers on the IoT button (version 2) and the IoT alarm dashboard.

4. We should also observe that a map of Toronto with a marker at the CN Tower should be displayed on the screen of the IoT alarm dashboard:

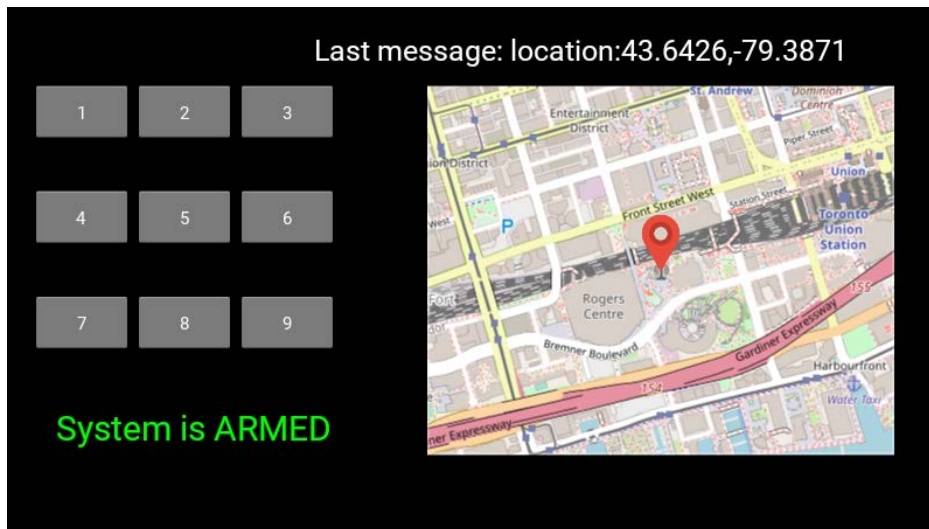


Figure 8.14 – IoT alarm dashboard showing the location where the alarm was activated

Having successfully tested the IoT alarm system, we've concluded the second part of this book. Throughout this journey, we've explored internet-based device communication and harnessed the capabilities of the Raspberry Pi Pico W microcontroller and Raspberry Pi 5 computer. This foundation will be invaluable for the advanced projects in this book.

If we really want a challenge, we could position each component of the IoT alarm system as far apart as possible, perhaps by giving a component to a friend in another city. Doing such an activity will help us appreciate not only IoT but our internet-connected world in general.

## Summary

In this chapter, we completed our IoT alarm system by building an IoT alarm dashboard. We began by understanding IoT alarm dashboards and their usage by considering an example of the level and temperature of a tank. Throughout this chapter, we went through the step-by-step process of setting up, coding, and testing our IoT alarm dashboard, including adding a custom stand for an external buzzer.

We constructed a dashboard for our IoT alarm system using Kivy on the Raspberry Pi 5. The dashboard integrated map functionality to pinpoint the location of any activated IoT alarm module.

As we close this section on the IoT alarm system, we look forward to the next exciting challenge: building a remote monitoring station using LoRa. This next section will introduce us to long-range communication, broadening our understanding and capabilities in IoT projects.

# Part 3:

## Creating a LoRa-Enabled IoT Monitoring Station

In this part, we will create a remote environmental monitoring station, using LoRa to collect and transmit data on temperature and humidity and transmit it over the internet. We will use this data to control an analog weather indicator.

This part has the following chapters:

- *Chapter 9, Understanding LoRa*
- *Chapter 10, Integrating LoRa with the Internet*



# Understanding LoRa

In this chapter, we explore the world of **LoRa** (short for **Long Range**), a key technology in IoT communication. LoRa is known for transmitting data over extensive distances with minimal power. We will explore its practical applications in areas such as agriculture, where it enables efficient management of large-scale sensor networks, and urban settings, where it assists in smart city initiatives such as street lighting control.

We will also investigate the **radio frequency spectrum** and understand how different frequencies are allocated for various wireless communications, allowing us to better comprehend the operational range of LoRa technology. By studying the frequency spectrum, we can identify which frequency bands are most suitable for LoRa transmissions.

Our focus then shifts to the practical aspect: building a LoRa sensory transmitter and LoRa receiver using a Raspberry Pi Pico and Pico W, respectively. We'll start with assembling the transmitter circuit by integrating an RFM95W LoRa module, a DHT22 temperature sensor, and a Raspberry Pi Pico. We will then house these components in a custom 3D-printed case. We'll emphasize using the standard Raspberry Pi Pico for its efficiency in tasks that don't require Wi-Fi, benefiting from its lower power consumption and reduced firmware overhead.

For the receiver, we use a Raspberry Pi Pico W, focusing on its Wi-Fi capabilities for future developments. We'll construct a custom case for the receiver, as with the transmitter, but with an LED for status indication.

The development process includes setting up CircuitPython, installing necessary libraries, and writing the code for both the transmitter and receiver. CircuitPython is an open-source derivative of MicroPython, developed by Adafruit, designed to simplify coding for microcontrollers. In our code, we'll use a delay between transmissions to adhere to European duty-cycle limitations.

Finally, we will test our application outdoors, demonstrating LoRa's impressive range capabilities as compared with the limited range of Wi-Fi.



We will cover the following topics in this chapter:

- Exploring LoRa
- Building a LoRa sensory transmitter
- Building a LoRa receiver

Let's begin!

## Technical requirements

The following are the requirements for completing this chapter:

- Intermediate knowledge of Python programming
- 1 x Raspberry Pi Pico WH (with headers) for development (may add headers to a Raspberry Pi Pico W instead)
- 1 x Pico GPIO expander for development
- 1 x Raspberry Pi Pico (for LoRa sensory transmitter)
- 1 x Raspberry Pi Pico W (for LoRa receiver)
- 2 x RFM95W LoRa modules
- 1 x DHT22 temperature and humidity sensor
- 1 x LED (any color)
- 1 x 220 Ohm resistor
- 4 x M3 10 mm bolts
- 12 x M2 5 mm screws
- 2 x M5 20 mm bolts
- 2 x M5 nuts
- Hot glue gun
- Access to a 3D printer or 3D printer service to print custom cases
- Build files for custom cases may be found in our GitHub repository

The code and build files for this chapter may be found here:

<https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter9>

## Exploring LoRa

LoRa is a wireless communication technology known for its ability to transmit data over long distances while consuming very little power. It has become increasingly significant in the field of IoT due to its efficiency and reliability in various environments. Originating as a solution for extending the range of wireless communications, LoRa technology has evolved to play a pivotal role in connecting a wide array of IoT devices.

In the following sections, we will look at the practical uses of LoRa before we explore the frequency spectrum and how it relates to LoRa.

### Practical uses for LoRa technology

LoRa technology is instrumental in areas such as agriculture, enabling farmers to deploy sensory networks for monitoring soil moisture, temperature, and other vital parameters over large areas, enhancing crop management and resource efficiency.

**LoRa nodes** offer a cost-effective solution for modernizing agricultural practices, contrasting with costlier alternatives such as **Long Term Evolution (LTE)**-based systems or extensive wired sensor networks. While LTE supports more data-intensive applications, its higher power requirements and operational costs can be prohibitive.

In *Figure 9.1*, we see an AI-generated image of LoRa nodes used to measure soil and weather conditions for a modern farm:



Figure 9.1 – Modern farm using LoRa technology

In urban environments, LoRa may be used to manage smart city applications such as street lighting. In *Figure 9.2*, we see a smart light post used in an urban environment. In this setting, control of the light is determined by a central office with a LoRa message sent to the light post to turn it on or off or control its brightness:

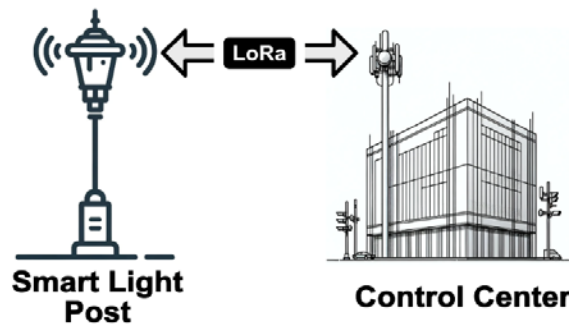


Figure 9.2 – Smart light post using LoRa messages for control

This method of control is beneficial as it allows for remote management of lighting based on real-time data and needs, such as adjusting brightness according to traffic conditions or time of day.

To better understand the application of LoRa communication technology, let's examine the radio frequency spectrum it uses, focusing on how different frequencies cater to various application needs and scenarios.

## Investigating the radio frequency spectrum

The radio frequency spectrum is used for various wireless communications, including television broadcasting, mobile data, satellite, and IoT technologies such as LoRa. Frequencies are allocated for specific uses to avoid interference and optimize communication efficiency. **Ultra high frequency (UHF)** bands, for example, are commonly used for television broadcasting, mobile phones, and Wi-Fi. Each frequency band has unique characteristics such as propagation range and penetration ability, making them suitable for specific applications.

In general, lower frequencies, characterized by their longer wavelengths, can travel greater distances and are more effective at penetrating obstacles compared to higher frequencies. Conversely, higher frequencies, while having shorter wavelengths, possess the ability to carry more data due to their larger bandwidth. These characteristics are crucial in determining the suitable frequency band for different types of communication.

A common example of the effectiveness of the penetration of low frequencies is how we often hear the bass, or lower frequencies, from music playing loudly inside a club when we are outside, but not higher frequencies, as lower frequencies are better at penetrating the walls of the club. This ability to penetrate through objects makes lower frequencies especially useful for communications that need to cover larger areas or pass through obstructions. For instance, lower frequency bands such as those

used in AM radio can cover vast geographical areas, while higher frequencies, despite offering more bandwidth, have a shorter range and are more suited to urban settings with **Line-Of-Sight (LOS)** communication. Higher frequencies are desirable for technologies such as cell phone communication because they can carry more data, providing greater capacity for voice and data transmission. In *Figure 9.3*, we see an illustration of the frequency spectrum used for wireless communications and associated technologies that operate in specific ranges:



Figure 9.3 – The frequency spectrum for wireless communications

LoRa operates in the unlicensed frequency range of 867-869 MHz in Europe, 902-928 MHz in North America, and 915-928 MHz in Australia (*Figure 9.4*). These frequencies are chosen for their balance of range and penetration, ideal for the low-power, long-range communication that LoRa enables.

The specific frequency bands can vary within the ranges shown in *Figure 9.4* based on national regulations. We should always check local regulations to ensure compliance with the specific frequencies allowed:

| Location      | Frequency Band |
|---------------|----------------|
| Europe        | 867 - 869 MHz  |
| North America | 902 - 928 MHz  |
| Australia     | 915 - 928 MHz  |
| Asia          | 923 - 925 MHz  |
| South Korea   | 920 - 925 MHz  |
| Japan         | 920 - 925 MHz  |
| India         | 865 - 867 MHz  |
| China         | 470 - 510 MHz  |

Figure 9.4 – LoRa frequencies based on location

The use of unlicensed bands comes with regulatory limitations to ensure fair usage and minimize interference. In Europe, the 868 MHz band is subject to a duty-cycle limitation of 1%, restricting transmission time. In North America, the 915 MHz band has dwell-time restrictions, limiting the occupancy time of a signal on a channel.

These limitations encourage efficient spectrum use and innovative communication protocol development, as seen in LoRa's effectiveness within these frameworks. We will consider these limitations when we start writing LoRa code in the upcoming section, *Building a LoRa sensory transmitter*.

Now that we have a basic understanding of the frequency spectrum and LoRa's position within it, let's explore the **Spreading Factor (SF)**, a key parameter in LoRa that impacts the network's range, data rate, and power efficiency.

## Understanding the LoRa SF

LoRa employs **Chirp Spread Spectrum (CSS)** technology, where signals vary in frequency over time to encode data. This technique boosts signal reliability and minimizes power consumption, making it ideal for IoT devices that operate over extended periods. The use of a broad frequency range in LoRa enables devices to maintain connectivity over long distances while consuming less power. In this process, data transmission involves changing the signal's frequency across a wide spectrum, significantly enhancing its resistance to interference and noise.

Complementing this, LoRa's SF, a key parameter in LoRa communication, ranges from SF7 to SF12. The SF determines the duration of each symbol (data packet) transmission, essentially balancing transmission range and data rate. Higher SFs, such as SF12, extend the range but reduce the data rate, making them suitable for long-distance communication. In contrast, lower SFs such as SF7 offer faster data rates over shorter distances. This flexibility allows LoRa to cater to a wide array of use cases, from densely populated urban areas in smart cities to remote areas requiring long-range monitoring.

In *Figure 9.5*, we see the SF illustrated in an agricultural setting:

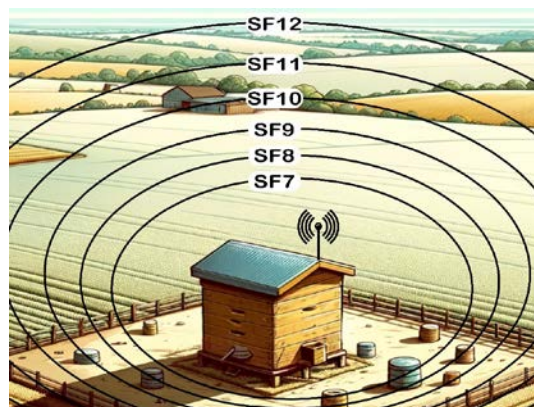


Figure 9.5 – SF illustrated in an agricultural setting

The range increases with the SF number, but higher SFs also require more power due to extended transmission times. For effective communication, it's essential that both the sender and receiver are configured with the same SF.

Now that we've explored LoRa communication technology, let's incorporate what we've learned into the Raspberry Pi Pico and Pico W.

## Using LoRa with the Raspberry Pi Pico and Pico W

In this chapter, we will develop both a LoRa sensory transmitter and a LoRa receiver. Our LoRa sensory transmitter will utilize the Raspberry Pi Pico, a DHT22 temperature sensor, and an RFM95W LoRa module. Our LoRa receiver will use a Raspberry Pi Pico W, an LED, and an RFM95W LoRa module. We may see our application outlined in *Figure 9.6*:

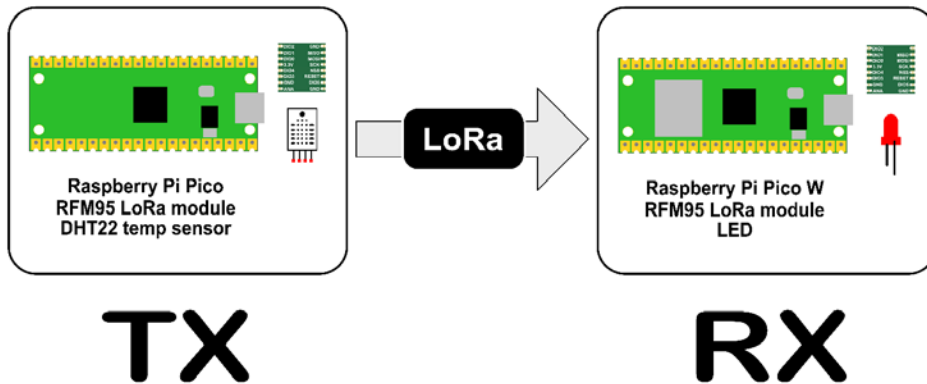


Figure 9.6 – LoRa sensory transmitter and LoRa receiver

We will begin creating our application by building a LoRa sensory transmitter. We will start with wiring up an RFM95W LoRa module to a Raspberry Pi Pico before we move on to installing the temperature sensor in a test circuit. Finally, we will install the components into a custom 3D-printed case.

## Building a LoRa sensory transmitter

In this section, we will build a LoRa sensory transmitter with a temperature sensor using a Raspberry Pi Pico, an RFM95W LoRa module, and a DHT22 temperature sensor. Our goal is to create a device that reads temperature and humidity data from a DHT22 temperature sensor and transmits this information using the RFM95W LoRa module. We will use the Raspberry Pi Pico and not Pico W for our design, although we could just as easily swap out the Pico for the Pico W.

For building and testing our circuit, we will use a Raspberry Pi Pico WH installed on a Pico GPIO expander. Using a Pico WH allows us to build and test circuits for both Pico and Pico W applications. We will use Pico W for the receiver part of our application and not the Pico WH due to size constraints.

**Advantages of using a standard Pico over a Pico W**

Besides its cost advantage, the Raspberry Pi Pico offers benefits over the Raspberry Pi Pico W for our LoRa sensory transmitter. The standard Pico, known for its lower power consumption, is well-suited for tasks that don't require Wi-Fi. Its simpler design leads to reduced firmware overhead, allowing the device to focus its resources on specific tasks rather than managing Wi-Fi connectivity.

We'll also build a custom case to house our components. This case will not only offer protection but will also ensure organized wiring and enhance the overall durability and portability of our LoRa sensory transmitter. We will start off by adding wires to the RFM95W LoRa module and wiring it up to our Raspberry Pi Pico WH for development. We will later replace the Pico WH with a Pico as we install our components into a custom case.

## Constructing our circuit

The RFM95W is a compact LoRa module favored for its long-range capabilities and low power consumption. Designed for sub-GHz frequency operation, it's well-suited for applications that demand efficient, long-distance wireless communication. Despite its small size, the RFM95W excels in covering greater distances than conventional wireless technologies, making it particularly effective in open environments.

We will start by soldering jumper wires onto the RFM95W.

***Adding wires to the RFM95W***

Measuring only 16 mm by 16 mm, the RFM95W module is tiny. Care must be taken when soldering the jumper wires to the holes of the RFM95W. This is not a soldering job for someone first learning to solder.

In *Step 1* of *Figure 9.7*, we see the RFM95W prior to adding our jumper wires. The type of jumper wires we add depends on the breadboarding option we choose. For example, in this case, we are adding female jumper wires (*Step 2* of *Figure 9.7*) as we will be using a GPIO expander for our Raspberry Pi Pico WH:

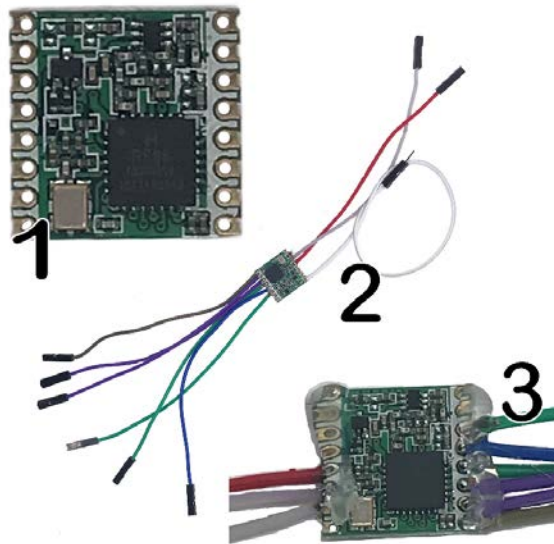


Figure 9.7 – Adding wires to the RFM95W

| RFM95W Terminal | Wire Length                                                                                                             |
|-----------------|-------------------------------------------------------------------------------------------------------------------------|
| DIO0            | 15 - 20cm                                                                                                               |
| 3.3V            | 15 - 20cm                                                                                                               |
| MISO            | 15 - 20cm                                                                                                               |
| MOSI            | 15 - 20cm                                                                                                               |
| SCK             | 15 - 20cm                                                                                                               |
| NSS             | 15 - 20cm                                                                                                               |
| RESET           | 15 - 20cm                                                                                                               |
| Antenna         | The length of the antenna wire depends on the frequency used:<br>433 MHz - 16.5cm<br>868 MHz - 8.2cm<br>915 MHz - 7.8cm |

Figure 9.8 – RFM95W pins for wiring



We do not need to solder wires to every terminal of the RFM95W. In *Figure 9.8*, we outline the terminals that we require jumper wires for and the length of wire needed:

Our table illustrates that the length of the wire used for our antenna differs based on the specific frequency model of the RFM95W LoRa module we choose. To account for this variation, we've designed different versions of our custom case to accommodate each model.

Each case version incorporates a straight antenna wire enclosure, effectively creating a built-in antenna specifically tailored for the respective frequency. This aspect of the design is particularly appealing for those of us who have encountered issues with low-cost antennas that don't accurately match their labeled frequencies.

#### Adafruit RFM95W LoRa Radio Transceiver Breakout

For those of us who wish to work with a LoRa board that is larger than the standard RFM95W, the Adafruit RFM95W LoRa Radio Transceiver Breakout is a great option. Unlike our RFM95W, this board is breadboard-friendly, using header pins.

To reinforce the soldered connections of the jumpers on our RFM95W, we can apply glue from a hot glue gun, as detailed in *Step 3* of *Figure 9.7*. We use these jumpers to plug the module into the Raspberry Pi Pico WH GPIO expander for initial testing and prototyping.

### Assembling our circuit

With the jumper wires soldered to our RFM95W model, we may now construct our circuit on a Raspberry Pi Pico GPIO expander. *Figure 9.9* illustrates a wiring diagram for the Raspberry Pi Pico, Raspberry Pi Pico WH, and the RFM95W LoRa module:

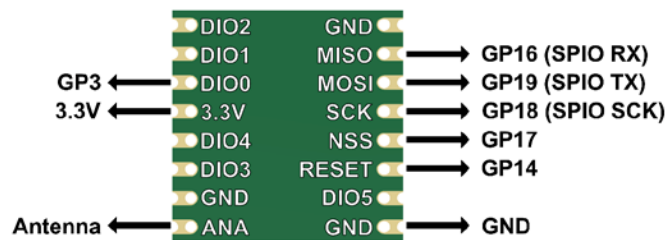


Figure 9.9 – Wiring the RFM95W module to the Raspberry Pi Pico

To complete our circuit, we add a DHT22 temperature sensor to our circuit using the wiring diagram in *Figure 9.10*:

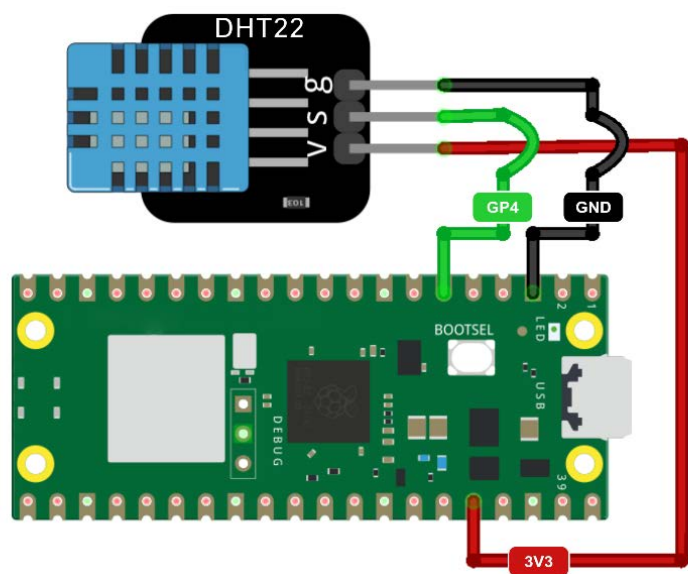


Figure 9.10 – Wiring up a DHT22 to the Raspberry Pi Pico (WH)

In *Figure 9.11 (A)*, we get a practical view of our circuit layout using a Raspberry Pi Pico with a GPIO expander (before the DHT22 is added). In our example, we’re utilizing a Raspberry Pi Pico WH and the GPIO expander for circuit construction and testing:

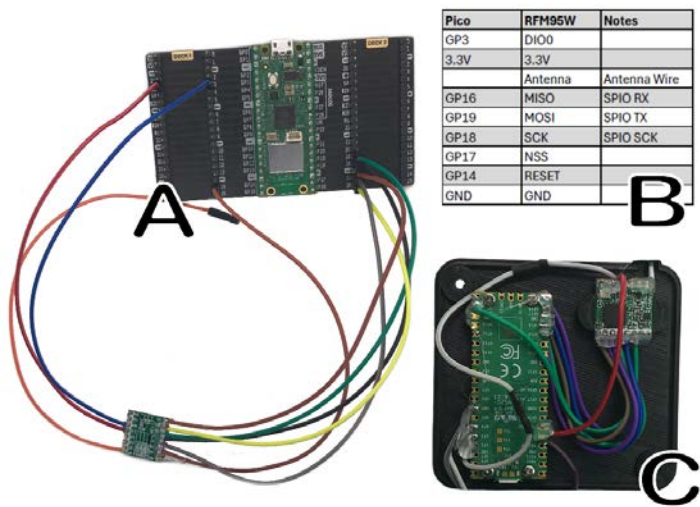


Figure 9.11 – Raspberry Pi Pico WH with GPIO expander and RFM95W LoRa module

We can see a table with the wire connections in *Figure 9.11 (B)* and what our Raspberry Pi Pico and RFM95W will look like wired together in the case in *Figure 9.11 (C)*.

With our circuit wired up, we are now ready to write our code to send temperature and humidity data through LoRa. We will use Thonny as our development environment and the CircuitPython firmware.

## Developing the code

For our application, we will use **CircuitPython** over **MicroPython** due to several key differences between these Python implementations. CircuitPython, a derivative of MicroPython developed by Adafruit, offers a more streamlined experience for specific use cases, particularly with its comprehensive library support.

We will start our development by installing the CircuitPython firmware onto our Raspberry Pi Pico WH.

### *Setting up CircuitPython and sensor libraries*

We develop our application for the Raspberry Pi Pico (WH) using the Thonny IDE, a tool that's compatible with various development environments such as Raspberry Pi, Windows, Linux, and macOS. For our example, we are using Thonny on Windows.

In terms of setting up the Raspberry Pi Pico (WH), the installation of CircuitPython via Thonny is a straightforward process, like how we would install MicroPython.

To install CircuitPython on our Raspberry Pi Pico (WH), we do the following:

1. If Thonny is not available on our operating system, we visit the Thonny website and download an appropriate version (<https://thonny.org>).
2. We then launch Thonny using the appropriate method for our operating system.
3. While holding the *BOOTSEL* button on the Pico (WH), the small white button near the USB port, we insert it into an available USB port and disregard any pop-up windows that may appear:

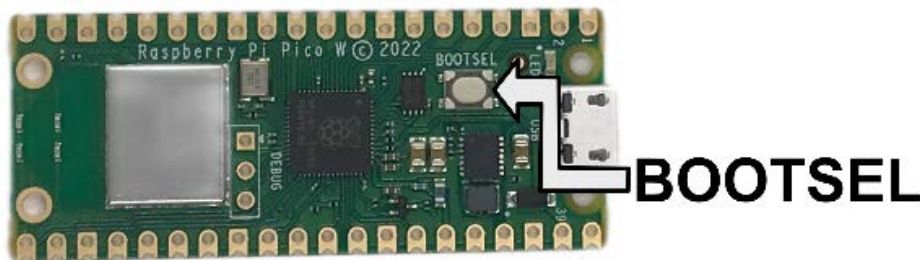


Figure 9. 12 – BOOTSEL button as shown on a Pico WH (Pico similar)

4. We then click on the interpreter information at the bottom right-hand side of the screen and select **Install CircuitPython...**:

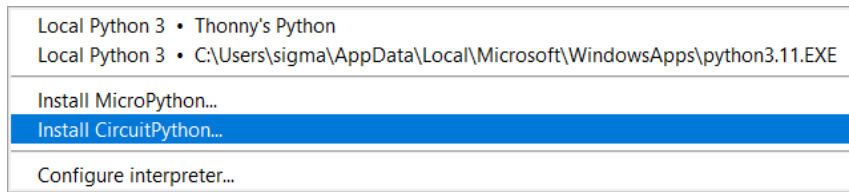


Figure 9.13 – Install CircuitPython... option

5. For **Target volume**, we select our Pico WH (RPI - RP2 (D : ) ). In our example, we select the **Raspberry Pi • Pico / Pico H** CircuitPython variant and the latest version:

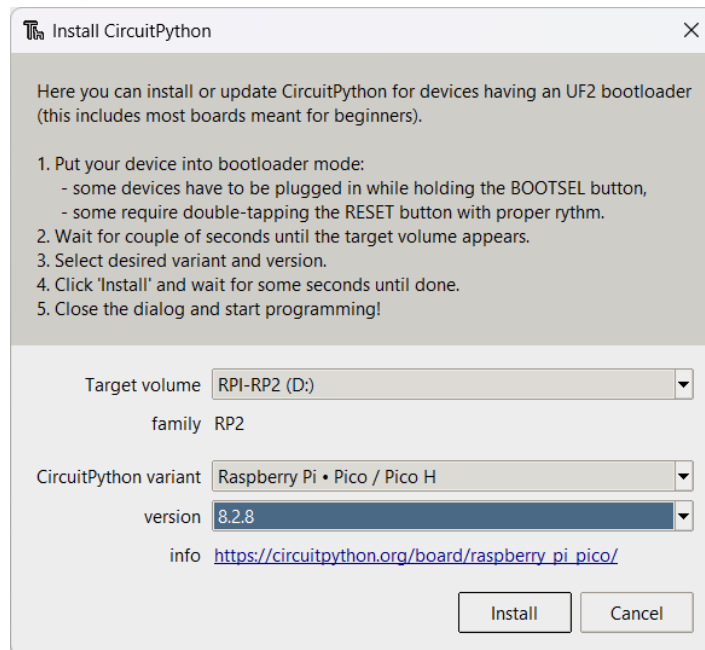


Figure 9.14 – Installing MicroPython on the Raspberry Pi Pico W

6. Even though we are developing on a Pico WH, we treat it as a Pico for development purposes. We click on the **Install** button and then the **Close** button once the installation has completed.

7. To have Thonny configured to run the CircuitPython interpreter on our Pico (WH), we select it from the bottom right-hand side of the screen:

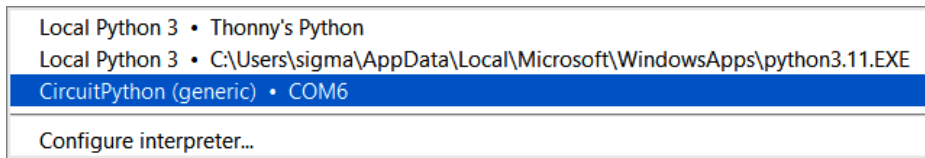


Figure 9.15 – Selecting the CircuitPython interpreter

8. We confirm that Thonny is using the CircuitPython interpreter on our Raspberry Pi Pico (WH) by checking the **Shell**:

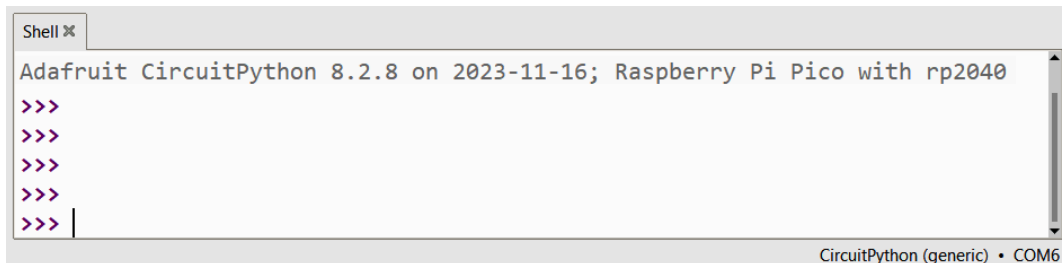


Figure 9.16 – CircuitPython prompt in Thonny

With CircuitPython installed, we are now ready to install the libraries we need for our code. This involves downloading the package from the Adafruit website and copying over the library files we need to our Raspberry Pi Pico (WH).

To do this, we do the following:

1. Using a web browser, we navigate to the following URL: <https://circuitpython.org/libraries>.
2. As we are using CircuitPython 8, we download the `adafruit-circuitpython-bundle-8.x-mpy-20231205.zip` ZIP file and unzip it to a location on our computer.
3. The two files we are interested in are `adafruit_rfm9x.mpy` and `adafruit_dht.mpy`, both of which may be found in the `lib` folder in the unzipped directory. These files are library files for our RFM95W and DHT22 sensors respectively. To install these libraries onto our Raspberry Pi Pico (WH) from Thonny, we locate them in the **Files** section and right-click to get the following dialog:

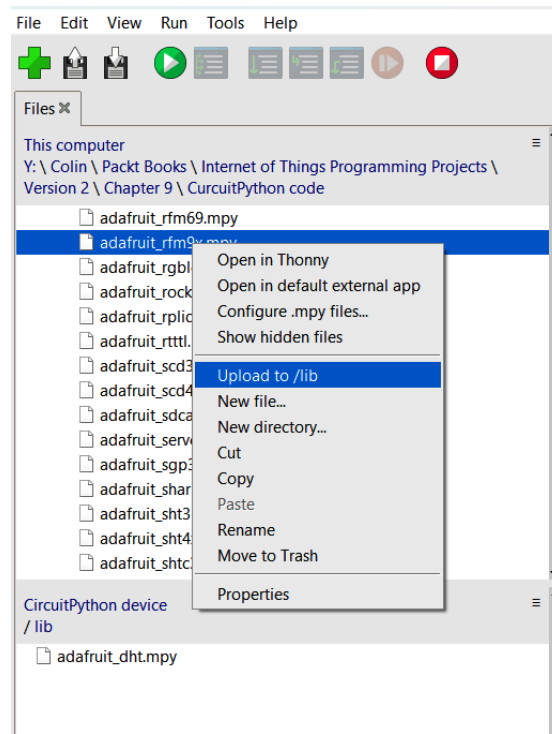


Figure 9.17 – Uploading a library file to the Raspberry Pi Pico (WH)

4. We must ensure that we upload these libraries to the `lib` folder on our Pico (WH) and not the root directory. This would involve double-clicking on the `lib` folder under the **CircuitPython device** section in Thonny to open it. After uploading the libraries to the Pico (WH), the file structure on our Pico (WH) should look like the following:

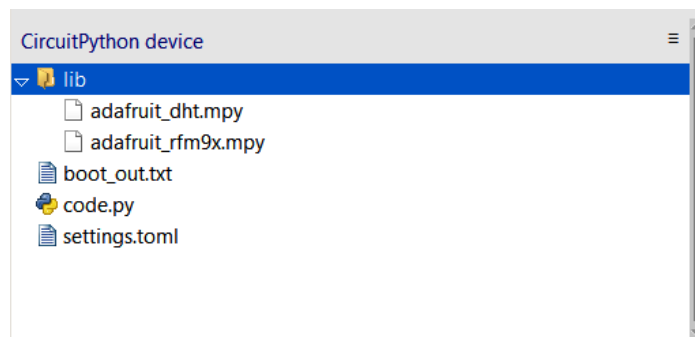


Figure 9.18 – Pico file structure after uploading libraries

With our circuit built and CircuitPython and the libraries installed, it is time to write our code. As we will see, we do not require a great amount of code to send temperature and humidity data through LoRa.

### ***Creating LoRa transmission code***

To comply with different regulatory standards, we'll use a delay for our LoRa transmissions. While Europe's 1% duty-cycle limit allows for a 99-second delay between messages, we'll extend this to 120 seconds, due to the stable nature of temperature and humidity data. This will result in a duty cycle of approximately 0.83%.

Although North America's regulations focus on dwell time, the period a transmitter occupies a frequency channel rather than a duty cycle, we're adopting this duty-cycle approach for uniformity. We will use the 915 MHz frequency version of the RFM95W for our application as the author is based in North America.

To write our LoRa transmission code, we do the following:

1. We connect our Raspberry Pi Pico (WH) to a USB port and launch Thonny. We may use our Raspberry Pi or another operating system for this. In our example, we are using Thonny on Windows.
2. We then activate the CircuitPython environment on our Pico (WH) by selecting it from the bottom right-hand side of the screen.
3. In a new tab, we enter the following code:

```
import time
import board
import busio
import digitalio
import adafruit_rfm9x
import adafruit_dht

spi = busio.SPI(board.GP18, MOSI=board.GP19, MISO=board.GP16)
cs = digitalio.DigitalInOut(board.GP17)
rst = digitalio.DigitalInOut(board.GP14)

rfm9x = adafruit_rfm9x.RFM9x(spi, cs, rst, 915.0)

dht_sensor = adafruit_dht.DHT22(board.GP4)

print("Sending temperature and humidity data every 120 seconds")

while True:
 try:
```

```
 temperature = dht_sensor.temperature
 humidity = dht_sensor.humidity
 data = f"Temp: {temperature}C, Humidity: {humidity}%"
 rfm9x.send(bytes(data, "utf-8"))
 print("Sent:", data)
 except RuntimeError as e:
 print("DHT22 read error:", e)

 time.sleep(120)
```

In our code, we do the following:

- I. **We start by importing the necessary libraries:** We import `time`, `board`, `busio`, `digitalio`, `adafruit_rfm9x` (for LoRa communication), and `adafruit_dht` (for the DHT22 sensor).
- II. **We then set up SPI communication:** We configure SPI with specific GPIO pins (GP18, GP19, GP16) for the RFM95W LoRa module.
- III. **We initialize Chip Select (CS) and Reset (RST) pins:** We set up digital I/O for CS (GP17) and RST (GP14) pins.
- IV. **We then create an RFM95W LoRa object:** We initialize the RFM9x object for LoRa communication at 915.0 MHz. This value should be set based on local regulations.
- V. **We initialize the DHT22 sensor:** We set up the DHT22 temperature and humidity sensor on GPIO 4 (GP4).
- VI. **We then print a status message:** We indicate that temperature and humidity data will be sent every 120 seconds.
- VII. **We continuously send data:** In an infinite loop, we read temperature and humidity from the DHT22 sensor, format the data, and send it over LoRa. If a read error occurs, we print an error message.
- VIII. **We then delay between transmissions:** We wait for 120 seconds before sending the next set of data.



4. To save the file, we click on **File | Save as...** from the drop-down menu. This will open the following dialog:

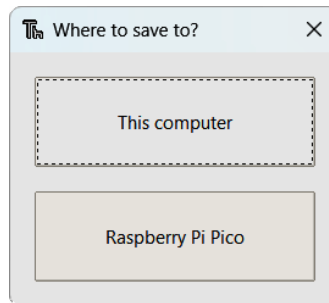


Figure 9.19 – Saving a file to our Raspberry Pi Pico (WH)

5. In this dialog, we are given the option to choose where to store our file. To save it on our Raspberry Pi Pico (WH), we click on the corresponding button.
6. We then give the file the name `code.py` and click **OK**. In the CircuitPython environment, the name `code.py` is special because the system automatically executes this file upon startup or reset, making it the default script that runs when the device powers up.

#### What SF are we using?

In our code, we don't explicitly set the SF, so the `adafruit_rfm9x` library's default is used. Typically, this is an SF of 7. Since we're using the default SF for both `transmit` and `receive` nodes, there's no need to focus on this setting in our application.

7. To run our code, we click on the green run button, hit *F5* on the keyboard, or click on the **Run** menu option at the top and then **Run current script**.
8. In the **Shell**, we'll see a notification confirming the creation of a LoRa message containing temperature and humidity data:

```
Shell ✕
>>> %Run -c $EDITOR_CONTENT
Sending temperature and humidity data every 120 seconds
Sent: Temp: 23.7C, Humidity: 32.9%
```

Figure 9.20 – LoRa message notification

To recap, we have just created a LoRa sensory transmitter that sends temperature and humidity sensory data wirelessly. In the absence of errors, it's reasonable to assume successful transmission of our LoRa message. However, without a LoRa receiver, we can't confirm this. We'll address this by building one in the next section. Before constructing the receiver, we'll first house our components in a custom 3D-printed case.

## Installing the components in a custom case

Continuing our practice from previous projects, we'll install our components in a custom case. This approach allows convenient placement of our LoRa sensory transmitter wherever it's needed. We can see the custom case for our LoRa sensory transmitter displayed in *Figure 9.21*.

The design of our custom case accommodates the DHT22 sensor, allowing it to extend from the front for accurate temperature and humidity readings. The antenna, a straight wire soldered to the RFM95W module, is housed in a protruding section attached to the base plate of the case. An antenna cover specifically designed for this purpose completes the enclosure, protecting and isolating the antenna. The 915 MHz version of the custom case is displayed in *Figure 9.21*:

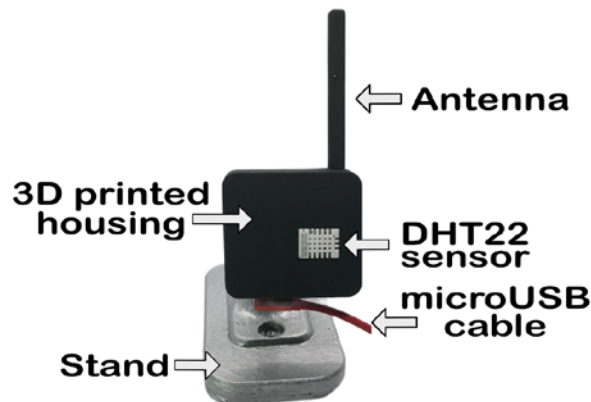


Figure 9.21 – Custom case used for our LoRa transmitter node

Our custom case is designed with a versatile GoPro-style hook at the back, enabling compatibility with various stands we've constructed in earlier chapters of the book. This feature allows for easy and flexible mounting of our LoRa sensory transmitter.

The micro-USB port on the Raspberry Pi Pico is exposed, allowing us to provide power to our device. We may also use this port to program our Raspberry Pi Pico.

### Using power banks for our Raspberry Pi Pico

We can power our node remotely using a standard cell phone power bank. However, it's important to select a power bank that doesn't automatically shut off due to low power draw, as the Raspberry Pi Pico has minimal power requirements.

We will start the construction of our custom case by identifying the parts.

### Identifying the parts of our custom LoRa case

Our custom case features 3D-printed parts that screw together. We may see the parts and major components displayed in *Figure 9.22*:

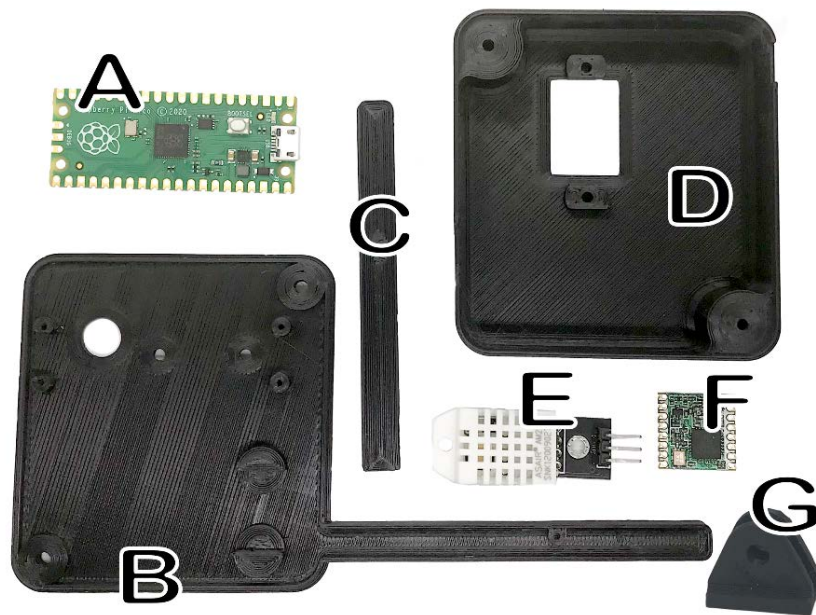


Figure 9.22 – Parts for custom case

Let's break down each part:

- Raspberry Pi Pico (A in *Figure 9.22*): We use the header-less version of the Raspberry Pi Pico due to space constraints.
- Backplate (B in *Figure 9.22*): The backplate secures both the Raspberry Pi Pico and the RFM95W LoRa module. The length of the extension for the antenna is based on the frequency model used. In this example, we see the 915 MHz model.
- Antenna cover (C in *Figure 9.22*): The antenna cover is used to enclose the wire antenna with the back plate (B).

- Front shell (*D* in *Figure 9.22*): The front shell holds the DHT22 sensor and encloses the back plate (*B*) to complete the case.
- DHT22 temperature and humidity sensor (*E* in *Figure 9.22*).
- RFM95W LoRa module (*F* in *Figure 9.22*): The version shown is the 915 MHz model.
- Hook (*G* in *Figure 9.22*).
- 2 x M3 10 mm bolts (not shown).
- 4 x M2 5 mm screws (not shown).

With the parts identified, it is now time to assemble our custom case.

### ***Building the custom LoRa case***

To build the custom case, we follow the steps shown in *Figure 9.23* and outlined next:



Figure 9.23 – Steps to build custom case

1. We start by securing the hook (*G* in *Figure 9.22*) to the back plate (*B* in *Figure 9.22*) using either epoxy glue or two M2 5 mm screws (*Figure 9.23*, Step 1).
2. Using the wiring diagrams from *Figures 9.9* and *9.10*, we solder the wires from the RFM95W and DHT11 sensors to the Raspberry Pi Pico.

3. Using four M2 5 mm screws, we secure the Raspberry Pi Pico (A in *Figure 9.22*) to the back plate (B in *Figure 9.22*) such that the USB port is facing down and pointing toward the bottom of the back plate or away from the antenna (*Figure 9.23, Step 2*).
4. Using a hot glue gun, we secure the DHT22 (E in *Figure 9.22*) to the front shell (D in *Figure 9.22*). Alternatively, two M3 5 mm bolts may be used depending on the holes present on the DHT22 (*Figure 9.23, Step 3*).
5. We friction fit the RFM95W (F in *Figure 9.22*) to the back plate (B in *Figure 9.22*) such that the antenna wire sits next to the wire slot on the back plate and extends through (*Figure 9.23, Step 3*). If the RFM95W does not stay in place, glue from a hot glue gun may be used to secure it in place.
6. Using two M3 10 mm, bolts we secure the back plate to the front shell (*Figure 9.23, Step 4*).
7. We secure the antenna cover (C in *Figure 9.22*) to the front of the back plate using two M2 5 mm screws.

As we used a separate Raspberry Pi Pico WH for writing our coding, we'll need to install CircuitPython, the necessary libraries, and our code onto the new Pico. Testing the code with Thonny is advisable to check for any issues that might arise during the installation of components into the custom case.

Now that our LoRa sensory transmitter is ready, we'll proceed to construct a LoRa receiver. This device will be responsible for receiving sensory information from the transmitter. We'll keep the receiver design simple, with its primary function being to acknowledge the receipt of LoRa messages. For this purpose, an LED will suffice to indicate the status.

## Building a LoRa receiver

In *Figure 9.6*, we see a LoRa receiver processing messages from our LoRa sensory transmitter. Our receiver design is straightforward, requiring only an LED to acknowledge received messages. We're utilizing a Raspberry Pi Pico W for the receiver as we plan to leverage its Wi-Fi capabilities in the next chapter and publish sensory data to the internet.

We will not cover the steps to install CircuitPython or the required libraries onto the Pico W of the LoRa receiver as we covered these steps already for the LoRa sensory transmitter, and we merely need to do the same for the receiver. We should use the Pico W version of CircuitPython for this part of the project as we will implement Wi-Fi functionality in the next chapter.

Also, we won't detail the construction of the custom case for the LoRa receiver as it mirrors the transmitter's process. The key variation is fitting an LED with a resistor and LED holder in place of the DHT22 sensor in the front shell. The steps for installing an LED were previously outlined in *Figures 6.22 and 6.23 of Chapter 6*.

In this section, we will focus on the code for the LoRa receiver and highlight the results of an outdoor test of both the LoRa sensory transmitter and the LoRa receiver.

We will start with a wiring diagram of the LED with a resistor to our Raspberry Pi Pico W.

## Wiring an LED to the Raspberry Pi Pico W

For our LoRa receiver, we require an LED to use for acknowledging LoRa messages. Based on what we have learned so far in this book, we could easily enhance our LoRa receiver with a more robust visualization such as an OLED screen. As we aim to focus only on acknowledging a LoRa signal, we will stick with a simple LED. We may monitor LoRa messages from the Shell in Thonny while running our LoRa receiver in that environment:

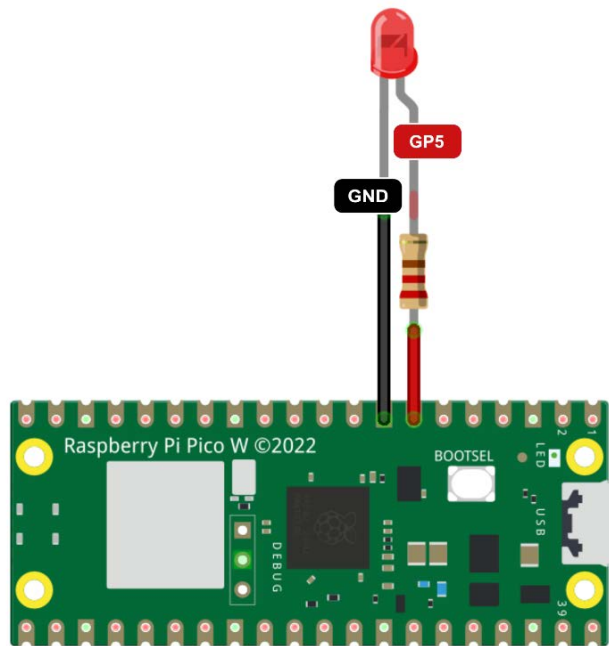


Figure 9.24 – Raspberry Pi Pico W, 220 Ohm resistor, and LED

To connect an LED to our Raspberry Pi Pico W, we solder a 220 Ohm resistor to the anode of an LED. We then connect the resistor to the GP5 port on the Pico. For ground, we connect the cathode of the LED to any GND pin on the Pico W (*Figure 9.24*).

With the 220 Ohm resistor and LED connected to our Pico W, we then connect an RMM95W LoRa module to our Pico W using the steps from the previous section, *Building a LoRa sensory transmitter*.

We are now ready to start writing code.

## Creating code to receive LoRa messages

Our LoRa receiver code uses the Adafruit `adafruit_rfm9x` library to listen for LoRa messages. Upon receipt, it prints the message to the Shell in Thonny and flashes the LED twice.

To write our LoRa receiver code, we do the following:

1. We connect our Raspberry Pi Pico W to a USB port and launch Thonny. We may use our Raspberry Pi or another operating system for this.
2. We then activate the CircuitPython environment on our Pico W by selecting it from the bottom right-hand side of the screen.
3. In a new tab, we enter the following code:

```
import time
import board
import busio
import digitalio
import adafruit_rfm9x

spi = busio.SPI(board.GP18, MOSI=board.GP19, MISO=board.GP16)
cs = digitalio.DigitalInOut(board.GP17)
rst = digitalio.DigitalInOut(board.GP14)

rfm9x = adafruit_rfm9x.RFM9x(spi, cs, rst, 915.0)

led = digitalio.DigitalInOut(board.GP5)
led.direction = digitalio.Direction.OUTPUT

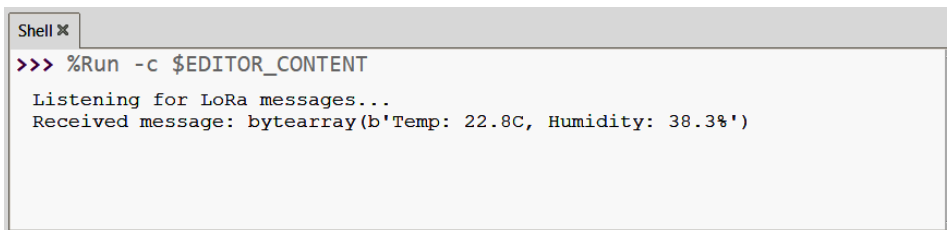
print("Listening for LoRa messages...")

def flash_led(times, duration):
 for _ in range(times):
 led.value = True
 time.sleep(duration)
 led.value = False
 time.sleep(duration)

while True:
 packet = rfm9x.receive()
 if packet is not None:
 print("Received message:", packet)
 flash_led(2, 0.5)
```

In our code, we do the following:

- I. We start by importing the necessary libraries: `time`, `board`, `busio`, `digitalio`, and `adafruit_rfm9x` for LoRa communication.
  - II. We then set up SPI communication by configuring SPI using GPIO pins GP18 (SCK), GP19 (MOSI), and GP16 (MISO).
  - III. We initialize CS and RST pins by setting up GP17 for CS and GP14 for RST.
  - IV. We create an RFM95W LoRa object and initialize the object for 915.0 MHz communication.
  - V. We initialize an LED on GP5.
  - VI. We then print a status message indicating that the device is listening for LoRa messages.
  - VII. We define a function to flash the LED so that we may flash the LED a specified number of times with a set duration.
  - VIII. In a continuous loop, we listen for LoRa messages, checking for incoming LoRa packets, printing any received message, and flashing the LED twice for 0.5 seconds each upon receiving a packet.
4. To save the file, we click on **File | Save as...** from the drop-down menu. This will open the **Where to save to?** dialog.
  5. In this dialog, we are given the option to choose where to store our file. To save it on our Raspberry Pi Pico W, we click on the corresponding button.
  6. We then give the file the name `code.py` and click **OK**.
  7. To run our code, we click on the green run button, hit *F5* on the keyboard, or click on the **Run** menu option at the top and then **Run current script**.
  8. If it is not already running, we power up and run the LoRa sensory transmitter.
  9. In the Shell, we'll see a notification confirming the receipt of a LoRa message:



```
Shell ✕
>>> %Run -c $EDITOR_CONTENT
Listening for LoRa messages...
Received message: bytearray(b'Temp: 22.8C, Humidity: 38.3%')
```

Figure 9.25 – Receiving LoRa messages

10. We should also observe our LED flashes twice.



With a positive result, we not only confirm that our LoRa receiver is working properly but our LoRa sensory transmitter as well. To take full advantage of our application, we need to use our transmitter and receiver outside. To do this, we should install our LoRa receiver in its own custom case. As mentioned, we follow the same steps outlined for the LoRa sensory transmitter substituting the LED with a resistor and the LED holder with a DHT22 temperature sensor.

With both the LoRa sensory transmitter and LoRa, it is time to take our application outdoors.

## Testing our application

LoRa communication is known for its ability to cover long distances, a feature that sets it apart in the field of wireless communication technologies. While typical LoRa transmissions range from a few kilometers in urban settings to over 10 kilometers in rural areas, the technology has demonstrated far greater potential under optimal conditions. A world record was established with a LoRa transmission reaching 766 kilometers (476 miles) using just 25 mW of transmission power. This record highlights LoRa's exceptional long-range capabilities, especially when conditions are favorable and the setup is optimized for maximum reach.

In *Figure 9.26*, we observe the results of testing our application over a modest distance of 160 meters:

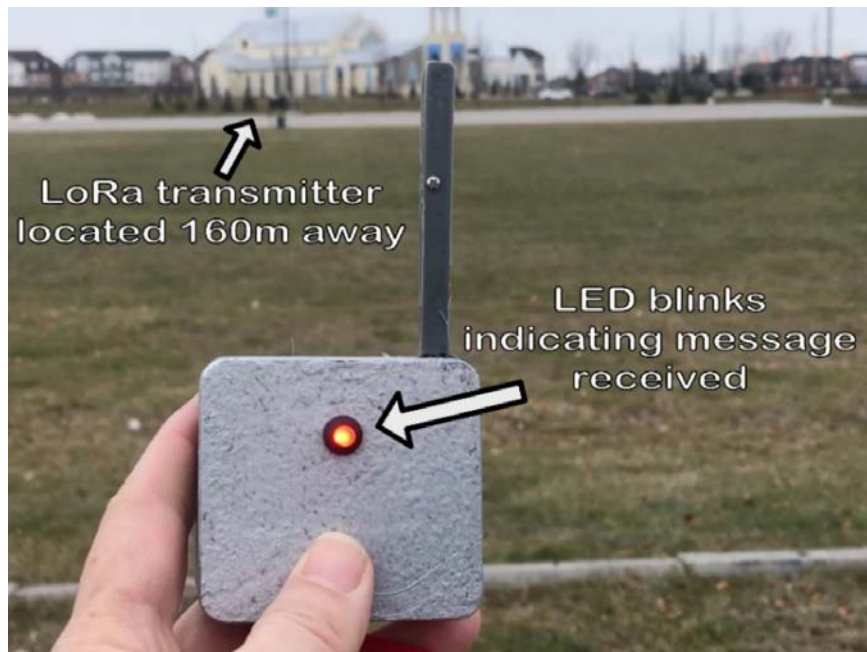


Figure 9.26 – Testing our LoRa sensory transmitter and LoRa receiver at a distance of 160 meters

**Alternative testing methods**

We may also test our application by applying heat from a source such as a hair dryer, or we may even place our LoRa transmitter in a fridge and observe the results.

This distance, while modest for LoRa, is significant when considering applications such as farm monitoring, where sensors can be spread over large areas, far beyond the reach of traditional Wi-Fi networks, which are typically constrained to about 50 meters indoors and less than 100 meters in open spaces.

## Summary

In this chapter, we explored LoRa technology, an important component in IoT communication. We started by discussing LoRa's ability to transmit data over long distances with low power use, emphasizing its importance in wireless technology. We looked at its applications in agriculture, where it improves sensory network management for crop management, and in urban settings for smart city initiatives such as street lighting control.

We then examined the technical aspects of LoRa within the radio frequency spectrum. This includes understanding how frequencies are allocated for wireless communications, which is important for identifying the right frequency bands for LoRa transmissions.

For our hands-on sections, we built a LoRa sensory transmitter and LoRa receiver using Raspberry Pi Pico and Pico W respectively. This involved constructing a transmitter with a LoRa module and temperature sensor and a LoRa receiver using a simple LED acknowledgment.

In the next chapter, we will take things further as we connect our LoRa receiver to the internet and use this to control a new version of the analog-metered weather indicator we created in *Chapter 3*.



# Integrating LoRa with the Internet

In this chapter, we will take the sensory information from our remotely placed LoRa sensory transmitter and publish it to the internet using our Raspberry Pi Pico W-equipped **LoRa** receiver. Our internet destination will be an MQTT instance on our CloudAMQP server. We will then connect a modified version of the analog-metered weather indicator we created in *Chapter 3* to our MQTT server and use this data to position the needle and set the color of the LED indicator based on the humidity reading. This modification will entail swapping out the Raspberry Pi we installed on the analog-metered weather indicator for a Raspberry Pi Pico WH (also referred to as a Raspberry Pi Pico W with headers) and the single-color LED with an RGB LED.

By transitioning from a standard weather web service to utilizing a remote LoRa sensory transmitter that measures temperature and humidity, and subsequently converts this data into MQTT messages, we are effectively creating a customized weather web service – a service that is powered by LoRa technology for its data transmission needs and MQTT for internet communication.

We will conclude this chapter with a look at various other technologies for **Internet of Things (IoT)** communication, such as **LoRaWAN** and cellular technologies, and explore the benefits and drawbacks of each technology.

We will cover the following in this chapter:

- Connecting our LoRa receiver to the internet
- Creating a new weather indicator
- Exploring other IoT communication protocols

Let's begin!

## Technical requirements

The following are the requirements for completing this chapter:

- Intermediate knowledge of Python programming.
- 1 X Raspberry Pi Pico WH.
- 1 X LoRa sensory transmitter from *Chapter 9*.
- 1 X LoRa receiver built with the Raspberry Pi Pico W from *Chapter 9*.
- A CloudAMQP account for the MQTT server instance.
- 1 X SG90 servo motor.
- 1 X common anode RGB LED.
- 3 X 220 Ohm resistors.
- 1 X 8 mm LED holder.
- 9 X M3 10 mm bolts.
- 4 X M2 8 mm screws.
- 1 X M5 20 mm bolt.
- 1 X M5 nut.
- Epoxy glue for constructing the weather indicator faceplate.
- A hot glue gun.
- A color printer to print out the faceplate graphic.
- A digital cutting machine such as a Silhouette Cameo. This is optional as the faceplate graphic may be cut out by hand. Silhouette Studio 3 file provided.
- Access to a 3D printer to print the weather indicator stand.

The code for this chapter may be found here: <https://github.com/PacktPublishing/Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter10>

## Connecting our LoRa receiver to the internet

Our first task is to revise the code on our CircuitPython-based LoRa receiver, enabling it to forward LoRa messages received by our CloudAMQP server. We'll use Adafruit CircuitPython libraries for this update. Before building and programming our new weather indicator to process MQTT messages, we'll test the MQTT functionality using the MQTT-Explorer app in Windows. We can see an outline of this chapter's project in the following diagram:

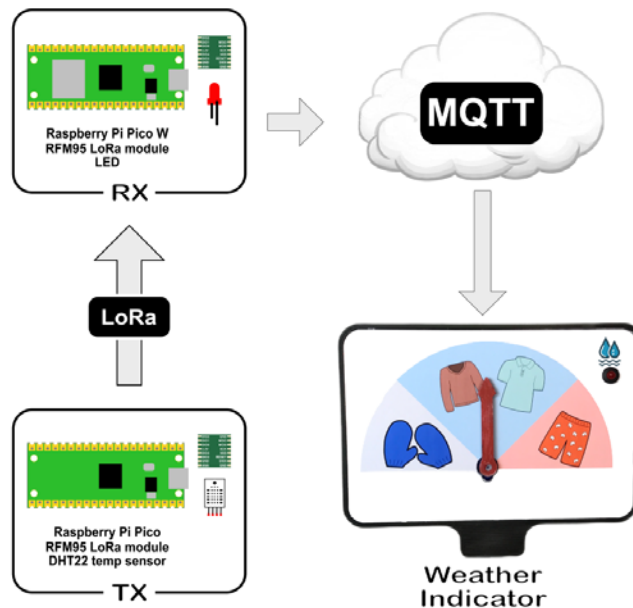


Figure 10.1 – Controlling an analog-metered weather indicator with LoRa-transmitted sensory data

In *Figure 10.1*, we can see that our Raspberry Pi Pico LoRa sensory transmitter from *Chapter 9* sends temperature and humidity data through LoRa to the LoRa receiver we also built in *Chapter 9*. We do not need to change the code for our transmitter as it performs what we need it to do. As mentioned, we will update the code for our LoRa receiver as we require it to connect to the internet and send MQTT messages.

Once we establish that our MQTT code is working properly, we will build a new version of the weather indicator and program it to respond accordingly to the MQTT messages.

We will start by adding the CircuitPython MQTT library to our LoRa receiver.

## Installing the CircuitPython library for MQTT

MQTT libraries for microcontrollers enable IoT devices to use the MQTT protocol, a messaging system optimized for low bandwidth and minimal device resources. These libraries, designed for various microcontroller platforms, enable connections to MQTT brokers, message publishing, and topic subscriptions.

Among these, the Adafruit MiniMQTT library is notable for CircuitPython devices. This library offers a straightforward API, suitable for boards such as the Raspberry Pi Pico W. It supports key MQTT features such as publish/subscribe and works with various MQTT brokers.

To install the MiniMQTT library, we do the following:

1. Using a web browser, we navigate to the URL <https://circuitpython.org/libraries>.
2. As we are using CircuitPython 8, we download the `adafruit-circuitpython-bundle-8.x-mpy-20231205.zip` ZIP file and unzip it to a location on our computer.
3. We want to install the library files in the `lib` folder on our Raspberry Pi Pico W and not the root directory. So, we need to double-click on the `lib` folder under the **CircuitPython** section in Thonny to open it.
4. The files we are interested in from the Adafruit library are in the `adafruit_minimqtt` folder. To install these files onto our Raspberry Pi Pico W from Thonny, we open the folder in the **Files** section in Thonny and right-click to get the following dialog:

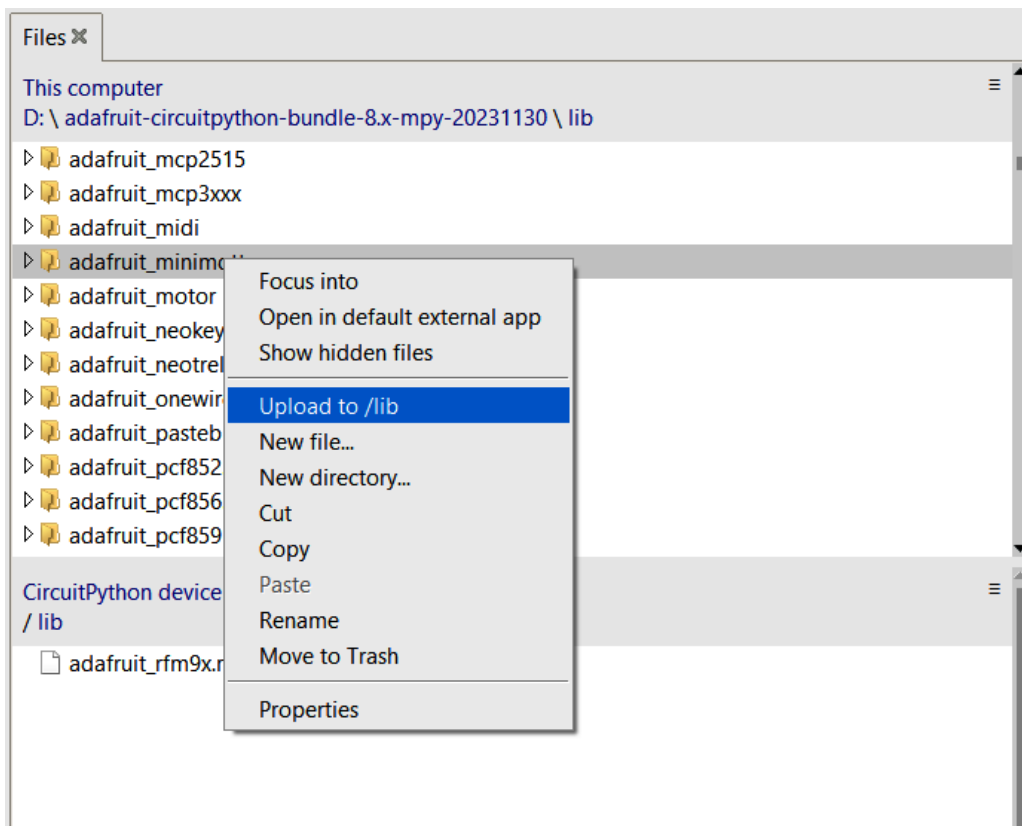


Figure 10.2 – Uploading the MQTT library to the Raspberry Pi Pico

5. After uploading the libraries to the Pico W, the file structure on our Pico should look like the following:

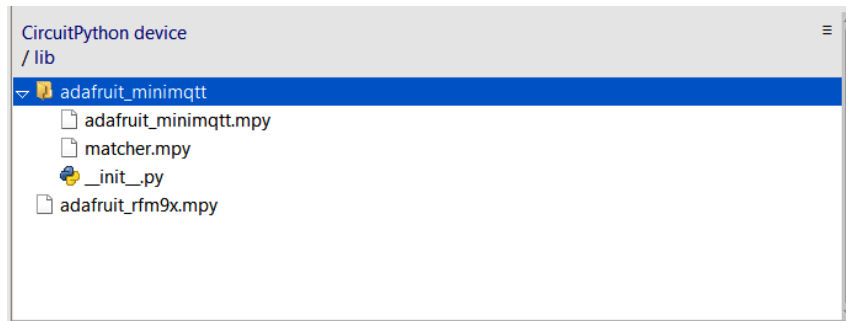


Figure 10.3 – Pico W file structure after uploading Mini MQTT

With the Adafruit MQTT library set up on our Raspberry Pi Pico W, our next step is to configure a CloudAMQP instance to broker our MQTT messages.

Let's do that now.

## Creating a CloudAMQP instance for our application

Having already set up an account in *Chapter 7*, we're now prepared to create a CloudAMQP instance to act as the broker for MQTT messages from our Raspberry Pi Pico W. Following the steps outlined in the *Setting up a CloudAMQP instance* section of *Chapter 7*, we create a new instance, call it `RemoteWeatherStation`, and record the details for use in our LoRa receiver code.

We'll utilize the WebSocket UI of our CloudAMQP instance to observe messages from our LoRa transmitter, enabling us to test our application thoroughly before constructing and implementing our new weather indicator.

Let's go ahead and modify the code on our LoRa receiver by adding MQTT functionality so that it can publish MQTT messages to the `RemoteWeatherStation` instance.

## Adding MQTT functionality to the LoRa receiver

With our CloudAMQP instance created, we will now enhance the code on our LoRa receiver to incorporate Wi-Fi and MQTT functionalities. This modification enables the receiver to effectively utilize LoRa to receive sensory data from our transmitter and then leverage Wi-Fi for internet connectivity.



Utilizing the MQTT protocol, the receiver can transmit this sensory data to our CloudAMQP server. This integration not only demonstrates the synergy between different communication technologies – LoRa for local, long-range transmission and MQTT over Wi-Fi for global reach – but also greatly expands the capabilities of our IoT ecosystem. Our LoRa receiver becomes a bridge, gathering sensory information locally and distributing it over the internet.

To modify the LoRa receiver code, we do the following:

1. We connect our Raspberry Pi Pico W from our LoRa receiver to a USB port and launch Thonny using a computer and operating system of our choice.
2. We then activate the CircuitPython environment on our Pico W by selecting it from the bottom right-hand side of the screen.
3. In a new tab in the editor section (hit *Ctrl* + *N*), we start by entering our imports:

```
import time
import board
import busio
import digitalio
import adafruit_rfm9x
import wifi
import socketpool
from adafruit_minimqtt.adafruit_minimqtt import MQTT
```

In our code, we have the following:

- The `time` library is used for timing and delays in the code.
- The `board` module provides access to the basic pin setup of the microcontroller.
- The `busio` module is used to create bus communication interfaces, essential for devices such as the RFM9x LoRa module.
- `digitalio` is utilized for managing digital I/O pins.
- The `adafruit_rfm9x` module is specifically for interfacing with the RFM95W LoRa radio modules installed on the LoRa receiver.
- The `wifi` module is included to handle Wi-Fi connections on the microcontroller.
- `socketpool` provides a way to manage network sockets, which are needed for internet communication. In our CircuitPython code, this module is essential for efficiently managing network sockets, which is crucial for MQTT communications via the `adafruit_minimqtt` library. It provides stable TCP/IP connections over Wi-Fi.
- `adafruit_minimqtt` is imported to enable MQTT protocol communication, allowing the device to publish and subscribe to MQTT topics.

4. After our imports, we set our Wi-Fi network (SSID) and Wi-Fi password:

```
WIFI_SSID = 'MySSID'
WIFI_PASSWORD = 'wifi-password'
```

5. We then enter code to initialize the GP5 GPIO pin as a digital output to control an LED connected to that pin on the microcontroller:

```
led = digitalio.DigitalInOut(board.GP5)
led.direction = digitalio.Direction.OUTPUT
```

6. We define the `flash_led()` method, which blinks an LED a specified number of times, with each blink lasting for the duration set in seconds:

```
def flash_led(times, duration):
 for _ in range(times):
 led.value = True
 time.sleep(duration)
 led.value = False
 time.sleep(duration)
```

7. Next, we define the `connect_to_wifi()` function. This function repeatedly tries to connect to Wi-Fi with the provided SSID and password, signaling a failed attempt by blinking an LED twice for 2 seconds each, followed by a 5-second pause before retrying. Upon successful connection, it exits the loop and blinks the LED four times, each for 1 second, to indicate a successful Wi-Fi connection:

```
def connect_to_wifi(ssid, password):
 while True:
 try:
 print("Trying to connect to WiFi...")
 wifi.radio.connect(ssid, password)
 print("Connected to Wi-Fi!")
 flash_led(4, 1)
 break
 except Exception as e:
 print("Failed to connect to WiFi. Retrying...")
 flash_led(2, 2)
 time.sleep(5)

connect_to_wifi(WIFI_SSID, WIFI_PASSWORD)
flash_led(4, 1)
```

8. We then create the `pool` variable, an instance of `socketpool.SocketPool`, using the `wifi.radio` object, which manages and provides network socket connections for Wi-Fi communication:

```
pool = socketpool.SocketPool(wifi.radio)
```

9. Next, our code sets up the MQTT configuration, defining the server address (`MQTT_SERVER`), port number (`MQTT_PORT`), user credentials (`USERNAME` and `PASSWORD`), device identifier (`DEVICE_ID`), and MQTT topic (`MQTT_TOPIC`) for communication. We obtain these values from the CloudAMQP instance we set up for our application:

```
MQTT_SERVER = "mqtt-server-url"
MQTT_PORT = 18756
USERNAME = "instance-username"
PASSWORD = "instance-password"
DEVICE_ID = "LoRaReceiver"
MQTT_TOPIC = "WeatherInfo"
```

10. We then configure the **Serial Peripheral Interface (SPI)** for the RFM95W LoRa module, setting up the SPI bus with GPIO pins GP18, GP19, and GP16 for SCK, MOSI, and MISO, respectively, and initialize digital I/O pins GP17 and GP14 for **chip select (CS)** and **reset (RST)** functions. The CS pin is used to select the LoRa module for communication, while the RST pin is employed to reset the module, ensuring it starts in a known state:

```
spi = busio.SPI(board.GP18, MOSI=board.GP19, MISO=board.GP16)
cs = digitalio.DigitalInOut(board.GP17)
rst = digitalio.DigitalInOut(board.GP14)
```

11. We initialize the RFM9x LoRa radio module by creating an instance of `adafruit_rfm9x.RFM9x` with the previously configured SPI bus (`spi`), chip select (`cs`), and reset (`rst`) pins, and set the operating frequency to 915.0 MHz as our example is built for use in North America:

```
rfm9x = adafruit_rfm9x.RFM9x(spi, cs, rst, 915.0)
```

12. We then set up an MQTT client by creating an instance of MQTT with the specified MQTT broker details (`MQTT_SERVER` and `MQTT_PORT`), user credentials (`USERNAME` and `PASSWORD`), and the previously created socket pool (`pool`) for network communication:

```
mqtt_client = MQTT(broker=MQTT_SERVER, port=MQTT_PORT,
username=USERNAME, password=PASSWORD, socket_pool=pool)
```

13. Our code connects the MQTT client to the MQTT broker and prints a message indicating that the system is now ready to listen for incoming LoRa messages:

```
mqtt_client.connect()
print("Listening for LoRa messages...")
```

14. We create a continuous loop to check for incoming packets from the RFM9x LoRa module; upon receiving a packet, our code decodes the message to UTF-8 format, prints the received message, flashes the LED twice for 0.5 seconds each, then publishes the message to the specified MQTT topic and prints a confirmation of the sent MQTT message:

```
while True:
 packet = rfm9x.receive()
 if packet is not None:
 message = packet.decode("utf-8")
 print("Received LoRa message:", message)
 flash_led(2, 0.5)
 mqtt_client.publish(MQTT_TOPIC, message)
 print("Sent MQTT message:", message)
```

15. To save the file, we click on **File | Save as...** from the drop-down menu. This will open the **Where to save to?** dialog.

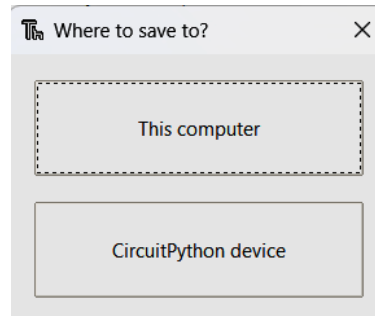


Figure 10.4 – Saving our file to our Raspberry Pi Pico W

16. In this dialog, we are given the option to choose where to store our file. To save it on our Raspberry Pi Pico W (CircuitPython device), we click on the corresponding button.
17. We then give the file the name `code.py` and click **Ok**.
18. To run our code, we click on the green run button, hit *F5* on the keyboard or click on the **Run** menu option at the top, and then select **Run current script**.
19. If it is not already running, we power up and run the LoRa sensory transmitter.
20. We verify the correct functioning of our code by checking the receipt of MQTT messages in the MQTT-Explorer app in Windows.
21. We should also observe our LED flashes twice.

By taking temperature and humidity readings from the LoRa sensory transmitter and relaying them to our MQTT server using our LoRa receiver, much like a baton pass in a relay race, we have effectively established our own weather information web service.

With our setup ready, we can now utilize the MQTT message data to operate our improved analog meter-style weather indicator. Our first step is to construct the device, followed by programming it to respond to incoming data.

## Creating a new weather indicator

In this section, we introduce an upgraded weather indicator, building on the version from *Chapter 3*. This model features an RGB LED instead of a single-color LED. We will use it to indicate humidity by setting the color to red for low humidity, green for a humidity level between 30 and 50 percent (considered a comfortable humidity level for people), and blue for a humidity above 50 percent. The device now uses the economical Raspberry Pi Pico WH as opposed to the more expensive Raspberry Pi 5. A new addition is the reset button for the Raspberry Pi Pico WH, allowing us to reset the Pico if we need to.

The design of our upgraded weather indicator includes a new feature: a split stand specifically engineered for easier printing using a **Fused Deposition Modeling (FDM)** 3D printer. By dividing the stand into two separate parts, each piece can be printed flat. When printed flat, the layer lines in each section of the stand run perpendicular to the primary stress axis encountered during use. This perpendicular arrangement of the layers effectively distributes stress across the structure, making the stand more resilient and less prone to breakage or warping under load.

We will start the construction of our new weather indicator by building the split stand.

### Building the split stand

We may use any of the stands we have built so far in the book to mount our weather indicator faceplate. For our example, we introduce a split stand that may be printed using a standard FDM 3D printer or a liquid resin 3D printer. The parts of the split stand can be seen in the following figure:



Figure 10.5 – Split stand printed with PLA and a FDM 3D printer

The parts of the split stand are as follows:

- A: Right split stand
- B: Left split stand
- C: Base
- D (not shown): 9 X M3 10 mm bolts
- E (not shown): 4 X 12 mm rubber pads (optional)

The .stl files for our split stand are located in the `Build Files` folder of this chapter's GitHub repository. All three parts shown in *Figure 10.5* may be printed together on a standard Ender-3-sized print bed (220 mm by 220 mm by 250 mm).

When using an FDM printer, it's important to slice the print file with support structures, especially since the outer ring of the split stand *floats* in space when printed flat. For the split stand, **Polylactic Acid (PLA)** material is the optimal choice due to its printing ease and reliable results.

We can see how we may position the parts of the split stand on a print bed in the following figure.

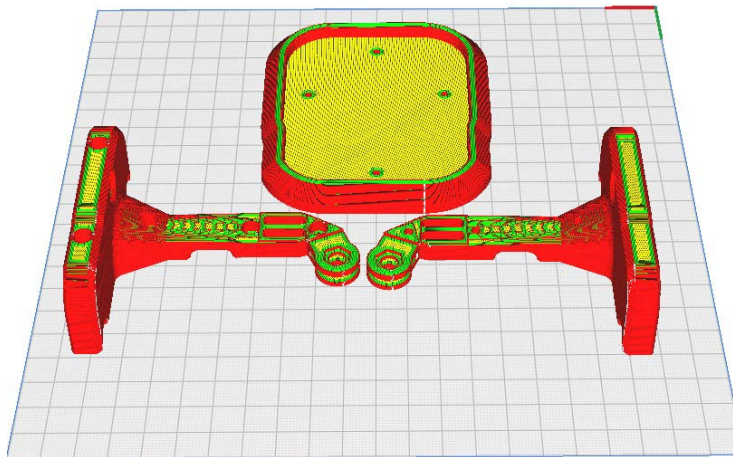


Figure 10.6 – Laying out the parts of the split stand for 3D printing on a print bed

To construct the stand, we follow the steps shown in *Figure 10.7*:



Figure 10.7 – Constructing the split stand

1. Utilizing an M3 tap, we carefully create screw threads on the right side of the split stand (A from *Figure 10.5*). This step is optional as the holes should be large enough to allow M3 bolts to screw in without tapping.
2. Using five M3 10 mm bolts, we secure the left split stand (C from *Figure 10.5*) to the right split stand (A from *Figure 10.5*).
3. Next, we attach four standard 12 mm rubber pads to the base's bottom as feet, using epoxy glue for enhanced adhesion. This step, while not mandatory, adds stability as the stand can function without the rubber pads.
4. Using an M3 tap, we create screw threads in the base. This step is optional as the holes should be large enough to allow M3 bolts to screw in without tapping.
5. Using four M3 bolts, we secure the assembled stand to the base (C from *Figure 10.5*).

The assembled split stand can be painted for aesthetic improvement. Once the split stand is finished, the next step is to assemble the faceplate of our weather indicator.

## Building the faceplate

The faceplate for our new enhanced weather indicator is very similar to the faceplate we built in *Chapter 3*. The exceptions are the use of a Raspberry Pi Pico WH over a Raspberry Pi 5 and an RGB LED, which will allow us to use various colors for our indicator.

We can see the parts that make up the faceplate in the following figure:



Figure 10.8 – Parts that make up the faceplate of our weather indicator

- A: Faceplate front.
- B: Faceplate alignment tool.
- C: Raspberry Pi Pico WH.
- D: Faceplate front button. Used to reset the Raspberry Pi Pico WH.
- E: Hook.
- F: Faceplate back.
- G: Faceplate graphic.
- H: RGB LED with resistors and jumper wires (jump ahead to the upcoming section *Adding jumper wires up our RGB LED* for construction of this component).
- I: SG90 servo motor.
- J: 8 mm LED holder.



To build the faceplate for our new weather indicator, we follow the steps in the following diagram:

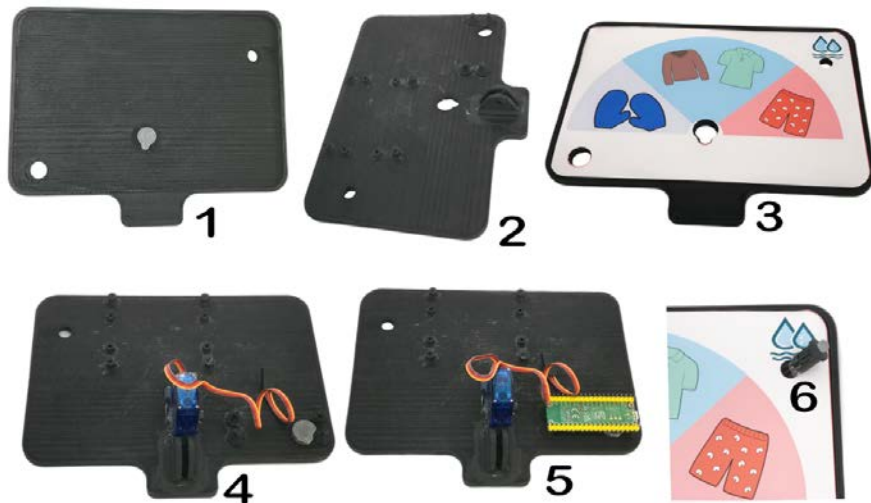


Figure 10.9 – Construction of the faceplate for our new weather indicator

1. Using the alignment tool (*B* from *Figure 10.8*) and epoxy glue, we align and glue the front of the faceplate (*A* from *Figure 10.8*) to the back of the faceplate (*F* from *Figure 10.8*).
2. We glue the hook (*E* from *Figure 10.8*) to the back of the faceplate (*F* from *Figure 10.8*) into the slot provided.
3. We print the faceplate graphic (*G* from *Figure 10.8*) on printable vinyl using a color printer. Next, we trim the graphic using a digital cutter, such as a Silhouette Cameo, or by hand and attach it to the faceplate's front, ensuring proper alignment of the holes.
4. We align the SG90 servo motor (*I* from *Figure 10.8*) with its corresponding hole and secure it in place to the back of the faceplate (*F* from *Figure 10.8*) with glue from a hot glue gun.
5. We secure the Raspberry Pi Pico WH (*C* from *Figure 10.8*) to the back of the faceplate (*F* from *Figure 10.8*) with four M3 8 mm screws such that the pins are facing up and the USB port is facing out.
6. We push the RGB LED (*H* from *Figure 10.8*) from the back of the faceplate to the front and secure it in place with the 8 mm LED holder (*J* from *Figure 10.8*).

Having constructed the faceplate and stand, we can now fasten the faceplate to the stand using an M5 20 mm bolt and nut. Once this is done, we're prepared to proceed with connecting the RGB LED and servo motor to the Raspberry Pi Pico WH.

## Configuring the RGB LED indicator

For our application, we will use an RGB LED to represent the humidity reading we get from our remote LoRa transmitter. An RGB LED allows us to represent any color as it combines three LEDs (red, green, and blue) into one housing.

Before we wire up and install our RGB LED, let's get a little understanding of what RGB LEDs are.

### *Understanding RGB LEDs*

RGB LEDs combine red, green, and blue light to produce various colors. Each color has a pin for control. There are two types: common cathode and common anode (*Figure 10.10*). Common cathode LEDs have a single ground pin shared by all colors, requiring a positive voltage to the color pins for illumination. Common anode LEDs share a positive pin, needing a ground connection on the color pins to light up.

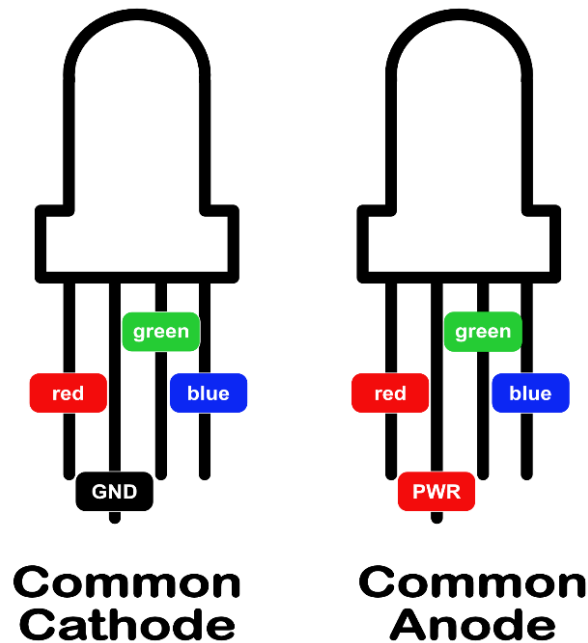


Figure 10.10 – Two different types of RGB LEDs

The type determines how we connect and control them, with common cathode LEDs needing a high signal and common anode LEDs needing a low signal for activation.

**Differentiating between common cathode and common anode RGB LEDs**

To determine whether our RGB LED is a common anode or common cathode type, we use a multimeter set to diode test mode. First, we identify the longest leg of the LED, as this is usually the common connection. We then connect the multimeter red probe to the longest leg and the black probe to one of the other legs. If the LED lights up, the longest leg is the anode, indicating a common anode LED. If it doesn't light up, we'll switch the probes. A lit LED with the black probe on the longest leg means it's a common cathode.

For our application, we will be using a common anode RGB LED, and we will program it to change colors based on humidity levels. For low humidity levels, it will be red, indicating dry conditions. It will be green to indicate a humidity level between 30 and 50 percent and blue to represent humidity levels higher than 50 percent.

***Adding jumper wires to our RGB LED***

To prepare our RGB LED for connection with the Raspberry Pi Pico WH, we'll solder female jumper wires to each of its leads. For the RGB LED's protection, we'll solder 220 Ohm resistors onto the red, green, and blue leads, positioning them between the LED's legs and the jumper wires. In our example, we use a yellow wire for power, a red wire for the red LED, a green wire for the green LED, and a blue wire for the blue LED, as shown in the following figure.

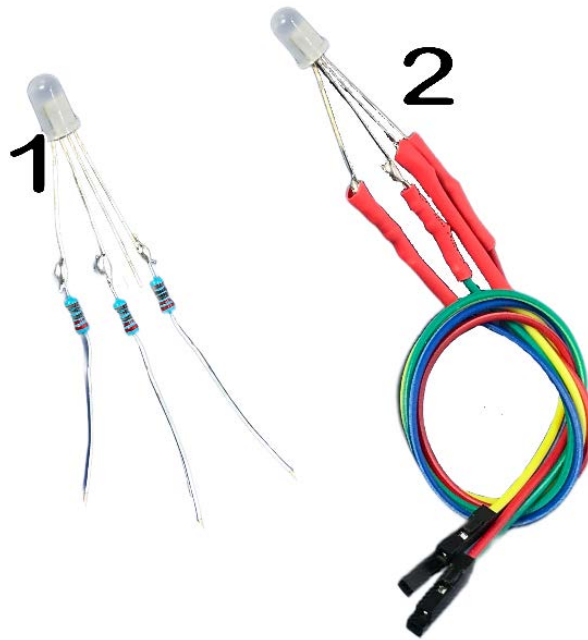


Figure 10.11 – Adding jumper wires to the RGB LED

Applying heat shrink, as shown in Step 2 of *Figure 10.11*, reinforces the solder connections by providing both physical strength and electrical insulation.

### ***Connecting the RGB LED to our Raspberry Pi Pico WH***

We will power our RGB LED with the 3V3 power port from our Raspberry Pi Pico WH. For wiring, use the following diagram:

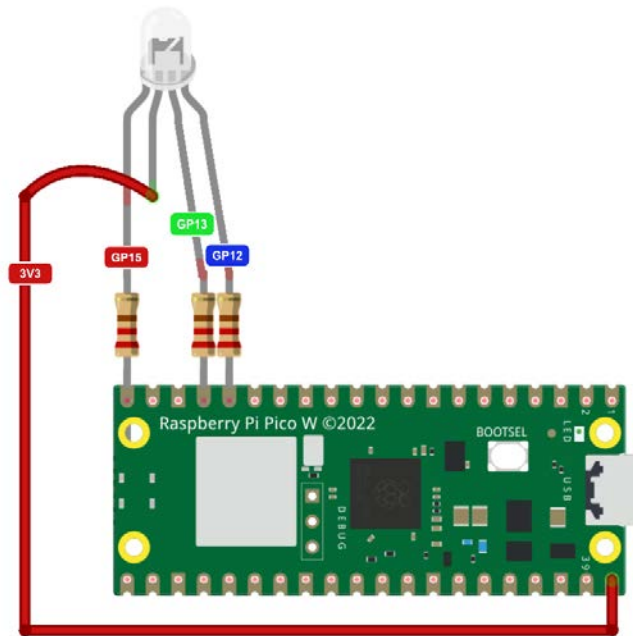


Figure 10.12 – Wiring diagram for RGB LED and Raspberry Pi Pico WH

The connections are as follows:

- 3V3 power connects to the common anode.
- Red connects to GP15 using a 220 Ohm resistor.
- Green connects to GP13 using a 220 Ohm resistor.
- Blue connects to GP12 using a 220 Ohm resistor.

With the RGB LED connected to the Raspberry Pi Pico WH, it is time to test it with code.

### ***Testing our RGB LED circuit***

To test out our RGB LED circuit, we will run code to turn on each of the three colors. We will write our code in MicroPython and use the Thonny IDE.

To test our RGB LED with code, we do the following:

1. We connect our Raspberry Pi Pico WH to a USB port and launch Thonny. We may use our Raspberry Pi or another operating system for this. If MicroPython has not been installed, we follow the steps from the *Using a Raspberry Pi Pico W with MQTT* section in *Chapter 6*, to install it on our Raspberry Pi Pico WH.
2. We then activate the MicroPython environment on our Pico by selecting it from the bottom right-hand side of the screen.
3. In a new tab in the editor section (hit *Ctrl + N*), we enter the following code:

```
from machine import Pin
import utime

red = Pin(15, Pin.OUT)
green = Pin(13, Pin.OUT)
blue = Pin(12, Pin.OUT)

def set_color(r, g, b):
 red.value(r)
 green.value(g)
 blue.value(b)

while True:
 # Red
 set_color(0, 1, 1)
 utime.sleep(1)

 # Green
 set_color(1, 0, 1)
 utime.sleep(1)

 # Blue
 set_color(1, 1, 0)
 utime.sleep(1)
```

4. To save the file, we click on **File | Save as...** from the drop-down menu. This will open the following dialog:

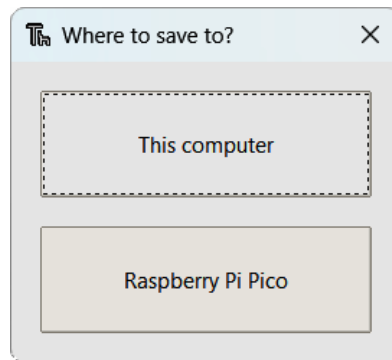


Figure 10.13 – Saving a file to our Raspberry Pi Pico

5. In this dialog, we are given the option to choose where to store our file. To save it on our Raspberry Pi Pico WH, we click on the corresponding button.
6. We then give the file the name `main.py` and click **Ok**. In the MicroPython environment, the name `main.py` is special because the system automatically executes this file upon startup or reset, making it the default script that runs when the device powers up.

Before we test out our code, let's break it down:

- I. We start by importing the `Pin` class from the `machine` module and the `utime` module.
- II. We then initialize GPIO pins GP15, GP13, and GP12 as output for red, green, and blue LEDs, respectively.
- III. We then define the `set_color(r, g, b)` function to control the RGB LED color.
- IV. We define an infinite loop:
  - i. We set the LED to red for 1 second.
  - ii. We then change the LED to green for 1 second.
  - iii. Finally, we set the LED to blue for 1 second.
7. To run our code, we click on the green run button, hit *F5* on the keyboard or click on the **Run** menu option at the top, and then select **Run current script**.
8. We should observe the RGB LED on our weather indicator cycle turning red, then green, then blue.

With our RGB LED successfully wired and tested, it is time to hook up the servo motor to our Raspberry Pi Pico WH.

## Configuring the servo motor

With our RGB LED installed and tested, it is now time to shift our focus to the servo motor on our weather indicator. Echoing our approach in *Chapter 3*, integrating a servo motor into our design offers an excellent means to bridge the analog and digital worlds; it enables us to create an analog-style meter where a needle, moved precisely by the servo, visually represents various data points.

We will start by wiring up our servo motor before turning our attention to testing it through code.

### *Wiring up our servo motor*

For our weather indicator, we will be incorporating an SG90 servo motor. The SG90 servo typically comes with three wires: the power wire (usually red), the ground wire (usually brown or black), and the signal wire (usually orange or yellow).

To wire our servo motor to our Raspberry Pi Pico WH, we will start by removing the wires from the connector housing just as we did in *Chapter 3*.

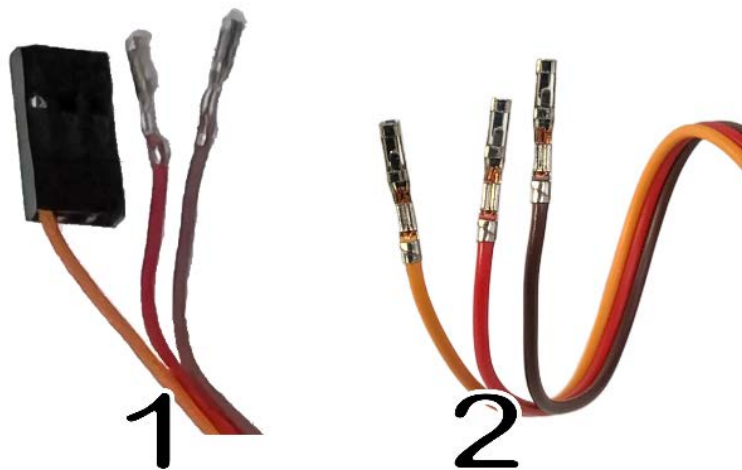


Figure 10.14 – Removing wires from the connector of the SG90 servo

Instead of reinserting the wires into the connector housing, we'll directly connect them to the pins of the Raspberry Pi Pico WH.

We will power our servo motor with the VBUS power port from our Raspberry Pi Pico WH, as VBUS provides the necessary 5V power supply directly from the USB connection, which is ideal for the typical operating voltage of most servo motors such as the SG90.

For wiring, use the following diagram:

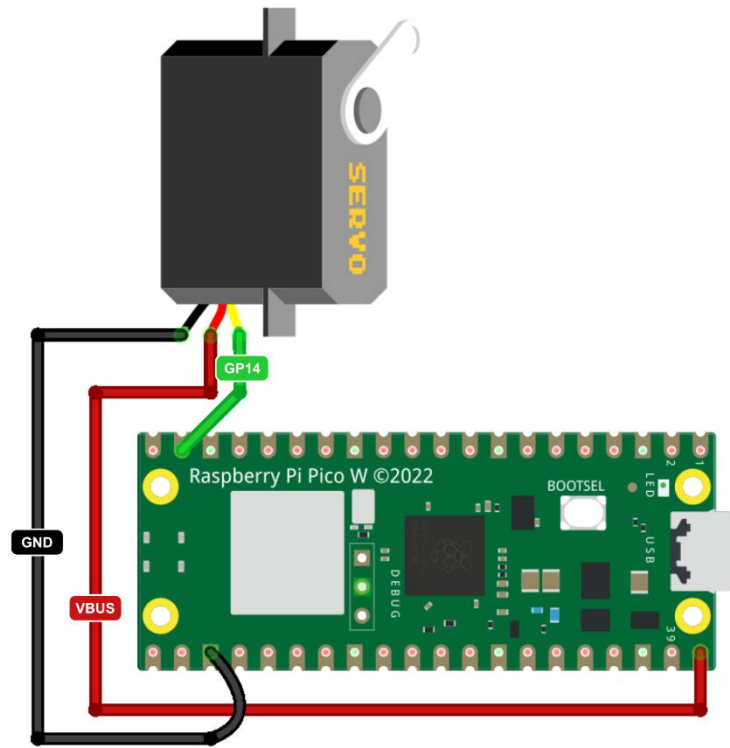


Figure 10.15 – Wiring the servo motor to the Raspberry Pi Pico WH

The connections are as follows:

- VBUS power connects to the positive wire of the servo.
- The signal wire connects to GP14 .
- GND connects to a GND port on the Raspberry Pi Pico WH.

#### Important note

For simplicity, *Figure 10.15* illustrates only the servo motor connected to the Raspberry Pi Pico WH. However, our actual circuit includes both the RGB LED and the servo motor.

With the servo motor connected to the Raspberry Pi Pico WH, it is time to test it with code.



### Testing our servo motor

To verify the functionality of our servo motor circuit, we'll execute a test code that cycles the motor through a full 180-degree range, ending at its minimum position. Once the motor reaches this minimum point, we will attach the arrow to our device. We will write our code in MicroPython and use the Thonny IDE. Our code will be organized into two files: `servo.py`, dedicated to managing the servo motor's control, and `main.py`, which will serve as the primary execution script.

To test our servo motor with code, we do the following:

1. We connect our Raspberry Pi Pico WH to a USB port on our computer and launch Thonny. We may use our Raspberry Pi or another operating system for this.
2. We then activate the MicroPython environment on our Pico by selecting it from the bottom right-hand side of the screen.
3. We will start with the code to control the servo motor. In a new tab in the editor section (hit *Ctrl + N*), we enter the following code:

```
from machine import Pin, PWM
import utime

class Servo:
 def __init__(self, pin):
 self.servo = PWM(Pin(pin))
 self.servo.freq(50)

 def set_position(self, angle):
 # Reverse the angle
 reversed_angle = 180 - angle

 # Convert the reversed angle to duty cycle
 duty = int((reversed_angle / 18) + 2)
 self.servo.duty_u16(duty * 65536 // 100)
```

In our code, we start by importing the necessary modules: `Pin` and `PWM` from `machine` for hardware control and `utime` for time-related functions.

We then define the `Servo` class to control a servo motor:

- I. The `__init__()` method initializes the servo on a specified GPIO pin as a **Pulse-Width Modulation (PWM)** output and sets the PWM frequency to 50 Hz, suitable for standard servo motors
- II. The `set_position()` method sets the servo to a specified angle:
  - i. We invert the angle to allow for reversed movement of the servo.

- ii. We then convert this reversed angle to a corresponding PWM duty cycle.
  - iii. We use the `duty_u16()` method to set the duty cycle, scaling it to a 16-bit value for PWM control.
4. To save the file, we click on **File | Save as...** from the drop-down menu.
5. We are given the option to choose where to store our file. To save it on our Raspberry Pi Pico WH, we click on the corresponding button.
6. We then give the file the name `servo.py` and click **Ok**.
7. To create our `main.py` file, we open a new editor tab in the editor section (hit `Ctrl + N`) and enter the following code:

```
from servo import Servo
import utime

servo = Servo(14)
servo.set_position(0)
utime.sleep(1)
servo.set_position(90)
utime.sleep(1)
servo.set_position(180)
utime.sleep(1)

Return servo to initial position
servo.set_position(0)
```

In our code, we have the following:

- I. We start by importing the `Servo` class from our newly created `servo` module and `utime` for timing functions.
  - II. We then create an instance of the `Servo` class that we call `servo`, using GPIO pin GP14.
  - III. Our code moves the servo to 0 degrees and waits for 1 second.
  - IV. Our code then adjusts the servo to 90 degrees and waits for another 1 second.
  - V. We set the servo to 180 degrees, followed by a 1-second pause.
  - VI. Finally, our code returns the servo to the initial position of 0 degrees.
8. To run our code, we click on the green run button, hit `F5` on the keyboard or click on the **Run** menu option at the top, and then select **Run current script**.

9. We should observe our servo motor go through a range of motions before stopping at the minimum position. One notable improvement with using the Raspberry Pi Pico WH over the Raspberry Pi 5 for connecting servo motors is the absence of servo motor jittering. The servo motor exhibits less jitter when used with the Raspberry Pi Pico WH compared to the Raspberry Pi due to the Pico's direct, hardware-level PWM control, which ensures more precise and stable signal delivery to the servo.
10. It is at this point where we place the arrow on the servo motor. We place the arrow such that it is pointing toward the gloves on our graphic. The arrow should fit snugly; however, a bit of sanding may be required to get it to fit.



Figure 10.16 – Placing the needle at the zero position

Having successfully wired and tested the RGB LED and servo motor with our Raspberry Pi Pico WH, we're now ready to develop the code that will control our weather indicator, utilizing data extracted from MQTT messages.

## Programming our weather indicator

The development of our weather indicator is streamlined, as the components' direct placement on the faceplate allows for immediate code implementation and testing, eliminating the need for breadboarding. This also reduces the chance of error as we do not need to configure the circuit a second time.

The software architecture of our application is shown in the following figure.

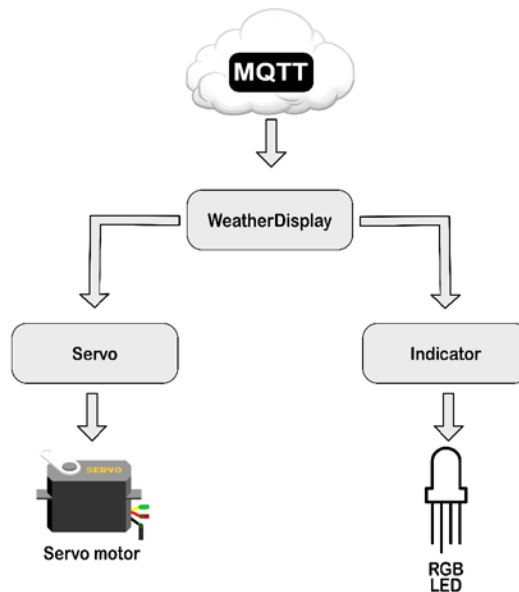


Figure 10.17 – Weather display code layout

Our code is organized into three files, each containing a distinct class. The `main.py` file houses the `WeatherDisplay` class, which retrieves and parses MQTT messages from our CloudAMQP instance. Based on the temperature value in these messages, the `WeatherDisplay` class adjusts the servo position (and thus the weather indicator's needle) using the `Servo` class. It also interprets humidity data to control the RGB LED's color via the `Indicator` class.

To create our weather indicator code, we do the following:

1. We connect our Raspberry Pi Pico WH from our weather indicator to a USB port on our computer and launch Thonny. We may use our Raspberry Pi or another operating system for this.
2. We then activate the MicroPython environment on our Pico WH by selecting it from the bottom right-hand side of the screen.
3. We will start with the code to control the RGB LED. This code will introduce the `Indicator` class, designed to manage the RGB LED's color transition from red to green to blue, reflecting changes in humidity levels. In a new tab in the editor section (hit `Ctrl + N`), we enter the following code:

```
from machine import Pin
import utime
```

```
class Indicator:
 def __init__(self):
 self.red = Pin(15, Pin.OUT)
 self.green = Pin(13, Pin.OUT)
 self.blue = Pin(12, Pin.OUT)

 def set_color(self, r, g, b):
 self.red.value(r)
 self.green.value(g)
 self.blue.value(b)

 def set_indicator(self, value):
 # Turn off all LEDs initially
 self.set_color(1, 1, 1)

 if value <= 30:
 # Turn on red LED
 self.set_color(0, 1, 1)
 elif 30 < value <= 50:
 # Turn on green LED
 self.set_color(1, 0, 1)
 else:
 # Turn on blue LED
 self.set_color(1, 1, 0)

 def flash_led(self, times):
 for _ in range(times):
 self.set_color(0, 0, 0)
 utime.sleep(0.5)
 self.set_color(1, 1, 1)
 utime.sleep(0.5)
```

In our code, we have the following:

- I. We start by importing the `Pin` class from the `machine` module and `utime` for handling time-based functions.
- II. We define the `Indicator` class and initialize it with three attributes representing red, green, and blue LEDs, set as output pins on GP15, GP13, and GP12, respectively.
- III. The `set_color()` method controls the state of each color in the RGB LED based on the input parameters. In this method, for a common anode RGB LED, a binary 0 activates a color (turns it on), while a 1 deactivates it (turns it off). The method is used to selectively turn on either the red, green, or blue component of the LED.

- IV. We then create the `set_indicator()` method:
  - i. Our code turns off all LEDs initially.
  - ii. Our code turns on the red LED for values  $\leq 30$ , the green LED for values between 30 and 50, and the blue LED for values  $> 50$ .
- V. In our `flash_led()` method, we flash all three LEDs (white light) for a specified number of times (`times`), with each flash lasting 0.5 seconds with a 0.5-second interval where LEDs are off.
4. To save the file, we click on **File | Save as...** from the drop-down menu.
5. We are given the option to choose where to store our file. To save it on our Raspberry Pi Pico WH, we click on the corresponding button.
6. We then give the file the name `indicator.py` and click **Ok**.
7. The second class in our code is a new version of the servo class. To create this class, we open a new tab in the editor section (hit *Ctrl* + *N*), and enter the following code:

```
from machine import Pin, PWM
import utime

class Servo:
 def __init__(self, pin):
 self.servo = PWM(Pin(pin))
 self.servo.freq(50)

 def set_position(self, value):
 int_value = int(value)
 angle = 180 - (int_value / 40) * 180
 angle = max(0, min(angle, 180))

 # Convert the angle to duty cycle
 duty = int((angle / 18) + 2)
 self.servo.duty_u16(duty * 65536 // 100)
```

In our code, we start by importing the `Pin` and `PWM` modules from `machine`, and `utime` for time-related functions.

We then define the `Servo` class to control servo motors:

- I. In the constructor (`__init__()`), we initialize a PWM object on the specified pin.
- II. We set the PWM frequency to 50 Hz, a standard frequency suitable for servo motors.

- III. We then create a `set_position()` method that does the following:
  - i. Converts the input value into an integer.
  - ii. Maps the input range (0-40) to a servo angle range (0-180 degrees) in reverse to align with the installation of our servo motor.
  - iii. Ensures the calculated angle is within the valid range (0-180 degrees).
  - iv. Converts the angle to a duty cycle suitable for the servo.
  - v. Sets the PWM duty cycle to position the servo.
8. To save the file, we click on **File | Save as...** from the drop-down menu.
9. We are given the option to choose where to store our file. To save it on our Raspberry Pi Pico WH, we click on the corresponding button.
10. We then give the file the name `servo.py` and click **Ok**.
11. The `WeatherDisplay` class, located in the main execution file, is responsible for subscribing to the `WeatherInfo` MQTT topic and handling the messages received. For its operation, the `micropython-umqtt.simple` library is necessary. To install this library, refer to the *Improving on our IoT button with the Raspberry Pi Pico W* section in *Chapter 7*.
12. To create the `WeatherDisplay` class, we begin by entering the import statements in a new tab in the editor section (hit `Ctrl + N`):

```
import network
import utime
from umqtt.simple import MQTTClient
from servo import Servo
from indicator import Indicator
```

In our code, we do the following:

- I. We import the `network` module for Wi-Fi connectivity functions.
  - II. We import the `utime` module for time-related functions.
  - III. Our code imports `MQTTClient` from `umqtt.simple` to handle MQTT communications.
  - IV. We import our `Servo` class from our newly created local module for servo motor control.
  - V. We then import our `Indicator` class from our newly created local module for RGB LED indicator control.
13. We then define the class name and initialization method where we define our Wi-Fi and MQTT parameters:

```
class WeatherDisplay:
 def __init__(self):
```

```
WiFi Information
self.ssid = "MySSID"
self.wifi_password = "ssid-password"

MQTT Information
self.mqtt_server = "driver.cloudmqtt.com"
self.mqtt_port = 18756
self.username = "mqtt-username"
self.mqtt_password = "mqtt-password"
self.device_id = "WeatherDisplay"
self.mqtt_topic = "WeatherInfo"

self.indicator = Indicator()
self.servo = Servo(14)
```

14. The `connect_wifi()` method establishes a connection between the Raspberry Pi Pico WH and the local Wi-Fi network. It continuously attempts to connect until successful, and upon establishing a connection, it indicates success by flashing the RGB LED in white four times:

```
def connect_wifi(self):
 wlan = network.WLAN(network.STA_IF)
 wlan.active(True)
 if not wlan.isconnected():
 print('Connecting to WiFi...')
 wlan.connect(self.ssid, self.wifi_password)
 while not wlan.isconnected():
 pass
 print('WiFi connected, IP:', wlan.ifconfig()[0])
 self.indicator.flash_led(4)
```

15. We use the `connect_mqtt()` method to connect to our CloudAMQP instance and subscribe to the `WeatherInfo` topic (as set by the `mqtt_topic` variable) and set our callback function to `on_message_received()`:

```
def connect_mqtt(self):
 self.client = MQTTClient(self.device_id,
 self.mqtt_server,
 self.mqtt_port,
 self.username,
 self.mqtt_password)
 self.client.set_callback(
 self.on_message_received)
 self.client.connect()
 self.client.subscribe(self.mqtt_topic)
```



16. The `on_message_received()` function processes received MQTT messages by parsing temperature and humidity data and then updating the servo position and RGB LED indicator accordingly:

```
def on_message_received(self, topic, msg):
 print("Received:", topic, msg.decode())
 temperature, humidity =
 self.parse_message(msg)
 if temperature is not None:
 self.servo.set_position(temperature)
 if humidity is not None:
 self.indicator.set_indicator(humidity)
```

17. The `parse_message()` function extracts and returns temperature and humidity values from a decoded MQTT message, handling any exceptions and returning None values if parsing fails:

```
def parse_message(self, msg):
 try:
 parts = msg.decode().split(',')
 temperature_str = parts[0].split('Temp:')[1].
split('C')[0].strip()
 humidity_str = parts[1].split('Humidity:')[1].
split('%')[0].strip()

 temperature = float(temperature_str)
 humidity = float(humidity_str)

 return temperature, humidity
 except Exception as e:
 print("Error parsing message:", str(e))
 return None, None
```

18. The `run` method, the final method defined in the `WeatherDisplay` class, initiates the Wi-Fi connection, connects to the MQTT client, and continuously checks for MQTT messages, handling any errors encountered during message reception:

```
def run(self):
 self.connect_wifi()
 self.connect_mqtt()
 while True:
 try:
 self.client.check_msg()
 except Exception as e:
 print("Error checking MQTT message:",
 str(e))
 utime.sleep(5)
```

19. After writing the `WeatherDisplay` class code, we instantiate it and call its `run()` method, initiating the weather display functionality, which includes establishing Wi-Fi and MQTT connections and processing incoming messages:

```
Create and run the weather display
weather_display = WeatherDisplay()
weather_display.run()
```

20. To save our file, we click on **File | Save as...** from the drop-down menu.
21. We are given the option to choose where to store our file. To save it on our Raspberry Pi Pico WH, we click on the corresponding button.
22. We then give the file the name `main.py` and click **Ok**.
23. To run our code, we click on the green run button, hit `F5` on the keyboard, or click on the **Run** menu option at the top, and then select **Run current script**.
24. We should observe the RGB LED flashing four times after connection to our Wi-Fi network.
25. We should observe that the arrow moves to a position indicating the temperature coming from the temperature sensor on our LoRa sensory transmitter.
26. We should observe that the color of the RGB LED is either red, green, or blue, indicating the humidity level measured by our LoRa sensory transmitter.



Figure 10.18 – The three devices that make up our IoT weather service, starting from the left, are the LoRa sensory transmitter (*Chapter 9*), the LoRa receiver/ internet gateway (*Chapter 9*), and the analog-metered weather indicator

With the completion of our project, we've successfully built an IoT weather service and display console, combining real-time data collection with an interactive display. This system utilizes MQTT for data transmission, servo motors for representing data physically, and an RGB LED for visual alerts.

## Exploring other IoT communication protocols

LoRaWAN, operating on the same sub-gigahertz radio frequencies as LoRa, is an advanced protocol for wireless communication that enables long-range transmissions with low power consumption. This shared frequency band is a key feature that allows both LoRaWAN and LoRa to transmit data over several kilometers, which is especially beneficial in areas where traditional connectivity is scarce.

In the context of IoT applications such as weather monitoring, LoRaWAN offers significant advantages. For example, a network of sensors collecting environmental data such as temperature and humidity could transmit this information over long distances to a central gateway connected to the internet, leveraging LoRa's long-range capabilities. This gateway then relays the data to a cloud server for processing and analysis.

However, for our weather indicator project, using the full LoRaWAN setup would be considered overkill. Our project utilizes a simpler setup involving two Raspberry Pi Pico microcontrollers – one equipped with LoRa for data transmission (Raspberry Pi Pico) and the other with Wi-Fi capability (Raspberry Pi Pico W). This setup effectively demonstrates the capabilities of LoRa for short-range IoT communication, capitalizing on the same long-range, low-power characteristics of the LoRa frequencies, but without the complexity and infrastructure requirements of a complete LoRaWAN network.

Cellular services can also be used for IoT communication, offering extensive coverage and higher data transfer speeds. Cellular IoT devices, such as those using 4G LTE or 5G networks, can transmit larger amounts of data over long distances. This makes cellular services suitable for more data-intensive applications or those requiring real-time, high-speed communication.

While cellular IoT offers broader coverage and higher data throughput, it often comes with higher power consumption and complexity compared to LoRa-based solutions. Thus, for small-scale or low-power projects such as our weather indicator, the simplicity and efficiency of using LoRa modules with Raspberry Pi Pico microcontrollers provide a more suitable and cost-effective solution.

**Sigfox** is a global network operator providing dedicated cellular connectivity for IoT and **Machine-to-Machine (M2M)** communications. It uses a unique technology for wireless transmission that allows for long-range, low-power communication. Sigfox operates in the sub-gigahertz frequency band and is designed for small data payload transmissions, typically up to 12 bytes per message. This limited data capacity makes it ideal for devices that need to send small, infrequent bursts of data, such as sensors in smart meters, agricultural monitors, and asset tracking systems. Sigfox's network architecture is distinguished by its simplicity, efficiency, and cost-effectiveness, making it a popular choice for applications where low-cost and low-power operations are critical.

LoRaWAN, cellular networks, and Sigfox are among the leading communication protocols used in IoT applications. Each has its unique features and use cases. Here's a comparative table outlining their advantages and disadvantages:

| Protocol                        | Advantages                                                                                                                                                                                                                                                                      | Disadvantages                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LoRaWAN                         | <ul style="list-style-type: none"> <li>• Long range (up to 15 km)</li> <li>• Low power consumption</li> <li>• Good penetration in urban environments</li> <li>• Open protocol with a growing ecosystem</li> </ul>                                                               | <ul style="list-style-type: none"> <li>• Lower data rates</li> <li>• Limited bandwidth and duty cycle restrictions</li> <li>• Requires gateway for internet connectivity</li> </ul>                                                                                                                                                                                                                  |
| Cellular (4G/5G)                | <ul style="list-style-type: none"> <li>• High data throughput</li> <li>• Wide coverage and reliability</li> <li>• Supports real-time communication</li> <li>• Well-established infrastructure</li> </ul>                                                                        | <ul style="list-style-type: none"> <li>• Higher power consumption</li> <li>• Subscription costs</li> <li>• Potentially overpowered for simple tasks</li> </ul>                                                                                                                                                                                                                                       |
| Sigfox                          | <ul style="list-style-type: none"> <li>• Ultra-low power consumption</li> <li>• Long-range capabilities</li> <li>• Simple to deploy and maintain</li> <li>• Ideal for small, infrequent data transmissions</li> </ul>                                                           | <ul style="list-style-type: none"> <li>• Very limited data payload (12 bytes/message)</li> <li>• Limited to 140 messages per day</li> <li>• Proprietary technology with limited flexibility</li> </ul>                                                                                                                                                                                               |
| LoRa with Wi-Fi microcontroller | <ul style="list-style-type: none"> <li>• Long range (up to 15 km for LoRa)</li> <li>• Low power consumption for LoRa transmissions</li> <li>• Cost-effective solution for small-scale projects</li> <li>• Flexible and easy integration with existing Wi-Fi networks</li> </ul> | <ul style="list-style-type: none"> <li>• Limited to the range of the Wi-Fi network for internet connectivity</li> <li>• Tied to a single microcontroller and may not be used for mobile applications</li> <li>• Requires additional hardware (Wi-Fi microcontroller) for internet connectivity</li> <li>• May require a more complex setup and coding compared to using a single protocol</li> </ul> |

Table 10.1 – Comparing IoT communication protocols

Each protocol serves different IoT scenarios:

- **LoRaWAN:** Ideal for applications requiring long-range communication and low power usage, such as agricultural sensors or smart city applications.
- **Cellular networks:** Suitable for high-data, real-time applications such as video surveillance, automotive applications, or any scenario requiring broad geographic coverage.
- **Sigfox:** Excels in scenarios where devices only need to send small amounts of data over long distances, such as utility metering or asset tracking.
- **LoRa with Wi-Fi microcontroller:** Best for small-scale, localized IoT projects requiring the long-range capability of LoRa and the internet connectivity provided by Wi-Fi, such as home automation, local environmental monitoring, or DIY IoT projects.

For small-scale or low-power projects such as our weather indicator, using LoRa modules with Raspberry Pi Pico microcontrollers presents a more suitable and cost-effective solution than the options mentioned. LoRa with microcontrollers balances range, data handling, and power efficiency without the need for more complex infrastructure.

## Summary

In this chapter, we integrated LoRa-based data transmission with internet connectivity using MQTT. We modified our CircuitPython-based LoRa receiver's code, allowing it to send sensory data to the CloudAMQP server, transforming it into an internet gateway. The weather indicator was upgraded, replacing the Raspberry Pi 5 and single-color LED with a Raspberry Pi Pico WH and an RGB LED, which displayed temperature and humidity data via MQTT messages.

Additionally, we constructed a new split stand for the indicator, designed for easy assembly and improved stability. We concluded the chapter by examining IoT communication protocols such as LoRaWAN, cellular networks, and Sigfox, assessing their suitability for different IoT applications.

We engaged in a hands-on exploration of integrating LoRa with internet connectivity via MQTT, a practical demonstration of how disparate technologies can work together in IoT applications. By upgrading the weather indicator, we not only learned about iterative design and troubleshooting but also gained insights into the decision-making process involved in IoT projects, such as choosing the Raspberry Pi Pico WH over the Raspberry Pi 5 for controlling servo motors.

This chapter marks the end of the book's third part, focused on IoT communications with an emphasis on LoRa. In the next chapter, we transition to robotics, where we'll start on our journey to build an internet-connected robot car.

# Part 4:

## Building an IoT Robot Car

In this part, we use all the knowledge we've gained from the previous chapters to build an IoT robot car that is controlled through the internet. This project will be the most advanced and demonstrate the ultimate power of IoT technology, controlling anything from anywhere.

This part has the following chapters:

- *Chapter 11, Introducing ROS*
- *Chapter 12, Creating an IoT Joystick*
- *Chapter 13, Introducing Advanced Robotic Eyes for Security (A.R.E.S.)*
- *Chapter 14, Adding Computer Vision to A.R.E.S.*



# Introducing ROS

In this chapter, we introduce the **Robot Operating System (ROS)**, a powerful tool for developing robotics applications. We will explore the significance of ROS in the field of robotics and detail the process of setting up ROS on a Raspberry Pi 4. This involves replacing the standard Raspberry Pi OS with Ubuntu, due to Ubuntu's compatibility and optimization for specific versions of ROS.

We will start our hands-on exercises setting up and running **TurtleSim**, a user-friendly ROS simulator. We will do this to gain knowledge of basic ROS concepts and operations. Starting with simple keyboard controls, we will learn how to command and maneuver a virtual robot within the simulator environment. We will follow this up by progressing to controlling the TurtleSim simulator with **Message Queuing Telemetry Transport (MQTT)** messages as we start to bridge the gap between simulation and real-world application.

Building on the skills developed through working with TurtleSim will prepare us for constructing our advanced IoT robot named A.R.E.S. (short for Advanced Robot Eyes for Security). A.R.E.S. is the final and most sophisticated project of the book and will comprise the remaining chapters.

We will cover the following topics in this chapter:

- Exploring ROS
- Installing Ubuntu and ROS onto our Raspberry Pi
- Running and controlling a simulated robot

Let's begin!

## Technical requirements

The following are the requirements for completing this chapter:

- Intermediate knowledge of Python programming
- Basic knowledge of the Linux command line



- A CloudAMQP account for the MQTT server instance
- Late-model Raspberry Pi 4 or any computer capable of installing Ubuntu (Ubuntu installed on a Mac mini was used in this chapter)
- microSD card and microSD – USB adapter

The code for this chapter may be found here:

<https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter11>

## Exploring ROS

In this section, we will explore the basics of ROS. This section is by no means an in-depth replacement for the excellent documentation that may be found on the ROS website at [www.ros.org](http://www.ros.org).

ROS is an **Open Source Software (OSS)** development kit for robotics applications, providing a standard platform that bridges the gap between research and production. Designed to speed up the robotics development process, ROS simplifies the creation of robotic systems. It is 100% open source and commercially friendly.

Our goal here is to establish a common knowledge base, equipping us with the essential concepts and tools of ROS needed for this chapter's project and, eventually, our A.R.E.S. robot. We will start by outlining the main project of this chapter: the control of the ROS TurtleSim virtual robot using MQTT messaging.

## Reviewing our TurtleSim controller ROS application

TurtleSim is a lightweight robot simulator provided by ROS and is primarily used as an educational tool for learning ROS concepts. It offers a simple interface for teaching the basics of ROS, allowing users to experiment with commands and observe the behavior of a simulated robot in a safe and controlled environment.

For the main project in this chapter, we will control a TurtleSim instance with MQTT messages as we command the robot to draw a circle or stop moving based on the message we transmit to the `move` topic of our MQTT instance.

In *Figure 11.1*, we see our application illustrated. Messages created using the MQTT-Explorer in Windows are sent to the `circle` node we create in ROS. Based on the message, either `draw_circle` or `stop`, we send a `cmd_vel` topic ROS message to a TurtleSim instance. This internal messaging uses publishers and subscribers and is like the way MQTT communication is performed:

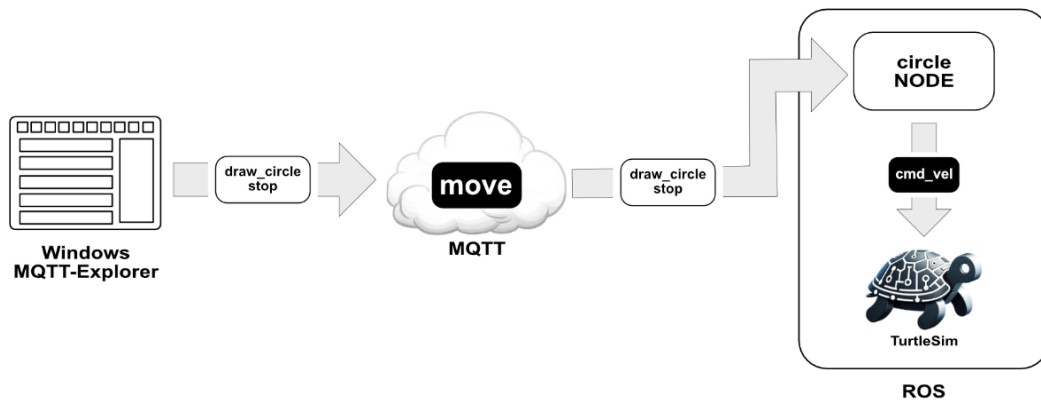


Figure 11.1 – Using MQTT messages to control a simulated robot

For the application in this chapter, we will be sending messages directly using the MQTT-Explorer app. In the next chapter, we will build an IoT joystick with a Raspberry Pi Pico W and use it to control the simulated robot.

Now that we've outlined our chapter's goals, let's step back to gain a basic understanding of ROS concepts. This overview will provide us with a fundamental insight into its architecture and role in robotics.

## Understanding ROS node communication

ROS supports a wide range of applications and platforms, including Linux, Windows, macOS, and embedded systems, making it highly versatile. Its modular framework is based on the concepts of nodes, topics, services, and actions. Nodes in ROS are individual processes that perform specific computations, and topics serve as communication channels where nodes exchange messages using a publisher-subscriber mechanism.

For our application, we will be using a topic and publisher-subscriber model to send `vel_msg` messages of the `cmd_vel` topic from our custom `circle` node to an instance of a TurtleSim virtual robot. Although a publisher may have many subscribers, we will only be using one instance of TurtleSim to subscribe to the publisher we will build in our custom node. We see this illustrated in the following diagram:

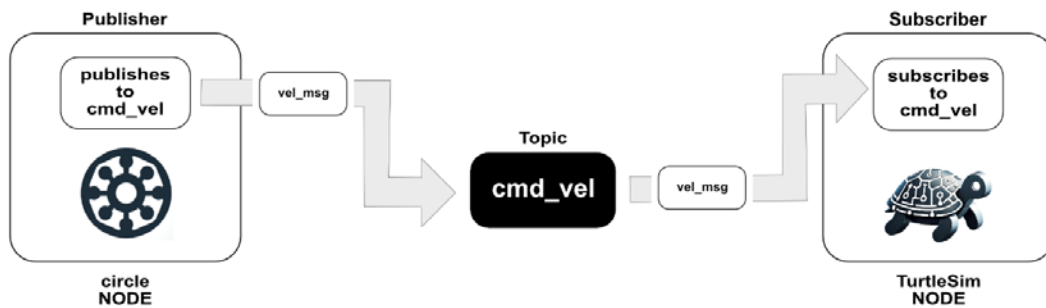


Figure 11.2 – A view of the publisher-subscriber model inside ROS

Services in ROS offer a way for nodes to perform request-response interactions, useful for tasks that require immediate feedback. Actions provide a means for executing long-running tasks that require continuous feedback and the possibility of cancellation. We will not be using services or actions for our application.

Now that we have a little understanding of ROS node communication, let's look at the project structure in ROS.

## Investigating ROS project structure and organization

The structure of a ROS project is organized for the efficient development and management of robotics applications. At its core, a ROS project is built around the concept of packages and workspaces, which are essential for organizing the various components of a robotics project. The following list summarizes the key concepts for a ROS project:

- **Packages and workspaces:** A ROS workspace is a directory (or a folder) where ROS packages are developed and compiled. Each package, typically a directory within the workspace, represents a specific functionality or component of the robot, such as sensors, actuators, algorithms, or even a collection of related nodes. Packages can contain ROS nodes, libraries, datasets, configuration files, or anything that constitutes an independent and reusable module.
- **ROS nodes and communication:** Within these packages, the primary executable units are the nodes. Each node is designed to perform a specific task, such as controlling a motor, processing sensor data, or performing computations. Nodes communicate with each other over topics, services, or actions. Topics allow for asynchronous, publish-subscribe communication, ideal for streaming data such as sensor readings or control commands. Services provide synchronous, request-response interaction, useful for tasks that need immediate feedback. Actions are suited for long-running tasks that require continuous feedback and the possibility of cancellation. As mentioned, we will be creating a custom node to communicate with the TurtleSim simulator using topics and the publish-subscribe method.

- **Build system and package management:** ROS uses a build system (such as `catkin` in ROS 1 or `colcon` in ROS 2) to compile and manage packages. The build system handles dependencies and integrates packages into the ROS ecosystem. We will be using `colcon` to compile our project.
- **Parameter server and launch files:** The parameter server in ROS is a shared, multi-variable dictionary accessible by nodes at runtime, allowing them to configure parameters dynamically. Launch files are another crucial aspect of ROS projects. They are XML/YAML files that specify which nodes to run, set parameters, and configure the node network for a specific use case, such as launching a robot in a simulation environment. We will not be using launch files for our application and will instead launch both the TurtleSim simulator and our custom node using the `ros2 run` command.

We will construct our workspaces, build our packages, and execute our program directly from the command line. Before we do this, let's explore how ROS distributions align with Ubuntu versions.

## Aligning ROS distributions with Ubuntu LTS versions

Each ROS distribution is paired with a designated Ubuntu **Long-Term Support (LTS)** version, a strategy that guarantees stability and compatibility as it gives ROS developers a consistent code base to work with.

The ROS release policy is clear: one Ubuntu LTS version per ROS release, with no new Ubuntu version support after the release. To take advantage of this pairing of Ubuntu with ROS, we will install Ubuntu 22.04 and the Humble Hawksbill version of ROS 2 onto our Raspberry Pi 4 (for those of us who wish to run Ubuntu on a computer other than a Raspberry Pi, we may skip the next section).

We will start by flashing Ubuntu onto a microSD card using the Raspberry Pi Imager.

## Installing Ubuntu and ROS onto our Raspberry Pi

In this section, we will walk through the steps of installing Ubuntu 22.04 and ROS Humble Hawksbill on our Raspberry Pi 4. This involves choosing the correct Ubuntu image, flashing it onto a microSD card using the Raspberry Pi Imager, and setting up the Raspberry Pi to boot with Ubuntu.

### Why are we using a Raspberry Pi 4 and not a Raspberry Pi 5?

At the time of writing, the Raspberry Pi 5 does not support Ubuntu 22.04 and thus the latest version of ROS.

We will start by launching the Raspberry Pi Imager on a computer of our choice.

## Installing Ubuntu on our Raspberry Pi 4

To install Ubuntu onto our Raspberry Pi 4, we will use the Raspberry Pi Imager to burn the operating system onto a microSD card, which we will install on our Pi. The Raspberry Pi Imager is a tool created by the Raspberry Pi Foundation and is used to simplify the process of imaging microSD cards with the Raspberry Pi operating system and other compatible systems.

The Raspberry Pi Imager is available for Windows, macOS, and Linux and may even be installed on the Raspberry Pi itself.

### Using the Raspberry Pi Imager on a Raspberry Pi

We may use the Raspberry Pi Imager on our Raspberry Pi to burn an image onto a microSD card. The process involves using an SD card reader connected to the Raspberry Pi's USB port. We may install the imager from a Terminal in the Raspberry Pi OS with the `sudo apt install rpi-imager` command.

To burn Ubuntu onto a microSD card for use with a Raspberry Pi, we do the following:

1. Using an internet browser, we navigate to the following website to download the Raspberry Pi Imager:

<https://www.raspberrypi.com/software/>

2. To begin the imaging process, we insert a microSD card into a USB microSD adapter and then connect the adapter to a USB port on our computer:



Figure 11.3 – A USB microSD adapter with microSD card inserted

3. Next, we install the Raspberry Pi Imager software on our computer. Once installed, launching the imager will bring up its main screen, ready for us to select an operating system image and the target microSD card for the installation:

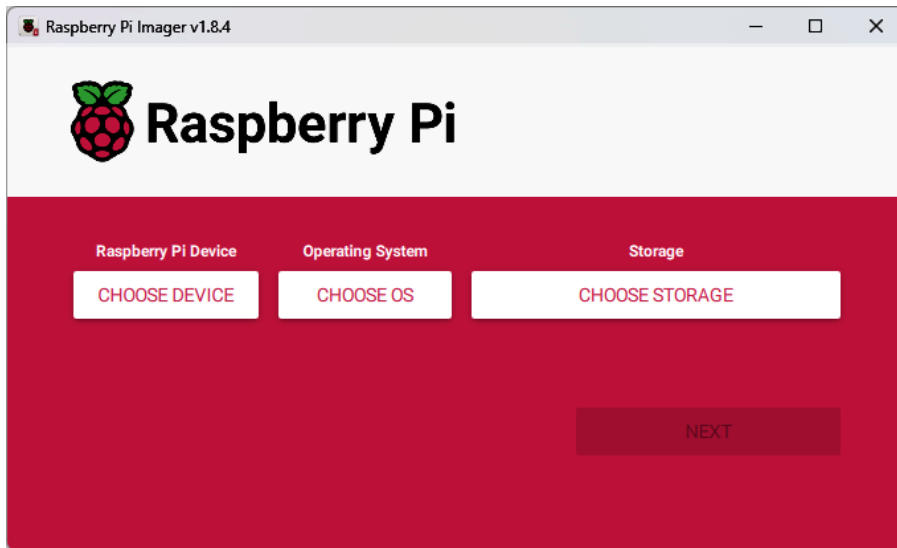


Figure 11.4 – Raspberry Pi Imager

4. Under **Raspberry Pi Device**, we select **NO FILTERING**.
5. For **Operating System**, we select **Other General Purpose OS**, then **Ubuntu**, then **Ubuntu Desktop 22.04.3 LTS (64-BIT)**.
6. We then click on the **CHOOSE STORAGE** button and select the microSD card we inserted into our computer.
7. Our selections should look like the following:

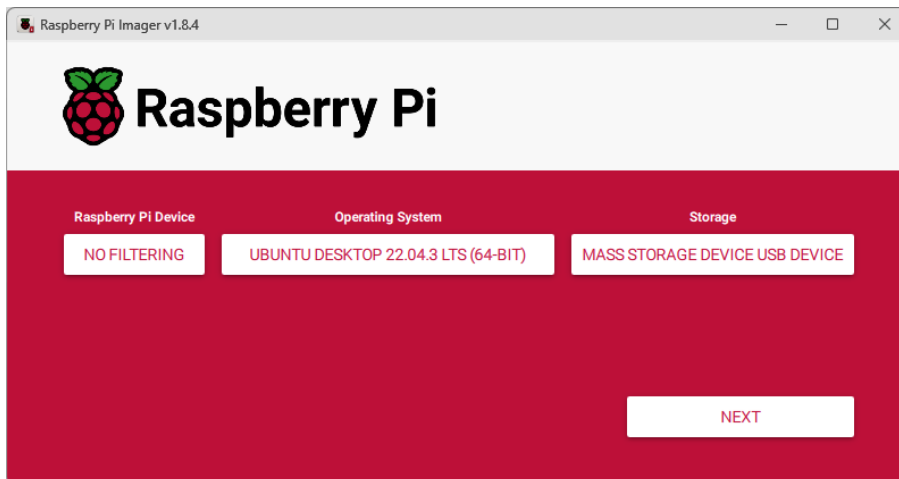


Figure 11.5 – Raspberry Pi Imager after selections

**Selecting the correct version of Ubuntu**

At the time of writing, the latest version of ROS 2 supports Ubuntu 22.04. Therefore, in our example, we will use Ubuntu 22.04 LTS, even though the most recent version of Ubuntu is 23.10. For future iterations of ROS 2, it's important to select the version of Ubuntu that corresponds with the ROS 2 release in use.

8. To start the burning process, we click on the **NEXT** button and click on the **Yes** button on the **Warning** dialog to continue.
9. Once the writing process is complete, the Raspberry Pi Imager will perform a verification to ensure the image has been correctly written to the microSD card. Upon successful verification, we should see a **Write Successful** screen, indicating that the microSD card is now ready to be used in our Raspberry Pi with the newly installed operating system.
10. With the image burned onto our microSD card, we then proceed to install the card onto our Raspberry Pi and follow the steps to complete the installation of Ubuntu onto the Raspberry Pi.

With Ubuntu now running on our Raspberry Pi, the next step is to install the appropriate version of ROS. Despite its name, ROS isn't an operating system. Instead, it functions as a middleware or software framework, providing tools and libraries for building and managing robotic applications.

## Adding ROS to our Ubuntu installation

Recapping our *Exploring ROS* section, we emphasized the importance of matching each ROS distribution with a specific Ubuntu LTS version to ensure stability. Having installed Ubuntu 22.04 on our Raspberry Pi, we're now ready to install ROS.

As of this writing, there are two ROS versions compatible with Ubuntu 22.04: Humble Hawksbill and Iron Irwini. Humble Hawksbill is an LTS version, meaning it's designed for stability and extended support, ideal for longer-term projects and those seeking a stable development environment. Iron Irwini, on the other hand, is a non-LTS version, typically featuring more cutting-edge changes but with a shorter support life cycle.

We will use Humble Hawksbill for our application as we desire stability over new features for what we are doing.

**Important note**

The ROS installation instructions provided here are current as of the time of writing. However, it's recommended to consult the official web page for the most up-to-date guidance, as there may be updates or changes.

To install Humble Hawksbill on our Raspberry Pi, we do the following:

1. To view the latest versions of ROS, we navigate to the following website:  
<https://www.ros.org/blog/getting-started/>
2. To view the current versions of ROS, we scroll down to the **Installation** section of the page where we see the following:

|                                                                                   |                                                                                   |                                                                                    |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
|  |  |  |
| <b>Get ROS Noetic Ninjemys on Ubuntu Linux</b>                                    | <b>Get Humble Hawksbill on Ubuntu Linux 22.04, Windows 10</b>                     | <b>Get Iron Irwini on Ubuntu Linux 22.04, Windows 10</b>                           |
| (Recommended for Latest ROS 1 LTS)                                                | (Recommended for ROS 2 LTS)                                                       | (Recommended for Latest ROS 2)                                                     |
| <a href="#">Install</a>                                                           | <a href="#">Install</a>                                                           | <a href="#">Install</a>                                                            |

Figure 11.6 – Current versions of ROS

3. To proceed, we click on the **Install** link under the **Humble Hawksbill** section.  
This will bring us to the Humble Hawksbill installation page.
4. To install Humble Hawksbill for Ubuntu, we click on the **Debian packages** link (highlighted with a red box in the following figure) under **Ubuntu Linux – Jammy Jellyfish (22.04)**:

### Binary packages

Binaries are only created for the Tier 1 operating systems listed in [REP-2000](#). Given the nature of Rolling, this list may be updated at any time. If you are not running any of the following operating systems you may need to build from source or use a [container solution](#) to run ROS 2 on your platform.

We provide ROS 2 binary packages for the following platforms:

- Ubuntu Linux - Jammy Jellyfish (22.04)
  - **Debian packages (recommended)**
  - "fat" archive
- RHEL 8
  - [RPM packages](#) (recommended)
  - "fat" archive
- [Windows \(VS 2019\)](#)

Figure 11.7 – Binary packages for ROS



- To set the locale in Ubuntu, we simply copy the commands from the **Set locale** section of the web page. This can be done quickly by clicking the **Copy** icon on the top right of the code box, which appears when we hover over it. We then paste and execute these commands into the Ubuntu Terminal to complete the locale setting process:

### Set locale

Make sure you have a locale which supports `UTF-8`. If you are in a minimal environment (such as a docker container), the locale may be something minimal like `POSIX`. We test with the following settings. However, it should be fine if you're using a different UTF-8 supported locale.

```
locale # check for UTF-8

sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8

locale # verify settings
```

Copy 

Figure 11.8 – Setting the locale in Ubuntu

- To configure our system to access and authenticate the ROS software repository, we copy, paste, and execute the code from each section under the **Setup Sources** section:

### Setup Sources

You will need to add the ROS 2 apt repository to your system.

First ensure that the [Ubuntu Universe repository](#) is enabled.

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

Now add the ROS 2 GPG key with apt.

```
sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros
```

Then add the repository to your sources list.

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http
```

Figure 11.9 – Configuring our Ubuntu installation for access to the ROS software repository

7. To update our system's packages, we type the following command into the Terminal:

```
sudo apt update
```

8. We then upgrade the packages on our system with the following command:

```
sudo apt upgrade
```

9. We will be using the desktop version of ROS. We install it with the following command:

```
sudo apt install ros-humble-desktop
```

10. As we will be creating our own nodes, we will need the ROS development tools. We install these tools with the following command:

```
sudo apt install ros-dev-tools
```

With ROS and the ROS development tools installed, we are now ready to explore. We will start off with a simple publish-subscribe example.

## Testing our ROS installation

We may test our new ROS installation with different components, regardless of the programming languages used to write these components. A common approach to this, and the approach we will take, is to use a simple publisher-subscriber model where the publisher is written in C++ and the subscriber in Python.

In Figure 11.10, the nodes shown are part of the ROS desktop installation. The figure highlights a **Publisher** node, written in C++, which sends a Hello World: xx message followed by a sequential number to the chatter topic. The **Subscriber** node is written in Python and subscribes to the chatter topic:

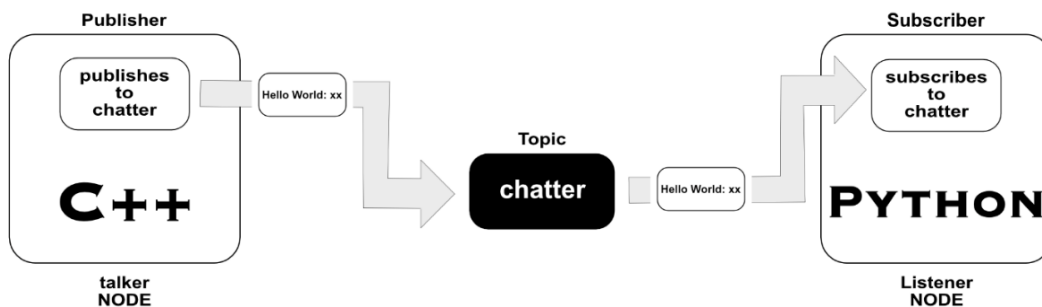


Figure 11.10 – ROS publisher-subscriber using C++ publisher and Python subscriber

To run the example, we do the following:

1. In Ubuntu, we open a new Terminal and type in the following command:

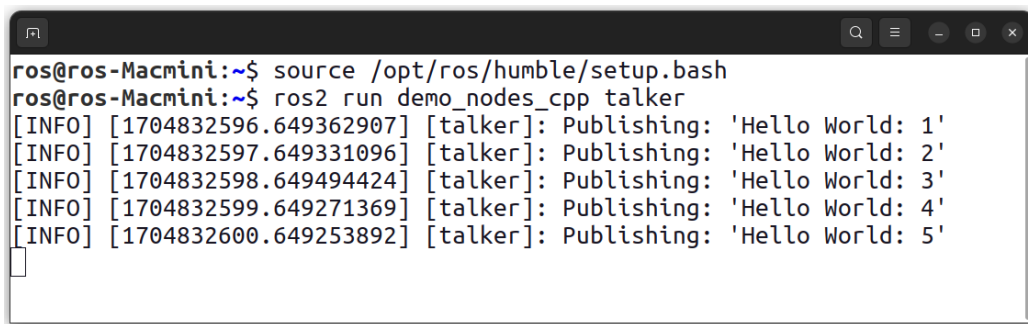
```
source /opt/ros/humble/setup.bash
```

We use this command to initialize the ROS environment for the current Terminal session, thereby enabling ROS commands and package usage.

2. To launch the publisher, we type the following:

```
ros2 run demo_nodes_cpp talker
```

With this command, we launch the ROS 2 node named `talker` from the `demo_nodes_cpp` package. We should observe messages printed to the console:

A terminal window titled 'ros@ros-Macmini:~\$' showing the execution of ROS 2 commands. The user enters 'source /opt/ros/humble/setup.bash' and then 'ros2 run demo\_nodes\_cpp talker'. The output shows five lines of log messages from the 'talker' node, each indicating a published message: 'Hello World: 1' through 'Hello World: 5'. Each message is preceded by an '[INFO]' log level, a timestamp in brackets, and the node name '[talker]'. The terminal window has standard Ubuntu window controls (minimize, maximize, close) and a search icon in the top right corner.

```
ros@ros-Macmini:~$ source /opt/ros/humble/setup.bash
ros@ros-Macmini:~$ ros2 run demo_nodes_cpp talker
[INFO] [1704832596.649362907] [talker]: Publishing: 'Hello World: 1'
[INFO] [1704832597.649331096] [talker]: Publishing: 'Hello World: 2'
[INFO] [1704832598.649494424] [talker]: Publishing: 'Hello World: 3'
[INFO] [1704832599.649271369] [talker]: Publishing: 'Hello World: 4'
[INFO] [1704832600.649253892] [talker]: Publishing: 'Hello World: 5'
```

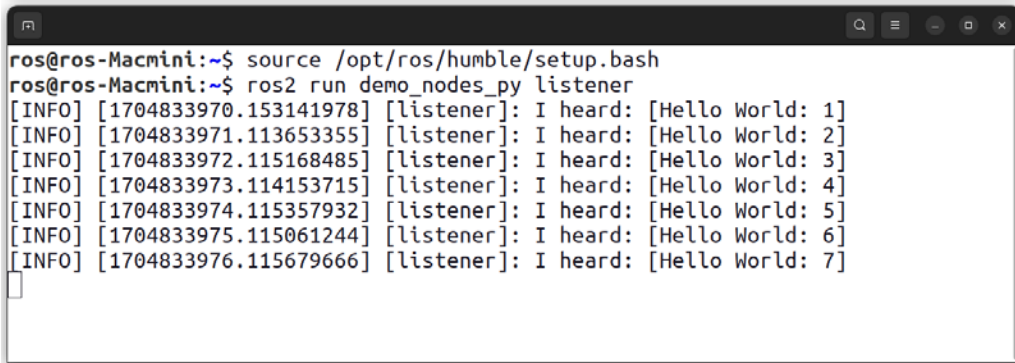
Figure 11.11 – Output from running the talker node

We can see from the Terminal that an output of `Hello World:` followed by an incrementing number is published by the `talker` node. Although we can't tell from the output, the topic our node is publishing to is the `chatter` topic.

3. To receive the messages, we open a new Terminal in Ubuntu and launch the Python subscriber node with the following commands:

```
source /opt/ros/humble/setup.bash
ros2 run demo_nodes_py listener
```

With every new Terminal we open, we must source the ROS installation to enable ROS commands. We should observe an acknowledgment of the published messages in our new Terminal:

A terminal window on a Mac mini showing the execution of ROS nodes. The prompt is 'ros@ros-Macmini:~\$'. The user enters 'source /opt/ros/humble/setup.bash'. The prompt changes to 'ros@ros-Macmini:~\$'. The user enters 'ros2 run demo\_nodes\_py listener'. The terminal displays seven lines of log output, each starting with '[INFO]' followed by a timestamp and the message '[listener]: I heard: [Hello World: X]' where X ranges from 1 to 7.

```
ros@ros-Macmini:~$ source /opt/ros/humble/setup.bash
ros@ros-Macmini:~$ ros2 run demo_nodes_py listener
[INFO] [1704833970.153141978] [listener]: I heard: [Hello World: 1]
[INFO] [1704833971.113653355] [listener]: I heard: [Hello World: 2]
[INFO] [1704833972.115168485] [listener]: I heard: [Hello World: 3]
[INFO] [1704833973.114153715] [listener]: I heard: [Hello World: 4]
[INFO] [1704833974.115357932] [listener]: I heard: [Hello World: 5]
[INFO] [1704833975.115061244] [listener]: I heard: [Hello World: 6]
[INFO] [1704833976.115679666] [listener]: I heard: [Hello World: 7]
```

Figure 11.12 – Receiving messages from the talker node

With this, we have successfully installed and tested our ROS installation. In the next section, we will run a simulated robot and control it through MQTT messages.

## Running and controlling a simulated robot

In this section, we explore the use of TurtleSim, a lightweight and user-friendly tool within ROS for simulating a robot's movements and behaviors. TurtleSim serves as an excellent educational resource, especially for beginners in ROS, to understand basic concepts such as node interactions, message passing, and simple robotic commands.

By using TurtleSim, we will learn how to create a ROS node to control a virtual robot. We will control the TurtleSim robot with MQTT messages. In the upcoming chapters, we will use what we learn here to convert our robot into a physical robot we will call A.R.E.S.

We'll begin by launching a TurtleSim node within Ubuntu and then control it using a separate ROS node.

### Launching and testing TurtleSim

As mentioned, TurtleSim is designed to help new users familiarize themselves with ROS functionalities such as nodes, topics, and services through a simple interface. By running TurtleSim, we can simulate a robot's movement and behavior in a controlled environment.

To launch TurtleSim, we do the following:

1. In Ubuntu, we open a new Terminal and type in the following commands:

```
source /opt/ros/humble/setup.bash
ros2 run turtlesim turtlesim_node
```

The first command initializes the ROS environment, while the second runs the `turtlesim_node` from the `turtlesim` package. Once these commands are executed, a TurtleSim window should appear, displaying a graphic of a simulated turtle positioned in the center of the screen:



Figure 11.13 – TurtleSim robot

2. To control the turtle, we use another node. To do this, we open another Terminal in Ubuntu and execute the following commands:

```
source /opt/ros/humble/setup.bash
ros2 run turtlesim turtle_teleop_key
```

The first command initializes the ROS environment, while the second executes the `turtle_teleop_key` node from the `turtlesim` package. This node allows us to control the TurtleSim robot with our keyboard. Our Terminal should look like the following:

```
ros@ros-Macmini:~$ source /opt/ros/humble/setup.bash
ros@ros-Macmini:~$ ros2 run turtlesim turtle_teleop_key
Reading from keyboard

Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F' to cancel a rotation.
'Q' to quit.
█
```

Figure 11.14 – Running the `turtle_teleop_key` node

3. We may move our TurtleSim robot simply by holding down the arrow keys on our keyboard. We may also rotate our robot using any of the keys listed in the Terminal:



Figure 11.15 – Results of moving the TurtleSim robot

4. To visualize node and topic connections in ROS, we can use `rqt_graph`, a graphical tool that displays how nodes interact. It's especially helpful for debugging and deciphering the network within a ROS system. To launch `rqt_graph`, we enter the following commands in a new Ubuntu Terminal:

```
source /opt/ros/humble/setup.bash
rqt_graph
```

5. The first command initializes the ROS environment, while the second loads the `rqt_graph` tool. We should see the following window:

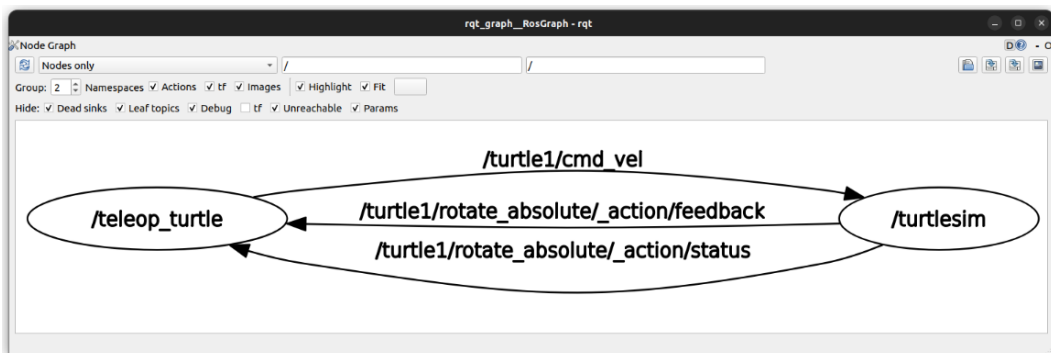


Figure 11.16 – Using the `rqt_graph` tool in ROS

6. From our graph, the `cmd_vel` topic is a key communication channel in ROS that connects the `teleop_turtle` node, acting as a controller, to the `turtlesim` node, which simulates the robot. In our ROS graph, we refer to the TurtleSim instance as `turtle1`, and it's this name we use to identify the specific turtle being controlled when interacting with the `cmd_vel` topic. The graph also shows `rotate_absolute` actions. For our basic robot control application, we are only interested in `cmd_vel` topic communications.

#### Why do we launch `turtle_teleop_key` but view `teleop_turtle`?

The difference between the `teleop_turtle` node name in the `rqt_graph` tool and the `ros2 run turtlesim turtle_teleop_key` command stems from the framework's naming conventions and structure. The `turtle_teleop_key` command refers to the executable file in ROS that, when run, initializes a ROS node. This node is internally named `teleop_turtle` within the ROS environment for communication and identification. This approach allows flexibility in ROS, where a single executable can launch different nodes, and node names can be dynamically changed for specific needs or configurations. The node name is essential for network communication, such as publishing to topics, while the executable name is just for starting the node.

Our exercise demonstrates how we may control a robot using a node – in this case, the `teleop_turtle` node, which allows us to control the TurtleSim robot with our keyboard. In the next section, we will create our own node, which will allow us to control the robot via MQTT messaging.

## Creating an ROS workspace and package

In ROS 2, the structure and creation of packages are critical for organizing, distributing, and compiling our code. The creation of a package in ROS 2 involves using `ament` as the build system and `colcon` as the build tool. We have the option to create packages in either C++ or Python.

The contents of a ROS 2 package vary depending on whether it's a CMake or Python package. Typically, a CMake package includes a `CMakeLists.txt` file and a `package.xml` file, along with directories for source code and package headers. A Python package, on the other hand, will include `package.xml` and `setup.py` files and a directory with the same name as the package containing an `__init__.py` file, among others. For our application, we will use Python.

We can have multiple packages in a ROS 2 workspace, each in its own folder, and these packages can be of different build types. It's recommended to keep packages within the `src` folder of our workspace to maintain organization.

In the following figure, we can see the structure of the ROS workspace we will build, with the dot (.) representing the root directory of the workspace. The workspace is simply a directory in our filesystem. Under the `src` folder, we have our single package called `mqtt_robot`:

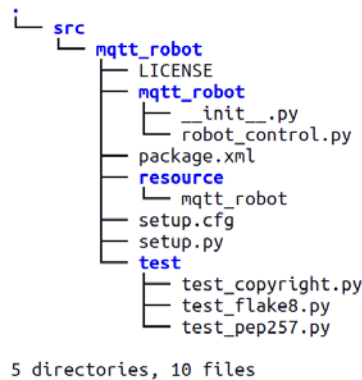


Figure 11.17 – ROS workspace structure

To build our ROS application, we will begin by creating a workspace, a Linux directory that includes a `src` subfolder for organizing our application's ROS packages.

To do so, we do the following:

1. We open an Ubuntu Terminal and execute the following command in our home directory:

```
mkdir -p chl1_ws/src
```

#### Important note

Creating our workspace in the home directory simplifies future navigation, as we can use the `~` character as a shortcut in our commands.

With this command, we create a workspace. The `-p` flag in the `mkdir` command ensures that any necessary parent directories are created as part of the new directory path.

2. With the folder created, we navigate to the `src` folder with the following command:

```
cd chl1_ws/src
```

3. To initialize our ROS environment, we execute the following command:

```
source /opt/ros/humble/setup.bash
```

4. We then create our package by executing the following command:

```
ros2 pkg create --build-type ament_python --license Apache-2.0
--node-name draw_circle mqtt_robot --license Apache-2.0
```

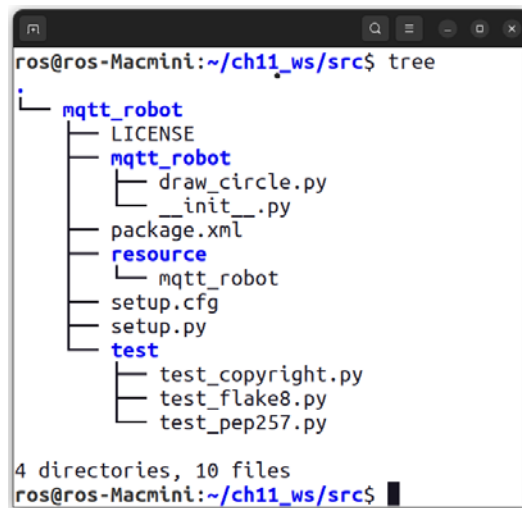
In the command we use to build our package, we specify the Apache 2.0 License.



### What is the Apache 2.0 License?

This license is an open source license that allows for commercial and non-commercial use and modification, with the requirement to disclose major changes in distributed versions and the explicit grant of patent rights to users. In our case, this disclosure requirement applies only if we were to modify the existing code of the build tools or other Apache 2.0 licensed software, not to the new code we write ourselves after the package creation.

5. With this line, we created a new package named `mqtt_robot` with the Python build type and generated a node named `draw_circle`. To view the new file structure, we execute the `tree` command:



```

ros@ros-Macmini:~/ch11_ws/src$ tree
.
├── mqtt_robot
│ ├── LICENSE
│ ├── mqtt_robot
│ │ ├── draw_circle.py
│ │ └── __init__.py
│ ├── package.xml
│ ├── resource
│ │ └── mqtt_robot
│ ├── setup.cfg
│ ├── setup.py
│ └── test
│ ├── test_copyright.py
│ ├── test_flake8.py
│ └── test_pep257.py
4 directories, 10 files
ros@ros-Macmini:~/ch11_ws/src$

```

Figure 11.18 – Workspace file structure under the `src` directory

With our `ch11_ws` workspace and `mqtt_robot` package created; we are now ready to start modifying the generated code for our purposes. We will start with the `draw_circle.py` script.

## Modifying the generated Python code

After the code generation of our package, we are presented with a folder called `mqtt_robot` under the `src` folder. This folder represents our package. Inside is another folder with the same name, `mqtt_robot`. It is in this second `mqtt_robot` folder where we find the main Python code for our application.

To create the logic for our application, we will modify the `draw_circle.py` script. To do so, we do the following:

1. For our application, we require the `paho-mqtt` library for MQTT communications. In an Ubuntu Terminal, we type the following command to install the library:

```
pip install paho-mqtt
```

2. We then navigate to the folder that contains the `draw_circle.py` script with the following command:

```
cd ~/ch11_ws/src/mqtt_robot/mqtt_robot
```

3. To open the `draw_circle.py` file in a text editor, we execute the following command:

```
gedit draw_circle.py
```

4. We start by deleting all the code. We then start our new code with imports:

```
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
import paho.mqtt.client as mqtt
```

In our code, we have the following:

- I. `import rclpy`: Imports the ROS 2 client library for Python, allowing the script to interact with ROS 2 functionalities and create ROS 2 nodes.
  - II. `from rclpy.node import Node`: Imports the `Node` class from the `rclpy` module, enabling the script to define custom nodes for ROS 2 applications.
  - III. `from geometry_msgs.msg import Twist`: Imports the `Twist` message type from the `geometry_msgs` package; we use this for sending commands to move the TurtleSim robot.
  - IV. `import paho.mqtt.client as mqtt`: Imports the Paho MQTT client library we will use for MQTT protocol communication.
5. We define an `MQTTMessage` class, which includes an initialization method and a method to set a flag. The `__init__()` method initializes the `should_draw_circle` attribute as `False`, and the `set_flag()` method updates this attribute to `True` when our class receives the `draw_circle` message. We set `should_draw_circle` to `False` when a stop message is received:

```
class MQTTMessage:
 def __init__(self):
 self.should_draw_circle = False
```

```
def set_flag(self, message):
 if message == 'draw_circle':
 self.should_draw_circle = True
 elif message == 'stop':
 self.should_draw_circle = False
```

6. We derive the `CircleMover` class from ROS 2's `Node` class. This class is designed for controlling the movement of a simulated turtle in `TurtleSim` based on MQTT messages. The class initializes with an MQTT message handler, sets up a publisher for the `turtle1/cmd_vel` topic to control movement, and configures an MQTT client for connecting to a broker and handling incoming messages:

```
class CircleMover(Node):
 def __init__(self, mqtt_message):
 super().__init__('circle_mover')
 self.mqtt_message = mqtt_message
 self.publisher = self.create_publisher(
 Twist, 'turtle1/cmd_vel', 10)
 timer = 0.1 # seconds
 self.timer = self.create_timer(timer,
 self.timer_callback)
 self.vel_msg = Twist()
 # Initialize MQTT Client and set up callbacks
 self.mqtt_client = mqtt.Client()
 self.mqtt_client.on_connect = self.on_connect
 self.mqtt_client.on_message = self.on_message
 self.mqtt_client.username_pw_set("username",
 "password")

 self.mqtt_client.connect("
 driver.cloudmqtt.com",
 port, 60)
 self.mqtt_client.loop_start()

 def on_connect(self, client, userdata, flags, rc):
 client.subscribe("move")

 def on_message(self, client, userdata, msg):
 self.mqtt_message.set_flag(
 msg.payload.decode())
```

7. The `timer_callback()` function inside the class determines the turtle's movement based on the `should_draw_circle` flag set by MQTT messages, enabling dynamic control of the turtle through MQTT:

```
def timer_callback(self):
 if self.mqtt_message.should_draw_circle:
 self.vel_msg.linear.x = 1.0
 self.vel_msg.angular.z = 1.0
 else:
 self.vel_msg.linear.x = 0.0
 self.vel_msg.angular.z = 0.0
 self.publisher.publish(self.vel_msg)
```

8. To complete our code, we define a `main()` function for our ROS 2 Python script, which initializes the ROS client library, creates an instance of the `MQTTMessage` class, and then an instance of the `CircleMover` class using the MQTT message handler. It runs the ROS node with the `rclpy.spin()` method, keeping the node active and responding to callbacks. Upon termination, it destroys the node and shuts down the ROS client library. We use the `main()` function as the entry point for our script, executing it when the script is run directly:

```
def main(args=None):
 rclpy.init(args=args)
 mqtt_message = MQTTMessage()
 circle_mover = CircleMover(mqtt_message)
 rclpy.spin(circle_mover)
 circle_mover.destroy_node()
 rclpy.shutdown()

if __name__ == '__main__':
 main()
```

9. With our code completed, we save the file using the same name, `draw_circle.py`.

Our next step is to update the `package.xml` file to include Python library dependencies for our code.

## Updating package.xml

The `package.xml` file in ROS is a descriptor that includes essential information about a ROS package, such as its name, version, maintainers, licenses, and dependencies. Crucial for the build process, it informs the `colcon` build tool about the dependencies needed for compiling the package. It is created and maintained by the developer.

For our purposes, we will modify `package.xml` to inform it of the Python libraries our code needs to compile.

To do so, we do the following:

1. We open an Ubuntu Terminal and navigate to the folder containing the `package.xml` file:

```
cd ~/ch11_ws/src/mqtt_robot
```

2. To open `package.xml` in a text editor, we execute the following command:

```
gedit package.xml
```

3. We update the XML by adding the following lines before the last tag (`</package>`):

```
<exec_depend>rclpy</exec_depend>
<exec_depend>paho-mqtt</exec_depend>
```

4. These lines indicate to the compiler that `rclpy` (Python client library for ROS) and `paho-mqtt` (used for MQTT communications) are execution dependencies for the ROS package, meaning these packages are required for running the ROS nodes contained in the package. We save our changes and close the editor.

With updates to `package.xml`, we are now ready to compile our code and run our new node.

## Compiling and running our code

To compile our code, we use the `colcon` ROS tool, a command-line tool used for compiling ROS packages, handling dependencies, and orchestrating builds across multiple packages in the workspace. For our application, we require it to compile only one package.

To compile and execute our new code, we do the following:

1. In Ubuntu, we open a new Terminal and source our ROS 2 environment:

```
source /opt/ros/humble/setup.bash
```

2. We then navigate to the root of our workspace:

```
cd ~/ch11_ws
```

3. To compile our code, we execute the following command:

```
colcon build
```

**Why do we create a package from the `src` folder but compile from the root?**

It's worth noting that package creation and compilation in a ROS workspace occur at different levels. While we create individual packages within the `src` folder of a workspace, compilation is done from the workspace's root folder. This distinction is key: creating packages is a localized action within `src`, but compiling with `colcon` at the workspace root ensures all packages within `src` are built together.

4. Upon completion, a message confirming a successful build will appear in the Terminal.
5. With our code compiled, it is time to source our new ROS environment. We do this with the following command:

```
source ~/ch11_ws/install/setup.bash
```

6. This is like how we source the ROS environment. To run our node, we execute the following command:

```
ros2 run mqtt_robot draw_circle
```

7. Here, we are running our new node, `draw_circle`, from the package we created, `mqtt_robot`. Upon execution, our Terminal will enter a wait state, ready to respond to incoming events or actions.
8. If not already running, we launch a TurtleSim instance.

We may notice that nothing is happening. The turtle in TurtleSim is not moving, and the Terminal where we launched our node is in a waiting state. To make the turtle move, we need to send an MQTT message to our new node.

Let's do that now.

## Controlling our robot with an MQTT message

We see a high-level overview of our application in *Figure 11.1*, where an MQTT message using MQTT-Explorer directs our ROS-simulated robot. The `draw_circle` message prompts the turtle to draw a circle, while `stop` halts its movement. This forms the foundational basis for our project, which we'll expand with additional features in the upcoming chapters.

To control the TurtleSim robot from an MQTT message, we do the following:

1. Using the MQTT-Explorer app in Windows, we publish a `draw_circle` message to the `move` topic.

2. Upon sending, we should observe that our TurtleSim robot starts to move in a circle:

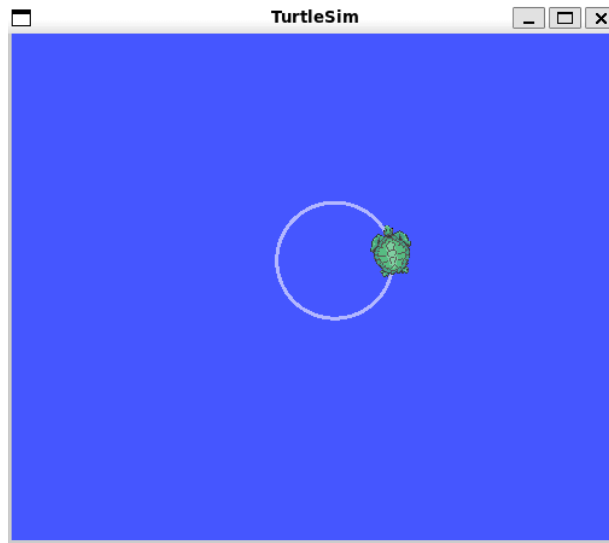


Figure 11.19 – TurtleSim robot moving from MQTT message

3. To stop the robot, we send a `stop` message under the `move` topic using the MQTT-Explorer app.
4. We should observe that the TurtleSim robot stops moving.

Our demonstration of controlling a virtual robot using MQTT messages lays the groundwork for applying these learned concepts to our upcoming real robot project, A.R.E.S.

## Summary

In this chapter, we began our exploration into ROS. We started by setting up ROS on a Raspberry Pi 4, going for Ubuntu over the standard Raspberry Pi OS for better compatibility with ROS.

Our hands-on journey started with TurtleSim, a user-friendly ROS simulator. We learned basic ROS operations and concepts, starting with keyboard controls to maneuver a virtual robot. We then advanced to using MQTT messages for control, bridging the gap between simulation and real-world application.

This experience with TurtleSim is foundational for our main project, A.R.E.S., an advanced IoT robot that will be developed in the upcoming chapters.

In the next chapter, we will return to building IoT devices as we build an MQTT joystick to control our TurtleSim robot.

# Creating an IoT Joystick

In this chapter, we will create an IoT joystick for remotely controlling a ROS TurtleSim robot. We will build upon our experience from *Chapter 7*'s IoT button project as well as *Chapter 11*'s introduction to the TurtleSim virtual robot.

Utilizing the Raspberry Pi Pico WH's Wi-Fi capabilities, we will demonstrate the practical application of IoT in robotics. The chapter outlines the design and construction of a USB-powered joystick, integrating components such as a PS2 joystick module, an LED, and an arcade button. We will use this IoT joystick to control our A.R.E.S. robot in the coming chapters.

We will use a Raspberry Pi Pico WH for this chapter, although a Raspberry Pi Pico W would work just as well.

We will cover the following topics in this chapter:

- Understanding our IoT joystick application
- Wiring up our circuit
- Developing the code for our IoT joystick
- Creating a custom ROS node for our application
- Constructing the IoT joystick case

Let's begin!

## Technical requirements

In this chapter, you will require the following:

- Intermediate knowledge of Python programming
- Basic knowledge of the Linux command line.
- A CloudAMQP account for the MQTT server instance



- Ubuntu-ROS installed computer from the previous chapter
- Access to a 3D printer or 3D printing service
- Raspberry Pi Pico WH
- Raspberry Pi Pico WH GPIO expander
- PS2 joystick module
- LED with 220 Ohm resistor
- 24 mm arcade-style pushbutton
- 4 x M3 10 mm bolts
- 4 x M2 5 mm screws
- 8 x M2.5 5 mm bolts
- 4 x M2.5 10 mm standoffs
- 1 x 8 mm LED holder
- Wires to connect Raspberry Pi Pico WH to GPIO expander
- Build files for custom cases may be found in our GitHub repository

The code for this chapter may be found here:

<https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter12>

## Understanding our IoT joystick application

In *Chapter 7*, we developed a device to remotely arm an IoT alarm system. Building on this experience, our application in this chapter involves creating an IoT joystick using a Raspberry Pi Pico WH. This joystick will remotely control a ROS TurtleSim robot and thus display a practical application of IoT in robotics.

We will use the Raspberry Pi Pico WH for our design given Wi-Fi capability and pre-soldered pin headers. In our application, we'll integrate a PS2 joystick module, an LED, and an arcade-style pushbutton with a Raspberry Pi Pico WH. The Raspberry Pi Pico WH will be programmed to transmit MQTT messages that reflect the joystick's movements, the status of the joystick button, and the arcade button's state. Additionally, the LED will serve a dual-purpose role, indicating both the Wi-Fi connectivity status and the MQTT connection status sequentially:

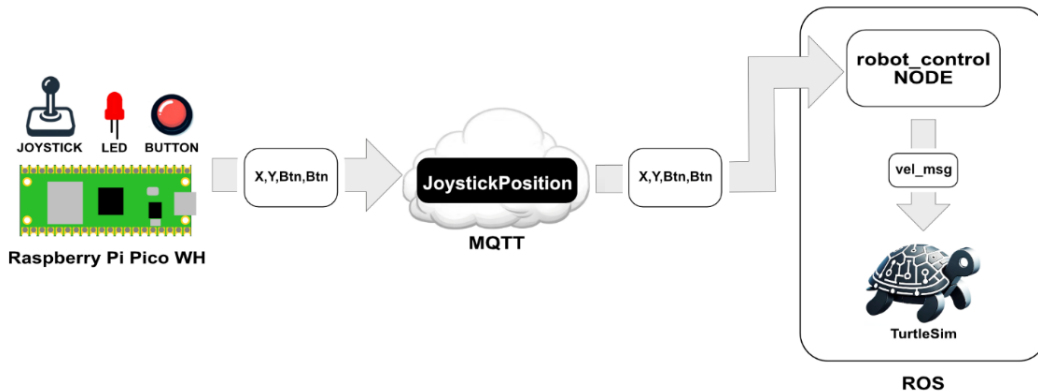


Figure 12.1 – Overview of the IoT joystick application

As we see in *Figure 12.1*, our Raspberry Pi Pico WH transmits the state of the  $x$  axis,  $y$  axis, and two buttons as an MQTT message under the `JoystickPosition` topic to our `robot_control` custom ROS node. Our ROS node then in turn transmits `vel_msg` messages to an instance of a TurtleSim robot.

We will start by wiring up the circuit for our IoT joystick.

## Wiring up our circuit

To simplify wiring, we're using a GPIO expander along with our Raspberry Pi Pico WH. The case that we will build later allows for easy transfer of the test circuit wiring, highlighting the practicality of using a GPIO expander. We can see the components that make up the IoT joystick in the following figure:

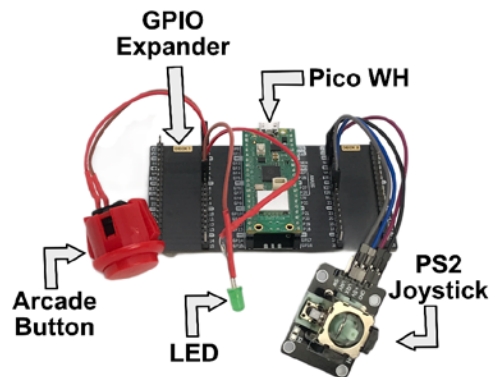


Figure 12.2 – Components of the IoT joystick

In *Figure 12.2*, the components we are using include a 24 mm arcade-style button, a green LED (any color will do) soldered to a 220 Ohm resistor (refer to *Chapter 3* for directions on this), and a PS2 joystick module.

This setup makes it easier to transfer our components to a custom case for final installation. Using these components, we wire up our circuit using *Figure 12.3* as a reference:

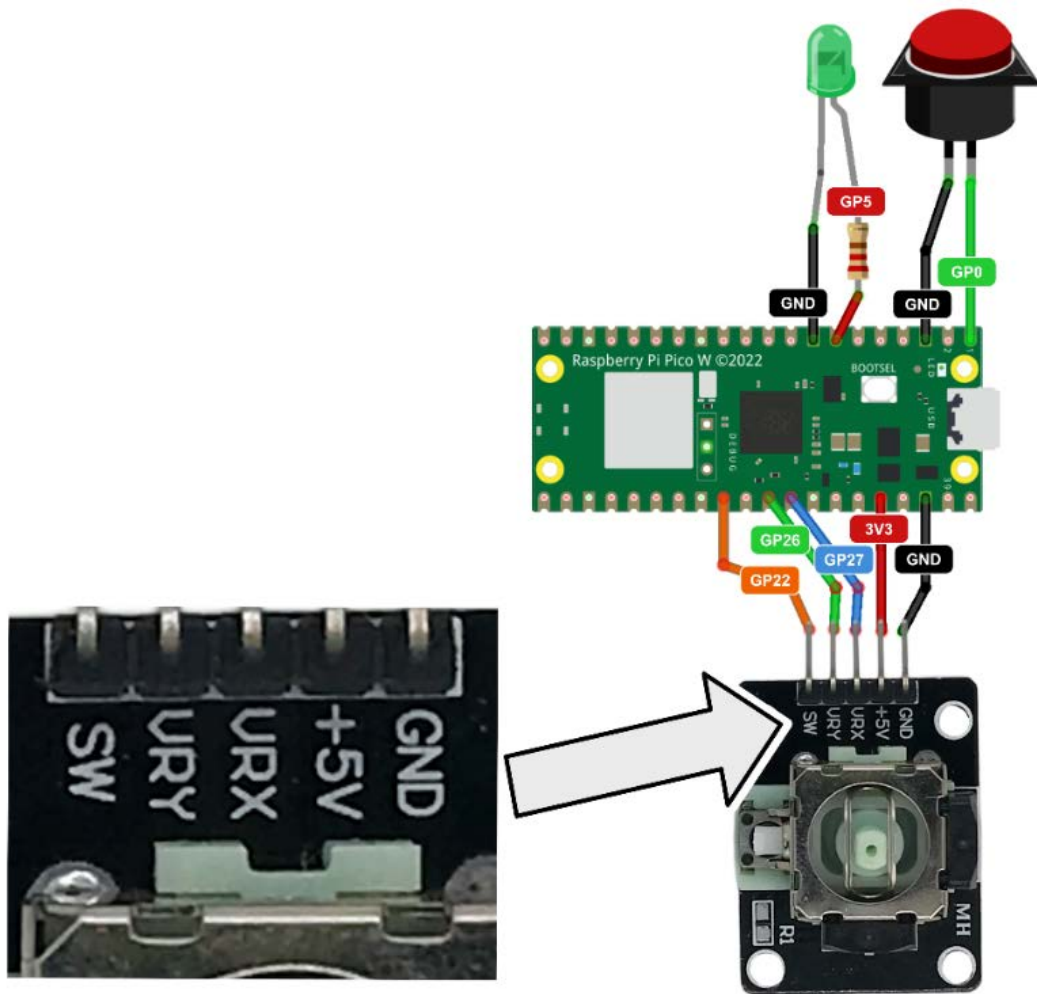


Figure 12.3 – Wiring up the IoT joystick

The connections are as follows:

- The arcade-style button connects to GP0 and GND.
- The positive end of the LED with a 220 Ohm resistor connects to GP5.
- The negative end of the LED connects to GND.
- SW (switch) from the PS2 joystick module connects to GP22.
- VRY (y axis) from the PS2 joystick module connects to GP26.
- VRX (x axis) from the PS2 joystick module connects to GP27.
- +5V (power) from the PS2 joystick module connects to 3V3.
- GND from the PS2 joystick module connects to GND.

With our circuit wired up, we are ready to start coding our application.

## Developing the code for our IoT joystick

We will install CircuitPython on our Raspberry Pi Pico WH and use Thonny for development. Our Pico WH code will consist of two files, one to encapsulate the joystick functionalities and the other to transmit MQTT messages.

We will start by setting up our Raspberry Pi Pico WH for development.

### Setting up our Raspberry Pi Pico WH

For our IoT joystick, we will install CircuitPython and use the Adafruit MiniMQTT library. We could just as easily use MicroPython and the `micropython-umqtt.simple` package. However, using CircuitPython for the Raspberry Pi Pico WH in our IoT joystick application offers more stable and well-maintained libraries compared to MicroPython.

To install CircuitPython on our Raspberry Pi Pico WH, we do the following same steps as we did in *Chapter 9*:

1. If Thonny is not available on our operating system, we visit the Thonny website and download an appropriate version (<https://thonny.org>).
2. We then launch Thonny using the appropriate method for our operating system.
3. While holding the *BOOTSEL* button on the Pico WH (the small white button near the USB port), we insert it into an available USB port and disregard any pop-up windows that may appear.
4. We then click on the interpreter information at the bottom right-hand side of the screen and select **Install CircuitPython...**:

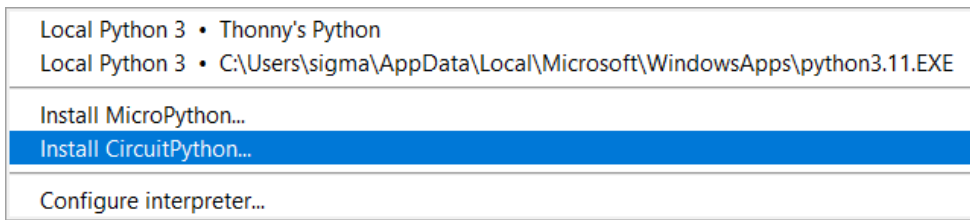


Figure 12.4 – Install CircuitPython... option

- For our target volume, we select our Pico WH (**RPI-RP2 (D:)** in our example). We then select the latest version of the CircuitPython variant – **Raspberry Pi • Pico W / Pico WH**:

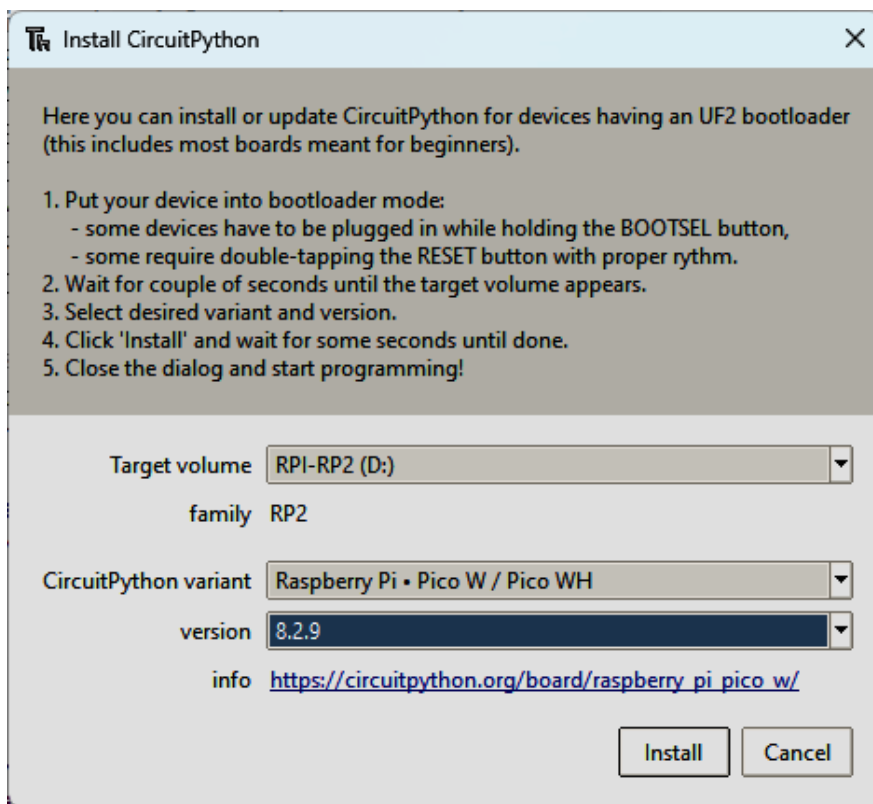


Figure 12.5 – Installing CircuitPython on the Raspberry Pi Pico WH

- We click on the **Install** button and then the **Close** button once the installation has completed.
- To have Thonny configured to run the CircuitPython interpreter on our Pico, we select it from the bottom right-hand side of the screen:

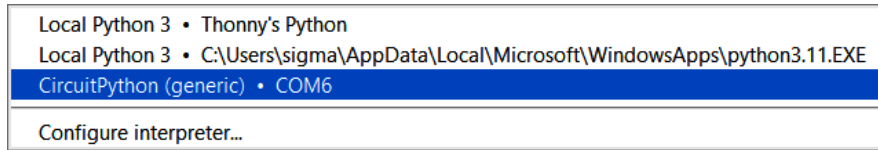


Figure 12.6 – Selecting the CircuitPython interpreter from our Pico WH

8. With CircuitPython installed on our Raspberry Pi Pico WH, the next step is to install the Adafruit MiniMQTT library. To do so, we follow the steps outlined in *Chapter 10*, in the *Installing the CircuitPython library for MQTT* section.

With our Raspberry Pi Pico WH ready for development, it's time to create a `Joystick` class.

## Creating a Joystick class

As mentioned, our Pico WH code is broken down into two files, one to encapsulate the joystick in a class we call `Joystick` and the other to send MQTT messages based on the values of this class. We start with the joystick code.

To write the joystick code, we do the following:

1. We connect our Raspberry Pi Pico WH to a USB port and launch Thonny. We may use our Raspberry Pi or another operating system for this.
2. We then activate the CircuitPython environment on our Pico WH by selecting it from the bottom right-hand side of the screen.
3. In a new editor, we start our code with the imports:

```
import board
import digitalio
import analogio
import time
```

In our code, we have the following:

- `import board`: Accesses the board-specific pins and hardware interfaces, crucial for interfacing with the GPIO pins on the Raspberry Pi Pico W.
- `import digitalio`: Manages digital input and output, such as reading the state of buttons or controlling LEDs, essential for digital signal interactions.
- `import analogio`: Facilitates analog input functionalities, such as reading sensor data that vary over a range, a common requirement in projects involving variable inputs such as potentiometers.
- `import time`: Provides time-related functions, enabling tasks such as introducing delays in the program execution, which is useful in controlling the flow and timing of operations.

4. We then define a `Joystick` class and set the variables:

```
class Joystick:
 def __init__(self):
 self.adc_x = analogio.AnalogIn(board.GP27)
 self.adc_y = analogio.AnalogIn(board.GP26)

 self.button = digitalio.DigitalInOut(board.GP0)
 self.button.direction = digitalio.Direction.INPUT
 self.button.pull = digitalio.Pull.UP

 self.button2 = digitalio.DigitalInOut(board.GP22)
 self.button2.direction = digitalio.Direction.INPUT
 self.button2.pull = digitalio.Pull.UP

 self.mid = 32767
 self.dead_zone = 10000
```

In our code, we have the following:

- Initialization (`__init__()` method): Our code sets up the `Joystick` class.
  - `self.adc_x` and `self.adc_y`: These are analog input objects for the  $x$  and  $y$  axes of the joystick, connected to the GP27 and GP26 pins, respectively. They read the analog values from the joystick's potentiometers.
  - `self.button`: A digital input/output object for a button, connected to the GP0 pin. It's configured as an input with a pull-up resistor, which is a common setup for buttons. This variable represents the state of our arcade-style button.
  - `self.button2`: Similar to `self.button`. This represents a second button connected to the GP22 pin, also set as an input with a pull-up resistor. This variable represents the state of the joystick button (activated by pushing down on the joystick).
  - `self.mid`: The midpoint value for the analog readings, used to determine the neutral position of the joystick.
  - `self.dead_zone`: The dead zone threshold determines the joystick's sensitivity, distinguishing slight, unintentional movements from deliberate ones. This accounts for minor variances when the joystick is in its neutral position.
5. With our variables set, we write our first method, which we call `get_binary_value()`. This function is designed to interpret the joystick's position as a binary output. It first checks if the joystick's current position, represented by `value`, is within a predefined dead zone around the midpoint (`self.mid`). If so, it returns 0, indicating the joystick is in a neutral position. If the joystick's position is outside this dead zone, the function returns -1 for positions below the midpoint (negative direction) and 1 for positions above it (positive direction):

```
def get_binary_value(self, value):
 if abs(value - self.mid) < self.dead_zone:
 return 0
 return -1 if value < self.mid else 1
```

6. We then define our second method, `read()`. This method consolidates the joystick's and buttons' statuses into a single output. It computes binary values for the *x* and *y* axes of the joystick using the `get_binary_value()` method, translating analog readings into binary (-1, 0, or 1) based on position. It also assesses the states of two buttons, converting their digital values into a Boolean format (pressed or not pressed). The method then returns a tuple containing these binary values and button states:

```
def read(self):
 x_val = self.get_binary_value(self.adc_x.value)
 y_val = self.get_binary_value(self.adc_y.value)

 button_state = not self.button.value
 button2_state = not self.button2.value

 return x_val, y_val, button_state, button2_state
```

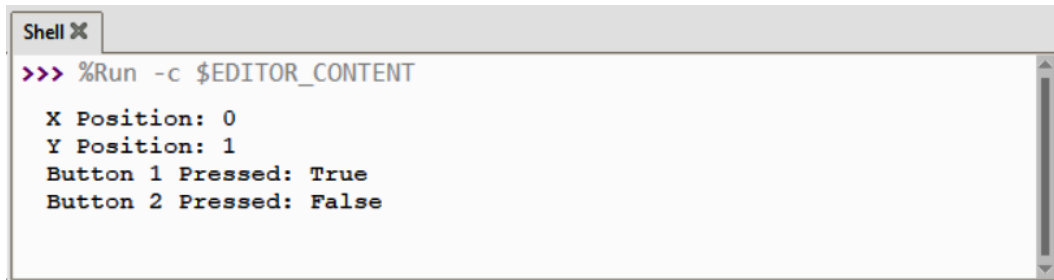
7. We then write our test code. We use this code to test our `Joystick` class. It initializes an instance of the `Joystick` class and enters an infinite loop to continuously test the functionality of the joystick. In each iteration of the loop, it reads the current positions of the joystick's *x* and *y* axes and the states of the two buttons using the `read()` method. These values are then printed to the console, displaying the joystick's *x* and *y* positions along with the press status of each button. The `if __name__ == "__main__":` block ensures that this main loop runs only if the script is executed as the main program, not when imported as a module, allowing us to easily test our `Joystick` class:

```
if __name__ == "__main__":
 joystick = Joystick()

 while True:
 x, y, button_state, button2_state = joystick.read()
 print("X Position:", x)
 print("Y Position:", y)
 print("Button 1 Pressed:", button_state)
 print("Button 2 Pressed:", button2_state)
 time.sleep(5)
```



8. To save the file, we click on **File | Save as...** from the drop-down menu. We save our file as `joystick.py` to our Raspberry Pi Pico WH.
9. To run our code, we click on the green **Run** button, hit *F5* on the keyboard, or click on the **Run** menu option at the top, and then **Run current script**.
10. In the shell, we should observe the joystick values change as we move the joystick and press the buttons:



```
>>> %Run -c $EDITOR_CONTENT
X Position: 0
Y Position: 1
Button 1 Pressed: True
Button 2 Pressed: False
```

Figure 12.7 – Joystick and button state

With our `Joystick` class and `joystick.py` file written and successfully tested, we are now ready to write the code to send out the joystick status via MQTT messaging.

## Sending MQTT messages from our IoT joystick

With the `Joystick` class created and tested, it is time to write the code to send the joystick and button status in an MQTT message.

To write the joystick code, we do the following:

1. We connect our Raspberry Pi Pico WH to a USB port and launch Thonny. We may use our Raspberry Pi or another operating system for this.
2. We then activate the CircuitPython environment on our Pico WH by selecting it from the bottom right-hand side of the screen (see *Figure 12.6*).
3. In a new editor, we start our code with the imports:

```
import time
import board
import wifi
import socketpool
import digitalio
from adafruit_minimqtt.adafruit_minimqtt import MQTT
from joystick import Joystick
```

In our code, we have the following:

- `import time`: Incorporates time-related functions, useful for delays and timing control in the code.
- `import board`: Provides access to the hardware-specific details of the Raspberry Pi Pico WH, particularly its GPIO pins.
- `import wifi`: Enables Wi-Fi functionality, crucial for network connectivity in IoT applications.
- `import socketpool`: Manages network sockets, used for network communications, such as MQTT messaging.
- `import digitalio`: Allows for digital input and output control, useful for managing LEDs, buttons, and other digital components.
- `from adafruit_minimqtt.adafruit_minimqtt import MQTT`: Imports the Adafruit MiniMQTT library, which is used for handling MQTT protocol communication, a standard for IoT messaging.
- `from joystick import Joystick`: Imports our custom Joystick class for handling the logic for interfacing with our joystick module.

4. We then set our variable declarations:

```
WIFI_SSID = 'MySSID'
WIFI_PASSWORD = 'SSID-password'
MQTT_SERVER = "mqtt-server"
MQTT_PORT = 18756
USERNAME = "mqtt-username"
PASSWORD = "mqtt-password"
MQTT_TOPIC = "JoystickPosition"

led = digitalio.DigitalInOut(board.GP5)
led.direction = digitalio.Direction.OUTPUT
```

In our code, we have the following:

- `WIFI_SSID` and `WIFI_PASSWORD`: These variables store the credentials for the Wi-Fi network, essential for connecting the Raspberry Pi Pico WH to the internet.
- `MQTT_SERVER`, `MQTT_PORT`, `USERNAME`, `PASSWORD`: These settings are for configuring the MQTT client. They specify the server address, port number, and authentication credentials needed to connect to our CloudAMQP server.
- `MQTT_TOPIC`: Defines the MQTT topic under which the messages will be published. In this case, it's set to `JoystickPosition`, indicating that messages related to the joystick's position will be sent to this topic.

- `led = digitalio.DigitalInOut(board.GP5)`: Initializes a digital output on pin GP5 of the Raspberry Pi Pico W for our LED.
  - `led.direction = digitalio.Direction.OUTPUT`: Sets the direction of the pin to output, allowing our program to control the LED (for example, turning it on or off).
5. We use the `flash_led()` method to turn on and off our LED to use it as a status indicator. Our method takes in times and duration values to allow for custom flashing of the LED based on a particular program state:

```
def flash_led(times, duration):
 for _ in range(times):
 led.value = True
 time.sleep(duration)
 led.value = False
 time.sleep(duration)
```

6. The `connect_to_wifi()` method is used to connect our Raspberry Pi Pico WH to our Wi-Fi router and the internet. This function continuously attempts to connect to Wi-Fi, using the `WIFI_SSID` and `WIFI_PASSWORD` credentials. On failure, it prints an error message, flashes an LED, and retries after a 3-second pause:

```
def connect_to_wifi():
 while True:
 try:
 print("Trying to connect to WiFi...")
 wifi.radio.connect(WIFI_SSID, WIFI_PASSWORD)
 print("Connected to Wi-Fi!")
 break
 except Exception as e:
 print("Failed to connect to WiFi. Retrying...")
 flash_led(1, 2)
 time.sleep(3)
```

7. The `connect_to_mqtt()` function attempts to establish a connection to the MQTT server in a loop. If the connection is successful, it prints a confirmation message and exits the loop. In case of a connection failure, it reports the failure, activates a half-second LED flash as an error indicator, and then waits for 3 seconds before retrying:

```
def connect_to_mqtt(mqtt_client):
 while True:
 try:
 print("Trying to connect to MQTT Broker...")
 mqtt_client.connect()
 print("Connected to MQTT server!")
```

```

 break
 except Exception as e:
 print("Failed to connect to MQTT. Retrying...")
 flash_led(1, 0.5)
 time.sleep(3)

```

8. Our code then calls `connect_to_wifi()` to establish a Wi-Fi connection. Next, a socket pool is created from `wifi.radio` to manage network communication. An MQTT client is then instantiated with the specified server, port, and user credentials and connected to the MQTT broker using `connect_to_mqtt(mqtt_client)`. After establishing an MQTT connection, the LED is set to a steady on state (`led.value = True`) as an indicator of successful setup. Finally, an instance of the `Joystick` class is created, readying it for capturing user inputs:

```

connect_to_wifi()

pool = socketpool.SocketPool(wifi.radio)

mqtt_client = MQTT(broker=MQTT_SERVER, port=MQTT_PORT,
username=USERNAME,
password=PASSWORD,
socket_pool=pool)

connect_to_mqtt(mqtt_client)

led.value = True

joystick = Joystick()

```

9. The `send_mqtt_message()` function in the code is responsible for formatting and sending joystick data over MQTT. It accepts the joystick's *x* and *y* axes' values and the states of two buttons. The states of the buttons are converted to `True` or `False` based on whether they are pressed. The function then constructs a message string that includes the *x* and *y* positions and the states of both buttons. This message is published to the MQTT topic defined by `MQTT_TOPIC`, allowing the joystick's status to be transmitted via the MQTT protocol:

```

def send_mqtt_message(x, y, button1, button2):
 button1_state = True if button1 else False
 button2_state = True if button2 else False
 message = f'X: {x}, Y: {y}, Button 1: \
 {button1_state}, Button 2: {button2_state}'
 mqtt_client.publish(MQTT_TOPIC, message)

```

10. The `main()` function in the code represents the primary loop for reading joystick inputs and sending corresponding MQTT messages. Within an infinite loop, it continually calls the `joystick.read()` method to get the current positions of the joystick's *x* and *y* axes and the states of two buttons. It then passes these values to the `send_mqtt_message()` function to format and transmit them as MQTT messages. A 1-second delay (`time.sleep(1)`) is included at the end of each loop iteration to manage the frequency of MQTT transmissions. The `if __name__ == "__main__":` block ensures that this main loop runs only if the script is executed as the main program, not when imported as a module:

```
def main():
 while True:
 x, y, button1_pressed, button2_pressed = joystick.read()
 send_mqtt_message(x, y, button1_pressed, button2_
pressed)
 time.sleep(1)

if __name__ == "__main__":
 main()
```

11. To save the file, we click on **File | Save as...** from the drop-down menu. We save our file as `code.py` to our Raspberry Pi Pico WH.
12. To run our code, we click on the green **Run** button, hit *F5* on the keyboard, or click on the **Run** menu option at the top, and then **Run current script**.
13. To test our code, we connect the MQTT-Explorer app to our CloudAMQP server and observe the messages received.
14. We should observe outputs as three distinct values: 1, 0, and -1, indicating their neutral, positive, and negative positions respectively. Additionally, we should see the state of two buttons (True or False): Button 1 corresponds to the arcade-style button, and Button 2 indicates the joystick's click action.

With the code completed for our IoT joystick, it is time to create our custom `robot_control` ROS node so that we may control the TurtleSim robot.

## Creating a custom ROS node for our application

In *Chapter 11's Creating an ROS workspace and package* section, we outlined how to set up a `circle` node for TurtleSim robot control. Following this blueprint, we'll now create a `robot_control` node on the ROS-equipped Ubuntu system from *Chapter 11*. This involves setting up a new ROS workspace and package for the `robot_control` node, enabling us to control the TurtleSim robot using our new IoT joystick.

To ensure clarity and avoid any potential mix-up with the existing `circle` node, we'll create a new workspace and package for the `robot_control` node, despite the possibility of reusing those from *Chapter 11*. This approach allows us to maintain distinct environments for each project.

## Creating our custom `robot_control` node

As we have already installed the `paho-mqtt` Python library onto our Ubuntu installation, we will not need to install it again.

To create our new node, we do the following:

1. We open an Ubuntu Terminal and execute the following command in our home directory:

```
mkdir -p ch12_ws/src
```

2. With the folder created, we navigate to the `src` folder with the following command:

```
cd ch12_ws/src
```

3. To initialize the ROS environment, we execute the following command:

```
source /opt/ros/humble/setup.bash
```

4. We then create our package by executing the following command:

```
ros2 pkg create --build-type ament_python --license Apache-2.0
--node-name robot_control mqtt_robot
--license Apache-2.0
```

5. With this line, we created a new package named `mqtt_robot` with the Python build type and generated a node named `robot_control`. This will give us a Python script named `robot_control.py`. To navigate to the folder that contains this script, we enter the following command:

```
cd ~/ch11_ws/src/mqtt_robot/mqtt_robot
```

6. To open the `robot_control.py` file in a text editor, we execute the following command:

```
gedit robot_control.py
```

7. We start by deleting all the code. We then start our new code with imports:

```
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
import paho.mqtt.client as mqtt
```

In our code, we have the following:

- `import rclpy`: Imports the ROS 2 client library for Python, allowing the script to interact with ROS 2 functionalities and create ROS 2 nodes
  - `from rclpy.node import Node`: Imports the `Node` class from the `rclpy` module, enabling the script to define custom nodes for ROS 2 applications
  - `from geometry_msgs.msg import Twist`: Imports the `Twist` message type from the `geometry_msgs` package; we use this for sending commands to move the `TurtleSim` robot
  - `import paho.mqtt.client as mqtt`: Imports the Paho MQTT client library we will use for MQTT protocol communication
8. In our code, we create an `MQTTMessage` class. This class effectively parses and updates its properties based on the content of an MQTT message, which contains joystick position data and button states:

```
class MQTTMessage:
 def __init__(self):
 self.x = 0
 self.y = 0
 self.button1 = False
 self.button2 = False

 def update_values(self, message):
 parts = message.split(',')
 self.x = float(parts[0].split(':')[1])
 self.y = float(parts[1].split(':')[1])
 self.button1 = parts[2].split(':')[1].strip() == "True"
 self.button2 = parts[3].split(':')[1].strip() == "True"
```

In our code, we have the following:

- `class MQTTMessage`: Defines a class for handling MQTT messages.
- `__init__()` method:
  - Initializes `x` and `y` to 0, representing default positions.
  - Sets `button1` and `button2` to `False`, indicating their default unpressed state.
- `update_values()` method:
  - Takes a `message` string as input and splits it into parts based on commas.
  - Parses the `message` to extract and convert `X` and `Y` values to floats.
  - Determines the states of `button1` and `button2` by splitting the string parts and comparing them with `"True"`. The `strip()` method is used to remove any leading/trailing whitespace.

9. The `RobotController` class is a subclass of the `Node` class:

```
class RobotController(Node):
 def __init__(self, mqtt_message):
 super().__init__('robot_controller')
 self.mqtt_message = mqtt_message
 self.publisher = self.create_publisher(
 Twist, 'turtle1/cmd_vel', 10)
 timer_period = 0.1
 self.timer = self.create_timer(
 timer_period, self.timer_callback)
 self.vel_msg = Twist()
 self.mqtt_client = mqtt.Client()
 self.mqtt_client.on_connect = self.on_connect
 self.mqtt_client.on_message = self.on_message
 self.mqtt_client.username_pw_set(
 "mqtt-username", "mqtt-password")
 self.mqtt_client.connect(
 "driver.cloudmqtt.com", 18756, 60)
 self.mqtt_client.loop_start()
```

In our code, we have the following:

- We define `RobotController` as a subclass of `Node`
- `__init__()` method:
  - Initializes the node with the name `robot_controller`
  - Stores `mqtt_message` passed as a parameter
  - Creates a publisher for ROS `Twist` messages on the `turtle1/cmd_vel` topic
  - Sets a timer for a periodic callback function with a 0.1-second interval
  - Initializes `self.vel_msg` as a `Twist` object for velocity commands
- MQTT client configuration:
  - Creates a new MQTT client
  - Sets up connection and message callbacks (`on_connect()` and `on_message()` methods)
  - Configures the client with a username and password for MQTT
  - Establishes a connection to the MQTT server at `driver.cloudmqtt.com` on port 18756 (verify port number in the **Instance Details** section of the CloudMQTT dashboard)
  - Starts the MQTT client loop for asynchronous operation



10. The `on_connect()` method is used to handle MQTT client connections. Upon successfully connecting (indicated by `rc` being 0), it prints a confirmation message and subscribes the client to the `JoystickPosition` topic, enabling the client to receive related messages. If the connection fails, it displays an error message with the specific `rc` code to help diagnose the issue. The method's parameters follow the Paho MQTT library's conventions:

```
def on_connect(self, client, userdata, flags, rc):
 if rc == 0:
 print("Connected successfully to MQTT Broker")
 client.subscribe("JoystickPosition")
 else:
 print(f"Failed to connect with error code {rc}.")
```

11. The `on_message()` method is used to handle incoming MQTT messages. Upon receiving a message, it decodes the message payload from bytes to a string and then updates the MQTT message object with the new values using the `update_values()` method:

```
def on_message(self, client, userdata, msg):
 self.mqtt_message.update_values(msg.payload.decode())
```

12. The `timer_callback()` method is the final method in the `RobotController` class. It adjusts the robot's movement based on the state of two buttons. If `button1` is pressed, it sets the robot to draw a circle in the counterclockwise direction. Pressing `button2` does the opposite, making the robot move in the clockwise direction. If neither button is pressed, the robot's linear and angular velocities are set based on the joystick's Y and X positions, respectively. After setting the velocities, the updated `vel_msg()` function is published to control the robot's movement:

```
def timer_callback(self):
 if self.mqtt_message.button1:
 self.vel_msg.linear.x = 1.0
 self.vel_msg.angular.z = 1.0
 elif self.mqtt_message.button2:
 self.vel_msg.linear.x = -1.0
 self.vel_msg.angular.z = -1.0
 else:
 self.vel_msg.linear.x = float(self.mqtt_message.y)
 self.vel_msg.angular.z = float(self.mqtt_message.x)
 self.publisher.publish(self.vel_msg)
```

13. The `main()` function is executed outside any class and serves as the entry point for running a ROS 2 node integrated with MQTT for robot control. It begins with initializing the ROS client library, then creates an instance of the `MQTTMessage` and `RobotController` classes, passing the MQTT message object to the latter. The application enters a ROS event loop to

process callbacks, including MQTT messages, ensuring the robot responds to joystick commands. Upon exit, it shuts down by destroying the ROS node and terminating the ROS client library:

```
def main(args=None):
 rclpy.init(args=args)
 mqtt_message = MQTTMessage()
 robot_controller = RobotController(mqtt_message)
 rclpy.spin(robot_controller)
 robot_controller.destroy_node()
 rclpy.shutdown()

if __name__ == '__main__':
 main()
```

14. With our code completed, we save the file using the same name, `robot_control.py`. Our next step is to update the `package.xml` file to include Python library dependencies for our code (refer to *Chapter 11* for more information on ROS packages). To do so, we open an Ubuntu Terminal and navigate to the folder containing the `package.xml` file:

```
cd ~/ch12_ws/src/mqtt_robot
```

15. To open `package.xml` in a text editor, we execute the following command:

```
gedit package.xml
```

16. We update the XML by adding the following lines before the last tag (`</package>`):

```
<exec_depend>rclpy</exec_depend>
<exec_depend>paho-mqtt</exec_depend>
```

17. We save our changes and close the editor.

With updates to `robot_control.py` and `package.xml`, we are now ready to compile our code and run our new node.

## Controlling a ROS TurtleSim robot using our IoT joystick

Before we can run our new node, we must compile it. As we did in *Chapter 11*, we compile our code using the `colcon` ROS tool. To compile and execute our new code, we do the following:

1. In Ubuntu, we open a new Terminal and source our ROS 2 environment:

```
source /opt/ros/humble/setup.bash
```

2. We then navigate to the root of our workspace:

```
cd ~/ch12_ws
```

3. To compile our code, we execute the following command:

```
colcon build
```

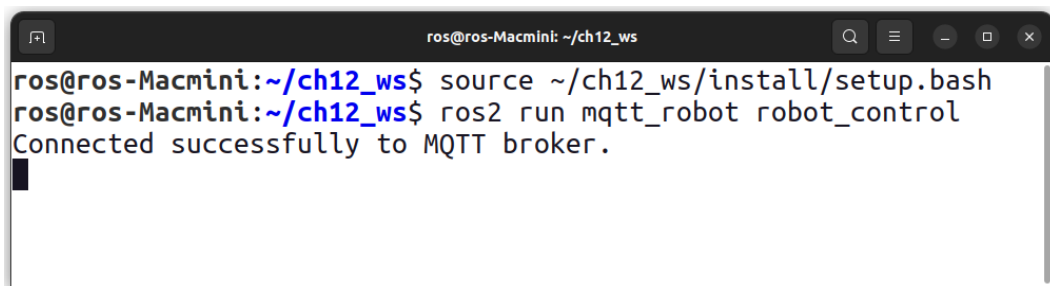
4. Upon completion, a message confirming a successful build will appear in the terminal.
5. With our code compiled, it is time to source our new ROS environment. We do this with the following command:

```
source ~/ch12_ws/install/setup.bash
```

6. This is similar to how we source the ROS environment. To run our node, we execute the following command:

```
ros2 run mqtt_robot robot_control
```

7. Here, we are running our new node, `robot_control`, from the package we created, `mqtt_robot`. We should observe a message indicating that we have successfully connected to the MQTT broker:

A terminal window titled 'ros@ros-Macmini: ~/ch12\_ws' showing the execution of two commands. The first command is 'source ~/ch12\_ws/install/setup.bash' and the second is 'ros2 run mqtt\_robot robot\_control'. The output of the second command is 'Connected successfully to MQTT broker.' followed by a cursor.

```
ros@ros-Macmini:~/ch12_ws$ source ~/ch12_ws/install/setup.bash
ros@ros-Macmini:~/ch12_ws$ ros2 run mqtt_robot robot_control
Connected successfully to MQTT broker.
```

Figure 12.8 – Running the `robot_control` node

8. In a separate Ubuntu Terminal, we launch an instance of the TurtleSim robot with the following commands:

```
source /opt/ros/humble/setup.bash
ros2 run turtlesim turtlesim_node
```

9. To start our IoT joystick, we may simply plug it into a USB power source or run the `code.py` program from Thonny.

10. We should observe that we can navigate the TurtleSim robot with our IoT joystick. Pressing and holding the main arcade-style button will make the robot draw a circle in the counterclockwise direction, while clicking and holding the joystick control down should make the robot draw a circle in the clockwise direction:



Figure 12.9 – Results of pressing the arcade-style button on the IoT joystick followed by pressing the joystick control button

With this, we have successfully managed the remote control of a virtual robot using our IoT joystick, demonstrating the system's global reach. This application demonstrates the vast potential for IoT and robotics integration.

For our final step, we'll encase the IoT joystick's components in a custom-designed housing. This enhances usability by making the device easier to manage and operate. Additionally, the custom case offers protection for the electronics and gives our application a professional finish.

## Constructing the IoT joystick case

As mentioned, our custom case gives our IoT joystick a professional look and gives us something to place in our hands. We assemble the custom case using 3D-printed parts and some common components.

The .stl files for the 3D-printed parts of our case may be found under the **Build Files** section of this chapter's GitHub repository. We may see the parts that make up the case in the following figure:

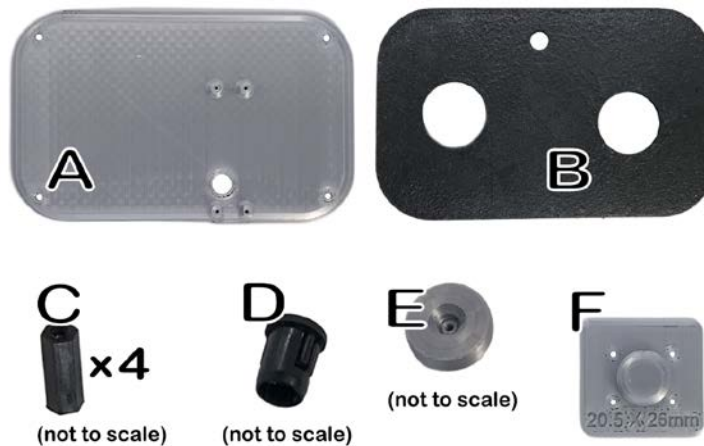


Figure 12.10 – Parts that make up the IoT joystick case

The parts that make up the IoT joystick case from *Figure 12.10* are as follows:

- *A*: Backplate. 3D printed from the .stl file.
- *B*: Front shell. 3D printed from the .stl file (the part in the figure has been painted).
- *C*: 4 x M2.5 10 mm stand-offs.
- *D*: 8 mm LED holder.
- *E*: Thumbstick for PS2 joystick stick module. 3D printed from the .stl file.
- *F*: Drilling guide for PS2 joystick module. 3D printed from the .stl file. We use a drilling guide as there are discrepancies in the position of mounting holes for various versions of the PS2 joystick module. The version we are using in our example has mounting holes that are 20.5 mm apart on the *y* axis and 26 mm apart on the *x* axis.
- *G*: 1 x M2 8 mm screw (not shown).
- *H*: 4 x M2 5 mm screws (not shown).
- *I*: 8 x M2.5 5 mm bolts (not shown).
- *J*: 4 x M3 10 mm bolts (not shown).

To construct the IoT joystick case, we begin by securing the thumbstick (*E* from *Figure 12.10*) to the PS2 joystick (as shown in *Figure 12.11*). We are replacing the thumbstick that comes with the PS2 joystick with our own to allow for more range. We can see the construction of the IoT joystick in the following figure:

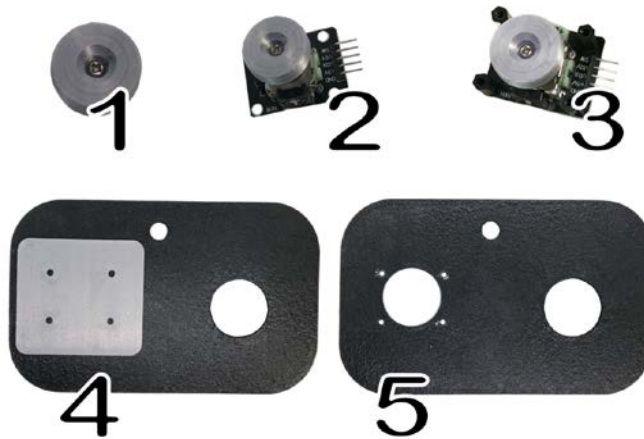


Figure 12.11 – Preparing our custom case for the PS2 joystick

To prepare our case for the PS2 joystick, we do the following:

1. Secure the M2 8 mm screw to the thumbstick such that a portion of it extends through the thumbstick (*Step 1* from *Figure 12.11*).
2. Fasten the thumbstick to the stem of the PS2 joystick by screwing it in by hand (*Step 2* from *Figure 12.11*).
3. Attach the four M2.5 10 mm standoffs to the PS2 joystick using four M2.5 5 mm bolts (*Step 3* from *Figure 12.11*).
4. Line up the drilling guide to the joystick hole of the front shell matching the orientation shown in *Step 4* of *Figure 12.11*. The PS2 joystick will be installed such that the pins will extend out to the right.
5. Using an appropriately sized drill bit, drill four holes into the front shell (*Step 5* from *Figure 12.11*).

With the thumbstick installed and the holes drilled, it is now time to construct the case. To do so, we follow the steps in the following figure:



Figure 12.12 – Constructing the IoT joystick case

Using *Figure 12.12* as a reference, we construct the IoT joystick case as follows:

1. We start by securing the Raspberry Pi Pico WH to the back plate using four M2 5 mm screws, positioning the USB port downward for easy access. This setup ensures the Pico's *Reset* button is accessible through a designated hole in the back plate. Opting for a Pico WH provides easy integration with our GPIO expander from the test circuit. While a Raspberry Pi Pico W can also be used, it requires soldering to install, making the Pico WH a more convenient choice for this application (*Step 1* from *Figure 12.12*).
2. We then secure the PS2 joystick to the front shell using four M2.5 5 mm bolts. We must ensure that we install the P2 joystick such that the pins are pointing to the right of the case (*Step 2* from *Figure 12.12*).
3. Using the LED holder, we attach the LED with the resistor to the front shell. We secure the arcade-style button to the front shell, as well as using the appropriate hole (*Step 3* from *Figure 12.12*).
4. We secure the backplate to the front shell using four M3 10 mm bolts (*Step 4* from *Figure 12.12*).
5. To power the IoT joystick, we attach a micro-USB cord to the USB port of the Raspberry Pi Pico WH.

We may now plug in our IoT joystick and use it to control a TurtleSim robot.

## Summary

In this chapter, we assembled our IoT joystick and used it to control a TurtleSim virtual robot. We started with component wiring and then proceeded to write code for transmitting joystick movements via MQTT messages.

Our application culminated in encasing the components in a custom 3D-printed case, enhancing the joystick's usability and durability. Through this application, we displayed the seamless integration of IoT devices with robotic systems.

In the next chapter, we will convert our virtual TurtleSim robot into a real-life physical robot and control it with our new IoT joystick.





# Introducing Advanced Robotic Eyes for Security (A.R.E.S.)

In this chapter, we will convert our TurtleSim virtual robot to a real-life robot we will call A.R.E.S. (short for Advanced Robotic Eyes for Security). A.R.E.S. will feature a video feed that we view over our local network using the VLC media player. We will control A.R.E.S. using the IoT joystick we created in *Chapter 12*.

We will construct A.R.E.S. using a Raspberry Pi as the brain or sensory input and a Raspberry Pi Pico for control of the motors, LEDs, and buzzer. We will use standard motors and a robotics board with our Raspberry Pi Pico H for motor control. We will 3D print the frame using the `.stl` files located in the `Build Files` directory of this chapter's GitHub repository.

We will cover the following topics in this chapter:

- Exploring our A.R.E.S. application
- Constructing A.R.E.S.
- Software setup and configuration
- Programming A.R.E.S. with ROS

Let's begin!

## Technical requirements

You will require the following to learn comprehensively from this chapter:

- Intermediate knowledge of Python programming
- Basic knowledge of the Linux command line

- A CloudAMQP account for the MQTT server instance
- IoT joystick from *Chapter 12*
- Access to a 3D printer or 3D printing service
- Build files for custom cases may be found in our GitHub repository

Refer to the *Constructing A.R.E.S.* section for hardware components required.

The code for this chapter may be found here:

<https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter13>

## Exploring our A.R.E.S. application

The A.R.E.S. robot presents an integration of various IoT components. It's operated via the IoT joystick we created in *Chapter 12* and communicates commands through MQTT to the Raspberry Pi. Our design will incorporate both a Raspberry Pi 3B+ and a Raspberry Pi Pico H. In the following diagram, we see the outline of the A.R.E.S. robot, including the connection from the IoT joystick:

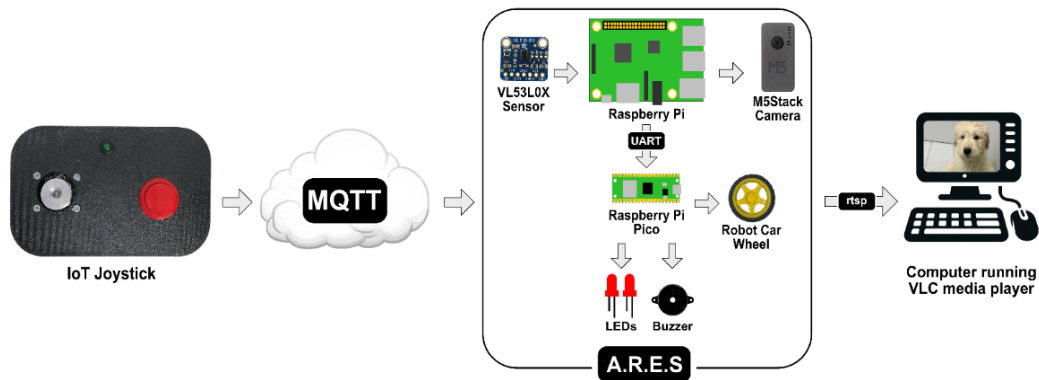


Figure 13.1 – The A.R.E.S. robot application

The Raspberry Pi 3B+, serving as the brain, uses **UART** (short for **Universal Asynchronous Receiver/Transmitter**) communication to relay commands to a Raspberry Pi Pico H, which in turn controls the car's movements, LEDs, and buzzer, responding dynamically to inputs. Equipped with a VL53L0X sensor, A.R.E.S. can measure distances, allowing it to avoid obstacles. Additionally, an M5Stack camera mounted on A.R.E.S. streams real-time video, which can be viewed on any computer using a VLC media player via the **Real-Time Streaming Protocol (RTSP)**.

### Using a Raspberry Pi 3B+ for A.R.E.S.

For A.R.E.S., we're selecting the Raspberry Pi 3B+ over newer models such as the 4 or 5 due to its power efficiency and cost. Its ability to run on standard cell phone battery packs makes it ideal for our needs, while its lower price and availability as a current model ensure both economic and practical benefits.

Kicking off the A.R.E.S. robot project, we'll first assemble the 3D-printed frame and install the necessary components. A.R.E.S. is engineered for compactness, making it an ideal robotic platform for educational purposes. Once the frame is complete, we'll move on to software, configuring the operating system on our Raspberry Pi 3B+ and programming the Raspberry Pi Pico H. Let's begin with constructing the frame.

## Constructing A.R.E.S.

A.R.E.S. consists of a frame made with 3D-printed parts and common components such as DC motors, LEDs, a Raspberry Pi 3B+, a Raspberry Pi Pico H, **ToF** (short for **time of flight**), a sensor, a Wi-Fi camera, battery packs, and various bolts and screws.

We will start our construction of A.R.E.S. by identifying the 3D-printed parts.

### Identifying the 3D-printed frame parts

We may find the .stl files of these parts under the `Build Files` directory of this chapter's GitHub repository. In the following figure, we see the parts printed out:

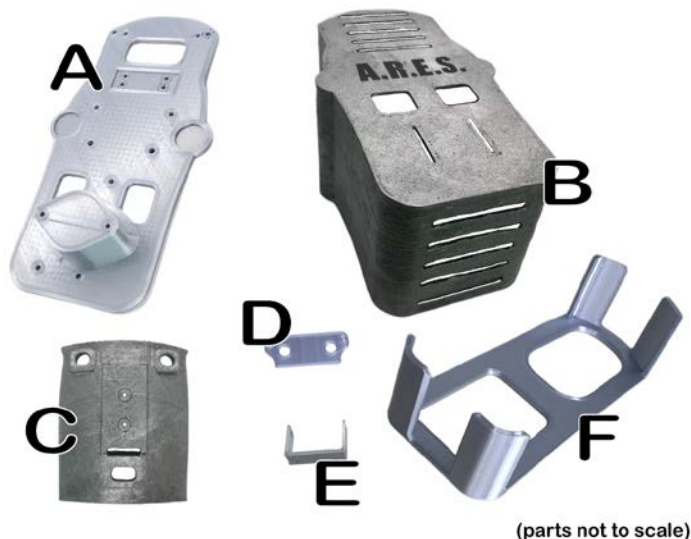


Figure 13.2 – A.R.E.S. 3D-printed parts

The 3D-printed parts that make up the frame of A.R.E.S. are the following:

- A: Base
- B: Shell
- C: Face
- D: Battery pack riser
- E: Motor bracket
- F: Testing stand (optional base used for testing purposes)

With the 3D-printed frame parts identified, let's look at the components used to construct A.R.E.S.

### Identifying the components used to create A.R.E.S.

The components we use to construct A.R.E.S. are standard electronic components and may be easily purchased online from vendors such as Amazon or AliExpress. The following figure outlines the components we use:

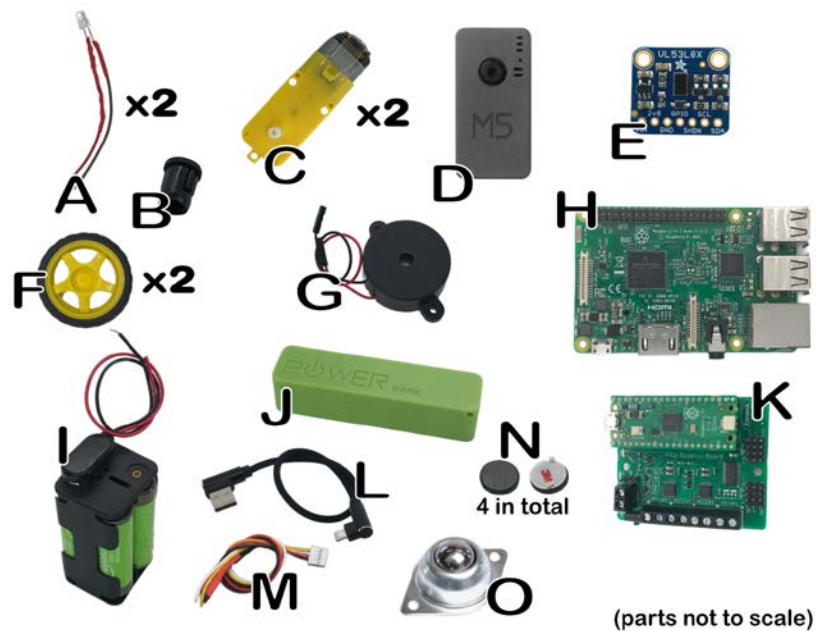


Figure 13.3 – Components that make up the A.R.E.S. robot

---

The components to construct A.R.E.S. are the following:

- *A*: 2 x LED with 220 Ohm resistor
- *B*: 2 x 5 mm (8 mm wide) LED holder
- *C*: 2 x TT DC robot motor
- *D*: M5Stack Timer Camera X with mount (not shown)
- *E*: Adafruit VL53L0X ToF sensor
- *F*: 2 x TT robot car wheel
- *G*: SFM-27 buzzer
- *H*: Raspberry Pi 3B+
- *I*: Battery pack for 4 AA batteries (with batteries)
- *J*: Cell phone USB battery pack
- *K*: Raspberry Pi Pico H with Kitronik Simply Robotics Motor Driver Board
- *L*: micro-USB to USB cable (short)
- *M*: Grove connector to female jumper wire connectors for connecting the camera to the GPIO ports of the Raspberry Pi 3B+
- *N*: 4 x 2 mm thick, 18 mm diameter magnets with double-sided adhesive pads
- *O*: Caster (32 mm width)
- *NOT SHOWN*: 18 x M3 10 mm bolts, 4 x M3 20 mm bolts, 8 x M3 nuts, 6 x M2.5 10 mm bolts, 2 x M4 10 mm bolts, 4 x M2.5 40 mm standoffs, 3 x M3 20 mm standoffs, jumper wires, crimping kit with connectors and wires (optional but recommended), hot glue gun, soldering iron

With our components in place, let's start to build our A.R.E.S. robot.

## Building A.R.E.S.

Using our 3D-printed frame parts and electronic components, it is now time to build A.R.E.S. To construct A.R.E.S., we use the following diagram as a guide:

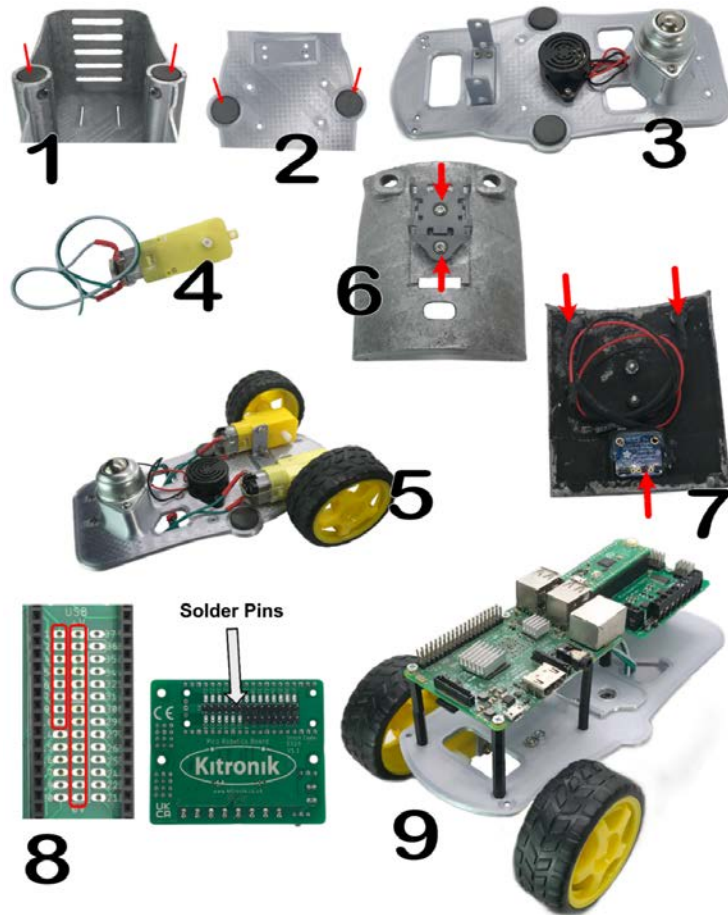


Figure 13.4 – Building the A.R.E.S. robot

The steps are as follows (the numbered steps also correspond to the numbered components in the figure):

1. Using double-sided adhesive tape (usually packaged with the product), we secure two magnets (*N* from *Figure 13.3*) to the shell (*B* from *Figure 13.2*).
2. Using the opposite polarity magnets (*N* from *Figure 13.3*), we attach the magnets (be sure to test before fastening) to the base (*A* from *Figure 13.2*).
3. Using two M4 10 mm bolts, we fasten the SFM-27 buzzer (*G* from *Figure 13.3*) to the base (*A* from *Figure 13.2*). The bolts should tap into the buzzer base; however, M4 nuts may be required. In this step, we also secure the caster (*O* from *Figure 13.3*) to the base (*A* from *Figure 13.2*) using two M3 bolts.

4. We solder 20 cm wires to each of the terminals of the TT motor (C from *Figure 13.3*).
5. Using the motor bracket (E from *Figure 13.2*), we fasten the TT motors (C from *Figure 13.3*) with the TT robot car wheel (F from *Figure 13.3*) to the base (A from *Figure 13.2*).
6. Using two M3 10 mm bolts, we fasten the camera mount that comes with the M5Stack Timer Camera X (D from *Figure 13.3*) to the face (C from *Figure 13.2*).
7. Using the LED holders (B from *Figure 13.3*) and LEDs with resistors (A from *Figure 13.3*), we thread the LEDs through the appropriate holes in the face (C from *Figure 13.2*). We secure the VL53L0X ToF sensor (E from *Figure 13.3*) to the face (C from *Figure 13.2*) using glue from a hot glue gun. We may also glue the LEDs in place to keep them from moving.
8. Access to the GP pins on the Raspberry Pi Pico H is required, but they are engaged within the motor board's **DIP** (short for **Dual Inline Package**) socket, rendering them inaccessible. To overcome this, we need to solder header pins to the underside of the motor board, enabling us to connect the SFM-27 buzzer (G from *Figure 13.3*) and LEDs with resistors (A from *Figure 13.3*) to the Raspberry Pi Pico H.
9. We secure four M2.5 40 mm standoffs to the front of the base (A from *Figure 13.2*) and three M3 20 mm standoffs to the back of the base (A from *Figure 13.2*) using 10 mm M2.5 and 10 mm M3 bolts respectively. We may secure the Raspberry Pi 3B+ (H from *Figure 13.3*) and Pico H with the motor board (K from *Figure 13.3*) to the standoffs; however, this will only be temporary as these components will be moved around as we wire up A.R.E.S. We may also temporarily place the battery pack riser (D from *Figure 13.2*) in place. We use the riser to cover the wires and provide a flat surface to place the battery pack (I from *Figure 13.3*) on.

With the frame assembled and components in place, it is now time to wire our components to the Raspberry Pi and Pico H.

## Wiring up A.R.E.S.

Wiring up A.R.E.S. requires connections to the Raspberry Pi 3B+ and Kitronik motor board. Using *Figure 13.1* as a reference, we can see that we connect the VL53L0X ToF sensor and M5Stack Timer Camera X to the Raspberry Pi 3B+, and the TT DC robot motors, LEDs with resistors, and buzzer to the Raspberry Pi Pico H using the motor board. We also wire the Raspberry Pi 3B+ to the Raspberry Pi Pico H to each other for use with UART communication.

We start our wiring with robot motors. In the following figure, we see a closeup of the motor board with the terminals to connect the battery pack and motors highlighted:



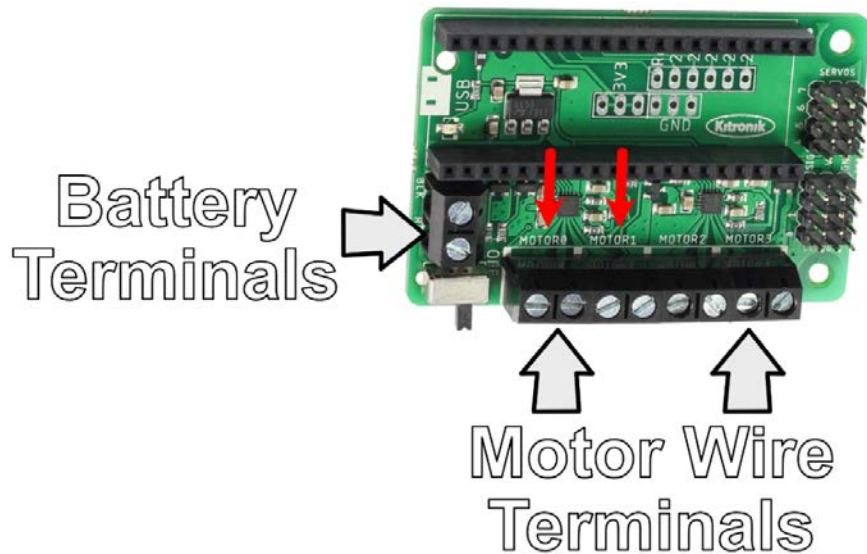


Figure 13.5 – Kitronik motor board for the Raspberry Pi Pico H

To wire up our motors, we do the following:

1. Attach the wires from the right motor, as depicted in *Step 9 of Figure 13.4*, to **Motor0** on the motor board. The polarity of the wires is not critical at this stage since we can correct their orientation later if necessary.
2. Attach the wires from the left motor to **Motor1** on the motor board.
3. Attach the wires from the AA battery pack to the battery terminals on the motor board, taking polarity into account.

With the robot motors and battery wires attached, it is now time to wire up the rest of the components. We will be using standard female jumper wires to make the connections. Although not required, having the ability to create our own jumper wires using a crimping kit makes for a clean and organized wiring setup. We use the wiring diagram in *Figure 13.6* for reference:

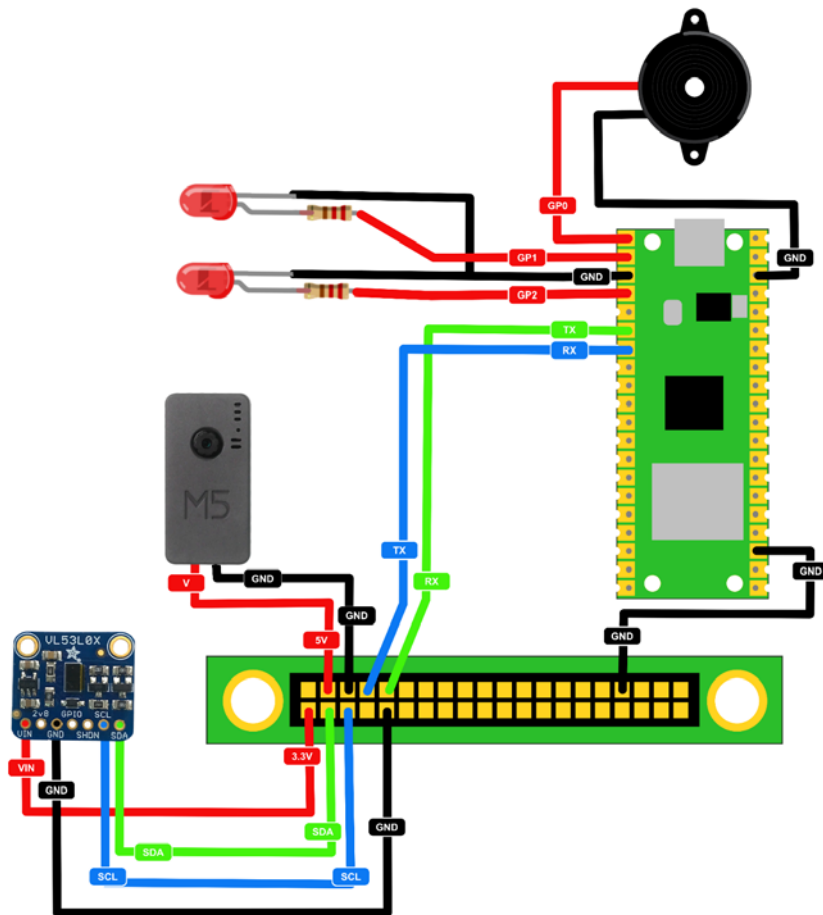


Figure 13.6 – Component wiring diagram for A.R.E.S.

To wire up the rest of our components, we do the following:

1. Using a grove connector to the female jumper wire connectors (*M* from *Figure 13.3*), we connect the M5Stack Timer Camera X to the Raspberry Pi 3B+ by attaching 5V from the Raspberry Pi to the V connector on the camera, and GND from the Raspberry Pi to G on the camera.
2. We connect VIN on the ToF sensor to 3.3V on the Raspberry Pi.
3. We connect SDA on the ToF sensor to SDA (GPIO 2) on the Raspberry Pi.
4. We connect SCL on the ToF sensor to SCL (GPIO 3) on the Raspberry Pi.
5. We connect GND on the ToF sensor to GND on the Raspberry Pi.

6. We connect TX (GPIO 14) on the Raspberry Pi to RX (GP5) on the Raspberry Pi Pico H or pin 7 on the motor board.
7. We connect RX (GPIO 15) on the Raspberry Pi to TX (GP4) on the Raspberry Pi Pico H or pin 6 on the motor board.
8. We connect GND on the Raspberry Pi to GND on the Raspberry Pi Pico H or a GND (0V) pin on the motor board.
9. We connect the positive wire on the SFM-27 buzzer to GP0 on the Raspberry Pi Pico H or pin 1 on the motor board.
10. We connect the negative wire on the SFM-27 buzzer to a GND (0V) pin on the motor board.
11. We connect the positive ends of the LEDs with resistors to GP1 and GP2 or pins 2 and 4 on the motor board.
12. We connect the negative ends of the LEDs with resistors to a GND (0V) pin on the motor board.

We may need to move the Raspberry Pi and motor board around as we make the connections. Also, it is advisable that we do not attach the face (*C* from *Figure 13.2*) to the base (*A* from *Figure 13.2*) initially as we require access to the microSD port on the Raspberry Pi 3B+.

With the wiring in place, let's set up the software for A.R.E.S. We will start by installing Ubuntu onto our Raspberry Pi 3B+.

## Software setup and configuration

To set up the software architecture of A.R.E.S., we will run a script from this chapter's GitHub repository. The script starts by ensuring that it is run with root privileges, updates and upgrades the system, and installs essential utilities and interfaces such as **I2C** (short for **Inter-Integrated Circuit**) and UART. It then proceeds to install Adafruit Blinka to support CircuitPython libraries, set up ROS Humble Hawksbill for robotics programming, and install the Colcon build system for software compilation.

The script also takes care of dependency management through `rosdep` and adds the ROS 2 environment setup to the `bashrc` file for easy access. By the end of the process, our Raspberry Pi 3B+ is fully configured for A.R.E.S.

Before running the script, we will use the Raspberry Pi Imager to burn Ubuntu onto a microSD card and install the card onto our Raspberry Pi 3B+. As the slot for the microSD card is in the front of the A.R.E.S. robot, the face will cover it. Thus, we will keep the face disconnected from the base while we install Ubuntu, as we see in the following figure:

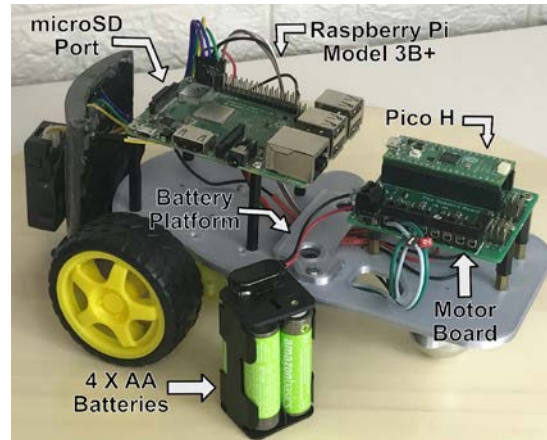


Figure 13.7 – Side view of A.R.E.S. with the face detached to allow access to the microSD card

We will run the Raspberry Pi Imager from a computer of our choice. For the examples in this chapter, we will install it on a Windows computer.

## Installing Ubuntu onto our Raspberry Pi 3B+

The Raspberry Pi Imager is a versatile tool designed to simplify the process of installing operating systems on Raspberry Pi devices. Developed by the Raspberry Pi Foundation, this utility allows us to flash various operating systems onto an SD card, which can then be used to boot and run on a Raspberry Pi.

While the Raspberry Pi Imager primarily supports the installation of Raspberry Pi OS (formerly known as Raspbian), its capabilities extend to a range of other operating systems. This allows us to experiment with different environments or require specific functionalities that are better supported by alternative OSs.

To use the Raspberry Pi Imager, we simply download and install the application on our computer, select the desired operating system from its extensive list, and then choose the target SD card for installation. The Raspberry Pi Imager can be installed on various operating systems, including Windows, macOS, and Linux. For example, in this chapter, we will install it onto a Windows machine. We will burn a command-line version of Ubuntu 22.04 to correspond to the Humble Hawksbill version of ROS.

To use the Raspberry Pi Imager to install Ubuntu onto the Raspberry Pi 3B+ on our A.R.E.S. robot, we navigate to the URL and download the imager for the OS we are using (<https://www.raspberrypi.com/software/>) and proceed to install the tool as follows:

1. We insert our microSD into a port on our computer.

2. After installation, we open the tool and select **RASPBERRY PI 3** for **Raspberry Pi Device**, **UBUNTU SERVER 22.04.4 LTS (64-BIT)** for **Operating System**, and the microSD card we inserted for the **Storage** option:

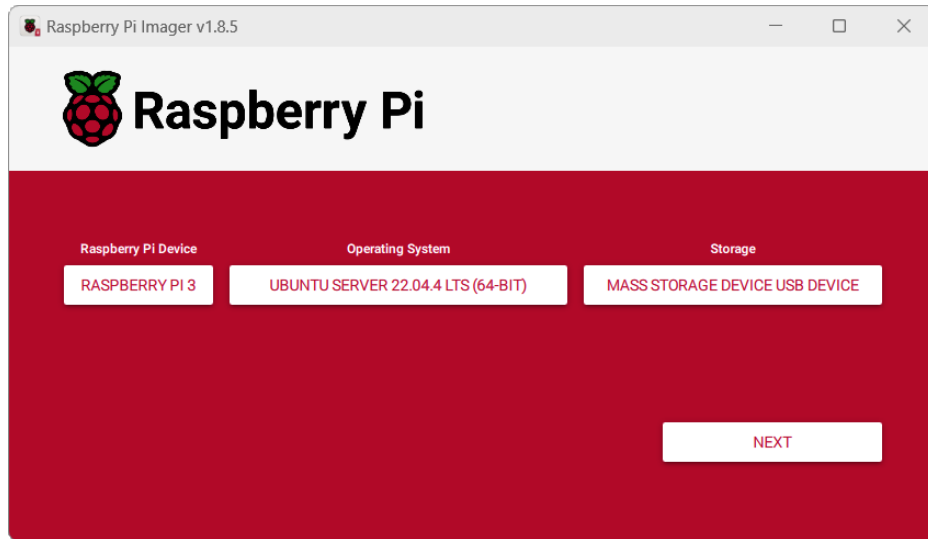


Figure 13.8 – Setting up the Raspberry Pi Imager

To proceed, we click on the **NEXT** button. This will bring us to the **Use OS customisation?** dialog:

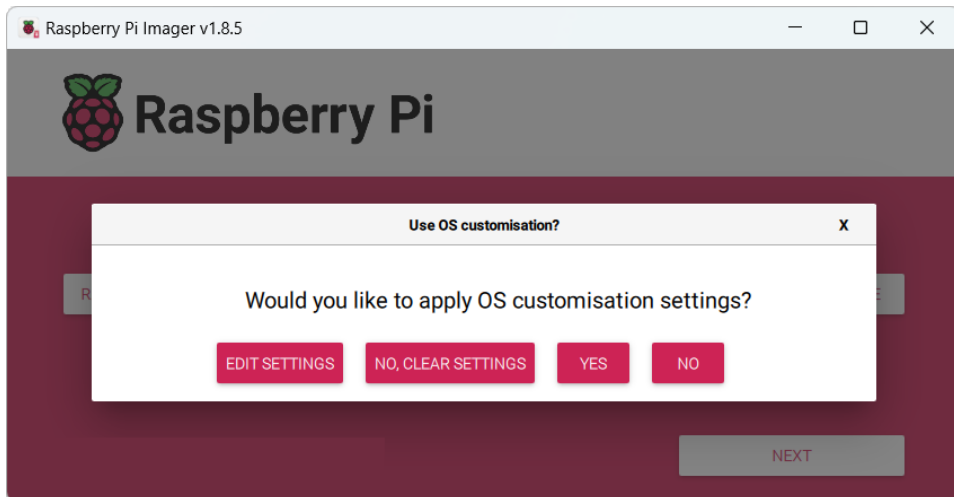
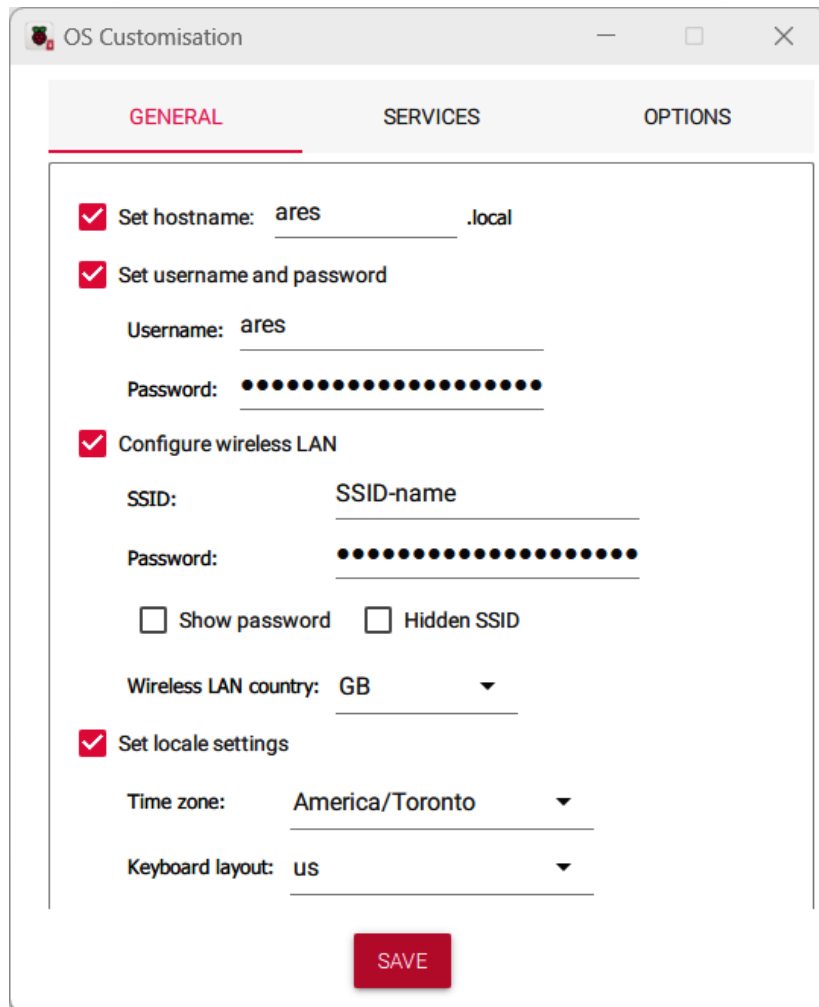


Figure 13.9 – Imager customization dialog

3. As we would like to set the name of the computer and the network, we click on the **EDIT SETTINGS** button and get the following screen:



The screenshot shows a window titled "OS Customisation" with three tabs: "GENERAL", "SERVICES", and "OPTIONS". The "GENERAL" tab is selected and highlighted with a red underline. The settings within this tab are as follows:

- ☒ Set hostname:  .local
- ☒ Set username and password
  - Username:
  - Password:
- ☒ Configure wireless LAN
  - SSID:
  - Password:
  - ☐ Show password    ☐ Hidden SSID
  - Wireless LAN country:  ▼
- ☒ Set locale settings
  - Time zone:  ▼
  - Keyboard layout:  ▼

At the bottom center of the window is a red button labeled "SAVE".

Figure 13.10 – OS Customisation screen

We set both the hostname and username to `ares`. We provide a password for the username and enter our SSID (LAN network) and SSID password.

- To enable remote access through SSH, we click on the **SERVICES** tab at the top and select **Enable SSH**:

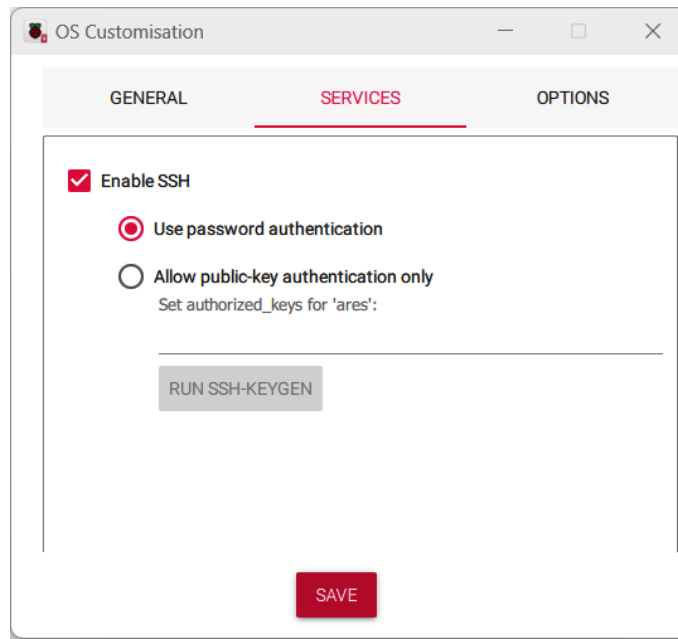


Figure 13.11 – Enable SSH

- To save our settings, we click on the **SAVE** button.
- To apply the settings, we click on the **YES** button.

We will then be presented with a warning:

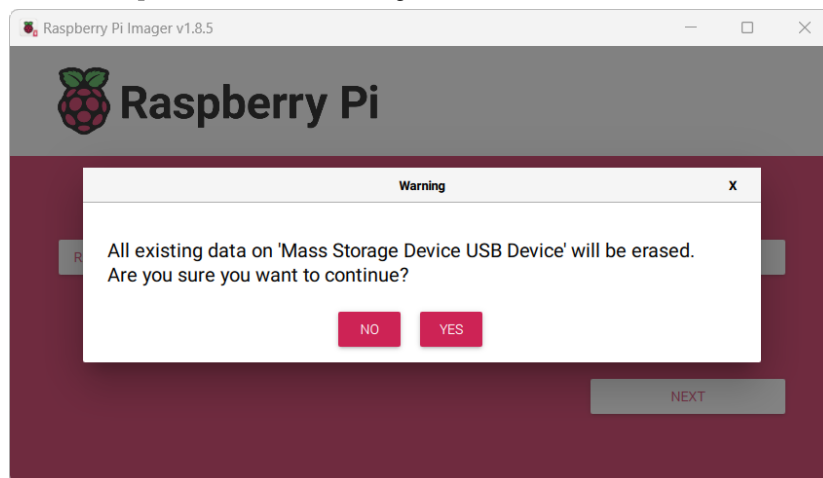


Figure 13.12 – Warning message

7. We click on **YES** as we want to erase any data on our microSD card and replace it with the Ubuntu operating system.

The Raspberry Pi Imager will then proceed to install the Ubuntu 22.04 operating system onto our microSD card, which we will install onto the Raspberry Pi 3B+ on A.R.E.S. We will not need to set up a Wi-Fi network or enable SSH.

With Ubuntu installed, it is now time to install ROS and the Python libraries we need for A.R.E.S. We will automate this with a specialized script stored in our GitHub repository.

## Running the installation script

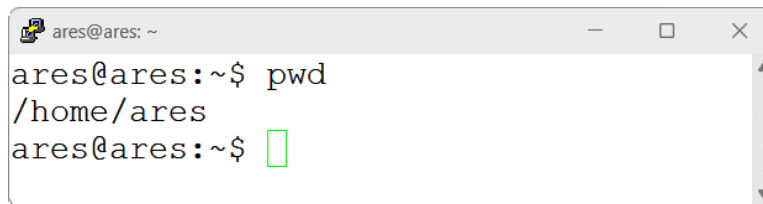
In previous chapters, we manually installed development libraries, a thorough yet time-intensive process. Now, leveraging our familiarity with Python libraries, we will streamline the setup of A.R.E.S. with ROS and necessary libraries through a script from our GitHub repository.

Executed with root privileges on Ubuntu, this script automates the installation, including ROS. Despite the direct installation into the OS diverging from *best practices*, it simplifies the process. For future projects, it is suggested that readers explore containerization with tools such as Docker.

To run the A.R.E.S. installation script, we do the following:

1. Using *Figure 13.7* as a reference, we ensure that we have access to the ports on the Raspberry Pi 3B+.
2. We connect a monitor, keyboard, and mouse to our Raspberry Pi and insert the freshly imaged microSD card.

As the server version of Ubuntu is command-line based, we will not be presented with a GUI when we boot up our Raspberry Pi. We log in using the credentials set during the imaging process. We should be in the home directory once logged in. We may verify this with the `pwd` command:

A terminal window titled 'ares@ares: ~' with standard window controls. The prompt is 'ares@ares:~\$'. The user has entered 'pwd' and the output is '/home/ares'. The prompt is now 'ares@ares:~\$' followed by a green cursor.

```
ares@ares:~$ pwd
/home/ares
ares@ares:~$
```

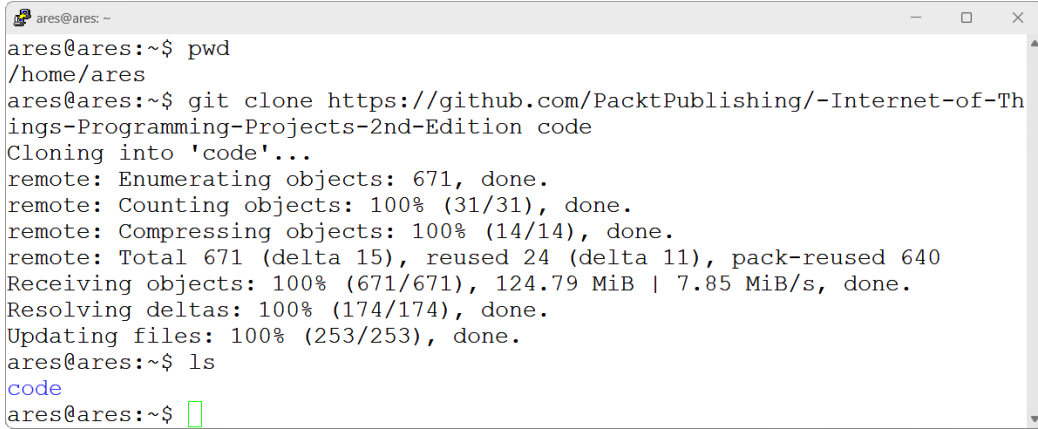
Figure 13.13 – Verifying current directory



3. The setup script is in the book's GitHub repository. To download the script and the Python code we use for the A.R.E.S. robot, we clone the repository onto our Raspberry Pi with the following command:

```
git clone https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition.git code
```

4. After running this command, we will have all the GitHub files inside a new directory called `code`. We may verify the creation of the `code` directory by running the `ls` command:



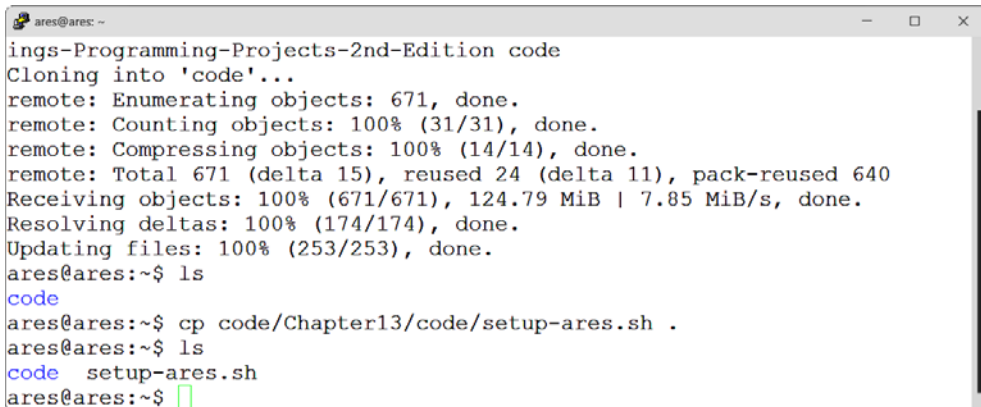
```
ares@ares: ~$ pwd
/home/ares
ares@ares:~$ git clone https://github.com/PacktPublishing/-Internet-of-Things-Programming-Projects-2nd-Edition code
Cloning into 'code'...
remote: Enumerating objects: 671, done.
remote: Counting objects: 100% (31/31), done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 671 (delta 15), reused 24 (delta 11), pack-reused 640
Receiving objects: 100% (671/671), 124.79 MiB | 7.85 MiB/s, done.
Resolving deltas: 100% (174/174), done.
Updating files: 100% (253/253), done.
ares@ares:~$ ls
code
ares@ares:~$
```

Figure 13.14 – Cloning repository

The script is located inside subdirectories of the `code` directory. We copy it to our current directory (`.`) with the following command:

```
cp code/Chapter13/code/setup-ares.sh .
```

5. We verify that our script was copied over successfully with the `ls` command:



```
ings-Programming-Projects-2nd-Edition code
Cloning into 'code'...
remote: Enumerating objects: 671, done.
remote: Counting objects: 100% (31/31), done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 671 (delta 15), reused 24 (delta 11), pack-reused 640
Receiving objects: 100% (671/671), 124.79 MiB | 7.85 MiB/s, done.
Resolving deltas: 100% (174/174), done.
Updating files: 100% (253/253), done.
ares@ares:~$ ls
code
ares@ares:~$ cp code/Chapter13/code/setup-ares.sh .
ares@ares:~$ ls
code setup-ares.sh
ares@ares:~$
```

Figure 13.15 – Verifying successful copy of setup script

6. We execute the script with admin privileges with the following command:

```
sudo bash setup-ares.sh
```

Executing the script with admin privileges ensures it has the necessary permissions to perform system-level changes and installations without encountering access restrictions. Our script initially updates our system before installing ROS and the necessary Python libraries. It should take several minutes to complete. After completion, we should be presented with the IP address of our machine so that we may log in remotely with SSH. This will be necessary when A.R.E.S. is running remotely:

```
ares@ares: ~
Skip end-of-life distro "jade"
Skip end-of-life distro "kinetic"
Skip end-of-life distro "lunar"
Skip end-of-life distro "melodic"
Add distro "noetic"
Add distro "rolling"
updated cache in /home/ares/.ros/rosdep/sources.cache
Adding ROS 2 environment setup to your bashrc...
Setup complete! Please reboot your system.
Use SSH to login to this Raspberry Pi. The IP address is:
10.0.0.36 2607:fea8:fe6:8800::538 2607:fea8:fe6:8800:ba27:ebff:fee
d:88b0
ares@ares:~$
```

Figure 13.16 – Result of running the setup script

7. With the completion of our setup script, we are now able to fasten the robot face to the base plate of A.R.E.S.:

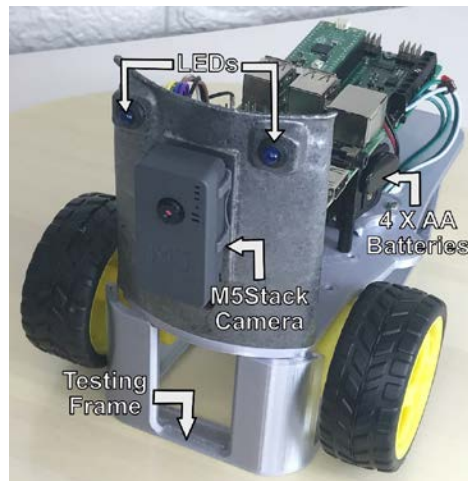


Figure 13.17 – Front view of A.R.E.S.

With the construction of A.R.E.S. and the OS on our Raspberry Pi 3B+ installed, it is time to install the code on our Raspberry Pi Pico H. Referring to *Figure 13.1*, we can see that A.R.E.S. uses the Pico H to control the motors, LEDs, and buzzer.

We will start by writing code to control the LED and buzzer.

## Creating alarm code for the Pico H

To program our Pico H, we will need to connect a micro-USB cable to the USB port on the Pico H. Despite our intentions to stagger the heights of the Pico H and Raspberry Pi 3B+, we may need to temporarily dismount the Raspberry Pi 3B+ from its standoffs to attach the micro-USB cable to the Pico H.

Once the micro-USB cable is attached, we may plug our Pico H into a computer of our choice and run Thonny. We will create a class called `Alarm` inside a file called `device_alarm.py` on our Pico H to encapsulate the alarm functionality. For simplicity's sake, we will couple the flashing of the LEDs with the activation of the buzzer.

To do this, we do the following:

1. Referring to the *Setting up our Raspberry Pi Pico WH* section from *Chapter 12*, we install CircuitPython onto our Raspberry Pi Pico although we select **Raspberry Pi • Pico / Pico H** for the **CircuitPython variant** option:

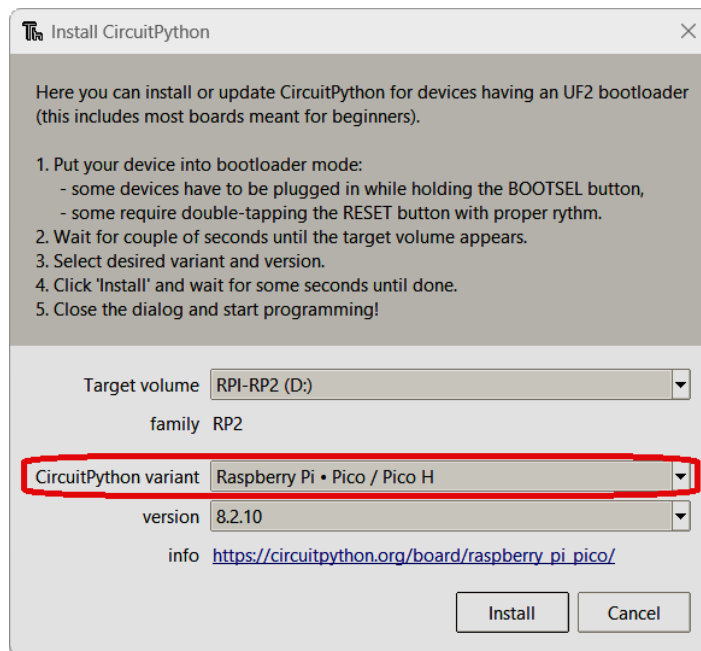


Figure 13.18 – Installing CircuitPython onto our Raspberry Pi Pico H

2. We then activate the CircuitPython environment on our Pico H by selecting it from the bottom right-hand side of the screen.
3. In a new editor, we start our code with the imports:

```
import time
import board
import pwmio
import digitalio
```

In our code, we have the following:

- `import time`: Provides time-related functions, enabling tasks such as introducing delays in the program execution, which is useful for controlling the flow and timing of operations.
- `import board`: Accesses board-specific pins and hardware interfaces, crucial for interfacing with the GPIO pins on the Raspberry Pi Pico W.
- `import pwmio`: We use this library to enable precise control over the buzzer by manipulating the frequency and duty cycle of **PWM** (short for **Pulse Width Modulation**) signals, affecting the pitch and volume of the sound produced.
- `import digitalio`: Manages digital input and output, such as reading the state of buttons or controlling LEDs, which is essential for digital signal interactions.

4. We then define an Alarm class and create an initialization method:

```
class Alarm:
 def __init__(self, buzzer_pin=board.GP1, led_pin1=board.GP0,
 led_pin2=board.GP2, frequency=4000):
 self.buzzer = pwmio.PWMOut(buzzer_pin,
 frequency=frequency, duty_cycle=0)

 self.led1 = digitalio.DigitalInOut(led_pin1)
 self.led1.direction = digitalio.Direction.OUTPUT

 self.led2 = digitalio.DigitalInOut(led_pin2)
 self.led2.direction = digitalio.Direction.OUTPUT
```

In our code, the following happens:

- I. We define a class named `Alarm`. The `__init__()` method takes optional parameters for the buzzer pin, two LED pins, and buzzer frequency with default values.
- II. We then initialize the buzzer on the specified pin as a PWM output with the given frequency and a duty cycle of 0 (off state).
- III. Our code sets up two LEDs on the specified pins as digital outputs, ready to be turned on or off.

5. Our class contains only one method, `activate_alarm()`:

```
def activate_alarm(self, num_of_times=5):
 blink_rate = 0.5
 for _ in range(num_of_times):
 self.buzzer.duty_cycle = 32768
 self.led1.value = True
 self.led2.value = True
 time.sleep(blink_rate)

 self.buzzer.duty_cycle = 0
 self.led1.value = False
 self.led2.value = False
 time.sleep(blink_rate)
```

In our code, we do the following:

- I. We define an `activate_alarm()` method within the `Alarm` class to activate the alarm a specified number of times (default is 5).
  - II. Inside the method, we set the `blink_rate` variable to 0.5 seconds, then loop for the specified number of times, toggling the buzzer and LEDs on and off according to the `blink_rate` variable.
6. To test our code and wiring, we use the following code:

```
alarm = Alarm(buzzer_pin=board.GP1, led_pin1=board.GP0, led_
pin2=board.GP2)
alarm.activate_alarm(10)
```

7. To save the file, we click on **File | Save as...** from the drop-down menu. We save our file as `device_alarm.py` to our Raspberry Pi Pico H.
8. To run our code, we click on the green **Run** button, hit *F5* on the keyboard, or click on the **Run** menu option at the top, and then **Run current script**.
9. We should observe the buzzer and LEDs blink for 10 repetitions.

#### Tip

To prevent the execution of test code within our application, we either delete or comment out this code segment.

With the alarm code written and tested, it is now time to test the motors.

## Testing and controlling the motors

To encapsulate the motor control functionality, we create a class called `Wheel` inside a file called `wheel.py`.

To do this, we do the following:

1. Our code requires the `PicoRobotics.py` library that may be found in this chapter's GitHub repository under `code | PicoH`. To download the library to our Pico H using Thonny, we first find the `lib` directory on our computer. We then right-click on the `lib` directory and select **Upload to /:**

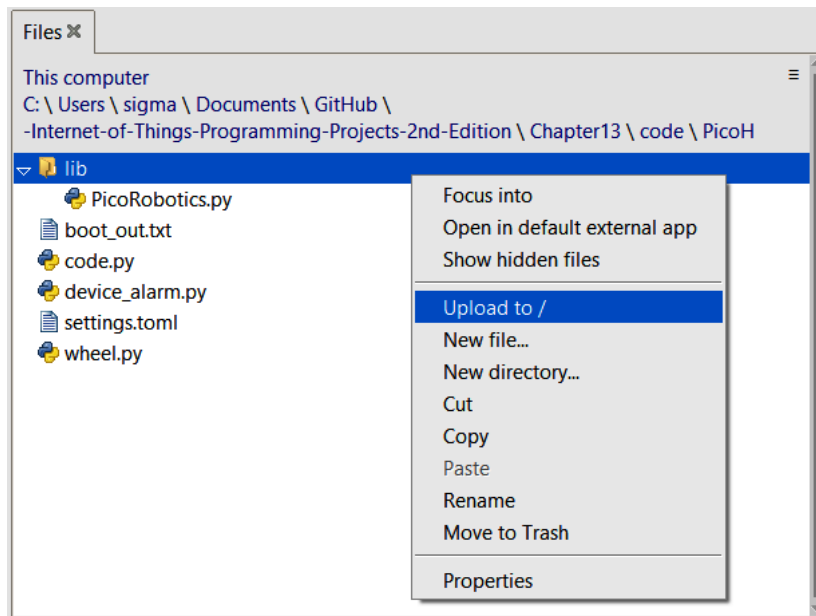


Figure 13.19 – Uploading the lib directory to our Pico H from our computer

2. We open a new editor in Thonny and start our coding by importing the `KitronikPicoRobotics` and `time` libraries into our program:

```
from PicoRobotics import KitronikPicoRobotics
import time
```

3. These libraries will allow us to interface with the Pico robotics board. We then define our class and methods:

```
class Wheel:
 def __init__(self, speed):
```

```
 self.motor_board = KitronikPicoRobotics()
 self.speed = speed

 def forward(self):
 self.motor_board.motorOn(1, "f", self.speed)
 self.motor_board.motorOn(2, "f", self.speed)

 def reverse(self):
 self.motor_board.motorOn(1, "r", self.speed)
 self.motor_board.motorOn(2, "r", self.speed)

 def turn_right(self):
 self.motor_board.motorOn(1, "r", self.speed)
 self.motor_board.motorOn(2, "f", self.speed)

 def turn_left(self):
 self.motor_board.motorOn(1, "f", self.speed)
 self.motor_board.motorOn(2, "r", self.speed)

 def stop(self):
 self.motor_board.motorOff(1)
 self.motor_board.motorOff(2)
```

In our code, the following happens:

- I.     We define a `Wheel` class.
- II.    The `__init__()` constructor initializes the class, setting instance variables including `motor_board`, which encapsulates the functionality of the Pico robotics motor board, and the `speed` parameter to control motor speed.
- III.   We implement a `forward()` method to move both wheels forward at the specified speed.
- IV.    We implement a `reverse()` method to move both wheels in reverse at the specified speed.
- V.     We then implement a `turn_right()` method to rotate the robot right by running the left wheel forward and the right wheel in reverse at the specified speed.
- VI.    We implement a `turn_left()` method to rotate the robot left by running the right wheel in reverse and the left wheel forward at the specified speed.
- VII.   We then implement a `stop()` method to stop both motors, halting the robot's movement.

4. To test our code and wiring, we use the following code:

```
#Test code
wheel = Wheel()
wheel.forward()
time.sleep(1)
wheel.reverse()
time.sleep(1)
wheel.turn_right()
time.sleep(1)
wheel.turn_left()
time.sleep(1)
wheel.stop()
```

5. Before we run the code, we must ensure that the power switch on the motor board is turned on and the AA battery pack is connected:

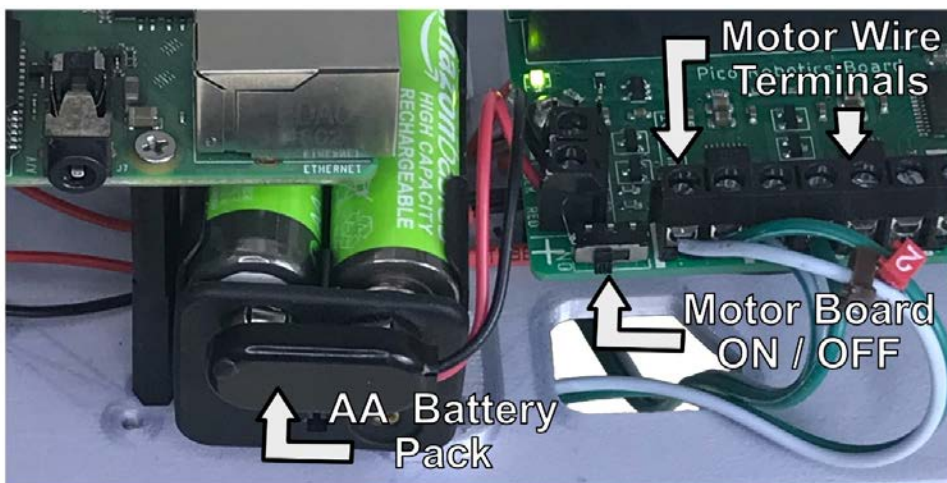


Figure 13.20 – Closeup of the motor board and AA battery pack

6. For testing purposes, we place A.R.E.S. on the testing stand to keep its wheels off the ground (*Figure 13.17*).
7. To save the file, we click on **File | Save as...** from the drop-down menu. We save our file as `wheel.py` to our Raspberry Pi Pico H.
8. To run our code, we click on the green **Run** button, hit *F5* on the keyboard, or click on the **Run** menu option at the top, and then **Run current script**.



We should observe the wheels on A.R.E.S. go through a sequence of moving forward, moving in reverse, moving right, and moving left, before stopping.

**Tip**

To avoid having our test code run outside of our test, we comment or delete it and save `wheel .py` onto our Pico H again.

At this stage, we ensure the wheels move in the desired direction by adjusting the motor wiring on the motor board, which may involve rearranging the wires at the motor wire terminals.

Having correctly configured and tested the LEDs, buzzer, and motors connected to the Pico H, we will now proceed to test communications between the Raspberry Pi Pico H and the Raspberry Pi 3B+ onboard A.R.E.S.

## Testing communication between Pi and Pico

In *Figure 13.1*, we observe that communication between the Raspberry Pi 3B+ and the Raspberry Pi Pico H on A.R.E.S. is done through UART. Specifically, messages are sent from the Raspberry Pi 3B+ to the Pico H to control the LEDs, buzzer, and motors connected to the Pico H. We wired up the two devices through their respective GPIO ports when we built A.R.E.S.

In this section, we will test communication using a Python test script located in this chapter's GitHub repository and a new file we will create on our Pico H. We will start with our Pico H.

### *Creating the Pico H script*

To create the code on our Pico H that will await commands from the Raspberry Pi 3B+, we do the following:

1. We open a new editor in Thonny and start our coding by importing the libraries we need for our program:

```
import board
import busio
import time
from wheel import Wheel
from device_alarm import Alarm
```

In our code, we do the following:

- I. We start by importing the `board` module for accessing physical pin definitions.
- II. We then import the `busio` module for bus communication (UART) functionalities.
- III. We use the `time` module for performing delays.

IV. We import our `Wheel` class from our `wheel` module.

V. We then import the `Alarm` class from the `device_alarm` module we created.

2. With our imports in place, we set our variable declarations:

```
wheel = Wheel(20)
alarm = Alarm()
uart = busio.UART(board.GP4, board.GP5, baudrate=115200)
```

In our code, we have the following:

- `wheel = Wheel(20)`: Creates an instance of the `Wheel` class with a speed parameter set to 20.
- `alarm = Alarm()`: Initializes an instance of the `Alarm` class.
- `uart = busio.UART(board.GP4, board.GP5, baudrate=115200)`: Establishes a UART communication link using pins GP4 and GP5 on the Pico H, setting the baud rate to 115200.

3. We then create a function to clear out our UART buffer by continuously reading it until no data remains (to ensure accurate and current data communication by removing old or irrelevant data that could lead to errors):

```
def clear_uart_buffer():
 while uart.in_waiting > 0:
 uart.read(uart.in_waiting)
```

4. Our code then runs in a continuous loop waiting for messages over UART:

```
while True:
 data = uart.read(uart.in_waiting or 32)

 while '<' in message_buffer and '>' in message_buffer:
 start_index = message_buffer.find('<') + 1
 end_index = message_buffer.find('>', start_index)
 message = message_buffer[start_index:end_index].strip()
 message_buffer = message_buffer[end_index+1:]
 print("Received:", message)

 if message == 'f':
 print("Moving forward")
 wheel.forward()
 elif message == 'b':
 print("Moving in reverse")
 wheel.reverse()
```

```
elif message == 'l':
 print("Left turn")
 wheel.turn_left()
elif message == 'r':
 print("Right turn")
 wheel.turn_right()
elif message == 'a':
 print("Alarm")
 wheel.stop()
 alarm.activate_alarm(2)
elif message == 's':
 print("Stop")
 wheel.stop()
```

In our code, the following happens:

- I. We continuously read up to 32 bytes of data from the UART connection.
  - II. We strip away the enclosing angle brackets.
  - III. We print out the received message for debugging purposes.
  - IV. We use an `if` statement to execute a specific action based on the message content:
    - The robot moves forward if the message is `'f'`.
    - The robot moves in reverse if the message is `'b'`.
    - The robot turns left if the message is `'l'`.
    - The robot turns right if the message is `'r'`.
    - The robot activates an alarm and stops movement if the message is `'a'`.
    - The robot stops any movement if the message doesn't match any of the specified commands.
  - V. Our code then clears the UART buffer to remove any remaining data.
  - VI. We then introduce a brief delay of 0.1 seconds to prevent overwhelming the CPU.
5. To save the file, we click on **File | Save as...** from the drop-down menu. We save our file as `code.py` to our Raspberry Pi Pico H.
  6. To run our code, we click on the green **Run** button, hit `F5` on the keyboard or click on the **Run** menu option at the top, and then **Run current script**.

After executing `code.py`, it is expected that our code will not produce any output in the Shell, indicating it is in a state of waiting for communication from the Raspberry Pi. With the Pico H set up and waiting for messages it is now time to execute our test script from the Raspberry Pi.

## Running UART test code from the Raspberry Pi

Inside this chapter's GitHub repository, we have a file named `uart-test.py` that we may use to test the connection between the Raspberry Pi and Pico H on our A.R.E.S. robot. In this section, we will SSH into our Raspberry Pi from a Windows computer using PuTTY and run the test, all the while keeping our Pico connected through Thonny.

To do this, we do the following:

1. Using a program such as PuTTY on Windows, or a Terminal on a Linux-based system, we log in to our Raspberry Pi 3B+ using the IP address (as the hostname) we obtained after running the setup script:

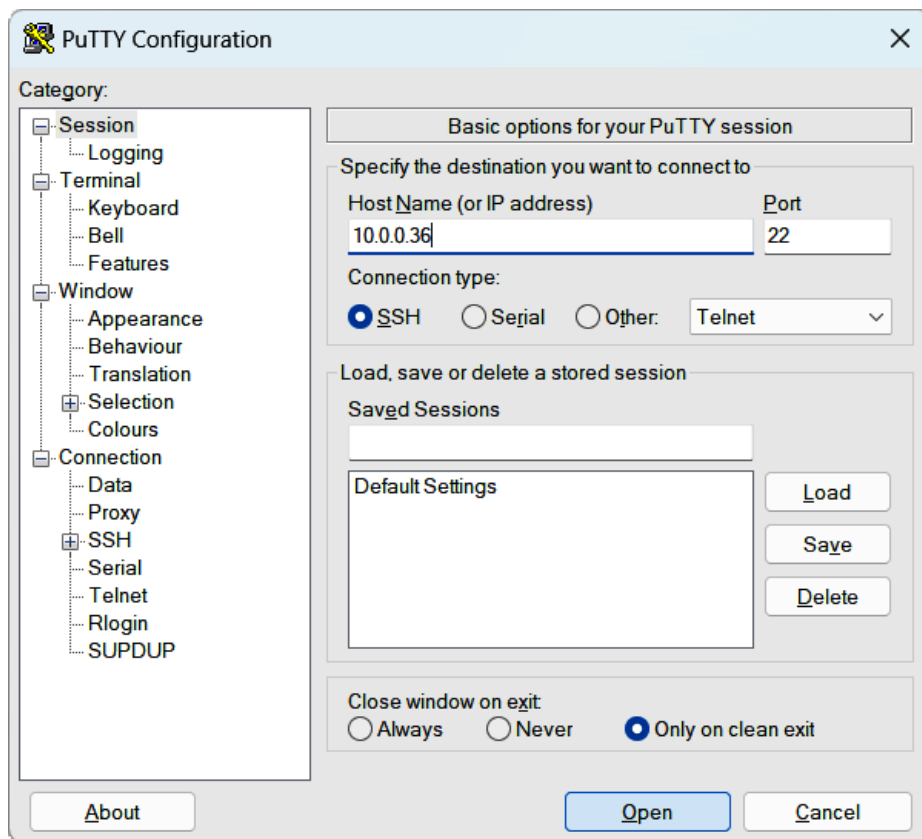


Figure 13.21 – Using PuTTY in Windows to log in to our Raspberry Pi 3B+

2. If this is the first time logging in through PuTTY, we may get a security alert. We click **Accept** to proceed.
3. To copy our test program to the current directory, we run the following command (we must not forget the dot):

```
cp code/Chapter13/code/uart-test.py .
```

4. If we wish to view the script before running it, we may do so using `vi` with the following command:

```
vi uart-test.py
```

Running the command will produce the following output:

A screenshot of a terminal window titled 'ares@ares: ~'. The window displays the contents of a file named 'uart-test.py' in a vi editor. The code is as follows:

```
import serial
import time

ser = serial.Serial('/dev/serial0', 115200, timeout=1)

def send_message(message):
 ser.write((message + "\n").encode())

send_message("a")
```

The bottom right of the window shows the cursor position '9,15' and the word 'All'.

Figure 13.22 – Viewing `uart-test.py` in the `vi` editor

5. We close the editor with the following command:

```
:q
```

6. To address the lack of serial port access under the default username, the test must be conducted with admin privileges. With the Pico H connected to Thonny and `code.py` executing, the following command initiates the test:

```
sudo python3 uart-test.py
```

7. We should observe the LEDs and buzzer activating in two pulses, with Thonny's output confirming receipt of an alarm message:

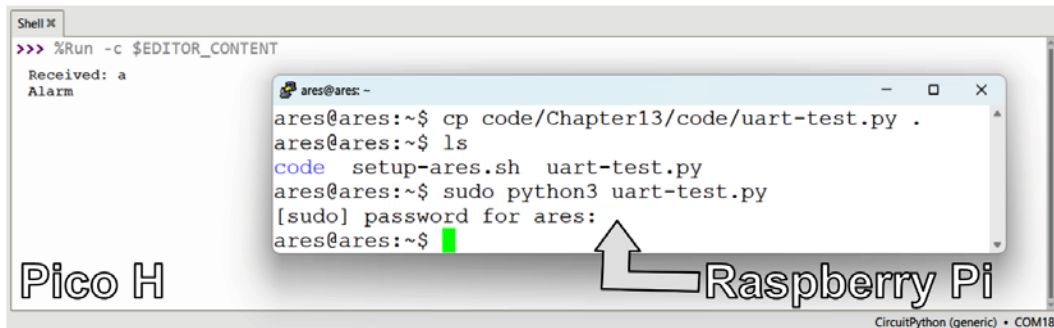


Figure 13.23 – Results from running the uart-test.py test script

With the successful testing of the UART connection between the Raspberry Pi and Raspberry Pi Pico H on our A.R.E.S. robot, it is now time to test the distance sensor.

## Testing the ToF sensor

To measure distances in front of A.R.E.S., we will use the VL53L0X ToF sensor from Adafruit. The sensor is capable of measuring distances from 30 mm to 1.2 meters with high accuracy, using a tiny laser to detect light's travel time. Its narrow beam overcomes the limitations of sonar or **infrared (IR)** sensors, making it ideal for precision tasks in robotics and interactive projects. Compatible with 3-5V and I2C communication, it's designed for easy use with various microcontrollers.

For A.R.E.S., we have the VL53L0X connected to our Raspberry Pi 3B+. We will use it in our design to stop our robot from moving forward once it detects an object less than 10 cm away.

To test the sensor, we run a test script available in this chapter's GitHub repository. To run the test, we do the following:

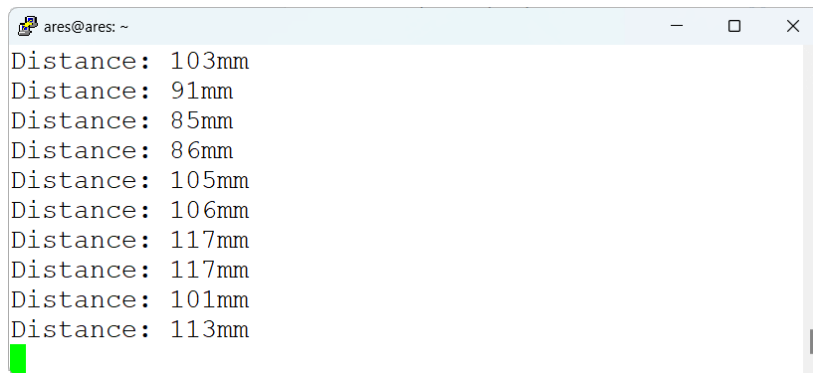
1. Using a program such as PuTTY on Windows, or a Terminal on a Linux-based system, we log in to our Raspberry Pi 3B+ using the IP address we obtained after running the setup script.
2. To copy our test program to the current directory, we run the following command (we must not forget the dot):

```
cp code/Chapter13/code/distance-sensor-test.py .
```

3. To run the test, we execute the following command:

```
python3 distance-sensor-test.py
```

4. By positioning a hand at varying distances from the sensor, which is integrated into A.R.E.S.'s mouth, we should observe corresponding variations in the sensor's output values displayed in the terminal:

A terminal window titled 'ares@ares: ~' showing the output of the 'distance-sensor-test.py' script. The output consists of ten lines, each starting with 'Distance:' followed by a value in millimeters. The values are: 103mm, 91mm, 85mm, 86mm, 105mm, 106mm, 117mm, 117mm, 101mm, and 113mm. A green cursor is visible at the end of the last line.

```
ares@ares: ~
Distance: 103mm
Distance: 91mm
Distance: 85mm
Distance: 86mm
Distance: 105mm
Distance: 106mm
Distance: 117mm
Distance: 117mm
Distance: 101mm
Distance: 113mm
```

Figure 13.24 – Output from the testing of the VL53L0X ToF sensor

#### Are ToF sensors the same as distance sensors?

A ToF sensor, measuring the time for light to bounce back from an object, provides precise distance readings. In contrast, traditional distance sensors, often using ultrasonic or IR technology, gauge distances based on sound waves or light intensity. ToF sensors typically offer higher accuracy and reliability across various ranges compared to these common distance sensors.

With the ToF sensor operational, we're set to configure A.R.E.S.'s camera, which, unlike the Raspberry Pi and Pico H, streams video outside the ROS environment, accessible by any network device.

We will use the Arduino IDE to program the camera.

## Streaming video from A.R.E.S.

For video streaming, we'll use the M5Stack Timer Camera X, powered by an ESP32 chip with a 3-million-pixel (ov3660) sensor for up to 2048x1536 pixel images. Although it supports I2C for configuration, we'll directly power it with the Raspberry Pi 3B+'s 5V supply, bypassing the I2C setup. The camera serves as the nose of our A.R.E.S. robot.

We will use the Arduino IDE and a program provided by M5Stack to set up the camera. To do so, we do the following:

1. Using a web browser, navigate to the Arduino website and download the latest Arduino IDE from <https://www.arduino.cc/en/software>.
2. Once downloaded, we install the Arduino IDE and open it.
3. To add our M5Stack Timer Camera X library and example code to the Arduino IDE, we select **File | Preferences** (in Windows) and add the following URL to the **Additional boards manager URLs** box: [https://m5stack.oss-cn-shenzhen.aliyuncs.com/resource/arduino/package\\_m5stack\\_index.json](https://m5stack.oss-cn-shenzhen.aliyuncs.com/resource/arduino/package_m5stack_index.json).
4. The dialog box should look like the following:

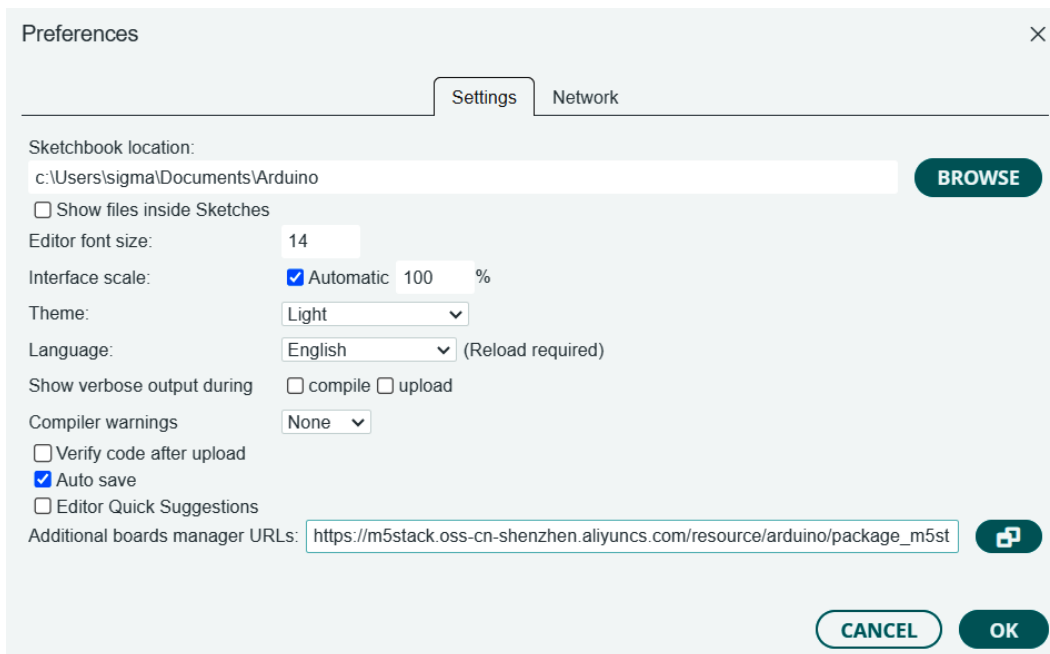


Figure 13.25 – Adding M5Stack boards to the Arduino IDE

5. We click **OK** to close the dialog.
6. Using a USB-C cable, we connect our Timer Camera X to the computer running the Arduino IDE. We may remove the camera from the face of A.R.E.S. to make it easier to access the USB-C port.



7. To set **M5TimerCAM** as the device, we click on **Tools | Board | M5Stack | M5TimerCAM**:

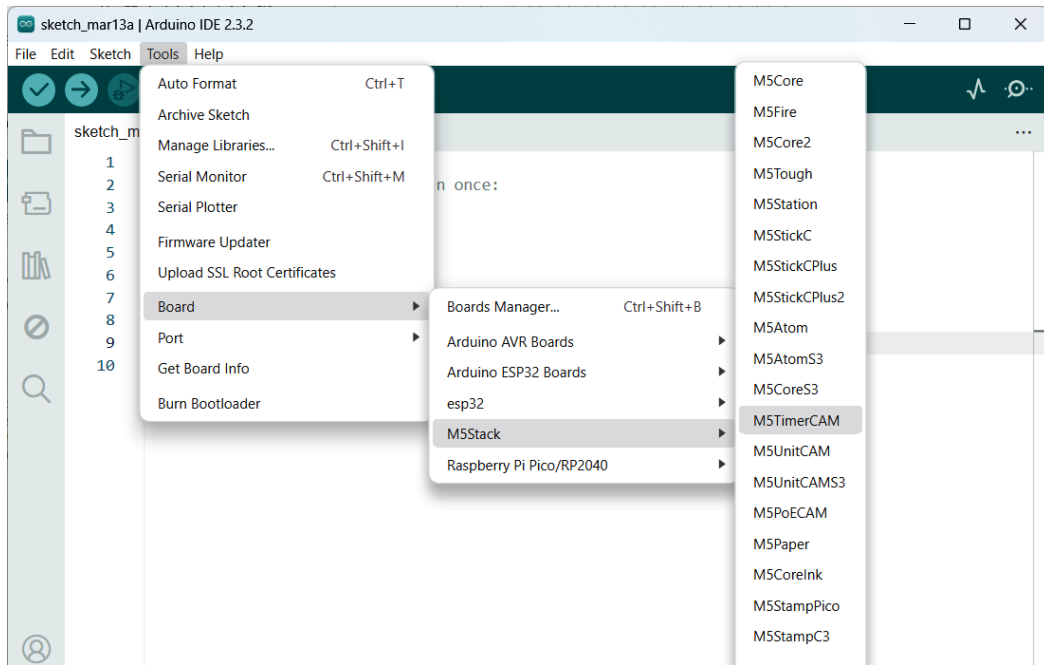


Figure 13.26 – Selecting M5TimerCAM

8. Next, we need to select the port our camera is connected to. To do this, we click on **Tools | Port** and select the port our camera is connected to (it helps to only have the camera connected to our computer as there will be only one option).
9. To access the M5Stack Timer Camera example code, we click on **Tools | Manage Libraries...** and search for **Timer-CAM**.
10. We then hover our mouse beside the title of the section until three dots appear and select **Examples | rtsp\_stream**:

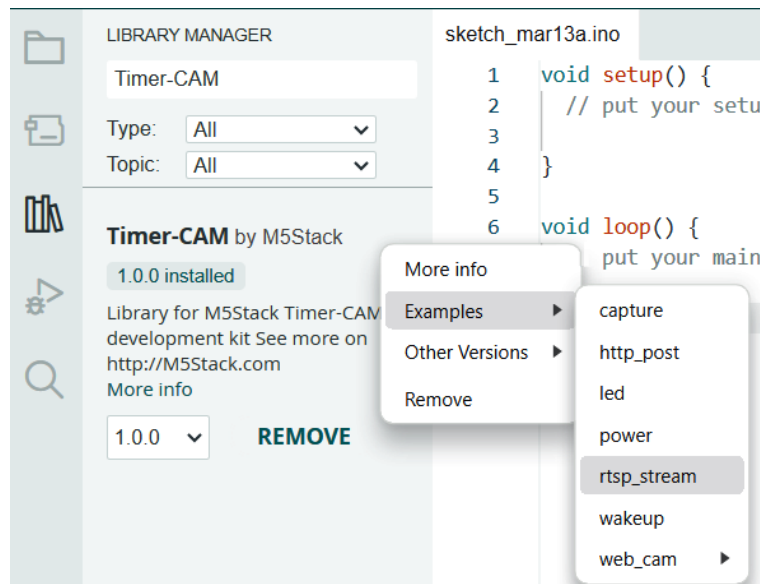


Figure 13.27 – Selecting the rtsp\_stream example code

11. This will open another Arduino window with the example code.
12. We need the Serial Monitor to find the address where the video will be broadcast. To load the Serial Monitor, we click on **Tools | Serial Monitor**:

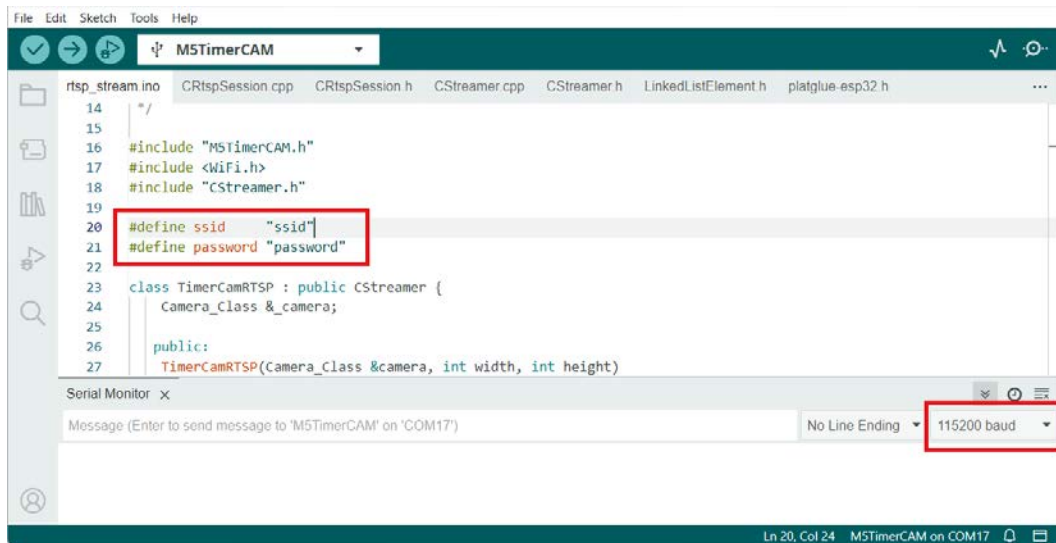


Figure 13.28 – Viewing the Serial Monitor

13. We set the baud rate to 115200 and enter the SSID name and password into the code (areas annotated in the previous figure).
14. To upload the code to our camera, we click on the **Upload** button, which looks like this: 
15. After compilation, the code is uploaded to our camera. We may view the `rtsp` address in the Serial Monitor:

```
.....
Creating TSP streamer

Connecting to Apollo
....
Connected to Apollo
IP address: 10.0.0.98
RTSP URL: rtsp://10.0.0.98:8554/mjpeg/1
```

Figure 13.29 – Output to the Serial Monitor

16. We copy the `rtsp` URL and paste it into a VLC media player by clicking on **Media | Open Network Stream...** in the VLC media player:

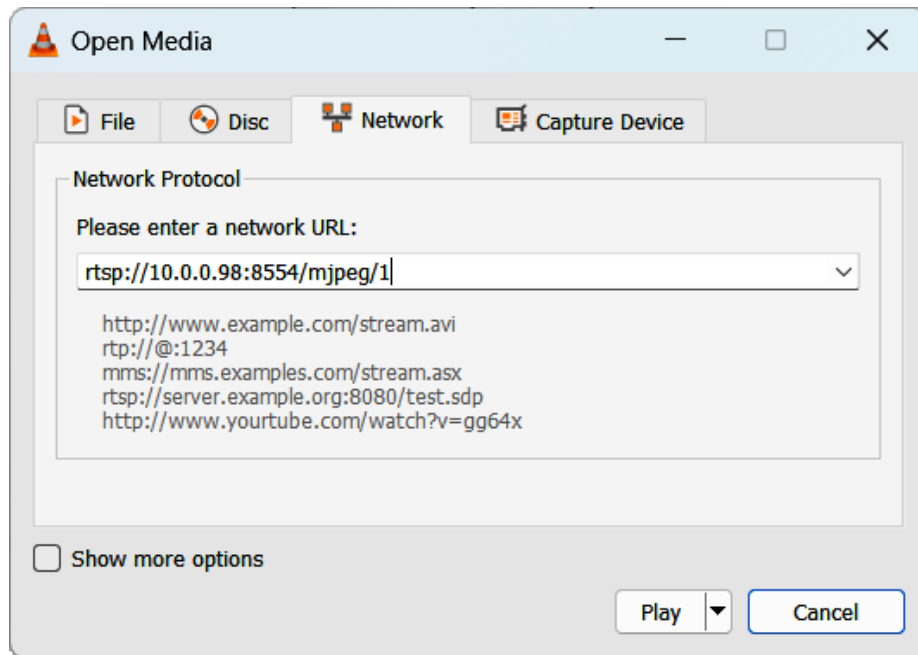


Figure 13.30 – Video streaming using the VLC media player

17. To begin streaming, we click on the **Play** button.
18. We should observe a feed from our camera:

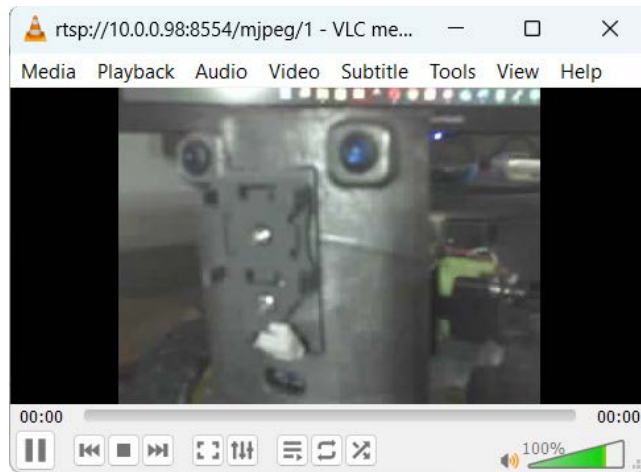


Figure 13.31 – Video feed from our camera as shown in the VLC media player

A.R.E.S. is currently missing its nose, but after testing the camera, we can disconnect the USB-C cable and reattach it to A.R.E.S.'s face. This completes the testing phase for A.R.E.S.'s components, paving the way for us to develop a ROS node and enable control of A.R.E.S. over the internet.

## Programming A.R.E.S. with ROS

Now that A.R.E.S. has been assembled and the essential software and firmware have been installed, we are ready to employ ROS for remote control over the internet. During our setup, we installed ROS along with all the necessary libraries. Along with our setup script, we downloaded test scripts from our GitHub repository, which we ran to ensure everything was functioning correctly.

A pre-existing ROS workspace also exists in our GitHub repository. To create a ROS node with this code, simply transfer the pre-existing workspace to our home directory and execute a `colcon build` command.

To do this, we do the following:

1. Using a program such as PuTTY on Windows, or a Terminal on a Linux-based system, we log in to our Raspberry Pi 3B+ using the IP address we obtained after running the setup script.
2. To copy our ROS workspace to the current directory, we run the following command (we must not forget the dot):

```
cp -r code/Chapter13/code/ares_ws .
```

3. We then navigate into our workspace with the following command:

```
cd ares_ws
```

4. We source the ROS environment with the following command:

```
source /opt/ros/humble/setup.bash
```

5. Before we build our ROS node, let's look at the code we will be building. To use nano to view our code, we type the following:

```
nano src/ares/ares/robot_control.py
```

6. As our code is very similar to the `robot_control.py` file from *Chapter 12*, we will explore only certain parts of the code. We start with code located inside the initialization method we use to give us the permissions to use the `serial0` port:

```
password = 'sudo-password'
command = 'chmod a+rw /dev/serial0'
subprocess.run(f'echo {password} | sudo -S {
command}', shell=True, check=True)
```

#### Important note

It is highly advisable not to put the admin password inside a file due to security concerns. However, with our application operating in a strictly controlled development environment where access is tightly restricted, we bypass this guideline. We require this password so that we may change permissions on the `serial0` port. Without it, we would not have access and thus could not send commands to the Pico H.

In our code, the following happens:

- I. We store a sudo user's password in the `password` variable. We set the command to change permissions of `/dev/serial0` to read and write for all users in the `command` variable.
  - II. Our code then executes the command using `sudo` without manual password entry by piping the password into `sudo -S`, utilizing `subprocess.run` with shell execution enabled and enforcing command success with `check=True`.
7. Inside the initialization method, we also set the `ser` instance variable to be equal to `serial0`, the port we connect our Pico H to:

```
self.ser = serial.Serial('/dev/serial0', 115200, timeout=1)
```

8. Our `send_message()` method formats a command to sit within open (<) and closed (>) angle brackets and sends out messages over the serial port:

```
def send_message(self, command):
 if command.strip() == 's' and
self.last_command_sent == 's':
 print("Skip sending 's' command
to avoid sending it two times in a row")
 return

 framed_command = f"<{command}>\n"
 print(f"Sending framed command:
 {framed_command.strip()}")
 self.ser.write(framed_command.encode
 self.get_logger().info(f"Sent command: {command.
strip()}")

 self.last_command_sent = command.strip()
```

In our code, the following happens:

- I. We check if the current command is 's' and if the last sent command was also 's' to prevent sending 's' consecutively. This is done as the stop command is the default command when there is no engagement with the IoT joystick and thus can flood the communication channel with redundant signals, potentially causing unnecessary processing and response delays in the system.
- II. If the preceding condition is met, our code skips sending the command and logs a message about it.
- III. We then format command with opening (<) and closing (>) angle brackets, followed by a newline.
- IV. We log the framed command being sent. Our code then sends the framed command over the serial port using `.encode()` to convert it to bytes.
- V. We log the original command (stripped of whitespace) as sent. Our code then updates `self.last_command_sent` with the current command (stripped of whitespace) for future checks.

To build our code, we execute the `colcon` command:

```
colcon build
```

9. After building our node, we source it with the following command:

```
source install/setup.bash
```

10. We are now ready to run our node to have our robot controlled by our IoT Joystick. We do so with the following command:

```
ros2 run ares robot_control
```

**Tip**

It is advisable to have A.R.E.S. up on the test stand before we send it commands. With our node running, we may control A.R.E.S. using the IoT joystick we built in *Chapter 12*.

We have just controlled a robot over the internet using MQTT and ROS. Controlling a robot over the internet using MQTT and ROS not only demonstrates the technical feasibility of remote robotic operations but also highlights the potential for use cases where remote monitoring and intervention are critical, such as in **disaster recovery (DR)**, hazardous environment exploration, and healthcare support.

## Summary

In this chapter, we integrated MQTT and ROS and created the A.R.E.S. robot. Using MQTT, a lightweight messaging protocol, enabled us to have efficient and reliable communication between the robot and our IoT joystick. ROS offers us a robust framework for developing complex robotic applications. By choosing ROS to build A.R.E.S., we leverage its vast ecosystem of tools and libraries, ensuring our robot is not only capable of performing advanced tasks but is also scalable and adaptable for future enhancements.

As we built and programmed A.R.E.S., we could easily imagine using our knowledge to build more advanced robots capable of performing complex tasks, interacting seamlessly with humans, and adapting to various environments and challenges autonomously.

In our next and final chapter, we will add vision recognition to A.R.E.S.

# 14

## Adding Computer Vision to A.R.E.S.

In this final chapter, we will be adding computer vision to A.R.E.S. This will give A.R.E.S. the ability to recognize objects as well as alert us via text message to the presence of these objects. For our example, we will be recognizing dogs, although we could just as easily set up our object recognition code to recognize other objects.

We will start our journey by exploring **computer vision** and what it is before downloading and installing the **Open Source Computer Vision (OpenCV)** library and the **You Only Look Once (YOLO)** object detection system. After setting up these tools, we will explore hands-on examples.

By the end of this chapter, you will have built a smart video streaming application that utilizes the video stream from the camera on A.R.E.S.

We will cover the following topics in this chapter:

- Exploring computer vision
- Adding computer vision to A.R.E.S.
- Sending out a text alert

Let's begin!

### Technical requirements

You will need the following to complete this chapter:

- Intermediate knowledge of Python programming
- The A.R.E.S. robot from *Chapter 13*
- A computer with a GUI-style operating system, such as the Raspberry Pi 5, macOS, or Windows



The code for this chapter can be found here: <https://github.com/PacktPublishing/Internet-of-Things-Programming-Projects-2nd-Edition/tree/main/Chapter14>.

## Exploring computer vision

Computer vision began in the 1950s, evolving significantly with key milestones such as image processing algorithms in the 1960s and the introduction of GPUs in the 1990s. These advancements improved processing speeds and complex computations and enabled real-time image analysis and sophisticated models. Modern computer vision technology is a result of these developments. The following diagram shows where computer vision sits in terms of artificial intelligence:

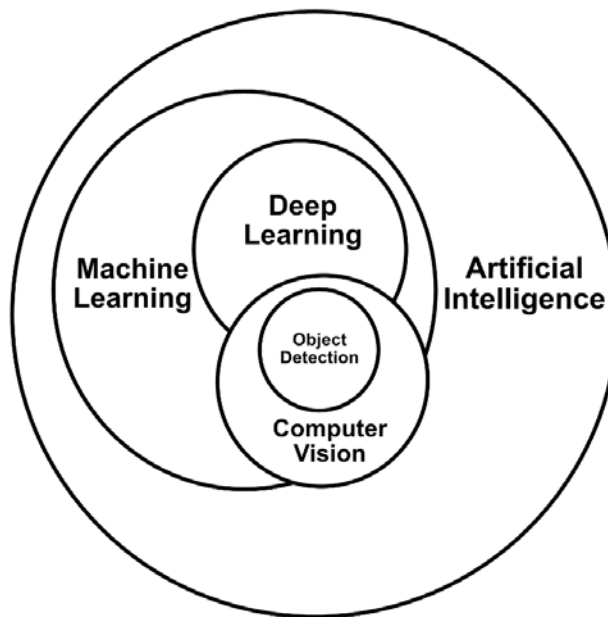


Figure 14.1 – Artificial intelligence

*Figure 14.1* shows a set of concentric circles representing the relationship between different fields of artificial intelligence. At the core is computer vision, surrounded by **object detection**, which is a subset of deep learning, something that's nested within machine learning. All are encompassed by the broader field of artificial intelligence.

Not all computer vision techniques involve machine learning or deep learning, but object detection, a part of computer vision, often does.

### What is the difference between image recognition, object recognition, object detection, and image classification?

In computer vision, terms such as **image recognition**, **object recognition**, **object detection**, and **image classification** describe specific processes. Image recognition detects features or patterns within an image. Object recognition moves beyond this to identify distinct objects within an image or video, although it does not specify their precise locations, concentrating on identifying *what* objects are present rather than *where* they are. Object detection, conversely, not only identifies objects but also locates them spatially, often using bounding boxes. Meanwhile, image classification involves analyzing an entire image to assign it to a specific category, such as determining whether an image shows a dog, cat, or car. For A.R.E.S., we want a video feed that creates a bounding box around an object that's been detected. So, we will use object detection in our application.

In this chapter, we will integrate OpenCV and the YOLO deep learning algorithm into A.R.E.S. so that we can use object detection to detect the presence of a dog.

We will start our foray into computer vision by familiarizing ourselves with the OpenCV library.

## Introducing OpenCV

OpenCV is a foundational tool in the field of computer vision that offers a vast range of capabilities for real-time image processing. OpenCV supports a multitude of applications, from simple image transformations to complex machine learning algorithms.

OpenCV not only allows for rapid prototyping but also supports full-scale application development across various operating systems, making it an excellent choice for hobbyists, educators, and commercial developers alike.

In this section, we will explore the core functionalities of OpenCV. We will start by creating a Python virtual environment and a project folder.

### Viewing an image using OpenCV

Getting started with OpenCV can be as simple as displaying an image in a window. This basic exercise introduces us to key functions for image loading, handling, and window operations in OpenCV. Follow these steps:

1. Start by opening a terminal window. We can use the Raspberry Pi operating system on our Raspberry Pi 5 or another operating system of our choice.
2. To store our project files, we must create a new directory and subdirectory (for images) with the following command (Linux commands are being used here):

```
mkdir -p Chapter14/images
```

3. Then, we navigate to the new directory:

```
cd Chapter14
```

4. We will need an image file to test OpenCV with. To download one directly from this chapter's GitHub repository to our new `images` subdirectory, we can run the following command:

```
curl -o images/toronto.png https://raw.githubusercontent.com/PacktPublishing/Internet-of-Things-Programming-Projects-2nd-Edition/main/Chapter14/images/toronto.png
```

5. Next, we need to create a new Python virtual environment for our project by running the following command (we may need to install the Python `venv` library if it is not already installed):

```
python -m venv ch14-env --system-site-packages
```

6. With our new Python virtual environment created, we can source into it with the following command:

```
source ch14-env/bin/activate
```

7. For our project, we need the `opencv-python` library. We can install this library with the following Terminal command:

```
pip install opencv-python
```

8. We close the terminal by running the following command:

```
exit
```

9. Next, we launch Thonny and source our newly created Python virtual environment:

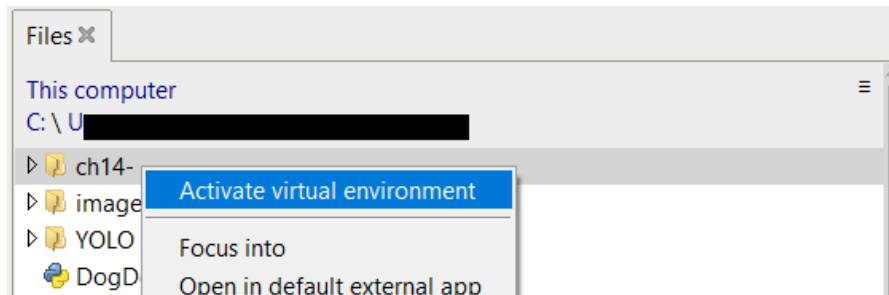


Figure 14.2 – Sourcing our Python virtual environment

10. Next, we'll create a new tab. We can do this by selecting **File** and then **New** or by hitting *Ctrl* + *N* on our keyboard.
11. Enter the following code in the editor:

```
import cv2 as cv

img = cv.imread('images/Toronto.png')
cv.imshow('Downtown Toronto', img)
cv.waitKey(0)
cv.destroyAllWindows()
```

Let's take a closer look at this code:

- I. First, we import the OpenCV library and give it an alias of `cv` for easier reference in the code.
  - II. Our code then reads the `Toronto.png` image file into the `img` variable from the `images` folder.
  - III. Next, we create a window named `Downtown Toronto` and display `img` within this window.
  - IV. Then, our code waits indefinitely for a key event before moving on to the next line of code. The `0` value means it will wait until a key is pressed.
  - V. Finally, we destroy all the windows that have been created during the session and ensure no window from the OpenCV UI remains open after the script is run. This could potentially cause a memory leak.
12. We save the code with a descriptive name such as `Toronto.py` in our `Chapter14` project folder.
  13. We run the code by clicking on the green run button, hitting *F5* on our keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.

We should see a window appear that contains an image of Toronto:



Figure 14.3 – OpenCV window popup showing downtown Toronto (image: Maximillian Dow)

14. To close the pop-up window, we hit any key on our keyboard.

Although very simple, this exercise lays the groundwork for more complex computer vision projects. Before we move on to using artificial intelligence, we will investigate using OpenCV to view the camera feed coming from A.R.E.S.

## Streaming video using OpenCV

In *Chapter 13*, we streamed a video from A.R.E.S. using the VLC media player. To utilize the video coming from A.R.E.S., we can use OpenCV for real-time image and video analysis.

To view our video feed using OpenCV, we must do the following:

1. We launch Thonny and source the `ch14-env` Python virtual environment.
2. We create a new tab by selecting **File** and then **New** or by hitting `Ctrl + N` on our keyboard.
3. We enter the following code in the editor:

```
import cv2

stream_url = '<<rtsp address>>'
cap = cv2.VideoCapture(stream_url)

if not cap.isOpened():
 print("Error: Could not open stream")
 exit()

while True:
 ret, frame = cap.read()

 if not ret:
 print("Error: Can't receive frame. Exiting ...")
 break

 cv2.imshow('A.R.E.S. Stream', frame)

 if cv2.waitKey(1) & 0xFF == ord('q'):
 break

cap.release()
cv2.destroyAllWindows()
```

Let's take a closer look at this code:

- I. We start by importing the OpenCV library.
- II. Then, we define the RTSP URL as a string for the video stream source.
- III. Our code creates a `VideoCapture` object that attempts to open the video stream from the specified RTSP URL. If the stream can't be opened, an error message is printed.
- IV. Then, we start an infinite loop to continuously fetch frames from the stream.
- V. After, we attempt to read the next frame from the stream.

- VI. We print an error message if a frame can't be received and exit the loop.
  - VII. We display the current frame in a window titled `A.R.E.S. Stream`.
  - VIII. Then, we allow the user to close the stream window manually if they press `q` on their keyboard.
  - IX. Next, release the video capture object, freeing up resources and closing the video file or capturing device.
  - X. Finally, we close all OpenCV windows, cleaning up any remaining resources associated with the window displays.
4. We save the code with a descriptive name such as `video-feed.py` in our `Chapter14` project folder.
  5. We run the code by clicking on the green run button, hitting `F5` on our keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.
  6. We should see a window appear, displaying the feed from the camera on A.R.E.S:

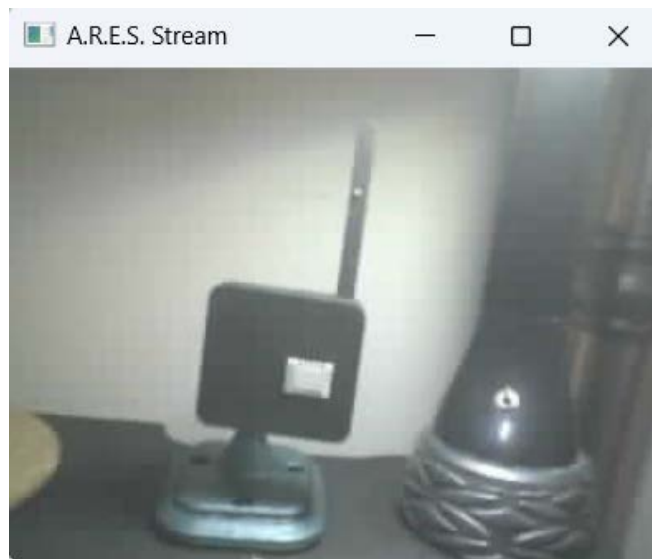


Figure 14.4 – Video feed from the camera on A.R.E.S.

7. To close the pop-up window, we hit `q` on our keyboard.

Now that we have some experience using OpenCV to view images and videos, let's take this a step further and have it identify objects, specifically dogs, as we continue our computer vision journey.

We will start by looking at neural networks and how they are used to identify objects.

## Understanding YOLO and neural networks

In this section, we'll focus on YOLO and the various layers of neural networks so that we can construct object detection code that can identify dogs. Turning our attention to *Figure 14.1*, we can see that object detection is a part of computer vision, where both deep learning and machine learning techniques are used.

### Machine learning versus deep learning

Machine learning is a subset of artificial intelligence where algorithms use statistical methods to enable machines to improve with experience, typically requiring manual feature selection. In contrast, deep learning, a specialized subset of machine learning, operates with neural networks, which automatically extract and learn features from large volumes of data. This is ideal for complex tasks such as image and speech recognition. While machine learning works with less data and provides more model transparency, deep learning requires substantial data and computational power, often acting as a *black box* with less interpretability.

To represent deep learning, YOLO uses a sophisticated neural network that assesses images in a single sweep.

## Exploring object detection

As mentioned previously, object detection is the process of finding objects in an image or video feed. The following figure illustrates the sequential stages of an object detection algorithm, using the example of an image of a dog:

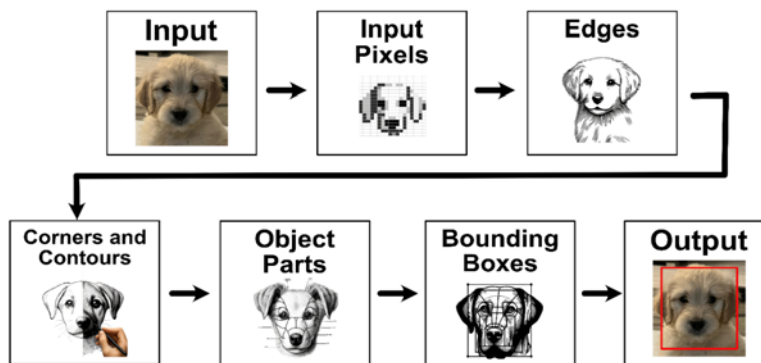


Figure 14.5 – Stages of object detection in computer vision

First, we have the original image as the input. We proceed by decomposing it into input pixels and then identify the edges, corners, and contours for structural interpretation. Our algorithm proceeds to recognize individual object parts and places bounding boxes around these components. This leads to the final output, which highlights the detected object within the image.



Now that we understand how object detection works when used with neural networks, let's consider an example. In the next subsection, we will use YOLO to identify a dog in a picture.

### ***Using YOLO to identify a dog in a picture***

In this section, we will write a program using OpenCV and the YOLO deep learning algorithm to detect a dog in a picture.

To get started, we need to download the YOLO configuration files, the pre-trained weights, and the `coco.names` file, which contains the list of classes recognized by the model. These files are typically available on the official YOLO website or reputable GitHub repositories dedicated to YOLO. The configuration file (`yolov4.cfg`) outlines the network architecture, the weights file (`yolov4.weights`) contains the trained model parameters, and the class names file lists the object categories the YOLO model can detect, all of which are crucial for the object detection task at hand.

To make this easier, we have included all the files you'll need for this exercise in this chapter's GitHub repository.

#### **What is the `yolo4.weights` file?**

The `yolov4.weights` file contains pre-trained weights for the YOLOv4 object detection model, enabling it to accurately detect and locate objects in images and videos. Since this file is too large to be included in this chapter's GitHub repository, you'll need to download it from the official YOLO website or GitHub repository (<https://github.com/AlexeyAB/darknet/releases>).

To create our object detection code, follow these steps:

1. We start by opening a terminal window. We can use the Raspberry Pi operating system on our Raspberry Pi 5 or another operating system of our choice.
2. We navigate to our `Chapter14` project directory with the following command:

```
cd Chapter14
```

3. To store our YOLO files, create a new directory with the following command:

```
mkdir YOLO
```

4. We copy the `coco.names`, `yolov4.cfg`, and `yolov4.weights` files from the YOLO directory of this chapter's GitHub repository to our local YOLO directory using whichever method suits us.
5. For our test image, we copy the `dog.png` file from the images directory of this chapter's GitHub repository to our project folder's images directory.

6. We launch Thonny and source the `ch14-env` Python virtual environment.
7. We create a new tab by selecting **File** and then **New** or by hitting `Ctrl + N` on our keyboard.
8. In the editor, we add the necessary imports for our code. Here, we'll need OpenCV as our library and NumPy for its mathematical functions:

```
import cv2
import numpy as np
```

9. Now, we load the YOLO algorithm by running the following lines of code:

```
net = cv2.dnn.readNet("YOLO/yolov4.weights", "YOLO/yolov4.cfg")
classes = []
layer_names = net.getLayerNames()
```

Let's take a closer look at this code:

- I. First, we initialize the YOLO network by loading the pre-trained weights (`yolov4.weights`) and configuration (`yolov4.cfg`). This creates a neural network ready for object detection.
  - II. Then, we create an empty list intended to store the class names (for example, dog and cat) that YOLO can detect, once they are read from a file.
  - III. Our code then retrieves the names of all the layers in the YOLO network. These layer names are used to identify output layers. This is crucial for obtaining detection results.
10. We enter the following code to fetch the indices of the final layers in the YOLO neural network. These directly output detection results using the OpenCV `getUnconnectedOutLayers()` method:

```
output_layer_indices = net.getUnconnectedOutLayers()
```

11. Next, create a list of the names of the output layers of the YOLO neural network by indexing into the list of all layer names using the indices provided by `output_layer_indices`, adjusted for zero-based indexing. This corresponds to the *Bounding Boxes* stage of the algorithm, as outlined in *Figure 14.5*:

```
output_layers = [layer_names[i - 1] for i in output_layer_indices]
```

12. Next, we read the `coco.names` file, which contains the list of object classes that the YOLO model can identify, and creates a list of class names by removing any leading or trailing whitespace from each line. The following code finds and stores the index of the "dog" class within that list, effectively preparing the program to specifically recognize and identify dogs in the images processed by the YOLO model:

```
with open("YOLO/coco.names", "r") as f:
 classes = [line.strip() for line in f.readlines()]
dog_class_id = classes.index("dog")
```

13. The following code reads the `dog.png` image from the `images` directory, scales it down to 40% of its original size to reduce computational load, and extracts its dimensions and color channel count. The resizing step is crucial because YOLO models typically expect a fixed input size, and resizing helps to match that requirement while also accelerating the detection process due to the smaller image size:

```
img = cv2.imread('images/dog.png')
img = cv2.resize(img, None, fx=0.4, fy=0.4)
height, width, channels = img.shape
```

14. Next, we must convert the resized image into a **blob** – a preprocessed image that’s compatible with the neural network – by normalizing pixel values and setting the size to 416x416 pixels, a standard input size for YOLO models. Then, we must set this blob as the input to the neural network. Finally, we must perform a forward pass through the network using the specified output layers to obtain the detection predictions. This includes class labels, confidences, and bounding box coordinates. The following snippet corresponds to the action that takes place between the *Input Pixels* and the *Bounding Boxes* stages shown in *Figure 14.5*. It processes the image through various layers to detect objects, with the final line in the snippet producing the detections that lead to the *Output* stage:

```
blob = cv2.dnn.blobFromImage(img, 0.00392, (416, 416), (0, 0, 0),
True, crop=False)
net.setInput(blob)
outs = net.forward(output_layers)
```

15. The following code analyzes the results from the neural network’s forward pass, filtering and processing detected objects for the `dog` class with a confidence level above 50%. It calculates the bounding box coordinates based on the object’s center, width, and height, then stores these coordinates along with the detection confidence and class ID in corresponding lists. This aligns with the *Bounding Boxes* stage shown in *Figure 14.5*, where the processed outputs are used to specifically locate and classify the detected objects within the image. This sets the stage for the final visual representation in the *Output* phase, where these bounding boxes are drawn to indicate where dogs are located within the image:

```
class_ids = []
confidences = []
boxes = []
for out in outs:
 for detect in out:
 scores = detect[5:]
 class_id = np.argmax(scores)
 confidence = scores[class_id]
```

```

if confidence > 0.5 and class_id == dog_class_id:
 center_x = int(detection[0] * width)
 center_y = int(detection[1] * height)
 w = int(detection[2] * width)
 h = int(detection[3] * height)

 x = int(center_x - w / 2)
 y = int(center_y - h / 2)
 boxes.append([x, y, w, h])
 confidences.append(float(confidence))
 class_ids.append(class_id)

```

16. The following code uses **non-maximum suppression (NMS)** via the `NMSBoxes` function from OpenCV to refine the detection results by reducing overlap among bounding boxes, ensuring that each detected object is represented only once. After determining the best bounding boxes based on their confidence and overlap, it iterates through these optimized boxes to visually annotate the image. It does this by drawing a rectangle for each box and labeling it with the corresponding class name. This final step marks and identifies the detected objects (dogs) in the image, aligning with the *Output* stage shown in *Figure 14.5*:

```

indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.5,
0.4)
for i in indexes.flatten():
 x, y, w, h = boxes[i]
 label = str(classes[class_ids[i]])
 cv2.rectangle(img, (x, y), (x + w, y + h), (0,
255, 0), 2)
 cv2.putText(img, label, (x, y + 30),
cv2.FONT_HERSHEY_PLAIN, 3,
(0, 255, 0), 3)

```

17. In the final section of our code, we display the processed image with detected objects marked by bounding boxes. The `cv2.imshow("Image", img)` function displays the image in a window titled "Image". The `cv2.waitKey(0)` function pauses the execution of the script, waiting indefinitely for a key press to proceed, allowing the user to view the image for as long as needed. Finally, `cv2.destroyAllWindows()` closes all OpenCV windows opened by the script, ensuring a clean exit without leaving any GUI windows open:

```

cv2.imshow("Image", img)
cv2.waitKey(0)
cv2.destroyAllWindows()

```

18. We save the code with a descriptive name, such as `recognize-dog.py`, in our Chapter14 project folder.
19. We run the code by clicking on the green run button, hitting *F5* on our keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.
20. We should observe a pop-up window appear with a bounding box around the face of the dog:

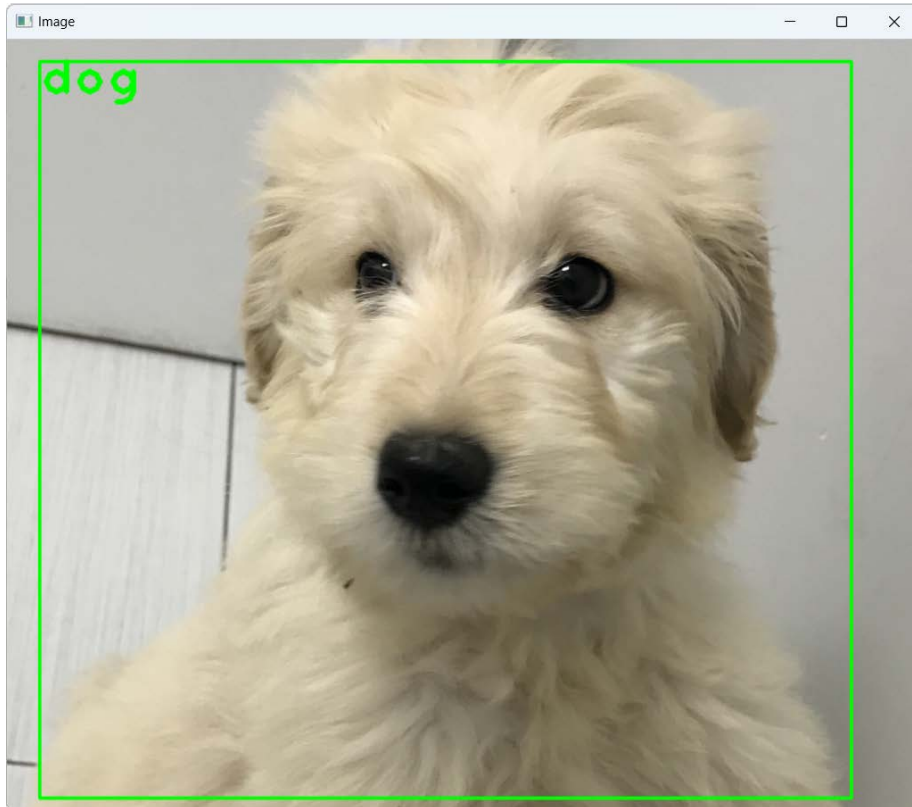


Figure 14.6 – YOLO library used to recognize a dog

21. We press any key on our keyboard to close the pop-up window.

As we can see, our program can identify a dog from a picture. If we were to provide a picture of any object identified in the `coco.names` file (a person, for example), our program should be able to identify that.

Now that we have a little exposure to YOLO, neural networks, and object detection, let's add this functionality to A.R.E.S. We will program our application to send a text message whenever A.R.E.S. detects a dog.

## Adding computer vision to A.R.E.S.

In the previous section, we explored OpenCV and YOLO, using OpenCV to view images and video feeds, and YOLO to identify a dog in a picture. In this section, we'll apply what we've learned to create a smart video streaming application that represents the eyes of A.R.E.S. We'll only use dogs as an example, but we could easily adapt this application for tracking other objects.

We will start by encapsulating our YOLO code into a class called `DogTracker` before creating a video streaming application using this class with OpenCV.

### Creating the `DogTracker` class

The `DogTracker` class embodies the artificial intelligence component of A.R.E.S. Although it could be installed directly on the Raspberry Pi 3B+ within A.R.E.S. and accessed remotely via the streaming window application, we will install it on a computer alongside our streaming application for simplicity and improved performance. In our example, we will utilize a Windows PC.

To create the `DogTracker` class, we must do the following:

1. We launch Thonny and source our `ch14-env` Python virtual environment.
2. We create a new tab by selecting **File** and then **New** or by hitting `Ctrl + N` on our keyboard.
3. Next, we add the necessary imports:

```
import cv2
import numpy as np
```

4. Then, we define our class and initialization method:

```
class DogDetector:
 def __init__(self, model_weights, model_cfg, class_file):
 self.net = cv2.dnn.readNet(model_weights, model_cfg)
 self.layer_names = self.net.getLayerNames()

 output_layer_indices = self.net.
getUnconnectedOutLayers()
 if output_layer_indices.ndim == 1:
 self.output_layers = [self.layer_names[
i - 1] for i in output_layer_indices]
 else:
 self.output_layers = [self.layer_names[
i[0] - 1] for i in output_layer_indices]

 with open(class_file, "r") as f:
```

```

 self.classes = [line.strip() for line in
f.readlines()]
 self.dog_class_id = self.classes.index("dog")

```

5. Now, we must define our only method: `detect_dogs()`. This method processes video frames to detect dogs using a YOLO neural network model. It begins by resizing the input frame for optimal processing and creates a blob from the resized image, which is then fed into the neural network. The network outputs detection results, which include bounding boxes, confidences, and class IDs for detected objects. The method checks each detection to see if it meets the confidence threshold and corresponds to the class ID for dogs. If such detections are found, it calculates and stores their bounding box coordinates. NMS is then applied to refine these bounding boxes by reducing overlaps. If any boxes remain after this process, it confirms the presence of dogs, draws these boxes on the frame, and labels them. Finally, the method returns the processed frame, along with a Boolean indicating whether any dogs were detected:

```

def detect_dogs(self, frame):
 img_resized = cv2.resize(frame, None, fx=0.4, fy=0.4)
 height, width, channels = img_resized.shape
 dog_detected = False

 blob = cv2.dnn.blobFromImage(img_resized, 0.00392,
(416, 416), (0, 0, 0), True, crop=False)
 self.net.setInput(blob)
 outs = self.net.forward(self.output_layers)

 class_ids = []
 confidences = []
 boxes = []

 for out in outs:
 for detection in out:
 scores = detection[5:]
 class_id = np.argmax(scores)
 confidence = scores[class_id]
 if confidence > 0.5 and class_id ==
self.dog_class_id:
 center_x = int(detection[0] * width)
 center_y = int(detection[1] * height)
 w = int(detection[2] * width)

```

```

 h = int(detection[3] * height)
 x = int(center_x - w / 2)
 y = int(center_y - h / 2)
 boxes.append([x, y, w, h])
 confidences.append(float(confidence))
 class_ids.append(class_id)

 indexes = cv2.dnn.NMSBoxes(boxes, confidences,
 0.5, 0.4)

 if indexes is not None and len(indexes) > 0:
 dog_detected = True
 indexes = indexes.flatten()

 for i in indexes:
 x, y, w, h = boxes[i]
 label = str(self.classes[class_ids[i]])
 cv2.rectangle(img_resized, (x, y), (x + w, y +
 h), (0, 255, 0), 2)
 cv2.putText(img_resized, label, (x, y - 5), cv2.
 FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

 return img_resized, dog_detected

```

6. We save the code with a descriptive name, such as `DogDetector.py`, in our `Chapter14` project folder.

Here, we reorganized the `recognize-dog.py` code from the previous section into a class that we will use for our smart video streamer. With this class in place, it's time to create our streaming application. We will use OpenCV for this.

## Building a smart video streamer

We will use the `detect_dogs()` method inside the `DogDetector` class to identify dogs from the video stream coming from A.R.E.S. As mentioned previously, we could easily change our code so that we can use YOLO to identify other objects. Recognizing dogs presents a fun way for those of us with dogs to program A.R.E.S. as a sort of pet detection robot. We will install our smart video streamer onto the same computer as our `DogDetector` class.



To create the smart video streamer, follow these steps:

1. We launch Thonny and source our `ch14-env` Python virtual environment.
2. We create a new tab by selecting **File** and then **New** or by hitting `Ctrl + N` on our keyboard.
3. We start by adding our imports:

```
import cv2
from DogDetector import DogDetector
import time
```

4. Then, we define our variable declarations:

```
detector = DogDetector("YOLO/yolov4.weights", "YOLO/yolov4.cfg",
 "YOLO/coco.names")
stream_url = '<<rtsp address>>'
cap = cv2.VideoCapture(stream_url)
cv2.namedWindow("Dog Detector", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Dog Detector", 800, 600)
last_time = time.time()
```

5. The bulk of our code sits inside an infinite loop. This code continuously captures frames from a video source, checking if each frame is successfully retrieved. If a second passes since the last processed frame, it detects dogs in the current frame using the `detect_dogs()` method, updates the time marker, and displays the result; if the `q` key is pressed, the loop breaks, and the video capture and any OpenCV windows are cleanly released and closed:

```
try:
 while True:
 ret, frame = cap.read()
 if not ret:
 break

 current_time = time.time()
 if current_time - last_time >= 1.0: # 1.0 seconds
 result_frame, dog_detected =
 detector.detect_dogs(frame)
 last_time = current_time

 cv2.imshow("Dog Detector", result_frame)
 if dog_detected:
 print('Dog detected!')
```

```
 if cv2.waitKey(1) & 0xFF == ord('q'):\n break\n finally:\n cap.release()\n cv2.destroyAllWindows()
```

6. We save the code with a descriptive name, such as `smart-video-feed.py`, in our Chapter14 project folder.
7. We run the code by clicking on the green run button, hitting *F5* on our keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.

We should observe a pop-up window appear with a video feed from A.R.E.S. Our application should detect the presence of a dog:

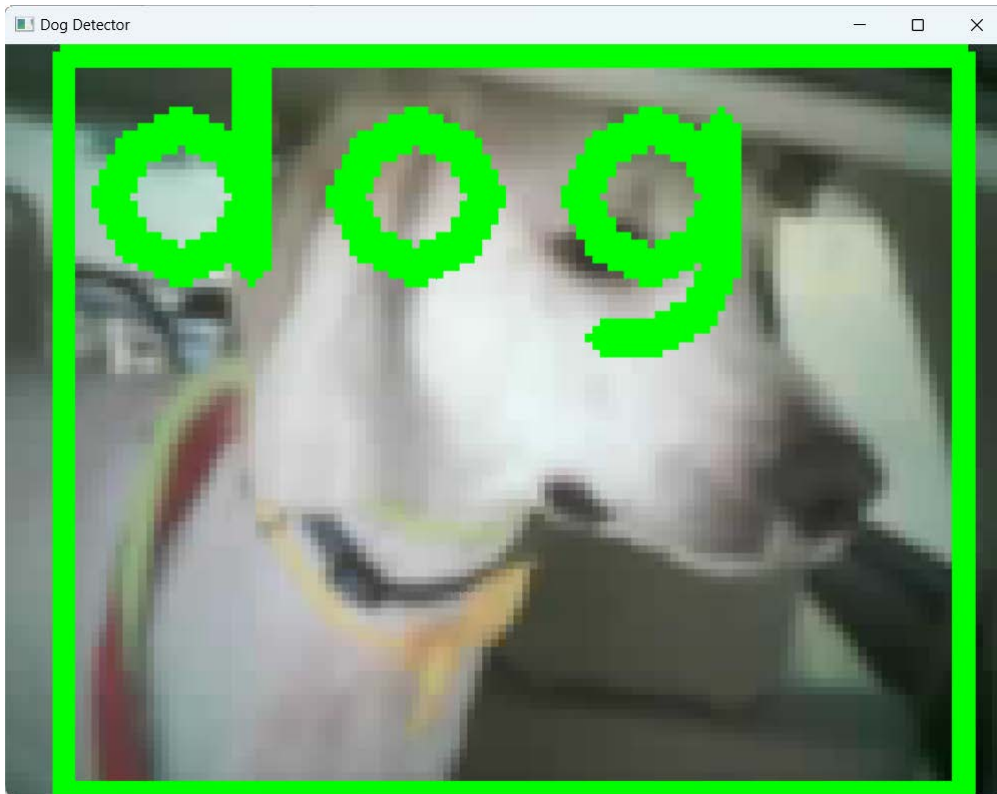


Figure 14.7 – Detecting a dog using our smart video streamer

With this, we have successfully added artificial intelligence in the form of object detection to A.R.E.S. We may adjust the size, font, and color of the frame in our `DogDetector` class. As impressive as getting object detection to work with A.R.E.S. is, we will take this a step further and introduce text notification, turning the smart video streaming functionality into a true IoT application.

## Sending out a text alert

To make A.R.E.S. a true IoT device, we will add text functionality. This will give A.R.E.S. the ability to send out text alerts when an object of interest – in our case, a dog – is detected. We will use the Twilio service for this.

We will start by setting up our Twilio account and testing the number we are assigned before we integrate text messaging functionality into A.R.E.S. We must ensure that we follow the upcoming steps carefully so that we can set up our Twilio account successfully.

## Setting up our Twilio account

Setting up a Twilio account involves registering on their website, where we'll be provided with an account SID and an auth token to authenticate API requests. Once registered, we can also obtain a Twilio phone number, which is necessary for sending SMS messages and making calls through their service. In this section, we will set up our Twilio account and send a test SMS message.

To set up our Twilio account, follow these steps:

1. Using a web browser, we navigate to `www.twilio.com` and click the blue **Start for free** button:

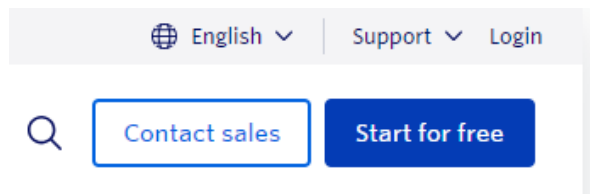
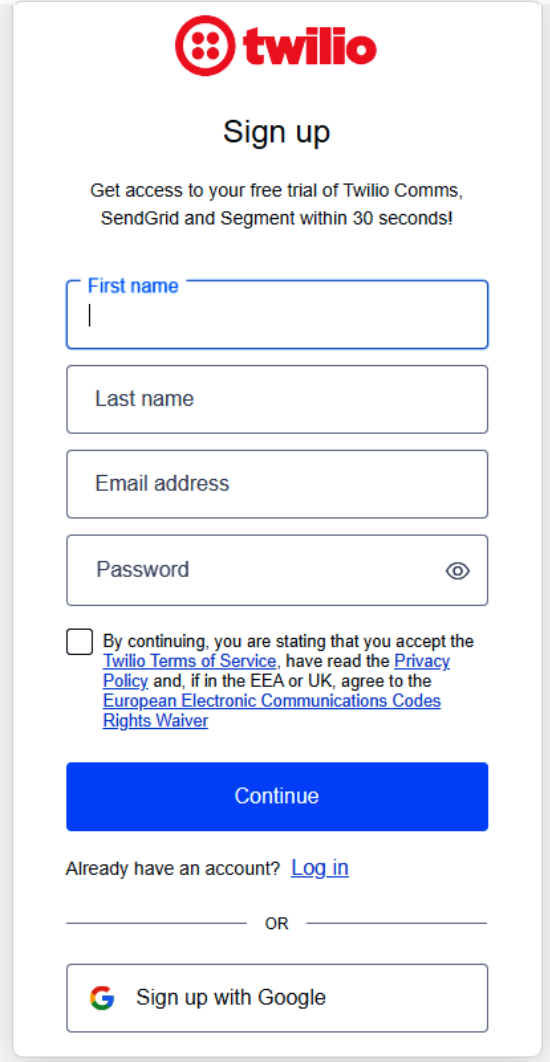


Figure 14.8 – The Twilio website

2. This will take us to the **Sign up** page. Here, we can create a Twilio account or use a Google account:



The image shows the Twilio sign-up page. At the top is the Twilio logo, which consists of a red circle with four dots inside, followed by the word "twilio" in a bold, lowercase, sans-serif font. Below the logo is the heading "Sign up" in a large, bold, sans-serif font. Underneath the heading is a sub-headline: "Get access to your free trial of Twilio Comms, SendGrid and Segment within 30 seconds!". The form contains four input fields: "First name" (with a blue border and a cursor), "Last name", "Email address", and "Password" (with a toggle icon on the right). Below the password field is a checkbox with the text: "By continuing, you are stating that you accept the [Twilio Terms of Service](#), have read the [Privacy Policy](#) and, if in the EEA or UK, agree to the [European Electronic Communications Codes Rights Waiver](#)". Below this is a large blue button labeled "Continue". Under the button is the text "Already have an account? [Log in](#)". Below this is a horizontal line with "OR" in the center. At the bottom is a button with the Google logo and the text "Sign up with Google".

Figure 14.9 – Twilio's Sign up page

- To verify our new account, we type in a phone number and click on the blue **Send code via SMS** button:



**We'll also need to verify your phone number**

- ☎ So you can hit the ground running and start using our **Twilio** services.
- ✓ To verify you during log in through **two-factor authentication (2FA)**.
- 🔒 To help us **mitigate fraud and abuse**.

---

Country Phone Number

ca (+1) Canada ✕

[Send code via SMS](#) [Send code via voice call](#)

Figure 14.10 – Verify page

- We should receive a text message containing a verification code. We enter the number and click on the blue **Verify** button:



**Check your phone for a verification code**

Twilio Verify has sent the code to:  
**+1416** XXXXXXXXXX

Enter verification code

[Verify](#) [Resend code via SMS in 7](#)

---

[Send code via voice call](#)

[Verify with another phone number](#)

Figure 14.11 – Verification code step

5. This will take us to the **You're all verified!** page, which will provide us with a **Recovery code** value. We click on the blue **Continue** button to go to the next page:

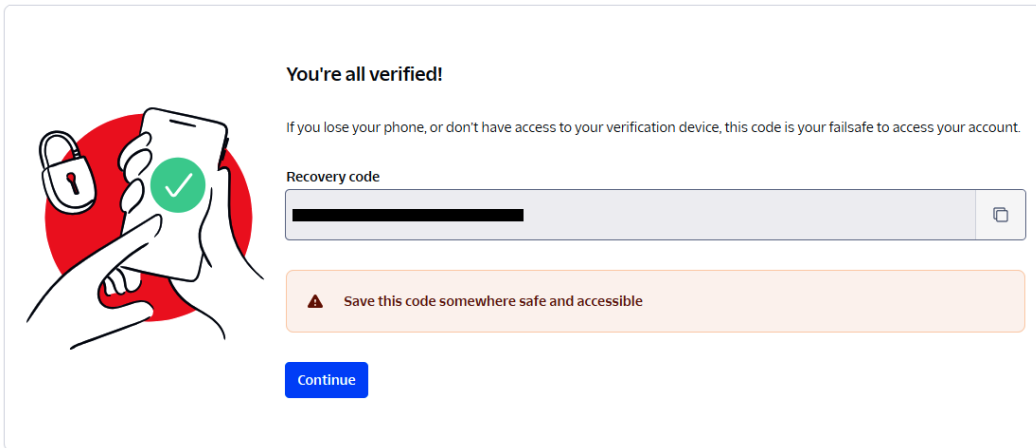


Figure 14.12 – Recovery code

6. The next page allows us to customize our Twilio experience:

## Ahoy Colin , welcome to Twilio!

Tell us a bit about yourself so we can personalize your experience. You will have access to all Twilio products.

• What do you plan to build with Twilio?

Other ▼

• Which Twilio product are you here to use?

SMS ▼

• Which best describes you/your organization?

- ☐ Business
- ☐ Nonprofit or government entity
- ☐ Sole proprietor / Self-employed
- ☒ Hobbyist or Student

• How do you want to build with Twilio?

- ☒ With code  
Customize exactly what you want
- ☐ With minimal code  
Build on top of our code samples
- ☐ With no code at all  
Launch a starter app with no code

• What is your preferred coding language?

Python ▼

• Would you like Twilio to host your code?

Host your Twilio app on our secure servers

- ☒ Yes, host my code on Twilio
- ☐ No, I want to use my own hosting service

Your billing country is Canada. [Change](#)

[Get Started with Twilio](#)

Figure 14.13 – Customizing Twilio

7. The dashboard screen allows us to get a Twilio phone number. We click on the blue **Get phone number** button to continue.
8. The next page provides us with a Twilio phone number we can use. We click on the blue **Next** button to proceed.
9. On the next screen, we can test out our new Twilio number. The **To phone number** field and the **From phone number** field should be prepopulated with the number we provided and our new Twilio number, respectively. To test out our new number, type a message in the **Body** field, such as **IoT test**, and click on the blue **Send test SMS** button:

## Step 2: Send your first SMS

### • To phone number

+1416 [REDACTED]

The phone number you verified your account with.

### • From phone number

+125 [REDACTED]

Your new Twilio phone number, provisioned in Step 1.

### • Body

IoT test

Send test SMS

Figure 14.14 – Testing our new Twilio phone number

10. We should receive a text message with our test message, **IoT test**, on the phone we provided the number to Twilio on:



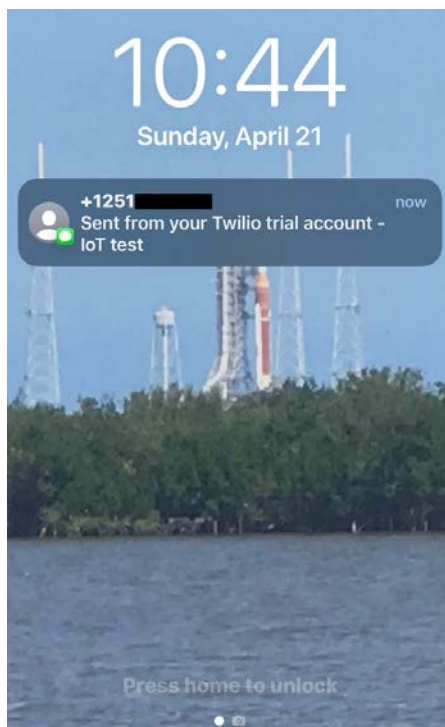


Figure 14.15 – Test message successfully received, as seen on a cell phone

With our Twilio account set up and our phone number tested, we are ready to incorporate text messaging into A.R.E.S.

### Adding text message functionality to A.R.E.S.

To integrate text messaging functionality into A.R.E.S., we will develop a new class named `TwilioMessage` and a new version of the `smart-video-feed.py` script:

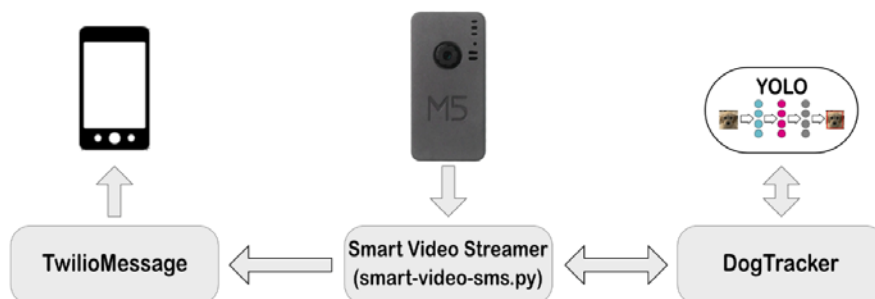


Figure 14.16 – Adding text functionality to A.R.E.S.

The `TwilioMessage` class will encapsulate communication to the Twilio server. As shown in *Figure 14.16*, our new `TwilioMessage` class is called from our smart video streamer and sends out text messages.

We will start by creating this class.

### ***Creating the `TwilioMessage` class***

To create the `TwilioMessage` class, we must do the following:

1. We launch Thonny and source our `ch14-env` Python virtual environment.
2. As we require a library from Twilio to make our code work, we will install it in our Python virtual environment. To do so, open the system shell by clicking on **Tools | Open system shell...** in Thonny:

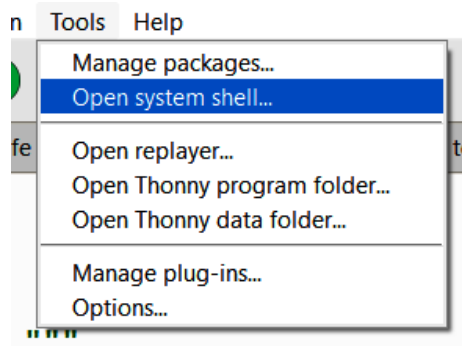


Figure 14.17 – Open system shell...

3. At the command prompt, we execute the following command:

```
pip install twilio
```

4. Once the library has been installed, we close the terminal.
5. We create a new tab by selecting **File** and then **New** or by hitting `Ctrl + N` on our keyboard.
6. We add the following code to the editor:

```
from twilio.rest import Client

class TwilioMessage:
 def __init__(self, account_sid, auth_token, from_number):
 self.client = Client(account_sid, auth_token)
 self.from_number = from_number
```

```

def send_sms(self, to_number, message):
 sms = self.client.messages.create(
 body=message,
 from_=self.from_number,
 to=to_number
)
 print(f"Message sent with SID: {sms.sid}")
if __name__ == "__main__":
 twilio_message = TwilioMessage(
 'account_sid', 'auth_token', '+twilio_number')
 twilio_message.send_sms('+our_number', 'Hello from
A.R.E.S.')

```

Let's take a closer look at our code:

- I. First, we import the Twilio client and define the `TwilioMessage` class.
  - II. Then, we initialize our class with Twilio credentials (account SID, auth token, and Twilio phone number).
  - III. The `send_sms()` method sends an SMS to a specified number and prints the message SID after sending.
  - IV. In the main execution block, an instance of `TwilioMessage` is created with Twilio credentials, and a test SMS is sent to our number.
7. We save the code with a descriptive name, such as `TwilioMessage.py`, in our `Chapter14` project folder.
  8. We run the code by clicking on the green run button, hitting *F5* on our keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.
  9. We should receive a text message saying, `Hello from A.R.E.S.` on our cell phone.

With the `TwilioMessage` class created, it is time to modify the smart video streamer code so that text messages will be sent when a dog or dogs are detected. We will do that in the next section.

### ***Modifying the smart video streamer***

The final step in providing text message functionality in A.R.E.S. is to create a new smart video streamer script to send a text message when a dog is detected. To limit the number of messages sent, we will increase the time between frames to 5 seconds.

To modify our smart video streamer, follow these steps:

1. In Thonny, we create a new tab by selecting **File** and then **New** or by hitting *Ctrl + N* on our keyboard.

2. We add the following code to the editor:

```
import cv2
from DogDetector import DogDetector
from TwilioMessage import TwilioMessage
import time

detector = DogDetector("YOLO/yolov4.weights", "YOLO/yolov4.cfg",
"YOLO/coco.names")
twilio_message = TwilioMessage('account_sid', 'auth_token',
'+twilio_number')

stream_url = '<<rtsp address>>'
cap = cv2.VideoCapture(stream_url)
cv2.namedWindow("Dog Detector", cv2.WINDOW_NORMAL)
cv2.resizeWindow("Dog Detector", 800, 600)
last_time = time.time()

try:
 while True:
 ret, frame = cap.read()
 if not ret:
 break

 # Check if 1 second has passed
 current_time = time.time()
 if current_time - last_time >= 5.0:
 result_frame, dog_detected =
detector.detect_dogs(frame)
 last_time = current_time

 cv2.imshow("Dog Detector", result_frame)
 if dog_detected:
 twilio_message.send_sms(' +phone_num', 'Dog(s)
detected!')

 if cv2.waitKey(1) & 0xFF == ord('q'):
 break
finally:
 cap.release()
 cv2.destroyAllWindows()
```

3. We save the code with a descriptive name, such as `smart-video-sms.py`, in our Chapter14 project folder.

4. We run the code by clicking on the green run button, hitting *F5* on our keyboard, or clicking on the **Run** menu option at the top and then **Run current script**.
5. We should observe a window appear with a video feed coming from A.R.E.S. that provides object detection functionality when a dog is present.
6. We should receive a text message once a dog has been detected:

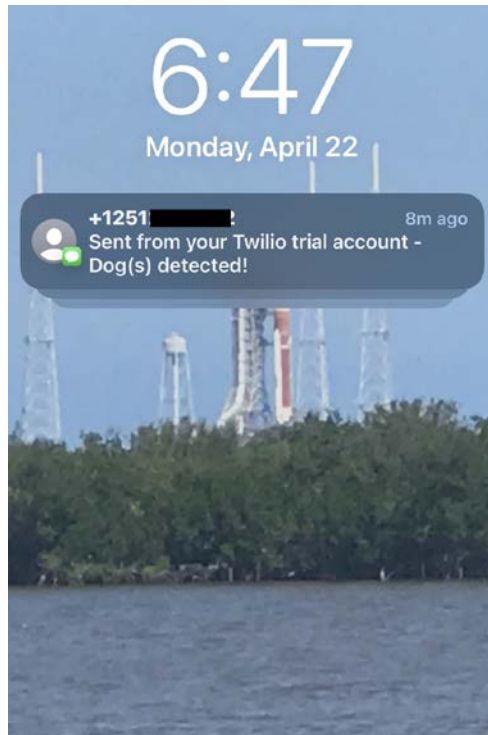


Figure 14.18 – A text message indicating that a dog or dogs were detected

With that, we have successfully added text message functionality to A.R.E.S. to alert us when an object we are interested in – in this case, a dog – is detected.

## Summary

In this chapter, we explored the field of computer vision and successfully integrated it into our A.R.E.S. robot car. By incorporating this technology, A.R.E.S. can now process and interpret visual data. We also added text messaging functionality to A.R.E.S., turning our robot car into a true IoT device.

Although not implemented, we could easily imagine how we would take A.R.E.S. to the next level by incorporating obstacle avoidance based on the data we get back from YOLO. We may also imagine how we could make A.R.E.S. follow a certain object if desired.

---

This chapter marks the end of our IoT journey together. Throughout, we have explored the world of IoT while utilizing and building our programming skills – from using the Sense HAT to serve up web services data to utilizing LoRa for long-range communication to implementing advanced features such as computer vision in a robot car we control using the internet. This adventure has not only taught us technical know-how but also showed us the potential of IoT to innovate and solve any real-world challenges in the future.

It's been a pleasure.



# Index

## Symbols

3.5-inch LCD screens 112  
7-segment display 111  
8 x 8 dot-matrix display 111  
16 x 2 LCD screens 111

## A

Acrylonitrile Butadiene Styrene (ABS) 173  
actuators 139  
Adafruit RFM95W LoRa Radio  
  Transceiver Breakout 246  
Advanced Robotic Eyes for  
  Security (A.R.E.S.) 18  
  3D-printed frame parts, identifying 353, 354  
  building 355-357  
  components, identifying to create 354, 355  
  computer vision, adding to 403  
  constructing 353  
  programming, with ROS 385-388  
  Raspberry Pi 3B+, using for 353  
  software configuration 360, 361  
  software, setting up 360, 361  
  text message functionality,  
    adding to 414, 415  
  video, streaming from 380-385  
  wiring up 357-360  
**alarm circuit**  
  configuring 156, 157  
**alarm code**  
  creating, for Pico H 368-370  
**alarm module application**  
  running 169-171  
**alarm module client code**  
  alarm module application, running 169-171  
  buzzer, activating through code 161-163  
  main code, creating 164-169  
  writing 160, 161  
**alarm system**  
  building 144-147  
**Alpha Vantage**  
  reference link 45  
**Alpha Vantage API** 45  
**Analog-to-Digital Converter**  
  (ADC) 137, 182  
**Apache 2.0 license** 318  
**API key**  
  generating, for application 119, 120  
**Application Programming**  
  **Interface (API)** 35, 37  
  versus web services 37



**Arduino IDE**

reference link 381

**A.R.E.S. robot application**

exploring 352, 353

**Automatic Dependent**

**Surveillance-Broadcast (ADS-B) 14**

**B**

**blob 400**

**C**

**cellular networks 298**

**Chirp Spread Spectrum (CSS) 242**

**circuit**

wiring up 327-329

**circuit construction 244**

assembling 246-248

jumper wires, adding to RFM95W 244, 246

**CircuitPython 248**

**code development**

CircuitPython, setting up 248-251

LoRa transmission code, creating 252-255

sensor libraries, setting up 248-251

**common cathode RGB LEDs**

versus common anode RGB LEDs 280

**communication**

testing, between Pi and Pico 374

**computer vision**

adding, to A.R.E.S. 403

exploring 390, 391

object detection, exploring 397, 398

OpenCV 391

YOLO and neural networks 397

**computer vision, to A.R.E.S.**

DogTracker class, creating 403-405

smart video streamer, building 405-408

**custom robot\_control node**

creating 339-343

**custom ROS node**

creating 338

custom robot\_control node,  
creating 339-343

ROS TurtleSim robot, controlling  
with IoT joystick 343-345

**D****Dashboard class**

adding 122-128

**deep learning**

versus machine learning (ML) 397

**DELETE method 37**

**development environment**

setting up 140-142, 157-160

**Digital Serial Interface (DSI) 111**

**Digital-to-Analog Converter (DAC) 182**

**direct current (DC) motor 77**

**disaster recovery (DR) 388**

**displays compatible**

investigating, with Raspberry Pi 110, 111

**distance sensors**

versus ToF sensor 380

**DogTracker class**

creating 403-405

**Dual Inline Package (DIP) 357**

**dynamic widgets 126**

**E**

**engineering-grade resin 173**

**external alarm buzzer stand**

building 230

parts, identifying 230, 231

stand, building 231, 232

## F

**Fast Healthcare Interoperability**

**Resources (FHIR) 35**

**Fused Deposition Modeling**

**(FDM) 85, 173, 206, 274**

## G

**General Purpose (GP) 9**

**General-Purpose Input/Output (GPIO) 3**

on Raspberry Pi 134

**GET method 37**

**GNU Compiler Collection (GCC) 19**

**GNU Debugger (GDB) 19**

**GO-NO-GO application for**

**decision making 59, 60**

building 70

GO-NO-GO client code, writing 66-69

green checkmark, creating on Sense HAT 60

NO-GO animation, creating

on Sense HAT 62-66

**GPIO expander 194**

**GPIO port 75**

**GPIO Zero 81, 82**

using, to control LEDs 84

using, to control servo 82, 83

**green checkmark**

creating, on Sense HAT 60-62

## H

**Hardware Attached on Top (HAT) 3**

**HC-SR501 PIR sensor**

controls 175

**Health Level Seven International (HL7) 35**

**HTTP methods**

DELETE method 37

GET method 37

POST method 37

PUT method 37

**Humble Hawksbill 308**

installing 309

## I

**image classification 391**

**image recognition 391**

**indicators 139**

**Inertial Measurement Unit (IMU) 183**

**infrared (IR) sensor 379**

**in-plane switching (IPS) 182**

**installation script**

running 365-368

**Integrated Development**

**Environment (IDE) 11, 157**

**Inter-Integrated Circuit**

**(I2C) 9, 136 182, 360**

features 136

**Internet of Things (IoT) 3, 33, 265**

**Internet Protocol (IP) 38**

**interrupt request (IRQ) 169**

**IoT alarm dashboards**

exploring 211

security alarm dashboard 212

using, for industrial processes 211, 212

**IoT alarm module case**

building 171-176

parts, identifying of custom case 172-174

**IoT alarm system**

running 232-234

**IoT-based robotics project 17, 18**

**IoT button 179**

- creating, with M5Stack ATOM Matrix 181
- examples 180
- technologies, exploring 180
- testing 189, 190
- used, for turning M5Stack  
ATOM Matrix 186-189
- utilizing 179

**IoT communication protocols**

- exploring 296-298

**IoT information display**

- application, running 128, 129
- creating 112, 113
- Dashboard class, adding 122-128
- development environment,  
setting up 113, 115
- MyApp class, adding 122-128
- software architecture 113
- TrafficMap class, creating 119
- WeatherData class, creating 116-118

**IoT joystick application 326, 327**

- code development 329
- Joystick class, creating 331-334
- MQTT messages, sending 334-338
- Raspberry Pi Pico WH, setting up 329-331

**IoT joystick case**

- constructing 345-348

**J****JavaScript Object Notation (JSON) 38****JR-style servo connector 75****K****keypad initialization process 226****Kivy 113****L****LEDs 78, 79**

- connecting, to Raspberry Pi 79, 80
- controlling, with GPIO Zero 84

**Line-Of-Sight (LOS) 241****Long Range (LoRa) 237**

- exploring 239
- practical uses 239, 240
- radio frequency spectrum,  
investigating 240, 242
- Spreading Factor (SF) 242, 243
- with Raspberry Pi Pico 243
- with Raspberry Pi Pico W 243
- with Wi-Fi microcontroller 298

**Long Range Wide Area Network  
(LoRaWAN) 182, 265, 298****Long Term Evolution (LTE) 239****LoRa nodes 239****LoRa receiver 265**

- application testing 262, 263
- building 258
- CircuitPython library, installing  
for MQTT 267, 269
- CloudAMQP instance, creating  
for application 269
- code, creating to receive LoRa  
messages 260, 261
- connecting, to internet 266, 267
- LED, wiring to Raspberry Pi Pico W 259
- MQTT functionality, adding to  
LoRa receiver 269-274

**LoRa sensory transmitter**

- building 243, 244
- circuit construction 244
- code development 248
- components installation 255

custom LoRa case part, building 257, 258  
 custom LoRa case part, identifying 256, 257

### **LoRa transmission code**

creating 252-255

## **M**

### **M5Stack ATOM Matrix**

firmware, flashing 183, 184  
 program, configuring 184-186  
 turning, into IoT button 186-189  
 used, for creating IoT button 181

### **M5Stack devices**

ATOM Matrix 183  
 ENV III HAT 182  
 exploring 181  
 M5Stack Basic 182  
 M5StackC PLUS 182  
 M5Stack Unit LoRaWAN915 182

### **M5Stack Unit LoRaWAN915 182**

### **machine learning (ML) 66**

versus deep learning 397

### **Machine-to-Machine (M2M) 296**

### **MapQuest Developer**

reference link 119

### **Message Queuing Telemetry**

#### **Transport (MQTT) 17, 301**

CircuitPython library, installing 267, 269  
 devices, categorizing in two roles 151  
 functionality, adding to LoRa  
   receiver 269-273  
 investigating 150  
 publish-subscribe model in 151, 152  
 Quality-of-Service (QoS) 152  
 Raspberry Pi Pico W, using 154

### **MicroPython 248**

using 157

### **Mobile Industry Processor Interface (MIPI) 112**

### **motors**

controlling 371-374  
 testing 371-374

### **MQTT Explorer app 204**

### **MQTT fundamentals**

exploring, with MQTTHQ  
 web client 152-154

### **MQTTHQ web client**

MQTT fundamentals, exploring  
 with 152-154

### **MQTT message**

robot, controlling with 323, 324

### **MyApp class**

adding 122-128

## **N**

### **neural networks 397**

### **NO-GO animation**

creating, on Sense HAT 62-66

### **non-maximum suppression (NMS) 401**

## **O**

### **object detection 390, 391**

exploring 397, 398  
 YOLO, using to identify dog  
   in picture 398-402

### **object recognition 391**

### **OpenCV 391**

used, for streaming video 395, 396  
 used, for viewing image 391-394

### **Open Source Software (OSS) 302**

### **OpenWeatherMap 73, 94**

**operating systems**

- features 14, 15

- investigating, for Raspberry Pi 14, 15

**Organic Light-Emitting Diode (OLED) 11, 109****over-the-air (OTA) 183****P****package.xml file**

- updating 321

**passive infrared (PIR) 181**

- exploring 142-144

**Pi and Pico**

- communication, testing between 374

**Pibrella HAT 12, 13****Pico H**

- alarm code, creating for 368-370

- script, creating 374-376

**Pi Zero 5****Polyethylene Terephthalate**

- Glycol (PETG) 85, 173

**Polylactic Acid (PLA) 85, 173, 275****POST method 37****Programmable Real-Time Units (PRUs) 11****publish-subscribe model 150**

- in MQTT 151, 152

**Pulse-Code Modulation (PCM) 138**

- features 138

**pulse-width modulation**

- (PWM) 78, 162, 286, 369

**PUT method 37****Python virtual environment**

- activating 218

- using, for development 81

**Q****Quality-of-Service (QoS)**

- in MQTT 152

**R****radio frequency spectrum 237****Raspberry Pi**

- displays compatible, investigating with 110, 111

- GPIO 134

- LEDs, connecting to 79, 80

- operating systems. investigating for 14, 15

- Raspberry Pi Imager, using 306

- SG90 servo motor, connecting to 75, 76

- UART test code, running from 377-379

- used, for connecting web services 38, 39

**Raspberry Pi 3B+**

- Ubuntu, installing onto 361-365

- using, for A.R.E.S. 353

**Raspberry Pi 4B**

- in custom case 4

**Raspberry Pi 5 alarm dashboard**

- code, writing 216

- creating 213

- IoT alarm module code, modifying 213-215

- IoT alarm module, geolocation 216

**Raspberry Pi 5 alarm dashboard code**

- buzzer code, testing 219-221

- buzzer code, writing 219-221

- buzzer, wiring up 219

- development environment,

  - setting up 216-218

- Kivy dashboard, creating 221-229

- writing 216

**Raspberry Pi 7-inch Touch Display 111**

**Raspberry Pi and Sense HAT**

development environment, setting up 21-23

**Raspberry Pi development 20**

tools 19

working with 18, 19

**Raspberry Pi, for IoT**

T.A.R.A.S 17, 18

using 16

web services, utilizing advantages 16, 17

**Raspberry Pi GPIO**

pin communication protocols 135

pinout diagram 134, 135

**Raspberry Pi GPIO, pin**

communication protocols

I2C 136

PCM 138

SPI 136, 137

UART 137

**Raspberry Pi HATs 12**

Pibrella HAT 12, 13

Raspberry Pi Sense HAT 13, 14

**Raspberry Pi Imager**

using, on Raspberry Pi 306

**Raspberry Pi models**

alternatives, exploring 10-12

comparison, with various models 5-10

exploring 5-10

**Raspberry Pi OS**

reference link 361

**Raspberry Pi Pico**

using, in Long Range (LoRa) 243

**Raspberry Pi Pico W**

using, in Long Range (LoRa) 243

using, with MQTT 154

**Raspberry Pi Pico W IoT button 190, 191**

alarm module code, modifying 192

building 193

buzzer message, activating 196-198

CloudAMQP instance, setting up 191, 192

components 193, 194

components, installing 206-208

packages, setting up 195, 196

primary functionality, coding 199-203

running 204, 206

wiring up 194, 195

**Raspberry Pi Pico W, with MQTT**

alarm circuit, configuring 156, 157

alarm module client code, writing 160, 161

development environment,

setting up 157-160

RP2040 chip 155

**Raspberry Pi Sense HAT 13, 14****Raspbian 361****Real-Time Clock (RTC) 182****Real-Time Streaming Protocol (RTSP) 352****Representational State Transfer (REST) 34****RESTful web services**

investigating 36-38

**RGB LED indicator, weather indicator**

circuit, testing 282, 283

configuring 279

connecting, to Raspberry Pi Pico WH 281

jumper wires, adding 280

**Robot Operating System (ROS) 15, 301, 302**

adding, to Ubuntu installation 308-311

distributions, aligning with

Ubuntu LTS versions 305

installation, testing 311-313

node communication 303, 304

organization, investigating 304, 305

project structure, investigating 304, 305

reference link 302

used, for programming A.R.E.S 385-388

workspace and package, creating 316-318

**ROS TurtleSim robot**

controlling, with IoT joystick 343-345

**RP2040 chip 155****S****screen types**

3.5-inch LCD screens 112

7-segment display 111

8 x 8 dot-matrix display 111

16 x 2 LCD screens 111

Raspberry Pi 7-inch Touch Display 111

small OLED screens 111

**Sense HAT**

green checkmark, creating 60-62

in custom case 4

NO-GO animation, creating 62

used, for connecting web services 38, 39

**Sense HAT development 20**

animations, creating 27-30

scrolling environmental data

display, creating 30-32

sensor data, reading 23-27

**sensors 139****Serial Clock Line (SLC) 136****Serial Data In (SDI) 136****Serial Data Line (SDA) 136****Serial Data Out (SDO) 136****Serial Peripheral Interface  
(SPI) 9, 136, 183, 272**

features and advantages 137

**servo motor jittering 83****servo motors 75, 77, 78****servo motor, weather indicator**

configuring 284

testing 286-288

wiring up 284, 285

**SG90 servo motor 75**

connecting, to Raspberry Pi 75, 76

**Shapeways**

reference link 85

**Sigfox 296, 298****Silhouette Cameo 88****Simple Object Access Protocol (SOAP) 34****simulated robot**

code, compiling 322, 323

code, executing 322, 323

controlling 313

controlling, with MQTT message 323, 324

generated Python code, modifying 318-321

package.xml, updating 321, 322

ROS workspace and package,  
creating 316-318

running 313

**Single-Board Computer (SBC) 154****smart video streamer**

building 405-408

modifying 416, 418

**SOAP web services**

using 35, 36

**Spreading Factor (SF) 242****standard monitor**

using 113

**static widgets 126****stock ticker application**

API key 45

creating 44, 45

enhancing 48-51

web services client code, writing 46-48

**T****text alert**

sending out 408

text message functionality, adding  
to A.R.E.S. 414, 415

Twilio account, setting up 408-414

**text message functionality, to A.R.E.S.**  
 smart video streamer, modifying 416, 418  
 TwilioMessage class, creating 415, 416

**This Amazing Raspberry-Pi Automated Security agent (T.A.R.A.S)** 17, 18

**Time of Flight (TOF)** 140, 353

**ToF sensor**

testing 379, 380  
 versus distance sensors 380

**TrafficMap class**

API key, generating for application 119, 120  
 coding 120-122  
 creating 119

**TurtleSim** 301, 313

launching 313  
 testing 314-316

**TurtleSim controller ROS application**

reviewing 302

**Twilio account**

setting up 408-414

**TwilioMessage class**

creating 415, 416

## U

**UART test code**

running, from Raspberry Pi 377-379

**Ubuntu**

installing, on Raspberry Pi 4 306-308  
 installing, onto Raspberry Pi 3B+ 361-365  
 version, selecting 308

**Ubuntu installation**

ROS, adding to 308-311

**Ubuntu Long-Term Support (LTS) version**

ROS distributions, aligning with 305

**Ultra high frequency (UHF)** 240

**Universal Asynchronous Receiver-Transmitter (UART)** 9, 137, 352

features 137

## V

**video**

streaming, from A.R.E.S. 380-385

## W

**WeatherData class**

creating 116-118

**weather display applications**

API key 52, 53  
 developing 51, 52  
 GO-NO-GO application for  
 decision making 59, 60  
 weather information on Sense HAT 55-58  
 weather information ticker, creating 53

**weather indicator**

creating 274  
 faceplate, building 276, 278  
 programming 288-296  
 RGB LED indicator, configuring 279  
 servo motor, configuring 284  
 split stand, building 274-276

**weather indicator application**

main methods, adding 105-107  
 needle, calibrating 95  
 software architecture 94, 95  
 updateDashboard() function,  
 adding 105-107  
 WeatherDashboard class, creating 102-105  
 WeatherData class, creating 96-102

**weather indicator stand**

8mm LED holder 87  
 alignment tool 88



- arrow 88
- assembling 86
- back plate 87
- base 87
- building 85
- face stick 88
- front plate 87
- LED, installing 92
- M5 20mm bolt 88
- M5 nut 88
- parts 86, 87
- pedestal, assembling 90
- pipe base 87
- plate, assembling 88, 89
- plate bracket 87
- plate, connecting to pedestal 92
- plate hook 87
- PVC pipe 88
- Raspberry Pi, installing 91
- servo, installing 91
- weather information on Sense HAT 55-58**
- weather information ticker**
  - creating 53
  - web service, testing 53, 54

## **web services**

- approaches 35
- code, writing 41-44
- connecting, with Raspberry Pi 38, 39
- connecting, with Sense HAT 38, 39
- development environment, setting up 39-41
- exploring 34, 35
- RESTful web services, investigating 36-38
- SOAP web services, using 35, 36
- utilizing, for IoT applications 16, 17
- versus API 37

## **Y**

### **YOLO 397**

- using, to identify dog in picture 398-402

### **yolo4.weights file 398**